IBM VisualAge TeamConnection Enterprise Server

User’s Guide

Version 3.0

SC34-4499-04

IBM VisualAge TeamConnection Enterprise Server

User's Guide

Version 3.0

SC34-4499-04

July, 1999

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

This edition applies to Fixpack 3.0.3 of the licensed program IBM TeamConnection Enterprise Server and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications by phone or fax. The IBM Software Manufacturing Company takes publication orders between 8:30 a.m. and 7:00 p.m. eastern standard time (EST). The phone number is (800) 879-2755. The fax number is (800) 284-4721.

You can also order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Attn: Information Development
Department T99B/Building 062
P.O. Box 12195
Research Triangle Park, NC, USA 27709-2195

You can fax comments to (919) 254-0206.

If you have comments about the product, address them to:

IBM Corporation
Attn: Department TH0/Building 062
P.O. Box 12195
Research Triangle Park, NC, USA 27709-2195

You can fax comments to (919) 254-4914.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Notices

Trademarks

About this book

This book is part of the documentation library supporting the IBM TeamConnection licensed programs. It is a guide for client users.

For additional information when performing TeamConnection tasks, refer to the *Commands Reference* when entering commands or online help when using the graphical user interface (GUI).

This book is available in PDF format. Because production time for printed manuals is longer than production time for PDF files, the PDF files may contain more up-to-date information. The PDF files are located in directory path nls\doc\enu (Intel) or softpubs/en_US (UNIX). To view these files, you need a PDF reader such as Acrobat.

How this book is organized

"Part 1. Introducing TeamConnection" on page 1, gives all users an overview of the concepts of TeamConnection and introduces the terminology that is used throughout this book.

"Part 2. Developing a product using TeamConnection" on page 13, describes the different interfaces and basic TeamConnection tasks. It uses scenarios to explain how to do many tasks.

This part is for everyone using TeamConnection to do daily work. The information is meant for both the person who uses the command line interface and the person who uses the GUI, as instructions for both are provided.

"Part 4. Using TeamConnection to build applications" on page 115, tells how to use the TeamConnection build function. For information in installing and administering the build function, refer to the *Administrator's Guide.*

"Part 5. Using TeamConnection to package products" on page 177, tells how TeamConnection helps you automate the packaging and distribution of your application.

Information on customer service, a bibliography, and a glossary are included at the back of this book.

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, HP-UX, and Solaris 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

IBM VisualAge TeamConnection Enterprise Server for OS/2

NT IBM VisualAge TeamConnection Enterprise Server for Windows/NT

95 IBM VisualAge TeamConnection Enterprise Server for Windows 95

IBM VisualAge TeamConnection Enterprise Server for AIX

IBM VisualAge TeamConnection Enterprise Server for HP-UX

IBM VisualAge TeamConnection Enterprise Server for Solaris

Tell us what you think

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other IBM VisualAge TeamConnection Enterprise Server documentation fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

Part 1. Introducing TeamConnection

This section presents an overview of the TeamConnection product. The information is intended for everyone who will be using TeamConnection.

Additional conceptual information is provided in Parts 3, 4, 5, and 6.

Chapter 1. An introduction to TeamConnection

TeamConnection is a team programming environment that helps you manage and control your development projects, increase team productivity, and increase software quality. You can use TeamConnection to communicate with and share data among team members to keep up with the many tasks in the development life cycle, from planning through maintenance.

TeamConnection helps you streamline the following tasks:

- *Configuration management*: the process of identifying, organizing, managing, and controlling software modules as they change over time. This includes controlling access to your software modules and providing notification to team members as software modules change.
- *Release management*: the logical organization of objects that are related to an application. The release provides a logical view of objects that must be built, tested, and distributed together. Releases are versioned, built, and packaged.
- *Version control*: the tracking of relationships among the versions of the various parts that make up an application. Version control enables you to build your product using stable levels of code, even if the code is constantly changing. It provides control over which changes are available to everyone and, optionally, allows more than one developer at a time to update a part.
- *Change control*: the controlling of changes to parts that are stored in TeamConnection. TeamConnection keeps track of any part changes you make and the reasons you make them. Your development team can build releases with accuracy and efficiency, even as the parts evolve. The product ensures that the change process is followed and that the changes are authorized. After changes are made, it allows you to integrate the changes and build the application. TeamConnection tracks all changes to the parts across multiple products and environments.

  The *change control process* is configurable. Your team can decide how strict the change control should be, from loose to very tight. You can also adjust the level of control as you move through a development cycle.
- *Build support*: the function that enables you to define the structure of your application and then to create it within TeamConnection from your input parts. Independent steps in a build can run in parallel on different servers, thus reducing your build time. You can build applications for platforms in addition to the one TeamConnection runs on—currently, you can use TeamConnection to build applications on AIX, HP-UX, OS/2, Windows NT, Windows 95, Solaris, MVS, and MVS OpenEdition.
- *Packaging support*: the preparation of your application for electronic distribution to other users.

This chapter defines the basic terms and concepts you need to make the most of TeamConnection. Read this chapter first; then decide which information you need next:

Topic and description	**Page**
Developing products using TeamConnection: • Getting familiar with the interfaces • The basics of using TeamConnection • More about *defects* and *features* • Following TeamConnection processes	14

Topic and description	Page
Using TeamConnection to build applications: • Build concepts • Installing build agents and processors • Working with build scripts and builders • Working with *parsers* • Building an application	116
Packaging applications: • Using the packaging function • Using the Gather utility • Using the NVBridge utility	“Chapter 15. Using TeamConnection to package a product” on page 179

TeamConnection definitions

The following definitions are in logical order rather than alphabetical.

TeamConnection’s client/server architecture

Figure 1 is an example of a network of TeamConnection clients and servers.

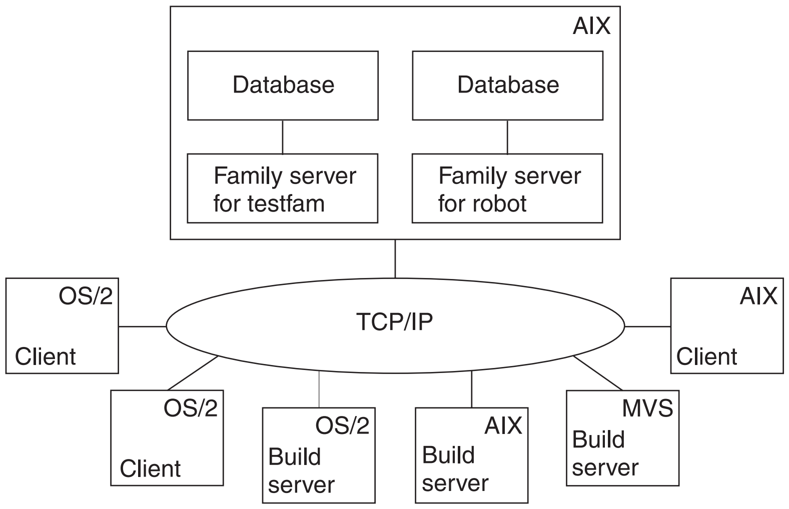


Figure 1. A sample TeamConnection client/server network

TeamConnection *family servers* control all data within the TeamConnection environment. Data stored in a family server’s database includes:

- Text objects, such as source code and product documentation
- Binary objects, such as compiled code
- Modeled objects that are stored in the information model by tools such as VisualAge Generator
- Other TeamConnection objects that are metadata about the other objects

A TeamConnection *client* gives team members access to the development information and parts stored on the database server.

TeamConnection database

TeamConnection is built on IBM's DB2 Universal Database. Please refer to the DB2 documentation referenced in this document's "Bibliography" on page 267 for detailed information on DB2 database configuration, administration, and utilities.

Interfaces

TeamConnection provides the following interfaces that you can use to access data:

- A web-based client
- A command line interface that lets you type TeamConnection commands from a prompt or from within TeamConnection

You can use any interface to do your TeamConnection work, or you can switch among them. This book usually gives instructions for the standard GUI and for the command line interface.

For more information, see "Chapter 2. Getting familiar with the TeamConnection client interfaces" on page 15.

Families

A *family* represents a complete and self-contained collection of TeamConnection users and development data. Data within a family is completely isolated from data in all other families. One family cannot share data with another.

Refer to the *Administrator's Guide* for more information about families.

Users and host lists

Users are given access to the TeamConnection development data in a specific family through their *user IDs*. Each family has at least one *superuser*, who has privileged access to the family. The superuser gives other users the *authority* to perform some set of *actions* on particular data. Depending on the authority granted to a user, that user might in turn be able to grant some equal or lesser level of authority to other users. However, the ability to grant authority for some actions is reserved to the superuser. There are no actions which the superuser cannot perform.

For host-based authentication, each user ID is associated with a *host list*, which is a list of client machine addresses from which the user can access TeamConnection when using that ID.

A single user can access TeamConnection from multiple systems or logins. Likewise, a single system login can act on behalf of multiple users. The set of authorized logins for a TeamConnection user ID makes up the user's host list.

It is also possible to authenticate users through the use of passwords, either in place of host lists or as an alternative form of authentication.

Refer to the *Administrator's Guide* for more information.

Parts

TeamConnection *parts* are objects that users and tools store in TeamConnection. They include text objects, binary objects, and modeled objects. These parts can be stored by the user or the tool, or they can be generated from other parts, such as when a linker generates an executable file. Parts can also be groupings of other TeamConnection objects for building and distribution, or simply for convenient reference. Common part actions include the following:

Create
To store a part from your workstation on the server; from that time on, TeamConnection keeps track of all changes made to the part. Or, to create a part to use as a place holder to store the output of a build.

Check out
To get a copy of a part so that you can make changes to it.

Check in
To put the changed part back into TeamConnection.

Extract
To get a copy of the part *without* making changes to the *current version* in TeamConnection.

Edit To change a part from within TeamConnection using a specified editor.

Build To construct an output part from parts that you have defined to TeamConnection as input to the output part.

These are simplified definitions of part actions; there is more about the actions you can perform against parts in “Chapter 3. The basics of using TeamConnection” on page 25.

The current version of each part is stored in the TeamConnection database along with previous versions of each part. You can return to previous versions if you need to.

Components

Within each family, development data is organized into groups called *components*. The component hierarchy of each family includes a single top component, called *root*, and *descendants* of that root. Each *child component* has at least one parent component. A child can have multiple parents.

The following figure depicts a component hierarchy.

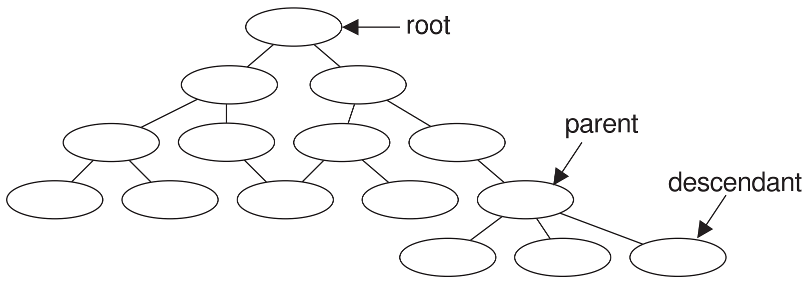


Figure 2. Sample of a component hierarchy

TeamConnection uses components to organize development data, control access to the data, and notify users when certain actions occur. Descendant components inherit access and notification information from ancestor components. Information about the components is stored in the database, including:

- The component's position in its family hierarchy.
- The user who owns the component. The component *owner* is responsible for managing data related to it, including defects or features.
- The users who have access to the component and the level of access each user has. This information makes up the component's *access list*.
- The users who are to be notified about changes to the component. This set of users is called the *notification list*.
- The *process* by which the component handles defects and features.

Releases

An application is likely to contain parts from more than one component. Because you probably want to use some of the same parts in more than one application, or in more than one version of an application, TeamConnection also groups parts into *releases*. A release is a logical organization of all parts that are related to an application; that is, all parts that must be built, tested, and distributed together. Each time a release is changed, a new version of the release is created. Each version of the release points to the correct version of each part in the release.

Each part in TeamConnection is managed by at least one component and contained in at least one release. One release can contain parts from many components; a component can span several releases. Figure 3 shows the relationships between parts, the releases that contain them, and the components that manage them.

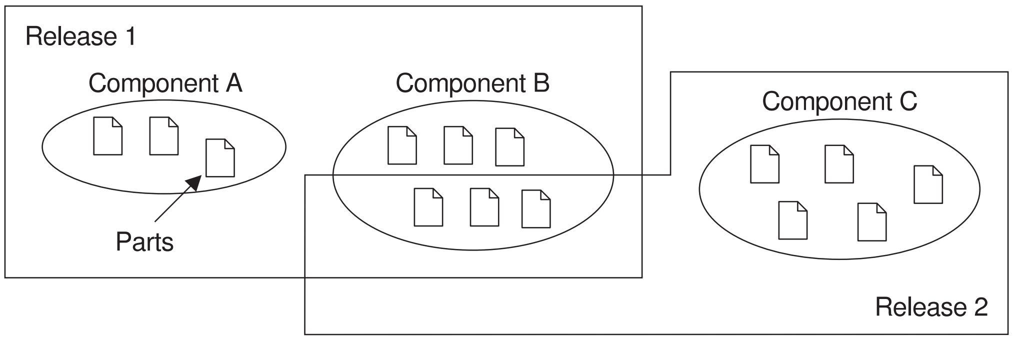


Figure 3. Parts, releases, and components

Each time a new development cycle begins, you can define a separate release. Each subsequent release of an application can share many of the same parts as its predecessor. Thus maintenance of an older release can progress at the same time as development of a newer one. Each release follows a process by which defects and features are handled.

Workareas

A release contains the latest ″official″ version of each of its parts. As users check parts out of the releases, update them, and then check them back in, TeamConnection keeps track of all of these changes, even when more than one user updates the same part at the same time. To make this possible, TeamConnection uses something called a *workarea*.

A workarea is a logical temporary work space that enables you to isolate your work on the parts in a release from the official versions of the parts. You can check parts out to a workarea, update them, and build them without affecting the official version of the parts in the release. After you are certain that your changes work, you *integrate* the workarea with the release (or *commit* the driver that the workarea is a member of, if you are using the driver subprocess). The integration makes the parts from your workarea the new official parts in the release.

You can do the following with workareas:
- Check out parts from a release
- Update any or all of the checked-out parts
- Get the latest copies of the parts in the release, including any changes integrated by other users
- Get the latest copies of the parts in another workarea
- *Freeze* the workarea, making a snapshot of the parts as they exist at a particular instant in case you need to return to it later
- Build the parts in the workarea
- Move all parts back into the release by integrating the workarea

For more information, see “Using workareas” on page 27.

Drivers

A driver is a collector for workareas. You create drivers associated with specific releases so that you can exercise greater control over which workareas are integrated into the release and commit the changes from multiple workareas simultaneously.

When a workarea is added to a driver, it is called a *driver member.* A single workarea can be a member of more than one driver. By making a workarea part of a driver, you associate the parts changed in relation to that workarea with the specified driver. These parts must be members of the release associated with the driver.

Drivers enable you to place the following controls over workarea integrations:
- Define and monitor prerequisite and corequisite workareas to ensure that mutually dependent changes are integrated in proper order.
- Monitor and resolve conflicting changes to the same part (if you use concurrent development).
- Restrict access to driver members so that they can be changed only by users with proper authority.

Defects and features

A defect is a record of a problem to be fixed. A feature is a record of a request for a functional addition or enhancement. Both must be associated with a workarea and must follow the processes defined for the component and release that are associated with the workarea. TeamConnection tracks both objects through their life cycles as developers change and commit parts.

You can use defects and features to record problems and design changes for things other than the products you are developing under TeamConnection control. For example, you can use defects to record information about personnel problems, hardware problems, or process problems. You can use features to record proposals for process improvements and hardware design changes.

For more information, see “Working with defects and features” on page 36.

Processes

An application changes over time as developers add features or correct defects. TeamConnection controls these changes according to the *processes* you choose for your application’s components and releases. A process enforces a specific level of control to part changes and ensures that actions occur in a specified order.

Two separate types of processes are defined: component processes, which can be different for each component within a family, and release processes, which apply to all activities associated with a given release. Component or release processes are built from a number of lower-level processes, or *subprocesses*, that are included with the TeamConnection product.

A defect or feature written against a component moves through successive *states* during its life cycle. The TeamConnection actions that you can perform against it depend on its current state. The component processes define these actions. You can require users to do some, all, or none of the following for tracking defects and features:

dsrFeature
: Design, size, and review changes to be made for features

verifyFeature
: Verify that the features have been implemented correctly

dsrDefect
: Design, size, and review fixes to be made for defects

verifyDefect
: Verify that the fixes work

At the release level you can require some, all, or none of the following subprocesses:

track
: This subprocess is TeamConnection’s way of relating all part changes to a specific defect or feature and a specific release. Each workarea gathers all the parts modified for the specified defect or feature in one release and records the status of the defect or feature. The workarea moves through successive states during its life cycle. The TeamConnection actions that you can perform against a work area depend on its current state.

  You must use the *track subprocess* if you want to use any of the other release subprocesses.

approval
: This subprocess ensures that a designated *approver* agrees with the decision to incorporate changes into a particular release and electronically signs a record. As soon as approval is given, the changes can be made.

fix
: This subprocess ensures that as users check in parts associated with a workarea, an action is taken to indicate that they have completed their

portion. When everyone is done, the owner of the *fix record* (usually the component owner) can change the fix record to complete. The parts are then ready for integration.

driver A *driver* is a collection of all the workareas that are to be integrated with each other and with the unchanged parts in the release at a particular time. The driver subprocess allows you to include these changes incrementally so that their impact can be evaluated and verified before additional changes are incorporated. Each workarea that is included in a driver is called a *driver member*.

test The test subprocess guarantees that testing occurs prior to verifying that the fix is correct within the release.

TeamConnection is shipped with several predefined processes. If these do not apply to your organization, you can configure your own processes by defining different combinations of subprocesses.

See “Chapter 4. The states of TeamConnection objects” on page 39 for an explanation of TeamConnection states.

Build

The TeamConnection build function automates the process of building individual parts or entire applications, both in the work group LAN environment and on an enterprise server. This function enables you to reliably and repeatedly build the same output from the same inputs. You can also build different outputs from the same inputs for different environments.

You start a build against an output part that has an associated *builder*. A builder is an object that describes how to translate input parts to get the desired output, such as a linker or compiler. An input part might have an associated parser, which determines the dependencies for the input parts in a build.

The build function does the following:

- Tracks build times of inputs and outputs so that it builds only those parts that are out of date themselves or that have out of date dependents. You can also force a build regardless of the build times.
- Enables you to spread the build over multiple machines running at the same time or into multiple processes running on a single machine, such as on MVS.

For more information, see “Part 4. Using TeamConnection to build applications” on page 115.

Packaging

Packaging is any of the steps necessary to distribute software and data onto the machines where they are to be used. TeamConnection includes two tools that you can use to automate the electronic distribution of TeamConnection-managed software and data:

Gather
An automated data mover for server or file transfer-based distribution.

Tivoli Software Distribution
: A bridge utility that automates the installation and distribution of software or data using Tivoli as the distribution vehicle.

For more information, see "Part 5. Using TeamConnection to package products" on page 177

Roles people play

Because TeamConnection is extremely flexible, no two projects are likely to use it in the same way, and the jobs that people perform likewise vary. Still, TeamConnection tasks can be grouped into the following general categories:

System administrator

Has *superuser* access to the family server and database administration access to the database management system. This administrator is responsible for the following:

- Installing and maintaining the TeamConnection server
- Maintaining and backing up the database used by TeamConnection

Note: On UNIX systems, the system administrator must also have root access to the TeamConnection server.

Family administrator

Has superuser access to the family server and database administration access to the database management system. This administrator is responsible for the following:

- Planning and configuring TeamConnection for one or more families
- Managing user access to one or more families
- Creating and updating configurable fields
- Configuring release and component processes for a family
- Creating and updating user exits
- Monitoring the user activity of a family
- Maintaining one or more families

Build administrator

This administrator is responsible for the following:

- Setting up and maintaining build servers
- Planning for builds
- Creating builders and parsers
- Starting and stopping build servers
- Defining *pools*
- Monitoring build performance
- Creating driver members
- Committing and completing drivers
- Extracting releases
- Packaging and distributing applications

End user

End users, such as project leaders, programmers, and technical writers, use one or more TeamConnection families to control and maintain application development data.

Part 2. Developing a product using TeamConnection

This section is for anyone who uses the TeamConnection client to do daily work. The information is meant for both the person who uses the command line interface and the person who uses the GUI; instructions for both are provided.

All the tasks in this part are done from a client machine.

Before reading this section, you should be familiar with the TeamConnection terminology and concepts presented in “Chapter 1. An introduction to TeamConnection” on page 3.

Chapter 2. Getting familiar with the TeamConnection client interfaces

TeamConnection provides two main interfaces that you can use to access data:

- A web-based client.
- A command line interface that lets you type TeamConnection commands from a prompt (provided you have installed this function).

You can use either of the interfaces to do all of your TeamConnection work, or you can switch back and forth between the two. You might find that some tasks are easier to do from the GUI, while others are easier to do from the command line.

The examples throughout “Part 2. Developing a product using TeamConnection” on page 13 give instructions for both GUI and command line interface usage.

This chapter helps you to begin using the TeamConnection client interfaces. It describes the following:

- Using the GUI
  - Starting the GUI
  - Getting around in the GUI
  - Using the command line interface

Before you can use TeamConnection, someone in your organization with superuser or admin authority, such as your family administrator, must create for you a unique user ID and a host list entry

Using the GUI

To use the TeamConnection GUI efficiently, select your Preference settings to suit your working environment, and then become familiar with the Home Page window and how you can save time by adding your most common tasks to **My Reports** in the Tasks menu.

Starting the GUI

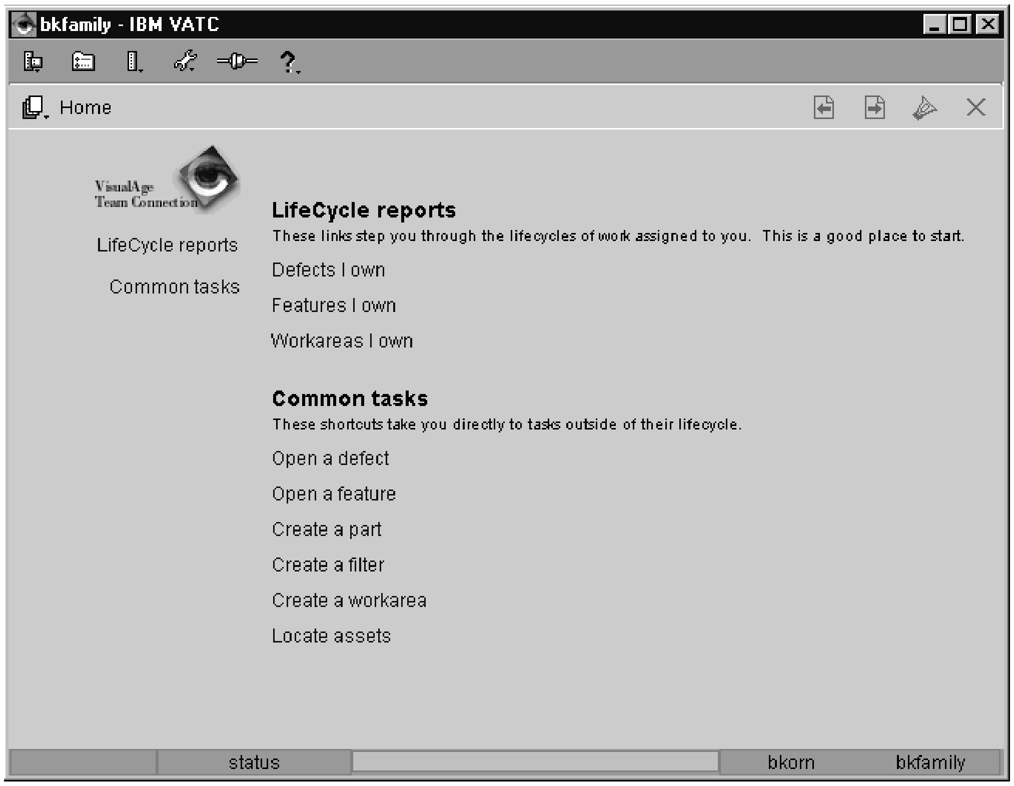
You can start the TeamConnection GUI by entering the following URL in your web browser:

```
http://servername/teamc/familyname/
```

where *servername* is the machine where the family is running, and *familyname* is the name of the family. For example, to access the testfam family on the server test.raleigh.company.com, you would enter the following URL:

```
http://test.raleigh.company.com/teamc/testfam/
```

The TeamConnection Home Page appears.


bkfamily - IBM VATC

Home

LifeCycle reports

These links step you through the lifecycles of work assigned to you. This is a good place to start.

Defects I own

Features I own

Workareas I own

Common tasks

These shortcuts take you directly to tasks outside of their lifecycle.

Open a defect

Open a feature

Create a part

Create a filter

Create a workarea

Locate assets

Figure 4. TeamConnection Home Page

Initially, a set of common tasks appears in the Home Page. These tasks can be changed by your TeamConnection administrator. Additional tasks can be found in **My Reports**, which you can select from the Task menu. As you become more familiar with TeamConnection and see what tasks you do most often, you can add your own tasks to **My Reports**.

Performing tasks with the GUI

There are several ways you can perform TeamConnection tasks with the GUI. You can:

- Select a task from the Home Page.

  This method provides a fast path within the GUI. When you select a task, TeamConnection performs the underlying query or command and then displays the requested information. The tasks that appear on the Home Page can be customized by your TeamConnection Administrator.
- Select an action from the Actions menu and then select the object you want to work with. For example, to view a specific defect, select **Defects** → **View** from the Actions menu, and then type the name of the defect in the window that appears.

  This method is useful when you know the exact names of the objects you want to work with.
- Create a filter that displays the objects you want to work with. To create a filter, select **Create a filter** from the Common Tasks section of the Home Page. A wizard appears that walks you through how to create a filter. (You can turn wizards off in your Preferences settings). After creating the filter, you can save it so that it will appear when you select **My Reports** from the Tasks menu.

This method is useful when you do not know the exact name of the object you want to work with, or you want to view a list of objects.

When viewing a list of objects, you can choose an action to perform on the object by either:

– Selecting an action button at the bottom of the screen
– Displaying a pop-up menu of actions by placing the mouse pointer over the object and pressing mouse button 2.

The action buttons at the bottom of the screen represent typical actions that users perform on the object. The pop-up menu contains *all* the actions that are valid for the object.

LifeCycle Navigator

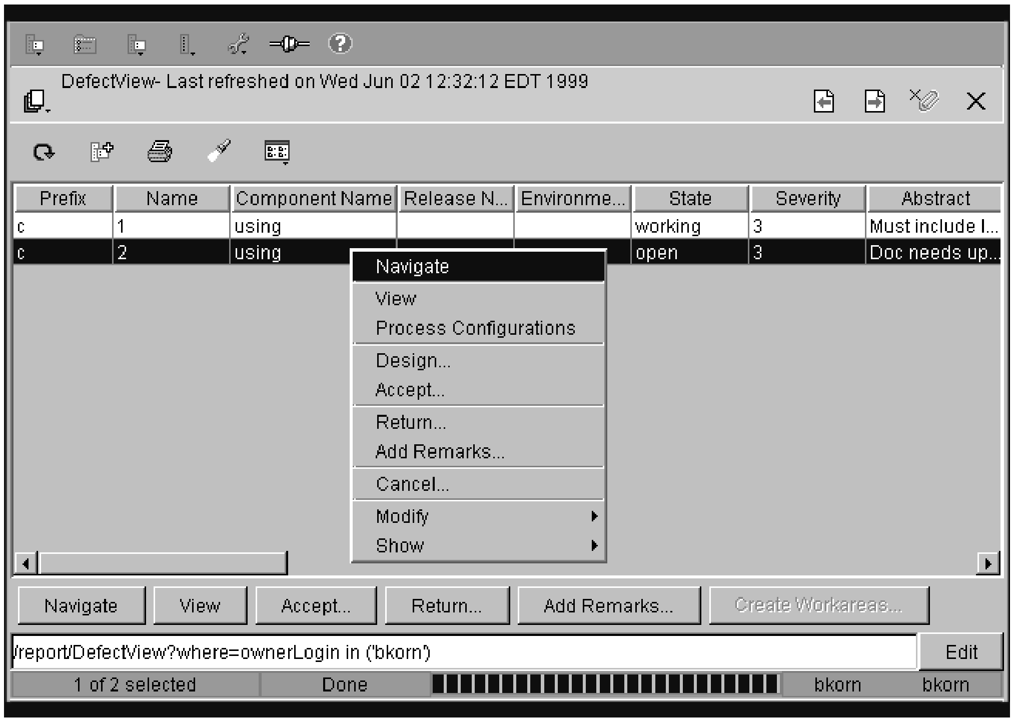
Many objects in TeamConnection have a life cycle or process that the object goes through from the time it is created until the time it is no longer used. For example, when a defect is opened, it starts off in the Open state, and can then go into the Working state, and then into the Verify state, and then into the Closed state. The actions that you can perform on a defect changes as its state changes. (See “Chapter 4. The states of TeamConnection objects” on page 39 for more information about object life cycles.)

As you become familiar with the way TeamConnection works and with the life cycles that objects go through, you can perform an action on an object by selecting an action from the **Actions** menu. However, an even more convenient way to work with objects that have life cycles is by using the **LifeCycle Navigator**.

The LifeCycle Navigator presents you with a graphical view of the life cycle of an object and enables you to perform actions on the object based on where the object is in its life cycle.

To start the LifeCycle Navigator for an object, select the object and then select either:

- The **Navigate** push button
- The **Navigate** choice from the object’s pop-up menu.


DefectView- Last refreshed on Wed Jun 02 12:32:12 EDT 1999

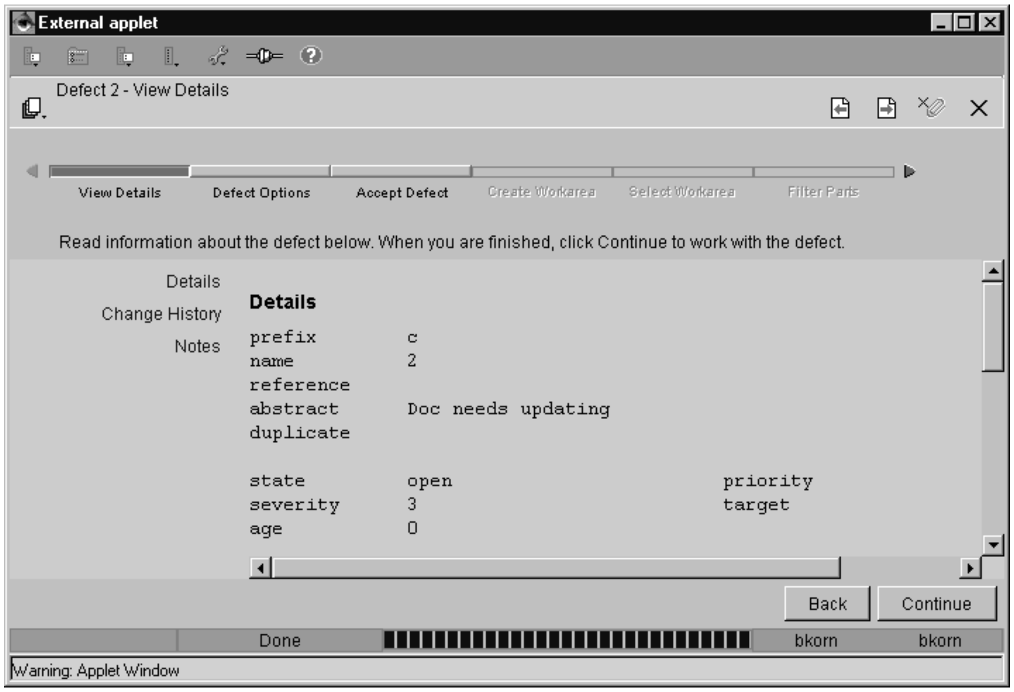
Prefix	Name	Component Name	Release N...	Environme...	State	Severity	Abstract
c	1	using			working	3	Must include l...
c	2	using			open	3	Doc needs up...

/report/DefectView?where=ownerLogin in ('bkorn')

1 of 2 selected Done bkorn bkorn

Figure 5. Starting the LifeCycle Navigator

The LifeCycle Navigator appears:


External applet

Defect 2 - View Details

Read information about the defect below. When you are finished, click Continue to work with the defect.

Details

```
prefix       c
name         2
reference
abstract     Doc needs updating
duplicate

state        open                     priority
severity     3                        target
age          0
```

Done bkorn bkorn

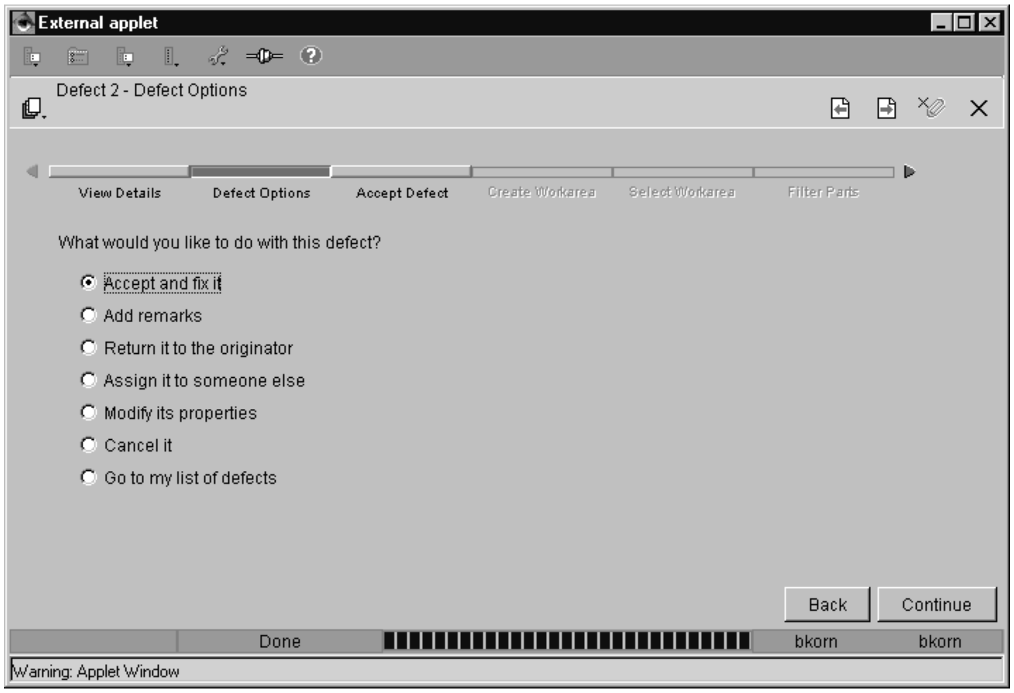
Warning: Applet Window

Figure 6. LifeCycle Navigator

The LifeCycle Navigator (Figure 6) consists of a bar with buttons that represent steps in an object’s life cycle. You can advance from one step in an object’s life cycle to another by selecting these steps. If the steps appear greyed out, they are not currently selectable; however, these greyed steps show future steps in the object's life cycle. There are also arrows on the LifeCycle Navigator that indicate the directions in which an object can move in its life cycle.

When you select a step in the LifeCycle Navigator, it turns green and a form associated with the step displays. For example, in Figure 6 on page 18, the **View Details** step is selected with the Details form displayed.

The life cycle steps that appear on the LifeCycle Navigator change according to the actions you select. For example, if you select **Defect Options** and then select **Accept and fix it** from the Defect Options form, the LifeCycle Navigator shows that the next step in the defect's life cycle as **Accept Defect** (Figure 7):

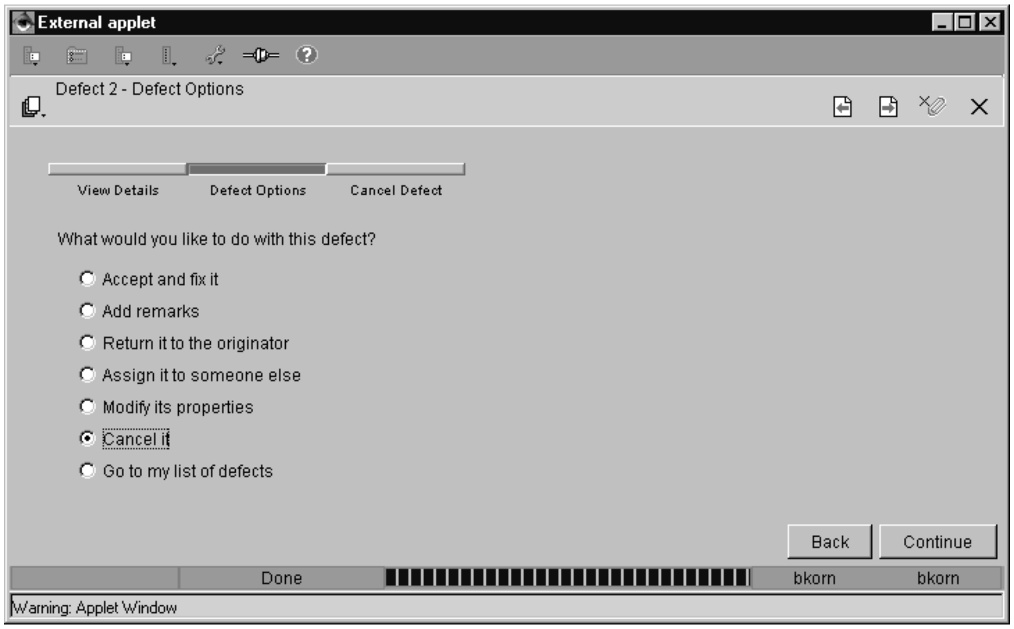


What would you like to do with this defect?

- [✓ Accept and fix it]
- Add remarks
- Return it to the originator
- Assign it to someone else
- Modify its properties
- Cancel it
- Go to my list of defects

Figure 7. LifeCycle Navigator - Accepting a Defect

If you select the **Cancel it** from the Defect Options form, the LifeCycle Navigator shows that the next step in the defect's life cycle as **Cancel Defect** (Figure 8 on page 20).


External applet

Defect 2 - Defect Options

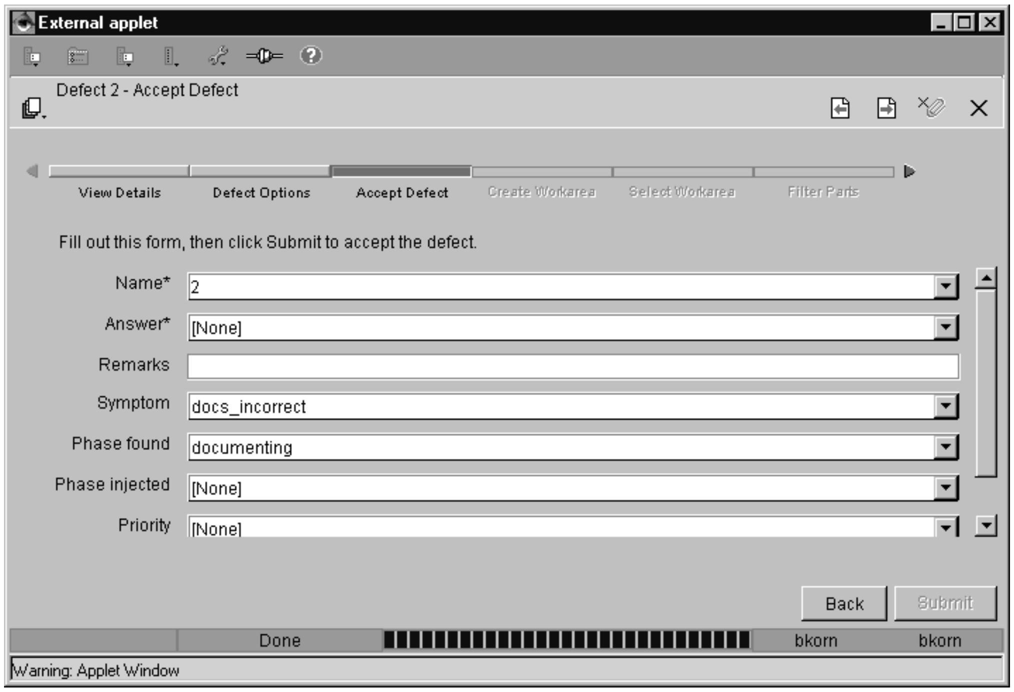
What would you like to do with this defect?

- Accept and fix it
- Add remarks
- Return it to the originator
- Assign it to someone else
- Modify its properties
- [✓ Cancel it]
- Go to my list of defects

Figure 8. LifeCycle Navigator - Cancelling a Defect

Notice that if you elect to cancel the defect, the arrows in the LifeCycle Navigator disappear, because once a defect is cancelled, the defect no longer has a path it can progress to.

To start the processing of a selected option, you select either the next step in the LifeCycle Navigator or the **Continue** push button. For example, to accept the defect, you would select **Accept and fix it** and then select the **Continue** push button. This activates the **Accept Defect** step (Figure 9 on page 21) and displays a form in which you enter the information for accepting a defect.



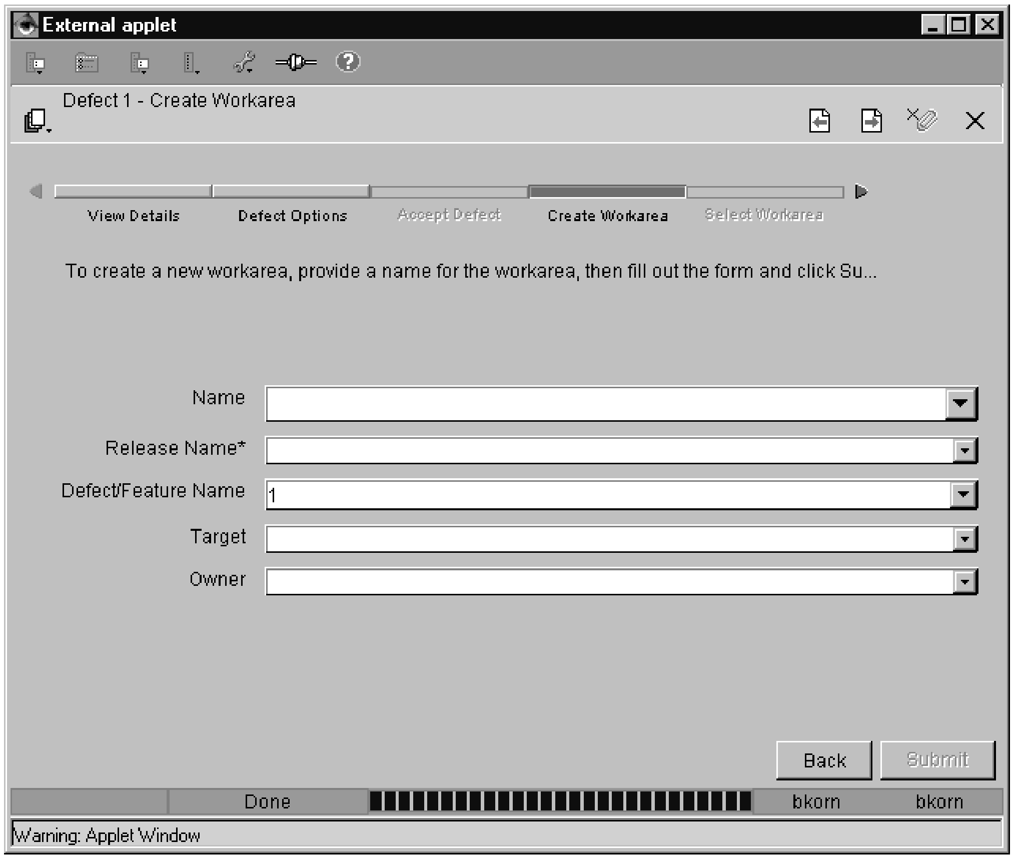
Fill out this form, then click Submit to accept the defect.

Name*	2
Answer*	[None]
Remarks	
Symptom	docs_incorrect
Phase found	documenting
Phase injected	[None]
Priority	[None]

Figure 9. LifeCycle Navigator - Accept Defect

When you have completed the Accept Defect form, the **Submit** button becomes active; selecting this button causes TeamConnection to move the defect into the Working state.

If you return to the LifeCycle Navigator after the defect is in the Working state, you will see that the **Create Workarea** step is no longer grey. This is because creating a workarea for a defect that is in the Working state is the next logical step in the object's life cycle. If you select the **Create Workarea** step (Figure 10 on page 22 the form for creating the workarea appears.


External applet

Defect 1 - Create Workarea

To create a new workarea, provide a name for the workarea, then fill out the form and click Su...

Name	
Release Name*	
Defect/Feature Name	1
Target	
Owner	

Done bkorn bkorn

Warning: Applet Window

Figure 10. LifeCycle Navigator - Create Workarea

After you have completed the form for creating a workarea, selecting the **Submit** push button causes TeamConnection to create the workarea.

As you can see from the above examples, the LifeCycle Navigator makes it easy to work with objects that have life cycles. It does this by laying out the path that an object can follow as well as the actions you can perform on the object. Other objects that you can currently work with using the LifeCycle Navigator include features and workareas.

Setting your preferences

If you select the Preferences icon, a window appears in which you can specify views, fonts, wizards and other settings for the GUI:

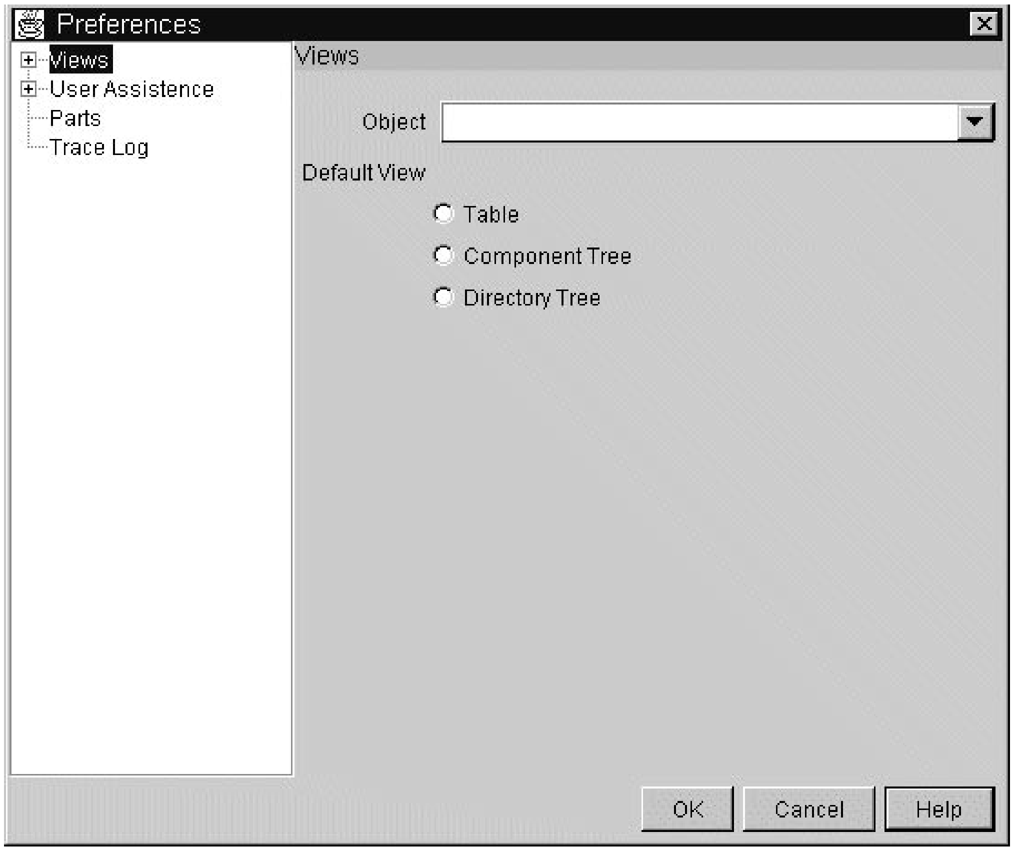


Figure 11. Preferences window

For information regarding the preferences you can select, refer to the on-line help.

Using the command line interface

To use the command line interface effectively, you must be familiar with the actions that you can perform using TeamConnection commands. A complete description of each command, including examples for each, is available in the *Commands Reference*

To view the syntax of a TeamConnection command online, type the following at a prompt:

```
teamc commandName
```

Where *commandName* is the name of the TeamConnection command.

You can type TeamConnection commands from a prompt within any directory (provided you have installed this function); the TeamConnection GUI does not need to be started.

Before you start to use the command line interface, you might want to set the most used environment variables, such as TC_FAMILY or TC_COMPONENT. You are not required to set these environment variables, but if you do not, you will need to specify them in the command when required.

You set environment variables differently for different platforms:

- AIX, HP-UX, and Solaris users set environment variables in the .profile (sh, ksh environment), .dtprofile (cde environment), or .cshrc (csh environment).
- OS/2 users set environment variables in the config.sys file or from a command line prompt.
- Windows 95 and Windows NT users set environment variables in the Windows Control Panel.
- Some environment variables are set in your config.sys file during installation.

You can override the value you set for an environment variable by using the corresponding flag in the command. For example, you have the TC_FAMILY environment variable set to robot, but you need a file from another family named octo, so you issue the following command:

```
teamc part -extract hello.c -family octo -release 9501
```

"Appendix A. Environment Variables" on page 201 provides a complete list of the TeamConnection environment variables.

Chapter 3. The basics of using TeamConnection

Users of TeamConnection perform a number of basic tasks, such as checking parts in and out of TeamConnection, as well as testing and verifying part changes. Before attempting these tasks, you should read this chapter for the basic concepts behind the tasks.

Before reading this chapter you should read "Chapter 1. An introduction to TeamConnection" on page 3 and be familiar with the different objects, such as components and releases. The other chapters in this part of the book define in more detail how to perform TeamConnection tasks.

Laying the groundwork

Before you use TeamConnection, your administrator will have already created your family's component structure. Components help you manage your parts and control access to the data. Your TeamConnection family also contains releases. A release identifies a version of all the parts that comprise an application at a given point in time. When you create a release, you specify the component that will manage it. One component manages a release, but many components can manage the individual parts associated with that release.

A single part can be associated with more than one release, but it is managed by one component. When you create a part, you specify the release that you want to associate with the part and the component that you want to manage it. At any time, you can link the created part to other releases so that the part can be shared, or you can change its managing component.

Before you start working with parts, you need to be familiar with your family's component structure. This will help you when trying to locate parts within TeamConnection and when writing defects and features. If you have the proper authority for the component, you can view the component structure by issuing the following command from the command prompt:

```
teamc report -raw -view bCompView -where "name='root'"
```

Authority to perform tasks

As a TeamConnection user, you are automatically given the authority to perform some basic tasks. You can:

- Open defects and features
- Add notes to existing defects and features
- Modify the information for your user ID
- Display information about any user ID
- *Search* for information within TeamConnection to create reports

You receive authority to perform additional actions when you become the owner of a TeamConnection object, such as a component or a part, or when authority is explicitly given to you by the component owners.

If you attempt an action that you do not have authority to do, TeamConnection tells you so. When this happens, you can ask the component owner, the family administrator, or a user with superuser authority to grant you the necessary authority.

Note: You can issue queries to generate reports of data from tables and views using the -view action flag. If you do not specify selection criteria, such as the fields and the search conditions you want to use, the report query selects all entries for the table or view indicated that the user has authority to access. This command does not show any objects in components that you are not authorized to access

"Appendix H. Authority and notification for TeamConnection actions" on page 235 lists the types of authority you need in order to perform various TeamConnection actions.

Finding objects within TeamConnection

All TeamConnection objects are stored on a server in a database. To find one or more of these objects within a family, do one of the following:

- Use the report command with the -view action flag from a command line.

  Command usage is explained in the *Commands Reference*
- Create an object filter by selecting **Create a Filter** from the Home Page.

The database used by TeamConnection is case-sensitive. Consequently, you need to refer to and search for objects in the correct case. For example, if a component is stored in the database as `hand`, you would not find it if you typed `Hand` or `HAND`. Therefore, it is important that your organization sets a naming convention that everyone follows.

Note: We recommend that you do **not** use mixed case in your naming convention; rather, use either all lowercase or all uppercase.

Finding parts

There are three filters that you can use to find parts within TeamConnection:

Parts Use this filter when you want to limit your search to a particular context of a workarea or driver in a release, or a particular version of a release. This is generally the view users will use most often.

If you specify only a release, TeamConnection lists the committed parts for that release. However, if you want a list of all parts in a specified workarea and release, TeamConnection displays all the parts visible to the workarea. This includes parts that are committed to the release as well as changed parts that are visible only to the work area.

PartsOut
Use this filter to display a list of the files that you have checked out of TeamConnection. This filter is optimized to do fast searches.

PartFull
Use when you want to search for parts across releases, components, or workareas. For example, suppose you want a list of all the optics.c parts. Unlike the Parts filter, you can specify one or more release or workarea names.

You can also use this filter to display only parts that have been changed in a workarea. For example, you check out robot.c to workarea 310:1, and that is the only part that you have changed. If you use the PartFull Filter to query for all the parts in workarea 310:1, only one record is returned.

Note: When creating a filter, use the forward slash (/) when specifying path names, such as directory/file.txt. The Intel convention of using the backward slash is not recognized in the Filters window.

Refer to the online help for more information on how to use Filter windows.

Using workareas

A workarea is a logical temporary work space that enables you to isolate your work on the parts in a release from the official versions of the parts. You can check parts out to a workarea, update them, and build them without affecting the official version of the parts in the release. You must create a workarea before you can create, check out, or check in parts. If your component's process includes a design, size, review subprocess for defects or features and the release follows a tracking subprocess, a workarea is automatically created when sizing records exist and the associated defect or feature is accepted. TeamConnection associates these workareas with the appropriate defect or feature.

The parts in a workarea do not become available in the release until the workarea is integrated. Also, if your release follows a driver subprocess, parts that have been changed do not become available in the release until the associated driver is committed. However, users who have the authority to access the workarea can view and work with the parts in it.

You can save intermediate versions of the parts in your workarea by *freezing* your workarea. Every time you freeze a workarea, TeamConnection saves a revision level of the workarea. When you freeze workarea 123:1, for example, a version called 123:2 is created. This version contains information about each part in the workarea and its current version at the time the workarea was frozen. It may contain version 1 of part optics.c, for example. If you freeze the workarea again later, a new version called 123:3 is created with information about the versions of the parts in the workarea when it was frozen. This version may contain version 2 of part optics.c. Each of these workarea versions is saved in the database and you can retrieve the versions of the parts they contain before you integrate the workarea into the release. Therefore, you should freeze a workarea whenever there is a possibility that you will want to return to that version of the workarea. For example, you might be adding a major feature to the code, and you want to be able to return to something that works in case the application no longer builds. When you integrate a workarea or commit a driver, the workarea is frozen automatically.

Naming your workareas

When TeamConnection automatically creates a workarea, the workarea is given the same name as the defect or feature it was created for plus the initial version number, :1. When you create a workarea, you can also give it the same name as the defect or feature, or you could give it any other name. Where possible, we recommend that you name it after a defect or feature, or relate the name to the change that is being made.

Here are some things you should know before you name a workarea:

- Workarea names must be unique within the context of a release.
- After you create a workarea, you cannot delete it. You can, however, cancel the workarea in the following situations:
  – No part changes were made.

  – You undo the changes you made.
- With the proper authority, other users in your organization will be able to access your workarea and make changes to the parts. This means that you need to make it easy for them to locate the workarea. Following your local naming conventions will help.
- After the workarea is integrated with the release, you cannot reuse the workarea. If the defect is still in the working state, you can create another workarea with a different name after the initial work area is integrated with the release.

Guidelines for using workareas

Workareas enable you to track changes to your parts and to return to an older version of your parts. The way you use workareas, however, can affect performance. The more changes you make in a workarea, the longer it takes to integrate the workarea. Also, if you leave workareas in the Fix state for extended periods of time, system performance will start to degrade. Consequently, you should integrate your workarea regularly. As a rule of thumb, you should not make more than 500-1000 changes in a workarea before integrating the workarea.

Another thing you can do to increase performance in relation to workarea use is to prune your release. When you prune a release, you delete a branch of versions from the release. The branch consists of all the versions of a specific workarea that have been integrated.

For example, suppose you have a work area and a corresponding version called REL1. Then, you freeze the work area multiple times, creating versions REL1:2., REL1:3, and REL1:4. Finally, you integrate your REL1 work area. Pruning version REL1 deletes versions REL1 and REL1:2, REL1:3, and REL1:4.

You can prune a release by selecting **Release → Prune** from the Actions menu.

If you don’t want to manually prune your releases, you can enable automatic version pruning for the release. When autopruning is enabled, workarea versions are automatically deleted when you integrate a workarea or commit a driver to the release. Be careful if you enable autopruning, because when workareas are deleted, tracking information is also destroyed.

Creating parts

A TeamConnection part is controlled by the TeamConnection server. A TeamConnection part is uniquely identified by the *path name* of the part, the part *type*, and the name of the release in which it is contained. You must specify both the release name and the path name whenever you perform a TeamConnection action on a part. Multiple releases can share the same part.

When you create a part, you do one of the following:
- Take an existing text or binary file that is on your workstation and place it into TeamConnection.
- Create an empty part that has no content. Empty parts are used as place holders until an application is built. For example, you can create a place holder for an executable part that will be created by a build. See “Creating the build tree for the application” on page 165 for an example of creating a place holder.

After you put a part under TeamConnection control, the official copy of the part resides in the database. The copy on your workstation is changed to read-only mode. You can then change the part by checking it out to your workstation.

Use the online help facility if you need assistance when creating parts.

Naming your parts

If your organization has a naming convention, be sure to follow it when naming your parts. Part names created on the server are case-sensitive and therefore must be retrieved using the same case in which they were created.

When you name TeamConnection parts, you can specify only the base name, such as hand.c, or you can specify the directory path in addition to the base name, such as robot\hand\hand.c. Specifying the path name as part of the name lets you have several identical base name parts included in the same release, such as `robot\hand\msg.h` and `robot\optics\msg.h`.

You can also have identical part names within the context of a release as long as their part types are different, such as TCPart and vgdata.

Note: We recommend that you do **not** use mixed case when naming your parts; rather, use either all lowercase or all uppercase.

A part name can contain spaces if it is enclosed in double quotation marks during processing. For example:

```
teamc part -create "This is a long file name.txt"
```

The name with spaces will be shown as-is by the GUI (without the double quotes). If the name has spaces and is not enclosed in double quotation marks, then you may get an error message repeated many times, one for each ″token″ separated by spaces in the long name.

Note: The base name may contain a maximum of 63 characters, not including the double quotations. The path name, which includes the base name, may contain a maximum of 195 characters, not including the double quotations.

Preparing to build your parts

If you are going to use the TeamConnection build function, you must provide certain information about each part that is used in the build. You can provide this information when you create the parts or at a later date. You can also change the information at any time.

To associate a part with a build, you must specify the following information:
- The parent part that you want to associate the part with.
- The type of relationship the part has to the parent, such as:

  Input The part will be used as input to building its parent. An example of an input part is a C language source file, x.c, which is compiled to create its parent, x.obj.

  Output
  The part will be a generated output from the same build that creates its parent part. In other words, both the parent part and this child part are outputs when the parent part is built.

Dependent
The part will be needed for the build operation of its parent to complete, but it will not be passed directly to the builder. An example of this is an include file.

If you do not provide this information when you create the part, you can provide it later using the *connect* function.

You can also specify the builder or parser that a part will use, as well as any build parameters.

"Part 4. Using TeamConnection to build applications" on page 115 explains the build function in more detail.

Working with parts

After you create parts in TeamConnection, you will be checking them out to your workstation so that you can edit them, and then checking them back into TeamConnection. This section gives a brief overview of these tasks. "Chapter 5. Working with no component or release processes" on page 49 and "Chapter 6. Working with component and release processes" on page 75 go into more detail about these and other TeamConnection tasks.

Working in serial or concurrent development mode

A release is set up for either serial development or concurrent development mode. Once the development mode is set, you can change from serial mode to concurrent mode, and (under special circumstances outlined in the Administrator's Guide) from concurrent mode to serial mode. In serial development, a part is locked when a user checks it out, and no one else can update the part as long as it is checked out. In concurrent development, more than one user can simultaneously have the same part checked out.

In concurrent development, more than one user can check out and change the same part. Prior to integrating their changes, each user should refresh their workarea from the driver in which they plan to put it. The first user will be able to integrate their workarea, complete fix records for tracking releases, with their release. When the next user refreshes their workarea, TeamConnection recognizes that the parts differ and notifies them. It is up to this user to resolve the differences using the TeamConnection merge program or some other merge program. If the user fails to refresh their workarea from the driver, TeamConnection will not notify them that the parts differ until they try to add the workarea to the driver. They will then have to put the fix records back in the fix state, reactivate the workarea, refresh the workarea, reconcile the differences, refresh their workarea again, and reintegrate their workarea before they can add their changes to the driver.

Before getting parts from TeamConnection, you might want to determine if the development mode for the release is concurrent or serial. To do this,

1. Create a filter that displays the release.
2. Select **View** from the Release's pop-up menu.

Working with common parts

A *common* part is a part with identical content that is shared by two or more releases or two or more workareas. For example, when an identical part is needed in two separate releases, you can link the part from one release to the other (if you have the proper authority). Both releases would then have a link to the current version of that part.

When a common part is checked out of a release, TeamConnection locks the current version of the part in all releases if one of them uses serial development. When putting the part back into the release, one of the following actions reflects the change in all releases in which the part is common:

- You integrate the workarea when the driver subprocess is not followed, or
- You commit the driver when the driver subprocess is followed.

You can break the common link if you make changes to a common part and you do not want these changes reflected in other releases or workareas that link to the part. You can break the common link when you check out, check in, rename, delete, re-create, connect, or *disconnect* parts. When a part is common to more than two releases, you can maintain the common link with some of the releases while breaking the link with other releases. When a link is broken, the parts still share the same name, but the information contained in the parts is different.

Parts can also be linked between two or more workareas in the same or different releases, making the parts common to those workareas. For example, a user working in one workarea can link to the latest version of a part in another workarea of the same release (the part has yet to be integrated with the release). The part is then common to the two workareas within the same release. If you want to maintain the common link to all workareas, you must specify the names of the common workareas when you check in, rename, delete, or re-create the parts. As with common parts in releases, you can break the common link.

You can also link all the parts within a release to another release. This function is especially helpful when development begins on a new release of a product, and you want the parts in the new release to initially be the same as the parts in the current release. As development of the two releases continues, the common link between the parts can be broken to separate development of the new release from maintenance of the current release.

For more information about how to link parts, refer to the *Commands Reference* and online help.

Getting parts from TeamConnection

Checking out a part implies that you intend to modify it; extracting a part merely gives you a copy of the part. Normally, when you extract a part, you do not plan to change the current version in TeamConnection.

You must have the necessary authority to a component before you can check out or extract parts from that component. You need *PartExtract* authority to extract a part from TeamConnection; you need *PartCheckOut* authority to check a part out. See “Appendix H. Authority and notification for TeamConnection actions” on page 235 for a listing of all the TeamConnection actions and their authority requirements.

Parts are checked out to workareas. The workarea is where you store updated parts and do builds without affecting the version of the parts in the release. When a part is checked out of the release to the specified workarea, TeamConnection locks the part in the release if you are working in serial development. If you are working in concurrent development, the part is never locked. TeamConnection also puts a copy of the part on your workstation. It is here where you update the part. If a read-only copy of the part already exists on your workstation, it is renamed and saved as a backup copy. The renamed file will have an extra character inserted before the extension. On OS/2 and Windows platforms, this extra character is a ″$″ (dollar sign). On Unix platforms, this character is a ″_″ (underscore). If a backup copy already exists, it is deleted. When you are finished updating the part, you check it back in to the workarea. A workarea is optional when extracting a part.

When you extract a part, TeamConnection copies the part to your workstation without locking it. In other words, other users can still check out the same version of the part and make changes to it, even in serial development mode. By default, TeamConnection sets the extracted part to read-only access. This is done to keep you from inadvertently changing the part on your workstation when the part in TeamConnection is not locked. You can, however, change this when you are extracting the part. When you do this, be aware that someone else can change the official part in TeamConnection, making your workstation copy back level.

Where TeamConnection places a checked-out or extracted part on your workstation depends on the following:

- Your workstation’s *current working directory*
- Whether you use the -relative flag on the command line or the **Destination directory** field on the GUI

For more information about how these interact, refer to the part command examples in the *Commands Reference*.

When you want to make changes to a part, you can do the following:

- Check out one or more parts and edit the parts on your workstation. When you finish making changes to the parts, you check them back in.

If you are working in concurrent development and someone else changed a part while you had it checked out, you are asked to resolve the differences when you try to integrate your workarea.

Checking parts in to TeamConnection

After you have verified the accuracy of your part changes, you are ready to check them in to TeamConnection. Any parts that you have checked out, you have the authority to check back in.

As mentioned earlier, you check parts out to a work area so you can work on them. Therefore, when you check in a part, you must specify the workarea where that part is checked out. In other words, you check the part back in to the same workarea. When the part is checked in, the copy on your workstation is flagged read-only.

At this time, the changed part is visible in only the named workarea; it is not visible at the release or to any other workarea. This lets you test your changes by building the version of the code that is in your work area.

When you are satisfied with your changes, you can integrate the parts into the release by integrating your workarea. This action makes the workarea visible to all the users in the release.

If you are working in concurrent development mode, TeamConnection generates a *collision record* when a changed part conflicts with a previously committed part. For example, both you and Keith have hand.c checked out. Keith makes changes to the part and then integrates the workarea that contains that part. (Depending on the process being followed, Keith might have to commit the workarea rather than integrate it.) Later, after making changes to hand.c, you attempt to integrate the workarea that contains the part. Because the part was already integrated by Keith, you are notified of a collision and asked to refresh your workarea. After the refresh, you can view the collision record and decide how you want to resolve the conflicts. "Reconciling differences" on page 72 explains in more detail how this works.

Finding different versions of TeamConnection objects

TeamConnection version control maintains different versions of the following objects:
- Releases
- Workareas (and driver members)
- Drivers
- Parts

When you want to find and retrieve previous versions of these objects, it is helpful to know how TeamConnection creates and deletes previous versions of each object.

Some basics of TeamConnection versioning will help you understand how TeamConnection identifies unique versions of objects:
- When you first create an object, the initial version name is the object name suffixed with *:1*. When you create a new workarea called *myWorkArea,* for example, its version is *myWorkArea:1.* Subsequent versions are identified in numerical order: myWorkArea:2, myWorkArea:3, myWorkArea:4, and so on. Versions of releases and drivers are identified similarly: myRelease:1, myRelease:2, myRelease:3; myDriver:1, myDriver:2, myDriver:3; and so on.
- Unique versions of parts are identified by association with a specific version of a release, workarea, or driver. Your TeamConnection family may have three different versions of a part called *myPart,* for example: one associated with release myRelease:2, one associated with workarea myWorkArea:1, and one associated with workarea myWorkArea:2.

Versioning releases

TeamConnection creates new versions of releases whenever you do the following:
- Create a release

  This is the initial version of a release and contains no parts. When you create myRelease, for example, its version name is myRelease:1 and it contains no parts.
- Commit a workarea to the release

  Committing a workarea to a new release creates a new version of the release and adds the parts in the workarea to the release. When you commit workarea myWorkArea:1, for example, to myRelease:1, TeamConnection creates a version of myRelease called myRelease:2. It also associates the parts in myWorkArea:1 with myRelease:2.

- Commit a driver to a release

  Because drivers are simply collections of workareas, committing a driver to a release has the same effect as committing a workarea: TeamConnection creates a new version of the release. When you commit myDriver:2 to myRelease:2, for example, TeamConnection creates a version of myRelease called myRelease:3.

TeamConnection deletes versions of releases whenever you prune the release. Refer to “Guidelines for using workareas” on page 28 for an explanation of pruning.

Versioning workareas

TeamConnection creates new versions of workareas whenever you do the following:

- Create a workarea

  This is the initial version of a workarea. When you create myWorkArea, for example, its version name is myWorkArea:1.

- Refresh a workarea

  Refreshing a workarea updates it with any new versions of parts that have been integrated with the release. When a workarea is refreshed, two versions of the workarea are created. One of the contents before the refresh and one with the contents after the refresh.

- Freeze a workarea

  Freezing a workarea is like taking a snapshot of the workarea. It preserves the parts as they are at a given point in time. If you create workarea myWorkArea:1, add three new parts to it — called part1, part2, and part3 — and then freeze it, your family contains a workarea called myWorkArea:2, with part1, part2, and part3. The version name of each of these parts is myWorkArea:1. If you then alter part2 and freeze the workarea again, your family contains the following:
  - myWorkArea:1, with nothing in it
  - myWorkArea:2 contains part1, part2, and part3 at version myWorkArea:1
  - myWorkArea:3 contains part1 and part3 at version myWorkArea:1, and part2 at version myWorkArea:2

- Commit a workarea

  Committing a workarea adds the parts in the latest version of the workarea to the release. It also does the following:
  - Creates a new version of the release
  - Creates new versions of the parts in the release

  Using the previous example, if you commit myWorkArea:3 to myRelease:1, the following happens:
  - TeamConnection creates a new version of myRelease called myRelease:2.
  - TeamConnection creates new versions of the parts in myRelease:2.

Once a workarea has been committed, you can no longer use it for making part changes, and you cannot create a new workarea with the same name.

TeamConnection does not, by default, delete versions of workareas when you commit the workarea to the release. However, you can use the Prune Release window to manually delete or prune versions of the workarea or driver from the release. You can also enable the autopruning option for the release. If you enable autopruning, workarea versions are automatically deleted when you integrate a workarea or commit a driver. However, keep in mind that when autopruning is enabled:

- Performance can degrade when pruning occurs
- The version history for the workarea and driver is destroyed

Refer to the *Administrator's Guide* for more information on autopruning.

Versioning drivers

TeamConnection creates new versions of drivers whenever you do the following:

- Create a driver

  When you create a new driver, TeamConnection makes two versions of it: myDriver:1, for example and myDriver:2.
- Add a workarea (driver member) to a driver

  If you add myWorkArea:1 to myDriver:2, for example, TeamConnection creates a new version of myDriver called myDriver:3.
- Freeze a driver

  Freezing a driver is like taking a snapshot of the driver. It preserves the parts as they are when the driver is frozen. If you freeze myDriver:3, for example, TeamConnection creates a new version called myDriver:4.
- Refresh a driver

  Refreshing a driver updates the driver with all changes that have been made in all of its driver members. Refreshing a driver actually creates two new versions of the driver, as follows:
  1. Freezes the driver (so that TeamConnection can have a point to roll back to if an error occurs during the refresh operation).
  2. Updates the driver with any changes from the driver members
  3. Freezes the driver again, thus preserving a copy of the updated driver.

  If the current version of myDriver is myDriver:2, for example, and the parts in its driver members have been changed, then TeamConnection does the following when it refreshes the driver:
  1. Freezes myDriver, creating myDriver:3.
  2. Updates myDriver with changes from its driver members.
  3. Freezes myDriver again, creating myDriver:4.

  The result of refreshing myDriver (version myDriver:2) is two new versions: myDriver:3, containing a snapshot of the driver before the refresh, and myDriver:4, containing a snapshot of the driver after the refresh.

TeamConnection deletes versions of drivers whenever you remove driver members or commit a driver to a release.

- If you have a driver version myDriver:4 with driver members myWorkArea, yourWorkArea, and ourWorkArea, and you remove myWorkArea, then TeamConnection deletes driver versions myDriver:2, myDriver:3, and myDriver:4 and creates a new driver version called myDriver:5 containing members yourWorkArea and ourWorkArea. As a result, the family contains two versions of the driver, myDriver:1 and myDriver:5.
- When you commit a driver to a release, all intermediate versions of the driver (resulting from driver member add, driver freeze, driver refresh, or driver member remove operations) are deleted.

Versioning parts

TeamConnection versions parts in association with other TeamConnection objects, such as workareas. If, for example, you create part1 in myWorkArea:1, the current version of part1 is myWorkArea:1. If part1 is in release myRelease:2 and workarea myWorkArea:2, then you can view the version of the part for either the release or the workarea. The version label for part1 in myRelease:2 is myRelease:2 and in myWorkArea:2 is myWorkArea:2.

TeamConnection deletes part versions whenever it deletes versions of the object that the part is associated with. In addition to versioning in association with other TeamConnection objects, TeamConnection maintains versions of build output parts (parts that are created as the result of a build, such as an .exe file or a .hlp file). When you create a release, you can set the maximum number of versions of build output parts to maintain. If you set this maximum to 10, for example, then TeamConnection saves only 10 versions of build output parts and discards the oldest version each time a new version is created.

Working with defects and features

Defects are used to report problem information; features are used to record information about proposed design changes. After a defect or feature is opened, TeamConnection tracks the progress of the defect or feature through various states. The degree to which defects and features are tracked depends on the processes followed by the release and component to which they are assigned. The following describes actions that your defined processes might require:

Analyzing defects and features
The owner is responsible for analyzing a defect or feature after it is opened. The owner can then return it if it is not valid or feasible, reassign it to another user or component, or accept it for resolution.

Designing the resolution
After a defect or feature has been accepted, the actual resolution needs to be designed so that an informed evaluation can be made. This resolution needs to be designed by users who are familiar with the product or area affected by the defect or feature.

Identifying the required resources
Sizing records are created by the owner to identify the components and releases that might be affected by the defect or feature. Each owner of a component that is referenced in a sizing record needs to evaluate the impact of the defect or feature on the parts managed by the component. If the defect or feature requires changes to parts, the sizing record is accepted and sizing information is added.

When sizing records exist and the associated defect or feature is accepted, TeamConnection automatically creates a workarea.

Reviewing the design and resource estimates
After the resolution has been designed and the resources have been identified, the proposal needs to be reviewed. If the review indicates that work should continue on the defect or feature, it is accepted.

Resolving defects and implementing features
Resolving one defect or implementing one feature in one release can involve one or more users changing many parts. To change a part, a user must check out the part, make the changes required to resolve the problem

or implement the design change, and check the part back in. If the release follows a tracking process, all defects or features must be associated with a workarea. Parts that are checked out refer to the workareas that are monitoring the defect or feature.

Resolving a defect or implementing a feature also involves integrating the changed parts with changes made for other defects and features in that release. All changed parts are eventually integrated with the unchanged parts within the release.

Verifying the resolution of the defect or feature
The originator uses a verification record to acknowledge that the defect or feature was satisfactorily resolved or not. Accepting a verification record formally closes the defect or feature. Rejecting a verification returns the defect or feature to the working state.

“Chapter 4. The states of TeamConnection objects” on page 39 explains in more detail the various states that different TeamConnection objects can go through depending on the process that is being followed. A diagram in this chapter shows the flow of these states. You might want to study this information before you start to work with defects and features.

Testing and verifying part changes

You can use TeamConnection’s build function to build your program. Before you check in updated parts, you will probably want to verify the accuracy of your changes.

The scenarios in “Chapter 5. Working with no component or release processes” on page 49 and “Chapter 6. Working with component and release processes” on page 75 include information about testing and verifying part changes. “Part 4. Using TeamConnection to build applications” on page 115 provides detailed information about the build function.

Chapter 4. The states of TeamConnection objects

The actions that you can perform on certain TeamConnection objects are controlled by two factors:
- The process followed by the component and by the release
- The current state of the object

Certain TeamConnection objects follow certain states through their life cycle. An instance where an object might not follow all the possible states is when it moves through the states defined in the subprocesses of the component and the release. The following table briefly lists the component and release subprocesses. For more information on component and release subprocesses, refer to the *Administrator's Guide*.

Component subprocesses
- **dsrDefect** - Design, size, and review fixes to be made for defects
- **verifyDefect** - Verify that defect fixes work
- **dsrFeature** - Design, size, and review changes to be made for features
- **verifyFeature** - Verify that feature changes work

Release subprocesses
- **track** - Relate all part changes to a specific defect or feature and a specific release
- **approval** - Require all changes to be approved before incorporating them into a release
- **fix** - Use fix records to ensure that all required changes are made
- **driver** - Use drivers to integrate changes into a release
- **test** - Require all changes to be tested before they are integrated into the release

This chapter explains the possible states of certain TeamConnection objects and how objects are moved from one state to the next. (See "LifeCycle Navigator" on page 17 for a description of a tool that helps you work with objects that have life cycles.)

This chapter also explains how component and release subprocesses affect the flow of states. For a diagram showing the flow of states, see Figure 12 on page 43.

Defects and features

Use defects to report problem information; use features to record information about proposed design changes. After you open defect or feature, TeamConnection tracks the progress of the defect or feature through various states. Defects and features are tracked according to the processes followed by the release and component that they are assigned to. The possible states for defects and features are:

Open state

When you open a defect or feature, it is in the open state and you are considered the originator.

You assign the defect or feature to a component. The owner of this component becomes the feature or defect owner and is responsible for managing its resolution. The component you open a defect or feature against should be one that manages the parts affected by the enhancement or problem. Use the component descriptions and the structure of your family's hierarchy to find the most appropriate component. If you open a defect or feature in an inappropriate component, the component owner can reassign it.

While the defect or feature owner is responsible for implementation, the originator is responsible for verifying that the defect or feature is resolved correctly.

Returned state

A defect or feature owner can return a defect or feature to its originator. You can return a feature or defect from the open, design, size, or review state if you decide that the defect or feature is not feasible or not valid. You can return a defect or feature back to the working state only if it has no associated workareas. If there are associated workareas, you must cancel or undo them before you can return the defect or feature. When you return a defect or feature, add your reason for returning it so that the originator and any other users can evaluate why you believe it is not feasible or not valid.

Canceled state

A feature or defect in the open or returned state can be canceled only by its originator or by a superuser. A canceled defect or feature remains inactive unless it is reopened by the originator.

Design state

If the component to which a defect or feature is assigned includes the dsrDefect or dsrFeature subprocess, you move defects or features in the open or returned state to the design state.

In this state, the proposed change is designed, and a description of the design change is entered. The owner must describe the design change before the defect or feature can move to the next state.

If the release includes the fix subprocess, fix records are automatically created when a defect or feature is designed.

Size state

Defects or features move to this state after the owner enters design information.

In this state, users can create a *sizing record* for each release that contains parts affected by the enhancement or problem. A sizing record identifies the work that is required for and the resources affected by the defect or feature. The owner of the component that is referenced in the sizing record is the owner of the sizing record. The owner is responsible for entering information about the amount of work that is required to implement the feature or resolve the problem.

The sizing record owner can reject the sizing record if it does not affect the specified component. After all sizing records are either accepted or rejected, the defect or feature moves to the review state or returns to the design state if more design information is needed.

Review state

Defects or features move to this state after they have been sized. In this state, the design text and sizing records are reviewed to determine the feasibility of the proposal. The owner can do one of the following:

- Accept the defect or feature if all design and sizing records are acceptable. This moves the defect or feature to the working state.

- Return the defect or feature to the originator if all design and sizing records are not acceptable. If necessary, the originator can reopen a defect or feature.
- Move the defect or feature back to the design state if design modifications are needed.

Working state

Defects or features move to this state when the owner accepts the defect or feature when it is in the:

- Review state, if the component includes the dsrDefect or dsrFeature subprocess
- Open state, if the component does not include the dsrDefect or dsrFeature subprocess

When you accept a defect or feature, you accept the responsibility of resolving it. A defect or feature might require changes in more than one release.

What happens after a defect or feature is accepted varies according to the subprocesses in effect:

Component subprocesses

- **dsrDefect** or **dsrFeature** - TeamConnection creates a workarea in the approve state for each release identified in the accepted sizing records for the defect or feature.
- **verifyDefect** or **verifyFeature** - TeamConnection creates verification records in the notReady state.

Release subprocesses

- **fix** - TeamConnection creates fix records in the notReady state based on the sizing records.
- **approval** - TeamConnection creates approval records for each user on the release's approver list.

If the component does not include the dsrDefect or dsrFeature subprocess, then you must manually create a workarea before you can check out or create parts to address the defect or feature.

Verify state

Defects and features go through the verify state only if their component includes the verifyDefect or verifyFeature subprocess. Defects and features are automatically moved to this state when all workareas (there can be multiple workareas for the defect or feature) are integrated.

When a defect or feature is accepted, TeamConnection creates a *verification record*. This record lets the originator:

- Accept the fix if the resolution was satisfactory
- Reject the fix if not satisfied with the resolution
- Abstain if unable to assess the resolution

Once all verification records have been accepted or abstained, the defect or feature moves to the closed state. If a verification record is rejected, the defect or feature returns to the working state. The defect or feature cannot be closed until the verification records are accepted.

A defect or feature can have more than one verification record. For example, if defect 123 is returned because it is a duplicate of defect 122, a second verification record is created for defect 122. The originator of defect 123 is the owner of the second verification record for defect 122. If the originator is the same for both defects, only one verification record is created.

Note: For a discussion of verification records and test records, see "Verification and test records" on page 48.

Closed state
The closed state is the final state of a defect or feature.

If the defect is associated with multiple workareas, the defect will remain in the working state until all of the workareas are integrated.

If the component includes the verifyDefect or verifyFeature subprocess, the defect or feature automatically moves to the closed state after all verification records are in the accept or abstain state and all workareas are in the complete state. If a verification record is rejected, the defect or feature moves back to the working state. Otherwise, the defect or feature moves directly from the working state to the closed state when the first workarea moves to the complete state.

You cannot re-open a defect or feature that is in the closed state. If the defect or feature was not resolved correctly, you must open a new defect or feature to address the necessary changes.

The following is a graphical representation of the states of TeamConnection objects.

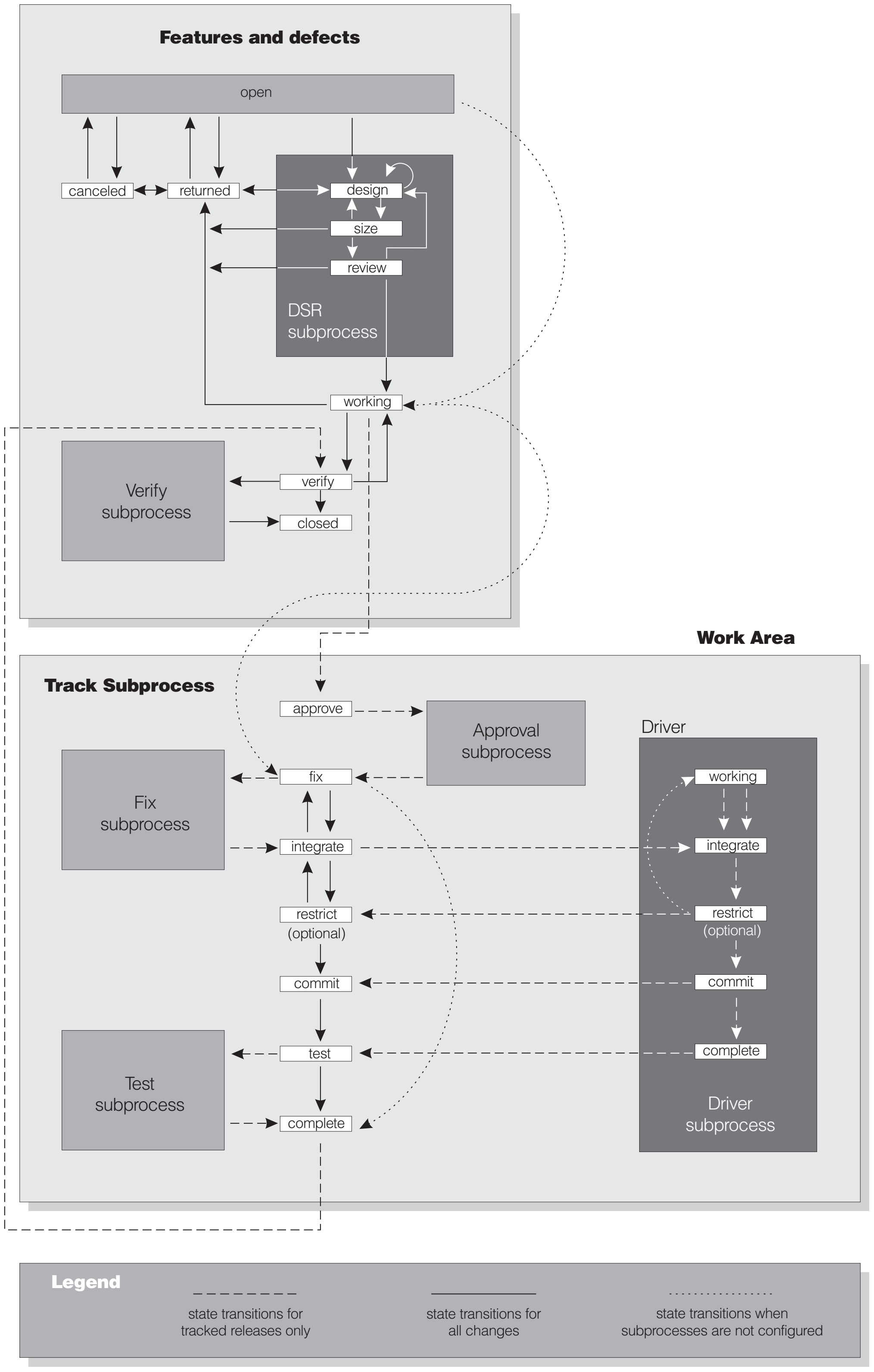


Figure 12. States of TeamConnection objects

The states of workareas

A workarea is a storage area where you can work on the parts in a release without affecting the ″official″ versions of those parts. A workarea can be associated with a specific defect or feature, but it does not have to be. These attributes can affect the state of a workarea:

workareafixhold	With the workareafixhold attribute and the fix subprocess, a workarea will remain in the fix state rather than moving to the integrate state when the final `fix -complete` command has been issued. A `workarea -integrate` command must be issued to move the workarea into the integrate state.
workareacommithold	If an environment list exists for the release associated with the workareas, the automatic transition to the test state can be disabled by including the `-workareacommithold` attribute in the release process. A `workarea -test` command must be issued to move the workarea into the test state.
workareatesthold	With the workareatesthold attribute and the test subprocess, a workarea will remain in the test state rather than moving to the complete state when the final test is marked. A `workarea -complete` command must be issued to move the workarea into the complete state.

Approve state

When a workarea is created, it goes to this state if the release includes the approval subprocess. TeamConnection creates an approval record for each user on the release’s *approver list*. Each approver indicates their evaluation of the changes in their approval record:

- Accept that work should continue
- Abstain if unable to assess if work should continue
- Reject if work should not continue

When all approval records are marked as abstain or accept, the workarea goes automatically to the fix state. If any approval record is marked as reject, the state of the workarea remains at approve. You can change rejected approval records to accept or abstain.

Fix state

If the release does not include the approval subprocess, workareas for the release begin in the fix state.

While the workarea is in this state, parts are checked out to the workarea, changes are made to these parts, and builds are done to verify the accuracy of the changes.

If the release includes the fix subprocess, any fix records created for a defect or feature move to the active state when a part change is associated with the workarea for the defect or feature. A fix record monitors the part changes within a single component. Fix records provide a mechanism for reviewing all part changes that apply to components before integrating those changes with changes made for other defects and features.

How fix records are handled varies according to the subprocesses in effect:

Component subprocesses

- **dsrDefect** or **dsrFeature -** TeamConnection creates fix records for features or defects when existing sizing records are accepted.

Release subprocesses

- **fix -** If a fix record does not already exist for the component, TeamConnection creates one when a part managed by that component is checked in to the database.

If neither of these subprocesses are in place and a defect or feature owner needs to create a workarea manually, he or she can create fix records at the same time. Existing fix records go to the active state when a part is checked in to the workarea.

Fix records provide a way of ensuring that all necessary part changes within the specified component have been made and are reviewed or inspected. The fix record owner is responsible for this review. When the fix record owner is satisfied that the part changes made within that component are complete and ready for integration with other parts in the release, the owner marks the fix record as complete. When all existing fix records for a workarea are complete, the workarea automatically moves to the integrate state.

Integrate state

Workareas can be moved to the integrate state as follows. If the release includes the fix and driver subprocesses, the workarea automatically moves to the integrate state when all fix records are complete. If all fix records are not complete, you can force a workarea to the integrate state, provided that no part changes are associated with the workarea. If the release does not include the fix and driver subprocesses, you must move the workarea to the integrate state manually.

While a workarea is in integrate state, you must add it to an existing driver as a driver member if the release includes the driver subprocess. All workareas in the integrate state do not have to be added to the same driver. The workarea stays in the integrate state until the driver in which it is a member is committed.

You can move workareas from the integrate to the following states, according the subprocesses in effect:

Release subprocesses

- **driver -** A workarea moves to the commit state when the driver it is a member of is committed, or to the restrict state when the driver is restricted. You can also force a workarea to the commit state, provided that no part changes are associated with the workarea.
- **test -** A workarea moves to the test state so that test records can be approved or rejected.

If the release does not include these subprocesses, you can manually complete a workarea in the integrate state.

Restrict state

Workareas can be moved to the restrict state only when the release includes the driver subprocess. The workarea moves automatically to the restrict state when the driver to which it belongs is restricted. If a workarea in this state is removed from the driver, it returns to the integrate state. Otherwise, the workarea remains in the restrict state until the driver to which it belongs is committed.

Commit state

Workareas can be moved to the commit state only when the release includes the driver subprocess. The workarea moves automatically to the commit state when the driver to which it belongs is committed. At this point, all parts that were changed in this release to resolve the feature or defect are committed. The workarea remains in the commit state until the driver to which it belongs is completed.

Test state
Workareas can be moved to the test state only when the release includes the test subprocess. When the associated driver moves to the complete state or when a workarea is committed without a driver, the workarea moves to the test state. The driver is then ready for formal testing in the specified environments. Test records for the workarea are created in the ready state when the workarea moves to the test state. The workarea stays in the test state until all test records are accepted, rejected, or abstained.

Complete state
The complete state is the final state of a work area; the workarea can no longer be used. If the test subprocess is not included in the release process, the workarea moves directly to this state when the associated driver is completed or when the workarea is explicitly integrated.

When a workarea is completed, the feature or defect associated with that workarea automatically moves to the verify or complete state. The defect does not leave the working state until the workarea for that release is completed.

The states of drivers

Drivers monitor and implement the integration of part changes within a release. Those part changes are included in a driver by adding the workareas containing the changed parts to the driver as driver members.

Working state
The working state is the initial state of a driver. While the driver is in this state, it is not associated with any workareas and, therefore, contains no part changes.

If the release includes the driver subprocess, drivers can be explicitly created at any time.

Integrate state
Each driver automatically moves to the integrate state when the first workarea is added to it as a driver member. If all work areas are removed from the driver, the driver automatically returns to the working state.

Workareas can be added to drivers as driver members when the driver is in the working, integrate, or restrict state and the workarea is in the fix state. Adding driver members to a driver in restrict state requires proper authority.

You can extract the driver when it is in the integrate state; however, only those parts that were changed in reference to driver members are extracted. This is referred to as extracting a *delta part tree*.

Restrict state
Before a driver is committed, you can move it to the restrict state. While a driver is in this state, workareas in the integrate state can be created for or deleted from the driver by only a superuser or an individual with the special authority of memberCreateR or memberDeleteR. This allows a build administrator to have better control over what is being built. The build administrator can delete driver members that are causing build errors or add driver members to fix build errors. You can then commit an error-free driver.

When a driver moves to the restrict state, all workareas that are included as driver members also move to the restrict state.

Commit state

Committing a driver commits all workareas included as driver members and all parts that were changed in reference to those workareas. TeamConnection commits only a successfully built driver. Committing a driver changes it to the commit state. You can, however, manually commit the driver. You can also commit an unsuccessful driver by using the force option.

When a driver moves to the commit state, all work areas that are included as driver members also move to the commit state. When a workarea is in the commit state, all part changes associated with the workarea become the ″official″ versions of the parts in the release and are visible to all users of the release.

A committed driver can be extracted as a full part tree as well as a delta part tree. A full part tree is the part structure of all the parts within the release.

Complete state

The complete state is the final state of a driver. In this state, the driver is ready for formal testing in the specified environments.

If the release includes the test subprocess, the workareas that are included as driver members move to the test state. Any existing test records for the workarea move to the ready state when the workarea moves to the test state. The workarea stays in the test state until all test records are accepted, rejected, or abstained.

Test records are used to record the outcome of environment tests for changes implemented in a driver. This record lets the owner:

- Accept the record if the test was satisfactory
- Reject the record if not satisfied with the test results
- Abstain if unable to assess the results

Once all test records have been accepted or abstained, the states of other objects change as follows:

- **Workareas** - Go to complete state.
- **Defects and features** - Go to verify state if the component includes the verifyDefect or verifyFeature subprocess; otherwise they go to the closed state.
- **Verification records** - Go to ready state and are sent to the defect or feature originators.

If the test subprocess is not configured, then workareas move to the complete state and any defects or features move to the verify state.

If the component includes a verifyFeature or verifyDefect subprocess, verification records move to the ready state and notification is sent to the originators of any defects or features that were addressed in the completed driver.

The commit and complete states of drivers differ as follows:

- When a driver is committed, all workareas are committed, but no changes occur in the states of defects or features associated with the workareas.

- When a driver is completed, then the states of other associated objects (such as test records, workareas, verification records, defects. and features) change according to the other subprocesses in effect:
  - **test** - Workareas go to the test state and test records are created in the ready state for each environment in the release's environment list.
  - **verify** - Verification records go to the ready state.

  If the release includes neither of these subprocesses, then the work area goes to the complete state and all features and defects associated with the workarea are closed.

Verification and test records

If you use both the verify component subprocess (verifyDefect or verifyFeature) and the test release subprocess, then TeamConnection creates both verification records for features or defects and test records for each environment defined in the release's environment list. These records serve different purposes:

- Verification records provide a means of accepting or rejecting the product changes made in response to defects or features and are thus specific in nature.
- Test records provide a means of accepting or rejecting the results of a build and are more global in nature.

These records are handled by different people and enable you to monitor your development progress in different ways. The sequence of creating and handling verification and test records is as follows:

1. Verification records are created in the notReady state when a defect or feature is accepted. This indicates that someone on the development team has begun implementing the changes warranted by the defect or feature, but the changes are not yet ready to be verified. A workarea is also created for the part changes.
2. When a driver is committed all part changes associated with the driver members are integrated into the release.
3. To create test records, the driver is completed. This action creates one test record for each environment on the release's environment list. The testers on your development team use the test records to accept or reject the results of their tests on the part changes.
4. After all test records have been accepted or abstained, the verification records are moved to the ready state. This indicates that the part changes have been tested in the context of the build and each individual defect or feature is ready to be accepted or rejected by the person who opened it.
5. The defect or feature originator accepts or abstains the verification record to close the defect or feature. The originator can also reject the verification record to move the defect or feature back to working state.

Chapter 5. Working with no component or release processes

To illustrate how to work with objects in a release that does not follow a tracking process or component processes, this chapter follows an example of a programming team that is developing the control systems for a robot. They are working in a family called robot.

Instructions for performing the task are given for both the graphical user interface (GUI) and the command line interface (Command).

This chapter illustrates two scenarios: working in serial development and working in concurrent development. Working in *serial development* means that after you check out a part, TeamConnection locks the part so that it cannot be updated by anyone else. Compare this to *concurrent development*, in which more than one person can simultaneously update the same part.

The following table directs you to the scenario you need:

For this scenario,	**Go to this page.**
Working in serial development	49
Working in concurrent development	68

Working in serial development

Alex is one of the programmers working on the robot application within a release called robot_control. The release does not follow a tracking process, and the release supports serial development. Even though the release does not follow a tracking process, defects are opened when problems are found.

This example assumes that the parts that Alex will work with have already been created in the release, and the build tree has been established. The build tree shows the hierarchy of objects that take part in the build of an application. It identifies parts as inputs, outputs, and dependencies of a build. For more information about build trees, see “Working with a build tree” on page 121 or “Creating the build tree for the application” on page 165.

This example also assumes that the family named robot has been defined by the TeamConnection administrator. Because Alex accesses information in several releases, he has not defined the release named robot_control. Therefore, he must explicitly identify the release when performing TeamConnection actions, but not the family.

A fellow team member, Carol, has discovered that the robot’s aperture is not working correctly. To address this problem, she opens a defect. To fix the problem, Alex needs to make some modifications to the parts in this release. This fix will require the tasks noted in the following table:

For information about this task,	**Go to this page.**
Accepting the defect	50
Creating a workarea	51

For information about this task,	Go to this page.
Checking out a part	53
Searching for a part	54
Checking in a part	55
Verifying and testing part updates	57
Freezing the workarea	62
Refreshing the workarea	63
Building the application	65
Integrating the workarea	66
Closing the defect	67

Accepting a defect

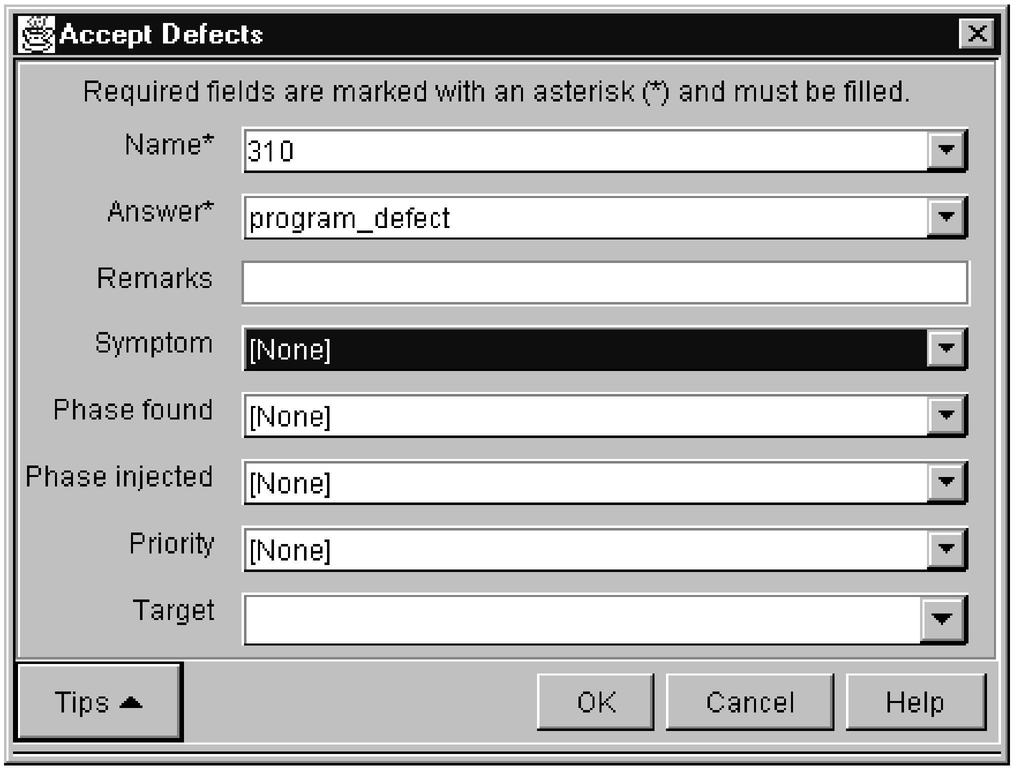
Alex is notified via electronic mail that defect 310 has been opened against the robot component. After some research, he agrees that there is a problem with the aperture of the robot's on-board camera, so he accepts the defect. Alex does one of the following:

GUI

From the GUI, he:

1. Selects **Defects → Accept** from the Actions menu on the Home Page. The Accept Defects window appears.

   Note: The Accept Defects window in this example may be different than one you may see based on your environment. Configurable fields may or may not be shown depending on any configurable fields set by you or your administrator,
2. Types `310` in the **Defects** field and selects **program_defect** from the **Answer** list.
3. Selects **OK**.


Accept Defects

Required fields are marked with an asterisk (*) and must be filled.

Name*	310
Answer*	program_defect
Remarks	
Symptom	[None]
Phase found	[None]
Phase injected	[None]
Priority	[None]
Target	

Figure 13. Accept Defects window

Command

From a command line, he issues the following command:

```
teamc defect -accept 310 -answer program_defect
```

Result

The defect goes to the working state.

Creating a workarea

Because the component is not following a design, size, and review process, Alex needs to manually create a workarea in which to modify and build his parts. (If the component follows a design, size, and review process, a workarea is automatically created when the defect moves to the working state, provided that sizing records have been accepted for the defect.)

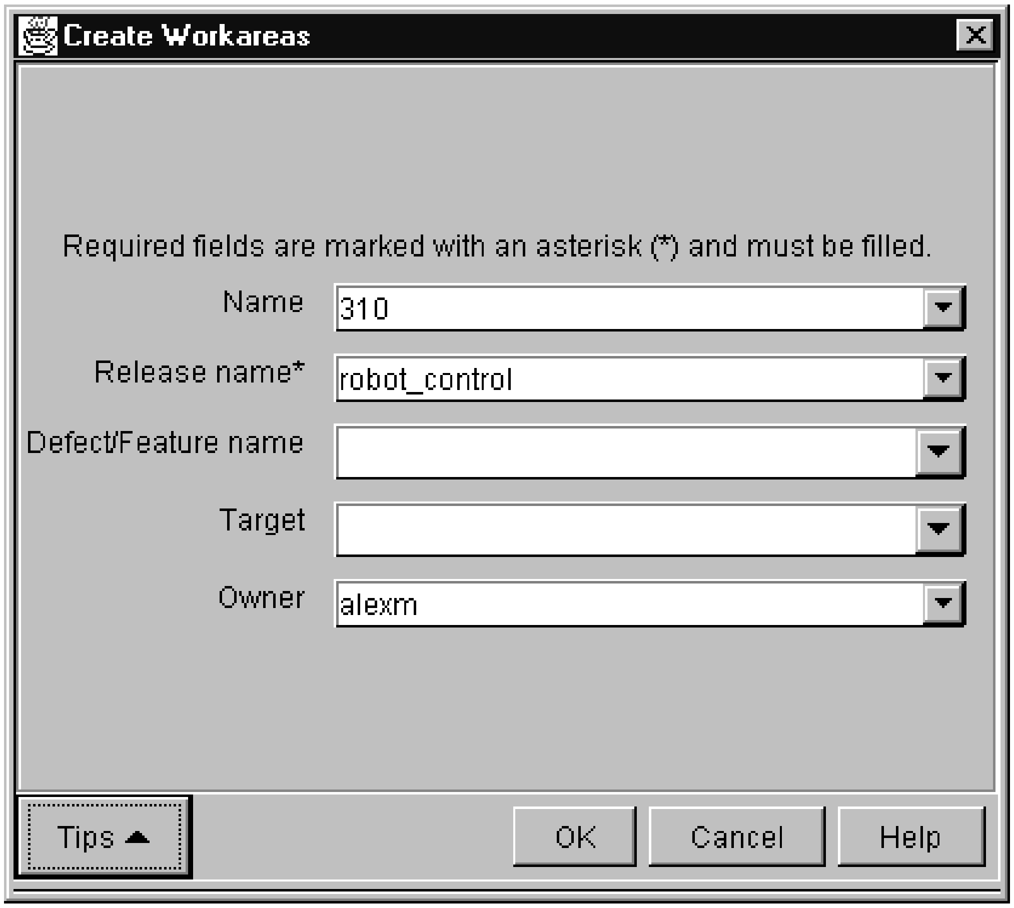
Before Alex checks out any parts, he creates a workarea that will contain the latest view of the parts in the release by doing one of the following:

GUI

From the GUI, he:

1. Selects **Workareas → Create** from the Actions menu on the Home Page.
2. Types `310` in the **Workareas** field and `robot_control` in the **Releases** field and selects **OK.**.

Note: 310 is the name of the defect that was opened for the problem, so this is how Alex wants to identify the work area.



Required fields are marked with an asterisk (*) and must be filled.

Name	310
Release name*	robot_control
Defect/Feature name	
Target	
Owner	alexm

Figure 14. Create Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -create -name 310 -release robot_control
```

Result

TeamConnection creates a workarea named 310 associated with release robot_control. The following parts are currently available in the latest view of release robot_control:

```
brain.c             leg.c
brain.obj           leg.obj
brain.exe           foot.c
arm.c               foot.obj
arm.obj             optics.c
hand.c              optics.obj
hand.obj
```

These parts are also visible in the workarea 310 because the workarea is associated with the release upon creation, and it contains the latest view of the entire release.

Checking out a part

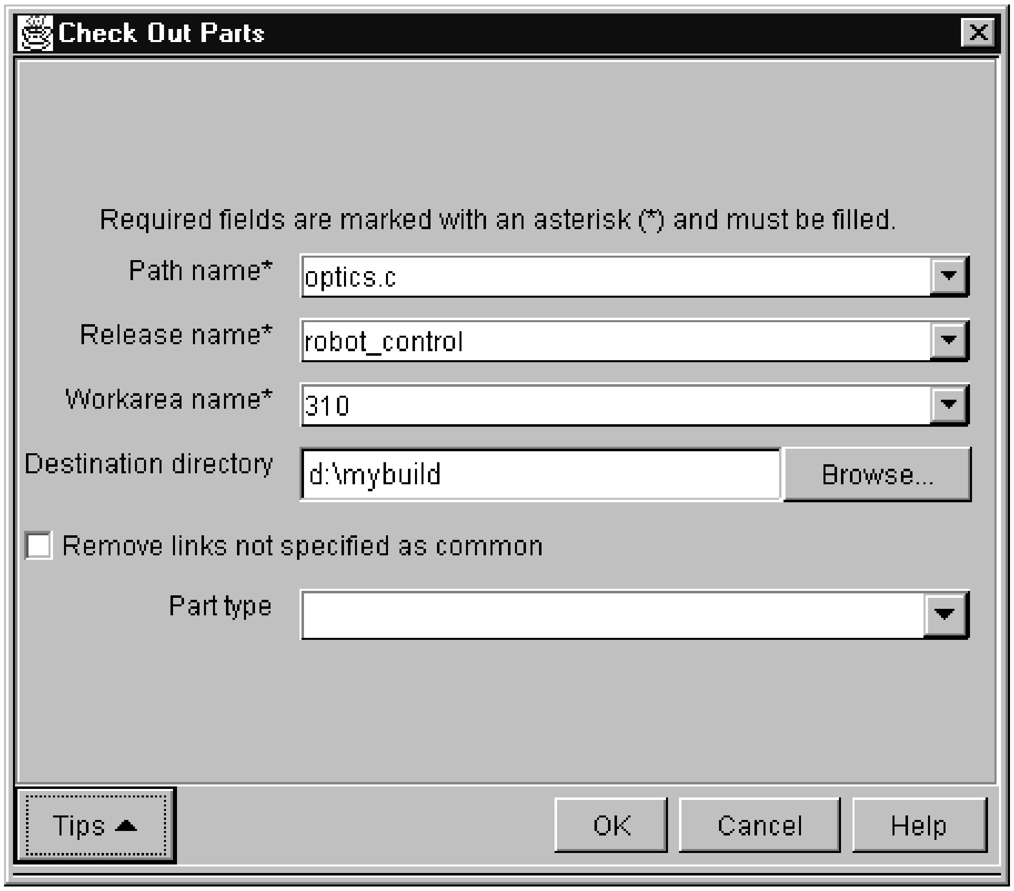
Alex wants to update a subroutine within optics.c, which controls the aperture of the robot’s on-board camera. He checks the part out to start the modifications.

Because Alex knows the exact name of the part, he does one of the following:

GUI

From the GUI, he:

1. Selects **Parts → Check out** from the Actions menu on the Home Page.
2. Types the following:
   - `optics.c` in the **Path names** field
   - `robot_control` in the **Release** field
   - `310` in the **Workarea** field
3. Selects **OK.**



Required fields are marked with an asterisk (*) and must be filled.

Path name* optics.c

Release name* robot_control

Workarea name* 310

Destination directory d:\mybuild

☐ Remove links not specified as common

Part type

Figure 15. Check Out Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the part optics.c is checked out from TeamConnection and placed in the directory specified on the Environment page of the Settings notebook of Alex's TeamConnection client. The part, optics.c, is locked. No other user can update the part until Alex integrates his workarea with the release.

Searching for a part

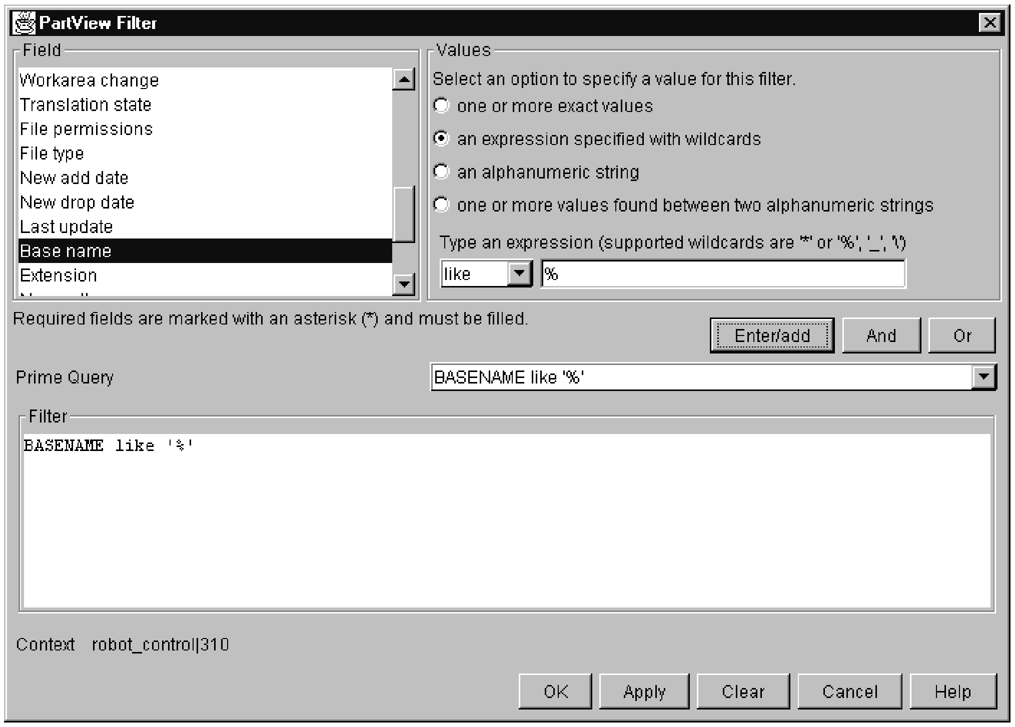
Because Alex knows exactly what part he wants to check out, he specifies the name of the part. If he does not know the name, Alex can use the Parts Filter window or the report command to search for the name. He can do one of the following:

GUI

From the GUI, he:

1. Selects **Common Tasks** → **Create a filter** → **Parts** → **Parts** from the Tasks menu on the Home Page.

   He does not select the **PartFull** choice because he wants to limit his search to a particular release and workarea. He uses **PartFull** when he wants to search for parts across releases, components, or workareas.
2. Does the the following to create the Parts Filter:
   - Selects the **Context** field, types `robot_control` in the **Release Name** field, and the selects the Enter/Add push button
   - Selects the **Limit to Workarea** combo box, types `310` in the **Name** field, and then selects the Enter/Add push button
   - Selects the **Base Name** field, selects the ″an expression specified with wildcards″ radio button, types `%` in the field next to the Like combo box, and then selects the Enter/Add push button. When he is done, the filter looks like Figure 16 on page 55.
   - Selects **OK** pushbutton. A window appears with the parts displayed.
3. Selects the ″Add current filter to my reports″ push button.


PartView Filter

Field
- Workarea change
- Translation state
- File permissions
- File type
- New add date
- New drop date
- Last update
- Base name
- Extension

Values
Select an option to specify a value for this filter.
- one or more exact values
- [✓ an expression specified with wildcards]
- an alphanumeric string
- one or more values found between two alphanumeric strings

Type an expression (supported wildcards are '*' or '%', '_', '\')
like %

Required fields are marked with an asterisk (*) and must be filled.

Prime Query: BASENAME like '%'

Filter
```
BASENAME like '%'
```

Context robot_control|310

Figure 16. Part Filter window

Alex does this because he realizes that he is going to use this query many times, so he wants to add the query to My Reports.

4. Places the mouse pointer over the part name optics.c and presses mouse button 2 to display the pop-up menu.
5. Selects **Check out**. The Check Out Parts window appears with most of the required fields pre-filled. Alex types the name of the destination directory in **Destination directory** field.
6. Selects **OK** to check out the part.

Command

From a command line, he issues the following command:

```
teamc report -view partView -where "baseName like '%.c'"  -release robot_control
  -workArea 310
```

This command returns a list of all the parts that match the query. After Alex determines which part he wants to check out, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the part optics.c is checked out from TeamConnection and placed in the appropriate directory. The part optics.c is locked. No other user can update the part until Alex integrates the workarea with the release.

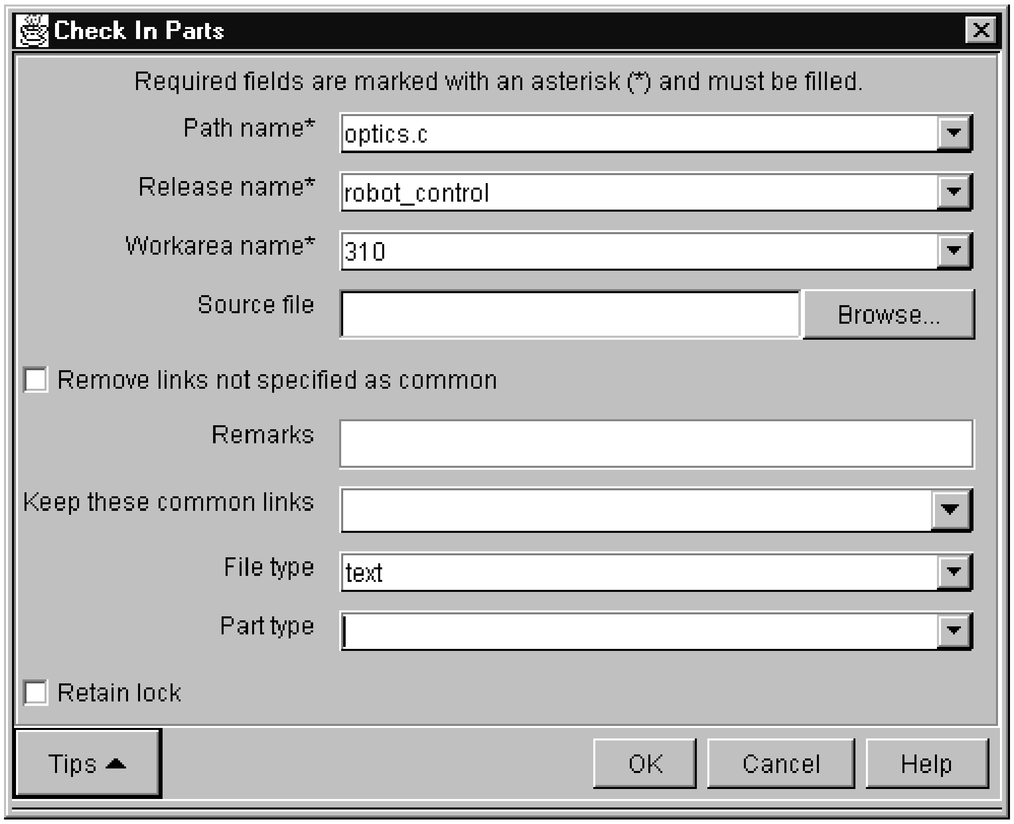
Checking in a part

Alex edits the part, making the modifications he thinks necessary. Now, he wants to test the modifications. First, he checks the changed part back into his workarea.

GUI

From the GUI, he:

1. Selects **Part → Check in** from the Actions menu.
2. Types the following in the Check In Parts window, and then selects **OK**:
   - `optics.c` in the **Path name** field
   - `robot_control` in the **Release name** field
   - `310` in the **Workarea name** field



Check In Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name* optics.c

Release name* robot_control

Workarea name* 310

Source file

Remove links not specified as common

Remarks

Keep these common links

File type text

Part type

Retain lock

Figure 17. Check In Parts window

Note: Alex follows these steps because he knows the exact name of the part that he is checking in. If he does not know the name, or if he is checking in many parts, he can instead create a filter that displays his checked out part and then save the filter query to **My Reports**.

He then selects the parts that he wants to check in.

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 310
```

Result

At this point, it is important to note that the part is checked in to work area 310 and is visible in workarea 310 only. The change to optics.c is not visible at the release level or to any other workarea. Only the 310 workarea contains the change, which is why Alex must specify the workarea on the check-out command. Because changes to parts are isolated within work areas, the check-out command must specify which workarea to use so that the correct copy of the part is retrieved.

Thus, workarea 310 contains the following parts:

```
brain.c            leg.c
brain.obj          leg.obj
brain.exe          foot.c
arm.c              foot.obj
arm.obj            optics.c (modification 1)
hand.c             optics.obj
hand.obj
```

Workarea 310 continues to contain the unchanged parts from the requested release view, but now the workarea is overlaid with changes local to the workarea — optics.c in this case. Alex has his own copy of the application that he can modify without impacting other developers. Alex has checked in optics.c; however, the modified part remains locked until the workarea is integrated with the release.

Verifying and testing part updates

Alex now requests a build of brain.exe, the high-level program for the robot control application.

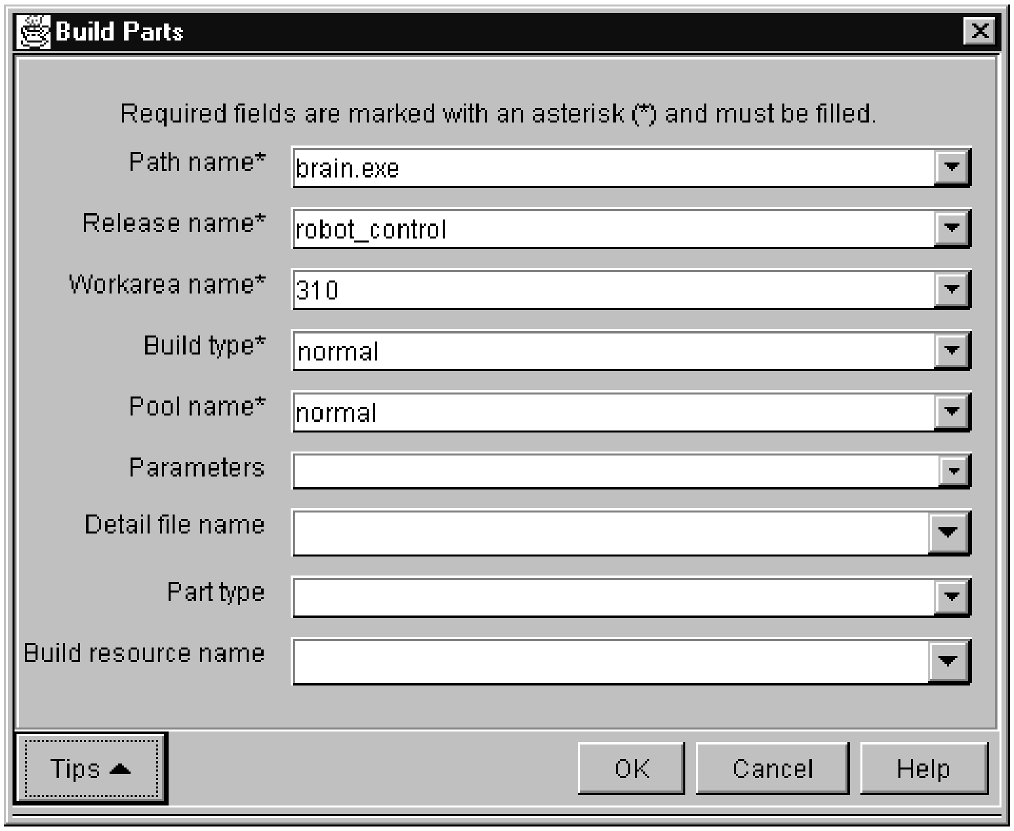
GUI

From the GUI, he:

1. Selects **Parts** → **Build** from the Actions pull-down menu. The Build Parts window appears.
2. Types the following, and then selects **OK** to start the build:
   - `brain.exe` in the **Path name** field
   - `robot_control` in the **Release** field
   - `310` in the **Workarea** field
   - `normal` in the **Pool** field

   The **Pool** field tells TeamConnection which set of build agents will handle this build. Alex got the name of the pool from his build administrator.

   Alex could have selected brain.exe from a list of parts on the Parts window, and then selected **Build** from the Selected menu. This action would have placed some information in the fields, such as the path name and release name.


Build Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name*	brain.exe
Release name*	robot_control
Workarea name*	310
Build type*	normal
Pool name*	normal
Parameters	
Detail file name	
Part type	
Build resource name	

Figure 18. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 310 -pool normal
```

Result

TeamConnection determines the parts that are needed for the build from the set of all the part versions that are currently visible from workarea 310. The following part versions are selected for build:

```
brain.c            leg.c
brain.obj          leg.obj
brain.exe          foot.c
arm.c              foot.obj
arm.obj            optics.c (modification 1)
hand.c             optics.obj
hand.obj
```

After the build is complete, TeamConnection stores the resulting outputs of the build in the workarea `310`. After the build, the workarea contains these parts:

```
brain.c                              leg.c
brain.obj                            leg.obj
brain.exe (contains modification 1)  foot.c
arm.c                                foot.obj
arm.obj                              optics.c (modification 1)
hand.c                               optics.obj (modification 1)
hand.obj
```

Note: For a detailed build example, see “Chapter 14. Building an application: an example” on page 163.

Extracting a part

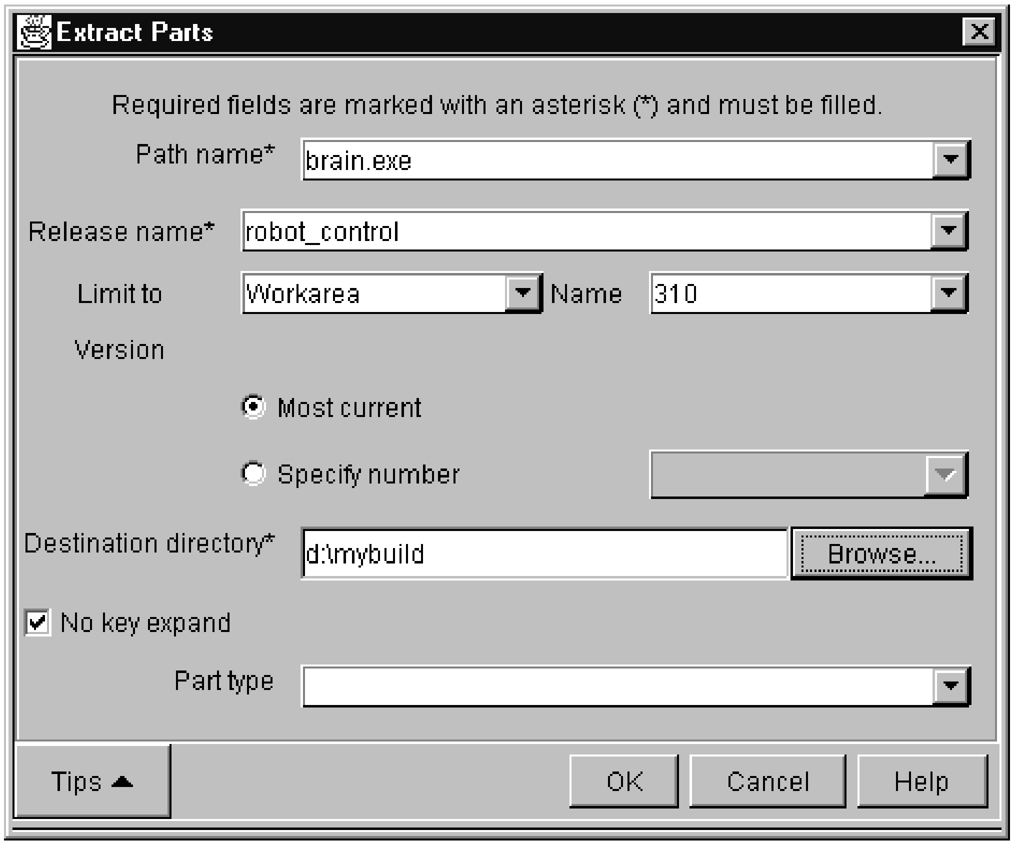
Next, Alex tests his modifications in the robot prototype in his office. He extracts the executable part from the workarea 310.

GUI

From the GUI, he:

1. Selects **Part → Extract** from the Actions menu.
2. Types the following in the Extract Parts window, and then selects **OK**:
   - `brain.exe` in the **Path name** field
   - `robot_control` in the **Release name** field
   - Selects Limit to Workarea and types `310` in the **Name** field

     Alex does this because he wants to extract the .exe part that is in his workarea. If he omits the workarea, he gets the latest committed version of the .exe part from the release.


Extract Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name* brain.exe

Release name* robot_control

Limit to Workarea Name 310

Version

(●) Most current

( ) Specify number

Destination directory* d:\mybuild

[✓] No key expand

Part type

Figure 19. Extract Parts window

Command

From a command line, he issues the following command:

```
teamc part -extract brain.exe -release robot_control -workarea 310
```

Result

This action places a copy of the part brain.exe in the current directory.

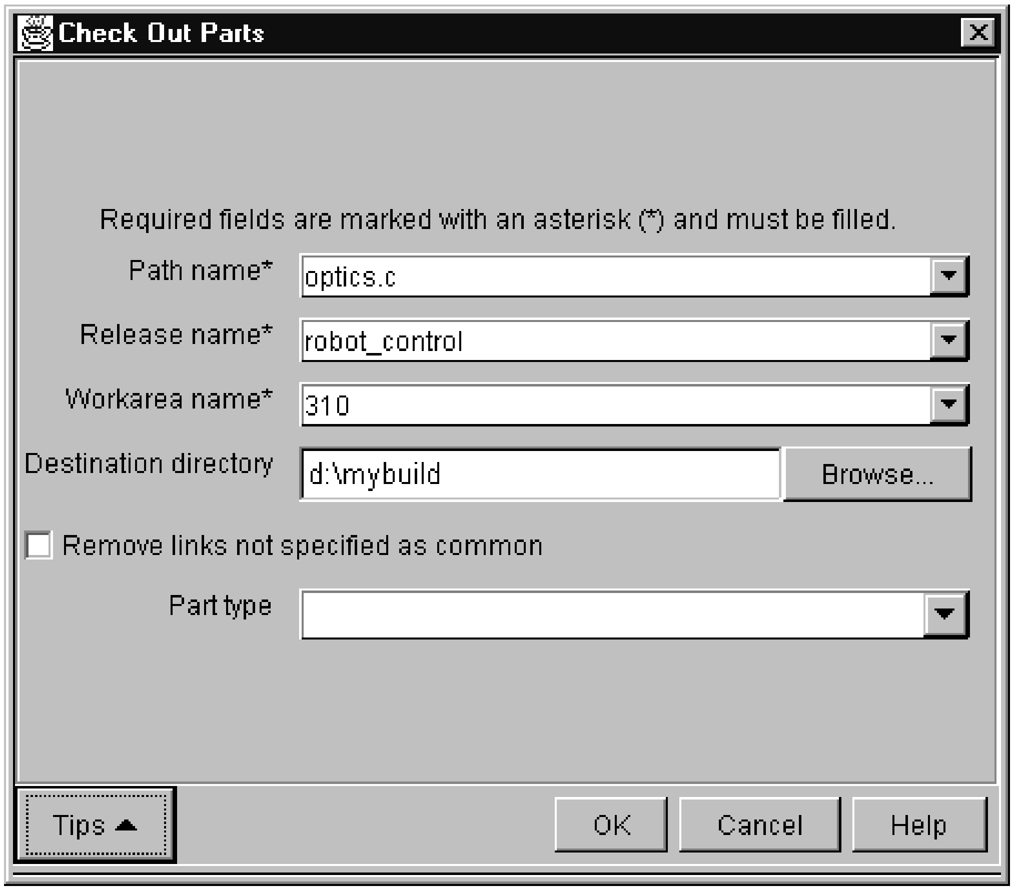
Checking out the part one more time

Alex then downloads brain.exe to his robot, runs his test, and determines that the modification did not work: the robot slams into the wall. However, Alex thinks he knows what the problem is, so he needs optics.c for further modifications. First, he checks out the part.

GUI

From the GUI, he:

1. Does one of the following to display the Check Out Parts window:
   - Selects **Part → Check out** from the Actions menu.
   - Selects the entry in **My Reports** that displays the list of parts, and then selects the part.
   - Re-opens the Parts window if it was minimized, and then selects the part.
2. When the Check Out Parts window appears, he types the necessary information and selects **OK.**



Required fields are marked with an asterisk (*) and must be filled.

Path name* optics.c
Release name* robot_control
Workarea name* 310
Destination directory d:\mybuild
☐ Remove links not specified as common
Part type

Figure 20. Check Out Parts

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the previously modified optics.c from workarea 310 is checked out and placed in the current directory.

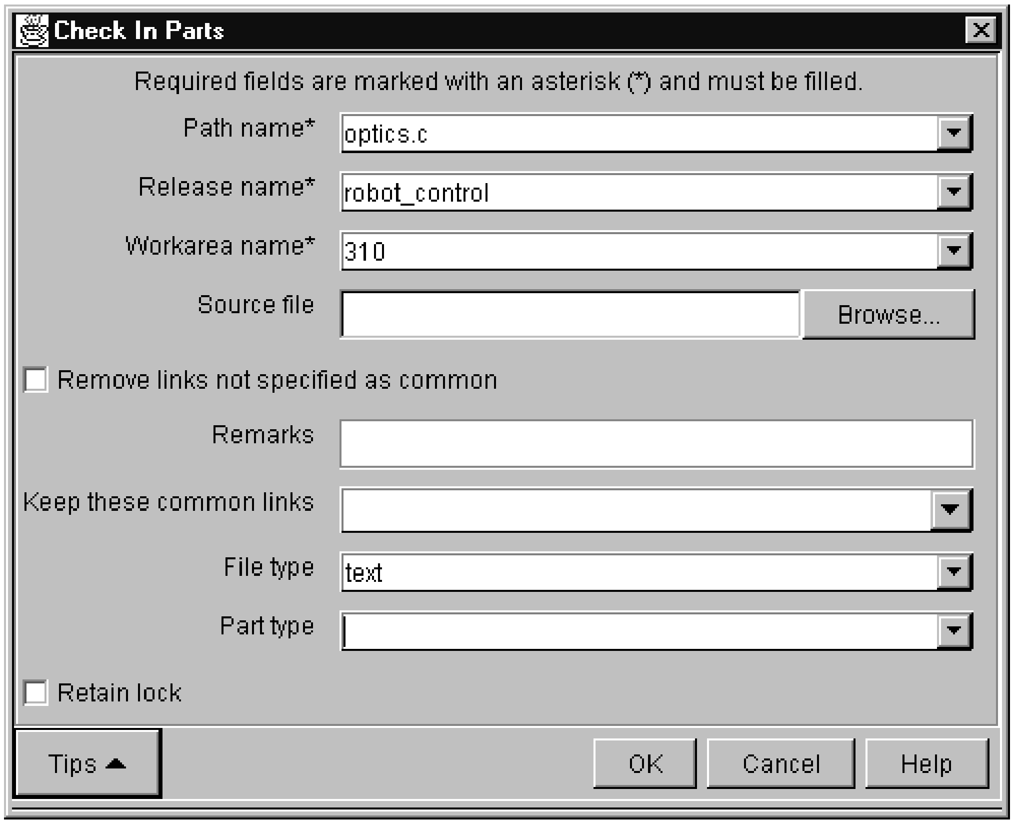
Checking the part back in

Alex makes his modification and checks the part in.

GUI

From the GUI, he:

1. Does one of the following to display the Check In Parts window:
   - Selects **Part → Check in** from the Actions menu.
   - Selects the entry in **My Reports** that displays all the parts he has checked out, and then selects the part.
   - Re-opens the Parts window if it was minimized, and then selects the part.
2. When the Check In Parts window appears, he types the necessary information and selects **OK.**



Required fields are marked with an asterisk (*) and must be filled.

Path name*	optics.c
Release name*	robot_control
Workarea name*	310
Source file	
☐ Remove links not specified as common	
Remarks	
Keep these common links	
File type	text
Part type	
☐ Retain lock	

Figure 21. Check In Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 310
```

Result

Now the workarea contains the following parts:

```
brain.c                               leg.c
brain.obj                             leg.obj
brain.exe (contains modification 1)  foot.c
arm.c                                 foot.obj
arm.obj                               optics.c (modification 2)
hand.c                                optics.obj (modification 1)
hand.obj
```

Because Alex did not specify that he wanted to save a copy of the work area by freezing it, optics.c (modification 1) was overwritten.

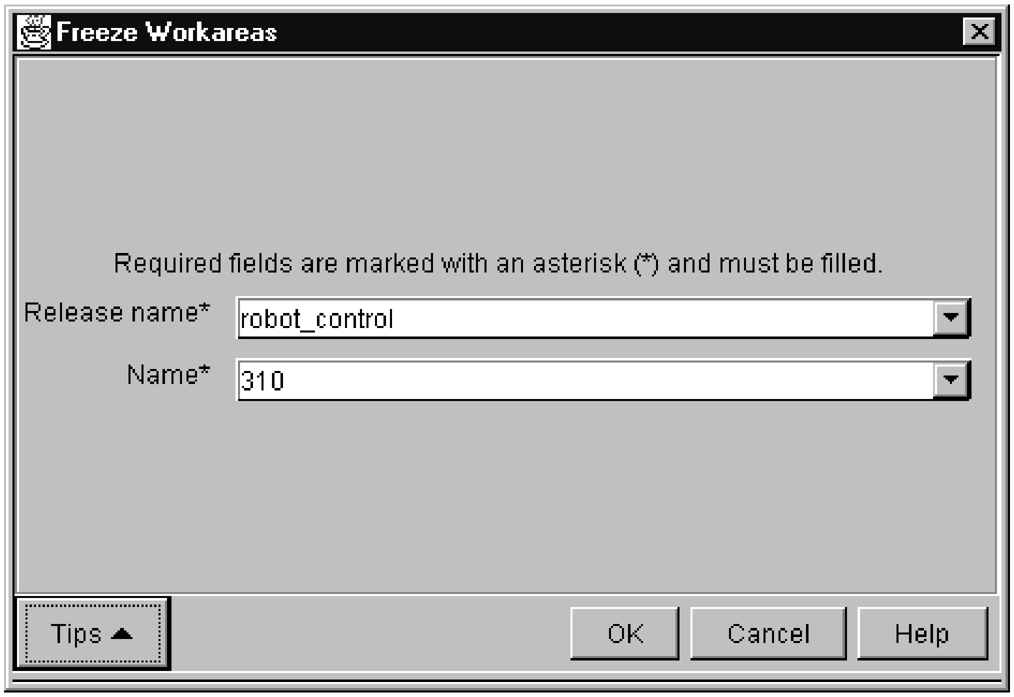
Freezing the workarea

Alex builds the application again, extracts the executable part, and runs his test. This time, everything works, and the robot successfully finds its way to the snack machine down the hall without hitting anything. Alex is very pleased, but he notices an unrelated problem in the robot's autofocus system. Before Alex begins repairing the autofocus subroutine, he wants to save a copy of the application as it exists now in his workarea. So, Alex does one of the following to freeze the workarea:

GUI

From the GUI, he:

1. Displays the Freeze Workareas window in one of the following ways:
   - Selects **Workarea → Freeze** on the Actions menu.
   - Selects `310` from the list of workareas on the Workareas window, then selects **Freeze** from the Selected pull-down menu.
2. Types `robot_control` in the **Release name** field and `310` in the **Name** field if the data is not already there.
3. Selects **OK**.


Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 310

Figure 22. Freeze Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -freeze 310 -release robot_control
```

Result

The freeze command saves the workarea 310. Thus, TeamConnection takes a snapshot of the workarea, with all its parts and their visible versions, and saves it. Alex can come back to this stage of development in the workarea if he wants. Note, however, that a freeze action does not make the changes visible to the other people working in the release, nor does it unlock the parts.

Refreshing the workarea

Alex finally finishes his work on the robot's optical systems after making three additional attempts at modifying optics.c and rebuilding the application. Alex modified and rebuilt the application a total of five times in the work area. Now, he wants to share his work with the rest of the team. His work area currently contains the following parts:

```
brain.c                              leg.c
brain.obj                            leg.obj
brain.exe (contains modification 5)  foot.c
arm.c                                foot.obj
arm.obj                              optics.c (modification 5)
hand.c                               optics.obj (modification 5)
hand.obj
```

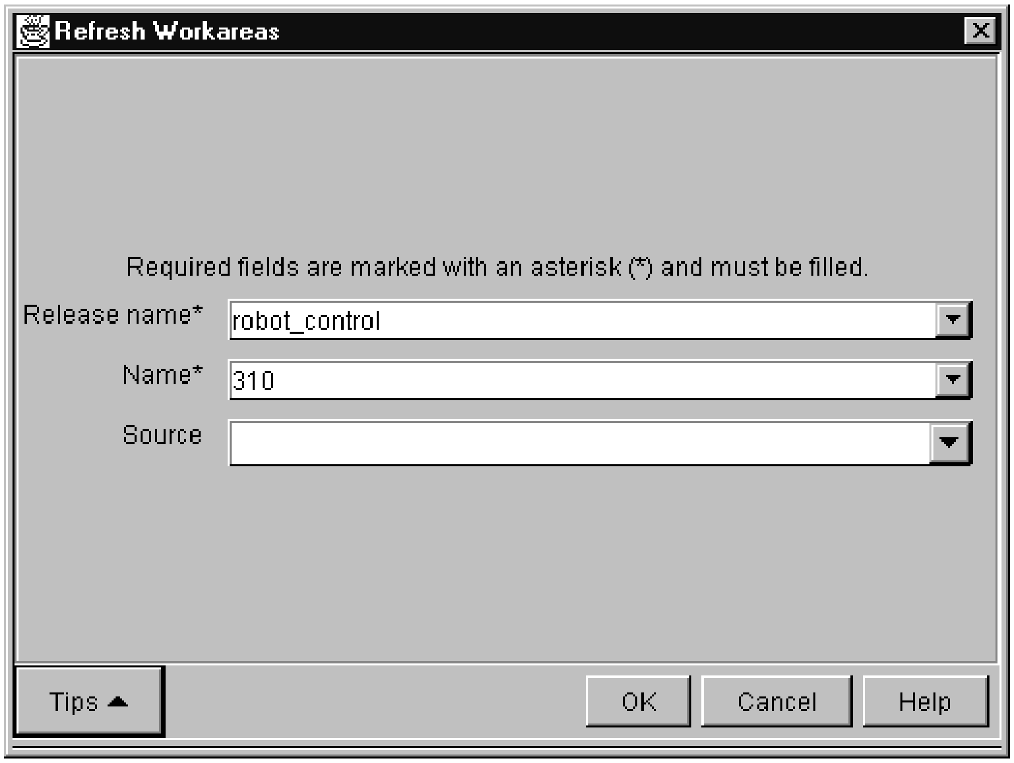
While Alex worked in his workarea, other members of the team were working on their own modifications. Some of these modifications have been integrated with the release, so the copy of the release that Alex has is probably stale. If he were to integrate his changes at this time with the release, he might cause the application to break.

Alex first refreshes his workarea with parts from the release by doing one of the following:

GUI

From the GUI, he:

1. Selects **Workarea → Refresh** from the Actions menu.
2. Types `robot_control` in the **Release name** field and `310` in the **Name** field.
   Alex wants to refresh from the release, so he does not specify a source.
3. Selects **OK**.


Refresh Workareas

Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 310

Source

Figure 23. Refresh Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -refresh 310 -release robot_control
```

Result

This action updates workarea 310 with any changes from the release, and it also freezes workarea 310, if it is not already frozen. Now Alex’s work area contains the following versions of parts:

```
brain.c  (Jenny's modification)                  leg.c
brain.obj (from Alex's build after refresh)      leg.obj
brain.exe (contains modification 5)              foot.c
```

```
arm.c                                                 foot.obj
arm.obj                                               optics.c (modification 5)
hand.c   (Joy's and Ken's modification)       optics.obj (modification 5)
hand.obj (from Alex's build after refresh)
```

None of the objects that Alex modified and none of the objects built as a result of Alex's modifications is overwritten by the refresh.

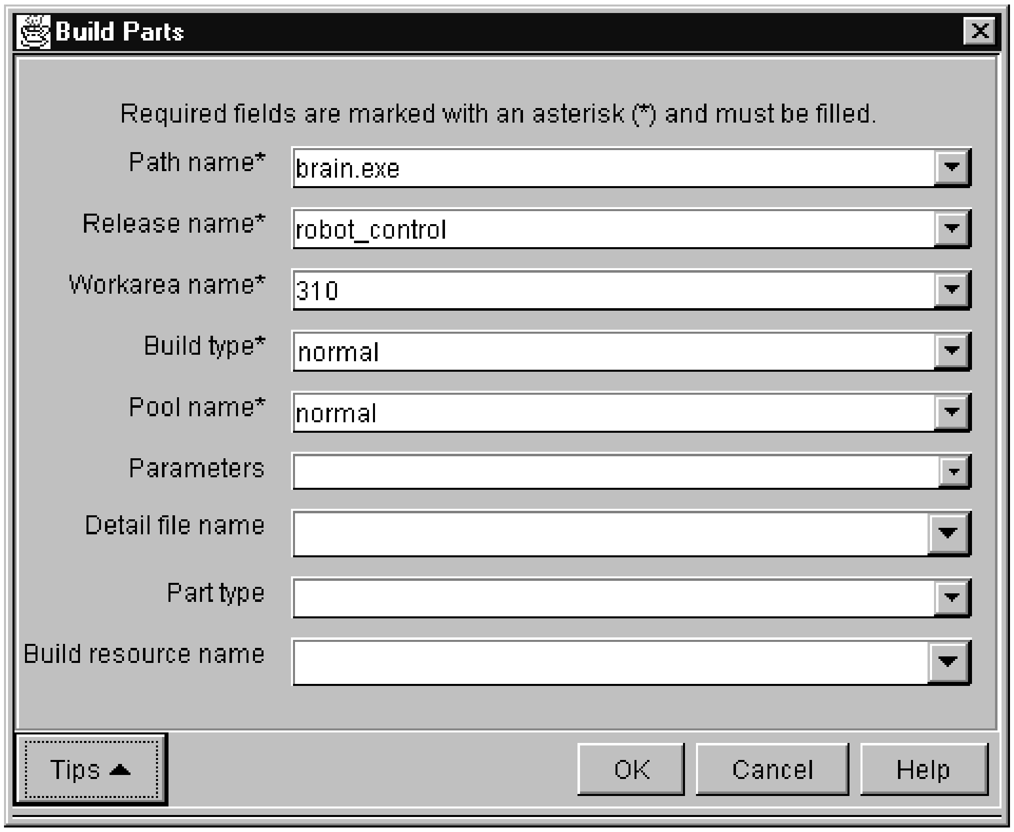
Building the application

Alex again builds the application brain.exe within his work area to determine whether his changes integrate with Jenny's, Joy's, and Ken's modifications.

GUI

Alex has a Parts window open with a list of all the parts that exist in workarea 310. He highlights the part brain.exe, and then does the following:

1. Selects **Build** from the Selected menu.
2. Types `normal` in the **Pool** field. The other required fields have the correct information.
3. Selects **OK** to start the build.



Build Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name*	brain.exe
Release name*	robot_control
Workarea name*	310
Build type*	normal
Pool name*	normal
Parameters	
Detail file name	
Part type	
Build resource name	

Figure 24. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 310 -pool normal
```

Result

Fortunately, nothing breaks, so Alex is ready to integrate his changes with the release.

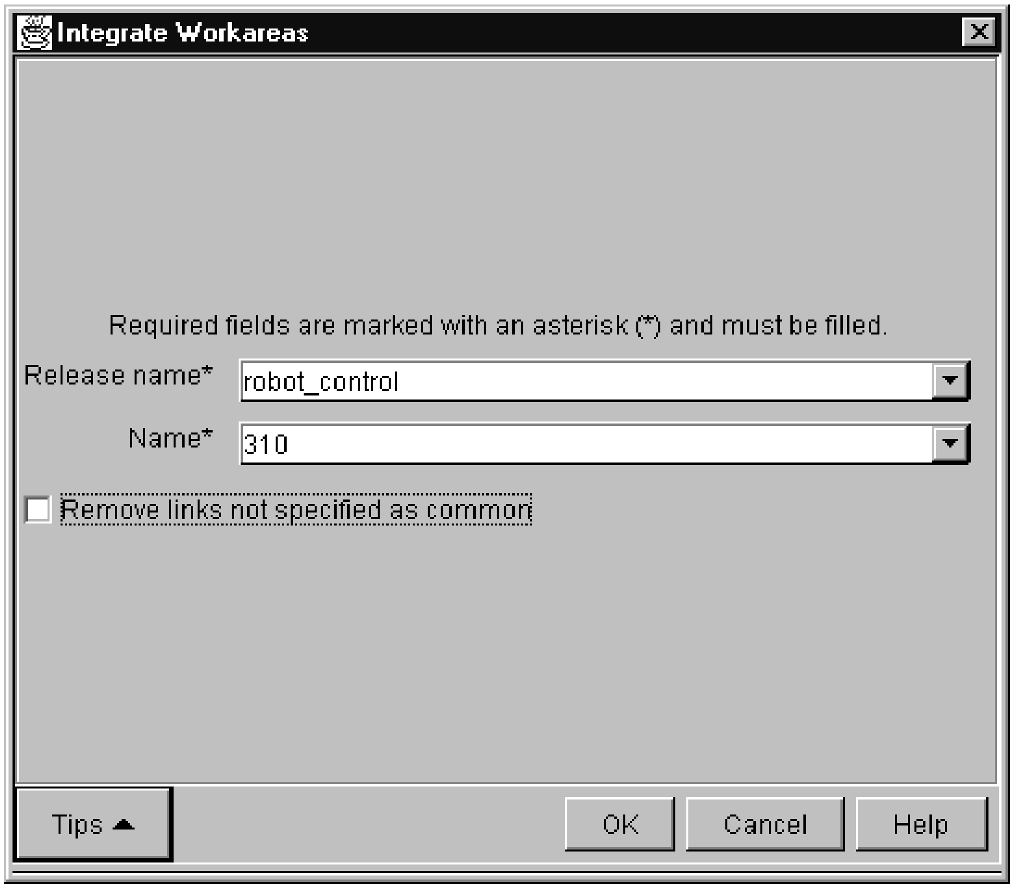
Integrating the workarea

To integrate his changes with the release, Alex must integrate the workarea he has been using with the release. This will make the workarea visible to all the users in the release. He does one of the following:

GUI

From the GUI, he:

1. Selects **Workareas → Integrate** from the Actions menu. The Integrate Workareas window appears.
2. Types `robot_control` in the **Release name** field and `310` in the **Name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 310

[ ] Remove links not specified as common

Figure 25. Integrate Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -integrate 310 -release robot_control
```

Result

TeamConnection first determines that Alex’s changes were built against the latest version of the release. Then TeamConnection makes Alex’s changes visible at the release level so that the other team members can see and use them. The following part versions are now visible from the release:

```
brain.c  (Jenny's modification)
brain.obj (from Jenny's build)
brain.exe (from Alex's build)
arm.c
arm.obj
hand.c   (Joy's modification, Ken's modification)
hand.obj (from Ken's build)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Alex's modification 5)
optics.obj (from Alex's build)
```

TeamConnection also makes a copy of the release before integrating Alex’s changes. If something doesn’t work, the users or the administrator can go back to the release prior to Alex’s integration. The part, optics.c, is now unlocked in the release. The workarea is now in the complete state and can no longer be used.

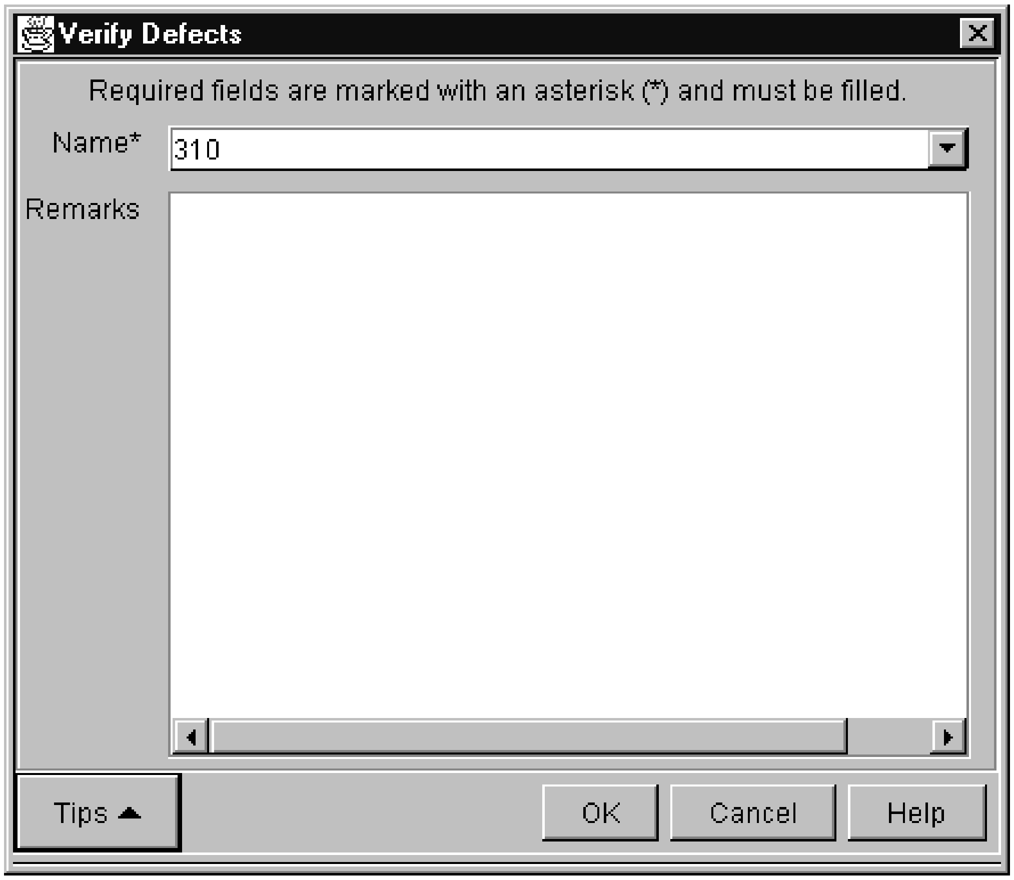
Closing a defect

Now that Alex is finished making changes to fix the problem reported in defect 310, he is ready to close the defect. He does one of the following:

GUI

From the GUI, he:

1. Selects **Defects → Verify** from the Actions menu on the Home Page. The Verify Defects window appears.
2. Types `310` in the **Name** field.
3. Selects **OK**.


Verify Defects

Required fields are marked with an asterisk (*) and must be filled.

Name* 310

Remarks

Figure 26. Verify Defects window

Command

From a command line, he issues the following command:

```
teamc defect -verify 310 -release robot_control
```

Result

Because the component does not include the verifyDefect subprocess in its process, the defect moves directly to the closed state.

Working in concurrent development

The previous section discussed working in a serial development environment. While Alex had optics.c in his workarea, no one else on the team could check out the part. TeamConnection allows you to hold the part until you are sure that it integrates with the rest of the application. Therefore, the lock is not released until the workarea as a whole is integrated with the release.

The scenario changes slightly for concurrent development. In this case, several users can work on the same part at the same time. These users must reconcile their changes as they integrate their workareas with the release.

The following tasks are required:

For information about this task,	Go to this page.
Refreshing the workarea from the driver	69
Integrating the workarea	71
Resolving differences	72

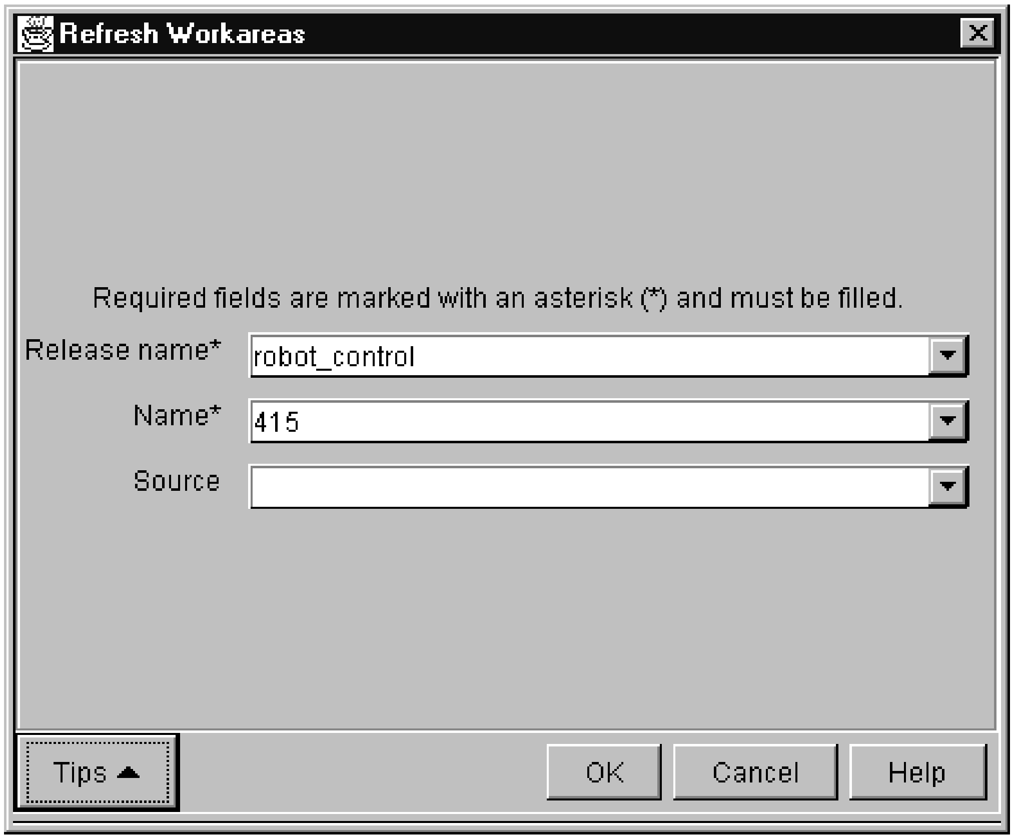
Refreshing the workarea

If Alex and Jenny are working on optics.c at the same time, they must resolve their part differences at some point, because both want to make their changes visible to the release. If Alex and Jenny were not required to do this before committing their workareas, the last developer to commit would always overlay the other’s changes. For this scenario, assume that Jenny finishes her changes first. The first thing she does is refresh her workarea.

GUI

From the GUI, she:

1. Selects **Workareas → Refresh** from the Actions menu.
2. Types `415` in the **Name** field.
3. Types `robot_control` in the **Release name** field.
4. Selects **OK**.


Refresh Workareas

Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 415

Source

Figure 27. Refresh Workareas window

Command

From a command line, she issues the following command:

```
teamc workarea -refresh 415 -release robot_control
```

Result

This command refreshes her workarea with the latest view of the release. Her workarea now contains the following part versions:

```
brain.c  (Jenny's modification 3)
brain.obj (Jenny's modification 3)
brain.exe (has Jenny's brain.c modification 3 and optics.c modification 4)
arm.c
arm.obj
hand.c   (Joy's modification, Ken's modification)
hand.obj (Joy's modification, Ken's modification)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Jenny's modification 4)
optics.obj (Jenny's modification 4)
```

Integrating the workarea

The refresh shows Jenny only the parts integrated with the release. She does not see Alex’s work because he has not integrated his workarea yet. Jenny rebuilds the application, tests it, and decides she is ready to integrate her changes. She does one of the following:

GUI

From the GUI, she:

1. Selects **Workarea** → **Integrate** from the Actions menu.
2. Types `415` in the **Name** field, and `robot_control` in the **Release name** field.
3. Selects **OK**.

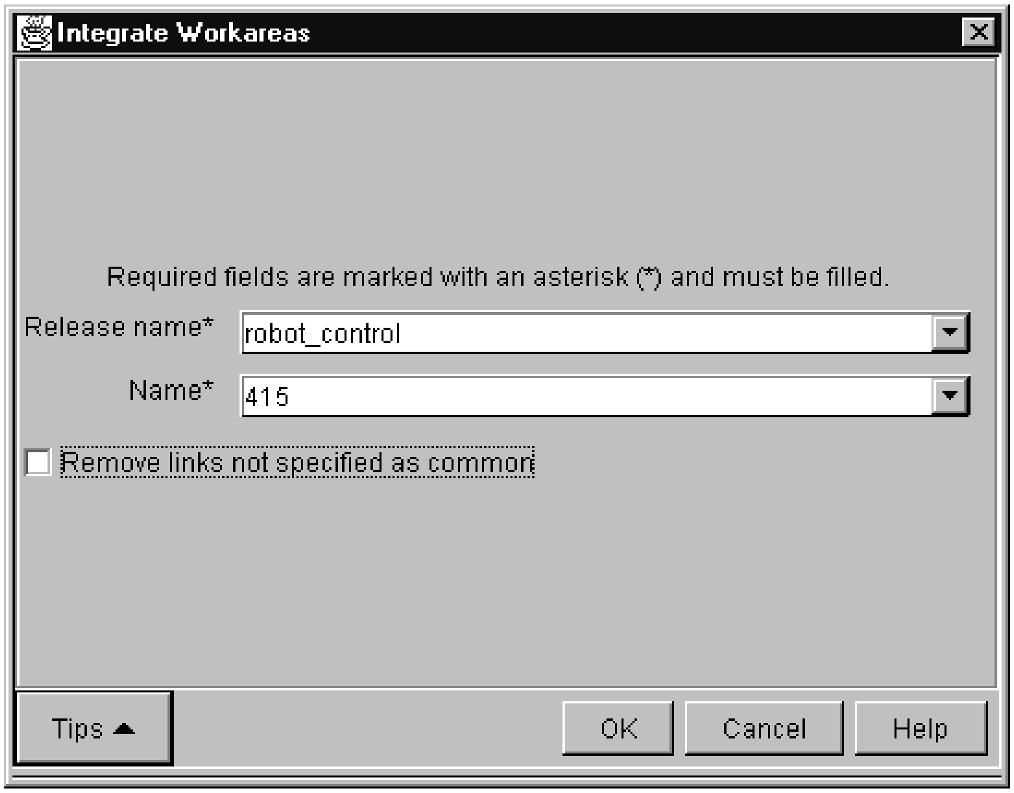

Integrate Workareas
Required fields are marked with an asterisk (*) and must be filled.
Release name*
robot_control
Name*
415
Remove links not specified as common
Tips
OK
Cancel
Help

Figure 28. Integrate Workareas window

Command

From a command line, she issues the following command:

```
teamc workarea -integrate 415 -release robot_control
```

Result

Because Jenny is up-to-date with the latest view of the driver, her changes are integrated after TeamConnection preserves a copy of the previous version of the release.

Reconciling differences

Later, Alex is ready to integrate his modifications. Alex issues a refresh command from the driver, as Jenny did (see page 69 for instructions).

This time, Alex receives a message that collision records were generated, because both he and Jenny have updated the same parts. At this time he does not know which parts collided. TeamConnection refreshes workarea 310 with the exception of the part optics.c, which had the collision. Alex’s workarea shows the following parts:

```
brain.c  (Jenny's modification 3)
brain.obj (Jenny's modification 3)
brain.exe (Contains Alex's modification 5)
arm.c
arm.obj
hand.c   (Joy's modification, Ken's modification)
hand.obj (Joy's modification, Ken's modification)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Alex's modification 5)
optics.obj (Alex's modification 5)
```

Alex can use either the GUI or the command line to reconcile the differences. Four steps are required from the command line:

1. Check out the latest uncommitted version.
2. Extract the latest committed version.
3. Run the merge program against the two parts.
4. Check in the resultant part.

However, on the GUI the reconcile action automatically does the preceding steps for you, which can save you a considerable amount of work if several parts require reconciliation.

GUI

From the GUI, he:

1. Creates a filter that displays the collison records. The Collision Records window appears with optics.c listed as the part having the collision.
2. Highlights the optics.c entry and selects **Reconcile** from the pop-up menu. The Reconcile Collision Record window appears with the required information pre-filled.

   Alex does not have to reconcile every part for which a collision record is created. He can choose either his copy or the copy at the release rather than combining the two. For example, if Alex wants to use his copy of optics.c without merging with the copy at the release level, he selects the reject action (of course, he would not do that without first talking with Jenny). If he wants to use the copy of optics.c at the release level without merging any of his changes into the copy at the release level, he selects the accept action.
3. Because Alex wants to combine the two sets of changes, he selects **Merge** to start the TeamConnection merge program, or any merge program of his choice. Alex merges the changes and then saves and exits from the merge program. TeamConnection checks the resultant part back in as part of this merge step.

   The online help provides information on how to use the merge program.
4. Selects **OK** from the Reconcile Collision Record window.

Command

From a command line, he does the following steps:

- Issues a report command to determine which parts are in conflict:

  ```
  teamc report -view collisionView -workarea 310
  ```

  This report tells him that optics.c is the part that collided and gives the *alternate version ID* of the part that caused the collision. Alex makes note of the alternate version ID, robot_control:2, because he needs to specify that in a later step.
- Extracts a copy of optics.c from the release:

  ```
  teamc part -extract optics.c -release robot_control -relative d:\temp
  ```

  By not specifying a workarea on the part -extract command, Alex ensures that he receives the last committed copy of the part at the release. Also, Alex specifies a *relative* path for the part extract. By specifying the relative directory, he prevents TeamConnection from placing the part in his default directory, where he normally works on checked-out parts. For more information about the -relative flag, refer to the *Commands Reference*
- Checks out his copy of optics.c from his workarea:

  ```
  teamc part -checkout optics.c -release robot_control -workarea 310
  ```

  Because he did not specify a relative path, this part is checked out to his working directory `d:\robot`.
- Uses the merge program to reconcile the two copies of optics.c:

  ```
  tcmerge d:\temp\optics.c  d:\robot\optics.c -out
       d:\robot\optics.c -prime d:\temp\optics.c
  ```

  If Alex decides not to merge the two parts, but instead wants to use his copy of optics.c, he uses the collision -reject command. Or, if he wants to use the copy of optics.c at the release level, he uses the collision -accept command.
- Checks the resultant copy of optics.c into his workarea and builds it against the rest of the system.
- After he is satisfied with the reconciled changes, he lets TeamConnection know that the previously discovered conflict is reconciled. Alex does this by completing the collision record that TeamConnection created when Alex attempted to integrate his copy of optics.c. He does the following:

  ```
  teamc collision -reconcile -path optics.c -release robot_control
     -workarea 310 -altversion robot_control:2
  ```

Result

Alex is now ready to make his changes visible to the release. He can use either the GUI or the command line to integrate the workarea.

He refreshes from the driver again. The integrate is permitted because a completed collision record exists for the conflict between the two versions of optics.c. However, if Ken or Joy had integrated a new version of optics.c while Alex was busy resolving the last collision, Alex’s driver add would fail. He would have to repeat the collision resolution process.

Chapter 6. Working with component and release processes

The previous chapter described how to work with parts when the release does not follow a tracking process. This chapter describes how to work with parts when a tracking process *is* followed and how to use component processes for features and defects.

When tracking is part of the process, users must associate any changes to their parts with the defects or features active for the release. This association is made through a workarea. The workarea is the object that ties a defect or feature with a specific release. When checking out a part, the user must specify the workarea with which the modification is associated. For any release and defect or feature pair, there can be multiple workarea objects.

Aside from their association with a defect or feature, the workareas for a full-tracking process environment are identical to those defined for working in a no-tracking process environment. Workareas maintain a separate view for the user working on the modifications associated with a defect or feature without affecting the release. This view can be integrated with the release at some point. A workarea is implicitly created when a defect or feature is accepted if the managing component follows a design, size, and review process for defects and features and if a sizing record is created. The workarea that TeamConnection creates is based on the sizing record and has the same name as the defect or feature. If sizing records were not created, you must explicitly create the workarea.

As an example of how this all works, suppose that the robot project from the previous chapter is entering system test. The administrator decides to turn on a full-tracking process for the release, such as track_full. This process includes the track, approval, fix, driver, and test subprocesses. The release follows concurrent development, and the component follows a design, size, and review process for both defects and features.

On a weekly basis the project leader, Carol, creates a driver. A *driver* monitors and implements the integration of part changes within a release. These part changes are included in a driver by adding the work areas referenced by the changed parts to the driver as *driver members*.

One of the testers for the robot project discovers that the autofocus mechanism in the robot's eye fails when the robot is placed in front of striped wallpaper. The tester must open a defect against the component optics, which is owned by Carol. Carol verifies that the problem does exist, accepts the defect, and assigns it to Alex. This fix will require the tasks noted in the following table:

For information about this task,	**Go to this page.**
Changing the defect owner	76
Accepting the defect	77
Approving the fix	79
Checking out a part	80
Checking in the changes	82
Freezing the workarea	83
Building the application	84

For information about this task,	Go to this page.
Accepting fix records	86
Adding a driver member	87
Returning the workarea to the fix state	89
Reactivating the fix record	90
Refreshing the workarea	91
Refreshing the driver	92
Building the driver	93
Restricting the driver	94
Integrating the parts	95
Completing the driver	96
Testing the built application	97

Moving through design, size, and review

Because the defect was created against a component that follows the design, size, and review process for defects, Carol must move the defect through this process before the defect can be accepted and parts can be checked out. As the names imply, the process requires that the following be done:

- Design what needs to be done in order to resolve the problem. She must enter design text before the defect can move to the size state.
- Size the amount of work that is required to resolve the problem. At this time, Carol creates a sizing record and specifies robot_control as the release that contains the parts that require changing. If parts in other releases require changing because of the defect, a sizing record is created for each release. A sizing record assures that a workarea is created when the defect is accepted. It identifies the work that is required for and the resources affected by the defect or feature. The owner of the component that is referenced in the sizing record is the owner of the sizing record. The owner is responsible for entering information about the amount of work that is required to implement the feature or resolve the problem.
- Review all design text and sizing records and determine if work should continue on the defect.

Changing defect ownership

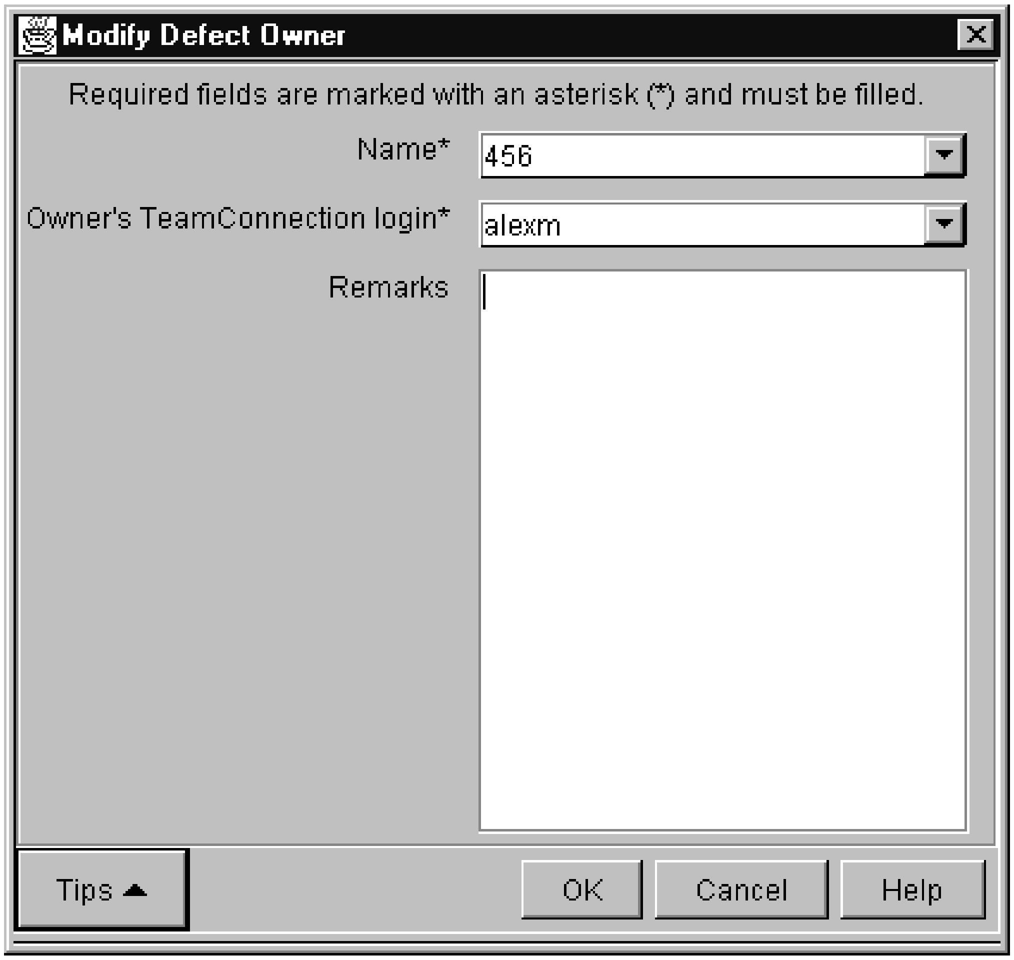
Because Carol is the component owner, she is currently defined as the owner of defect 456. But the problem is in Alex's code, so she wants him to own the defect. To reassign ownership, she does one of the following:

GUI

From the GUI, she:

1. Selects **Defects → Modify → Owner** from the Actions menu on the Home Page. The Modify Defect Owner window appears.
2. Types `456` in the **Name** field and types Alex's user ID, `alexm`, in the **Onwer's TeamConnection Login** field.

3. Selects **OK**.


Modify Defect Owner

Required fields are marked with an asterisk (*) and must be filled.

Name* 456

Owner's TeamConnection login* alexm

Remarks

Figure 29. Modify Defect Owner window

Command

From a command line, she issues the following command:

```
teamc defect -assign 456 -owner alexm
```

Results

Alex is now the owner of defect 456. He is responsible for fixing the problem and moving the defect through its various states.

Accepting a defect

When you accept a defect or feature, you accept the responsibility of resolving it. A defect or feature might require changes in more than one release. If the component includes the design, size, and review process, these releases were identified during the size state, and TeamConnection created a workarea for each identified release. If the component does not include the design, size, and review process, you will need to create a workarea manually.

When the first workarea moves to the complete state, the defect or feature automatically moves to the verify state or closed state.

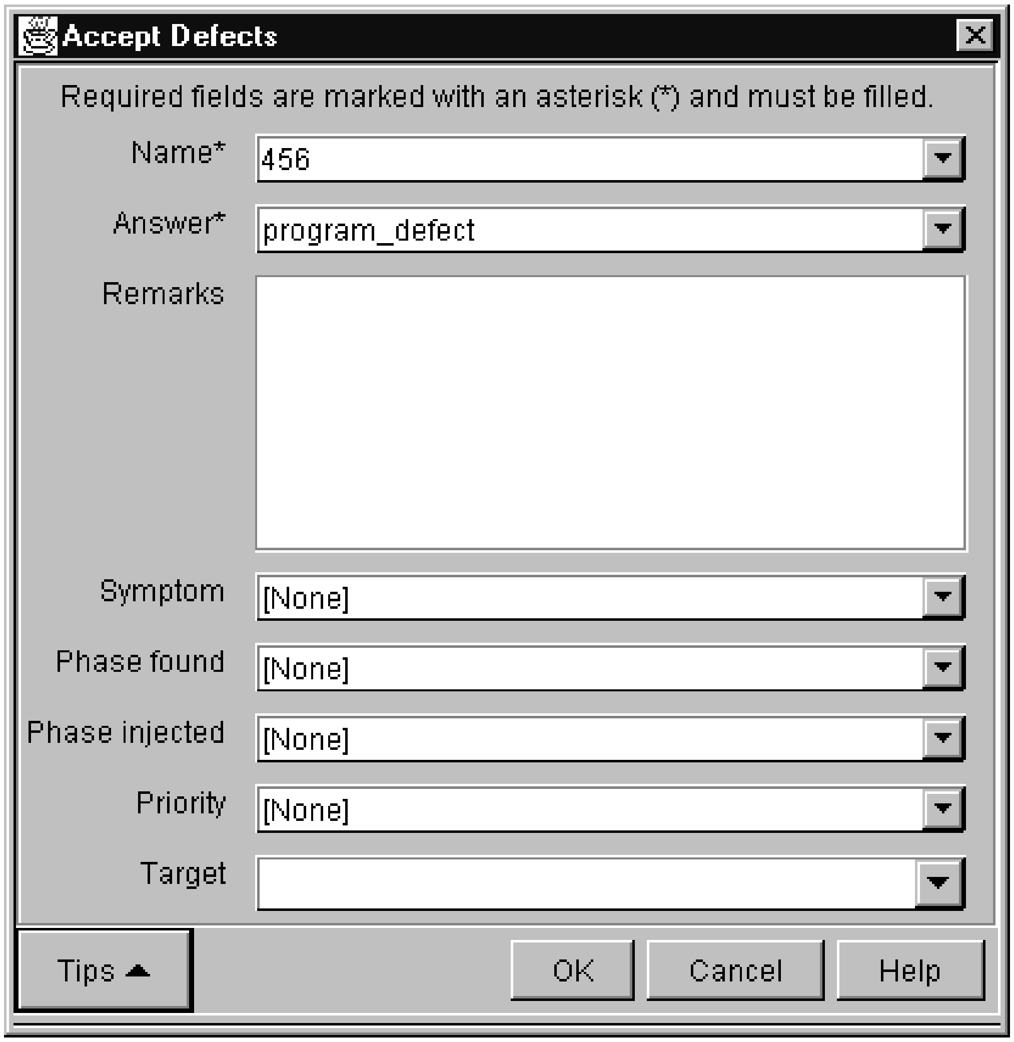
Alex, now the owner of the defect, accepts the defect by doing one of the following:

GUI

From the GUI, he:

1. Selects **Defects → Accept** from the Actions menu on the Home Page. The Accept Defects window appears.

   Note: The Accept Defects window in this example may be different than one you may see based on your environment. Configurable fields may or may not be shown depending on any configurable fields set by you or your administrator,
2. Types `456` in the **Name** field and selects **program_defect** from the **Answer** list.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Name*	456
Answer*	program_defect
Remarks	
Symptom	[None]
Phase found	[None]
Phase injected	[None]
Priority	[None]
Target	

Figure 30. Accept Defects window

Command

From a command line, he issues the following command:

```
teamc defect -accept 456 -answer program_defect
```

Results

Defect 456 moves to working state, and TeamConnection creates a workarea called 456. The workarea is associated with the release specified on the sizing record, which in this example is robot_control. When the workarea is created, a fix record is also created based on the sizing record. Because the approval subprocess is included in the release's process, the workarea is created in the approve state and the fix record is created in the notReady state.

Just as with a workarea that is explicitly created, the defect workarea contains a view of the current versions visible to the release. In this case, the contents of the workarea are:

```
brain.c                    leg.c
brain.obj                  leg.obj
brain.exe                  foot.c
arm.c                      foot.obj
arm.obj                    optics.c
hand.c                     optics.obj
hand.obj
```

Approving the fix

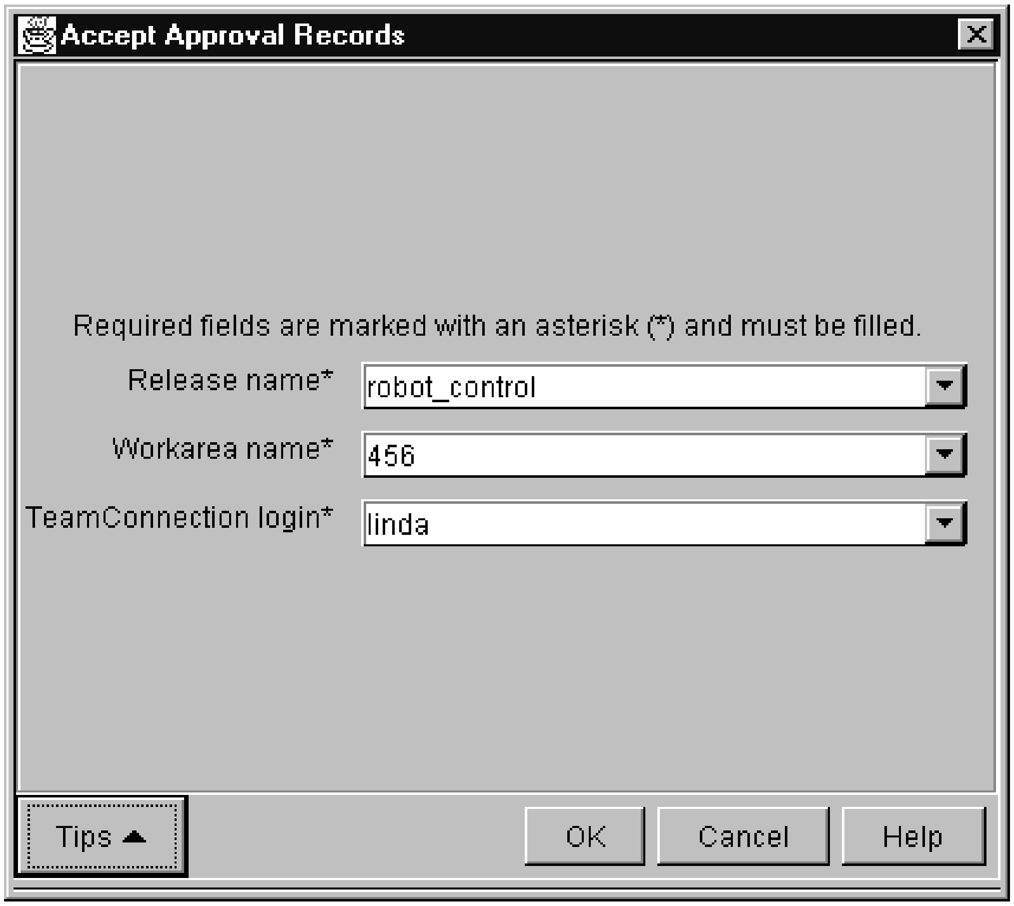
Because the full-tracking process includes the approval subprocess, each person identified on the approval list must approve the proposed changes before Alex can begin work on the defect.

Linda and Sam are both listed as approvers. They have been notified by TeamConnection that they have approval records. After reviewing the defect, they do one of the following to indicate their approval:

GUI

From the GUI, they:

1. Select **Records → Approval records → Accept** from the Actions menu.
2. Type `456` in the **Workarea name** field and `robot_control` in the **Release name** field.
3. Select **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Workarea name* 456

TeamConnection login* linda

Figure 31. Accept Approval Records window

Command

From a command line, they both issue the following command for the approval record that they have:

```
teamc approval -accept -workarea 456 -release robot_control
```

Results

After both Linda and Sam accept the approval records, TeamConnection moves the workarea to the fix state.

Checking out a part

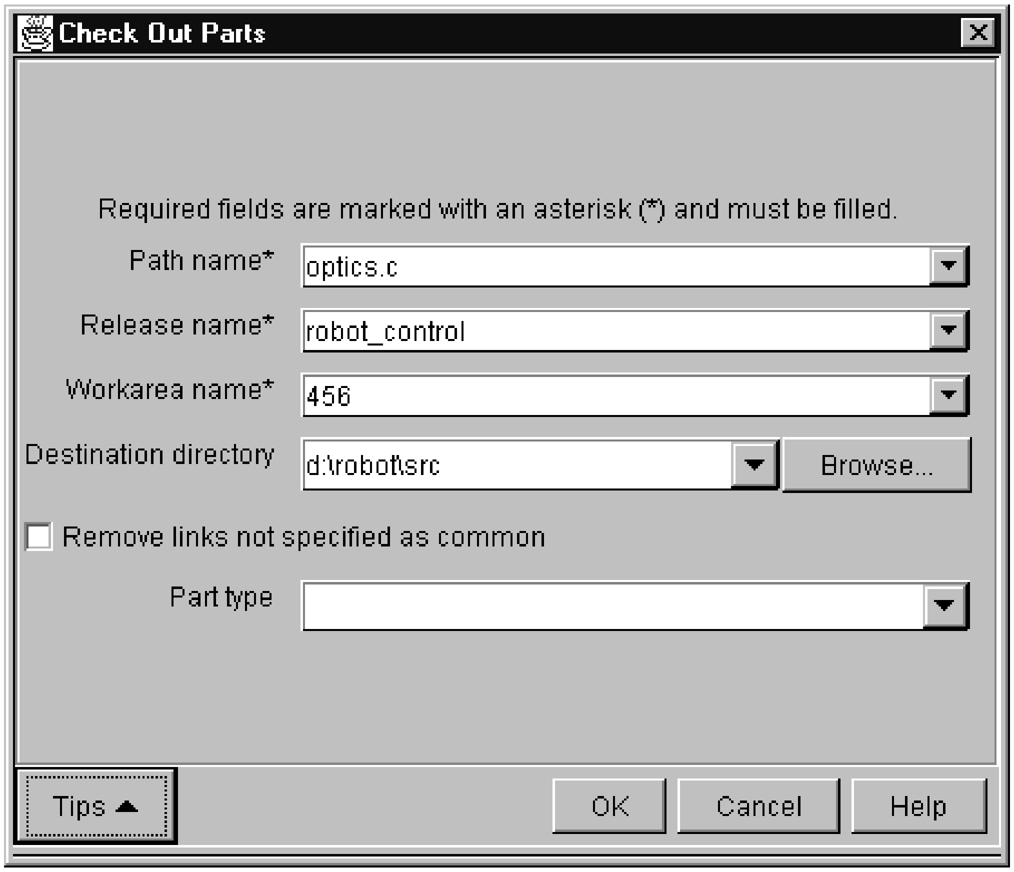
Now that the approval records have been accepted, Alex can check out the necessary parts. He decides that modifications are again required to the part optics.c. So, that is the part he checks out.

Alex must specify the workarea on the check-out command so that the part is obtained from the defect’s workarea. He does one of the following:

GUI

From the GUI, he:

1. Selects **Parts** → **Check out** from the Actions menu on the Home Page.
2. Types the following:
   - `optics.c` in the **Path name** field
   - `robot_control` in the **Release name** field
   - `456` in the **Workarea name** field
   - `d:\robot\src` in the **Destination directory** field
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Path name* optics.c

Release name* robot_control

Workarea name* 456

Destination directory d:\robot\src

☐ Remove links not specified as common

Part type

Figure 32. Check Out Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 456
  -relative d:\robot\src
```

Results

A copy of the part optics.c is checked out from TeamConnection and placed in the directory d:\robot\src. Because the release is following concurrent development mode, other users can also check out and change this part while Alex has it checked out.

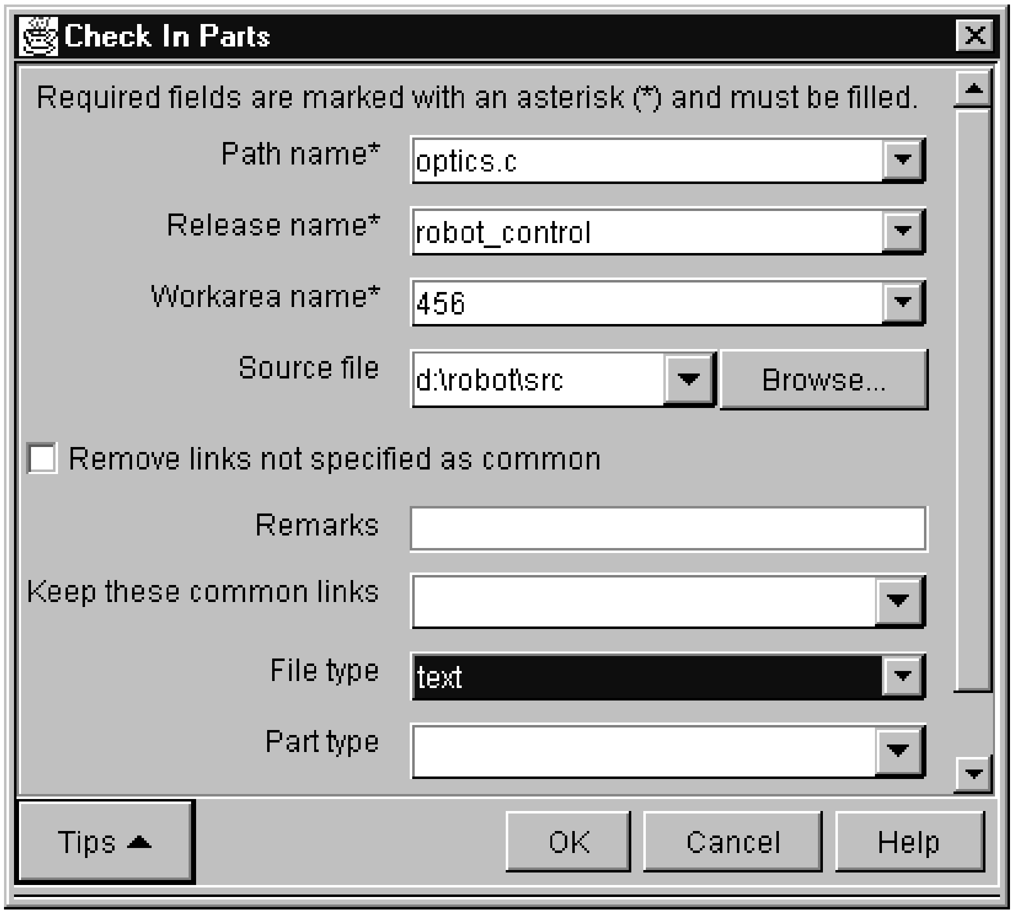
Checking in the changes

Alex makes his modifications and wants to test his corrections. First, he must check the part into the workarea. He does one of the following:

GUI

From the GUI, he:

1. Selects **Parts** → **Check in** from the Actions menu on the Home Page.
2. Types the following in the Check In Parts window, and then selects **OK**:
   - `optics.c` in the **Path name** field
   - `robot_control` in the **Release name** field
   - `456` in the **Workarea name** field
   - `d:\robot\src` in the **Source directory** field


Check In Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name* optics.c

Release name* robot_control

Workarea name* 456

Source file d:\robot\src

Remove links not specified as common

Remarks

Keep these common links

File type text

Part type

Figure 33. Check In Parts window

Note: Alex follows these steps because he knows the exact name of the part that he is checking in. If he does not know the name, or if he is checking in many parts, he can instead do one of the following to display a list of parts:
- Select the entry on his Home Page that displays the list of parts.
- Re-open the Parts window if it was previously minimized.
- Add an entry to his My Reports that lists all of his checked-out parts.

He then selects the parts that he wants to check in.

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 456
```

Results

Now the workarea contains the following part versions:

```
brain.c                          leg.c
brain.obj                        leg.obj
brain.exe                        foot.c
arm.c                            foot.obj
arm.obj                          optics.c (Alex's modification 1)
hand.c                           optics.obj
hand.obj
```

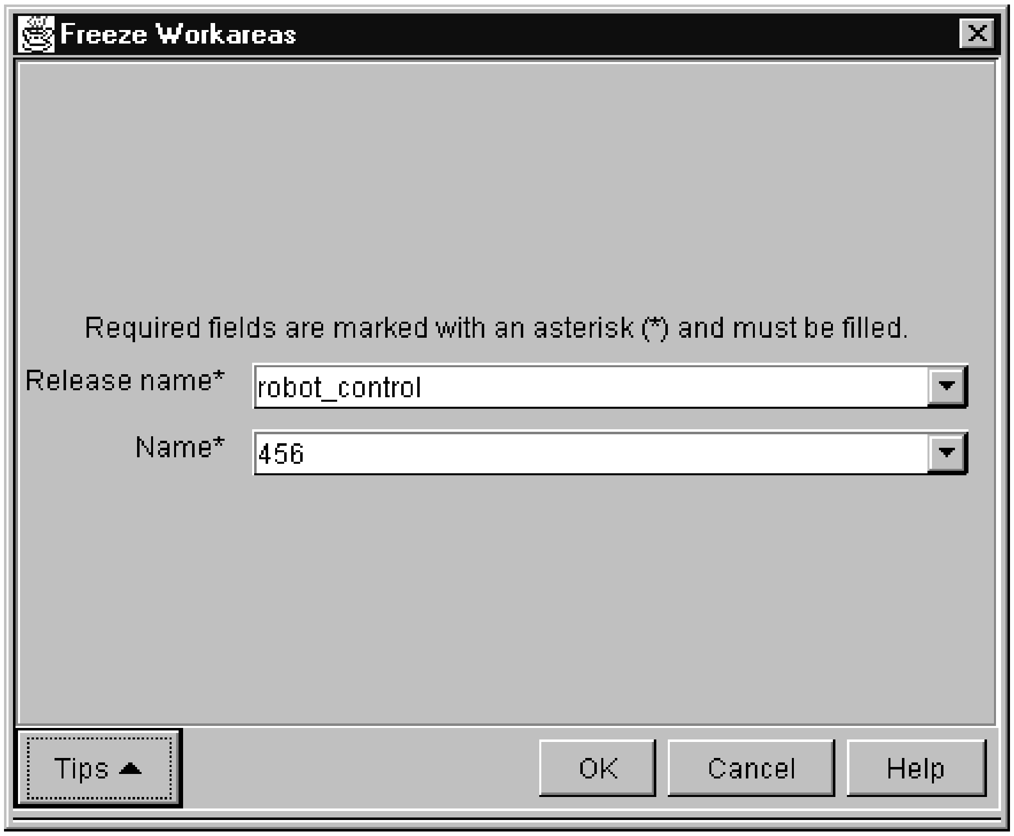
Freezing the workarea

Alex now wants to save, or freeze, the working system. He does one of the following:

GUI

From the GUI, he:

1. Selects **Workareas → Freeze** from the Actions menu on the Home Page.
2. Types `456` in the **Workarea name** field and `robot_control` in the **Release name** field if the information is not already there.
3. Selects **OK**.


Freeze Workareas

Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 456

Figure 34. Freeze Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -freeze 456 -release robot_control
```

Results

The freeze command saves workarea 456. Thus, TeamConnection takes a snapshot of the workarea, with all its parts and their visible versions, and saves it. Note, however, that a freeze action does not make the changes visible to the other people working in the release. This does not occur until the workarea is integrated.

Building the application

Alex now builds the application to verify that the changes he has made have fixed the problem. He does one of the following:

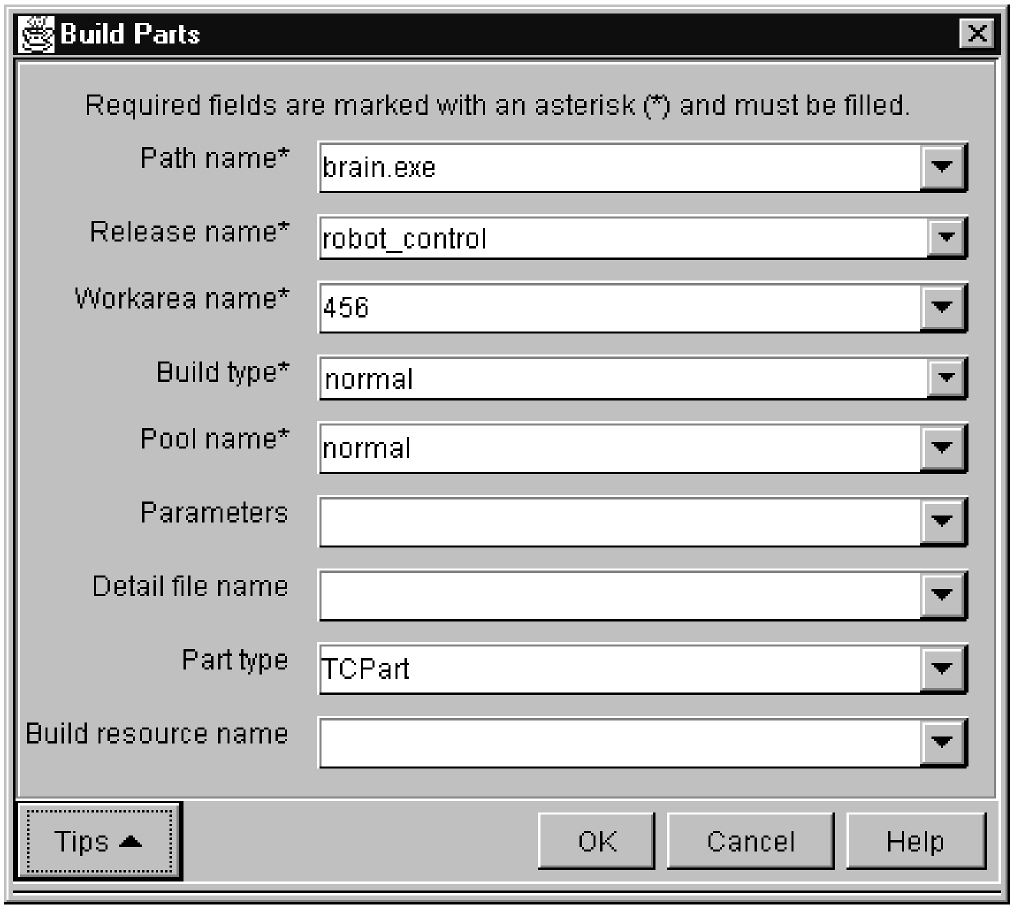
GUI

From the GUI:

Alex has a Parts window open with a list of all the parts that exist in workarea 456. He highlights the part brain.exe and then does the following:

1. Selects **Build** from the pop-up menu.
2. Types `normal` in the **Pool** field. The other required fields are pre-filled with the correct information.

3. Selects **OK** to start the build.



Build Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name*	brain.exe
Release name*	robot_control
Workarea name*	456
Build type*	normal
Pool name*	normal
Parameters	
Detail file name	
Part type	TCPart
Build resource name	

Figure 35. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 456 -pool normal
```

Results

Alex builds the application and tests the results. The modification seems to solve the problem.

Note: For a detailed build example, see “Chapter 14. Building an application: an example” on page 163.

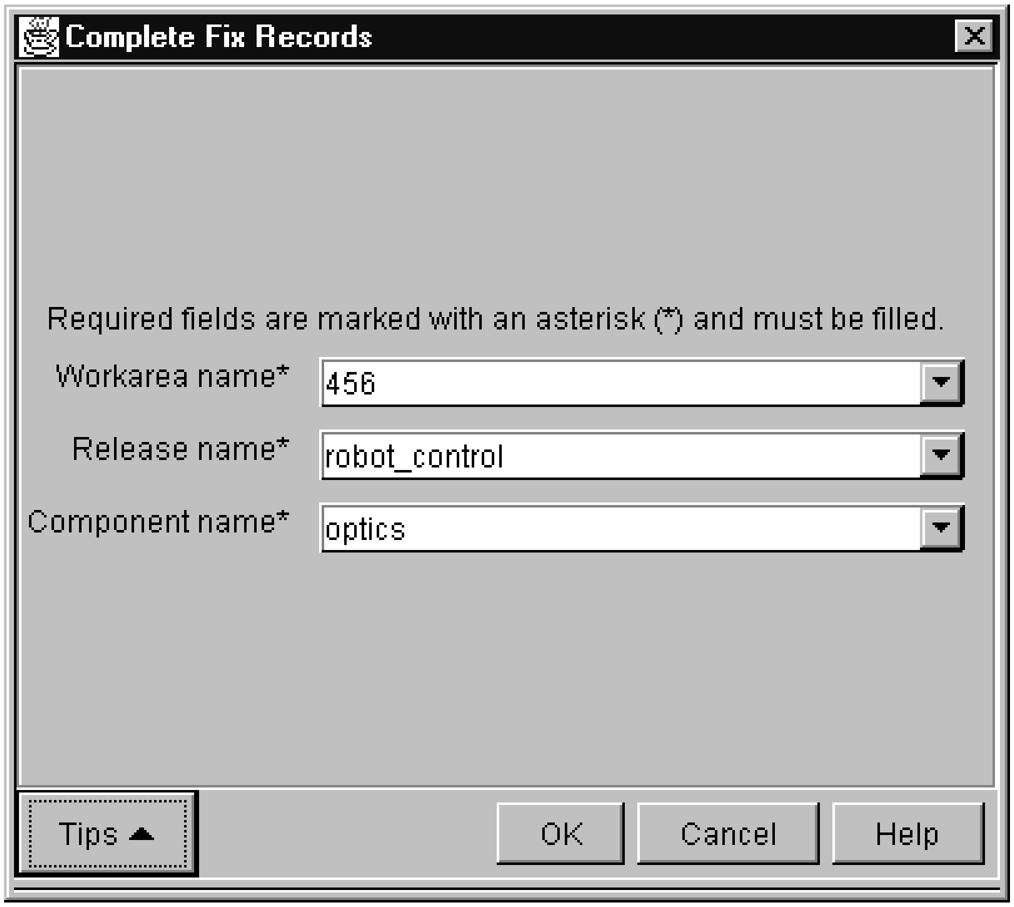
Accepting fix records

Alex is satisfied that the changes are complete and the part is ready to be integrated with other parts in the release. He does one of the following:

GUI

From the GUI, he:

1. Selects **Records → Fix → Complete** from the Actions menu on the Home Page.
2. Types the following in the Complete Fix Records window, and then selects **OK**:
   - `456` in the **Workarea name** field
   - `robot_control` in the **Releases name** field
   - `optics` in the **Component name** field



Required fields are marked with an asterisk (*) and must be filled.

Workarea name* 456

Release name* robot_control

Component name* optics

Figure 36. Complete Fix Records window

Command

From a command line, he issues the following command:

```
teamc fix -complete -workarea 456 -component optics -release robot_control
```

Results

The fix record moves to the complete state. Because only one fix record was generated for this defect, the workarea moves to the integrate state at the same time. When more than one fix record exists, they all must be completed before the workarea moves to the integrate state.

Integrating changed parts into a release

The changes that Alex has made are now ready to be put into the next set of changes scheduled to be integrated with the release. This set of changes is known as a *driver*.

A driver named 0105 currently exists, and several driver members have already been added to the driver. Therefore, the driver is in the integrate state.

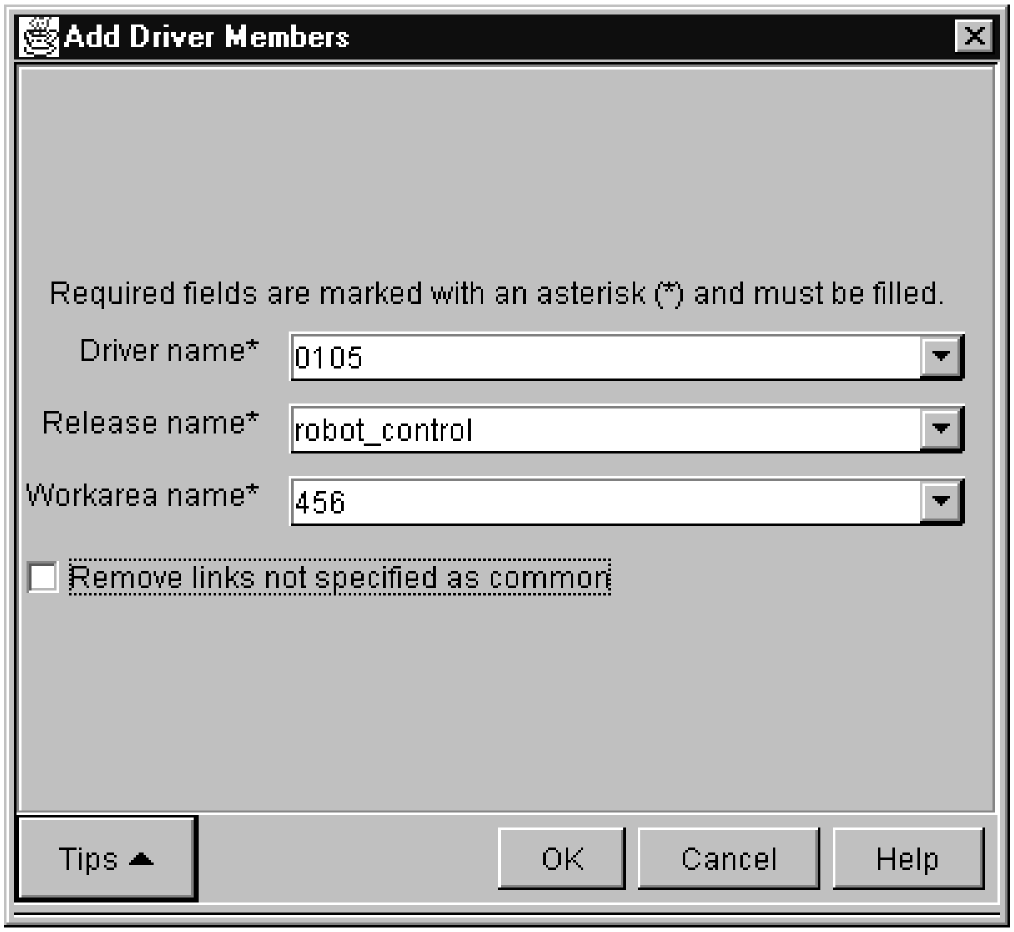
Adding a driver member

Carol, the project lead, adds workarea 456 as a *driver member* of driver 0105:

GUI

From the GUI, she:

1. Selects **Lists** → **Drive Member** → **Add** from the Actions menu on the Home Page.
2. Types the following:
   - `0105` in the **Driver name** field
   - `robot_control` in the **Release name** field
   - `456` in the **Workarea name** field
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Driver name* 0105

Release name* robot_control

Workarea name* 456

Remove links not specified as common

Figure 37. Add Driver Members window

Command

From a command line, she issues the following command:

```
teamc driverMember -create -driver 0105 -workarea 456 -release robot_control
```

Results

Carol previously created a driver member for driver 0105 that included changes to optics.c, so Carol is notified that collisions were detected. (Remember, the release is in concurrent development mode.)

Carol deletes the driver member for workarea 456. She then asks Alex to reconcile the collisions.

Reconciling the differences

Before Alex can reconcile the differences, he needs to do the following:

1. Return the workarea to the fix state
2. Reactivate the fix record
3. Refresh his workarea

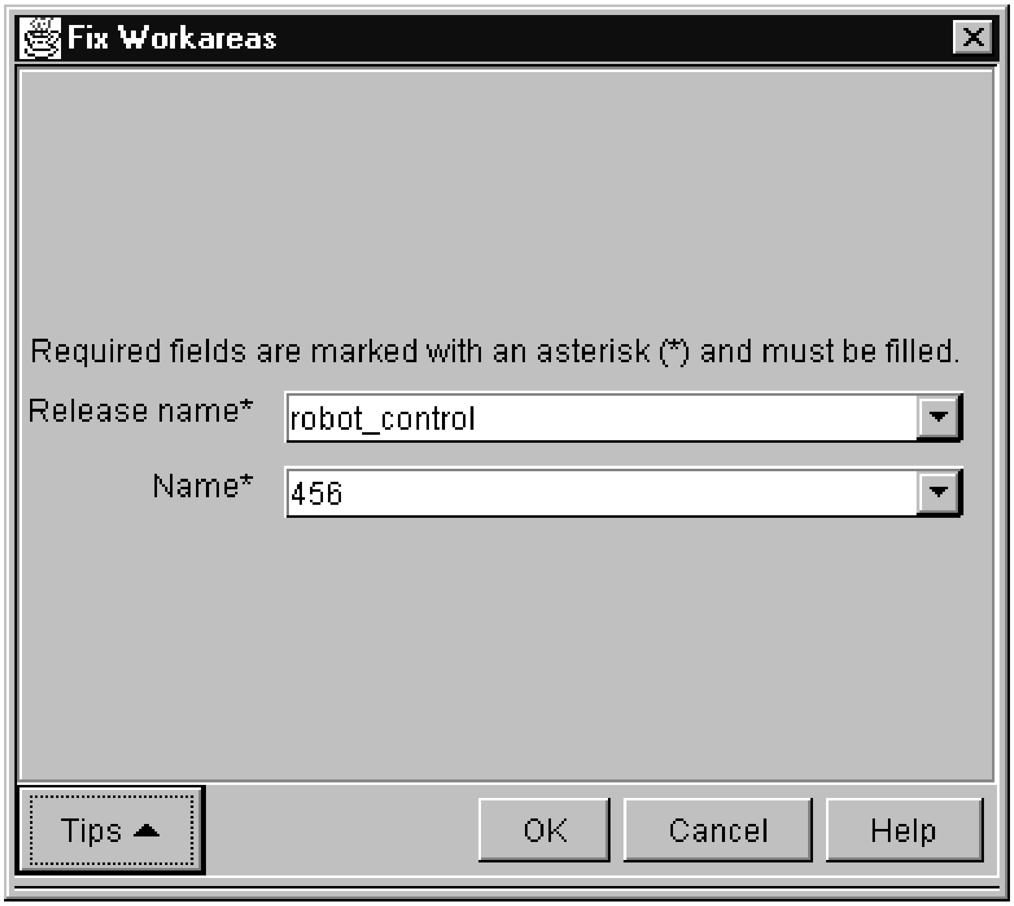
Returning the workarea to the fix state

The first step in reconciling the differences is for Alex to return workarea 456 to the fix state. He does one of the following:

GUI

From the GUI, he:

1. Selects **Workarea → Fix** from the Actions menu on the Home Page.
2. Types `456` in the **Name** field and `robot_control` in the **Release name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 456

Figure 38. Fix WorkAreas window

Command

From a command line, he issues the following command:

```
teamc workarea -fix 456 -release robot_control
```

Results

Workarea 456 is in the fix state. After the fix record is reactivated, Alex will check out optics.c from this workarea to reconcile the differences.

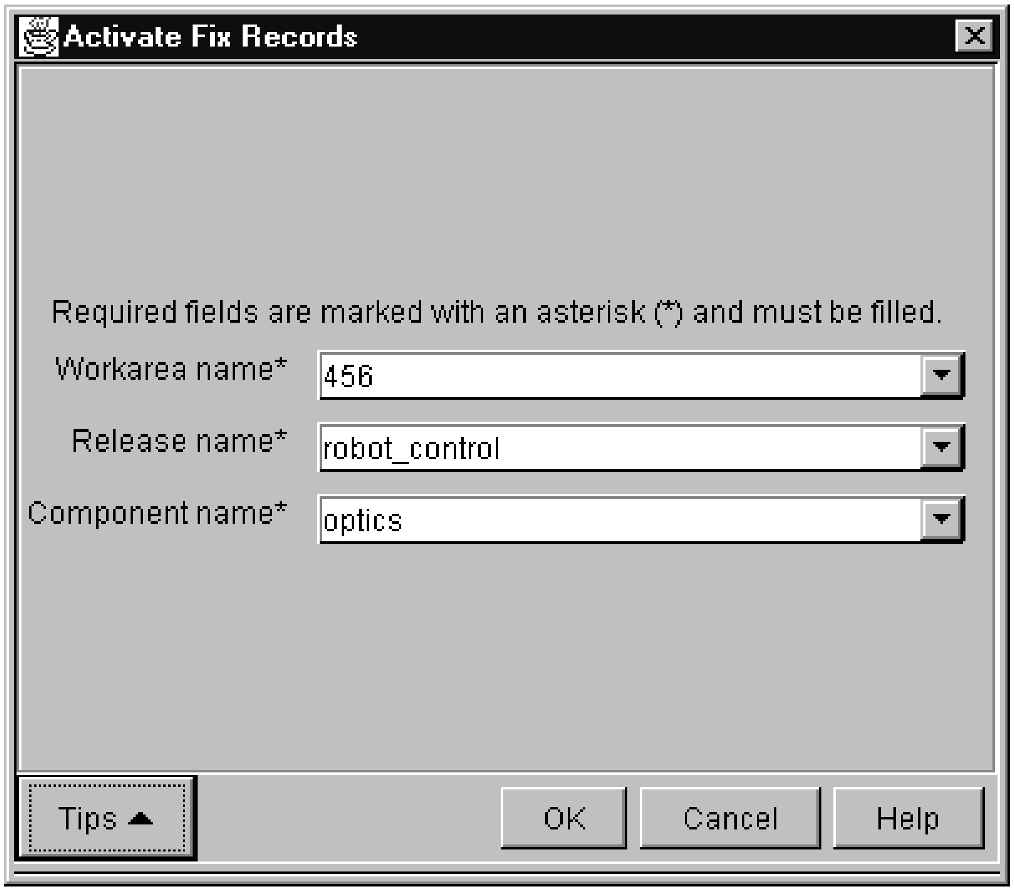
Reactivating the fix record

Currently, the fix record for workarea 456 is in the complete state. Alex must reactivate the fix record to move it back to the active state so that he can make the necessary changes to optics.c. He does one of the following:

GUI

From the GUI, he:

1. Selects **Records** → **Fix** → **Activate** from the Actions menu on the Home Page.
2. Types `456` in the **Workarea name** field and selects `robot_control` from the **Release name** field and `optics` from the **Component name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Workarea name* 456

Release name* robot_control

Component name* optics

Figure 39. Activate Fix Records window

Command

From a command line, he issues the following command:

```
teamc fix -activate 456 -release robot_control -component optics
```

Results

The fix record returns to the active state.

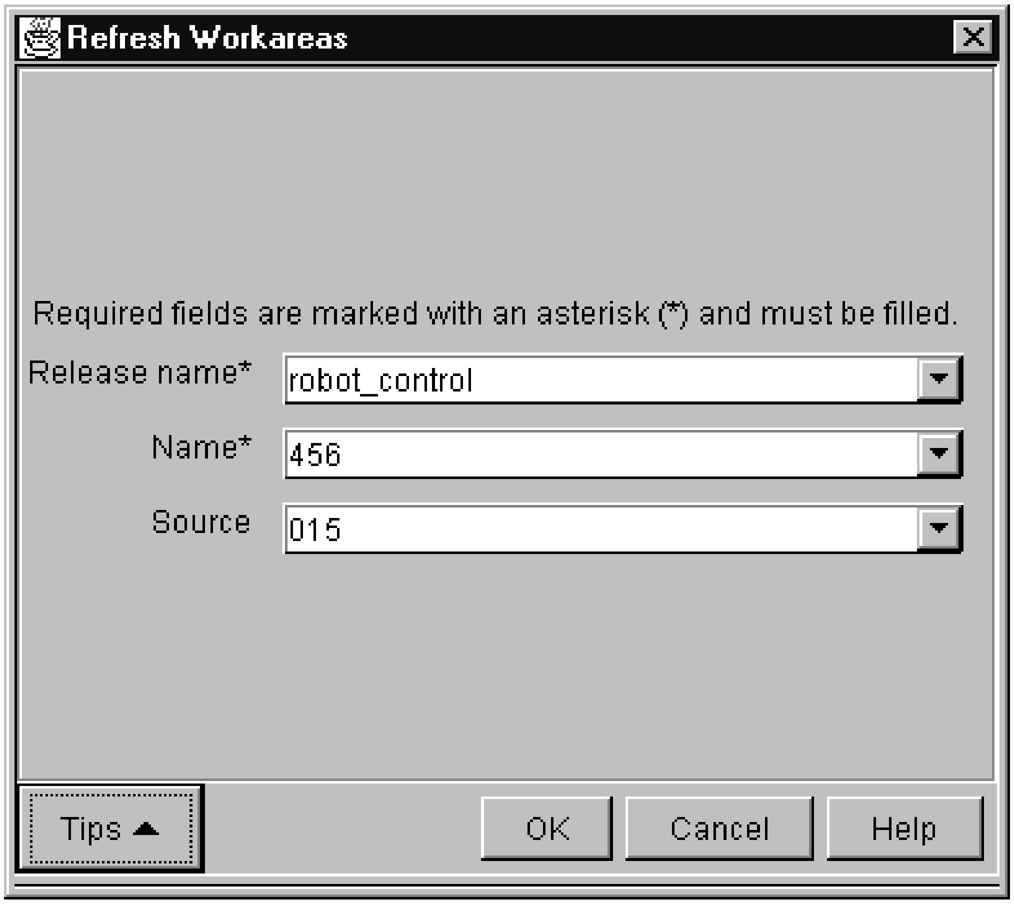
Refreshing the workarea

Alex now needs to refresh his workarea with the parts that are already in driver 0105. He does one of the following:

GUI

From the GUI, he:

1. Selects **Workareas → Refresh** from the Actions menu on the Home Page.
2. Types the following in the Refresh Workareas window and selects **OK**:
   - `456` in the **Name** field
   - `robot_control` in the **Release name** field
   - `0105` in the **Source** field



Required fields are marked with an asterisk (*) and must be filled.

Release name* robot_control

Name* 456

Source 015

Figure 40. Refresh Workareas window

Command

From a command line, he issues the following command:

```
teamc workarea -refresh 456 -release robot_control -source 0105
```

Results

TeamConnection notifies Alex of the collision, so his next step is to reconcile the differences. He follows the same procedure that is described on page 72.

Alex completes the fix record and then tells Carol that he has reconciled the part differences and that she can now create the driver member. She creates the driver member without any collisions this time.

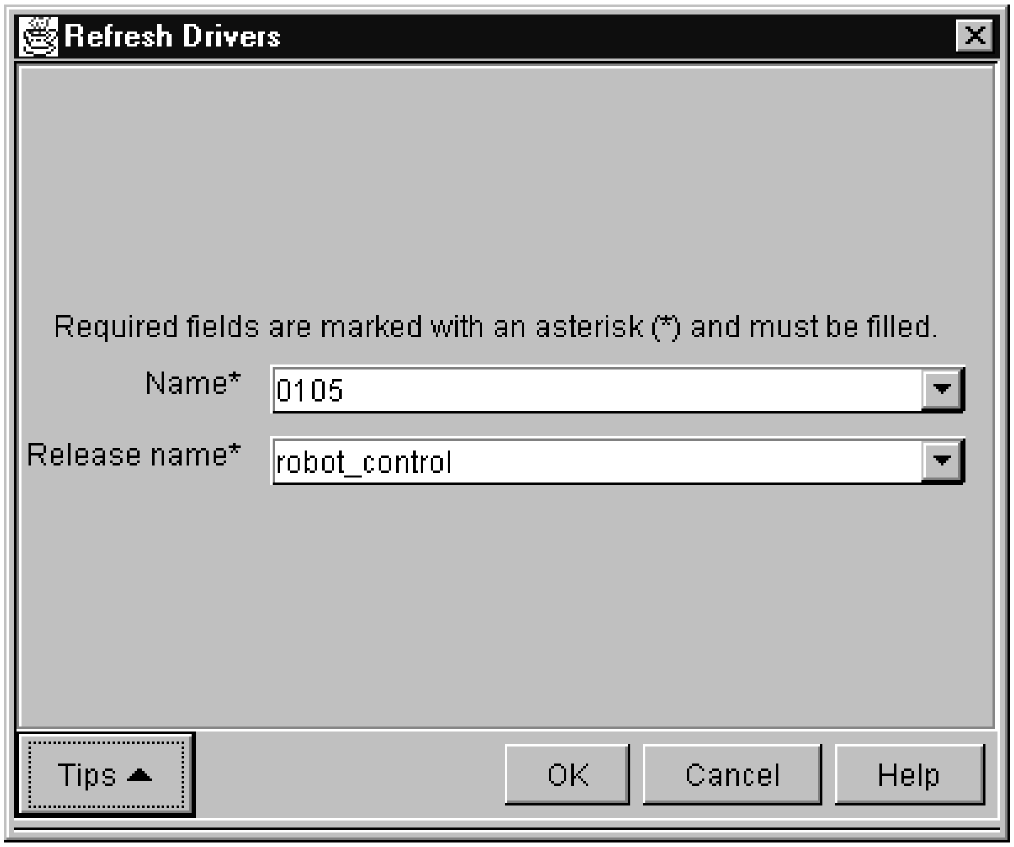
Refreshing the driver

Carol is ready to integrate the changes in driver 0105 with the release. Because other team leads have integrated changes as well, she wants to build her driver with the most current release part versions. She does one of the following:

GUI

From the GUI, she:

1. Selects **Drivers → Refresh** from the Actions menu on the Home Page.
2. Types `0105` in the **Name** field and `robot_control` in the **Release name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Name* 0105

Release name* robot_control

Figure 41. Refresh Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -refresh 0105 -release robot_control
```

Results

This command refreshes driver 0105 with any committed updates to the release.

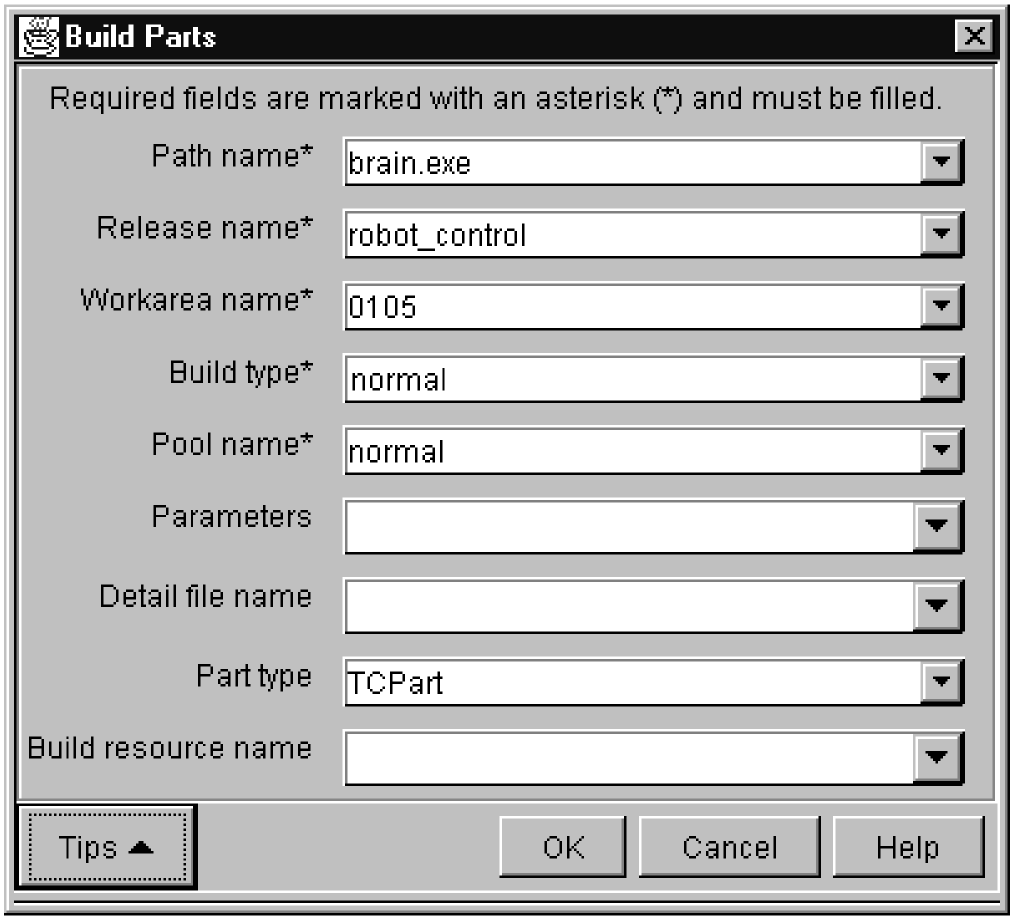
Building the driver

Carol builds the application using the parts current to driver 0105. She does one of the following:

GUI

From the GUI, she:

1. Creates a filter that displays brain.exe.
2. Highlights brain.exe and selects **Build** from the object's pop-up menu menu
3. Types the following in the Build Parts window:
   - `brain.exe` in the **Path name** field.
   - `robot_control` in the **Release name** field.
   - `0105` in the **Workarea name** field.
   - `normal` in the **Pool name** field.
4. Selects **OK** to start the build.



Required fields are marked with an asterisk (*) and must be filled.

Path name*	brain.exe
Release name*	robot_control
Workarea name*	0105
Build type*	normal
Pool name*	normal
Parameters	
Detail file name	
Part type	TCPart
Build resource name	

Figure 42. Build Parts window

Command

From a command line, she issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 0105 -pool normal
```

Results

Carol runs some simple regression tests to verify that the application built properly. She is satisfied with the results, and is ready for the next step — committing the driver changes to the release.

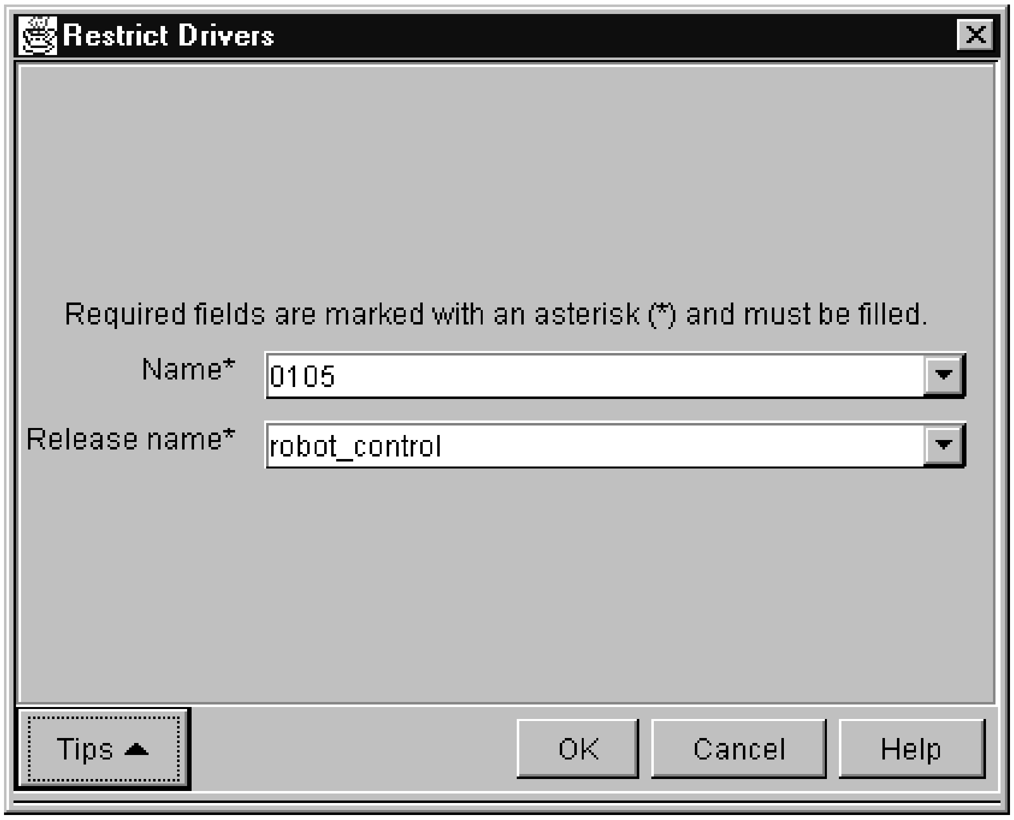
Restricting the driver

After all changes have been integrated with the release, Carol needs to make some final changes before building the driver. To enable her to make these changes while protecting the driver from access by anyone else, she needs to restrict access to it. She does one of the following:

GUI

From the GUI, she:

1. Selects **Driver → Restrict** from the Actions menu on the Home Page.
2. Types `0105` in the **Name** field and `robot_control` in the **Release name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Name* 0105

Release name* robot_control

Figure 43. Restrict Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -restrict 0105 -release robot_control
```

Results

This command restricts driver 0105 so that only Carol is able to make changes to it. Carol is now ready to build the application.

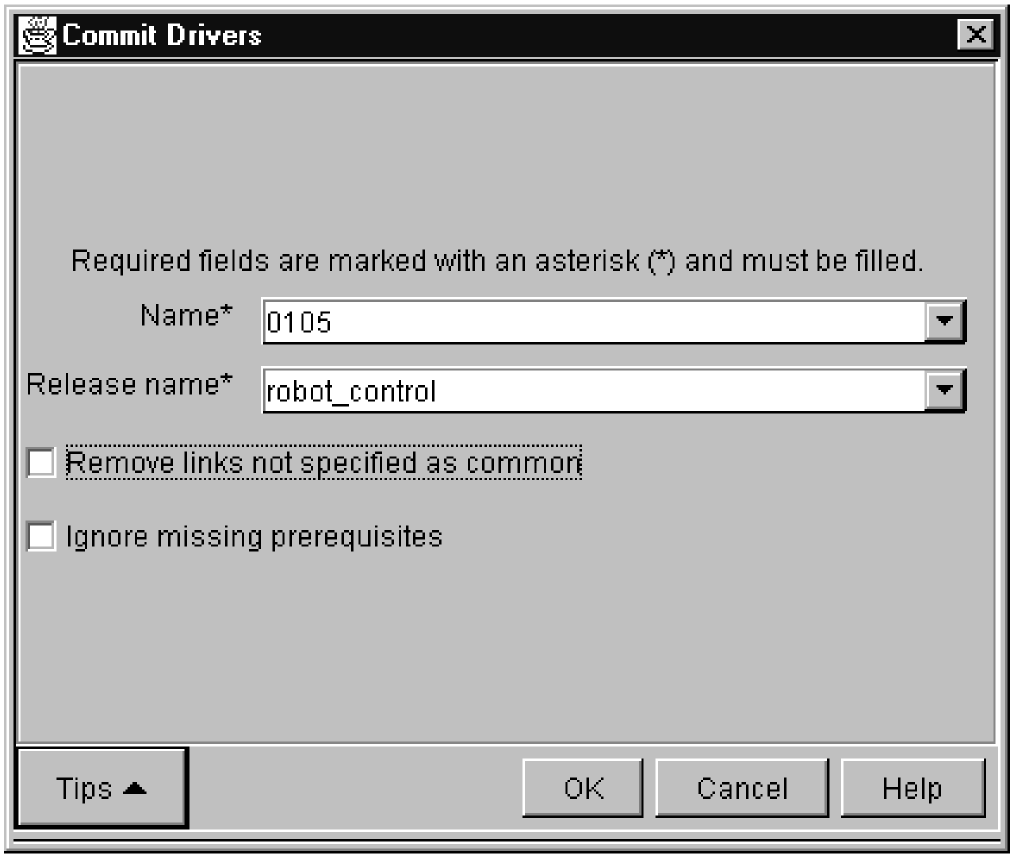
Integrating the parts

Carol commits the changes in the driver to the release by doing one of the following:

GUI

From the GUI, she:

1. Selects **Driver → Commit** from the Actions menu on the Home Page.
2. Types `0105` in the **Name** field and `robot_control` in the **Release name** field.
3. Selects **OK**.


Commit Drivers

Required fields are marked with an asterisk (*) and must be filled.

Name* 0105

Release name* robot_control

☐ Remove links not specified as common

☐ Ignore missing prerequisites

Figure 44. Commit Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -commit 0105 -release robot_control
```

Results

TeamConnection moves the part versions associated with driver 0105 into the release. Other members of the team can now view the changes. Committing a driver commits all workareas designated as driver members and all parts changed in reference to those workareas.

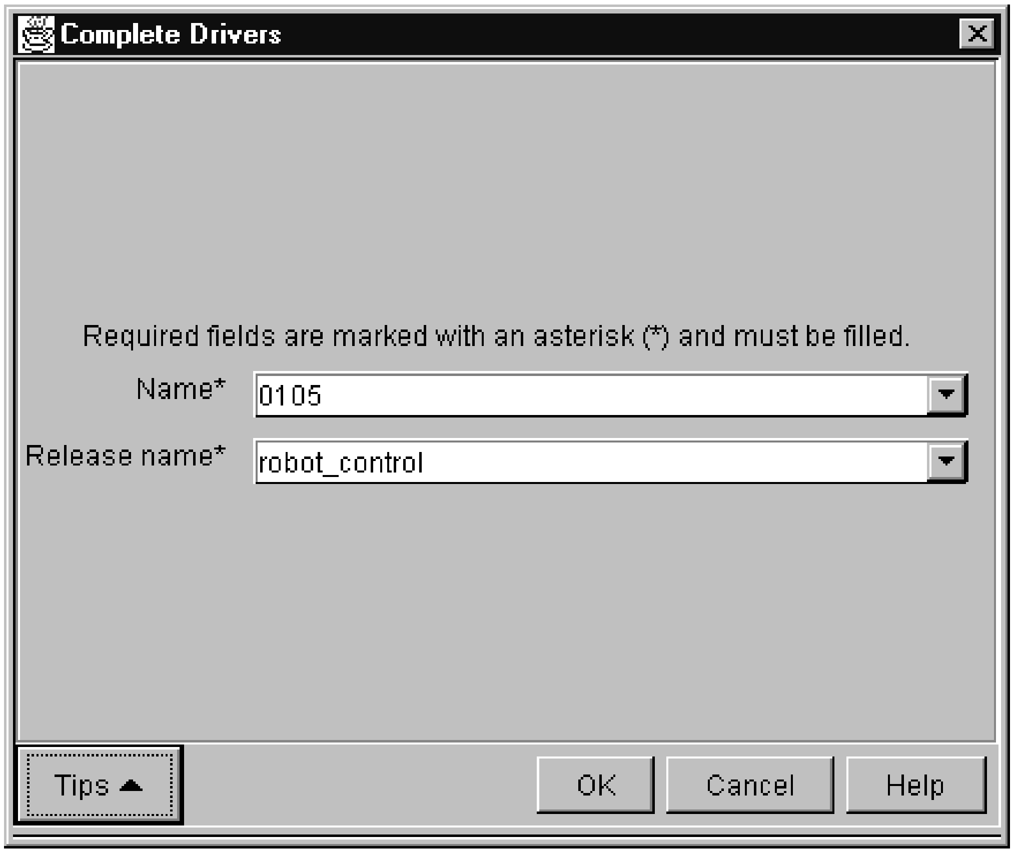
Completing the driver

The driver is ready for formal testing in the specified release's environment list. Testing is tracked using test records for each environment in which testing is to be done. To create the test records, Carol must complete the driver.

GUI

From the GUI, she:

1. Selects **Driver → Complete** from the Actions menu on the Home Page.
2. Types `0105` in the **Name** field, and selects `robot_control` from the **Release name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Name* 0105

Release name* robot_control

Figure 45. Complete Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -complete 0105 -release robot_control
```

Results

All the workareas in the driver are changed to the test state, and test records are created.

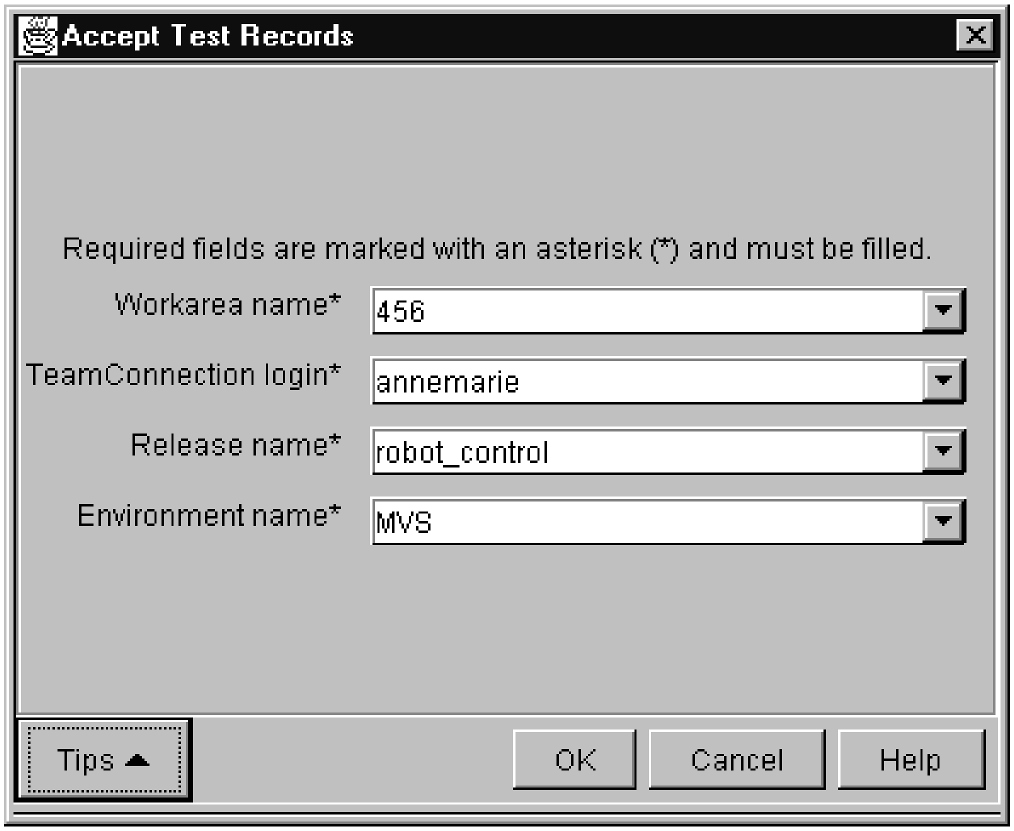
Testing the built application

Annmarie is the tester for the MVS version of the robot application. When she receives notification that the test record is in the ready state, she tests the part changes that were made within the release by Alex and several of his team members. The tests complete successfully, so she accepts the test record by doing one of the following:

GUI

From the GUI, she:

1. Selects **Records** → **Test** → **Accept** from the Actions menu on the Home Page.
2. Types `456` in the **Workarea name** field, and selects `robot_control` from the **Release name** field and `MVS` from the **Environment name** field.
3. Selects **OK**.



Required fields are marked with an asterisk (*) and must be filled.

Workarea name* 456
TeamConnection login* annemarie
Release name* robot_control
Environment name* MVS

Figure 46. Accept Test Records window

Command

From a command line, she issues the following command:

```
teamc test -accept -workarea 456 -release robot_control -env mvs
```

Results

Annmarie's test record moves to the accept state. However, workarea 456 will not go to the complete state until Tim, who is the tester for the OS/2 environment, marks his test record.

After all test records are moved from the ready state, the workarea moves to the complete state. Because the component process includes the verifyDefect subprocess, defect 456 moves to the verify state. A verification record for the defect is created in the ready state.

Using a configured process

The scenarios in this chapter and the preceding chapter illustrate one release with no process management enabled and another release with full process management enabled. However, administrators can define a release that requires users to work with some intermediate level of process management. That is, the administrator can remove some of the subprocesses from the full-tracking scenario.

For example, the administrator might want to eliminate the driver subprocess. If the driver subprocess is eliminated, the user cannot create driver members to associate the changes in a workarea with a driver. Likewise, users cannot commit drivers to integrate several workareas with the release. Instead, users integrate the changes for each workarea by integrating the workarea with the release.

To demonstrate how this works, assume that Carol and Alex are trying to fix the robot's dislike of striped wallpaper using a release without the driver subprocess enabled. Initially, the scenario is not affected by the absence of the driver subprocess. The defect is opened, and a work area is created. Alex, after receiving notice that he needs to solve the problem, goes through the process of checking out the faulty part, making fixes, checking the fixes into the workarea, and rebuilding. He can still freeze the workarea whenever he wants to save its current content.

The difference occurs when Alex is ready to integrate his changes with the release. When the driver subprocess is not enabled, Alex issues the following command:

```
teamc workarea -integrate 456 -release robot_control
```

This command moves the part versions associated with work area 456 into the release so they are visible to other developers. However, if collision records are created, TeamConnection flags the concurrent changes and stops the integration until the changes are reconciled and the corresponding collision records are completed.

Retrieving a past version of a part

Versioning is an insurance policy for the developer. By freezing the workarea, the developer knows that the parts currently visible in the workarea will be retained in their current form.

For this example, assume that Alex just updated the optics.c module to add support for a new zoom lens. Alex did a considerable amount of work on this task, and it required a dozen check-out, check-in, and build cycles before he finished. Alex's workarea now contains the following:

```
brain.c                              leg.c
brain.obj                            leg.obj
brain.exe (from Alex's build 12)     foot.c
arm.c                                foot obj
arm.obj                              optics.c (modification 12)
hand.c                               optics.obj (from Alex's build 12)
hand.obj
```

Next Alex must update the brain.c part to set the appropriate conditions for activating the new zoom capability. He does not yet want to integrate his changes to optics.c for the zoom lens with the release because they are of little value without his changes to brain.c. Also, he is not certain that he is completely done with optics.c until he completes the modifications to brain.c. Rather than integrate an incomplete change, he freezes his workarea by issuing the following command:

```
teamc workarea -freeze 1208 -release robot_control
```

This command takes a snapshot of the workarea and its parts in their current state.

As Alex works on the brain.c module, he makes sweeping modifications to optics.c to simplify the interface between brain.c and optics.c. Unfortunately, he realizes too late that the simplification he is pursuing will not work. Rather than spend several hours removing his updates to optics.c, he wants to start fresh from a copy of optics.c that does not contain the changes for the simplification.

Alex has frozen his workarea three times since beginning work on the zoom lens integration. Also, he has done additional check-ins to his workarea since his last freeze. He cannot remember the particular version of his workarea that contains the copy of optics.c that he wants. So, he wants to see all the versions of his workarea that he has saved. He issues the following report command:

```
teamc report -view versionView -where "workAreaName='1208' and
    releaseName='robot_control'" -stanza
```

This command returns a list of the versions frozen from workarea 1208. The report looks like this:

```
name                1208:1
workAreaName        1208
releaseName         robot_control
predecessor         robot_control:5
hasSuccessor        yes
releaseVersion      no
addDate             1995/01/11 14:30:26
freezeDate          1995/01/11 15:00:00

name                1208:2
workAreaName        1208
releaseName         robot_control
predecessor         1208:1
hasSuccessor        yes
releaseVersion      no
addDate             1995/01/12 09:25:13
freezeDate          1995/01/12 17:15:58

name                1208:3
workAreaName        1208
releaseName         robot_control
predecessor         1208:2
hasSuccessor        yes
releaseVersion      no
addDate             1995/01/14 11:13:25
freezeDate          1995/01/15 09:01:35
```

```
name                1208:4
workAreaName        1208
releaseName         robot_control
predecessor         1208:3
hasSuccessor        no
releaseVersion      no
addDate             1995/01/16 08:10:15
freezeDate          1995/01/16 10:05:11
```

So what does it all mean?

- `name` is the name of the version in the workarea.
- `workAreaName` is the name of the workarea that owns the version.
- `ReleaseName` is the name of the release that owns the version.
- `Predecessor` is the name of the version that precedes, or is the parent of, this version.
- `hasSuccessor` has a value of *yes* if the version has a successor, *no* if it does not.
- `releaseVersion` has a value of *yes* if the version is part of the release's main version history; the value is *no* if the version belongs to a workarea.
- `addDate` is the date and time the version was created.
- `freezeDate` is the date the version was frozen.

This report seems erroneous. TeamConnection returned four versions in the report even though Alex has executed the freeze command against his workarea only three times. The fourth version, 1208:4, is the unfrozen version in which Alex is currently making his changes.

Another concern might be the predecessor of the first version returned in the report. Why is its predecessor robot_control:5? At some point Alex began his work by making modifications to the latest code in the release. The first version of Alex's changes is based on the release version robot_control:5.

After reviewing the report, Alex thinks that his last working copy of optics.c was saved when he created version 1208:2. However, to make sure, he wants to see the parts modified in version 1208:2. He issues the following command:

```
teamc report -view partView -version 1208:2  -release robot_control
            -where "currentVersion='1208:2'" -stanza
```

This report returns a list of parts visible to version 1208:2 that have a currentVersion (or version ID) of 1208:2. If a part has such a version ID, the part was modified in the version 1208:2.

Note: If the -where clause were not specified, the report would return all of the parts visible from version 1208:2.

The TeamConnection system returns the following report:

```
baseName            optics.c
releaseName         robot_control
compName            robot_dev
versionSID          1208:2
addDate             02/02/94
lastUpdate          04/15/94
pathName            smarts\eyes\optics.c
nuVersionSID        1208:2
nuAddDate
nuDropDate
nuPathName
userLogin           alexm
fmode               0640
```

Because optics.c is the only part modified in version 1208:2, Alex assumes it is the copy he wants. He extracts the part by issuing the following command:

```
teamc part -extract optics.c -version 1208:2 -workarea 1208 -release robot_control
```

This command extracts the desired copy of optics.c from the frozen version 1208:2. Alex can then overlay the corrupted copy of optics.c that he has checked out with the copy he just extracted, and he can start over fresh. He can also check in the overlaid optics.c to his workarea.

This method works only for parts with a file type of TCPart. If your part has a type of something other than TCPart, you can do one of the following to restore the part:

- Use the undo action if restoring to the previous version.
- Use the link action to link to a previous version.

In addition to the reporting features mentioned above, Alex can also obtain a list of workareas by issuing the following command:

```
teamc report -view WorkAreaView -where "releaseName='robot_control'" -stanza
```

The report that is returned lists the workareas in the release robot_control. A user can also see the parts changed for each work area by specifying the -long parameter on this command.

Part 3. Using TeamConnection Integrated Notes Databases

This section is for anyone who will be setting up and using Integrated Notes Databases

Chapter 7. Introduction to TeamConnection Integrated Notes Databases

VisualAge TeamConnection Integrated Notes Databases provide a documentation facility that supports software development. You can use Integrated Notes Databases to:

- Communicate with TeamConnection from within Lotus Notes documents.
- Associate Notes documents with existing TeamConnection defects and features.
- Define and customize your own databases that can assist you in creating software design documents, project requirement documents, and test cases.
- Make your documents available to your team for review using the Notes review function.
- Archive documents
- Make your databases available on the web.

Prerequisites and dependencies

Before you set up Integrated Notes Databases, make sure the machine you are using (generically, the *Notes server*) and the client machines have Lotus Notes Release 4.5.3a or higher and a Version 3 ″classic″ TeamConnection client.

If you have Lotus Notes 4.6.2 or higher, you will able to use the new Web-Based Integrated Notes client. Also note that Lotus Notes 4.6.2 does not support OS/2 clients.

Note: The 3.0.3 TeamConnection Database requires a 3.0.3 Family server. Also note that you do not need the ″classic″ client when accessing a database in a web browser.

Online help and planning issues

TeamConnection Integrated Notes databases come with extensive online help, which you can use to help you plan for the type of databases you might want to create. You can access this information from the *About*, *Using*, and contextual help information available from the database **Help** menu.

The *About* document provides graphical previews for the Requirements, Development, and Test database types. These previews provide a graphical depiction of available documents and the document hierarchy for each database, along with the default setup values available to an administrator.

The *Setup* help, which you can access from the *About* document (or from Setup itself) provides information about access control lists, customization, and other information.

The *Using* document provides detailed information that users will need to take advantage of Notes Integrated Database functionality. For each database type, *Using* documents the specific steps required to create and respond to the available document types. In addition, *Using* provides a listing of each Notes element (form or view), along with a brief description of the context in which it is appropriate.

As you navigate through the integrated database, you can access help in the several ways:

For a view
Select the **Help** button in the action bar to look at help for the view.

For the document that you are editing
Select the **Help** button in the action bar to look at help for a document or form.

For a field in a form
Click in the field and look at the bottom of the screen for a short description of the field. Look to the right of the field to see if there is descriptive text for help in entering the correct format. Select the **Help** push button in the action bar to get more detailed help on the field.

Other planning issues

Before you create an Integrated Notes database, you might need to consider some issues that affect how the database will be used. Most notably, if you plan to access an Integrated Notes Database using a browser that will also access TeamConnection, you will need to find out if the server on which you plan to put the database will allow unrestricted agents to run on it. This is required to allow server-to-server communication between the Notes Domino server on which your database is published and the TeamConnection family server.

If you can run unrestricted agents on the server, you will then be able to configure your database for server-to-server communication when you setup the database.

Chapter 8. Creating and Maintaining Integrated Notes Databases

This section instructs you how to:

- Create and add an Integrated Notes Database to your workspace.
- Add a user to the database access control list.
- Replace a database's design from a template

Creating a database and adding it to your workspace

To help you with developing Notes applications, database templates are included on the IBM VisualAge TeamConnection Enterprise Server installation CD-ROM. These files, which serves as a single source for a number of database uses, are placed on the TeamConnection server during installation. The file you should use to create your database depends upon the release of Lotus Notes that you are using.

Lotus Notes 4.6.2 or higher

If you are using Lotus Notes 4.6.2 or higher, you should use the file called teamcweb.ntf to create your database. This file is located in *<x>:\<teamc>\nls\cfg\enu*.

To create a database using teamcweb.ntf, perform the following steps:

1. Copy teamcweb.ntf from the server to your local workstation into your local notes data directory (for example, *x:\notes\data*)
2. Select **File->Database->New**. The **New Database** window appears.
3. Select TeamConnection Template 3.0.3 from the combo box.
4. Name the database you are creating by typing a name in the **Title** field. As you type, a file name will be created in the **File Name** field. Make sure the name ends with the .NSF suffix. For example, development.nsf.
5. If you want your database to be created somewhere other than in your local notes data directory, select the folder icon button. The **Choose a folder** window appears. Select the directory on your local file system in which you want to store the database, and then select the **OK** push button. The directory name you selected appears in front of the database name in the **File Name** field.

   Note that you can also specify a specific Notes Server during this step
6. Select the **OK** push button. The new database is created, and an icon for it appears in your workspace.

When you have successfully created your database, you should then add yourself to the database's access control list. See "Adding a user to the access control list" on page 108.

Lotus Notes 4.6.1 or lower

If you are using Lotus Notes Release 4.5.3a - 4.6.1, you should use the file called teamc.nsf to create your database. This file is located in *<x>:\<teamc>\nls\cfg\enu*.

The database that is created using teamc.nsf is identical to the 3.0.2 version and thus provides limited web enablement. However the advantage of using teamc.nsf is that it works with OS/2 Notes clients.

To create a database using teamc.nsf, perform the following steps:

1. Copy teamc.nsf from the server to your local workstation into your local notes data directory (for example, *x:\notes\data*)
2. Select **File->Database->Open**. When the **Open Database** window appears, type *teamc.nsf* in the **File Name** field, and then select the **Add Icon** push button. Then close the **Open Database** window by selecting the **Done** push button. An icon for the TeamConnection Database appears in your workspace.
3. Select the TeamConnection Database icon.
4. Select **File->Database->New Copy**. The **Copy Database** window appears.
5. The **Server** field should contain the Local value. Replace the name in the **Title** field with a new name of your choosing.
6. Replace the name in the **File Name** field with a new name. The name you type should match the naming naming convention of your organization. The name must also have a .nsf suffix. For example: development.nsf.
7. If you want your database to be created somewhere other than in your local notes data directory, select the folder icon button. The **Choose a folder** window appears. Select the directory on your local file system in which you want to store the database, and then select the **OK** push button. The directory name you selected appears in front of the database name in the **File Name** field.

   Note that you can also specify a specific Notes Server during this step.
8. In the **Copy** field, select the **Database design only** radio button, and leave the **Access Control List** check box checked, so that the new database will inherit those characteristics from the empty template database.
9. Select **OK**. The new database is created, and an icon for it appears in your workspace.

When you have successfully created your database, you should then add yourself to the database's access control list.

Adding a user to the access control list

Before you can set up your database for use, you need to add yourself as an Administrator to the **Access Control List** for the database. To do this, complete the following steps:

1. Using mouse button 2, select the TeamConnection database icon on your Notes workspace page.
2. Select **Access Control**.
3. From the **Access Control List** window select **Add**.
4. Enter your Full User Name, exactly as it appears in your Notes Address Book, and then select **OK**.
5. Assign a value of **Person** in the **User type** field and **Editor** in the **Access** field.
6. Assign yourself the role of **Administrator** and **Author** by selecting them in the **Roles** box of the panel.
7. Assign a value of Person in the **User type** field and Manager in the **Access** field.
8. Select **OK**.

Note: You should adjust the default access as appropriate for your organization. The default access gives everyone full database access.

Setting up the database

After you have given yourself administrator authority for a database, you can then set up the database for use. There are two phases to setting up the database: initialization and customization. Initialization is mandatory; customization is optional.

To setup the database, double-click on the database icon and then select **Setup the Database**. The online help provides extensive information about setting up the database.

Customizing the database

You can use the **Customization** facility to customize the database design. You access this facility by selecting **Setup the Database** from the main database window

You can use the **Customization** facility to modify the following:

- Notes document names, titles, and subtitles.
- Structure of the Notes document hierarchy.
- States each Notes document may progress through.
- TeamConnection components and configurable fields information (refresh only).
- TeamConnection feature and defect attributes to store in Notes documents.
- Reconciliation of Notes and TeamConnection data.

The **Customization** setup facility addresses the following areas of database manipulation:

- Notes Database Customization
- Modify TeamConnection Access
- Reconciliation of Notes and TeamConnection Data

You can also use the customization wizard to help you customize the database, which you can access by selecting the **Customization Wizard** push button.

Testing the database

After you have set up your database, you should test the database to make sure it it working the way you want it to. At a minimum, you should perform the following activities:

- Create documents.
- Update documents. This should include testing the state system, which would involve moving a document's state to Approved.
- Reopen documents.
- Select **Feature** and **Defect** push buttons.
- Open Defects and/or Features.
- Examine views.

You should also add any users involved in the testing process to the access control list for this database. The default **Access** and **Role** of Author is recommended. You also might want to assign an access level of **editor** and a role of **project leader** to those members of your organization who will serve as project leaders.

When the database is working to your satisfaction (you can go back to the setup several times to make changes and refine it), you are ready to make it available for to your user community. Contact your Notes Administrator to arrange moving your database to a Domino Server. Be sure to delete any documents you created during local testing and evaluation.

Note: If document numbering is turned on, you may want to reset it when the database goes into production. To reset document numbering, perform the following steps:

1. Select the Notes **Navigator**.
2. Select **Administration->Document Control**.
3. Edit the document and set the number to `0` (zero).
4. Select **File->Save**.

When you are satisfied that the database is ready for use, put the database that you set up on a Notes server using the normal Notes process.

Note: To enhance performance for database users, you might want to turn replication on using the standard Notes process.

Database maintenance: replacing a design from a template

In the course of using Notes Integrated databases, you might have reason to replace your current database design with one from a new template. Replacing your database design changes the database's views (except for private ones), forms, agents and fields to match those in the template. You might want to replace your design when:

- You receive a new template from IBM that has fixes and new functions.
- You've made many local changes during advanced customization.

To replace a database's design with the one in a template, do the following:

1. Make a new copy of the database whose design you want to replace, using the following procedure:
   a. Select the TeamConnection Database.
   b. Select **File->Database->New Copy**. This displays the **Copy Database** window.
   c. The **Server** field should contain the Local value. You can supply the **Title** of your choice.
   d. Rename the file in the **File Name** field to match the naming conventions of your organization. This copy of the template must keep the .nsf suffix. For example: development.nsf.
   e. Select the folder icon button, which will open the **Choose a folder** window. Choose the directory on your local file system in which you want to store the database.
   f. In the **Copy** field select the **Database design and documents** radio button, and leave the **Access Control List** check box checked so that the new database will inherit those characteristics from the empty template database.
   g. Select **OK**.
2. Select the local database copy that you have just created from your workspace.
3. From the **File** menu, select **Database->Replace Design**.

4. Locate the template used to refresh the database (fhcnotes.ntf - TeamConnection Template 3.0.2) or (teamcweb.ntf - TeamConnection 3.0.3) in the **Replace Database Design** window. If the template is not on your Local server, which primes the list of template choices, you may have to select the **Template Server** button to locate the desired template.

   Important notice to administrators:

   Make sure that you select the right template. An inappropriate template can corrupt your database!
5. Select **Replace**.
6. Test the refreshed database copy extensively by performing the following activities:
   - Create documents.
   - Update documents. This should include testing the state system, which would involve moving a document's state to Approved.
   - Reopen documents.
   - Select **Feature** and **Defect** push buttons.
   - Open Defects and/or Features.
   - Examine views.
7. When you are satisfied that you successfully refreshed the local database copy, select the production (original) database from your workspace and repeat the refresh process, beginning with step 3.

   Important notice to administrators:

   Make sure that you select the right template. An inappropriate template may have a destructive impact on your database!
8. You should test the refreshed database extensively before notifying users that it is in production mode.

Migrating the Database (3.0.3. only)

TeamConnection Integrated Notes ships with a migration utility that enables you to migrate documents to release level 3.0.3. To migrate documents, you should first update the setup document and then migrate the documents.

Updating Setup (3.0.3 only)

Before you migrate your documents you need to manually update the setup document to properly access the TeamConnection family server. To do this, complete the following steps:

1. Double-click on the database icon and select **Setup the Database**.
2. Under the Customization section, open the section called ″Modify TeamConnection Access″.
3. Open the section called ″Modify or Refresh Family Information″.
4. Ensure that the values for **family**, **host**, and **port** are correct.
5. Verify access by selecting the **Test Connection** button.
6. Update TeamConnection access by selecting the **Apply** button, and then selecting **OK** in the TeamConnection pop-up window.
7. Read the section called **Enable Unrestricted Agents** if you plan to access your database from a web browser.

Migrating your documents (3.0.3 only)

After you have updated your setup document, you can then migrate your documents. To migrate documents, complete the following steps:

1. Make a backup copy of your database.
2. Replace your database design using the latest 3.0.3 template (teamcweb.ntf). Refer to “Database maintenance: replacing a design from a template” on page 110 for instructions on how to do this. Note that you must have Lotus Notes Release 4.6.2 or higher to use teamcweb.ntf.
3. Double-click on the database icon, and then select **Setup the Database**.
4. Select the **Database Migration** push button and and follow the online instructions.

Notes Client vs Web Client

TeamConnection now ships with a web-based integrated notes client in addition to the standard notes client. The advantage of the web-based client is that you can use it to access any database (that you are authorized to access) using any java-based web browser.

The advantage of the standard notes client is that it currently has more function built in, such as the ability to create defects and features.

The following table summarizes most of the differences between the standard Notes client and the web-based client:

Table 1. Differences between Note client and Web-based client

Standard Notes Client	**Web-Based Client**
Database Setup	Not supported. Use Notes client.
Create, update, archive, and delete documents. Circulate documents for review. Project Leader special functions.	Same capabilities
Open new defects and features from Notes or associate existing defects and features. Refresh defect and feature documents with latest TeamConnection family server information.	Associate existing defects and features with Notes documents (Defects and features can be opened using the TeamConnection web client. Refresh performed using periodic reconcile function.
Rich document editing capability	Limited editing capability. Primarily text based documents. Ability to attach documents containing rich text. Creation of document links not available.
Dependencies: Users must have Lotus Notes installed and the TeamConnection classic client installed on their workstation.	**Dependencies:** Users can access Notes Integration Databases and TeamConnection through a Web browser
Users must return to a view to select a different view.	Available views are listed at the end of every document.
When entering names, Notes prompts the full address	Users must input full Notes address, such as Name/Location/Organization

Table 1. Differences between Note client and Web-based client (continued)

Standard Notes Client	Web-Based Client
Test Case database only: Assisted creation of Execution records.	Not supported.

Making the database available on the web

If you have created your database on a Note server, you can access that database using a web browser by entering the following URL:

```
http://domino_host_name/dbname
```

where *domino_host_name* is the name of your Domino (Notes) server, and *dbname* is the name of your database.

For example, if your Domino host name is *myserv.stl.ibm.com* and your database name is *mybase.nsf*, you can access the database using the following URL:

```
http://myserv.stl.ibm.com/mybase.nsf
```

Before you can access a database in the manner described above, you might have to perform some administrative tasks. In general, these tasks are the same as those required to make any Notes database available from a web browser. Information on this can be found in the Domino User's Guide and in the Domino Administration Help. In general, the following conditions must apply:

1. The Domino server must be running.
2. If all databases are to be made available for browsing, the security settings in the Server document must permit anonymous connection and HTTP client access to databases. Even if the list of databases is not available to all users, authorized users can still access any database for which they are authorized via a URL of the form:

   ```
   http://server:port/directory/databaseName.nsf/?OpenDatabase
   ```

   where

   `server`

   is the domino server name,

   `port`

   is the TCP/IP port used by the server (optional if only one server is present),

   `directory`

   is the directory where the database resides, and

   `databaseName.nsf`

   is the file name of the database.
3. Depending on security requirements for the database, you might need to modify setting in the Access Control List If access is to be restricted, then Default access (which effects both Notes and web browser access) and Anonymous access (which effects only web browser access) can be adjusted.
4. Your server document should allow restricted LotusScript/Java agents to run. Normally a value of asterisk (*) is defined in the field Run restricted LotusScript/Java agents in the Agent Manager section of the server document.

5. The default launch for the database must be adjusted from the database properties in Notes. The launch should be changed from ″Open Designated Navigator in its own window″ to ″Restore as last viewed by user.″ This allow the Notes client users to select the navigator they wish and have it restored when they return to the database. Web browser users will always see the equivalent of the Notes navigator when they open the database. In Notes 4.6.2, a separate launch can be specified for the web browser. This should be set to ″Use Notes launch option″ in the scheme above. Another option is to set it to open the first document in a particular view.
6. A list of views is available at the end each document. This assists with general navigation. The hierarchy views will show the subordinate documents.

If you are using the 3.0.2 template

If you are using the 3.0.2 database/template (teamc.nsf/fhcnotes.ntf) the agent named ShowDocLinks might need to be resaved on the Domino server on which the database resides. This agent is run whenever a document is opened from a web browser.

Miscellaneous Hints and Tips

AIX considerations

If you receive the message, ″Error in loading DLL″, perform the following steps:
1. Edit your .profile in your home directory.
2. Find the line containing the LIBPATH.
3. Add :/usr/lib to the LIBPATH.
4. Run your .profile.

Renamed defects and features

Defects and features that are opened from a Notes database and subsequently renamed can no longer be found by the Notes database. If a defect or feature is renamed, you must do the following to allow access from Notes:
1. Delete the Notes document representing the feature or defect.
2. Use the ″associate defect/feature″ function to reestablish the defect/feature in the Notes database.

Bypassing the main panel

If you want to bypass the main panel, open the database properties and go to the **Launch** tab. In the field labeled **On Database Open**, set it to the value **Restore** as last viewed by the user.

Part 4. Using TeamConnection to build applications

This section tells how to install and use the TeamConnection build function.

Though build administrators will be most interested in this section, anyone who builds an application using TeamConnection will find the first and last chapters helpful.

Chapter 9. Basic build concepts

This chapter defines terms and briefly describes the TeamConnection pieces that work together in building an application. For more details, continue to the other chapters in this section.

The TeamConnection build function has numerous features:

- It builds applications for platforms in addition to those it runs on. Currently you can build applications using TeamConnection on the following platforms: AIX, HP-UX, Solaris, MVS, MVS/OE, OS/2, Windows NT, and Windows 95.
- Its graphical representation of the structure of an application makes it easier to visualize and change.
- It lets you build an application using any number of machines working in parallel.
- Because it is fully integrated with TeamConnection's version control system, it ensures that the correct versions of parts are used in a build.
- It can work not only with parts that represent files, such as C source files, but also with parts that represent objects, such as VisualAge Generator applications.
- It can manage other steps related to software packaging and distribution.

  For more information, see "Part 5. Using TeamConnection to package products" on page 177.

The physical structure of the build function

Figure 47 shows the structure of the TeamConnection build function:

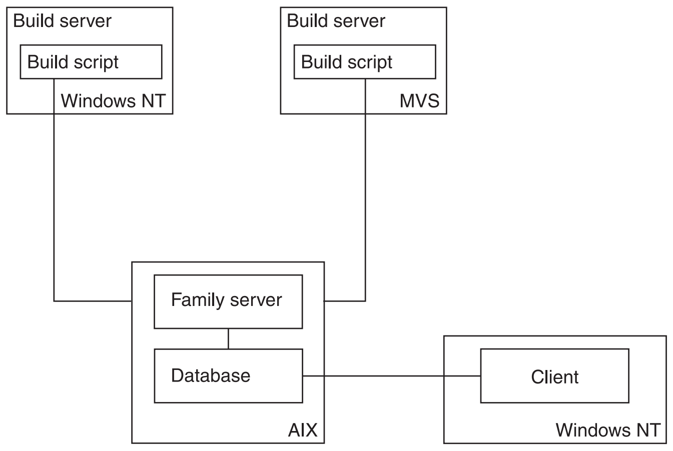


Figure 47. The physical structure of TeamConnection

Build servers are started by a TeamConnection administrator. For more information, see "Chapter 10. Installing, starting, and stopping build servers" on page 125.

The build object model

Figure 48 on page 120 shows the TeamConnection objects and events that constitute the build function, as illustrated in a sample application named msgcat.exe. This build object model is a conceptual model of the build function. When you use TeamConnection to define a build, you work with a *build tree* (a simplified graphical illustration of the build object model), which you can access through the TeamConnection GUI. “Working with a build tree” on page 121 explains build trees. This section explains the build objects and events represented in a build tree.

In TeamConnection, the build function is always described and discussed in terms of the final output of the build: the product or executable file that the build produces. For the sample application shown in this illustration, msgcat.exe is the build output and appears at the top of the build object model and as the top branch of the build tree illustrated on page 120. When you want to actually build the product, you request a build of msgcat.exe. TeamConnection uses the build tree that you define for this product to determine which objects and build events it needs to generate the final output. The objects and events that TeamConnection uses for a build include the following:

TeamConnection part
An object produced or used during a build, containing any data produced or used by the build. For example, a part called hello.c contains the source code for the application called msgcat. A part might be a text or binary file, or an object such as a VisualAge Generator generic collector.

Build event
An individual step in the build of an application, such as the compiling of hello.c into hello.obj.

A specific build request typically contains many build events. For example, if you start a build of an entire application, TeamConnectioninitiates build events for each compile and link operation.

Build requests are processed in a round-robin fashion for each TeamConnection family involved in a build. Build events are initiated in the order that that they are received by each build machine involved in the build request, sometimes in parallel.

Build event processing is internal to TeamConnection; you cannot interact with these processes directly.

Builder
An object that can transform a build event’s input parts into output parts by calling tools such as linkers or compilers. For example, one builder might know how to transform the input part hello.c into the output part hello.obj. A different builder might know how to transform hello.obj into msgcat.exe. Builders are associated with the parent, or output part, rather than the child, or input.

Build script
An object that a builder uses in transforming inputs to outputs; it is essentially a binding between TeamConnection and a transformation tool, such as a linker or compiler. In OS/2, Windows, UNIX, or MVS/OE environments, a build script is usually a command file, but it can be a string that calls the tool. In MVS, it is a file containing JCL-like instructions.

Parser

A tool that can read a source file and report back a list of dependencies of that source file. It frees a developer from knowing the dependencies one part has on other parts to ensure a complete build is performed. For example, a C parser can read a C source code file and report back a list of the files included by the source file or by the included files.

TeamConnection will re-verify all parser dependencies:

1. When the user creates or checks in the part, TeamConnection will add all parser dependencies that it can find.
2. During build, TeamConnection will again check all parser dependencies and update as needed.

Parent-child relationships in a build tree

One relationship that is important to understand and distinguish is the relationship between parent and child parts in a build tree.

Though parent-child relationships usually imply that the parent part generates the child part, in a TeamConnection build it is the opposite. Because TeamConnection places the build output at the top of the tree, it refers to the build output as the parent and to the build input as the child.

A child part can be related to a parent part one of three ways: it can be an input part, an output part, or a dependent part.

Input parts

A part used as direct input to your build. An example of this is a C language source part. If you start a build and this part has changed, the changed part will be part of the new build.

Output parts

A generated output from a build, such as an OBJ or EXE part, or a part with no contents that serves as an organizer object. If you start a build and this part has changed, the changed part will be included in the new build.

Dependent parts

A part needed for the build operation to complete but that is not passed directly to the compiler. An example of this is an include part. If you start a build and this part has changed, the changed part will be included in the new build.

Though parent-child relationships usually imply that the parent part generates the child part, in a TeamConnection build it is the opposite. Because TeamConnection places the build output at the top of the tree, it refers to the build output as the parent and to the build input as the child.

To understand how build output is generated, it may be easier to start at the bottom of the build object model and work your way up. In Figure 48 on page 120, hello.h and bye.h are C source files that are embedded in hello.c and bye.c, respectively. The parser, parser1, is able to read hello.c and bye.c to determine files they embed. This build object model contains three build events:

- The builder compiler1 compiles hello.c into hello.obj.
- The builder compiler1 compiles bye.c into bye.obj.
- The builder linker1 links hello.obj and bye.obj into msgcat.exe

This build object model contains the following parent-child relationships:

- msgcat.exe is the parent of hello.obj and bye.obj.
- hello.obj is the parent of hello.c
- bye.obj is the parent of bye.c

You establish these parent-child relationships between parts when you create the parts in TeamConnection.

Before you can build msgcat.exe, for example, you need to create a place-holder part for it and designate linker1 as its builder. You then create place-holder parts for hello.obj and bye.obj and designate compiler1 as their builder and msgcat.exe as their parent.

"Creating the build tree for the application" on page 165 walks you through an example of creating the build tree for this object model.

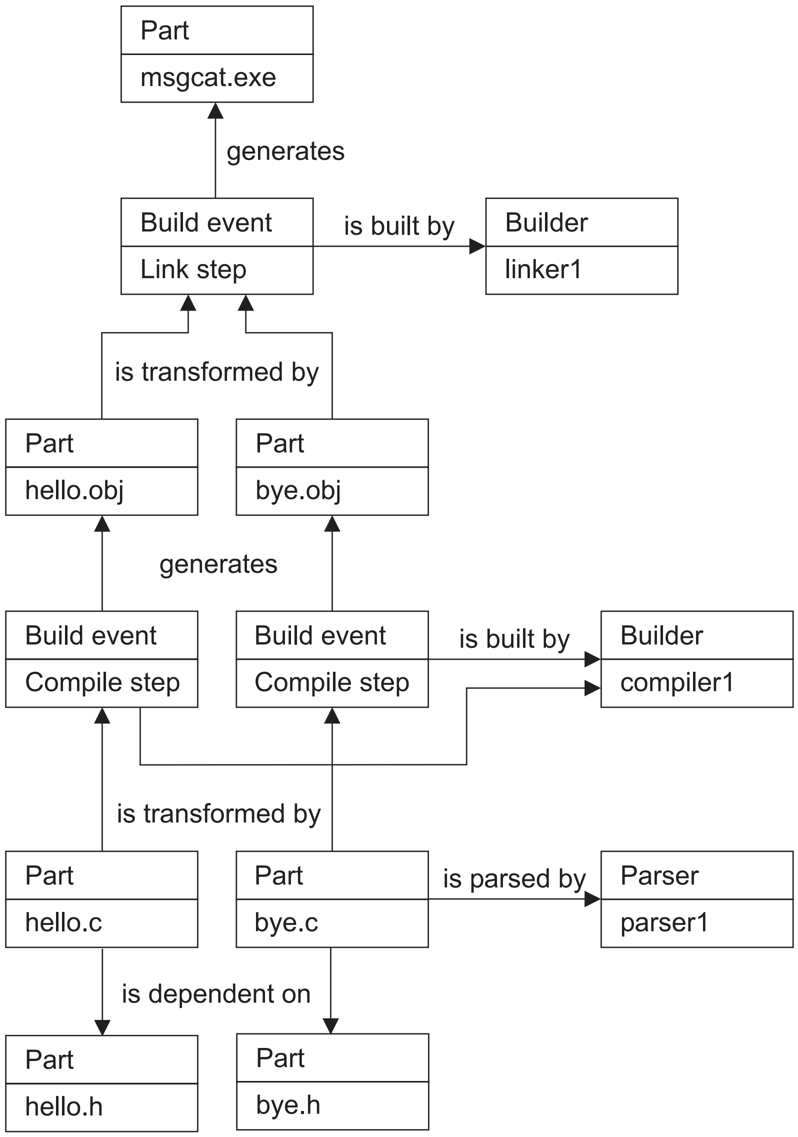


Figure 48. Sample build object model for msgcat.exe

Working with a build tree

Software developers must provide the information by which TeamConnection determines the build events that make up a build request. An application's build tree shows this information graphically.

A build tree is a simplified version of the build object model, showing the dependencies that the parts in an application have on one another. If you change the relationship of one part to another, the build tree changes accordingly. Figure 49 shows a build tree for the hello application:

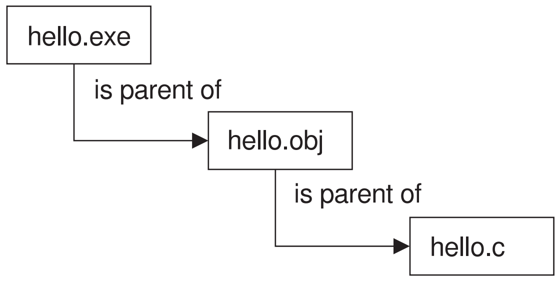


Figure 49. The build tree for the hello application

In this simple application, hello.c is compiled to create hello.obj, which in turn is linked to create hello.exe. The build tree shows that hello.exe is the parent of hello.obj, which in turn is the parent of hello.c. To build the entire application, you request to build hello.exe.

Just as the parts that make up an application are versioned, the relationships between these parts are versioned. Thus, more than one version of the build tree can exist. For example, Figure 50 on page 122 shows two different versions of the same build tree:

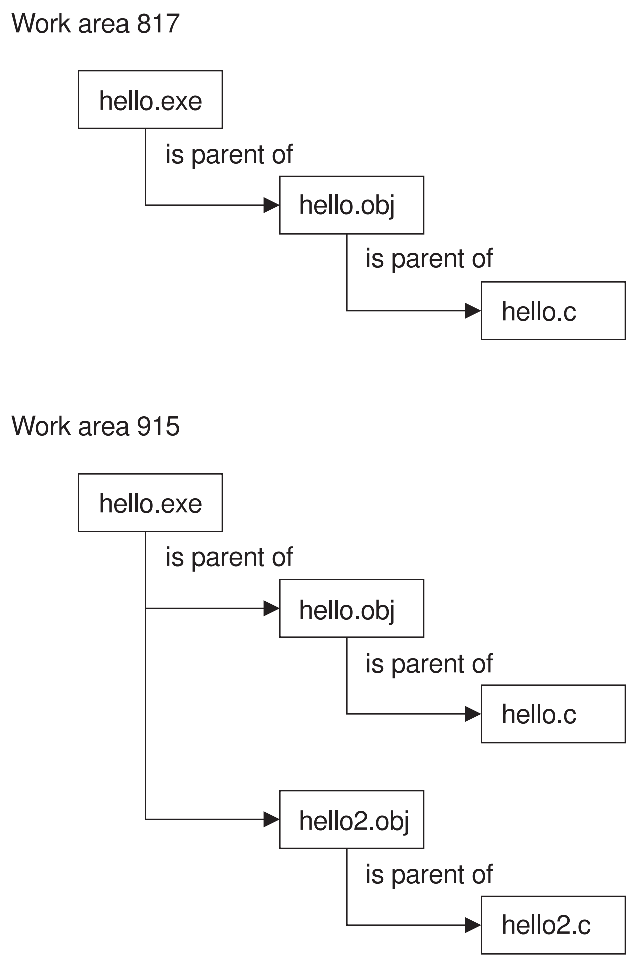


Figure 50. Two versions of a build tree

Putting the pieces together

The table that follows lists the tasks involved in preparing for building an application and in actually building it. Usually an administrator does the preparation steps, but anyone with the proper authority can do so.

For more information about this task,	**Go to this page.**
Creating build startup files	“Creating build startup files (for non-MVS environments)” on page 131
Starting build servers	“Chapter 10. Installing, starting, and stopping build servers” on page 125
Stopping build servers	“Stopping the build servers” on page 132
Writing a build script	136
Creating a builder	133
Creating a parser	157

For more information about this task,	Go to this page.
Defining a build tree	165
Starting a build	169
Stopping a build	174
Verifying the parts to be built	173

Chapter 10. Installing, starting, and stopping build servers

This chapter explains how to install, start, and stop a build server.

Installing the build function

Before installing build servers, you should be familiar with the build concepts found in “Chapter 9. Basic build concepts” on page 117.

For hardware requirements for the build server machines, refer to the requirements for your specific operating system in the *Installation Guide*.

When you install the various parts of TeamConnection, you can choose to group them on a single server machine, or you can distribute them in various combinations. For example, Figure 51 is a configuration that shows each component on a different machine:

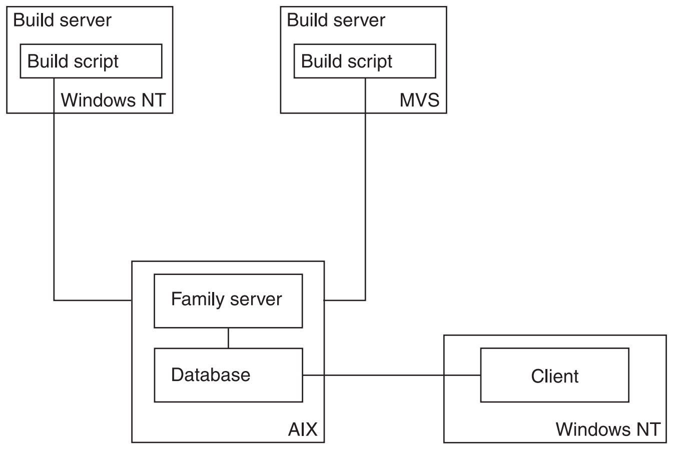


Figure 51. TeamConnection components on separate machines

The TeamConnection installation programs allow you to select specific components to install. You can use this program to install a build server, with or without other TeamConnection components. For instructions on installing TeamConnection, refer to the installation chapter for your platform in the *Installation Guide*.

Creating a build server on MVS

The code for an MVS build server is shipped with TeamConnection and installed on your TeamConnection server when you install the product.

The following are software requirements for the MVS build server:

- TCP/IP Version 3.2 for MVS
- OS/390 R3 LE

To install the build server on MVS, you create MVS data sets and then upload the TeamConnection files to these data sets. Follow these steps:

1. On your TeamConnection server, install the MVS build server component, following the instructions in the *Installation Guide*.
2. On MVS, create data sets with the following characteristics:

- An object data set [1] to contain object files: LRECL=80, RECFM=FB, BLKSIZE=3120, DSORG=PO (Approximate size: 15 cylinders on a 3390.)
- A load module data set [2] to contain TEAMCBLD and DLLs: LRECL=80,RECFM=U, BLKSIZE=3120, DSORG=PO (Approximate size: 15 cylinders on a 3390.)
- A data set to contain JCL for creating load modules [3]: LRECL=80, RECFM=VB, BLKSIZE=3120, DSORG=PO
- A JCL data set for the MVS build scripts and samples [4]: LRECL=80, RECFM=VB, BLKSIZE=3120, DSORG=PO
- An environment variable data set for the EDCENV DDname in runpgm.jcl [5]:LRECL=80, RECFM=VB, BLKSIZE=3120, DSORG=PS (optional, needed at runtime)

3. From the mvs subdirectory where TeamConnection is installed, do the following to upload the files to MVS:
   a. Type the following at a prompt and press Enter: `ftp` *hostname*. Specify your name and password, if required.
   b. Type the following, and press Enter after each:
      - `binary`
      - `cd` *`data set for object code`* [1]
      - `put fhccmnc.mvs fhccmnc`
      - `put fhcrscli.mvs fhcrscli`
      - `put teamcbld.mvs teamcbld`
      - Issue only one of the following commands according to the platform from which you are transferring the MVS files:

        From Intel platforms:

        ```
        put \$LANG\fhbmsg.mvs fhbmsg
        ```

        From UNIX platforms:

        ```
        lcd $LANG
        put FHBMSG.MVS fhbmsg
        ```

        Note: For *$LANG*, substitute the name of the language subdirectory, for example, enu for US English.
      - `put fhbtclnk.mvs fhbtclnk`
      - `put getdsn.mvs getdsn`
      - `ascii`
      - `cd` *`JCL data set for load module`* [3]
      - `put fhblink.jcl fhblink`
      - `put runpgm.jcl runpgm`
      - `put runpgmt.jcl runpgmt`
      - `put fhbmenv.var` *`environment variable data set`* [5]
      - `quit`
   c. This optional step installs the sample build scripts on MVS. It must be done from a machine that has the TeamConnection client installed.

      Note: The details in this step are subject to change. Please refer to updated documentation when attempting this in the future.

      If you are doing these steps from a different machine than in the previous step, repeat step *3a* from the bin subdirectory where the client is installed on this machine. Otherwise, change to the bin subdirectory where the client is installed. Then, type the following statements and press Enter after each:

- `ascii`
- `cd '`*`data set for teamproc jcl`*`'` [4]
- `put fhbmc.jcl fhbmc`
- `put edcc.jcl edcc`
- `put fhbmasm.jcl fhbmasm`
- `put fhbplked.jcl fhbplked`
- `put fhbcobm.jcl fhbcobm`
- `put fhbmpli.jcl fhbmpli`
- `put fhbtclnk.jcl fhbtclnk`
- `put fhbm370.jcl fhbm370`
- `put fhbmlink.jcl fhbmlink`
- `quit`

d. From MVS, do the following:
   1) Modify fhblink to customize the JCL to your MVS system.
   2) Submit this JCL to create the required load modules.

      When this JCL is executed successfully, the following modules are created:
      - TEAMCBLD (main executable)
      - FHBMSG (national language-specific message catalog DLL)
      - FHCCMNC (supporting DLL)
      - FHCRSCLI (supporting DLL)

      Note: All except for FHCCMNC are re-entrant.
   3) The object dataset [1] can be deleted.

Creating a build server on MVS/OE

The steps that follow describe how to install a build server on MVS/OE.

1. Install the MVS/OE build server component.
2. From the **oe** subdirectory where TeamConnection is installed, use ftp to upload the following files to MVS/OE, in binary.
   - rename teamcbld.oe to teamcbld (main executable)
   - rename fhccmnc.oe to fhccmnc (supporting DLL)
   - rename fhcrscli.oe to fhcrscli (supporting DLL)
   - teamcv3.cat (for required national language)
3. Using chmod in the OE shell, set the access flags for teamcbld, fhccmnc, and fhcrscli so that they are executable.

Starting build servers using teamcbld

You can start the build servers using the following line command:

```
teamcbld  [-c buildDir] [-e environment]
   [-p pool] [-f family]
   [-u userID] [-l login_password]
   [-k local_codepage] [-r remote_codepage]
   [-b become] [-s] [-n]
```

Where:

- *buildDir* specifies the directory `teamcbld` should be run in.
- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive. You can also set this value using the TC_BUILDENVIRONMENT environment variable.
- *pool* is the name of the build pool. You can also set this value using the TC_BUILDPOOL environment variable. This value is case sensitive, and should match the parameter specified in the Part -build command.
- *family* is the name of the TeamConnection family. If a *family* is not specified, the value of the TC_FAMILY environment variable is used as the default.
- *userID* is used only when the family server authentication level requires a TeamConnection user ID and password. A password parameter is not necessary when TC_LOGIN has been used.
- *login_password* is used only when the family server authentication level requires a TeamConnection user password. A password parameter is not necessary when the command line `tclogin` command has been used.
- *local_codepage* local machine's code page used for translation, if not specified, no codepage conversion will be done.
- *remote_codepage* family machine's code page used for translation, if not specified, no codepage conversion will be done.
- *become* is a user become value. The default is the value specified for the TC_BECOME environment variable.
- **-s** sends log file messages to the screen.The build server generates a log file called teamcbld.log. Build server log messages can be routed to the screen using the **-s** parameter.
- **-n** writes files:
  - writes the changed environment variables to a file called **tcbldenv.lst** instead of setting them in program's environment. The format of the file is `variable=value`.
  - writes the list of input files to a file called **tcbldin.lst**. One file per line, format is `pathName type`.
  - writes the list of output files to a file called **tcbldout.lst**. One file per line, format is `pathName type`.

You can also set the **-s** and **-n** build options using the TC_BUILDOPTS environment variable.

Two environment variables that directly affect build performance are:

- TC_BUILDMINWAIT - Minimum amount of time to wait (in seconds) between queries for new jobs. Default setting is 5, minimum setting is 3.
- TC_BUILDMAXWAIT - Maximum amount of time to wait (in seconds) between queries for new jobs. Default setting is 15, maximum setting is 300.

The command `teamcbld` will check the family for work to do every TC_BUILDMINWAIT. If it is not busy it will slow down to the TC_BUILDMAXWAIT time. The build administrator can adjust these variables to tune the frequency that the build server will check the family.

Starting an MVS build server

The MVS build server is a special client to the family server. It uses a user ID that must be defined to the family server. Depending on the authentication level in effect, a host list entry must be created for the user, and, if password authentication is required, the -l parameter must be used to specify the password when starting the build server.

The user ID used by the build server is the TSO user ID under which the build job is started. The user ID must always be in uppercase and must be created in uppercase on the family server. The user ID is not determined by the TC_USER environment variable.

You can start the MVS build server program TEAMCBLD, in Batch or under TSO. The RUNPGM JCL executes the MVS build server in batch. The RUNPGMT JCL executes the MVS build server under a TSO environment.

The following summarizes the actions to run the build server on MVS:

Modify the RUNPGM JCL for your installation.

Modify the environment variable dataset as needed. Two environment variables that directly affect build performance are:

- TC_BUILDMINWAIT - Minimum amount of time to wait (in seconds) between queries for new jobs. Default setting is 5, minimum setting is 3.
- TC_BUILDMAXWAIT - Maximum amount of time to wait (in seconds) between queries for new jobs. Default setting is 15, maximum setting is 300.

The `teamcbld` will check the family for work to do every TC_BUILDMINWAIT. If it is not busy it will slow down to the TC_BUILDMAXWAIT time. The build administrator can adjust these variables to tune the frequency that the build server will check the family.

Submit the JCL.

To start an MVS build server, do the following to modify the RUNPGM JCL:

1. Add a job card.
2. Modify the STEPLIB DD statement to point to the data set that contains the following load modules: TEAMCBLD, FHBMSG, FHCCMNC, and FHCRSCLI.
3. Modify the TEAMPROC DD statement to point to the data set that will contain all your MVS build scripts.

   Note: The TEAMPROC DD defines the dataset into which the build script from TeamConenction family will be placed during the build. If the builder is of type 'none', then the build script does not exist in the family, but must already be in this dataset. If the build script refers to other build scripts (using the EXEC card), then those scripts are expected to be in this dataset also. However an optional ddname, PROCLIBS, can be defined which will be searched for the referred build scripts which are not found in TEAMPROC.
4. Modify the EDCENV DD statement to point to the data set that contains the environment variables.
5. Modify the following statement. For RUNPGMT, the statement is equivalent. Parameters can be specified using the PARM field or, in some cases, environment variables.

```
//RUNPGM EXEC PGM=TEAMCBLD,
// PARM='ENVAR("_CEE_ENVFILE=DD:EDCENV")/[-e environment]
//     [-p pool] [-f family] [-u unit_name]
//     [-l login_password] [-k local_codepage]
//     [-r remote_codepage]'
```

Where:

- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive. You can also set this value using the TC_BUILDENVIRONMENT environment variable.
- *pool* is the name of the build pool. You can also set this value using the TC_BUILDPOOL environment variable. This value is case sensitive, and should match the parameter specified in the Part -build command.
- *family* is the name of the TeamConnection family. If a *family* is not specified, the value of the TC_FAMILY environment variable is used as the default.
- *unit_name* indicates the default unit type for dynamic data set allocations. VIO is the default.
- *login_password* is required only when the family server authentication level requires a TeamConnection user password.
- *local_codepage* indicates the code set that text data is converted to for the build. The default is IBM-1047.
- *remote_codepage* indicates the code set for data stored in TeamConnection. The default is IBM-850.

Notes:

a. TEAMCBLD converts the text files between the two codepages using the **iconv** codepage conversion functions, and therefore can support only those codepages that have the iconv conversion tables installed on the MVS system. If not specified, TEAMCBLD uses internal tables which correspond to single-byte codepages IBM-1047 and IBM-850.
b. The -k (*local_codepage*) and the -r (*remote_codepage*) flags are also used to convert any messages generated in the MVS environment which are sent back to the workstation client.

6. Submit the job.

Starting the MVS/OE build server

Under MVS/OE, the build server program can be started with the following teamcbld command:

```
teamcbld -e environment -p pool -f family [-l login_password] [-s] [-v] [-n]
  [-k local_codepage] [-r remote_codepage]
```

Where:

- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive. You can also set this value using the TC_BUILDENVIRONMENT environment variable.
- *pool* is the name of the build pool. You can also set this value using the TC_BUILDPOOL environment variable. This value is case sensitive, and should match the parameter specified in the Part -build command.

- *family* is the name of the TeamConnection family. If a *family* is not specified, the value of the TC_FAMILY environment variable is used as the default.
- **-s** sends log file messages to the screen.The build server generates a log file called teamcbld.log. Build server log messages can be routed to the screen using the **-s** parameter. **-s** must be lowercase. An uppercase **-S** turns it off.
- **-v** increases the level of messages written to the log. This option is equivalent to the verbose option on TeamConnection teamc commands **-v** must be lowercase. An uppercase **-V** reduces the level.
- *login_password* is used only when the family server authentication level requires a TeamConnection user password. The user ID used by the build server is the MVS/OE user ID under which the build job is started.
- *local_codepage* indicates the code set that text data is converted to for the build. The default is IBM-1047.
- *remote_codepage* indicates the code set for data stored in TeamConnection. The default is IBM-850.

Notes:

1. The command `teamcbld` converts the text files between the two codepages using the iconv codepage conversion functions, and therefore can support only those codepages that have the iconv conversion tables installed on the MVS system. If not specified, `teamcbld` uses internal tables which correspond to single-byte codepages IBM-1047 and IBM-850.
2. The -k (*local_codepage*) and the -r (*remote_codepage*) flags are also used to convert any messages generated in the MVS environment which are sent back to the workstation client.

TC_CATALOG, if specified, should be the pathname of the message catalog file. For example, */xxx/enu/teamcv3.cat*. If not specified, default (English) messages are used. The file teamcbld must be included in the PATH environment variable. The files fhcccmnc and fhcrscli must be included in LIBPATH.

Two environment variables that directly affect build performance are:
- TC_BUILDMINWAIT - Minimum amount of time to wait (in seconds) between queries for new jobs. Default setting is 5, minimum setting is 3.
- TC_BUILDMAXWAIT - Maximum amount of time to wait (in seconds) between queries for new jobs. Default setting is 15, maximum setting is 300.

The command `teamcbld` will check the family for work to do every TC_BUILDMINWAIT. If it is not busy it will slow down to the TC_BUILDMAXWAIT time. The build administrator can adjust these variables to tune the frequency that the build server will check the family.

Creating build startup files (for non-MVS environments)

You can create startup files for concurrently starting build servers with the family server using the teamcd command. This is the preferred method for starting build servers. When starting the build servers in this manner, you need to create a startup file.

Information about the build servers is put in a startup file and the name of the startup file is specified in one of two ways:
- In the teamcd command, using the -b parameter.
- In the TC_BUILD_RSSBUILDS_FILE environment variable.

You can store the build startup files wherever you like, provided that you give the full file path names for them in the -b parameters, or in the TC_BUILD_RSSBUILDS_FILE environment variable.

Stopping the build servers

To stop a build server, do one of the following:

- Close the window in which the build server is running.
- Press Ctrl+C when the build server window has focus.
- Close the window in which the family server was started if the build server was started with the teamcd command.

Stopping an MVS build server

To stop an MVS build server, cancel the RUNPGM job that was used to start it.

Chapter 11. Working with build scripts and builders

A builder is an object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers. For example, one builder might transform a COBOL source file into an object file. Another might transform a set of object files into an executable file. Builders use build scripts to invoke the tools that actually transform TeamConnection parts.

Usually a build administrator creates build scripts and builders, but anyone with the proper authority can do so.

This chapter tells how to create and maintain build scripts and builders. It assumes that you have read “Chapter 9. Basic build concepts” on page 117. The following table directs you to the tasks to be done:

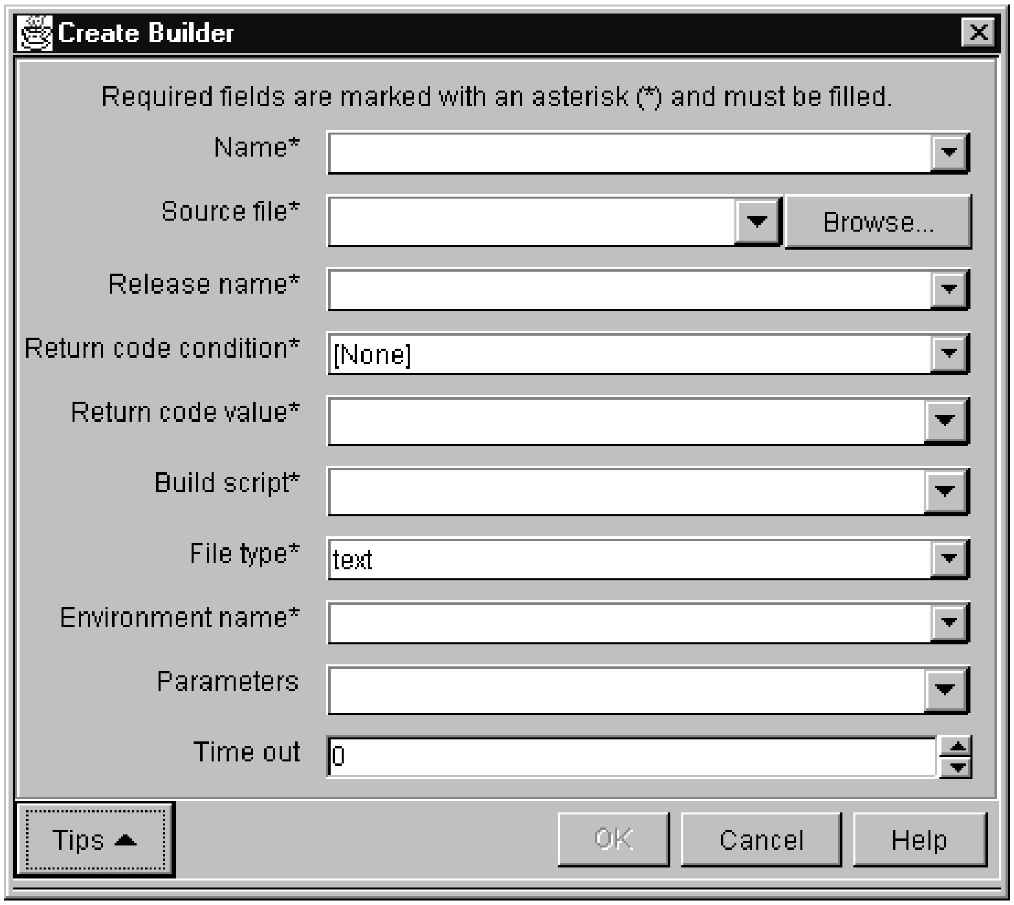
For more information about this task,	**Go to this page.**
Creating a builder	133
Writing a build script	136
Testing a build script	140
Updating a builder	140
Putting a builder to work	140
Removing a builder from a part	141

Creating a builder

As with most other TeamConnection operations, there are two ways you can create a builder: using the graphical user interface (GUI) or the command line interface.

To create a builder using the GUI:

1. From the Actions menu, select **Builder → Create**.
2. On the Create Builder window, type the requested information.



Create Builder
Required fields are marked with an asterisk (*) and must be filled.

Name*	
Source file*	
Release name*	
Return code condition*	[None]
Return code value*	
Build script*	
File type*	text
Environment name*	
Parameters	
Time out	0

Browse...
Tips
OK
Cancel
Help

Figure 52. Create Builder window

To create a builder using the command line:

From a command line, type the `teamc builder -create` command and press Enter. The complete command syntax is the following:

```
teamc builder -create name -condition RC_expression
        -environment name
        -from script_filespec
        -script name
        -value RC_value -release name
        -family name
        [-text | -binary | -none]
        [-parameters Parameters]
        [-timeout number] [-become user_name]
        [-verbose]
```

No matter which way you create a builder, you must specify a number of attributes for it. Together with the contents of the build script and the tools you use (the compilers, linkers, and so on), the following attributes define how a transformation takes place.

Builder
The name of the builder must be unique within a release. It can be anything you want; we recommend you establish and follow a meaningful naming convention. An example of a builder name is `c_set_2`.

Release
This is the name of the release that contains the builder. Builders are

release-specific objects. They are not versioned within a release; therefore you can have only one version of a builder at any time in a release.

To use the builder from a previous release, you can link to a part that uses it in that release. This action copies the builder to the new release. Otherwise, you must create the builder again in the new release.

Script, File type, and Source file

These fields work together to define the build script that the builder invokes to accomplish the transformation. (The **File type** field on the GUI corresponds to `-text`, `-binary`, and `-none` in the command. The **Source file** field on the GUI corresponds to the `-from` attribute.)

- If the build script is simple enough to be expressed in one line, you can specify it in the **Script** attribute when you create the builder, and specify a file type of `none`. At minimum, the script must specify the name of the transformation tool. For example, to invoke the C Set/2 compiler, you might specify these values:
  - **File type** - `none`
  - **Script** - `icc`

  See "Writing a simple build script" on page 138 for more information.
- If the build script is more complex, you must first create a separate file containing it; see "Writing an executable file for a build script" on page 139 for more information about how to write it. Specify the fully qualified path name of your file as the source file, and specify the file type as `text` or `binary`. TeamConnection can also detect the file type and store it in the proper format.

  When the builder is created, this source file is stored as part of the builder in the TeamConnection database; during a build, the build server creates and runs a local version of this file. Specify the name you want for this local file in the **Script** field. For example, you might specify these values:

  - **File type** - `text`
  - **Script** - `c_compile.cmd`
  - **Source file** - `c:\src\c_compile.cmd`

  When this builder is created, the contents of c:\src\c_compile.cmd are stored in the builder. When this builder is invoked, TeamConnection creates a file named c_compile.cmd, writes the build script into this file, and then runs it.
- If the builder is being used to only collect a set of build objects (for example, the **ALL:** rule in makefiles), specify these values:
  - **File type** - `none`
  - **Script** - `null`

  This prevents the build process from extracting input and output parts. See "Synchronizing the build of unrelated parts" on page 175 for an example.

Environment

This is the name of the environment supported by the builder, such as OS/2, Windows, AIX, HP-UX, Solaris, or MVS. The value that you specify here can be anything you like, but it must exactly match the environment value specified in the command used to start the build server. (See "Chapter 10. Installing, starting, and stopping build servers" on page 125 for more information.) It is recommended that you follow a naming convention for this attribute, using values such as `os2` and `mvs`.

Comparison operator and RC value

Together, these two attributes make up a Boolean expression that defines the criteria used to decide whether a specific build event was successfully accomplished, when evaluated against the value returned by the build script. (The **Comparison operator** and **RC value** fields on the GUI correspond to the `-condition` and `-value` attributes in the command.)

The values allowed for **Comparison operator** are as follows:

- **EQ** or **==** - Equals
- **LT** or **<** - Less than
- **LE** or **<=** - Less than or equals
- **GT** or **>** - Greater than
- **GE** or **>=** - Greater than or equals
- **NE** or **!=** - Not equal to

RC value can be any positive integer. An example of a Boolean expression formed from these two attributes is *return_value LE 4*, meaning that the build event is considered a success if the build script returns a value less than or equal to four.

Parameters

This is a string passed to the build script as its argument. If the string includes blanks, enclose the string in double quotes. For example, for a builder used for VisualAge C++ compiling, you might specify a parameter string of `"/Ss /Ge-"`. If the string includes a double quote, precede the double quote with a backslash (\). If the string includes a dash (-), precede the dash with a blank space, otherwise the string is interpreted as the start of a TeamConnection action flag.

Timeout

This attribute specifies the number of minutes that a build server will be allowed to complete a build event after it receives the build job from the TeamConnection family server. The default is 1440 minutes (24 hours). If the build event does not complete within this time, the build event fails.

Writing a build script

When you create a builder, you must specify a build script. The build script actually invokes the transformation tool and passes it parameters defined in the **Parameters** attribute of the builder.

Passing parameters to a build script

There are three places where parameters can be specified that affect the outcome of a build.

As attributes of a builder

Builder parameters are passed to the build script, after variable substitution is performed. Variables are substituted based upon the following syntax:

```
$(variable_name)
```

To pass parameters to your build script, specify them in the **Parameters** attribute of the builder. TeamConnection sets these variables before invoking the build script.

Note: If the command `teamcbld` included the `-n` flag, then the build script will process the tcbldenv.lst, tcbldin.lst, and the tcbldout.lst files instead of the variables set by the **Parameters** attribute.

In UNIX environments, you need to include an escape character before the **$** character, for example: `\$(TC_INPUT)`.

You can use the following TeamConnection environment variables:

TC_BUILD_USER
The user ID of the TeamConnection user who submitted the build.

TC_FAMILY
The TeamConnection family.

TC_RELEASE
The release of the parts that are being built.

TC_LOCATION
The current directory where the build script runs.

TC_INPUT
A list of the TeamConnection parts that are input to the object being built.

TC_INPUTTYPE
Identifies each input type.

Note: There is a one-to-one relationship between each object in the TC_INPUT list and this list of types (TCPart, for example).

TC_OUTPUT
A list of the parts that are being built in this build event.

TC_OUTPUTTYPE
Identifies each output type. The default is file.

Note: There is a one-to-one relationship between each object in the TC_OUTPUT list and this list of types (TCPart, for example).

TC_WORKAREA
The name of the workarea in which the build is being performed.

You can define other variables. These can be set when you start the build by specifying a value for **parameters** in the part -build command (from the command line or through the GUI). These variables are set in the parameters string passed to the build script.

These variables are also used to set environment variables before the build script is invoked.

As attributes of a part in the build tree
Parameters that are unique to a particular part are specified on the part -create and part -modify commands. Like the builder parameters, these parameters allow variable substitution.

When parameters are specified for a part, these parameters are used *in place* of the parameters specified for the builder. In other words, if both builder and part parameters are specified, the part parameters take precedence.

In addition, whenever parameters are specified for *any* part that is an output of a build event, they apply to *all* the outputs of that build event. For example, if a build event has two outputs, msg.exe and msg.map, then changing the part parameters to ″/Debug″ for either of the two parts has the same result. The next time the build event is processed, the ″/Debug″ parameter is used when invoking the build script that produces both msg.exe and msg.map.

You can also substitute the builder parameters into the file parameters by using the variable $(BUILDERPARMS). For example, you might use the following command:

```
teamc part -build myfile.c -parameters "/Ti+ $(TC_BUILDERPARMS)" ...
```

At build time, the parameters specified in the builder for myfile.c are substituted for $(TC_BUILDERPARMS).

As parameters of the part -build command

The part -build command parameters are not used the same way as the other two parameters. Instead, these parameters are used to set the values of environment variables that can be used for substitution into either the builder or part parameters. They are also set in the environment so they can be retrieved by the build script. In other words, they set up the environment used by the builder.

For example, if you issue a part -build command for msg.exe, you can specify `-parameters DEBUG=YES` and, inside of both the compile and link build scripts, retrieve the value of this variable from the environment, setting compiler or linker flags accordingly.

Writing a simple build script

This kind of build script is written into the **Script** attribute of the builder. When you create or modify the builder, you specify in this attribute the name of the transformation tool to be invoked.

For example, suppose you want to create a builder that compiles a C source file into a .exe file using IBM’s VisualAge C++ compiler. You specify the following attributes for the builder:

Build script

`icc`

Parameters

`"$(TC_INPUT) /Fe$(TC_OUTPUT)"`

You can create this builder using the following command:

```
teamc builder -create c_builder -env OS2 -script icc -none
  -parameters "$(TC_INPUT) /Fe$(TC_OUTPUT)"
```

If you use this builder to create hello.exe from hello.c, the command actually issued during the build process is the following:

```
"icc hello.c /Fehello.exe"
```

Writing an executable file for a build script

Suppose you need to build a C application and you want to specify at build time whether to use debug information. To do this, you define in the builder parameters a variable called *debug* and set the variable when you start the build. In this case, you need a build script that is a separate executable file to pass the debug parameter after the variable substitution.

For a build script of this form, you first write a program or command file; this file is stored in the TeamConnection database when you create the builder. When a build is performed, this build script file is extracted from the database and run. It interprets the parameters passed to it and then invokes the actual transformation tool, such as the compiler.

Our earlier example describes a builder that compiles a C source file into a .obj file using IBM’s VisualAge C++ compiler. Using this builder, you can specify at build time whether to use debug information. Here is the complete build script for such a builder, written in IBM’s REXX language (it could just as easily have been written in C or COBOL).

```
/* sample C Build Script using debug flag */
   parse arg parms

     environ  = 'OS2ENVIRONMENT'
     input    = VALUE('TC_INPUT',,environ)
     output   = VALUE('TC_OUTPUT',,environ)
     debug    = VALUE('DEBUG',,environ)

     if debug = 'YES' then
     do
       parms = parms || '/Ti+'
     end
     icc parms '/Fo'||output input

exit result
```

95

NT

Windows NT and 95 build scripts must be able to return a value for a return code. Because *.bat command files provide little support for programming logic and cannot return a value, use a compiled executable for your build script. TeamConnection provides two sample Windows build scripts and their source files. These samples, fhbwcomp.exe and fhbwlink.exe, are C programs for the Microsoft Visual C++ compiler and linker, respectively. Because these samples are C programs, they can also be used with the OS/2 build server with only slight modifications.

You can create the builder that invokes this build script using the following command:

```
teamc builder -create c_builder2 -script c_compile.cmd -parameters "/c"
   -from d:\teamc\c_compile.cmd
```

Where d:\teamc\c_compile.cmd is the file to be stored in the TeamConnection database and c_compile.cmd is the name of the local file that the build process creates and runs during a build.

To build hello.obj using the debug option, you use the following command:

```
teamc part -build hello.obj -parameters "debug=YES" -pool os2pool
```

The command issued by the build server is the following:

```
c_compile.cmd /c
```

In turn, the build script inspects the contents of the parameters it received in its argument list and from the environment, and it forms this command:

```
"icc /c /Ti+ /Fohello.obj hello.c"
```

Testing a build script

The easiest way to test a build script is to write a simple driver program that sets the environment variables that the build script will expect and then runs the script against local files.

For example, to test the example build script in “Writing an executable file for a build script” on page 139, write a program that sets the TC_INPUT, TC_OUTPUT, and DEBUG parameters, and then runs the command file against a local copy of hello.c. If the test is successful, the script correctly builds hello.obj in the current directory, and DEBUG is interpreted correctly.

Modifying the contents of a build script

Sometimes you need to modify the contents of a build script. Remember that a build script is stored as part of the builder itself. Because builders are not versioned, you do not check them out as you would most TeamConnection parts. Instead, follow these steps:

1. Extract the builder (in which the build script is stored) from the TeamConnection database.
2. Make your changes at your workstation.
3. Store the contents back into the TeamConnection database by using the builder -modify command.

For example, to modify the build script in “Writing an executable file for a build script” on page 139, you first issue the following command:

```
teamc builder -extract c_builder2 -to d:\build\c_builder2
```

Then, you use an editor to update d:\build\c_builder2. To move the updated build script back into TeamConnection, you issue the following command:

```
teamc builder -modify c_builder2 -from d:\build\c_builder2
```

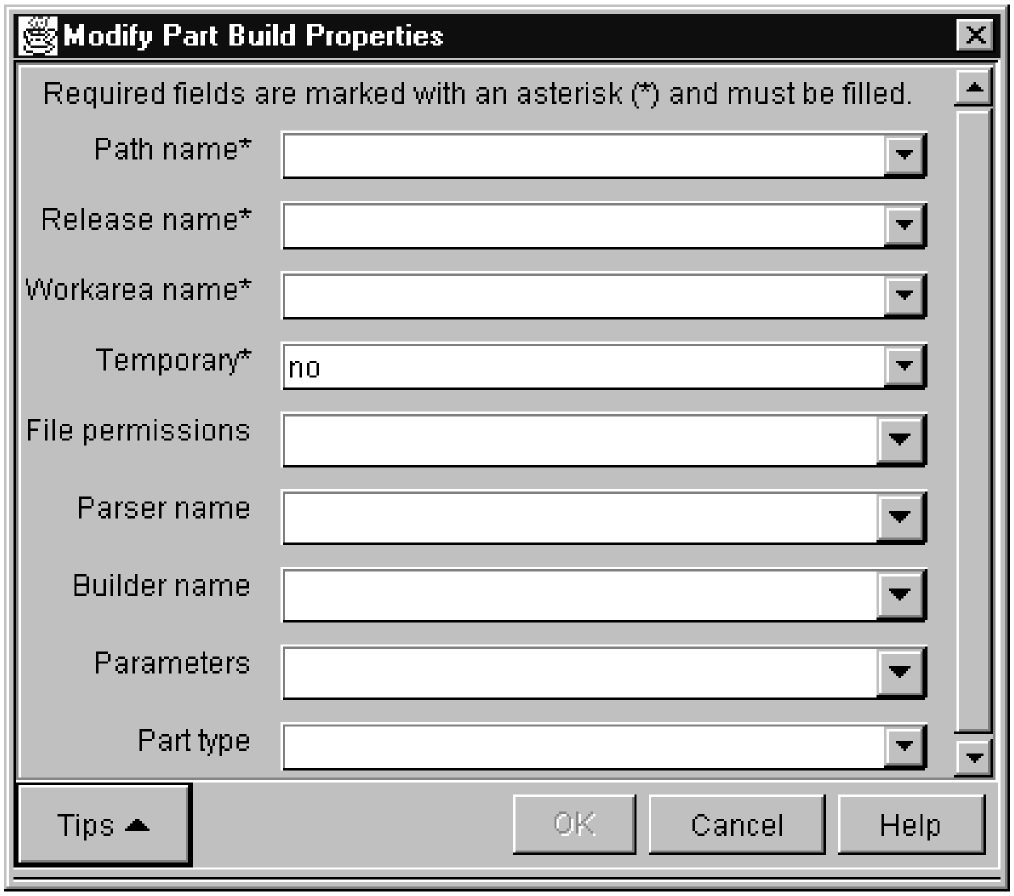
The builder is an implied dependency for any part that uses it. Therefore, the next time you build the application that uses the modified builder, all the parts that use it get rebuilt.

Putting a builder to work

For an application to use a builder, the builder must be attached to the TeamConnection parts that it will build.

For an existing part, do one of the following:

- **GUI:** From the Actions menu of the TeamConnection Home Page, select **Parts → Modify → Build Properties**. On the Modify Part Build Properties window, type the name of the builder in the **Builder** field.


Modify Part Build Properties

Required fields are marked with an asterisk (*) and must be filled.

Path name*	
Release name*	
Workarea name*	
Temporary*	no
File permissions	
Parser name	
Builder name	
Parameters	
Part type	

Figure 53. Modify Part Build Properties window

- From a command line, type the following and press Enter.

  ```
  teamc part -modify name -Builder name
  ```

  where the part *name* is the name of the output file to be created by this builder and the builder *name* is the name of the builder itself.

The complete syntax for this command is described in the *TeamConnection Commands Reference*.

You can also attach a builder to an output file when the part is created.

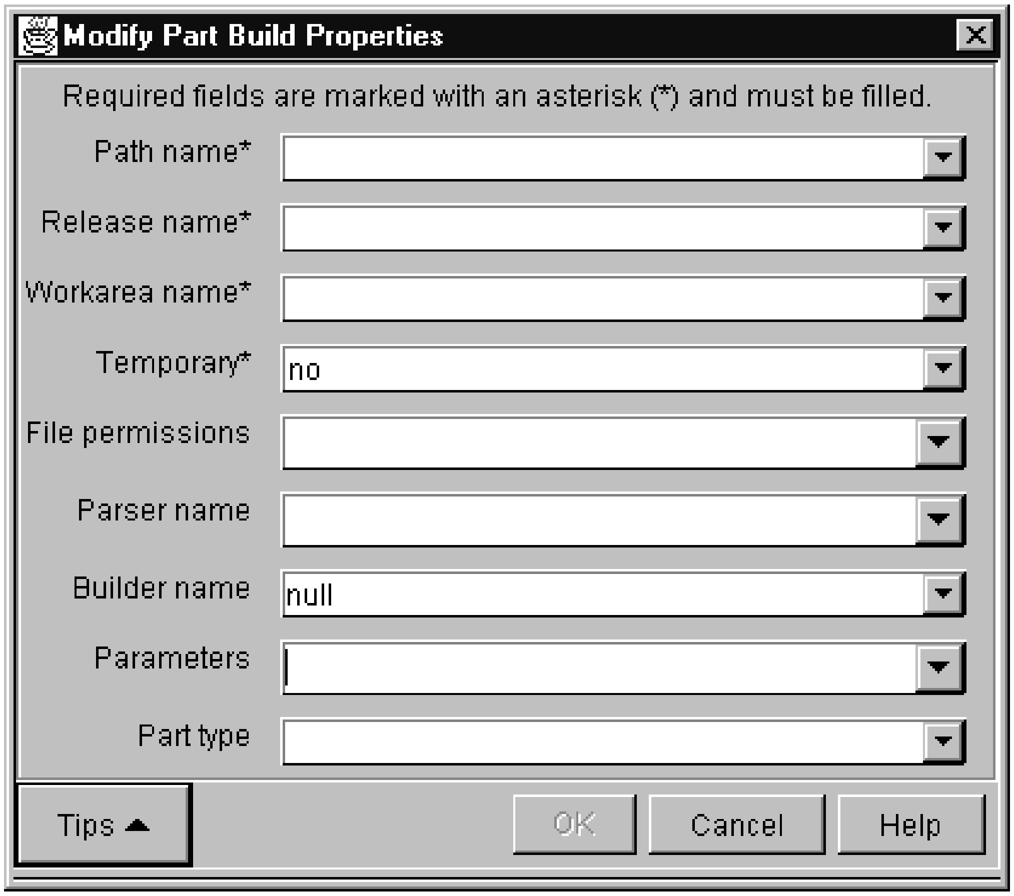
After you attach a builder to a part, the builder is ready for action. When the part is built, the builder invokes the build script, which in turn invokes a tool to transform the inputs of the part into the output.

For more information about attaching builders to the build tree, refer to “Creating the build tree for the application” on page 165.

Removing a builder from a part

If you no longer want to use a builder for a part, do one of the following:

- From the GUI, select **Parts → Modify → Build Properties** from the Actions menu of the TeamConnection Home Page. On the Modify Part Build Properties window, type `null` in the **Builder** field.



Modify Part Build Properties

Required fields are marked with an asterisk (*) and must be filled.

Path name*	
Release name*	
Workarea name*	
Temporary*	no
File permissions	
Parser name	
Builder name	null
Parameters	
Part type	

Figure 54. Modify Part Build Properties window

- From a command line, type the following:

```
teamc part -modify name -builder null -release name -family name
```

Working with VisualAge C++ and Templates

When using VisualAge C++ and templates, template-include objects are saved in a subdirectory of the current directory called TEMPINC, so that subsequent builds can use them. When you start a build from TeamConnection, you need to specify the /Ft(dir) parameter with your builder or use PRAGMA statements to update the template-include objects for subsequent builds. This parameter suppresses resolution of files and imbeds them within the object file.

You can specify the /Ft(dir) parameter with a builder as follows:

```
teamc builder -create c_builder -script icc -parameters "/FtE:\template"
```

Chapter 12. Working with MVS build scripts and builders

A builder is an object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers. For example, one builder might transform a COBOL source file into an object file. Another might transform a set of object files into an executable file. Builders use build scripts to invoke the tools that actually transform TeamConnection parts.

For MVS, a build script is a text file that contains JCL-like statements with additional TeamConnection syntax and substitutable variables. TeamConnection parses these statements and does the necessary allocations and program calls for a build.

Note: The builder type cannot be binary.

Usually a build administrator creates build scripts and builders, but anyone with the proper authority can do so.

This chapter tells how to create MVS build scripts and builders. It assumes that you have read “Chapter 9. Basic build concepts” on page 117. The following table directs you to the tasks to be done. In some cases, if the instructions are the same for OS/2 and MVS, the table refers you to topics in “Chapter 11. Working with build scripts and builders” on page 133.

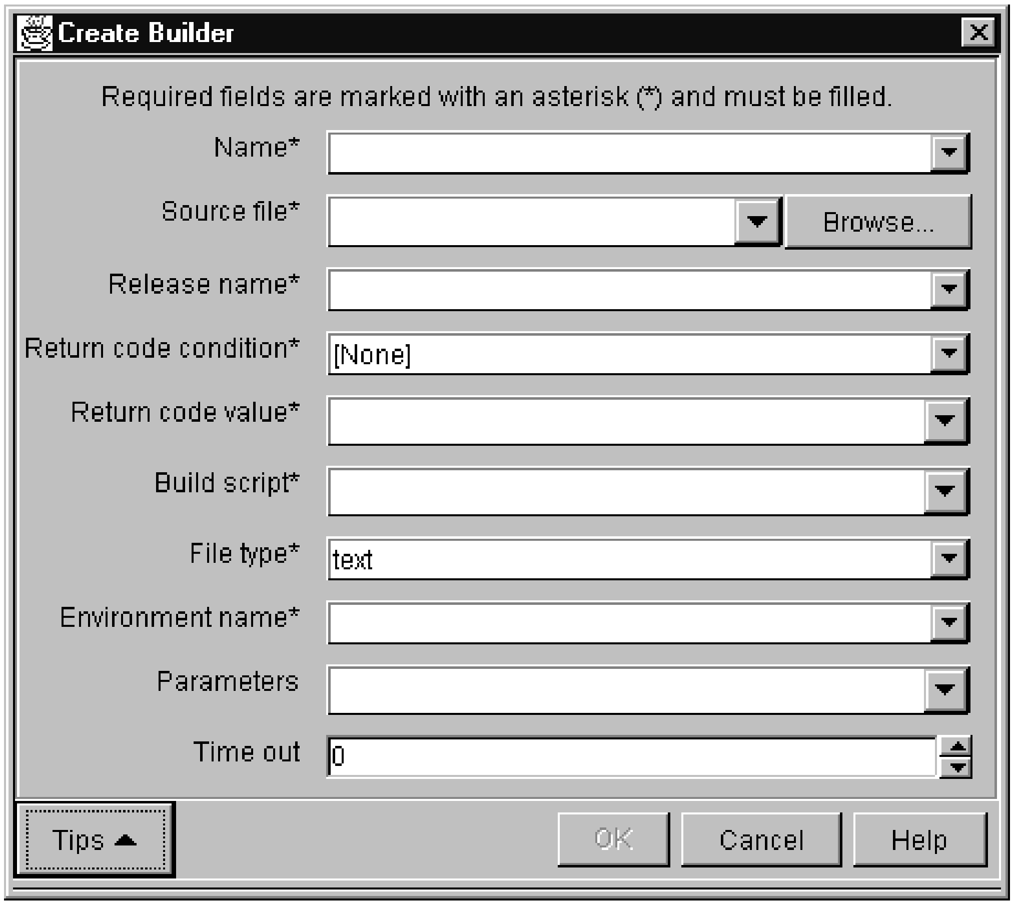
For more information about this task,	Go to this page.
Creating a builder for MVS builds	143
Writing an MVS build script	147
Testing a build script	140
Updating a builder	140
Putting a builder to work	140
Removing a builder from a part	141

Creating a builder for MVS builds

As with most other TeamConnection operations, there are two ways you can create a builder: using the graphical user interface (GUI) or the command line interface.

To create a builder using the GUI:

1. From the Actions menu on the Home Page, select **Builders** → **Create**.
2. On the Create Builders window, type the requested information.


Create Builder

Required fields are marked with an asterisk (*) and must be filled.

Name*	
Source file*	
Release name*	
Return code condition*	[None]
Return code value*	
Build script*	
File type*	text
Environment name*	
Parameters	
Time out	0

Figure 55. Create Builder window

To create a builder using the command line:

From an OS/2 command line, type the builder -create command and press Enter. The complete command syntax is the following:

```
teamc builder -create name  -condition RC_expression
        -environment name
        -from script_filespec
        -script name
        -value RC_value -release name
        -family nName
        [-text | -binary | -none]
        [-parameters parameters]
        [-timeout number] [-become user_name]
        [-verbose]
```

No matter which way you create a builder, you must specify a number of attributes for it. Together with the contents of the build script and the tools you use (the compilers, linkers, and so on), the following attributes define how a transformation takes place.

Builder
: The name of the builder must be unique within a release. It can be anything you want; we recommend you establish and follow a meaningful naming convention. An example of a builder name is `c370`.

Release
: This is the name of the release that contains the builder. Builders are

release-specific objects. They are not versioned within a release; therefore you can have only one version of a builder at any time in a release.

To use the builder from a previous release, you can link to a part that uses it in the previous release. This action copies the builder to the new release. Otherwise, you must create it again in the new release.

Script, File type, and Source file

These fields work together to define the build script that the builder invokes to accomplish the transformation. (The **File type** field on the GUI corresponds to `-text`, `-binary`, and `-none` in the command. The **Source file** field on the GUI corresponds to the `-from` attribute in the command.)

You must first create a separate file containing the build script. All MVS build scripts must be written using JCL statements and the TeamConnection syntax described in "Writing an MVS build script" on page 147. You can store the build script one of two ways:

- **To store the build script as part of the builder:** specify the fully qualified path name of your build script file as the source file, and specify the file type as `text`. When the builder is created, this source file is stored as part of it in the TeamConnection database.

  During a build, the build server creates and runs a local version of this file. Specify the name you want for this local file in the **Script** field. For example, you might specify these values:

  File type
  text
  Script fhbc
  Source file
  C:\build\script\fhbc.jcl

  When this builder is created, the contents of C:\build\script\fhbc.jcl are stored in the builder. When this builder is invoked, TeamConnection creates a file named FHBC in the data set referenced by the TEAMPROC ddname, writes the build script into this file, and then runs it.
- **To store the build script on MVS:** create the build script file and place it in the data set allocated to the TEAMPROC ddname in the RUNPGM JCL file. When you do this, specify the following attributes:

  File type
  none
  Script fhbc

  Do not specify a source file.
- If the builder is being used to only collect a set of build objects (for example, a VisualAge Generator collector part), specify these values:

  File type
  none
  Script null

See "Synchronizing the build of unrelated parts" on page 175 for an example.

Environment

This is the name of the environment supported by the builder, such as MVS.

The value that you specify here can be anything you like, but it must exactly match the environment value specified in the command used to start the build server.

It is recommended that you follow a naming convention for this attribute, using values such as `os2` and `mvs`.

Comparison operator and RC value

Together, these two attributes make up a Boolean expression that defines the criteria used to decide whether a specific build event was successfully accomplished, when evaluated against the value returned by the build script.

The **Comparison operator** and **RC value** fields on the GUI correspond to the `-condition` and `-value` attributes in the command.

The values allowed for these operators:

- **Comparison operator** are as follows:
  - **EQ or ==** - Equals
  - **LT or <** - Less than
  - **LE or <=** - Less than or equals
  - **GT or >** - Greater than
  - **GE or >=** - Greater than or equals
  - **NE or !=** - Not equal to
- **RC value** can be any positive integer.

An example of a Boolean expression formed from these two attributes is *return_value LE 4*.

This expression means that the build event is considered a success if the build script returns a value less than or equal to four.

Parameters

This is a string passed to the build script as its argument.

For example, for a builder used for linking load modules, you might specify a parameter string of `list,test`.

Timeout

This attribute specifies the number of minutes that a build server will wait for an invoked build script to return before concluding an error has occurred and stopping the build event.

If the timeout value is reached, the build event fails.

Because MVS builds are processed in batch mode but the build is submitted to the build server in real time, consider writing a user exit to check the time of day before allowing a build request to be submitted. Another approach to handling the timing of MVS builds is to start the MVS build server only at night and ensure that the MVS builders do not have short timeout values.

Writing an MVS build script

The best starting point for an MVS build script is an existing JCL fragment that is used for transforming inputs into outputs. For example, suppose you want to create a builder that compiles a C source file into an OBJECT file using IBM's C/370 compiler. You probably already have JCL that can be submitted as a batch job that does this.

When you create a build script for the MVS environment, you specify JCL statements with additional TeamConnection syntax. This build script is parsed by the build server. From the parsed results, TeamConnection allocates the specified ddnames and data sets; it then determines and executes the programs dynamically. The MVS build server uses the specialized TeamConnection syntax in the JCL to determine where to store the parts involved in an MVS build.

All statements in the MVS build script except for comments and inline data stream must start with two forward slashes (//).

Before you start writing your build script, refer to the manuals for the compiler, linker, or other transformation program to determine the data set requirements. Pay particular attention to the DCB attributes for LRECL, BLKSIZE, and RECFM.

Sample build scripts shipped with TeamConnection can be installed on MVS. Page 252 lists the sample build scripts. For instructions on installing these samples, refer to the *Administrator's Guide*

If you are debugging a build script, these manuals are also the first place to look for problems.

For more information about JCL syntax, refer to the *JCL User's Guide* and *JCL Reference* for your version of MVS. (These are listed in the bibliography at the back of this book.)

The following sample MVS build scripts are shipped with TeamConnection:

fhbmasm.jcl
Calls the MVS assembler

fhbcobm.jcl
Calls the MVS COBOL compiler

fhbmpli.jcl
Calls the PL/1 MVS compiler

File name conversions for MVS

TeamConnection file names are modified by the MVS build server according to the following rules:

- The directory path of a file name is not used. All characters of a file name up to and including the rightmost slash (/ or \) are thrown away.
- Lowercase characters are converted to uppercase characters.
- The file extensions are stripped from the right, up to and including the leftmost period. The extension, minus the period, is used by the MVS build tool to direct the file to particular data sets according to user-specified syntax in the MVS build scripts.
- The remaining name is truncated from the left, to a maximum of 8 characters.

- Names must contain characters that are valid in MVS. MVS allows the following characters:

  ```
  0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ$@#
  ```

  However, the name must begin with an alphabetic character.
- Underscore characters (_) in a base name are converted to at signs (@).

The following are examples of how a TeamConnection name is converted:
- A TeamConnection file name of src\build\fhbldobj.C is converted to FHBLDOBJ on MVS.
- A TeamConnection file name of src/build/fhbtruncate.c is converted to FHBTRUNC on MVS.

In both of these examples, the .C or .c is split away. The MVS build server uses the resulting extension to resolve and possibly allocate the MVS data sets needed for the build process. The extensions are required for parts that participate in an MVS build.

A TeamConnection file name of src\build\fhbtest.c.old is converted to FHBTEST, and c.old becomes the extension.

Passing parameters to an MVS build script

To pass parameters to your build script, specify them in the **Parameters** attribute of the builder. These are passed to MVS through the combination of the PARM keyword parameter on an EXEC card and the &TCPARM variable.

Note: Take extra care to use no single or double quotes in the **Parameters** attribute of the MVS builder definition. This rule follows standard JCL syntax for parameter substitution in the PARM keyword parameter of an EXEC statement.

You can use the **&TCPARM** variable in your MVS build scripts. This variable is substituted with any parameters that were specified using the -parameter attribute of the builder command or the **Parameters** field on the Create Builder window when the builder was created.

You can also use the following TeamConnection variables in writing MVS build scripts:

&TCBLDUSR
On MVS, can be used in JCL scripts, which will be substituted with the userID before processing the script.

&TCINPUT
This variable is used for in-stream data. For each build input, the line where &TCINPUT appears is duplicated and the variable &TCINPUT substituted with the input name.

&TCOUTPUT
This variable is used for in-stream data. For each build output, the line where &TCOUTPUT appears is duplicated and the variable &TCOUTPUT substituted with the output name.

&TCWKAREA
The name of the workarea in which the build is being performed.

&TCRELEAS
The name of the release in which the build is being performed.

Note: The &TCINPUT and &TCOUTPUT substitutable variables have limited scope in the MVS build scripts and should be used only within the in-stream data.

You can define other variables. You can set them by specifying a value for **Parameters** when you start a build. These variables are set in the parameters string passed to the build script.

Further, these variables can be used for variable substitution within MVS build scripts. Variable substitution works similarly to JCL variable substitution.

TeamConnection syntax for MVS build scripts

TeamConnection has extended the existing JCL syntax. The extended syntax tells the TeamConnection build server where to put the inputs, where to get the outputs, and where to get messages from the translators after an MVS build.

To direct inputs, outputs, and messages, add `TCEXT=`*xxx* to the data set attributes defined to a ddname, where *xxx* is one of the following:

- The base name extension from the TeamConnection part — for example, `TCEXT=H`, where H is the extension from A.H.
- One or more base name extensions from TeamConnection parts, surrounded by parentheses — for example, `TCEXT=(H,HPP)`, where H is an extension from A.H or HPP is an extension from A.HPP.
- The string `TCOUT`, which declares that the contents of the data set assigned to the ddname will be sent back to TeamConnection. Users can view this information in one of these ways:
  - From a command line prompt, typing `teamc part` *name* `-viewmsg` and pressing Enter
  - Selecting **Part → View → View build message** from the Actions menu on the Home Page

  Note: More than one ddname can specify `TCOUT`; the results are concatenated in the order of appearance.

When you add the TCEXT attribute for a ddname specification, you must also specify other attributes to allocate the data set through dynamic allocations:

- SPACE
- UNIT
- DCB, which includes the LRECL, BLKSIZE, and RECFM attributes

The UNIT attribute defaults to VIO unless the `-U` parameter is specified when the MVS build server is started.

For translation messages, you can allocate a data set to the ddname TC$LIST and specify the attributes yourself. Otherwise, the build server allocates this data set with the following attributes by default:

```
//TC$LIST DD  DCB=(RECFM=VB,LRECL=255,BLKSIZE=32640),
//  SPACE=(CYL,(2,1)),DISP=(NEW,DELETE),UNIT=VIO
```

Supported JCL syntax

The TeamConnection MVS build server supports only a subset of the available JCL syntax.

The following are not supported:
- A JOBSTEP statement
- `DISP=(..,PASS)...`

JCL procedures can be used on an EXEC statement. However, you must verify that any procedure called by the build script uses syntax that TeamConnection supports.

The following list indicates the positional and keyword parameters that are supported. You can verify the syntax in the *JCL Reference.*

EXEC statement

```
//label   EXEC positional_parameter,keyword_parameter
```

The following parameters are supported.

Positional parameters:
- `PGM=`*program_name*, where *program_name* is an executable load module
- `PROC=`*procedure_name*, where *procedure_name* is an existing JCL procedure
- *procedure_name*, where *procedure_name* is an existing JCL procedure

Keyword parameters:
- `PARM='`*information*`'`, where *information* is the parameter string passed to the load module.
- COND=(*code,operator* [*,stepname]*)
  - *code* is the value to test against the return code from a previous step
  - *operator* is the comparison to be made between the value for *code* and the return code
  - *stepname* is the step issuing the return code

All other keyword parameters are ignored and not used.

DD STATEMENT

```
//label  DD   keyword parameter
```

Positional parameters

The only supported positional parameter is `[*]`, which begins an in-stream data set containing no JCL.

Keyword parameters

The following keywords are supported.
- `DSN=`*data_set_name* or `DSNAME=`*data_set_name*
- `DISP=`*status* or `DISP=([`*status*`] [,`*normal-termination-disp*`] [,`*abnormal-termination-disp*`])`
  - Valid values for *status* are `NEW`, `OLD`, `SHR`, or `MOD`.

  - Valid values for *normal_termination_disp* or *abnormal_termination_disp* are `DELETE`, `KEEP`, `CATLG` or `UNCATLG`.
- `UNIT=`*`unit_type`*, where *unit_type* is any value allowed in JCL. The default is VIO unless a different default is set when the MVS build server is started.
- `SPACE=(`*`allocation_type`*`,(`*`primary`*`[, `*`secondary`*`] [,`*`directory`*`])[,RLSE] [,CONTIG])`
  - Valid values for *allocation_type* are `TRK`, `CYL`, or the block size.
  - *primary* is the primary number of the allocation type.
  - *secondary* is the secondary number of the allocation type.
  - *directory* is the number of directory blocks for a partitioned data set.
- `DCB=(LRECL=record_length,BLKSIZE=block_size,RECFM=record_format)`

  Valid values for *record_format* are `F`, `FB`, `V`, `VB`, or `U`).
- `DSORG=`*`data_set_organization`*

  Valid values for *organization* are the following:
  - `PO` for a partitioned data set
  - `PS` for a sequential data set
- `DDNAME=`*`label`*, where *label* is the later ddname label reference. This parameter is supported only for simple cases.
- `SYSOUT=`*`class`*

  This will always be allocated as a DUMMY DSN.

All other keyword parameters are ignored and not used.

Example of a build script for a C compile

The following JCL can be submitted as a batch job to do the following:
- Compile the source file member in the data set WELLSK.TEAMC.C
- Produce an object file member in the data set WELLSK.TEAMC.OBJ
- Produce a listing of the source file in the file member in the data set WELLSK.TEAMC.LISTING
- List the compiler messages in the file member in the data set WELLSK.TEAMC.ERROR

```
//COMPILE EXEC PGM=EDCCOMP,
// PARM='LO,SSCOMM,NOSEQ,NOMAR,LIS,FL(I),SO,DECK,TEST',
//         REGION=1536K
//STEPLIB  DD DSN=SYS1.EDC.SEDCCOMP,DISP=SHR
//         DD DSN=SYS1.EDC.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYSMSGS  DD DSN=SYS1.EDC.SEDCDMSG(EDCMSGE),DISP=SHR
//SYSIN    DD DSN=WELLSK.TEAMC.C(MEMBER),DISP=SHR
//USERLIB  DD DSN=WELLSK.TEAMC.H,DISP=SHR
//SYSLIB   DD DSN=SYS1.EDC.SEDCHDRS,DISP=SHR
//SYSPUNCH DD DSN=WELLSK.TEAMC.OBJ(MEMBER),DISP=SHR
//SYSLIN   DD SYSOUT=*
//SYSPRINT DD DSN=WELLSK.TEAMC.ERROR(MEMBER),DISP=SHR
//SYSCPRT  DD DSN=WELLSK.TEAMC.LISTING(MEMBER),DISP=SHR
//SYSUT1   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD SYSOUT=*
//
```

Figure 56. A JCL fragment for an MVS compile

The first step in converting the JCL fragment is to recognize the intent for each of the data sets and ddnames. For this C/370 compiler example, the SYSIN ddname needs to be associated with the source file, the SYSPUNCH ddname needs to be associated with the object file, and so on.

In each of these cases, the build script must tell the TeamConnection build server where to put or pick up the parts before and after the execution of the specified program (`PGM=EDCCOMP`).

Assume that your source files in TeamConnection have the extension .c, your object files have .obj, and your include files .h or .hpp. You allocate a data set to the SYSIN ddname to contain a source file with a .c extension. You specify the DCB, UNIT, DISP, and SPACE attributes to dynamically create this data set every time this build script is invoked. Notice that the attribute SPACE=(TRK,(10,5)) indicates a sequential data set organization.

You specify the output messages that will be returned to TeamConnection by using the TCOUT attribute. This attribute tells the MVS build server to return the information in the data set associated with the TCEXT=TCOUT attribute.

Note: The STEPLIB is renamed by the MVS build server to STEPLIBB for data set lookup of the program specified by the PGM parameter on an EXEC statement.

The following MVS build script is the result of converting the JCL fragment by adding the TeamConnection MVS JCL syntax.

```
//COMPILE EXEC PGM=EDCCOMP,
// PARM='LO,SSCOMM,NOSEQ,NOMAR,LIS,FL(I),SO,DECK,&TCPARM',
//         REGION=1536K
//STEPLIB  DD DSN=SYS1.EDC.SEDCCOMP,DISP=SHR
//         DD DSN=SYS1.EDC.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYSMSGS  DD DSN=SYS1.EDC.SEDCDMSG(EDCMSGE),DISP=SHR
//SYSIN     DD TCEXT=(C,CPP),DISP=(NEW,DELETE),
//          UNIT=SYSDA,SPACE=(TRK,(10,5)),
//          DCB=(RECFM=VB,LRECL=150,BLKSIZE=3200)
//USERLIB DD TCEXT=(H,HPP),DISP=(NEW,DELETE),
//          UNIT=VIO,SPACE=(TRK,(5,10,10)),
//          DCB=(RECFM=VB,LRECL=50,BLKSIZE=3200)
//SYSLIB   DD DSN=SYS1.EDC.SEDCHDRS,DISP=SHR
//SYSPUNCH DD TCEXT=OBJ,DISP=(NEW,DELETE),
//         UNIT=VIO,SPACE=(TRK,(10,5)),
//         DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIN   DD SYSOUT=*
//SYSPRINT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),UNIT=VIO,
//   DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSCPRT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),UNIT=VIO,
//   DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT1   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9   DD UNIT=VIO,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD SYSOUT=*
//
```

Figure 57. A JCL fragment converted to a build script

Example of a build script for a COBOL compile

TeamConnection provides a sample build script program for compiling MVS COBOL programs. This sample is called fhbcobm.jcl. It invokes a JCL procedure called IGYWC, which needs to be in the system PROCLIB concatenation or in the data set identified by the TEAMPROC DD statement in the MVS build job. You may need to adjust the default parameters for the system. The following JCL should work with any IBM COBOL/II type of compiler such as the IBM COBOL/II compiler IGYCRCTL:

```
//*-----------------------------------------------------
//* PROGRAM:  cobolcmp.jcl
//* IBM COBOL for MVS
//* Compile Only
//*
//*-----------------------------------------------------
//COBOLCMP EXEC PGM=IGYCRCTL,PARM='&TCPARM'
//*
//* INPUT FILES WITH EXTENSION CBL
```

```
//*
//SYSIN    DD  TCEXT=CBL,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,10)),UNIT=SYSDA,
//   DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//*
//* COPY FILES WITH EXTENSION CPY
//*
//SYSLIB   DD  TCEXT=CPY,DISP=(NEW,KEEP),
//   SPACE=(32000,(30,30,30)),UNIT=SYSDA,
//   DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//*
//SYSPRINT DD  TCEXT=TCOUT,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,30)),UNIT=SYSDA,
//   DCB=(RECFM=FBA,LRECL=133,BLKSIZE=3990)
//SYSLIN   DD  TCEXT=OBJ,UNIT=SYSDA,
//   DISP=(NEW,DELETE),SPACE=(32000,(30,10)),
//   DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//*
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//
```

Example of a build script for a link

Because MVS load modules are not easily transferable, TeamConnection provides a sample build script program that reads linkage editor SYSLIN control statements. This script produces a single file that can be returned from MVS and loaded into TeamConnection. You can later extract the file and transport it to MVS, where it can be link edited to produce an executable load module.

The next example shows this sample build script, named fhbtclnk.jcl, which is shipped with the TeamConnection client.

You can use either of the following for an INCLUDE control statement for the FHBTCLNK program:

- `INCLUDE DDNAME(MEMBER)`
- `INCLUDE DDNAME`

This syntax is a subset of the linkage editor INCLUDE card.

If the card is an INCLUDE ddname(MEMBER) control statement, the object code is copied into a sequential data set associated with the SYSMOD ddname. Otherwise, the control card is embedded in the data set associated with the SYSMOD ddname. This data set can be returned as the output from this build script.

```
//FHBTCLNK EXEC  PGM=FHBTCLNK,
// PARM='SIZE=(768K,192K),LIST,MAP,AMODE(31),RMODE(24),LET,XREF'
//STEPLIB  DD  DSN=userid.teamc.LOADLIB,DISP=SHR
//SYSMOD  DD TCEXT=LOAD,DISP=(NEW,DELETE),
//   SPACE=(32000,(30,10)),UNIT=VIO,
//   DCB=(RECFM=U,LRE10CL=80,BLKSIZE=3200)
//OBJ     DD  TCEXT=(OBJ,PRE),DISP=(NEW,DELETE),
//    UNIT=VIO,SPACE=(32000,(30,10,10)),
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSPRINT DD  TCEXT=TCOUT,DISP=(NEW,DELETE),
//   UNIT=VIO,SPACE=(TRK,(30,10)),
//   DCB=(RECFM=FB,LRECL=121,BLKSIZE=1210)
```

```
//SYSLIN   DD  *
 INCLUDE OBJ(&TCINPUT)
 ENTRY CEESTART
//
```

TCEXT attributes have been added to the following DD statements:

Data set	Purpose
SYSMOD	Return the output to check in to TeamConnection
OBJ	Receive the object files transported to MVS from TeamConnection
SYSPRINT	Return any FHBTCLNK messages to TeamConnection

In the SYSLIN data stream, the statement `INCLUDE OBJ(&TCINPUT)` will be duplicated for all of the inputs to this build. The &TCINPUT variable will be replaced with the base name of the input without the extension.

To use the output of this build script as an MVS executable, do the following:

1. Extract the output from TeamConnection.
2. Transfer the output as a binary file from your workstation to MVS (for example, using FTP).
3. Link edit this output into a load module. Possible SYSLIN control statements for the link step include the following:

   ```
   //SYSLIN DD *
     INCLUDE OBJECT(OUTPUT)
     NAME module(R)
   //
   ```

The output specified in `INCLUDE OBJECT(OUTPUT)` contains embedded control statements specified from the build script FHBTCLNK. The linkage editor recognizes these embedded statements and produces an executable load module from the output file. The NAME control statement cannot be embedded in the output data set.

Chapter 13. Working with parsers

This chapter describes how to create a parser. It assumes that you have read "Chapter 9. Basic build concepts" on page 117.

Consider the task of defining and maintaining a build tree. One of the more time-consuming, and error-prone, portions of this task is defining the dependencies that one TeamConnection part has on others.

For example, if hello.c includes hello.h, you need to define hello.h as a dependency of hello.c in the build tree. That sounds simple enough, but imagine a real application in which there are hundreds of dependencies and the dependencies have dependencies. Defining such a tree becomes very difficult; maintaining it, even more so.

To solve this problem and automate some of the work of defining and maintaining a build tree, you can instead use a parser object. The task of a parser is to inspect source code to determine dependencies. TeamConnection verifies all parser dependencies when the user creates or checks in the part and again during build. TeamConnection will add all parser dependencies that it can find and, for build, update them as needed. In the previous example, a parser can inspect hello.c, recognize that it has a dependency on hello.h, and create that dependency in the TeamConnection build tree.

Because parsers are language-dependent, you probably need a different parser for each language you use in a particular release. For example, you might have both a COBOL parser and a C parser in a release. Many parts in the release can use the same parser.

Usually a TeamConnection administrator defines parsers, but anyone with the proper authority can do so.

Creating a parser

As with most other TeamConnection operations, there are two ways you can create a parser: using the graphical user interface (GUI) or the command line interface.

To create a parser using the GUI:

1. From the Actions menu on the Home Page, select **Parser → Create**.
2. On the Create Parser window, type the requested information.

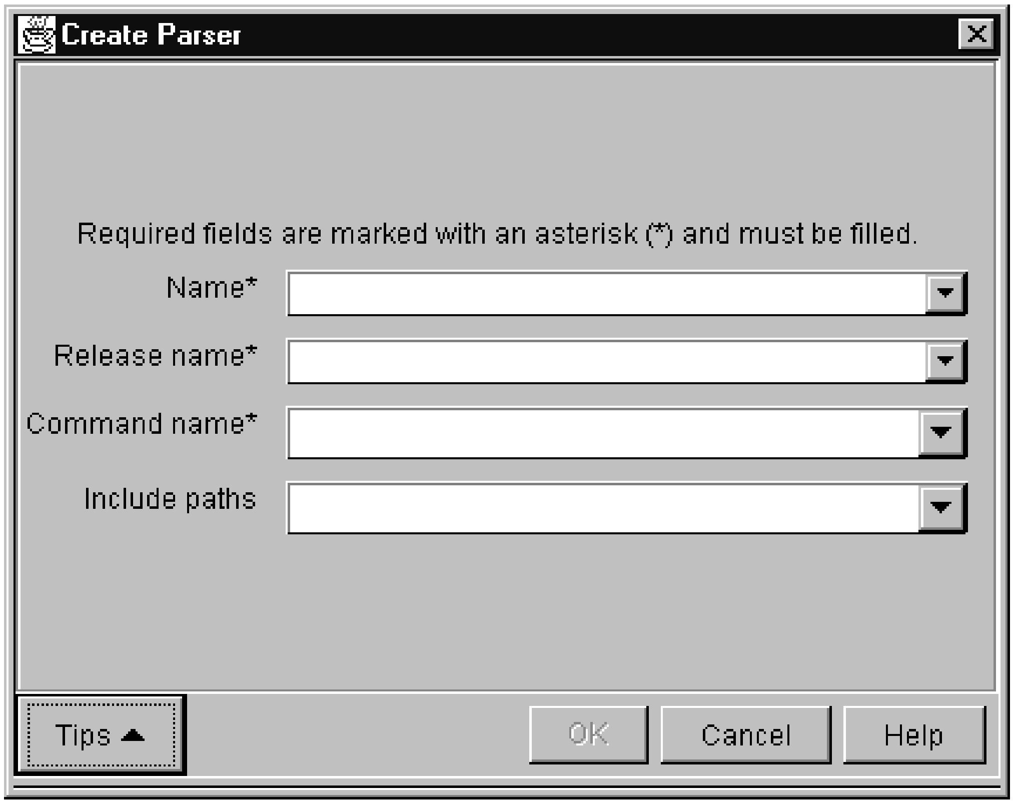


Figure 58. Create Parser window

From a command line, type the parser -create command and press Enter. The complete command syntax looks like the following:

```
teamc parser -create name  -command name -release name -family name
       [-include paths]
       [-become user_name] [-verbose]
```

No matter which way you create a parser, you must specify a number of attributes for it. Together with the contents of the parser command file, the following attributes define how a parser determines the dependencies for a TeamConnection part.

Parser
: The name of the parser must be unique within a release. It can be anything you want, but for best results, establish and follow a meaningful naming convention. An example of a parser name is `c_parser`.

Release
: This is the name of the release that contains the parser. Parsers are release-specific objects. They are not versioned within a release; therefore you can have only one version of a parser at any time in a release.

  To use the parser from a previous release, you can link to a part that uses it in that release. This action copies the parser to the new release. Otherwise, you must create the parser again in the new release.

Command
: This is the name of the command file that the parser invokes to determine the dependencies. It can be any file name that exists in the execution path of the family server at the time a build is performed. The parser command is run as a subprocess on the machine where the family server is located.

The task of the command file is to inspect the source file and return a list of dependencies. The syntax for invoking this command is discussed in “Writing a parser command file”.

Include

This is a concatenated set of paths that define where the parser looks for parts when processing the set of dependencies returned from the command file. These dependencies come in two types:

- A dependency in which the file is stored in the TeamConnection database. For example, hello.c includes hello.h, and both files are stored in the TeamConnection database. During a build, these dependencies must be extracted to a path accessible by the build server. Because a build extracts parts from TeamConnection, anyone requesting a build needs to have PartExtract authority to all parts involved in the build.
- A dependency on a file that is not stored in the TeamConnection database. An example of such a dependency is stdio.h, which is typically stored in a compiler’s include path and not in the TeamConnection database.

Each path named in **Include** is queried in the TeamConnection database to see if it contains a part matching the dependency name. For example, suppose you define a parser named c_parser with an include path as follows:

```
src\include;src\package;.;src\comm\include;
```

One of the parts to which this parser is attached, src\example.cpp, contains the statement `#include "example.hpp"`. Thus the command file for c_parser reports example.hpp as a dependency of src\example.cpp. The parser concatenates each path listed in c_parser’s include path with the name example.hpp, then inspects the contents of the TeamConnection database to see if a part with that name exists. So the TeamConnection database is queried first to find src\include\example.hpp, then src\package\example.hpp.

The period (.) in the include path tells TeamConnection to concatenate the path of the part to which the file is a dependent with the dependent’s file name. In this example, that means the TeamConnection database is queried to find a part named src\example.hpp.

Writing a parser command file

A parser command file accepts five parameters as input:

- *source file*—the name of the file that contains the source to be parsed.
- *dependency list file*—the name of a file into which the names of the dependent files should be written, one per line. For example, the contents of the file might look like this:

  ```
  hello.h
  stdio.h
  ```
- *family*—the family that contains the source to be parsed.
- *release*—the release that contains the source to be parsed.
- *workarea*—the workarea that contains the source to be parsed.

Both the source file and the dependency list file are created by the TeamConnection family server. They are erased after the parse is complete.

To write a command file, write a program, in any language, that does the following:

1. Reads the source file
2. Determines which other files are used by it
3. Writes out the list of such files into the dependency list file

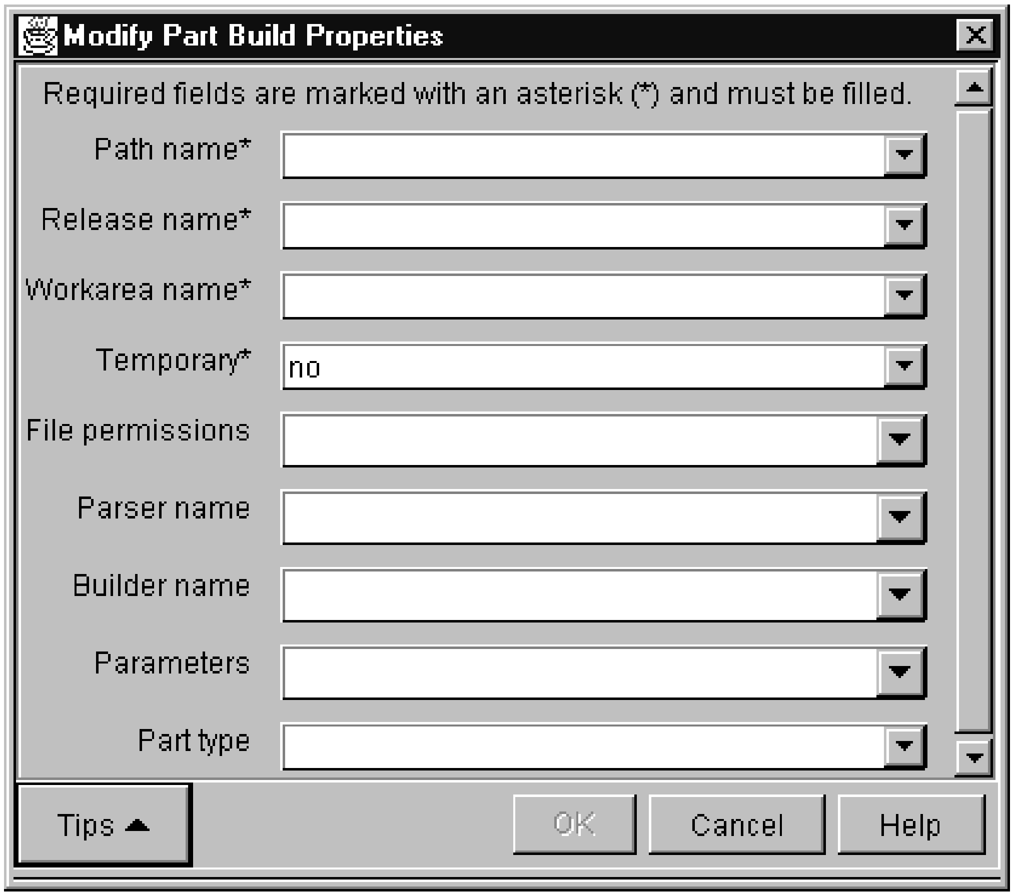
For example, for a C source file, the program could report a list of all the files included by the source file (using `#include` statements). For a COBOL program, `COPY` statements would be the cue. TeamConnection ships a sample of a command file named fhbopars.cmd. It is written in REXX.

Putting a parser to work

For an application to use a parser, the parser must be attached to the TeamConnection parts that it will check for dependencies. Unlike a builder, a parser is attached to the *input* part rather than the output.

To attach a parser to a part, do one of the following:

- From the GUI, select **Parts → Modify → Build Properties** from the Actions menu of the TeamConnection Home Page. On the Modify Part Build Properties window, type the name of the parser.


Modify Part Build Properties

Required fields are marked with an asterisk (*) and must be filled.

Path name*	
Release name*	
Workarea name*	
Temporary*	no
File permissions	
Parser name	
Builder name	
Parameters	
Part type	

Figure 59. Modify Part Build Properties window

- At a command prompt, type the following and press Enter:

```
teamc part -modify part -parser name -release name
  -family name
```

The complete syntax for this command is described in the *Commands Reference*.

You can also attach a parser to a part when the part is created.

After you attach a parser to a part, it is ready for action. The next time the part is checked in, when a part is created, or when the parser is attached, the parser will invoke its command file, which will report back a list of dependencies.

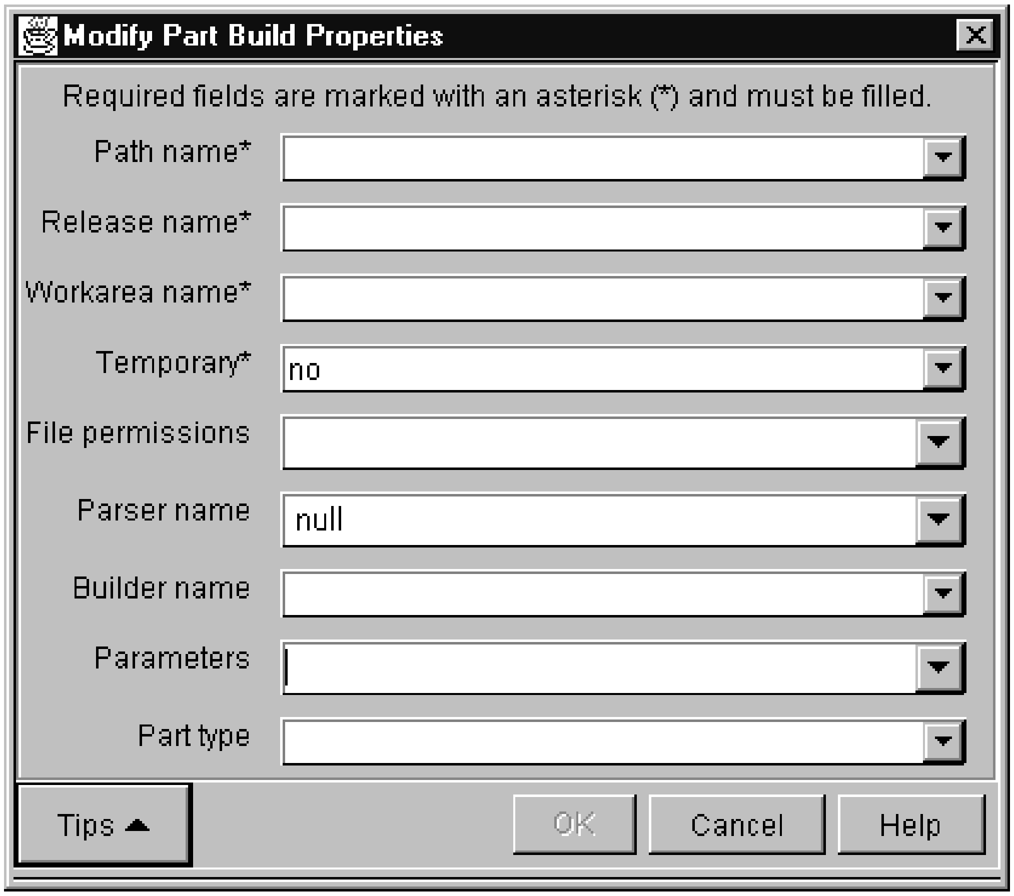
Using a parser does not keep you from defining dependencies manually by using the GUI or the part -connect command. If you explicitly define a dependency in this way, the dependency is not deleted unless you delete it, regardless of whether the parser would recognize it as such.

For more information about attaching parsers to the build tree, refer to “Creating the build tree for the application” on page 165.

Removing a parser from a part

If you no longer want to use a parser to determine dependencies for a part, do one of the following:

- From the GUI, select **Parts → Modify → Build Properties** from the Actions menu of the TeamConnection Home Page. On the Modify Part Build Properties window, type `null` in the **Parser** field.



Required fields are marked with an asterisk (*) and must be filled.

Path name*	
Release name*	
Workarea name*	
Temporary*	no
File permissions	
Parser name	null
Builder name	
Parameters	
Part type	

Figure 60. Modify Part Build Properties window

- From a command line, type the following:

```
teamc part -modify name -parser null
        -release name -family name
```

Chapter 14. Building an application: an example

This chapter uses an extended example to describe in more detail how each of the components of the build function work together. All commands used in this example are described in detail elsewhere in this publication. This example walks through the control flow for a sample application, explaining what happens at each step.

These are the tasks involved in building our sample application, msgcat.exe:

Task	Page
Starting build servers	164
Creating builders and parsers	165
Creating the application build tree	165
Starting the build	169
Monitoring the build	172
Building in spite of errors	173
Forcing a build of all parts	173
Finding out which parts will be built	173
Canceling a build	174

We will use a simple example build tree that looks like the following:

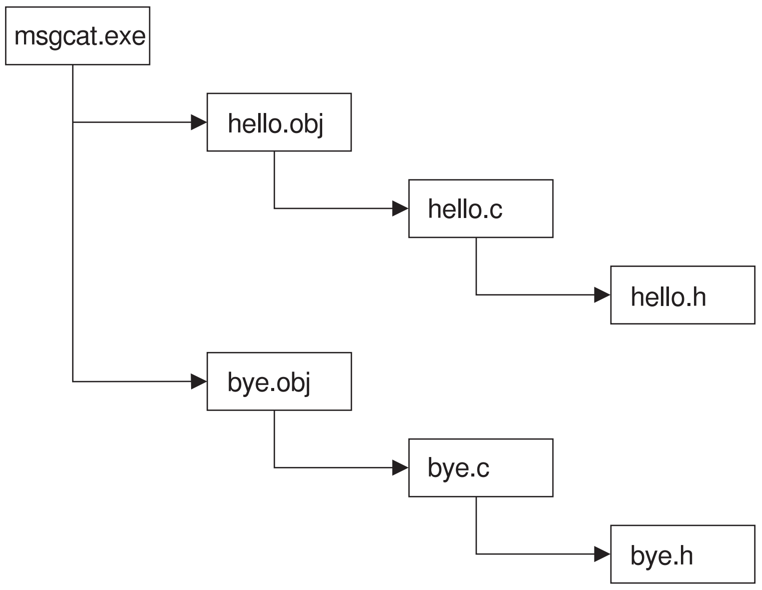


Figure 61. Sample build tree

For more examples of build trees, see “More sample build trees” on page 174.

In terms of the build object model, the objects that make up this tree look like this:

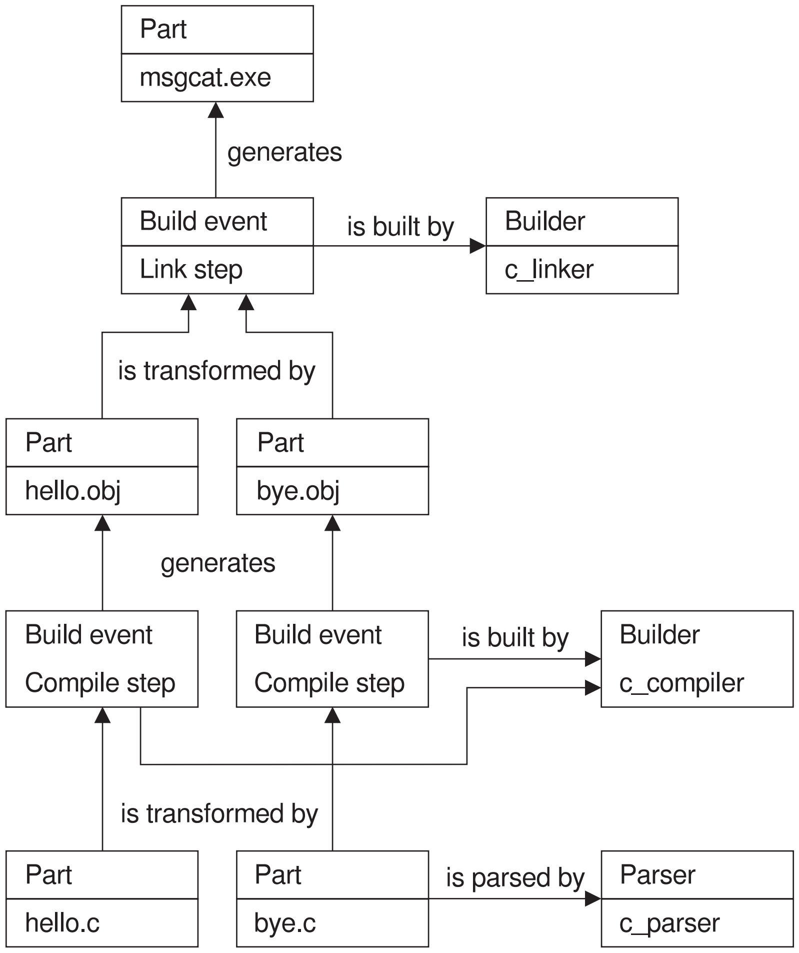


Figure 62. Sample build object model for msgcat.exe

Starting the build servers

The software development team in our example is building large applications using a family named testfam, so they set TC_FAMILY to `testfam`. They plan to spread the work across several build servers, taking advantage of TeamConnection's ability to perform multiple build events simultaneously.

Mark, the build administrator, has installed a number of build servers on the team's machines, for building OS/2 and MVS applications. As he starts them (in pairs), he groups them into *pools*, according to the work he expects to use them for.

Mark plans for the following pools:

mvspool
For MVS builds

pool1 For normal OS/2 builds

pool2 For fast, high-priority OS/2 builds

Each pool is formed as Mark starts build servers and assigns them to it. He starts the following build server (bldserv2):

```
teamcbld -b bldserv2 -p pool1 -e os2
```

The parameters specify the following:

-p	The build server is assigned to the pool named `pool1`.
-e	The environment is `os2`.

Use the teamcbld command to start the build server when the family server has already been started. To start the family server along with the build server, you can use the teamcd command.

Creating builders and parsers

For the parts of the application that are written in C language, Mark creates the following:

- A builder named c_compiler, to do the compiles
- A builder named c_linker, to do the links
- A parser named c_parser, to check for dependencies

For both builders Mark specifies `os2` as the **Environment**, the same as that of the build server (bldserv2) started earlier. Build events that use these builders (c_compiler and c_linker) can take place on this build server.

After he creates the builders and parsers for the applications, Mark spreads the following information to the programmers who will be using them:

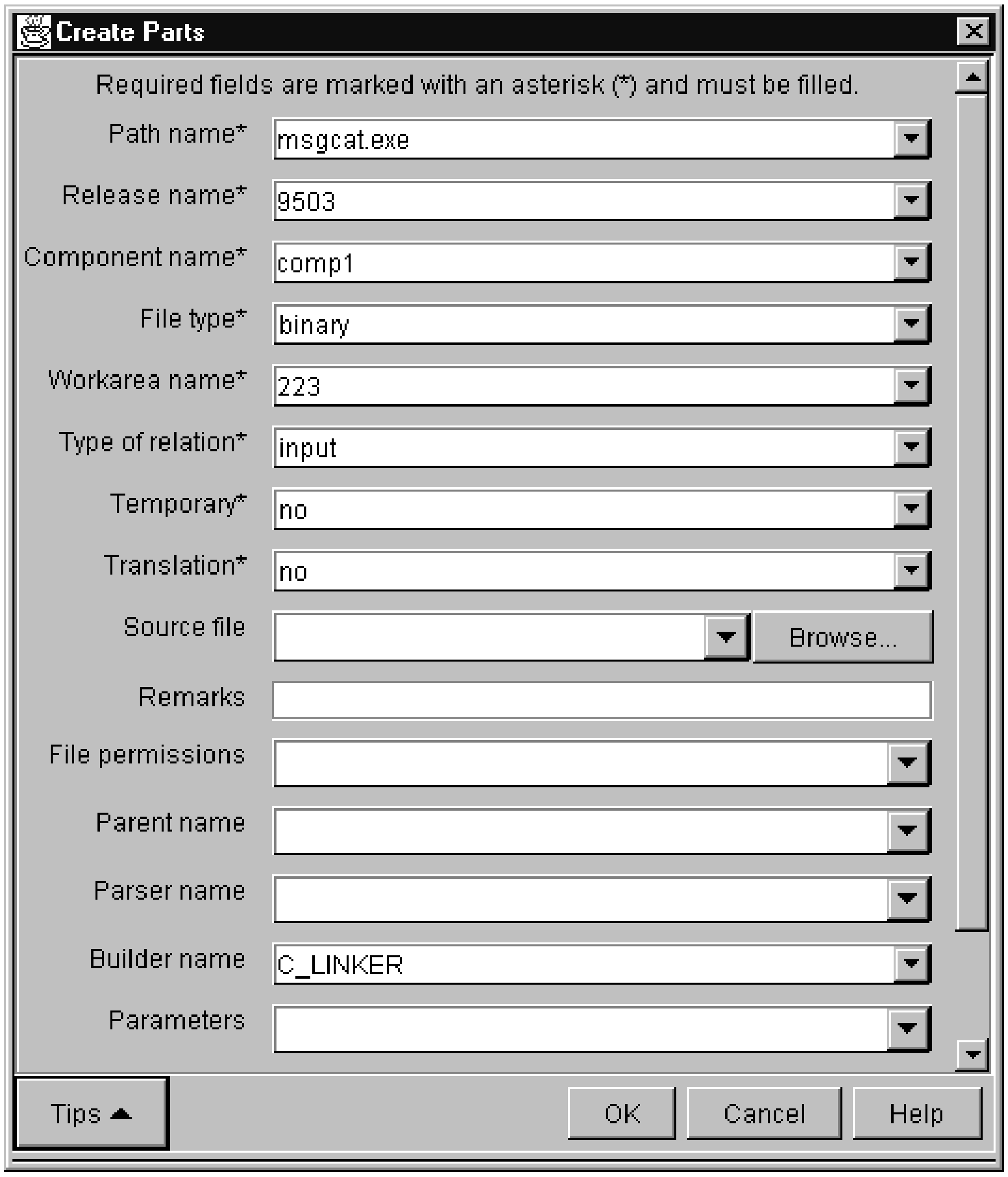
- The names of the build pools
- The names and purposes of the builders and parsers

Creating the build tree for the application

At this point, Greg begins defining the build tree for his portion of the application, as shown in Figure 61 on page 163. He has already created the files hello.c, hello.h, bye.c, and bye.h in the TeamConnectiondatabase. Now he does the following:

1. Creates a place-holder part for the output of the link step. This file, msgcat.exe, is the target for the entire build, the output of linking hello.obj and bye.obj using the builder c_linker, and the parent of hello.obj and bye.obj. Because the file has no contents initially, he selects **No source** (or specifies `-empty` on the command line), to identify it as a place holder.

   Using the GUI, he can create this file by selecting **Part → Create** from the Actions menu of the Home Page and completing the fields as shown in the following illustration:


Create Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name*	msgcat.exe
Release name*	9503
Component name*	comp1
File type*	binary
Workarea name*	223
Type of relation*	input
Temporary*	no
Translation*	no
Source file	
Remarks	
File permissions	
Parent name	
Parser name	
Builder name	C_LINKER
Parameters	

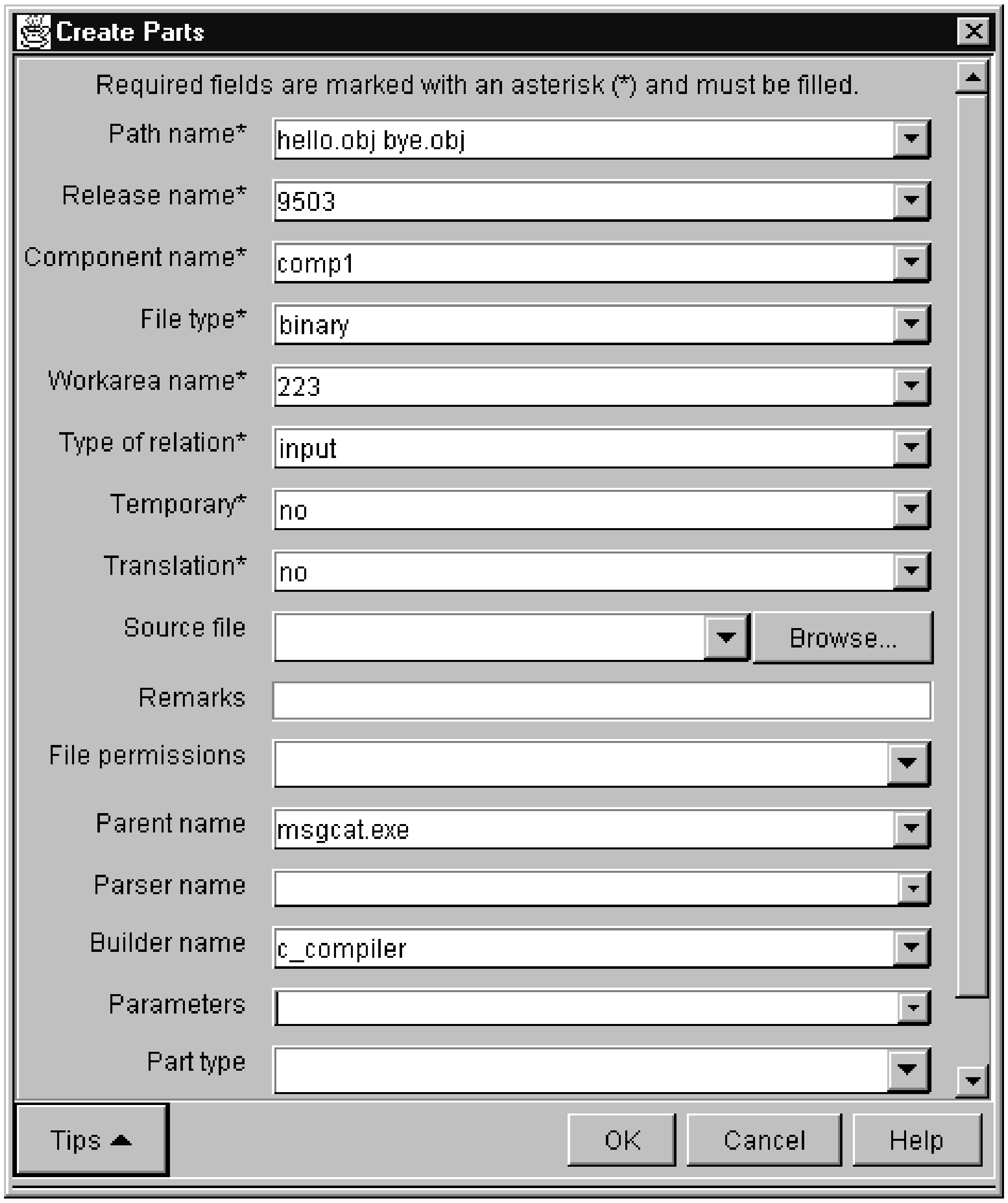
Figure 63. Create Parts window

Using the command-line interface, he can create the part by issuing the following command:

```
teamc part -create msgcat.exe -builder c_linker -binary -empty
  -release 9503 -workarea 223 -component comp1
```

2. Creates two place-holder parts for the output of compiling the .c files. These parts are the output of the compile step; c_compiler, the builder that manages that step, is attached to both of them. He indicates that they are input to their parent file, msgcat.exe.

   Using the GUI, he can create these files by selecting **Part → Create** from the Actions menu of the Home Page, and completing the fields as shown in the following illustration:



Required fields are marked with an asterisk (*) and must be filled.

Path name*	hello.obj bye.obj
Release name*	9503
Component name*	comp1
File type*	binary
Workarea name*	223
Type of relation*	input
Temporary*	no
Translation*	no
Source file	
Remarks	
File permissions	
Parent name	msgcat.exe
Parser name	
Builder name	c_compiler
Parameters	
Part type	

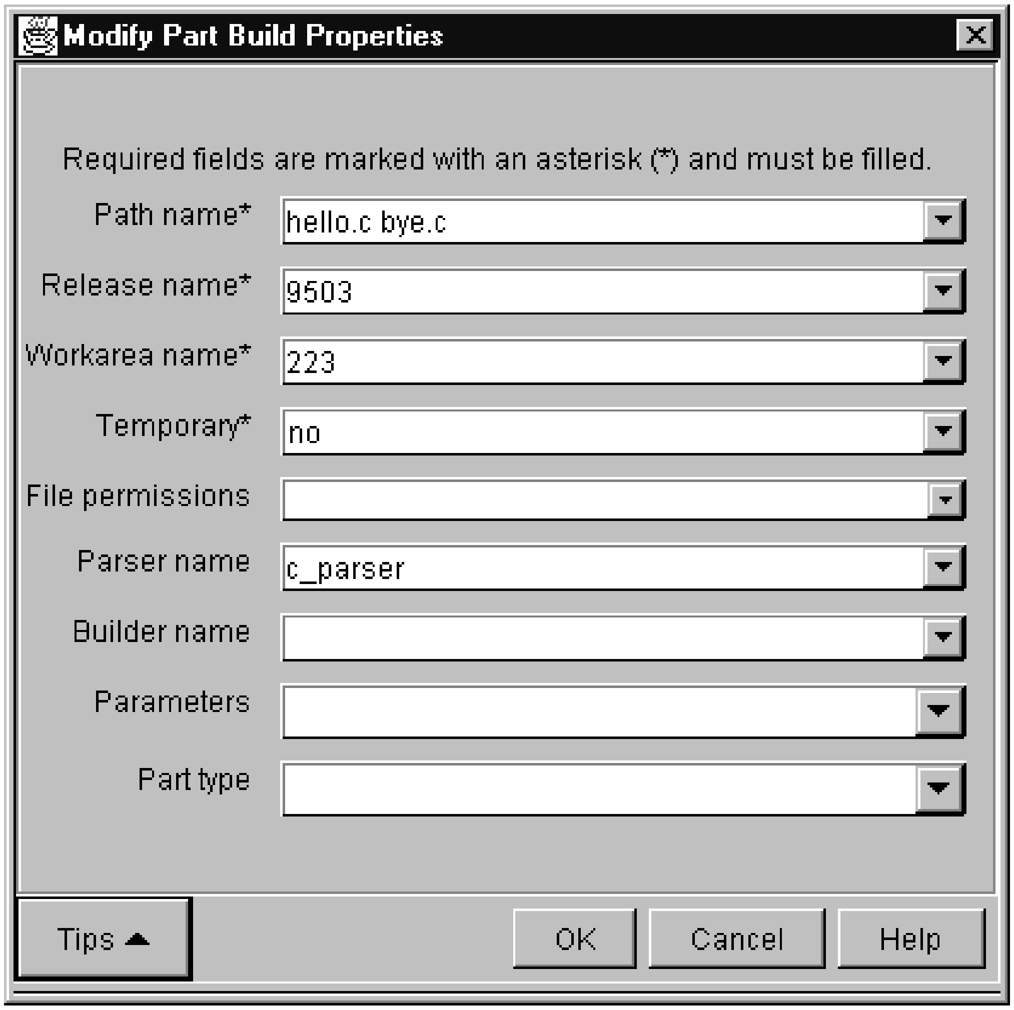
Figure 64. Create Parts window

Using the command-line interface, he can create the parts by issuing the following command:

```
teamc part -create hello.obj bye.obj -builder c_compiler -binary -empty
  -release 9503 -workarea 223 -component comp1 -parent msgcat.exe -input
teamc part -create hello.h bye.h -release 9503 -workarea 223
```

3. Attaches the parser `c_parser` to the .c files.

   Using the GUI, he can attach the parser to these files by selecting **Part → Modify → Build Properties** from the Actions menu of the Home Page, and completing the fields as shown in the following illustration:



Required fields are marked with an asterisk (*) and must be filled.

Path name*	hello.c bye.c
Release name*	9503
Workarea name*	223
Temporary*	no
File permissions	
Parser name	c_parser
Builder name	
Parameters	
Part type	

Figure 65. Modify Part Build Properties window

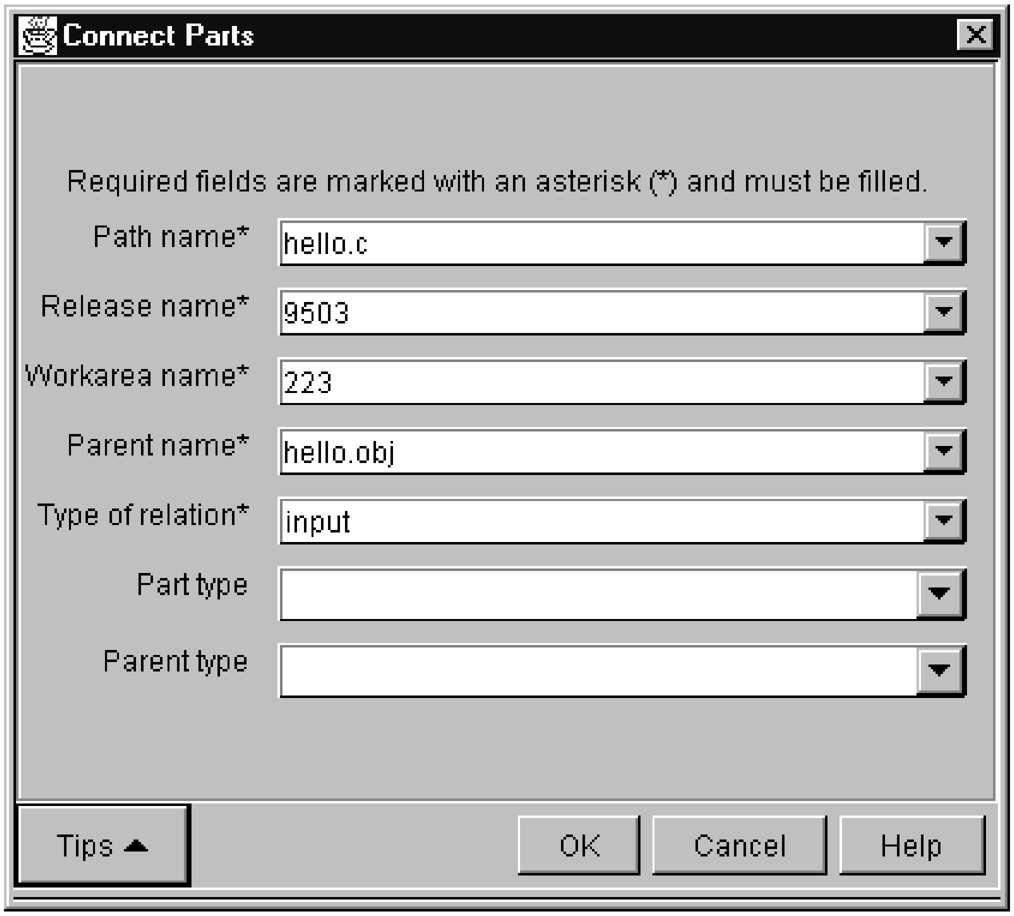
Using the command-line interface, he can attach the parser to these parts by issuing the following command:

```
teamc part -modify hello.c bye.c -parser c_parser -release 9503
  -workarea 223
```

Remember, the parser is attached to an *input* file. The part’s contents will be parsed and dependency information will be stored.

4. Connects the .c files into the build tree.

   Using the GUI, he can connect these files by selecting **Parts → Connect** from the Actions menu of the Home Page and completing the fields as shown in the following illustration. He needs to execute this function twice: once to connect hello.c to hello.obj and once to connect bye.c to bye.obj.



Required fields are marked with an asterisk (*) and must be filled.

Path name*	hello.c
Release name*	9503
Workarea name*	223
Parent name*	hello.obj
Type of relation*	input
Part type	
Parent type	

Figure 66. Connect Parts window

Using the command-line interface, he can connect these parts by issuing the following commands:

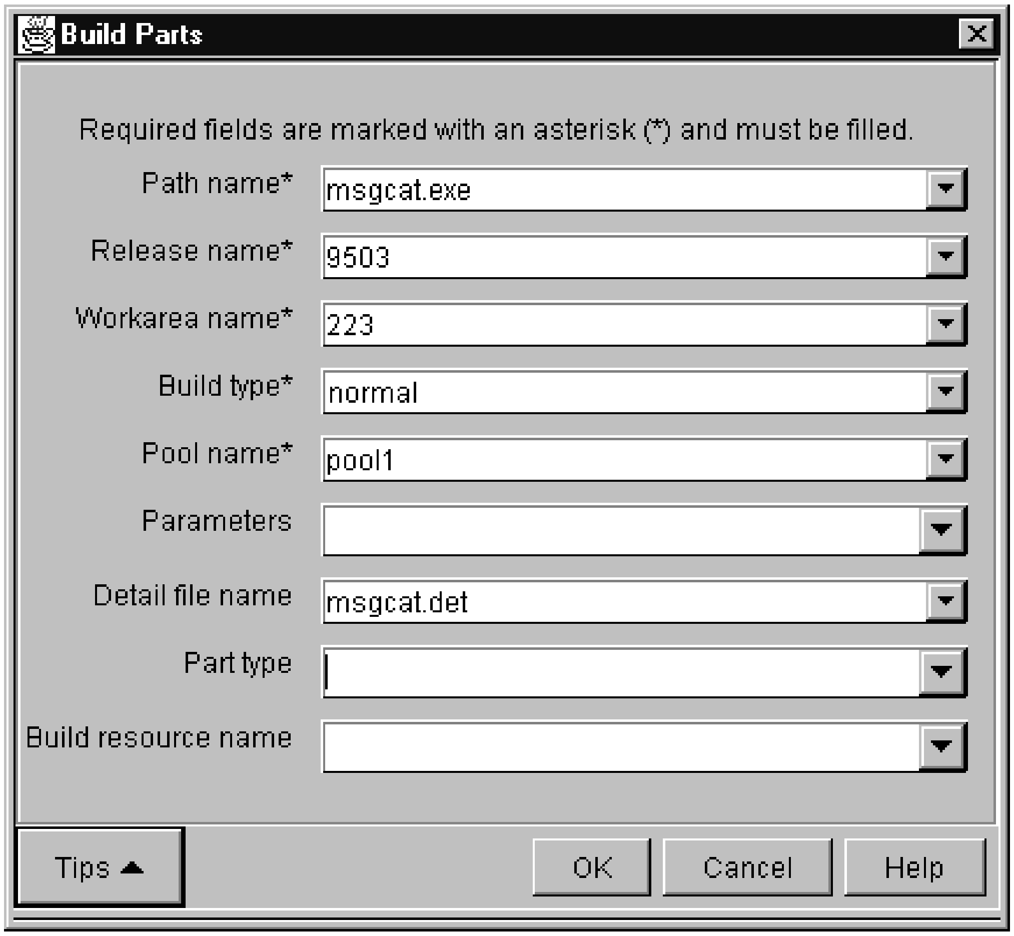
```
teamc part -connect hello.c -parent hello.obj -input -release 9503
  -workarea 223
teamc part -connect bye.c -parent bye.obj -input -release 9503
  -workarea 223
```

The .h parts are not connected because he expects the parser on hello.c and bye.c to find the correct dependencies.

Starting the build on the client

After much hard work on his source code, Greg is ready to start building his application.

Using the GUI, he can start the build by selecting **Part → build** from the Actions menu of the Home Page, and completing the fields as shown in the following illustration:


Build Parts

Required fields are marked with an asterisk (*) and must be filled.

Path name*	msgcat.exe
Release name*	9503
Workarea name*	223
Build type*	normal
Pool name*	pool1
Parameters	
Detail file name	msgcat.det
Part type	
Build resource name	

Figure 67. Build Parts window

Using the command-line interface, he can start the build by issuing the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223
  -pool pool1 -normal -detail msgcat.det
```

This command specifies the following:

Build target
: The name of the part at the top of the build tree, msgcat.exe, which is the final output of this build. TeamConnection uses the build target to determine the scope of the build.

Workarea
: The version of the TeamConnection parts and build tree to be used when performing this build. This version is completely specified by naming the family, release, and workarea: in this case, `-release 9503 -workarea 223`. The output of the build is placed in this workarea.

Build pool
: The set of build servers that should be used to process the build request, as defined when the build servers are started. The pool `pool1` includes the build server started in “Starting the build servers” on page 164.

Build mode
: How the build takes place. Possible values for this build option include the following:

Normal
Builds only the parts that are out-of-date. Processing stops after the first error is returned.

Force Builds all parts, even if they are not out-of-date. Processing stops after the first error is returned.

Unconditional
Builds only parts that are out-of-date but continues processing even if errors are returned. Note that outputs are not rebuilt for inputs that have failed.

Report
Gives a preview of what would be built if you invoked a build. The report identifies what steps would occur without any translations taking place.

In our example, Greg specifies `-normal`, which is the default. In this mode, only the parts that are stale with respect to their inputs are rebuilt. In other words, only the minimum amount of work to bring everything up-to-date is performed. “Running a build in spite of errors” on page 173 and following sections provide examples of using the other build modes.

In normal mode, the build is halted if an error is found. Any remaining build events in the build scope are canceled, but any build events already performed are not undone.

Detail file name
The name of an output file in which TeamConnection stores the collected stdout and stderr of the build scripts.

When Greg starts the build, the information from the command is passed to the testfam family server over TCP/IP. At this point, Greg’s TeamConnection client waits to receive confirmation from the family server that the build request was received and is being processed. Included in this processing; the family server does the second phase of parsing, where the validation of dependency information is done, and then it determines what needs to be rebuilt and adds this to the job queue.

Note: Only one build is allowed in a workarea at one time (though the build events that make up the build might be distributed to different build agents on a number of machines). So if Greg is sharing workarea 223 with Barbara, she cannot issue a part -build command in that same workarea until Greg’s build is complete.

TeamConnection handles the next parts of the build process automatically.

Putting the build scripts to work

At this point, the build server looks at the description of the event it has been asked to perform, then checks its cache for each part and the build script it needs. If it does not find the parts there, or if the cached parts are out of date, it invokes the build script, passing to it the names of the input and output parts and the parameters specified on the builder. The parts created by the build script and the return code generated by it are sent back to the build server, which then updates the contents of the TeamConnection database.

In our example, each of the build servers receives a compile event to perform. Each extracts the .c source files it needs from the TeamConnection database and the contents of the build script for the c_compiler builder. The build servers then run their build scripts.

The results (the .obj files and the return code) are sent back to the build servers. After updating the TeamConnection database, the build servers re-enter the polling loop to see if any more build events await their attention.

Because the compile steps are performed in parallel, Greg can build this application a little more quickly than if they had happened in serial mode. In this simple example, the difference is hardly noticeable; but in a large build of hundreds of parts, with multiple build servers available on a local area network, the performance improvement can be enormous.

Finishing the job and reporting the results to the user

The processing described by the previous two steps is repeated until there are no more build events remaining. The results of the build are displayed in the Build Progress window or in stdout. At this point the build is complete.

To complete our example, the previous two steps are repeated to complete the link step, using either of the build servers in pool1. Greg now can extract the resulting executable from TeamConnection, using the part -extract msgcat.exe command, and run it.

Monitoring the progress of a build

During the course of a build, you can monitor its progress in several ways:

- If the build was started from the command line, by issuing the report -view partview command against the workarea in which you are building. From this report, you can determine the states of the parts. Use the part -viewmsg command to see the build messages issued because of a failed build. For complete syntax of these commands, refer to the *Commands Reference*
- If the build was started from the GUI, in the Build Progress window. You can find the same information by looking at stdout.

Greg can see how the build is progressing by checking the Build Progress window. For example, he might see these messages:

```
6021-700 Number of distinct build events for this build: 3.
Build of 'hello.obj' started at '15:33:47 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'hello.obj' successfully completed at '15:34:45 1995-08-10'.
Completed Jobs: 1
Remaining Jobs: 2
Build of 'bye.obj' started at '15:34:49 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'bye.obj' successfully completed at '15:35:22 1995-08-10'.
Completed Jobs: 2
Remaining Jobs: 1
Build of 'msgcat.exe' started at '15:35:26 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'msgcat.exe' successfully completed at '15:35:56 1995-08-10'.
Completed Jobs: 3
Remaining Jobs: 0
Processing Completed for 'msgcat.exe'.
```

To see the commands that TeamConnection issued during the build, you can look at the detail file specified in the part -build command.

Running a build in spite of errors

If you find that a build is stopping because of errors, you can check the build detail file or the Build Progress window for the cause. If the error is minor, you might decide to run the build despite the errors — for example, when you are debugging. To do this, specify that you want the build to complete unconditionally.

In our example, when Greg builds msgcat.exe for the first time, he wants to find and correct any errors that occur during the build, so he uses the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223 -unconditional
```

As in normal mode, only the parts that are stale with respect to their inputs are rebuilt; only the minimum amount of work to bring everything up-to-date is performed.

However, even if an error is found, the build continues if possible. As with normal mode, if the build is halted, any build events remaining are canceled. Any build events already performed are not undone.

Building all parts, regardless of build times

To make sure that **all** parts in the build tree get built, whether or not they are stale, you specify the -force parameter on the part -build command.

In this mode, all parts that are descendants of the build target are rebuilt.

In our example, Greg can force TeamConnection to build all parts in the msgcat.exe build tree using the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223 -force -pool pool1
```

If an error occurs, the build is halted. Any remaining build events are canceled, but any build events already performed are not undone.

Finding out which parts will be built

Before running a build of a large application, you might want to find out exactly which parts will be built. If you specify that you want to run in `report` mode, TeamConnection determines what will be built in a normal build and produces a report showing the results.

If Greg really wants to see which parts of msgcat.exe will be built before he runs the actual build, he can issue the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea d410 -report -pool pool1
```

He sees the following report:

```
6021-700 Number of distinct build events for this build: 3.
6021-407 The builder c_compiler will be invoked.
6021-406 The builder parameters consist of:
           command:   compC.cmd
           input:     hello.c
           output:    hello.obj
           dependent: hello.h
6021-407 The builder c_compiler will be invoked.
6021-406 The builder parameters consist of:
           command:   compC.cmd
           input:     bye.c
           output:    bye.obj
           dependent: bye.h
6021-407 The builder c_linker will be invoked.
6021-406 The builder parameters consist of:
           command:   linkC.cmd
           input:     hello.obj bye.obj
           output:    msgcat.exe
           dependent:
```

The report shows that bye.obj and msgcat.exe must be rebuilt.

Canceling a build

To cancel a build that is in progress, do one of the following:

- If the build was started from the GUI, on the Build Progress window select the **Cancel Build** push button.
- If the build was started from the command line, type the following command and press Enter:

  ```
  teamc part -build name -cancel
  ```

  Where *name* is the part that you are building. Be sure to specify the same part name that you specified when starting the build, rather than a part that is lower in the build tree.

This command will stop all queued and in_progress builds. Any build events already performed for that build are not undone.

For example, if Greg cancels the build of msgcat.exe when the compile steps have been completed, then the link step is not performed. However, the newly compiled hello.obj and bye.obj are left in the database, with their build times updated.

More sample build trees

The msgcat.exe example is just one possible build tree. Here are some others.

Defining multiple outputs from a single build event

Figure 68 shows part of the build tree for robot.dll:

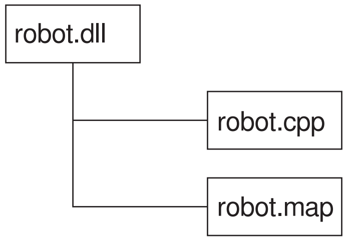


Figure 68. The build tree for robot.dll

Because the build tree shows the relationships between parts hierarchically, robot.map is a child of robot.dll, even though it is actually built from the same input part, robot.cpp. But robot.map is defined as an *output* of robot.dll. Here are the commands to set up this relationship.

First come the commands to create the parts:

```
teamc part -create robot.dll -builder dll_builder -binary -empty
  -release 9503 -component robot
teamc part -create robot.cpp -release 9503 -component robot

teamc part -create robot.map -builder dll_builder -binary -empty
  -release 9503 -component robot
```

Next are the commands to connect the parts into the build tree:

```
teamc part -connect robot.cpp -parent robot.dll -input -release 9503
teamc part -connect robot.map -parent robot.dll -output -release 9503
```

You might use this command to start the build:

```
teamc part -build robot.dll -workarea 915 -release 9503
```

The output of this build would be both robot.dll and robot.map. Any parameters specified in the `teamc part -build robot.dll` command would also apply to the build of robot.map.

Synchronizing the build of unrelated parts

An entire application can require multiple separate builds. For example, in the robot application, there might be one build to create the .dll parts, another to create the .exe parts, and so on. To ensure that the entire application gets built together, you can create a part that acts as a collector, with the .dll and .exe parts as input to it.

For example, Tim creates this build tree for the robot application:

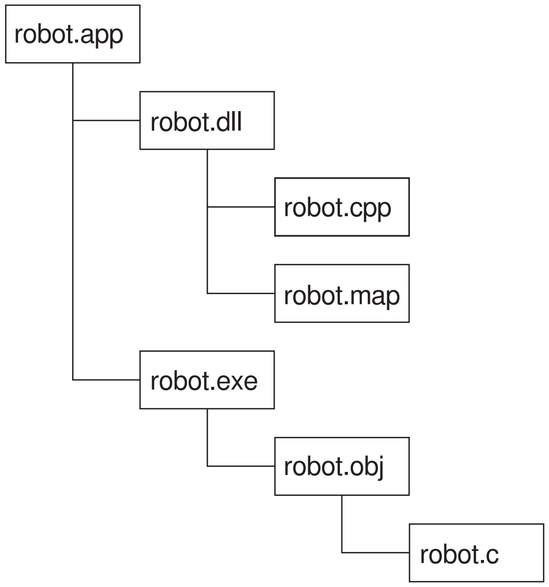


Figure 69. The build tree for robot.app

Assuming he already has the build trees for robot.dll and robot.exe set up, here is how he sets up the collector part:

1. He creates a null builder with no contents:

   ```
   teamc builder -create nullBuilder -script null -none -environment os2
     -condition == -value 0
   ```

2. He creates the collector part:

   ```
   teamc part -create robot.app -builder nullBuilder -none -release 9503
      -component robot
   ```

   The `-none` flag identifies this as a part that will never have any contents.

3. Tim connects the other parts to the collector:

   ```
   teamc part -connect robot.dll robot.exe -parent robot.app -input
      -release 9503
   ```

When Tim builds robot.app, the result is a build of both robot.dll and robot.exe.

Part 5. Using TeamConnection to package products

This section describes how to use the TeamConnection packaging function, which helps you automate the packaging and distribution of your applications. This section is written for the person in your organization who is responsible for software distribution.

Chapter 15. Using TeamConnection to package a product

After you have built an application to your satisfaction, it is time to distribute it to users. This chapter describes how you can use TeamConnection to help automate the packaging and distribution steps.

TeamConnection provides the following:

- Two electronic software distribution tools:
  - **Gather**, which moves an application's parts into a single directory in preparation for distribution.
  - **Tivoli Software Distribution**, a bridge tool that automates the installation and distribution of software or data using Tivoli software distribution tools as the distribution vehicle.
- Two sample build scripts for connecting the Gather and Tivoli Software Distribution tools with TeamConnection user-defined builders.
  - Gather - **gather.cmd** which specifies the `teamcpak gather` command.
  - Tivoli - **softdist** (on UNIX platforms) or **softdist.exe** (on Windows NT) which specifies the `teamcpak softdist` command.

To use TeamConnection in packaging a product, you might do any of the following tasks:

Task	Page
Setting up your application's build tree for packaging	179
Using the teamcpak gather command	183
Writing a package file for the gather tool	186
Using the teamcpak softdist command	"Using the teamcpak command with Tivoli Software Distribution" on page 191
Writing a package file for the Tivoli Software Distribution tool	"Writing a package file for Tivoli Software Distribution" on page 193

Setting up your build tree for packaging

When TeamConnection builds an application, the application's build tree identifies the parts to be built and the tools to use in building it. Similarly, when you use TeamConnection for packaging the application, the build tree can define the parts to be packaged and the tools to do it.

The output of a packaging step might be any of the following:

- The application's parts gathered into a new directory structure
- The distribution of the application using NVBridge
- The distribution of the application using some other distribution tool

Setting up a build tree for the gather tool

To gather the parts of your application into a single directory for distribution, you create an output part whose builder calls the gather tool, and you make this output part the top level of the build tree.

For example, for the robot control application, robot.app, the build tree might look in part like this:

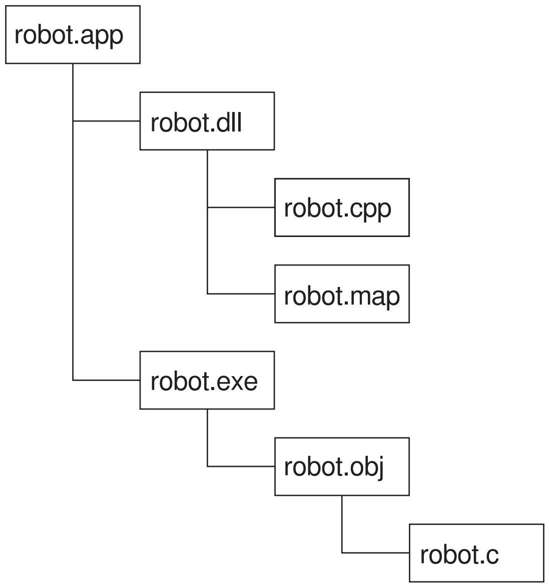


Figure 70. Part of the build tree for robot.app

After the application is built, the programming team needs to get it to the test team. They could extract the application, but doing a simple extract would preserve the existing structure, with parts contained in directories according to their application component. A better structure might be to place all of the .dll files in one directory, all of the .exe files in another, and so on. To move the parts into this structure, the test team does a different kind of build, using the gather tool.

To make this happen, Annmarie does the following:

1. She creates the top-level part for the new build tree. The name of this part is the same as the directory in which the gathered parts are to be placed. In this example, e:\robot is the output file from the gather step. Annmarie uses the following command:

   ```
   teamc part -create e:\robot -none -builder gather1 -family octo
      -release 9503 -workarea 410
   ```

2. She writes a package file that contains instructions for the gather tool and creates this file as a TeamConnection part:

   ```
   teamc part -create robot.pkf -text -parent e:\robot -input -family octo
      -release 9503 -workarea 410
   ```

   For more information, see “Writing a package file for the Gather tool” on page 186.

3. She creates a builder, gather1, that calls the gather tool:

   ```
   teamc builder -create gather1 -script gather.cmd
      -parameters "-o -x"  -release 9503
      -environment os2 -condition == -value 0 -family octo
   ```

gather.cmd is a sample build script that is shipped with TeamConnection. It specifies the `teamcpak gather` command.

4. She connects robot.exe and robot.dll to e:\robot as inputs:

```
teamc part -connect robot.exe -parent e:\robot -family octo
   -release 9503 -workarea 410

teamc part -connect robot.dll -parent e:\robot -family octo
   -release 9503 -workarea 410
```

5. She also connects a readme file for the application:

```
teamc part -connect read.me -parent e:\robot -family octo
   -release 9503 -workarea 410
```

As a result of Annmarie’s work, the build tree for e:\robot looks like this:

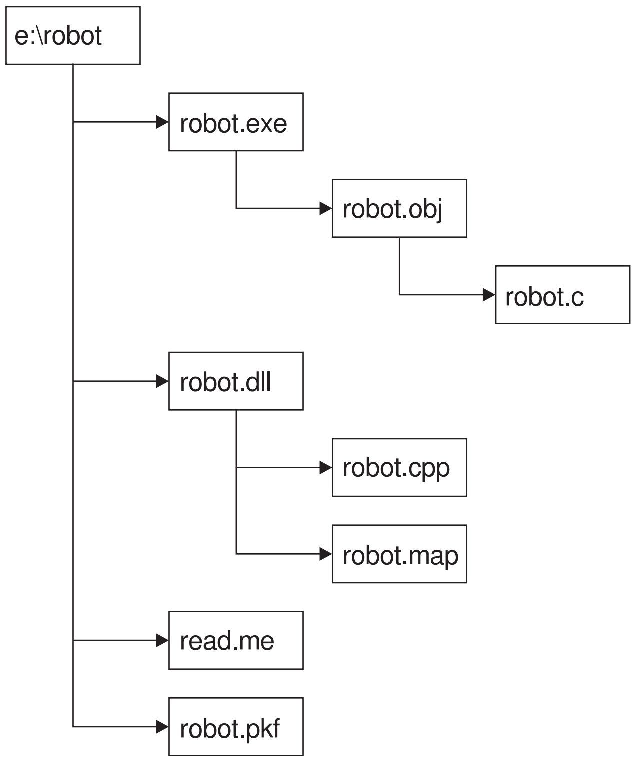


Figure 71. Adding the gather step to the build tree

The package file, robot.pkf, specifies the directories into which the robot files are gathered, with e:\robot as the target root directory. When Annmarie builds e:\robot, the .dll files are placed in e:\robot\dll; the .bin files are placed in e:\robot\bin. Instead of extracting the built application from TeamConnection, the test team can pull the application from e:\robot.

If Annmarie wants to gather the same files into a different target directory, all she needs to do is write a different package file and connect the parts to a different parent.

Chapter 16. Using the Gather tool

The Gather tool automates the movement of software and data from one directory to another on the same machine to prepare a package for electronic distribution. It can copy or erase files; it can create or delete directories.

This tool takes a list of input files and moves them into a directory structure as directed by a package specification file. You specify the target root directory path in this file, along with a collection of rules that instruct which files to copy to which directories. How these files and directories are actually handled is controlled via option flags.

By writing different package specification files, you can take the same input files and transfer them into different target directory structures.

Take the robot application as an example. We previously showed one possible directory structure, with each subdirectory containing files with the same extension:

```
e:\robot
   \dll
       hand.dll
       optics.dll

   \exe
       hand.exe
       optics.exe
```

By writing a different package file, you might put both .dll and .exe files in the same target directory:

```
f:\robot
    \bin
      hand.dll
      optics.dll
      hand.exe
      optics.exe
```

You can build both target directories concurrently.

Using the teamcpak command for the Gather tool

To start the Gather tool, use the teamcpak command. This command is found in the directory where the TeamConnection family server is installed. If it is started from a build script, it does not need to be in the execution path of the machine from which the build is started.

The complete command syntax for teamcpak gather looks like the following; you must supply a value for the words that start with a capital letter, such as String. You must specify the command parameters in the order shown.

```
teamcpak [-i] [-o "String"] gather Input_file...
```

Where

-i Specifies that only one *Input_file* is specified in the command: an include file containing the list of input files. This parameter is optional.

If you specify `-i`, it must precede the `gather` flag.

-o " String"
Specifies that the string listed in quotes be passed to the Gather tool. The opening quote must be followed by a blank. For a list of possible flags to be passed, see “Command line flags”.

This parameter is optional. If you do not specify `-o`, the default settings for the tool are used.

If you specify `-o`, it must precede the `gather` flag.

gather
Specifies the tool to be invoked. If you specify `-i` or `-o`, they must precede this value.

Input_files
Specifies the files to be copied and the name of the package specification file. You can specify this parameter in these ways:

- Specify the name of an include file, whose contents is a list of input files. One of these input files must be a package specification file with the extension .pkf. In this case, you must also specify the `-i` parameter.
- Specify a list of two or more files. One of the files must be a package specification file with the extension .pkf.
- Specify the directory from which the files are to be copied and the name of the package specification file.

If more than one package file is listed, the first package file on the command line or in the include file is used, and the others are treated as ordinary files.

Command line flags

You can specify the following flags in the teamcpak command, using the `-o` parameter. All of these flags are optional. If you do not specify a flag, the teamcpak command runs using defaults.

-a Assume that the target tree structure might not exist. If a required directory does not exist, create it and continue processing.

This flag cannot be specified if the `-t` flag is specified.

If neither `-a` nor `-t` is specified, the default is to assume that the desired tree structure already exists. No verification is performed to confirm that the directories exist. If they do not, the condition is detected while the package file rules are being processed. If you stop the teamcpak command, some target directories might contain updated files.

-f Force deletion from the root: if this is used in combination with `-t` or `-c` and TARGETROOT is a root directory (for example, `e:\, \, /`).

-t Ensure that the target tree is exactly the tree specified in the package file. If a directory of the same name exists, the Gather tool does the following:

- Erases the entire contents of the directory and all of its subdirectories
- Destroys the directory and all subdirectories
- Performs a mkdir command to create the entire tree structure again as specified in the package file

This flag cannot be specified if the `-a` flag is specified.

If an rmdir command fails during processing, the teamcpak command stops.

If neither `-a` nor `-t` is specified, the default is to assume that the desired tree structure already exists. No verification is performed to confirm that the directories exist. If they do not, the condition is detected while the package file rules are being processed. If you stop the teamcpak command, some target directories might contain updated files.

-m Accept missing source files.

If this flag is not specified, the default is to ensure that at least one file matches each source specification in the package file. If a match is not found, the Gather tool stops processing.

-d Accept duplicate files. If a file is found on the target directory that matches the source file specification, it is overwritten by the source file.

If this flag is not specified, the default is to ensure that no files on the target match the source file specification. For example, if the source specification is g*.c, and greg.c is found on the target, the Gather tool stops processing.

-c Clean up the target directories. Erase all files on all target directories that existed before writing source files to these directories. No confirmation messages are issued, and permission errors are ignored.

If this flag is not specified, the default is to write the source files into the target directories without erasing existing files.

-e End with delete. This action removes all source files and directories after the Gather tool successfully completes.

If this flag is not specified, the default is to end without deleting source files and directories.

-x Abort without recovery. If the program does not end successfully, no attempt is made to restore the file system.

If this flag is not specified, the Gather tool attempts to restore the file system if the program does not end successfully. To do this, the tool first backs up the file system. The backup directory is the value of the TMP environment variable.

Examples of the teamcpak gather command

The following are examples of the teamcpak gather command.

```
teamcpak gather d:\demoapp demoapp.pkf
teamcpak gather a.exe b.exe \help\*.hlp demoapp.pkf
```

In the first example, an input source directory is specified. In the second example, a list of files is specified. In both cases, the files are to be copied into target directories as specified in the demoapp.pkf file.

```
teamcpak -i -o " -t -m -x" gather myfiles.lst
```

The file myfiles.lst contains a list of files to be transformed by the Gather tool, and the name of the package file to be used in the gather. The `-o "-t -m -x"` parameter passes three flags to the Gather tool:

- `-t` specifies that, if the target directories already exist, they be destroyed and recreated.
- `-m` specifies that processing continues even if a source file cannot be found.
- `-x` specifies that, if the program does not end successfully, the file system is left as is, with no attempt to restore it.

Writing a package file for the Gather tool

Use the package file to specify the target directories and the rules for copying files for a gather operation. You can also specify user exit programs to run before, during, or after the gather operation.

A sample package file named gather.pkf is shipped with TeamConnection. You can customize it for your own gather operations.

Syntax rules for a Gather package file

Follow these syntax rules when you write a package file:

- Package files are free format. Text is not positional, and many statements can exist on the same line.
- Comments can appear anywhere within the file. Use the characters `#|` and `|#` as delimiters, as shown in the following example:

  ```
  #| This is a comment |#
  ```
- Package file keywords must be prefixed with a left parenthesis and must have a corresponding balanced right parenthesis to end the scope of the keyword.
- If the value for a keyword is a string that contains blanks or parentheses, enclose the string in double quotes.

The following shows the syntax of a package file for the Gather tool. Keywords must appear in the order shown. The first letter of an argument is capitalized; you must supply these values.

```
(DATA
  (PACKAGEFORMAT gather)
  (TARGETROOT Filename)
  (RULE
    (SOURCE Filename...)
    (TARGET Path)
    [(EXITPRIOR String... | EXITREPLACE String... | EXITPOST String...)] )
    )
    .
    .
  [(EXITPRIOR String...)]
  [(EXITPOST String...)]
)
```

Keywords for a Gather package file

DATA This keyword is required. It must be the first keyword in the package file, and it can be specified only once.

All other keywords are nested within the DATA clause.

PACKAGEFORMAT gather
This keyword is required. It can be specified only once. It tells the teamcpak command that this package file is for Gather.

TARGETROOT target_root_path
This keyword is required. It can be specified only once.

Use this keyword to identify the target root directory. Source files are copied to this directory as specified by the RULE statements.

Follow these guidelines when you select your TARGETROOT values:

- Include the drive letter along with the target directory.
- Specify a directory that contains few if any subdirectories that are unrelated to the data you are moving.
- If you specify a drive's root directory (`drive:\`), run the teamcpak command using the defaults or only the `-x` or `-x -a` flags.
- Do not set the value of TARGETROOT to `drive:\` under the following circumstances:
  - The TARGETROOT drive is the same as the drive from which the teamcpak command is run, and you have recovery set (that is, you have not specified `-o "-x"`).
  - The logical drive for the TARGETROOT has less than 50% free space, and you have recovery set (that is, you have not specified `-o "-x"`).

RULE This keyword is required. You can use one or more RULE keywords within a Gather package file.

Each RULE clause represents a set of Gather operations targeted for one target subdirectory. A RULE clause must contain one SOURCE and one TARGET keyword. The files in the SOURCE directory are copied to the TARGET path. The target path is derived by concatenating the value of TARGETROOT with a backslash (\), followed by the value of the TARGET keyword specified in the RULE clause.

A RULE clause can also contain one user exit clause: EXITPRIOR, EXITPOST, or EXITREPLACE. For a description of the exit keywords, go to page 189.

The following example copies all *.exe, *.cmd, and *.hlp files to target directory f:\demoapp\bin.

```
(DATA
   .
   .
   (TARGETROOT  f:\demoapp )
   .
   .
   (RULE
      (SOURCE  *.exe   *.cmd  *.hlp)
      (TARGET  bin   )
   )
   .
   .
)
```

SOURCE <list of file specifications>

This keyword is required once for each RULE clause. It must be the first keyword within the RULE clause.

This keyword specifies the files to be copied to the path specified by the TARGET keyword. Specify a list of file specifications separated by blanks. You can use the wildcard characters supported by OS/2 or Windows NT.

The directory from which these files are copied depends on how the input files are specified in the teamcpak command:

- If the teamcpak command specifies a source directory, the files specified in the SOURCE keyword come from that directory or subdirectories of it. The full path of the source files is constructed by concatenating the directory from the teamcpak command with a backslash (\), followed by the file specifications found in the SOURCE keyword. You can specify subdirectories in the SOURCE file specifications.

- If the teamcpak command specifies a list of files, these files are first copied to a temporary directory, then copied from there to the TARGET directories. In this case, you can use OS/2 or Windows NT wildcards to specify multiple file names in the SOURCE file specifications, but you cannot specify subdirectories.

In the following example, directory d:\demoapp is specified on the teamcpak command:

```
teamcpak -o "-x -t -m" gather d:\demoapp demoga.pkf
```

The resulting source path is the concatenation of d:\demoapp with the SOURCE file specifications. Therefore, all of the .exe files in the directory d:\demoapp\bin are copied to the target directory e:\demoapp\bin.

```
(DATA
   (TARGETROOT  e:\demoapp)
   .
   .
   (RULE
      (SOURCE  bin\*.exe)
      (TARGET bin)
   )
   .
   .
)
```

In the following example, a list of input files is specified on the teamcpak command:

```
teamcpak -o "-x -m" gather c:\a.exe c:\b.exe d:\rexx\*.cmd demoga.pkf
```

The resulting source path for the files in the SOURCE clause is the concatenation of the teamcpak temporary directory with the SOURCE file specifications. Therefore, the source for the *.exe files is d:\teamcpak.@@@\*.exe. The input files d:\teamcpak.@@@\a.exe and d:\teamcpak.@@@\b.exe are copied to the directory e:\demoapp.

```
(DATA
   (TARGETROOT  e:\demoapp)
   .
   .
   (RULE
      (SOURCE  *.exe    )
      (TARGET targetroot)
   )
   .
   .
)
```

TARGET Target_path

This keyword is required one time in each RULE clause. It must follow the SOURCE keyword.

The value specified by this keyword is used to construct the target path into which the files specified by the SOURCE keyword are copied. The value of the TARGETROOT keyword is concatenated with a backslash (\), followed by the value of the TARGET keyword.

If you specify `targetroot` as the value, files are copied directly to the target root directory, not to a subdirectory.

In the first RULE clause of this example, files are copied to the target directory f:\demoapp\bin\files. In the second RULE clause, the target directory is f:\demoapp.

```
(DATA
   (TARGETROOT  f:\demoapp )
   .
   .
   (RULE
      (SOURCE  *.bin *.dll     )
      (TARGET  bin\files )
   )
   (RULE
      (SOURCE  *.hlp           )
      (TARGET  targetroot )
   )
   .
   .
)
```

EXITPRIOR, EXITPOST, and EXITREPLACE String...

These keywords are optional. They specify a user exit program to run as part of the gather operation.

To specify an exit that is global to the Gather operation, specify EXITPRIOR or EXITPOST in the DATA clause. You can specify each of these keywords only once in the DATA clause. These keywords must come after all of the RULE clauses. EXITREPLACE cannot be used in the DATA clause.

You can also specify an exit that is specific to one RULE clause. Only one exit keyword is allowed in each RULE clause.

These keywords accept a list of strings separated by spaces. The first string is the name of the program to execute. The strings that follow are its parameters.

Using exit keywords in the DATA clause

When used within a DATA clause, these keywords identify a program or command to be executed within a command shell. EXITPRIOR executes before all RULE statements have been processed; EXITPOST, after all RULE statements.

The exit keywords accept any executable file or command. The exit program must return an integer return value, with zero meaning the exit was successful.

Using exit keywords in the RULE clause

EXITPRIOR, EXITPOST, and EXITREPLACE are optional within a RULE clause. Only one can be specified in any given RULE clause.

When used within a RULE clause, these keywords identify a program or command to be executed within a command shell before, after, or in place of processing of each Gather copy operation. The exit program is called once for each SOURCE specification entry within the SOURCE clause. Parameters are separated by spaces and passed to the exit in this order:

- Any parameters included in the invocation string
- The resolved SOURCE file specifications
- The resolved TARGET specification

The exit keyword accepts any executable file or command. The exit program must return an integer return value, zero meaning successful; it must also accept or ignore the additional Gather parameters added to the end of the invocation string.

When used in the context of the RULE clause, exit keywords must follow the TARGET keyword.

Using exit keywords: an example

In the following example, the first EXITPRIOR statement relates to the DATA clause and specifies a user backup exit program, which executes before performing Gather copy operations. This backup exit is passed two flags. The command stream executed in an OS/2 shell is:

```
"e:\util\backup.cmd \i \t"
```

The second occurrence of the keyword illustrates how to use it in the context of a RULE clause. In this example, an encryption program will run against each source file specification. The exit program is passed the \k:347867 key option, the value for the source specification, and the value for the target specification. In this example, the command stream executed in an OS/2 shell is:

```
"encrypt \k:347867 d:\demoapp\a.exe f:\demoapp\bin":
```

The package file looks like this:

```
(DATA
        (PACKAGEFORMAT gather)
        (TARGETROOT d:\tcws)
        (RULE
                (SOURCE *.exe *.cmd)
                (TARGET exe)
                #|this program will be run for each source file|#
                (EXITPRIOR encrypt \k:347867 )
        )
        (EXITPRIOR "e:\util\backup.cmd \i \t" )

)
```

Chapter 17. Using the Tivoli Software Distribution packaging tool

The Tivoli Software Distribution packaging tool supports automated distribution between a single Tivoli Software Distribution server and its TCP/IP-connected clients. The Tivoli Software Distribution tool works either by itself or in conjunction with TeamConnection's Gather tool to enable you to distribute files through Tivoli Software Distribution. To use this tool, you must be familiar with Tivoli configuration and system administration so that TeamConnection can start Tivoli Software Distribution to distribute file packages.

The Tivoli Software Distribution distribution tool must be run on a Tivoli managed node running on any of TeamConnection's UNIX platforms or Windows NT.

The Tivoli Software Distribution distribution tool includes a sample build script named softdist (on UNIX platforms) or softdist.exe (on Windows NT). It can be run from within a TeamConnection builder. This build script maps TeamConnection build parameters to the command line syntax for the Tivoli Software Distribution tool through the teamcpak command line interface.

You can use Tivoli Software Distribution as a builder for packaging in two ways:

- Integrate it with the gather step, so that the Gather tool leaves the package files in a directory from which Tivoli Software Distribution picks them up.
- Use it without the gather step. In this case, the build script for Tivoli Software Distribution must set up the directory and move files into it to interface correctly with the teamcpak command.

To simplify the interface, the Tivoli Software Distribution tool uses a select set of options. If you want to take full advantage of Tivoli Software Distribution features, you can import a Tivoli Software Distribution package specification. Importing a package specification provides you access to all Tivoli Software Distribution functions.

The Tivoli Software Distribution tool produces a Tivoli File Package, which is used for distribution.

Using the teamcpak command with Tivoli Software Distribution

To start the Tivoli Software Distribution tool, use the teamcpak command. This command is found in the directory where the TeamConnection family server is installed. If it is started from a build script, it needs to be in the execution path of the build server.

The complete syntax of the teamcpak softdist command follows. You must specify the command parameters in the following order:

```
teamcpak [-i] [-o "string"] softdist inputFile
```

-i Specifies that only one *inputFile* is specified in the command: an include file containing the list of input files. This parameter is optional.

-o "string"
Specifies that the string listed in quotes be passed to the Tivoli Software Distribution tool. For a list of possible flags to be passed, see "Command line flags" on page 192.

inputFile

Specifies the files to be copied and the name of the package specification file. You can specify this parameter in these ways:

- Specify the name of an include file, whose contents is a list of input files. One of these input files must be a package specification file with the extension .pkf. In this case, you must also specify the `-i` parameter.
- Specify a list of two or more files. One of the files must be a package specification file with the extension .pkf.
- Specify the directory from which the files are to be copied and the name of the package specification file.

If more than one package file is listed, the first package file on the command line or in the include file is used, and the others are treated as ordinary files.

The following are examples of specifying input files.

```
teamcpak -i softdist myInputFile
teamcpak softdist d:\inputDir\myPkfFile.pkf inputFile1 inputFile2 . . .
```

Command line flags

You can specify the following flags in the teamcpak command, using the `-o` parameter. All of these flags are optional.

-a Create directories on the target.

-c Clear the target (delete all specified files and directories) before the apply. If you use this option, do not use the -x option.

-t Overwrite existing files (delete specified files on the target prior to distribution).

-m Accept input errors, such as missing files and directories from the SOURCE keyword.

-n Send no notices to Tivoli. If you want to post Tivoli notices, you must configure Tivoli Notices before using this packaging tool.

-p Preview only; do not actually distribute files.

-r Reboot the target after distribution.

-x If an error occurs, leave any distributed files on the target; do not clean up. If you use this option, do not use the -c option.

-k Keep the Tivoli file package. To enable the Tivoli Software Distribution tool to perform more efficiently, the Tivoli Software Distribution package file is created when the package part is created and then destroyed and recreated whenever the part is modified. Use the -k option to prevent the package file from being destroyed.

Example of the teamcpak softdist command

The following is an example of the teamcpak softdist command.

```
teamcpak -i -o "-a -n -t" softdist Client.lst
```

The -i parameter specifies that the input file Client.lst is to be used. The -o parameter passes the following options to Tivoli Software Distribution:

- `-a` creates directories on the target.

- `-n` indicates that no error notices are to be sent to Tivoli Software Distribution.
- `-t` indicates that any existing files on the target are to be overwritten.

Writing a package file for Tivoli Software Distribution

This section describes the Tivoli Software Distribution package file keywords and their effect on normal processing behavior.

A sample package file named client.pkf is shipped with TeamConnection. You can customize it for your own use.

Syntax rules for a Tivoli Software Distribution package file

Follow these syntax rules when you write a package file:

- Package file keywords must appear in the order shown below.
- Package file keywords must be prefixed with a left parenthesis and must have a corresponding balanced right parenthesis to end the scope of the keyword.
- If the value for a keyword is a string that contains blanks or parentheses, enclose the string in double quotes.
- Default options are supplied for all Tivoli Software Distribution required Tivoli Software Distribution options. Specific options can be set in your TeamConnection package file.
- Comments can appear anywhere within the file. Use the characters `#|` and `|#` as delimiters, as shown in the following example:

  ```
  #| This is a comment |#
  ```

The following shows the syntax of a package file for Tivoli Software Distribution. The keywords must appear in the order shown here. You must supply the values for the strings that are shown in *italics*.

```
(DATA
  (PACKAGEFORMAT softdist)
  (TARGETROOT filename)
  (MANAGER ProfileManager)
  (NODES "ManagedNode... PCManagedNode...")
  (IMPORT filename |
      [(DISTRIBUTE [FULL | CHANGED])
      [(INSTALLPGM filename)]
      [(LOGNODE ManagedNode)]
      [(LOGFILE directory)]
   )
   )
```

Keywords for a Tivoli Software Distribution package file

DATA This keyword is required. It must be the first keyword in the package file, and it can be specified only once.

All other keywords are nested within the DATA clause.

Example:

```
(DATA
   .
   .
   other keywords go here
   .
   .
)
```

PACKAGEFORMAT softdist

This required keyword must be the first keyword within the DATA clause. It can be specified only once. It tells the teamcpak command that this package file is for Tivoli Software Distribution.

Example:

```
(DATA
   .
   .
   (PACKAGEFORMAT softdist)
   .
   .
```

TARGETROOT

This keyword specifies the directory path to which files are to be distributed on the target systems. You can specify only one target root. All target systems use identical target roots.

Example:

```
(DATA
   .
   .
   (TARGETROOT /usr/local/teamc/images)
   .
   .
```

MANAGER

This keyword specifies a Tivoli Software Distribution profile manager that you have already created in the Tivoli Software Distribution system.

Example:

```
(DATA
   .
   .
   (MANAGER Distrib1)
   .
   .
```

NODES

This keyword specifies the nodes to which the files are to be distributed. These must already have been defined to the profile manager as subscriber ManagedNodes or PCManagedNodes. To distribute files to non-subscribers, you need to use Tivoli Software Distribution options set in an import file package definition.

Example:

```
(DATA
   .
   .
   (NODES "tcaix01 tcaix02")
   .
   .
```

IMPORT

Use this keyword to select Tivoli Software Distribution options not supported in the -o parameter of the teamcpak softdist command. The *filename* parameter is the name of a Tivoli Software Distribution import file package definition. You can generate an import file using the Tivoli Software Distribution user interface. If you use the IMPORT keyword, then instead of calling the standard Tivoli Software Distribution packaging command the Tivoli Software Distribution tool will call wimprtfp to get all of the Tivoli Software Distribution configuration options. Using the IMPORT keyword disables other options and causes errors if they are specified.

If you specify the IMPORT keyword, do not specify the DISTRIBUTE, INSTALLPGM, LOGNODE, or LOGFILE keywords.

If you use the INCLUDE option in an import file, it is overridden by the list of files provided to the teamcpak command.

Example:

```
(DATA
   .
   .
   (IMPORT importFile)
   .
   .
```

DISTRIBUTE
Specify FULL to distribute all files or CHANGED to distribute only those changed since the last distribution. The default is FULL.

Example:

```
(DATA
   .
   .
   (DISTRIBUTE CHANGED)
   .
   .
```

INSTALLPGM
Use this keyword to specify an installation script to be run during distribution on each node that receives files. Specify the full file path name of the script.

Example:

```
(DATA
   .
   .
   (INSTALLPGM /tivoli/fpTeamcAIX/tcinstl.ksh)
   .
   .
```

LOGNODE
This keyword specifies the system on which the log file is located. The node name you specify must be a managed node. The default is the current build machine or a machine running teamcpak.

Example:

```
(DATA
   .
   .
   (LOGNODE tcaix04)
   .
   .
```

LOGFILE
This keyword specifies the directory path and file name of the log file on the log node. If you use the LOGNODE keyword, this keyword is required.

Example:

```
(DATA
   .
   .
   (LOGFILE /tmp/softdist.log)
   .
   .
```

Problem determination for the Tivoli Software Distribution tool

If you are having trouble distributing files using the Tivoli Software Distribution distribution tool, you can use the following tools or teamcpak options to determine what the problem is:

Log file
Check the file name you specified in the LOGFILE keyword for error messages.

Mail
Check Tivoli mail messages generated during the distribution.

-k option
Run the teamcpak command with the -k option to keep the package file after the distribution has been run. This allows you to reprocess a distribution from the Tivoli GUI and test variations.

-x option
Run the teamcpak command with the -x option to leave any distributed files on the target.

Trace facility
Run teamcpak with the trace facility. Use this facility only under guidance of an IBM service representative. See the *Administrator's Guide* for more information.

The following message displays when a Tivoli Software Distribution command fails during a distribution.

```
6022-303 Tivoli/Software Distribution %s command failed with return code: RC.
To correct problem use:
- package file parameters LOGNODE and LOGFILE to record Tivoli output,
- packaging option "-k" to keep Tivoli File Package and teamcpak log file
  or "-m" to ignore input errors,
- packaging option "-x" to not clean up files that are distributed,
- TeamConnection Trace facility (see TeamConnection Administration Guide)
- or Tivoli Trace facility (see Tivoli documentation)
```

Sample package file

The following is an example of scripts and items required to automatically execute packaging, distribution, and installation of files in a AIX-based system.

- The **teamcpak** command syntax that will execute subcommands or scripts for the package, distribute, and install functions.

  ```
  teamcpak -i -o "-a -n -t" softdist Client.lst
  ```

- The **Client.pkf** file you create containing keywords and parameters for distributing and packaging functions.

  ```
  (DATA
     (PACKAGEFORMAT softdist)
     (TARGETROOT /user/local/teamc/images)
     (MANAGER Distribi)
     (NODES perlovrs tcaix02)
     (INSTALLPGM /tivoli/fpTeamcAIX/tcinstall.ksh)
     (LOGNODE tcaix00)
     (LOGFILE /tmp/fpTeamcAIX.log)
  )
  ```

- The **Client.lst** file you create containing the list of files passed to **teamcpak**. The first line contains the package file by convention. The example also contains customized installation files (**tcinstall.ksh**), TeamConnection tar files, and an installation script (**tcinst.ksh**).

  ```
  /usr/teamc/tivoli/Client.pkf
  /usr/teamc/tivoli/tcinstall.ksh
  /tcinstall/v208/fullpak/aix4/tar/client.tar
  /tcinstall/v208/fullpak/aix4/tar/msgen_us.tar
  /tcinstall/v208/fullpak/tcinst.ksh
  ```

The following presents an example of a Tivoli installation script (**tcinstall.ksh**) that is copied to the target along with the tar files and the TeamConnection installation script (**tcinst.ksh**), then executed on each NODES entry.

```
#!/bin/ksh
# Clear existing log
#
# You can easily update these.....
#
INST_DIR=/usr/local/teamc
INST_TMP=${INST_DIR}/tcinstl.tmp
INST_OUT=${INST_DIR}/tcinstl.out
INST_ERR=${INST_DIR}/tcinstl.err
INST_LOG=${INST_DIR}/tcinstl.log
IMAGE_DIR=${INST_DIR}/images

rm -f $INST_ERR $INST_OUT $INST_LOG $INST_TMP >/dev/null 2>&1
mkdir -p ${IMAGE_DIR}

exec 1>${INST_ERR}
exec 2>&1
exec 3>${INST_LOG}

# Install VisualAge TeamConnection using responsefile
print -u3 Starting VisualAge TeamConnection installation at ′date′
print -u3 User id= ′id′
print -u3 Input: $*

# Set up installation environment
# - assumes bourne or korn shell
grep DB2_ROOTDIR '/.profile'
if (($? != 0))
then
   print -u3 Updating /.profile
   exec 4>>/.profile
   cd /
   print -u3 'DB2 and TeamConnection settings'
   print -u4 'DB2_ROOTDIR=/usr/lpp/ODI/DB24.0'
   print -u4 'export DB2_ROOTDIR'
   print -u4 'PATH=$PATH:$DB2_ROOTDIR/cset/bin'
   print -u4 'export PATH'
   print -u4 'LIBPATH=$LIBPATH:$DB2_ROOTDIR/common/lib'
   print -u4 'export LIBPATH'
   . /.profile
else
   print -u3 /.profile already updated
fi

# Set up error logging
# - if *.warning is in file (preceeded by spaces and tabs only
grep "^[]*\*.warning" /etc/syslog.conf
if (($? != 0))
then
   print -u3 'Updating /etc/syslog.conf'
   touch /var/spool/syslog
   chmod 666 /var/spool/syslog
```

```
   exec 4>> /etc/syslog.conf
   print -u4 '*.warning /var/spool/syslog'
   stopsrc -s syslog
   startsrc -s syslog
else
   print -u3 /etc/syslog.conf already updated
fi

# Update services file for tcocto family
grep "tcocto" /etc/services
if (($? != 0))
then
   print -u3 Updating /etc/services
   exec 4>> /etc/services
   print -u4 'tcocto  8888/tcp'
else
   print -u3 /etc/services already updated
fi

# Generate response file
###
### You can change to use enviroment variables!!
###
print -u5 '1'
print -u5 '5'
print -u5 '/usr/local/teamc/images'
print -u5 '/usr/local/teamc'
print -u5 '/usr/local/teamc/nls'
print -u5 'en_US'
print -u5 '/usr/local/teamc/X11'
print -u5 ''
print -u5 'i'

# Run provided TeamC install script
ls -laR ${IMAGE_DIR} >> ${INST_TMP} 2>&1
cd ${IMAGE_DIR}
${IMAGE_DIR}/tcinst.ksh < ${IMAGE_DIR}/tcinstl.response
if (($? != 0))
then
   # Failed installation
   print -u3 TeamC installation failed
   exit 1
else
   # Clean up installation directory after listing contents
   print -u3 We have successfully copied TeamC installation files
   print -u3 Installation directory contents:
   ls -laR ${INST_DIR} >> ${INST_TMP} 2>&1
fi

cd /
# Remove installation stuff
print -u3 TeamC cleaning up temporary installation directory
rm -rf ${IMAGE_DIR}
cat ${INST_TMP} >> ${INST_LOG}
rm -rf ${INST_TMP}
exit 0

# end of file
```

Part 6. Appendixes

Appendix A. Environment Variables

You can set environment variables to describe the TeamConnection environment in which you are working. Environment variables provide default settings and behaviors for many TeamConnection actions and processes. You can override the value you set for many of these variables by using the corresponding flag in a TeamConnection command or field on the TeamConnection GUI.

Some environment variables can be set either by your operating system (such as in your config.sys file or .profile) or by the TeamConnection Settings notebook. When an environment variable has a Settings notebook equivalent, TeamConnectionuses the two as follows:

- The environment variable controls the command line interface.
- The Settings notebook controls the graphical user interface.

If there is no Settings notebook equivalent for the environment variable, then the environment variable takes effect regardless of the interface you are using. See "Setting environment variables" on page 207 for more information about setting environment variables.

You are not required to set your TC_FAMILY environment variable for the TeamConnectionclient command line interface. However, if the TC_FAMILY environment variable is not set, the `-family` must be specified for every client command.

The following table lists the names of the TeamConnection environment variables, the purpose they serve, the equivalent TeamConnection command-line flag, the equivalent Settings notebook field, and the TeamConnection component that uses each environment variable.

Table 2. TeamConnection environment variables

Environment variable	Purpose	Flag	Setting	Used by
LANG	Specifies the language-specific message catalog.			Client, family server
NLSPATH	Specifies the search path for locating message files.		NLS path	Client, family server
PATH	Specifies where tcadmin is to search for the family create utilities.			Client, build server, family server
TC_BACKUP	Controls whether or not the following commands create backup files when a read-only copy of the file already exists on your workstation. If this environment variable is set to off or OFF, the commands do not create backup files. • builder -extract • part -checkout • part -extract • part -merge • part -reconcile			Family server

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_BACKUPCHAR	Specifies the character to be interted in the file name extension when TeamConnection creates a backup copy of a file during checkout and extract actions. The default backup characters are $ on Intel platforms and _ on UNIX platforms. If you check out or extract a file called myfile.ext, for example, and a read-only copy of this file already exists on your workstation, TeamConnection creates a backup copy called myfile.$ext or myfile._ext. On file systems requiring 8.3 file names (such as FAT file systems), the file extension is truncated to three characters (myfile.$ex or myfile._ex).			Family server
TC_BECOME	Identifies the user ID you want to issue TeamConnectioncommands from, if the user ID differs from your login. You assume the access authority of the user ID you specify.	-become	Become user	Client, build server (except mvs)
TC_BUILDENVIRONMENT	Specifies the build environment name, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive.	-e		Build server
TC_BUILDMINWAIT	Minimum amount of time to wait (in seconds) between queries for new jobs. Default setting is 5, minimum setting is 3.			Build server
TC_BUILDMAXWAIT	Maximum amount of time to wait (in seconds) between queries for new jobs. Default setting is 15, maximum setting is 300.			Build server

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_BUILDOPTS	Specifies build options for sending build log file messages to the screen, and setting the logging level. If you do not specify any of these options, then the build server writes build messages to the build log file (teamcbld.log), and writes a minimum level of messages to the log file. Some possible values are: • TOSCREEN (-s) sends the **teamcbld.log** file to the screen in addition to sending it to a file. • USEENVFILE (-n) – writes the changed environment variables to a file called **tcbldenv.lst** instead of setting them in program's environment. The format of the file is `variable=value`. – writes the list of input files to a file called **tcbldin.lst**. One file per line, format is `pathName type`. – writes the list of output files to a file called **tcbldout.lst**. One file per line, format is `pathName type`.	-s, -n		Build server
TC_BUILDPOOL	Specifies the build pool name.	-p	Pool	Build server
TC_BUILD_RSSBUILDS_FILE	Specifies the name of startup files to be used to provide information about build servers to the build process.			Build server
TC_CASESENSE	Changes the case of the arguments in commands, not in queries.		Case	Client

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_CATALOG	Specifies a specific file for the TeamConnectionmessage catalog. Sometimes, depending upon the operating system environment, the catalog open command will only look in a particular directory for the catalog. If the host is running multiple versions of TeamConnection, this variable may be used. To set this environment variable, specify the file path name of the message catalog as in the following example: `TC_CATALOG= "/`*`family`*`/msgcat/teamc.cat"`			Family server, oe build server
TC_COMPONENT	Specifies the default component.	-component	Component	Client, make import tool
TC_DBPATH	Specifies the database directory path. Family specific database files reside here.			Family server
TC_FAMILY	Identifies the TeamConnection family you work with.	-family	Family	Build server, client, family server, make import tool
TC_MAKEIMPORTRULES	Specifies the name of the rules file that TeamConnection uses when importing the makefile data into TeamConnection. If you set this environment variable, then you do not have to use the /u option with the fhomigmk command (Intel only). Specify the full path name of the rules file. If neither this environment variable nor the /u option is used, TeamConnection uses default rules.			Make import tool
TC_MAKEIMPORTTOP	Strips off the leading part of the directory name when importing parts into TeamConnection. For example, you have parts with the following directory structure: g:\octo\src\inc\. To create these parts without the g:\octo structure, you can set TC_MAKEIMPORTTOP=g:\octo before you invoke the make import tool. The parts created in TeamConnection have the directory structure of src\inc\.			Make import tool

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_MAKEIMPORTVERBOSE	Causes the -verbose flag to be added to part commands created by fhomigmk.			Make import tool
TC_MIGRATERULES	Specifies the name of a file containing the rules to be applied for migration of makefiles if the name is not supplied on the fhomigmk command (Intel only) line as a parameter.			Client
TC_MODELS	Specifies which models to load beyond the base TeamConnection models. Use thisd environment variable to enable tools that extend the TeamConnection model. The following list shows the values to set for TC_MODELS for other tools supported by TeamConection: **DataAtlas** TC_MODELS=″_ewswsdd _ewswhll _ewswims″ This environment variable is also used to specify which models to load in the TeamConnection Breditor (a tool offered by the ToolBuilder's Development Kit). The following are appropriate Breditor settings for each TeamConnection platform: **OS/2** fhcbred **Windows NT** fhmbred **AIX** fhcbred **Solaris** fhcbred **HP-UX** fhcbred			Sever
TC_MODPERM	Controls whether or not the read-only attribute is set after a part is created, checked in or unlocked in TeamConnection. To cause the read-only attribute to be set, specify TC_MODPERM=ON. To prevent the read-only attribute from being set, specify TC_MODPERM=OFF. The default is TC_MODPERM=ON.			Client

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_NOTIFY_DAEMON	An alternate way of starting notifyd with the teamcd command. If you set this environment variable, then you do not have to use the -n option with the teamcd command. Specify the full path name of the mail exit to use with notifyd.			Family server
TC_RELEASE	Specifies a release.	-release	Release	Client, make import tool
TC_REPORT_CHECKACCESS	Enables report access checking. With this option, TeamConnection checks a component's access list before allowing a user to view a component, defect, feature, or part. Only users who are granted authority in an authority group that includes the CompView, DefectView, FeatureView, and PartView actions can view reports for the component.Set TC_REPORT_CHECKACCESS=1 to enable or TC_REPORT_CHECKACCESS=0 to disable this function.			Family server
TC_TOP	Specifies the source directory.	-top	Top	Client
TC_TRACE	Specifies the variable that lets the user designate which parts should be traced. You should modify this only when directed to do so by an IBM service person. Otherwise it is set to null. To trace all parts, specify TC_TRACE=*.			Client, family server, build server
TC_TRACEFILE	Specifies the output (part path and name) of the trace that the user designates using TC_TRACE. The default trace file name is tctrace. For the MVS build server, the default trace file is stdout.			Client, family server, build server
TC_TRACESIZE	Specifies the maximum size of the trace file in bytes. If the maximum is reached, wrapping occurs. The default is one million bytes.			Client, family server, build server

Table 2. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_USER	Specifies the user login ID for single-user environments OS/2 and Windows 95 (if not using the login facility). This environment variable is not used in multiuser environments AIX, HP-UX, Solaris, MVS, MVS/OE, and Windows NT. If a user is using the Windows 95 login facility, this environment variable is not used.		User ID	Client, build server
TC_WORKAREA	Specifies the default workarea name.	-workarea	Workarea	Client, make import tool
TC_WWWPATH	Specifies the path for the HTML helps and image files for Web client.			Client, family server
TC_WWWDISABLED	Disables the Web client.			Family server

The following environment variables are dynamically set by the `teamcbld` processor before the build script is invoked:

Table 3. TeamConnection dynamically set build environment variables

Environment variable	Purpose	Flag	Setting	Used by
TC_BUILD_USER	Login of user who initiated the `part -build` command.			Build server
TC_INPUT	List of input files (separated by spaces).			Build server
TC_INPUTTYPE	List of input file types (such as `TCPart`).			Build server
TC_OUTPUT	List of output files.			Build server
TC_OUTPUTTYPE	List of output file types.			Build server
TC_LOCATION	Directory where build script is invoked.			Build server (except MVS build server)

Setting environment variables

For methods of setting your environment variables, refer to your operating system documentation. For example, you can use the following command to set the TC_FAMILY environment variable:

- OS/2 - `SET TC_FAMILY=familyName@hostname@portnumber`
- UNIX - `export TC_FAMILY=familyName@hostName@portNumber`

Appendix B. Importing makefile information into TeamConnection

TeamConnection provides a command to help you import makefile information into the TeamConnection database. The fhomigmk command (Intel only) reads a makefile and creates the parts in it. Build tree connections are created based on a rules file. The command syntax of the fhomigmk command is:

```
fhomigmk /m [makefile]
         /f [family]
         /r [release]
         /w [workarea]
         /c [command file]
         /u [rules file]
         /x
         /s
         /k
```

You can precede the parameter with either a slash (/) or a dash (-).

The parameters are defined as follows:

/m [*makefile*]
: The name of the makefile you want to import into TeamConnection. If you do not specify this parameter, TeamConnection uses `makefile`.

/f [*family*]
: The name of the TeamConnection family into which the makefile data will be imported. If not specified, TeamConnection uses the value of the TC_FAMILY environment variable. If the value of TC_FAMILY is not defined, the value `none` is used.

/r [*release*]
: The name of the TeamConnection release into which the makefile data will be imported. If not specified, TeamConnection uses the value of the TC_RELEASE environment variable. If the value of TC_RELEASE is not defined, the value `none` is used.

/w [*workarea*]
: The name of the TeamConnection workarea into which the makefile data will be imported. If not specified, TeamConnection uses the value of the TC_WORKAREA environment variable. If the value of TC_WORKAREA is not defined, the value `none` is used.

/c [*command file*]
: The name of the command file that will be produced and saved. If this file already exists, commands created by the specified makefile are appended to the existing contents.

/u [*rules file*]
: The name of the rules file that TeamConnection will use when importing the makefile data into TeamConnection. If not specified, TeamConnection uses the value of the TC_MAKEIMPORTRULES environment variable. If no rules file is found, TeamConnection uses default rules. "Creating a rules file" on page 210 explains the rules, the format of this file, and the default rules.

/x
: Specifies that you want to run the command file that was produced by the `/c` parameter.

/s Specifies that the build tree is to be displayed after the command is issued. If specified, the command file is run even if the `/x` parameter is not specified.

/k Specifies that you want TeamConnection not to erase the intermediate files it uses to process this command. This might be useful in debugging problems that arise during the import. However, in general, you will not specify this parameter. When specified, the following intermediate files are saved:

modified makefile
A modified form of the imported makefile. The command invocations (of things like linkers and compilers) are replaced by calls to a TeamConnection command that captures dependency data. To find the cause of import errors, type the following command at an OS/2 command line:

```
nmake -f mod_make
```

create file
A list of all the objects referenced by the makefile that should be created in the TeamConnection database.

connect file
A list of all the objects referenced by the makefile that should be connected to other objects in the TeamConnection database. Each line contains one dependency relationship in the format <child> <parent>.

TeamConnection provides an environment variable, TC_MAKEIMPORTTOP, that when set strips off the leading part of the directory name. For example, you have parts with the following directory structure: g:\octo\src\inc\. Because you want the parts created without the g:\octo structure, you set TC_MAKEIMPORTTOP=g:\octo before you invoke the make import tool. The parts created in TeamConnection have the directory structure of src\inc\.

Another environment variable provided by TeamConnection, TC_MAKEIMPORTVERBOSE, when set causes the -verbose flag to be added to part commands.

The following is an example of invoking the make import tool:

```
fhomigmk /m Mymak /w mywork /s /u myrules
```

In this example, the makefile called `mymak` is used to create a temporary command file containing TeamConnection commands. The commands are formed based on the rules defined in the file `myrules`. The family and release used in the commands are those specified in the environment variables TC_FAMILY and TC_RELEASE. The workarea used in the commands is `mywork`. After the commands are issued, the resulting build tree is shown using the TeamConnection GUI.

Creating a rules file

The import rules file is a text file that describes how you want TeamConnection to create and connect parts. In this file you supply a set of rules, one per line, using the following syntax:

file mask
: The mask specifying the names of the files to which this rule applies. The * and ? wildcards are supported. For example, you could specify file names such as *.cbl, abc*.cpp, or foo\src\*.obj.

type
: The type of contents of the files to which the rule applies when they are stored in TeamConnection as a part. Allowed values are binary, text, none, or ignore. If you specify ignore as the file type, then all files that match the file mask are bypassed.

builder
: The name of the TeamConnection builder to be associated with the part. The builder is not created for you. If you specify a builder, it must exist in TeamConnection before you run fhomigmk. A value of `none` means that no builder will be associated with the part.

parser
: The name of the TeamConnection parser to be associated with the part. The parser is not created for you. If you specify a parser, it must exist in TeamConnection before you run fhomigmk. A value of `none` means that no parser will be associated with the part.

connect
: How the part will be connected to other parts in TeamConnection. The following values are allowed:

- input
- output
- dependent
- none

When `none` is specified, the part is not connected to another part even though a dependency was found for the part in the make file. For example, when you indicate `none` for a file mask of *.h files, the *.h files are created in TeamConnection, but not connected to the files that include them. The value you will use most often is `input`.

content
: Where the initial content of the part can be found:

- `none` indicates that the part is initially created as *empty*.
- `directory\` indicates to concatenate with the name of the file in the makefile. This is where the contents are expected to be found.
- * indicates to use the name in the makefile, relative to the current working directory.

For example, if a makefile specifies a file src\abc.cbl and the makefile specifies f:\mysrc\, the content is expected in f:\mysrc\src\abc.cbl. For a file of *.cbl, the content is expected in src\abc.cbl relative to the current working directory.

parameters
: The build parameters to be attached to the part. Enclose the parameter in double quotes if it has spaces. Use the value `none` to indicate no parameters.

component
: The TeamConnection component that will contain the part. If none is specified, the value of the TC_COMPONENT environment variable is used.

As TeamConnection processes each part referenced in the makefile, it looks for a rule that matches the part name. If a match is found, the rule is used. The rules are searched from top to bottom. The first matching rule is used.

Comments are denoted by a pound sign (#) in the first column.

Columns are separated by spaces.

A sample rules file, called fhomigmk.rul, is supplied with TeamConnection. Use this file to help you create a rules file that is appropriate for your development environment.

The following is a simple example of an import rules file:

```
<top of file>
#-------------------------------------------------------------------------------
# file mask  type    builder  parser  connect  content  parameters  component
#-------------------------------------------------------------------------------
   *.exe     binary  linker   none    input    none     /Debug      ship
   *.obj     binary  icc      none    input    none     /Ti+        objects
   *.cpp     text    none     cplus   input    *        none        source
   *.h*      text    none     cplus   none     *        none        source
<end of file>
```

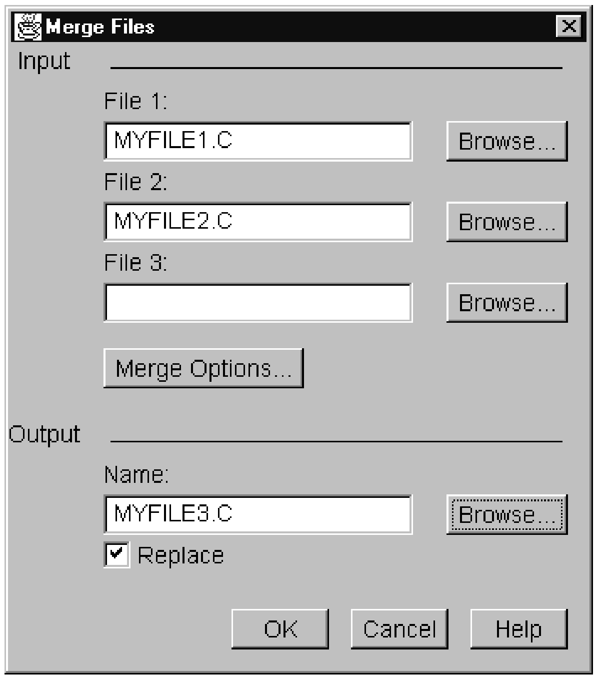
If you do not specify a rules file in the /u parameter of the fhomigmk command (Intel only), TeamConnection uses the value of the TC_MAKEIMPORTRULES environment variable. If no rules file is found, TeamConnection uses the following default rules:

```
<top of file>
#-------------------------------------------------------------------------------
# file mask  type    builder  parser  connect  content  parameters  component
#-------------------------------------------------------------------------------
   *.*       text    none     none    none     none     none        root
```

Appendix C. TeamConnection Merge

The TeamConnection VisualMerge tool enables you to view the differences between two or three selected files and to merge them together to make one single file.

You start the VisualMerge tool either by typing *tcmerge* at the command line or by selecting the TeamConnection Merge icon in the GUI. When you start the tool, the Merge Files window appears. From this window, you enter the names of the files you want to merge and the name of the file that you want to contain the merged output:



File 1: MYFILE1.C

File 2: MYFILE2.C

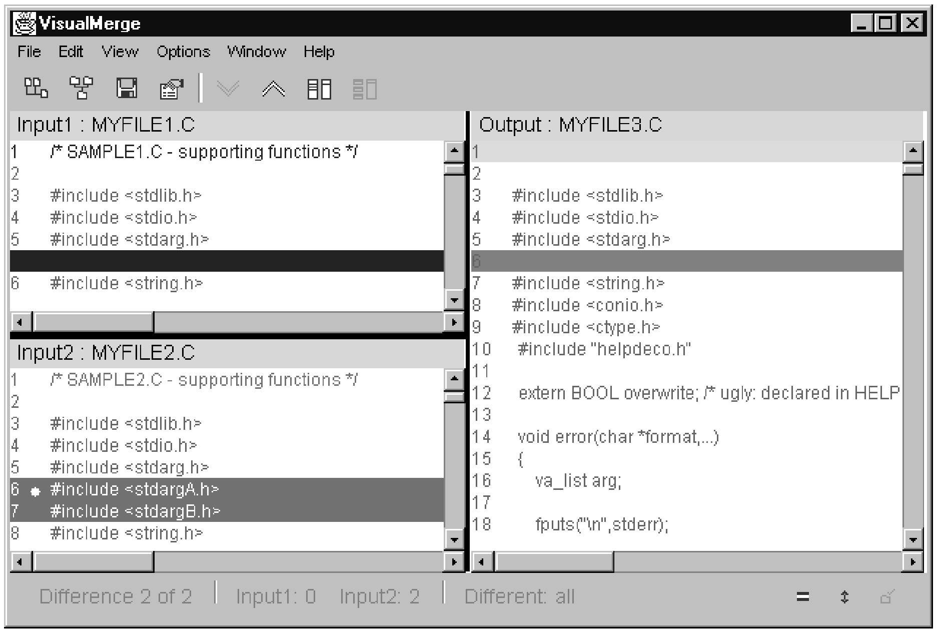
File 3:

Name: MYFILE3.C

[✓] Replace

Figure 72. Merge Files window

When you press the the OK push button, the Visual Merge window appears, showing the differences between the files you specified as well as the output file into which you can merge data:



Input1 : MYFILE1.C

```
/* SAMPLE1.C - supporting functions */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

#include <string.h>
```

Input2 : MYFILE2.C

```
/* SAMPLE2.C - supporting functions */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdargA.h>
#include <stdargB.h>
#include <string.h>
```

Output : MYFILE3.C

```


#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

#include <string.h>
#include <conio.h>
#include <ctype.h>
 #include "helpdeco.h"

 extern BOOL overwrite; /* ugly: declared in HELP

 void error(char *format,...)
 {
    va_list arg;

    fputs("\n",stderr);
```

Difference 2 of 2 | Input1: 0 Input2: 2 | Different: all

Figure 73. Visual Merge window

For more detailed instructions on using the Visual Merge tool, select **Help** from the Visual Merge menu bar.

You can also use the command line to enter information into the Visual Merge Tool. The following is the command line syntax for the VisualMerge tool with the parameter abbreviations shown. The first set of brackets encloses the abbreviation that may be used; the remainder shows the full parameter name. For example, `[-re[place]]` indicates that **`-re`** is the abbreviation for the **replace** parameter.

```
tcmerge <file1|directory1> <file2|directory2> [<file3|directory3>]
        [-ti[tles] <titlenames>] [-out <file|directory>]
        [-prime[out] <file|directory>] [-re[place]]
        [-ignoreco[lumns] <list of ranges to ignore>]
        [-ignoreb[lanks] <l|t|a|lt>] [-ignoreca[se]
        [-ignoreu[nique] <file|directory>]
        [-ignoreb[lanklines]] [-auto[merge]] [-nologo]
```

If you do not specify input parameters at the command prompt, the Merge Files window appears in which you enter information. If you only provide input and output files or directories, the following command defaults will be in effect:
- The output file or directory is not primed with one of the input files or directories.
- No columns will be ignored in difference calculation.
- No blanks will be ignored in difference calculation.

The following lists the parameters which may be used with the `tcmerge` command.

Parameter	**Description**
`<file1|directory1>`	First file or directory to merge. A combination of filenames and directory names is not valid.
`<file2|directory2>`	Second file or directory to merge.
`[<file3|directory3>]`	Third file or directory to merge. (Optional)
`[-ti[tles] <titlenames>]`	Names of files or directories being merged.
`[-out <file|directory>]`	The file or directory into which the merged differences will be written. This flag is optional.

Parameter	Description
`[-prime[out] <file|directory>]`	Primes the specified input file or directory to the output window.
`[-re[place]]`	The filename with the -out flag will be replaced. If not specified, and the filename specified with the -out flag exists, the -replace flag must be specified.
`[-ignoreco[lumns] <list of ranges to ignore>]`	List of column ranges to ignore during the compare. Entries must use the format <startColumn,endColumn> with no blanks within an entry. Specify one column <Column> where start and end would be the same number.
`[-ignoreb[lanks] <l|t|a|lt>]`	Specifies blanks to ignore. Specify one: **l** (leading), **t** (trailing), **lt** (leading trailing), or **a** (all blanks). Specify: **l** for blanks at the beginning of a line; **t** for blanks at the end of a line; **lt** for blanks at the beginning and end of the line; **a** for all blanks on a line.
`[-ignoreca[se]]`	Case of the characters is ignored. For example, there is no difference when an uppercase ″C″ and a lowercase ″c″ are compared.
`[-ignoreb[lanklines]]`	Any blank lines are ignored during the compare.
`[-ignoreu[nique] <file|directory>]`	An input file or directory is ignored during the compare.
`[-auto[merge]]`	Automatically merge differences between files or directories. When conflicts found, a GUI is invoked for you to resolve the conflict. Use this with the **-out** parameter to keep input source files unchanged. **Note:** Using this parameter is equivalent to using the TeamConnection `automrg` command. See the Commands Reference for additional information.
`[-nologo]`	Hides the TeamConnection logo.

Appendix D. Enabling an OS/2 Workframe project for TeamConnection

TeamConnection lets you create a Workframe version 3 project that has TeamConnection options as well as a set of TeamConnection actions. For each project, you specify on the Project Options window the values for these options. By doing this, you logically connect a Workframe project with a set of TeamConnection parts. This makes it easier for you to perform TeamConnection actions, such as checking parts in and out, directly from the Workframe.

Creating a TeamConnection-enabled Workframe project

Follow these steps to create a Workframe project that is enabled for TeamConnection.

1. On an OS/2 command line, type the following command and press Enter:

   ```
   fhotcini.cmd
   ```

   This command creates a TeamConnection **Project Smarts** catalog on your desktop. (If you have already created this catalog, there is no need to perform this step again for additional projects.)
2. Open the TeamConnection **Project Smarts** catalog. Select the TeamConnection project, and select the **Create** push button.
3. Specify the location for the TeamConnection project you want to create; then select **OK**.

   When the action completes, you will see a TeamConnection Project on your desktop.

Setting up your project options

Options are provided so that you can set up each TeamConnection Workframe project. To set the options, do the following:

1. Select **Tools Setup** from the project's Views pull-down menu.
2. Select the **Project Options** or **File Options** menu from any of the TeamConnection actions.

The following options are provided:

Family
: The TeamConnection family.

Release
: The TeamConnection release.

Workarea
: The TeamConnection workarea in which you will perform TeamConnection actions.

Query mask
: Any valid TeamConnection `-where` clause for parts. Leave blank to see all parts. (This is used in the project's **Show Parts** action.)

Show filter
: Check this if you want to display the PartFull Filter window instead of using the query mask in the **Show Parts** action.

Profile
Names the rules file to use for the **Import Make** action. Specify the fully qualified name unless you are sure it will be found in your path correctly. Select the **Find** push button if you need help.

Using your TeamConnection Workframe project

You can perform a set of TeamConnection actions from within your project:

- “Project actions” lists the actions you can perform without selecting a part.
- “Part actions” on page 219 lists the actions you can perform against a selected TeamConnection part.

Project actions

OS/2 has three project actions. They are:

1. Invoke the TeamConnection GUI
2. Show all parts from current context
3. Build All (build project target)

Extract part
Displays an unprimed Extract Parts window.

Checkout part
Displays an unprimed Check Out Parts window.

Checkin part
Displays an unprimed Check In Parts window.

Unlock part
Displays an unprimed Unlock Parts window.

Lock part
Displays an unprimed Lock Parts window.

Create part
Displays an unprimed Create Parts window.

Build part
Displays an unprimed Build Parts window.

View part contents
Displays an unprimed View Part Contents window.

View part information
Displays an unprimed View Part Information window.

Edit part
Displays an unprimed Edit Part window.

Show parts
If the project attribute **Show filter** is not set, issues a query based on the project attribute’s query mask. If the project attribute **Show filter** is set, displays the PartFull Filter window.

Part actions

Extract part
: Displays the TeamConnection Extract Parts window to extract the selected part.

Checkout part
: Displays the TeamConnection Check Out Parts window to check out the selected part to the workarea specified in the project options.

Checkin part
: Displays the TeamConnection Check In Parts window to check in the selected part to the workarea specified in the project options.

Unlock part
: Displays the TeamConnection Unlock Parts window to unlock the selected part.

Lock part
: Displays the TeamConnection Lock Parts window to lock the selected part.

Create part
: Displays the unprimed TeamConnection Create Parts window.

Build part
: Displays the TeamConnection Build Parts window to start a build of the selected part.

View part contents
: Displays the TeamConnection View Part Contents window for the selected part.

View part information
: Displays the TeamConnection View Part Information window for the selected part.

Edit part
: Displays the TeamConnection Edit Part window for the selected part.

Import makefile
: Imports the information contained in the selected makefile into the TeamConnection database.

The **Import makefile** action is restricted to files with the extension .mak. The other actions in this list apply to files of all types.

Using your project: a simple scenario

Suppose you are working on a defect in the family FAMILY1, release REL1_1. You have created a TeamConnection workarea called SANDBOX to work in. You want to use the Workframe to access your TeamConnection parts. Here is what you might do.

1. Create a TeamConnection Workframe project called DefectABC.
2. Open the project. Select **Tools setup** from the View menu.
3. Select any of the actions. Press mouse button 2 to display the context menu; then select **Project options** or **File options** from the context menu. The result is a window in which you can specify the TeamConnection information about the project.

4. Specify the family FAMILY1, the release REL1_1, and the workarea SANDBOX. Check the **Show filter** check box. Select **OK**.
5. Specify the general Workframe attributes of the project using the project's Settings notebook. These attributes include information such as the location of the OS/2 files for the project. For example, in this scenario, you specify that you want this project to contain all files in the directory c:\defect_abc, which is initially empty.
6. Select **TeamConnection → Show files** from the Project context menu. The PartFull Filter window is displayed. Specify the filter criteria; then select **OK**. For example, specify that you want to see all the parts with extensions .cpp, .exe, and .hpp. The Parts window is displayed.
7. Select the parts client.exe, server, client.cpp, client.hpp, server.cpp, and server.hpp. Select **Extract** from the context menu.
8. On the Extract Parts window, type `c:\defect_abc` in the **Target directory** field and select **OK**. Now you can interact with these parts directly from the Workframe project.
9. In the Workframe project, run the ipmd.exe debugger until you determine the cause of the problem. Suppose you find the bug is in client.cpp.
10. Go back to the Parts window. Select **client.cpp** from the list of parts, and select **TeamConnection → Checkout part** from the context menu for the object. The part is checked out to the SANDBOX workarea.
11. Edit the file to fix the problem; then select **TeamConnection → Checkin part** to check the part back into SANDBOX, the TeamConnection workarea from which it was checked out.
12. Build the part by selecting **TeamConnection → Build part** on the context menu for the file client.exe.
13. When the build completes, extract the resulting executable by selecting **TeamConnection → Extract part** from the file's context menu.
14. Run the executable to verify that the problem has been fixed.

Appendix E. Enabling a Workframe/NT project for TeamConnection

TeamConnection lets you create a Workframe/NT project that has TeamConnection options as well as a set of TeamConnection actions.

Workframe/NT doesn't use the TeamConnection dialogues. Instead, WorkFrame/NT uses monitored commands to invoke TeamConnection.

You must perform the following steps to integrate TeamConnection with WorkFrame/NT:

1. Install VisualAge C++
2. Install the TeamConnection Client
3. Make a backup copy of the WorkFrame Configuration File which is named vacpp.iws. This file is located in the MAINPRJ directory where VisualAge C++ is installed.
4. Replace or merge the WorkFrame Configuration File with the TeamConnection version of this file.

   If you have made changes to the WorkFrame Configuration File named vacpp.iws, you can use the TeamConnection Merge tool to view the differences between the two files and selectively merge the text into a single file. You can run the TeamConnection merge tool from a Windows command line. The syntax is:

   ```
   TCMERGE FILE1 FILE2
   ```

   Note: It is very important that you make a backup copy of the WorkFrame Configuration File (vacpp.iws) before making any changes to the file. Any changes made to the configuration files will not be migrated to a future version. Errors in the configuration file can prevent WorkFrame/NT from operating correctly.
5. Reboot to activate the WorkFrame/NT changes.

Setting up your project options:

All TeamConnection commands are monitored from inside an editor session. Environment variables are used to specify valid parameters for the project actions.

To set or change environment variables within WorkFrame/NT, from the Project's View menu, select Settings → Environment Variables.

You can use the following environment variables:

TC_FAMILY
The TeamConnection family. Required for all commands.

TC_USER
The current TeamConnection user. Required for all commands.

TC_BECOME
The current TeamConnection user. Required for all commands.

TC_RELEASE
The current TeamConnection release. Required for all commands.

TC_COMPONENT
The current TeamConnection component. Required for all commands.

TC_WORKAREA
The name of the current TeamConnection workarea. NULL is the default if a work area is not specified.

TC_TOP
Root of the work directory. Required for all commands.

TC_MAKEIMPORTRULES
The path and file name of the import rules file. Required for all Import Makefile.

DPATH
Includes all work directories for the WorkFrame project. Required for Import Makefile only if the project has more than one work directory. DPATH may already be set in the environment.

Each time you change the TC command parameters, you must also change the appropriate environment variable before selecting that TC action.

If you are using multiple directories in the WorkFrame project, you must set the TC_TOP environment variable to specify the top directory structure for each directory that you use.

Using your TeamConnection WorkFrame project

You can perform a set of TeamConnection actions from within your project:

- “Project actions” lists the actions you can perform without selecting a part.
- “Part actions” lists the actions you can perform against a selected TeamConnection part.

Project actions

WorkFrame/NT has three project actions. They are:

1. Invoke the TeamConnection GUI
2. Show all parts from current context
3. Build All (build project target)

Part actions

The current release and workarea are specified with environment variables. See “Setting up your project options:” on page 221.

TC Extract Part
Extract the specified part.

TC View Part Information
View information about the specified part.

TC Build Part
Build the specified output part.

TC Create Part
Create the specified part in TeamConnection.

TC Unlock Part
Unlock the specified part.

TC Import Makefile
Imports the information contained in the selected makefile into the TeamConnection database.

TC Checkin Part
Check in the specified part.

TC View Part Contents
Displays the contents of the specified part.

TC Lock Part
Locks the specified part.

TC Checkout Part
Checks out the specified part.

WorkFrame/NT uses selective part options:

- You can only checkin, checkout, or view contents of a non-binary part.
- Makefiles can only be imported.
- For WorkFrame/NT all TeamConnection actions are prefaced with ″TC″. For example, TC Checkout Part.

Appendix F. Source Code Control User's Guide

Differences between other source code control providers and TeamConnection

The purpose of this document is to help Visual Basic, Visual C++, and PowerBuilder users, make TeamConnection their Visual environments source code control provider. This document assumes the reader is a new user of TeamConnection, but has some familiarity with source code control.

Note: This component is for a client operating in a Windows 95 or NT environment. However, it may accept files from other environments providing the Win platforms can access them.

The following shows the version level supported by TeamConnection:

- PowerBuilder 5.0.04 or higher
- Visual Basic version 5.0 with SP2 applied or higer
- Visual C++ version 5.0 or higher
- IBM Visual Age Java 2.0 or higher

For the latest information on integrating third party development tools with TeamConnection, visit the following website:
http://www.software.ibm.com/ad/teamcon/

Projects vs Families

Most source code control providers group all code into projects. TeamConnection uses an object oriented approach that provides much more control over the software product while allowing greater flexibility. Projects have one dimension of control. Development environments like Visual Basic group all of their files into projects. Using projects to group source code has several limitations. First, the source code control system is limited to providing just version control. While version control is useful, once the enterprise-size organization is reached, it is often not sufficient to control just versions of the source code. TeamConnection provides not only versioning but defect and feature tracking, build and driver management, access control, and much more. TeamConnection uses families, releases, components, and workareas for management and control.

TeamConnection uses several layers of control. The highest level is the family. The family is the name of the data base, where TeamConnection stores all of the code, the versions, and all other information related to the code. A family represents a complete and self-contained collection of TeamConnection users and development data. Data within a family is completely isolated from data in all other families. One family cannot share data with another. It is important to know the name of the family where TeamConnection will store your code and associated information.

A part in TeamConnection is a collection of data that is stored by the family server. This can include files, text, objects, binary objects, or modeled objects. Parts can be stored by a user, a tool, or generated from other parts, such as when a linker generates an executable file.

Components are used to organize the data in a family. Components are arranged in a hierarchical tree structure, with a single top component called root. The component owns the parts that may be in it, and controls access to the parts. Once you are given access to a component, you have access to all the parts and subcomponents in that component. The component also controls the process that TeamConnection uses, for example, to report and fix a defect. Within each family, development data is organized into groups called components. The component hierarchy of each family includes a single top component, initially called root, and descendants of that root. Each child component has at least one parent component; a child can have multiple parents.

The release is somewhat analogous to a project. A release is a logical grouping of the components that make up a product. An application is likely to contain parts from more than one component. Because you probably want to use some of the same parts in more than one application, or in more than one version of an application, TeamConnection groups parts into releases. A release is a logical organization of all parts that are related to an application; that is, all parts that must be built, tested, and distributed together. Each time a release is changed, a new version of the release is created. Each version of the release points to the correct version of each part in the release. Each part in TeamConnection is managed by at least one component and contained in at least one release. One release can contain parts from many components; a component can span several releases. Each time a new development cycle begins, you can define a separate release. Each subsequent release of an application can share many of the same parts as its predecessor. You need to know the name of the release.

A workarea is basically a view of a release. For example, a workarea can be opened for each defect that needs to be fixed. More than one programmer can work in the same workarea at the same time. A programmer can have more than one workarea active at a time. A release contains the latest integrated version of each of its parts. As users check parts out of the releases, update them, and then check them back in, TeamConnection keeps track of all these changes, even when more than one user updates the same part at the same time.

You need to know the name of the workarea in which you will be working. A good practice is to create and name a workarea after the defect being addressed in the workarea. For example, name workarea W1557 for defect 1557. You can create a workarea if you have the authority in TeamConnection, but this must be done through the TeamConnection GUI.

For more information about families, releases, components, workareas, parts, and what you can do with them, see your TeamConnection Documentation.

Installing the TeamConnection source code control DLL

Before you can use the integrated support from your development environment you must install TeamConnection and the TeamConnection Source Code Control DLL. If you are using TeamConnection Version 2.0.8 or later, the source code control DLL is already installed.

Note: If you have not already done so, follow the directions and install the TeamConnection client for your workstation. The following directions assume that you have successfully installed the TeamConnection client.

Connecting TeamConnection to an IDE

In order for TeamConnection to be your source code control provider, both of the following Registry structure conditions must be in place:

- **HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider** contains the key name **ProviderRegKey**. If TeamConnection is your source code control provider, the value (Data) for this key name should be **SOFTWARE\IBM\VisualAge TeamConnection**. The key name may currently point to another provider if you have another provider installed.
- **HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider\InstalledSCCProviders** contains the key names for the available (installed) source code providers. The key name for TeamConnection is IBM VisualAge TeamConnection Enterprise Server and the data is **SOFTWARE\IBM\VisualAge TeamConnection** when the TeamConnection client is installed.

  Note: There may be other source code control providers currently designated (e.g., **Microsoft Visual SourceSafe** with data **SOFTWARE\Microsoft\SourceSafe**), depending on which other SCC providers have been installed.

Removing the TeamConnection Source Code Control DLL

To change the default source code control system for Visual Basic, Visual C++, or PowerBuilder, change the value in the ProviderRegKey to the registry key of another provider.

To remove TeamConnection, leave the value in the ProviderRegKey blank.

Using TeamConnection as your source code control provider

Once the installation procedure is complete, starting your development environment automatically links the TeamConnection Source Code Control DLL.

For the latest information on integrating third party development tools with TeamConnection, visit the following website:
http://www.software.ibm.com/ad/teamcon/

Before you start

There are several things you must know before you can start using TeamConnection as your source code control provider. If you are not sure of this information, contact your administrator. Your family administrator can help you find the following information:

- Family, defined by the following attributes:
  - Name
  - TCP/IP Host
  - Port Address
- Component
- Release
- WorkArea

You also need to know the project name. The project name is used by your development tool to relate to the TeamConnection attributes of family, release, workarea, and component by the Source Code Control DLL.

Opening a project

One of the few differences you see when using TeamConnection as your source code control provider occurs when you open or save a project (depending on the IDE you are using). When you open or save the new project, the **TeamConnection Settings** window opens. At the top of this window is a field with your development project name. In addition to the project name field, there are fields for family attributes, workarea, release, and component. If this is a new project, these fields are blank. If this is not a new project, the fields contain the previous values. You can change these values only when this window is open. If at anytime you decide to change any of these values, you might need to first close the project and reopen it.

Once all the fields are filled in, select **OK**. The project will open. If you select **Cancel**, the source code control system might disconnect from the development environment until another project is opened.

Under some versions of Visual Basic, for example, projects automatically close and open after certain operations. This causes this TeamConnection Settings window to open at times when it may appear unnecessary. When this occurs, select **OK**. If you select **Cancel**, you might be in a state that requires shutting down and restarting Visual Basic to reconnect the source code control system.

Integrated features

Once you open a project you can use the integrated features of the development environment to access your files in TeamConnection. The development environment keeps track of the files that are known to TeamConnection, and the checkout status of each file. For example, the development environment keeps track of files checked out to other users.

The exact steps necessary to perform each of the following actions depend on the development environment being used. However, for a given IDE, the steps are the same regardless of the source code control provider. For example, if you check out a file in the Visual C++ development environment when it is connected to Visual SourceSafe, the steps you use are exactly the steps you use when Visual C++ is connected to TeamConnection.

Check-in

The steps to check-in a file vary by the development environment. In most cases pressing mouse button 2 when the mouse pointer is over a file icon of a file checked out to you, brings up a menu that includes the file check-in option. Selecting the file checkin option opens the **Check-In** window. Checking the **keep checked out** box on the **Check-In** window sets the keep locked flag, TeamConnection saves the file, but keeps it checked out to you. Selecting **OK** causes the TeamConnection part check-in function to execute and the file is checked in.

Note: Depending on development environment and software levels, some features that follow may be unavailable.

Check-out

Similar to check-in, the check-out action can be started by pressing mouse button 2 on the file icon of a file not already checked-out. Check-out calls the TeamConnection Part Check-out function.

Uncheck-out

A checked out file can be unchecked out. Again this action can often be started by right clicking the file icon of a file that is checked out. Uncheck-out calls the part unlock function in TeamConnection.

Get Version

Rather than check out a file, you can also get the latest version of the file. Get Version calls the TeamConnection Part Extract function.

Adding Files to source code control

Adding a file that is not already under source code control places the selected file into the source code control system. Add calls the TeamConnection part create function.

Properties

Selecting **Properties** from a menu opens the properties GUI. Information that TeamConnection needs to correctly check out and check in parts is provided here. For example, the workarea field changes each time an existing workarea is integrated and a new workarea is created.

Full features of TeamConnection

Most development environments allow you to evoke TeamConnection from the menus. In Visual Basic, TeamConnection appears as an option in the Add-Ins menu. In Visual C++, TeamConnection appears in the Source Code Control option of the Tools menu. PowerBuilder uses the PowerBuilder Library Icon to designate TeamConnection as the Source Code Control provider. From the TeamConnection GUI you can create new workareas (if you have the correct authority), retrieve previous versions of a part, open or process defects, and perform many other actions against parts.

Migrating project data bases

One key issue for programmers and project managers moving from another source code control system to TeamConnection is how to migrate the database of projects. The following describes one way to bring the current level of code for a small to medium sized project into TeamConnection.

Migrating an existing project: The following example illustrates the simplest way to migrate an existing Visual Basic source code control database into TeamConnection. Lets say we were using the ABC source code control system, and we are going to migrate our project, Baylout, to TeamConnection. The idea is to extract all the files in Baylout using the ABC source code control system, and then add them as parts in TeamConnection. Follow the steps below to perform this migration.

1. Make ABC the default source code provider. To do this, set the registry key **ProviderRegKey** to point to the registry entry for source code control provider

ABC. See “Installing the TeamConnection source code control DLL” on page 226 for more information on how to perform this step. Once you complete this step, ABC will be the Source Code Control provider when we open Visual Basic.

2. Start the Visual Basic development environment.
3. Open project Baylout.
4. Extract all the files to your system.
5. Exit the Visual Basic development environment.
6. Edit the registry key **ProviderRegKey** to be:

   ```
   SOFTWARE\IBM\TeamConnection
   ```

   See “Installing the TeamConnection source code control DLL” on page 226 for more information on how to perform this step. TeamConnection is now the default source code control provider and is attached when the development environment starts.
7. Restart the Visual Basic development environment.
8. Open project Baylout again. Save the Baylout project. The **TeamConnection Settings** window will display.
9. When the **TeamConnection Settings** window opens it will have Baylout listed as the project. Fill in the values for family attributes, release, component, and workarea, then select **OK**.
10. Add the files to **TeamConnection** following the steps in the Visual Basic development environment.
11. Repeat these steps until all of your projects are migrated to TeamConnection.

Starting a new project

Starting a new project in Microsoft Visual Basic, Visual C++, or PowerBuilder is essentially the same regardless of the source code provider. The only operational difference is that the **TeamConnection Source Code Control Settings** window opens at some point. When the **TeamConnection Source Code Control Settings** window opens, enter the family attributes, component, workarea, and release.

Starting Visual Basic: In order to use TeamConnection source code control with Visual Basic. a subset of Microsoft Source Safe DLLs are required. After you have Microsoft Source Safe installed, you must make the following changes:

1. Add or update the following lines in the vbaddin.ini file :

   ```
   vbscc=0
   SccAddIn.SourceCodeControlAddIn=1
   ```
2. When you start Visual Basic after making these changes, you will see another check box in the **Add-Ins->Add-In Manager** panel called **Source Code Control Add-In**. You should check the new box, and uncheck the **Source Code Control** checkbox.

These steps should enable you to use the TeamConnection source code control with Visual Basic

To create a new project in Visual Basic, do the following:

1. Start Visual Basic.
2. Create and save a new project.
3. Select **Add Project to TeamConnection**from the **TeamConnection** option in the **Add-Ins** menu. The **TeamConnection Settings** window will display. Fill in the family attributes, release, component, and workarea, then select **OK**.

4. The **Add Project to TeamConnection** window opens. Select the files you want to add. Type a comment in the comment field. (Visual Basic requires that a comment be entered.) Select **OK**.

For more information, visit the following website: http://www.software.ibm.com/ad/teamcon/

Starting Visual C++: To create a new project under Visual C++, do the following:

1. Start the Visual Developers Studio as normal.
2. From the **File** menu, select **New**. The **New Project Workspace** window will open.
3. On the **New Project Workspace** window, do the following:
   a. Select the type of project
   b. Type a name
   c. Select **Create**.
4. Select **Project->Source Control->Add to Source Control** to open the **TeamConnection Settings** window.
5. On the **TeamConnection Settings** window, enter the family attributes, release, component, and workarea. Then, select **OK**.
6. Files can now be added to the project using the **Insert** menu.
7. To place the files under source code control, select the **Add to Source Code Control** option of the **Tools** menu.

For more information, visit the following website: http://www.software.ibm.com/ad/teamcon/

Starting PowerBuilder: To create a new project under PowerBuilder, do the following:

1. Start PowerBuilder.
2. Select the **Library** icon.

   Note: If TeamConnection Source Code Control is already installed (the **TeamConnection Settings** window will open), you can proceed with Step 5
3. From the **Source** menu, select **Connect**. The **Connect** dialog box will display.
4. Select SCC API from the **Connect** list box, and then select **OK**.
5. The **TeamConnection Settings** window will open. On the **TeamConnection Settings** window, enter the family attributes, release, component, and workarea. Then select **OK**.

   Note: The **TeamConnection Settings** window may redisplay. If this happens, select **OK** again.
6. Add parts to the TeamConnection. Select all parts that you want to add to TeamConnection, including the project folder in the **Library** window. You can select all of these objects simultaneously by dragging the mouse pointer while depressing the Control (**CTRL**) key.
7. From the **Source** menu, select **Register**. This may take some time. These registered parts should be added to the TeamConnection workarea previously specified in the **TeamConnection Settings** window

For more information, visit the following website: http://www.software.ibm.com/ad/teamcon/

Appendix G. Supported expandable keywords

TeamConnection supports expandable keywords in text files. When a file containing expandable keywords is extracted from TeamConnection, the current value of each keyword is added to the file. This information can help you identify what version of source code is used for your deliverables.

TeamConnection supports the following keywords.

Keyword	Description
$ChkD;	The time and date stamp applied during check in.
$FN;	The file name complete with its path.
$KW;	The start of keyword expansion. It expands to @(#).
$EKW;	Keyword expansion is ended until the next $KW; keyword.
$Own;	The user ID of the owner of the component that manages the part.
$Ver;	The version of the part in TeamConnection.

The following examples show lines of code that change in a text file as a user extracts a part. The text file used in this example is filex.hdr.

```
#ifndef_filex_hdr_
#define_filex_hdr_
  static char _filex_hdr[]="$KW; $FN; $Ver; $ChkD; $EKW;";
#endif
```

TeamConnection ignores keywords until it finds a $KW; keyword. It then expands all keywords until a $EKW; keyword is found. If the semicolon (;) following a keyword is omitted, the keyword is not expanded.

No change occurs when the part is checked in to TeamConnection. However, when the part is extracted, the keyword variables are updated. The following example shows how the keywords are expanded.

```
#ifndef_filex_hdr_
#define_filex_hdr_
  static char _filex_hdr[]="$KW=@(#) $FN=bin/filex.hdr; $Ver=1:1;
$ChkD=1998/03/20 18:13:19; $EKW;";
#endif
```

In the previous example, each keyword and its value appears in the output. The value of the keyword is replaced each time the part is extracted. If you do not want the keyword to appear in the output, add a minus sign (-) after the dollar sign ($). For example, if the statement is prepared as follows:

```
  static char _filex_hdr[]="$-KW; $-FN; $-Ver; $-ChkD; $-EKW;";
```

Then the expanded keywords will look like:

```
  static char _filex_hdr[]="$@(#) bin/filex.hdr 1:1
1998/03/20 18:13:19 $-EKW;";
```

Be aware that if a file is extracted, then locked and checked in, the version information can no longer be updated because the keyword does not appear in the output.

Note for AIX users

There is a problem with the AIX compilers in which static variables are stripped from the executable code during the optimization phase. To overcome this problem, you must use the static variables that use expanded keywords from VisualAge

TeamConnection for these variables to be left intact in the executable code. This will enable the Unix ″what″ command to find them.

Using the example above, after the variable *_filex_hdr* is defined, add the following source code: (The new code will never be executed, but this forces the optimizer to keep the defined variable inside the executable code:

```
int dummy1 = 1;  /* This will force the if statement to be always false */

/* You need to define the same number of many dummy variables for all the defined
    static variables */

char * dummy2;

dummy2 = _filex_hdrd;

 /* The following is needed in AIX in order to force the optimizer to keep the variables
    that have expandable keywords. This code is never executed. */

   if (dummy1 == -7077)
   {
       printf("%s\n", dummy2);
   }
```

Appendix H. Authority and notification for TeamConnection actions

TeamConnection ships with IBM-supplied authority groups, interest groups, component processes, and release processes. Your family administrator can modify these preconfigured authority groups, interest groups, and processes to fit the needs of your organization.

Each authority group consists of actions normally performed by a particular type of user. Your family administrator can modify these groups or create new ones to reflect the needs of your organization.

Authority groups provide explicit authority to perform the actions included in each group. You might also have implicit authority to perform certain actions according to the objects that you own. Authority groups are defined in a file called authorit.ld.

To determine your authority groups, from the Actions pull-down menu, select Lists → Access lists → Show authority actions. On the Show authority actions window select an action.

Each notification group consists of actions normally of interest to a particular type of user. Your family administrator can modify these groups or create new ones to reflect the needs of your organization. Interest groups are defined in a file called interest.ld.

To determine your interest notification groups, from the Actions pull-down menu, select Lists → Notification lists → Show interest actions. On the Show authority actions window select an action.

Notification for TeamConnection actions can be implicit or explicit. For example, the owner of an object receives implicit notification if an action is performed on an object, while users on a notification list receive explicit notification. (See the Administrator's Guide for more information.)

The following table lists all of the TeamConnection actions, the required level of implicit and explicit authority to perform the action, and the users who are notified when an action is performed. To explicitly assign authority to a user, add the user's ID to a component's access list.

Note: The user who performs the action is excluded from the notification that is sent out after the action is successfully completed.

For this action	These users have authority	These users are notified
AccessCreate	• Component owner • Explicitly defined for the component where access is being added	User being given new access, subscribers
AccessDelete	• Component owner • Explicitly defined for the component where access is being altered	User whose access was deleted, subscribers
AccessRestrict	• Component owner • Explicitly defined for the component where access is being restricted	User whose access is being restricted, subscribers

For this action	These users have authority	These users are notified
ApprovalAbstain	• Approval record owner • Explicitly defined for the component that manages the associated release	Approval record owner, subscribers
ApprovalAccept	• Approval record owner that manages the associated release	Approval record owner, subscribers
ApprovalAssign	• Approval record owner • Explicitly defined for the component that manages the associated release	New and original approval record owners, subscribers
ApprovalCreate	• Workarea owner • Explicitly defined for the component that manages the associated release	New approval record owner, subscribers
ApprovalDelete	• Explicitly defined for the component that manages the associated release	Approval record owner, subscribers
ApprovalReject	• Approval record owner • Explicitly defined for the component that manages the associated release	Approval record owner, subscribers
ApproverCreate	• Release owner • Explicitly defined for the component that manages the associated release	New approver, subscribers
ApproverDelete	• Release owner • Explicitly defined for the component that manages the associated release	Deleted approver, subscribers
BuilderCreate	• Explicitly defined for the component that manages the associated release	Subscribers
BuilderDelete	• Explicitly defined for the component that manages the associated release	Subscribers
BuilderExtract	• Explicitly defined for the component that manages the associated release	Not applicable
BuilderModify	• Explicitly defined for the component that manages the associated release	Subscribers
BuilderView	• Explicitly defined for the component that manages the associated release	Not applicable
CollisionAccept	• Component owner • Explicitly defined for the component that manages the associated release	Release owner, subscribers
CollisionReconc	• Component owner • Explicitly defined for the component that manages the associated release	Release owner, subscribers
CollisionReject	• Component owner • Explicitly defined for the component that manages the associated release	Release owner, subscribers
CompCreate	• Parent component owner • Explicitly defined for the parent component	New component owner

For this action	These users have authority	These users are notified
CompDelete	• Component owner • Explicitly defined for the component being removed	Component owner, subscribers
CompLink	• Component owner of the component being linked • Explicitly defined for the component being linked	Owners of both components, subscribers
CompModify	• Component owner • Explicitly defined for the component being modified	New component owner if applicable, subscribers
CompRecreate	• Parent component owner • Explicitly defined for the parent component	Owners of both components, subscribers
CompUnlink	• Component owner of the component being unlinked • Explicitly defined for the component being unlinked	Owners of both components, subscribers
CompView	• Component owner • Explicitly defined for the component being viewed	Not applicable
CoreqCreate	• Workarea owner of all specified workareas • Explicitly defined for the component managing the associated workarea and release	Not applicable
CoreqDelete	• Workarea owner of all specified workareas • Explicitly defined for the component associated with the release	Not applicable
DefectAccept	• Defect owner for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectAssign	• Defect owner, defect originator • Explicitly defined for the component associated with the defect **Note:** Originators who do not have DefectAssign authority can reassign the defect only when it is in the open state.	New owner, defect originator, duplicate defect originators, subscribers
DefectCancel	• Defect originator • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectClose	Automatic action; no authority is required	Defect owner, defect originator, duplicate defect originators, subscribers
DefectConfiginfo	Not applicable; this is a base authority that can be performed by all users in the family	Defect owner, defect originator, duplicate defect originators, subscribers
DefectDesign	• Defect owner • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers

For this action	These users have authority	These users are notified
DefectModify	• Defect owner can modify: – answer, abstract, environment, driver, prefix, reference, release, and all configurable fields • Defect originator can modify: – originator, severity, name, abstract, environment, driver, prefix, reference, release, and all configurable fields • Explicitly defined for the component associated with the defect, these users can modify: – abstract, answer, name, environment, driver, originator, prefix, reference, release, severity, phaseFound*, phaseInject*, priority*, symptom*, and target* *If these fields have been configured by the family administrator, the field names might differ from those shown.	Defect owner, defect originator, duplicate defect originators, subscribers
DefectOpen	Not applicable; this is a base authority that can be performed by all users in the family	Component owner, subscribers
DefectReopen	• Defect originator • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectReturn	• Defect owner • Explicitly defined for the component associated with the defect	Defect originator, duplicate defect originators, subscribers
DefectReview	• Defect owner • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectSize	• Defect owner • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectVerify	• Defect owner • Explicitly defined for the component associated with the defect	Defect owner, defect originator, duplicate defect originators, subscribers
DefectView	• Defect owner, defect originator • Explicitly defined for the component associated with the defect	Not applicable
DriverAssign	• Driver owner • Explicitly defined for the component associated with the release	New owner, subscribers
DriverCheck	• Driver owner • Explicitly defined for the component associated with the release	Not applicable
DriverCommit	• Explicitly defined for the component associated with the release	Subscribers
DriverComplete	• Explicitly defined for the component associated with the release	Subscribers

For this action	These users have authority	These users are notified
DriverCreate	• Release owner • Explicitly defined for the component associated with the release	Subscribers
DriverDelete	• Driver owner • Explicitly defined for the component associated with the release	Subscribers
DriverExtract	• Driver owner • Explicitly defined for the component associated with the release	Not applicable
DriverFreeze	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
DriverModify	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
DriverRefresh	• Explicitly defined for the component associated with the release	Component owner, subscribers
DriverRestrict	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
DriverView	• Driver owner • Explicitly defined for the component associated with the release	Not applicable
EnvCreate	• Release owner • Explicitly defined for the component associated with the release	Tester, subscribers
EnvDelete	• Release owner • Explicitly defined for the component associated with the release	Subscribers
EnvModify	• Release owner • Explicitly defined for the component associated with the release	Tester, subscribers
FeatureAccept	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureAssign	• Feature owner • Explicitly defined for the component associated with the feature	New owner, feature originator, duplicate feature originators, subscribers
FeatureCancel	• Feature originator • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureClose	Occurs automatically; no authority is required	Feature owner, feature originator, duplicate feature originators, subscribers

For this action	These users have authority	These users are notified
FeatureComment	Not applicable; this is a base authority that can be performed by all users in the family	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureDesign	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureModify	• Feature owner can modify: – abstract, prefix, reference, and all configurable fields • Feature originator can modify: – abstract, name, prefix, reference, and all configurable fields • Explicitly defined for the component associated with the feature, these users can modify: – abstract, name, originator, prefix, reference, priority*, and target* *If these fields have been configured by the family administrator, the field names might differ from those shown.	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureOpen	Not applicable; this is a base authority that can be performed by all users in the family	Component owner, subscribers
FeatureReopen	• Feature originator • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureReturn	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureReview	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureSize	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureVerify	• Feature owner • Explicitly defined for the component associated with the feature	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureView	• Feature owner • Explicitly defined for the component associated with the feature	Not applicable
FixActive	• Fix record owner, component owner, workarea owner • Explicitly defined for the component associated with the fix record	Subscribers

For this action	These users have authority	These users are notified
FixAssign	• Fix record owner, component owner, workarea owner • Explicitly defined for the component associated with the fix record	New fix record owner, subscribers
FixComplete	• Fix record owner, component owner, workarea owner • Explicitly defined for the component associated with the fix record	Subscribers
FixCreate	• Defect or feature owner, workarea owner • Explicitly defined for the component associated with the defect or feature	Subscribers
FixDelete	• Defect or feature owner, workarea owner • Explicitly defined for the component associated with the defect or feature	Subscribers
HostCreate	• Owner of the user ID for which a host list entry is being created or deleted • Superuser	Not applicable
HostDelete	• Owner of the user ID for which a host list entry is being deleted • Superuser	Not applicable
MemberCreate	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
MemberCreateR	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
MemberDelete	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
MemberDeleteR	• Driver owner • Explicitly defined for the component associated with the release	Driver owner, subscribers
NotifyCreate	• Component owner • Explicitly defined for the component associated with the notification list	Not applicable
NotifyDelete	• Component owner • Owner of user ID • Explicitly defined for the component associated with the notification list **Note:** Users can delete themselves from a notification list without requiring any authority	Not applicable
ParserCreate	• Explicitly defined for the component associated with the release	Subscribers
ParserDelete	• Explicitly defined for the component associated with the release	Subscribers

For this action	These users have authority	These users are notified
ParserModify	• Explicitly defined for the component associated with the release	Subscribers
ParserView	• Explicitly defined for the component associated with the release	Not applicable
PartAdd	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartBuild	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartCheckIn	• User who checked out or locked the part originally, component owner • Explicitly defined for the component associated with the part **Note:** The user who is explicitly given this authority can check in a part that is checked out by someone else.	Subscribers
PartCheckOut	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartChildInfo	• Component owner • Explicitly defined for the component associated with the part	Not applicable
PartConnect	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartDelete	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartDeleteForce	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartExtract	• Component owner • Explicitly defined for the component associated with the part	Not applicable
PartForceIn	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartForceOut	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartLink	• Component owner • Explicitly defined for the component associated with the part	Subscribers

For this action	These users have authority	These users are notified
PartLock	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartLockForce	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartMark	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartMerge	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartModify	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartOverrideR	Explicitly defined for the component associated with the release	Subscribers, user granted the override (if a user specified)
PartReconcile	• Component owner • Explicitly defined for the component that manages the associated release	Subscribers
PartRecreateForce	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartRecreate	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartRefresh	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartRename	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartRenameForce	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartResolve	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable
PartRestrict	Explicitly defined for the component associated with the release	Subscribers
PartTouch	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartUndo	• Component owner • Explicitly defined for the component associated with the part	Subscribers

For this action	These users have authority	These users are notified
PartUndoForce	• Component owner • Explicitly defined for the component associated with the part	Subscribers
PartUnlock	• User who checked out or locked the part originally, component owner • Explicitly defined for the component associated with the part	Subscribers
PartView	• Component owner • Explicitly defined for the component associated with the part	Not applicable
PartViewMsg	• Component owner • Explicitly defined for the component associated with the part	Not applicable
PrereqCreate	• Workarea owner of all specified workareas • Explicitly defined for the component managing the associated workarea and release	Not applicable
PrereqDelete	• Workarea owner of all specified workareas • Explicitly defined for the component managing the associated workarea and release	Not applicable
ReleaseCreate	• Explicitly defined for the component associated with the new release	New release owner, component owner, subscribers
ReleaseDelete	• Release owner • Explicitly defined for the component associated with the release	Release owner, component owner, subscribers
ReleaseExtract	• Release owner • Explicitly defined for the component associated with the release	Not applicable
ReleaseLink	• Release owner • Explicitly defined for the component associated with the release	Release owner, subscribers
ReleaseMerge	• Release owner • Explicitly defined for the component associated with the release	Release owner, subscribers
ReleaseModify	• Release owner • Explicitly defined for the component associated with the release **Note:** To identify a new component to manage the release, you must have ReleaseCreate in an authority group in the component that you are modifying	Release owner, subscribers, new owner (if applicable)
ReleasePrune	• Release owner • Explicitly defined for the component associated with the release	Subscribers

For this action	These users have authority	These users are notified
ReleaseRecreate	• Release owner • Explicitly defined for the component associated with the release	Release owner, component owner, subscribers
ReleaseView	• Release owner • Explicitly defined for the component associated with the release	Not applicable
Report	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable
ShadowCreate	Explicitly defined for the component associated with the release	Not applicable
ShadowDefine	Superuser	Not applicable
ShadowDelete	Explicitly defined for the component associated with the release	Not applicable
ShadowDisable	Explicitly defined for the component associated with the release	Not applicable
ShadowEnable	Explicitly defined for the component associated with the release	Not applicable
ShadowModify	Explicitly defined for the component associated with the release	Not applicable
ShadowRedefine	Superuser	Not applicable
ShadowSync	Explicitly defined for the component associated with the release	Not applicable
ShadowUndefine	Superuser	Not applicable
ShadowVerify	Explicitly defined for the component associated with the release	Not applicable
ShadowView	Explicitly defined for the component associated with the release	Not applicable
SizeAccept	• Sizing record owner • Explicitly defined for the component associated with the sizing record	Subscribers
SizeAssign	• Sizing record owner • Explicitly defined for the component associated with the sizing record	New sizing record owner, defect/feature owner, subscribers
SizeCreate	• Defect/feature owner • Explicitly defined for the component associated with the defect/feature	Component owner, defect/feature owner, subscribers
SizeDelete	• Defect/feature owner • Explicitly defined for the component associated with the defect/feature	Subscribers, sizing record owner, defect/feature owner
SizeReject	• Sizing record owner • Explicitly defined for the component associated with the sizing record	Subscribers
TestAbstain	• Test record owner • Explicitly defined for the component associated with the test record’s release	Subscribers

For this action	These users have authority	These users are notified
TestAccept	• Test record owner • Explicitly defined for the component associated with the test record's release	Subscribers
TestAssign	• Test record owner • Explicitly defined for the component associated with the test record's release	New test record owner, subscribers
TestReady	• Test record owner • Explicitly defined for the component associated with the test record's release	Subscribers
TestReject	• Test record owner • Explicitly defined for the component associated with the test record's release	Subscribers
UserCreate	Superuser	New user
UserDelete	Superuser	Not applicable
UserModify	• Owner of the user object can modify all characteristics except the superuser privilege • Must be a superuser to grant the superuser privilege	Not applicable
UserRecreate	Superuser	Not applicable
UserView	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable
VerifyAbstain	• Verification record owner • Explicitly defined for the component associated with the verification record's defect or feature	Subscribers
VerifyAccept	• Verification record owner • Explicitly defined for the component associated with the verification record's defect or feature	Subscribers
VerifyAssign	• Verification record owner • Explicitly defined for the component associated with the verification record's defect or feature	New verification record owner, subscribers
VerifyReady	Takes place automatically; no authority is required	Verification record owners
VerifyReject	• Verification record owner • Explicitly defined for the component associated with the verification record's defect or feature	Subscribers
WorkAreaAssign	• Workarea owner • Explicitly defined for the component associated with the release	New workarea owner, subscribers
WorkAreaCancel	• Defect or feature owner (if either exists), otherwise workarea owner • Explicitly defined for the component associated with the defect or feature	Subscribers, owners of approval records for workarea being canceled
WorkAreaCheck	• Workarea owner • Explicitly defined for the component associated with the release	Not applicable

For this action	These users have authority	These users are notified
WorkAreaCommit	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaComplet	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaCreate	• Defect or feature owner • Explicitly defined for the component associated with the defect or feature	Workarea owner, subscribers
WorkAreaFix	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaFreeze	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaIntegra	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaModify	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaReconcile	• Workarea owner • Explicitly defined for the component associated with the release	Workarea owner, subscribers
WorkAreaRefresh	• Workarea owner • Explicitly defined for the component associated with the release	Workarea owner, subscribers
WorkAreaTest	• Workarea owner • Explicitly defined for the component associated with the release	Subscribers
WorkAreaUndo	• Defect or feature owner (if either exists), otherwise workarea owner • Explicitly defined for the component associated with the release	
WorkAreaView	• Workarea owner • Explicitly defined for the component associated with the release	Not applicable

Appendix I. Sample REXX execs, build scripts, and parsers

This appendix is composed of the IBM-supplied REXX execs, build scripts, and parsers. Your family administrator can modify these samples to fit the needs of your organization.

The samples in this appendix may not be available on all platforms. Refer to the readme file for a complete list of samples available with TeamConnection. All samples are provided as-is and any use of or modifications to the samples are the sole responsibility of the customer.

Sample REXX execs

This section lists the sample REXX execs that are shipped with TeamConnection. The client.smp file contains this same listing. It is located in the bin subdirectory of the directory where the TeamConnection client is installed.

Users running these execs must have user and host access to your TeamConnectionfamily.

Most of the execs require input parameters, and some require that the `TC_FAMILY` or `TC_RELEASE` environment variables be set. If the user who is running the script is acting for another user, the `TC_BECOME` environment variable must also be set. These variables can be set from a command line prompt.

The following convention is used to show the required, optional, and selective input parameters:

- Brackets ([]) indicate that the input or variable is optional.
- Braces ({}) indicate that one of the inputs is required.
- An input or variable that is not surrounded by brackets or braces is required.

Script name	Function	Inputs	Environment variables
accComp	Lists the explicit access of users in a specified component.	componentName	TC_FAMILY [TC_BECOME]
compChld	Lists the direct children of a specified component. Also lists the description and owner of each child component.	componentName	TC_FAMILY [TC_BECOME]
compOwnr	Displays a list of component owners' addresses in a specified family.	familyName	[TC_BECOME]
compPrnt	Lists the parent component of a specified component.	componentName	TC_FAMILY [TC_BECOME]
compWalk	Displays the children and grandchildren of a specified component.	componentName	TC_FAMILY [TC_BECOME]
defClone	Creates a new defect based on values contained in a specified defect.	defectNumber	TC_FAMILY [TC_BECOME]
defDrvr	Lists all defects for a specified driver.	driverName	TC_FAMILY [TC_BECOME]
defFfea	Creates a new defect based on values contained in a specified feature.	featureNumber	TC_FAMILY [TC_BECOME]
defNew	Displays the number of the most recent defect that was entered in the system.		TC_FAMILY [TC_BECOME]
defReopn	Reopens a previously canceled or returned defect.	defectNumber	TC_FAMILY [TC_BECOME]

Script name	Function	Inputs	Environment variables
defRept	Generates a global defect report showing workareas, test records, approval records, and fix records.		TC_FAMILY [TC_BECOME]
defState	Lists the defect number of all defects that are in a specified state.	stateName	TC_FAMILY [TC_BECOME]
defStats	Displays total active defect statistics on a defect owner area basis.	ownerArea	TC_FAMILY [TC_BECOME]
defWRef	Displays the full details of defects that contain the specified reference field value.	reference	TC_FAMILY [TC_BECOME]
dfDesc	Displays the full remarks that were entered when the specified defect or feature was created.	{defectNumber | featureNumber}	TC_FAMILY [TC_BECOME]
feaClone	Creates a new feature based on values contained in a specified feature.	featureNumber	TC_FAMILY [TC_BECOME]
feaDrvr	Lists all features contained in a specified driver.	driverName	TC_FAMILY [TC_BECOME]
feaFdef	Creates a new feature based on the values contained in the specified defect.	defectNumber	TC_FAMILY [TC_BECOME]
feaNew	Displays the number of the most recent feature that was entered in the system.		TC_FAMILY [TC_BECOME]
feaReopn	Reopens a previously canceled or returned feature.	featureNumber	TC_FAMILY [TC_BECOME]
feaRept	Generates a global feature report showing workareas, test records, size records, and fix records.		TC_FAMILY [TC_BECOME]
feaState	Lists the feature number of all features that are in a specified state.	stateName	TC_FAMILY [TC_BECOME]
feaStats	Displays total active feature statistics on a feature owner area basis.	ownerArea	TC_FAMILY [TC_BECOME]
drvByDF	Lists the name of the drivers that contain a specified defect or feature.	{defectNumber | featureNumber}	TC_FAMILY [TC_BECOME]
drvrMem	Lists the defect and feature members of a specified driver for a specified release.	driverName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
mailTo	Sends a message to the addresses read through stdin.	messagefile subject	
ownerChg	Re-assigns all current work and objects owned by userLogin1 to userLogin2.	userLogin1 userLogin2	TC_FAMILY [TC_BECOME]
prtChckin	Checks parts into the TeamConnection family. When common parts are encountered, the script requests the releases for which the part should remain common.	partPathName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtChgDf	Lists all the parts that were changed for a specified defect or feature.	{defectNumber | featureNumber}	TC_FAMILY [TC_BECOME]
prtChgDr	Lists the parts that were changed for a specified driver.	driverName	TC_FAMILY [TC_BECOME]
prtComGt	Extracts all the parts associated with a specific component. The parts are placed in a directory that represents the release name to which the version of the part is associated. This directory is created relative to the relativePathName parameter.	componentName relativePathName [committed]	TC_FAMILY [TC_BECOME]
prtComp	Lists all parts related to a specified component.	componentName	TC_FAMILY [TC_BECOME]

Script name	Function	Inputs	Environment variables
prtHist	Lists all defect and feature numbers and abstracts that caused a change to a specified part in a specified release.	partName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtInfo	Displays information for a specified part.	partName	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtLock	Lists all parts that a specified user has locked.	userLogin	TC_FAMILY [TC_BECOME]
prtLokBy	Lists who has a specified part checked out.	partName	TC_FAMILY TC_RELEASE [TC_BECOME]
PrtPath	Finds and lists all parts that match a partial part path name.	partPathName	TC_FAMILY [TC_BECOME]
prtRel	Lists all parts related to a specified release.	releaseName	TC_FAMILY [TC_BECOME]
prtWaGt	Extracts all the parts associated with a specific workarea and places them in the path specified by the relativePathName parameter.	releaseName workareaName relativePathName	TC_FAMILY [TC_BECOME]
rByArea	Generates a manager's report based on the specified areas or departments of interest.	areaName ...	TC_FAMILY [TC_BECOME]
relOwner	Displays a list of addresses of all release owners in a specified TeamConnectionfamily.	familyName	[TC_BECOME]
showConf	Lists the valid values pertaining to a specified configurable type.	configType	TC_FAMILY [TC_BECOME]
userAuth	Lists the users who have the authority to give other users access to a specified component.	componentName	TC_FAMILY [TC_BECOME]
userInfo	Finds user information based on part of the user's name. A fuzzy search is performed.	userName	TC_FAMILY [TC_BECOME]
usersAll	Lists the addresses of all users in a specified TeamConnection family.	familyName	[TC_BECOME]
usrAcc	Lists the explicit access of a specified user for the specified component and its descendant components.	userLogin componentName	TC_FAMILY [TC_BECOME]
usrRept	Generates a user's report based on the specified user login.	userLogin	TC_FAMILY [TC_BECOME]
verByPrt	Lists the version numbers, release names, and path names for the specified part.	partName releaseName	TC_FAMILY [TC_BECOME]
waComit	Lists the workareas that are in the commit state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waFix	Lists all the workareas that are in the fix state for a given release.	releaseName	TC_FAMILY [TC_BECOME]
waInLvl	Lists the workareas that are in the integrate state and are associated with at least one development driver for the specified release.	releaseName	TC_FAMILY [TC_BECOME]
waInt	Lists the workareas that are in the integrate state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waPrdLv	Lists the workareas that are included in a production driver and are in the integrate state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waStat	Generates a workarea activity statistics report on a user area basis.	userArea	TC_FAMILY [TC_BECOME]
waTest	Lists the workareas that are in the test state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]

Sample build scripts

fhbcob2.cmd
Calls the COBOL Visual Set for OS/2 compiler.

fhbcob2l.cmd
Calls the COBOL Visual Set for OS/2 compiler and link editor.

fhbocomp.cmd
Calls the VisualAge for C++ icc compiler.

fhbolib.cmd
Calls the OS/2 implib utility.

fhbolin2.cmd
Calls the VisualAge for C++ icc link editor.

fhbolink.cmd
Calls the link386 link editor.

fhborc.cmd
Calls the OS/2 resource compiler.

fhbplbld.cmd
Calls the OS/2 PL/1 compiler.

fhbpllnk.cmd
Calls the OS/2 PL/1 link editor.

edcc.jcl
Calls the C/370 JCL procedure.

fhbcobm.jcl
Calls the COBOL for MVS compiler.

fhbm370.jcl
Calls the C/370 compiler.

fhbmasm.jcl
Calls the MVS assembler.

fhbmc.jcl
Calls the C/370 compiler.

fhbmlink
Calls the MVS linkage editor.

fhbmpli.jcl
Calls the PL/1 MVS compiler.

fhbplked.jcl
Calls the C370 prelinker.

fhbtclnk
Calls the TeamConnection pseudo linker.

fhbwcomp.c
Calls the Microsoft Visual C++ compiler

fhbwlink.c
Calls the Microsoft linker

gather.cmd
Calls the Gather tool.

Sample parsers

The following sample parser files are shipped with TeamConnection and are installed on the server in the SAMPLES directory.

fhbcbprs.cmd
A parser for COBOL applications.

fhbcpp1.dll, fhbcpp2.dll, fhbcpp3.dll, fhbcbp.exe, fhbcbprs.cmd
An OS/2 parser for (REXX) COBOL applications.

fhbopars.cmd
A parser for C applications.

fhbcpp.c, func.c, fhbcpp.h, fhbcpp.exe (Intel),
A C parser for COBOL applications.

fhbwpars.c, fhbwpars.exe(Intel)
A parser for C applications

fhbplprs.cmd (Intel)
A parser for PL/1 applications.

mvsasmp.c, mvsasmp.h, mvsasmp.exe (Intel), mvsasmp (UNIX)
A parser for MVS assembly language applications.

Sample package files

gather.pkf
A package file for the Gather tool.

softdist.pkf
A package file for the Tivoli Software Distribution tool.

Appendix J. XML Support in TeamConnection

XML is a text-based tag language that has become the standard for defining and sharing data on the World Wide Web. It facilitates the exchange of data between disparate sources, creating a standard format for the universal sharing of data.

One of the many benefits of using XML is that software developers no longer have to resort to proprietary mechanisms to format the data they are passing between applications or storing in files or databases. Applications use industry-agreed upon sets of XML tags and no longer need to be aware of the way the data is stored.

This section provides information on the XML support provided by TeamConnection. TeamConnection supports the following methods:

- get
- put

Details about each of these methods are discussed later in this section.

The following assumptions apply to using XML in TeamConnection:

- PUTs contain the entire set of objects in the views. TeamConnection will perform a merge operation on the server and remove old objects. Old objects are those that appear in the original view (on the server) but are not in the incoming XML stream.
- There is currently no support for explicit locking of objects as a separate XML request. Locking is allowed, however, during a GET (see the GET section below).
- Relationships/links between objects can be represented in different ways in XML (ie. embedded objects vs. separate objects). While TeamConnection will handle both cases in the incoming XML stream, it will only generate XML streams using one of these formats. TeamConnection does not currently use embedded objects.
- The ’get’ and ’put’ functions can only be performed on versioned objects however the ’query’ variant of the ’get’ function can be performed on either versioned or unversioned objects.

Format of the incoming XML stream/request

The XML request coming to XML needs to have the following MIME header:

```
verb url HTTP/1.1
HOST: tc-hostname:tc-portnum
Authorization: Basic encoded-password
Content-Type: text/xml
Context-Length: xxx (only on PUTs)

incoming-xml-stream(only on PUTs)
```

Notice the ’Content-Type’ tag. The XML interface will only recognise ’text/xml’ type of streams.

XML requests that support options expect the format of the URL to be as follows:

```
http://url?option1=value1&option2=value2...
```

TeamConnection ships with a sample C++ client program (called GETXML) that generates the XML header/request. The source code for this program (GETXML.CPP) is also included.

Get Method

The URL should be in the following form:

release/workarea/view/part-name?option=value-pairs

Depending on the type of 'get' operation, all or part of the release/workarea/view/part-name portion of the URL is required.

Default

If no ″action″ option is specified, the default GET method is invoked. Workarea, view and part-name are optional and should be used to reduce the scope of the requested data. For example, a request of 'r' will return an XML stream containing *all* of the parts in the release 'r'; 'r/w' will return all of the objects in workarea 'w' in release 'r'; 'r/w/BuildView' will return all of the objects contained in all of the BuildView's visible in 'r/w'; and 'r/w/BuildView/foo.exe' will return just the objects in the BuildView rooted off of the part called 'foo.exe'.

Part names containing slashes '/' should not be treated differently. So a URL request of 'r/w/BuildView/src/main/main.exe' will return all of the objects in the BuildView for src/main/main.exe.

Supported options

```
lock
    - type: Boolean
    - action: Will lock the TCPart's being retrieved if 'true'
force
    - type: Boolean
    - action: Will break common links (if needed during a lock) if 'true'
```

Sample:

```
getxml get http://dugnt:8765/r/w/buildviewonelevel/src/tcrs/deviant/exe

HTTP/1.0 207 Multi-Status
Server: VisualAge TeamConnection/3.0.2
Content-Type: text/xml
Content-Length: 1611

<?xml version=″1.0″ ?>
<A:multistatus xmlns:A='DAV:' xmlns:B='http://dugnt:8765/'>
<A:response>
<B:TCPart xmlns:C='http://dugnt:8765/TCPart'
 id=″33bc5102-a572-31d2-b429-0925c4c71d72″>
<C:adName>src/tcrs/deviant/exe</C:adName>
<C:systemTimeStamp>1999/01/06 09:15:10</C:systemTimeStamp>
<C:lastUpdateTimeStamp>1999/01/06 09:15:10</C:lastUpdateTimeStamp>
<C:buildStatus>out_of_date</C:buildStatus>
<C:type>none</C:type>
```

```
<C:parser/>
<C:lockedBy/>
<C:generatedBy href="#35861484-a572-31d2-b429-0925c4c71d72"/>
</B:TCPart>
<B:FHBGeneratesInfo xmlns:C='http://dugnt:8765/FHBGeneratesInfo'
 id="35861484-a572-31d2-b429-0925c4c71d72">
<C:controller href="#35861481-a572-31d2-b429-0925c4c71d72"/>
<C:PartTimeStamp>1999/01/06 09:15:36</C:PartTimeStamp>
<C:pBE href="#35861481-a572-31d2-b429-0925c4c71d72"/>
</B:FHBGeneratesInfo>
<B:FHBuildEvent xmlns:C='http://dugnt:8765/FHBuildEvent'
 id="35861481-a572-31d2-b429-0925c4c71d72">
<C:adName>1340.0</C:adName>
<C:systemTimeStamp>1999/01/06 09:15:09</C:systemTimeStamp>
<C:beTS/>
<C:builderRC>0</C:builderRC>
<C:outputFileParms/>
<C:builder>noop</C:builder>
<C:buildParms/>
<C:translates afterHref="0">459df418-a572-31d2-b429-0925c4c71d72</C:translates>
</B:FHBuildEvent>
<B:FHBTransformedByInfo xmlns:C='http://dugnt:8765/FHBTransformedByInfo'
id="459df418-a572-31d2-b429-0925c4c71d72">
<C:controller href="#35861481-a572-31d2-b429-0925c4c71d72"/>
<C:PartTimeStamp/>
<C:includeFlag>0</C:includeFlag>
</B:FHBTransformedByInfo>
</A:response>
</A:multistatus>
```

Test Server

If the URL contains the ″action=testServer″ option, then the rest of the URL is ignored and just information about the server is returned:

```
getxml get http://dugnt:8765/?action=testServer

HTTP/1.0 200 Ok
Server: VisualAge TeamConnection/3.0.2
Content-Type: text/xml
Content-Length: 240

<?xml version="1.0" ?>
<B:testServer xmlns:A='DAV:' xmlns:B='http://dugnt:8765/'>
<B:family>vgtstest</B:family>
<B:user>dug</B:user>
<B:version>3.0.2</B:version>
<B:os>NT</B:os>
<B:language>English</B:language>
</B:testServer>
```

List

If the URL contains the ″action=list″ option, then only only key pieces of information about the root objects are returned (overwise it’s just like a normal ’get’):

```
getxml get http://dugnt:8765/r/w/buildview/%25?action=list

HTTP/1.0 207 Multi-Status
Server: VisualAge TeamConnection/3.0.2
Content-Type: text/xml
Content-Length: 592

<?xml version="1.0" ?>
<A:multistatus xmlns:A='DAV:' xmlns:B='http://dugnt:8765/'>
<A:response>
<B:TCPart xmlns:C='http://dugnt:8765/TCPart'
 id="33bc5102-a572-31d2-b429-0925c4c71d72">
<C:adName>src/tcrs/deviant/exe</C:adName>
</B:TCPart>
<B:FHBuildEvent xmlns:C='http://dugnt:8765/FHBuildEvent'
 id="35861481-a572-31d2-b429-0925c4c71d72">
<C:adName>1340.0</C:adName>
</B:FHBuildEvent>
<B:TCPart xmlns:C='http://dugnt:8765/TCPart'
 id="459df402-a572-31d2-b429-0925c4c71d72">
<C:adName>src/tcrs/deviant/c</C:adName>
</B:TCPart>
</A:response>
</A:multistatus>
```

Query

If the URL contains the ″sql=*sql-query*″ option, then the rest of the URL is ignored and the *sql-query* text is passed on to DB2:

```
 getxml query http://dugnt:8765/?sql=select+name,buildstatus+from+buildviewonelevel

HTTP/1.0 200 Ok              Server: VisualAge TeamConnection/3.0.2
Content-Type: text/xml
Content-Length: 471

<?xml version="1.0" ?>
<resultSet xmlns:A='DAV:' xmlns:B='http://dugnt:8765/' rows="3" columns="2"
 query="select name,buildstatus from buildviewonelevel">
<B:row>
<B:NAME>src/tcrs/deviant/exe</B:NAME>
<B:BUILDSTATUS>out_of_date</B:BUILDSTATUS>
</B:row>
<B:row>
<B:NAME>1340.0</B:NAME>
<B:BUILDSTATUS>success</B:BUILDSTATUS>
</B:row>
<B:row>
<B:NAME>src/tcrs/deviant/c</B:NAME>
<B:BUILDSTATUS>success</B:BUILDSTATUS>
</B:row>
</resultSet>
```

PUT Method

The URL should be in the following form:

release/workarea/view/part-name?option=value-pairs

The incoming XML stream must have an initial ″multistore″ element that contains all of the objects that are to be stored. The root object must the part-name specified in the URL. The ’view’ specified in the URL is used to determine which ’old’ objects to remove from TeamConnection.

Creating Objects:

When a new subclass of TCPart is created, the following 2 attributes must be specified:

- adName (name of part)
- component (a default component my be specified on the URL - see below)

When a new non-TCPart is created, a controller attribute referencing the object’s controller must be specified:

```
<F:controller  href=″controller-guid″/>
```

Deleting Objects

To delete subclasses of TCPart, you must use the following DELETE XML element:

```
<E:DELETE  id=’object-guid’>
```

These delete elements should appear first in the XML stream (see the example below).

Ordered Attributes/Relationships

If an attribute is ordered then it must have an ’afterHref’ attribute in the XML element. For example:

<C:translates afterHref=″123456″>4321</C:translates> Indicates that the attribute ’translates’ points to an object with GUID ’4321’ and in the set of pointers that makes up the ’translates’ attribute, this pointer comes after the one that points to the object with GUID ’123456’. The first object in the collection should have an afterHref of ’0’.

If a link object is used to connect two objects then the ordering information can go on the link object rather than the on the attributes in the source or target objects. In this case, the ’afterHref’ attribute is expected on the XML element of link object. For example:

```
<B:ADLinkTCPartDependsOnTCPart  xmlns:
   C=’http://dugnt:8765/ADLinkTCPartDependsOnTCPart’
id=″111″  afterHref=″555″>
<C:controller  href=″#222″/>
<C:adTarget  href=″#333″/>
<C:adSource  href=″#222″/>
</B:ADLinkTCPartDependsOnTCPart>
```

indicates that in the collection of link objects in either the source or target attributes (depending on which one is ordered) this new ADLinkTCPartDependsOnTCPart comes after the ADLinkTCPartDependsOnTCPart with a GUID of ’555’. Remember that only one side maybe ordered, not both.

Supported options

```
createComponents
   - type: Boolean
   - action: Will create components as needed if ’true’
force
   - type: Boolean
   - action: Will break common links on checkin (as needed) if ’true’

component
   - type: String
   - action: Specifies the default component to use if it’s not specified
      as an attribute on the object.
```

Example:

```
getxml put http://dugnt:8765/r/w/TCPart/newpart inputFile

where ’inputFile’ contains:

<?xml version=″1.0″ ?>
<A:multistore xmlns:A=’DAV:’ xmlns:B=’http://dugnt:8765/’
   xmlns:C=’http://dugnt:8765/TCPart’>
<B:DELETE id=’360262f3-5e88-0012-0000-000000000000″>
<B:TCPart id=″360262f1-5e47-000e-0000-000000000000″>
<C:adName>newpart</C:adName>
<C:component>c</C:component>
<C:lastUpdateTimeStamp>1998/09/18 09:41:06</C:lastUpdateTimeStamp>
<C:type>text</C:type>
<C:parser/>
</B:TCPart>
</A:multistore>
```

Appendix K. Extra Build Features

The appendix provides the following information:
- Default rules for build related information
- Upper case parsers in Cobol

Default rules for build-related information.

TeamConnection ships with a file called TCPART.RUL that provides the default rules for build-related information when parts are created in TeamConnection. This information can be found in the file called TCPART.RUL, which is located in the SAMPLES directory. You can modify TCPART.RUL as needed.

Each entry specifies a rule that applies to parts which match the given criteria for each rule. When TeamConnection finds a rule which matches the part being created, it applies the rules for the part and its parent parts, as specified.

If the rules specify a parent for the given part, the parent part will also be created based on the parent's rules. The rules are evaluated from the top down until a match is found or until no more rules are found.

Column position is not important, but each tag must begin on a separate line. Lines starting with // are comments. The TCPART.RUL file can be specified by setting the TC_PART_RULESFILE environment variable on the server to the location of this file.

A match is determined by four possible criteria: rule name, mask, release, and component.

rule The identifier of the rule. This required field is only used to match outputs when they are being generated based on the input's rules. Note that more than one rule can have the same name. Evaluation will stop on the first match.

mask (required)
The name of the part. The * and ? wildcards are supported. Eg. *.cbl, abc*.cpp, foo\src\*.obj.

release (optional)
If present, this Rule only applies to parts created in the named release. If not present, the rule applies to all TC Releases.

component (optional)
If present, this rule only applies to parts created with the named component. If not present, the rule applies to all TC Components. You can specify the following fields for each rule:

type

The type of contents of this file when it is stored in TeamConnection as a part. Allowed values are binary, text or none. Default is binary. Note that type is only used for generated outputs.

builder

The name of the TeamConnection Builder to be associated with the part. The builder is not created for you but must exist. A value of none means no builder will be associated with the part. Default is to not associate the part with a builder.

parser

The name of the TeamConnection Parser to be associated with the part. The parser is not created for you but must exist. A value of none means no parser will be associated with the part, Default is to not associate the part with a parser.

parameters

The build parameters, if any, to be attached to the part. quotes if there are spaces included in the parameters. Use the value of none to indicate no parameters.

parentRule

The name of the rule for the matching parent part. If parentRule is specified, then TeamConnection will attempt to create the parent part associated with the matching rule for the parent. The parent rule will be matched based on rule name, release, and component. The parent name will be created based on the mask in the parent rule. If the parent already exists, TeamConnection will connect the part to the existing parent.

connect

How the part will be connected to its parent in TeamConnection. Valid values are input, output, or dependent. Default is input. Note that connect is only valid with parentRule.

```
//-----------------------------------------------------------------------------
// Rules for Release NOTRACK
//-----------------------------------------------------------------------------

rule: cexe
   release: NOTRACK
   mask: *.exe
   type: binary
   builder: Clink

rule: cobj
   release: NOTRACK
   mask: *.obj
   type: binary
   builder: Ccompile

rule: cppobj
   release: NOTRACK
   mask: *.obj
   type: binary
   builder: CPPcompile

rule: gather
   release: NOTRACK
   mask: *.out
   type: binary
   builder: gatherer

rule: csource
   release: NOTRACK
   mask: *.c
   parser: Cparser
   parentrule: cobj
   connect: input

rule: cinclude
   mask: *.h*
   release: NOTRACK
   parser: Cparser


//--------------------------------------------------------------------------------
// Rules for component subcomp1
//--------------------------------------------------------------------------------

rule: csource
   component: subcomp1
   mask: *.c
   parser: Cparser


//--------------------------------------------------------------------------------
// General Rules
//--------------------------------------------------------------------------------

rule: cexe
   mask: *.exe
   type: binary
   builder: linker

rule: cobj
   mask: *.obj
   type: binary
   builder: icc

rule: cppobj
   mask: *.obj
   type: binary
   builder: icc

rule: inf
   mask: *.inf
   type: binary
   builder: ipfc
   parser: c

rule: help
   mask: *.hlp
   type: binary
   builder: ipfc
   parser: c

rule: res
   mask: *.res
   type: binary
   builder: res

rule: csource
   mask: *.c
   parser: Cparser
   parentrule: cobj
   connect: input

rule: cppsource
   mask: *.cpp
   parser: Cparser
   parentrule: cppobj
   connect: input

rule: ipf
```

```
   mask: *.ipf
   parser: ipf

rule: cinclude
   mask: *.h*
   parser: c

rule: rch
   mask: *.rch
   parser: rc
```

Services and Support

VisualAge TeamConnection Services and Support

Services

IBM consultants are available to help you, from planning to production and everything in between. For information about these services, please visit the following web site:

`http://www.software.ibm.com/ad/teamcon/services/`

If you are interested in VisualAge TeamConnection Services, contact IBM Software Development Services via e-mail at:

`websphere_consulting@us.ibm.com`

Support

If you have a question or problem regarding VisualAge TeamConnection, you can find support information and our telephone numbers at the following web site:

`http://www.software.ibm.com/ad/teamcon/support/`

Newsgroup

You can access VisualAge TeamConnection technical information, exchange messages with other VisualAge TeamConnection users, and receive information regarding the availability of FixPaks by visiting our newsgroup at:

`news://news.software.ibm.com/ibm.software.teamcon`

Bibliography

IBM VisualAge TeamConnection Enterprise Server library

The following is a list of the TeamConnection publications. For a list of other publications about TeamConnection, including white papers, technical reports, a product fact sheet, and the product announcement letter, refer to the IBM VisualAge TeamConnection Enterprise ServerLibrary home page. To access this home page, select **Library** from the IBM VisualAge TeamConnection Enterprise Server home page at Web address http://www.software.ibm.com/ad/teamcon.

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

  Contains license, service, and warranty information.
- **Installation Guide (GC34-4742):**

  Lists the hardware and software that are required before you can install and use the IBM VisualAge TeamConnection Enterprise Server product, provides detailed instructions for installing the TeamConnection server and client.
- **Administrator’s Guide (SC34-4551):**

  Provides instructions for configuring the TeamConnection family server and administering a TeamConnection family.
- **Getting Started with the TeamConnection Clients (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.
- **User’s Guide (SC34-4499):**

  A comprehensive guide for TeamConnectionadministrators 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):

  Administrator’s Guide
  Getting Started with the TeamConnection Clients
  User’s Guide
  Commands Reference
  Quick Commands Reference
  Staying on Track with TeamConnection Processes

TeamConnection technical reports

The following is a list of technical reports available for TeamConnection. Refer to the IBM VisualAge TeamConnection Enterprise Server Library home page for the most up-to-date list of technical reports. To access this home page, select **Library** from the IBM VisualAge TeamConnection Enterprise Server home page at Web address http://www.software.ibm.com/ad/teamcon.

29.2147	SCLM Guide to TeamConnection Terminology
29.2196	Using REXX Command Files with TeamConnection MVS Build Scripts
29.2231	TeamConnection Interoperability with MVS and SCLM
29.2235	Using REXX Command Files with TeamConnection MVS Build Scripts for PL/I Programs
29.2266	TeamConnection frequently asked questions: National Language Support (NLS) and Double-Byte Character Sets (DBCS)
29.2307	Data Driven TeamConnection User Exits
29.2333	Evolution of a New TeamConnection Family, Common Dos and Don'ts
29.2357	Evolution of a New VisualAge TeamConnection Family: Taking Advantage of Automation
29.3076	Configuration and Administration of DB2 Universal Database V5 by Users of VisualAge TeamConnection Enterprise Server V3
29.3088	Moving a VisualAge TeamConnection Version 3 Family
29.3090	Evolution of a VisualAge TeamConnection family: Using the Web and Shadowing to Build and to Distribute
29.3094	VisualAge TeamConnection 3: How to Do Routine Operating System Tasks
29.3096	Comparison Between CMVC 2.3.1 and VisualAge TeamConnection Enterprise Server 3
29.3098	VisualAge TeamConnection Version 3: Simple Build Function in UNIX
29.3099	VisualAge TeamConnection V3 Frequently Asked Questions: GUI and Line Command Clients for UNIX, OS/2, and Windows 32-bit
29.xxxx	Migrating CMVC 2.3.1 to VisualAge TeamConnection V3 (draft)

DB2

The following publications are part of the IBM DB2 Universal Database library of documents for DB2 administration. DB2 publications are available in HTML format from the DB2 Product and Service Technical Library at the following Web address:

`http://www.software.ibm.com/data/db2/library/`

- *Administration Getting Started* (S10J-8154–00)

  An introductory guide to basic administration tasks and the DB2 administration tools.
- *SQL Getting Started* (S10J-8156–00)

  Discusses basic concepts of DB2 SQL.
- *Administration Guide* (S10J-8157–00)

  A complete guide to administration tasks and the DB2 administration tools.
- *SQL Reference* (S10J-8165–00)

  A reference to DB2 SQL for programmers and database administrators.
- *Troubleshooting Guide* (S10J-8169–00)

  A guide to identifying and solving problems with DB2 servers and clients and to using the DB2 diagnostic tools.
- *Messages Reference* (S10J-8168–00)

  Provides detailed information about DB2 messages.
- *Command Reference* (S10J-8166–00)

  Provides information about DB2 system commands and the command line processor.
- *Replication Guide* (S10J-0999–00)

  Describes how to plan, configure, administer, and operate IBM replication tools available with DB2.

- *System Monitor Guide and Reference* (S10J-8164–00)

  Describes how to monitor DB2 database activity and analyze system performance.
- *Glossary*

  A comprehensive glossary of DB2 terms.

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)
- NLS and DBCS
  - *AIX 4, General Programming Concepts: Writing and Debugging Programs.* (SC23-2533-02). See chapter 16 ″National Language Support″ for an updated contents of the AIX 3 material (see below).
  - *AIX 4, System Management Guide: Operating System and Devices* (SC23-2525-03). See chapter 10, ″National Language Support″ for system tasks.
  - *AIX Version 3.2 for RISC System/6000, National Language Support* (GG24-3850).
  - *Internationalization of AIX Software, A Programmer’s Guide* (SC23-2431).
  - *National Language Design Guide Volume 1* (SE09-8001-02). This manual contains very good information on how to enable an application for NLS.
  - *National Language Design Guide Volume 2* (SE09-8002-02). This manual provides information on the IBM language codes (consult the ″Language codes″ chapter).

Glossary

This glossary includes terms and definitions from the *IBM Dictionary of Computing*, 10th edition (New York: McGraw-Hill, 1993). If you do not find the term you are looking for, refer to this document's index or to the *IBM Dictionary of Computing*.

This glossary uses the following cross-references:

Compare to
Indicates a term or terms that have a similar but not identical meaning.

Contrast with
Indicates a term or terms that have an opposed or substantially different meaning.

See also
Refers to a term whose meaning bears a relationship to the current term.

A

absolute path name. A directory or a part expressed as a sequence of directories followed by a part name beginning from the root directory.

access list. A set of objects that controls access to data. Each object consists of a component, a user, and the authority that the user is granted or is restricted from in that component. See also *authority*, *granted authority*, and *restricted authority*.

action. A task performed by the TeamConnection server and requested by a TeamConnection client. A TeamConnection action is the same as issuing one TeamConnection command.

agent. See *build agent*.

alternate version ID. In collision records, the database ID of the version of a driver, release, or workarea where the conflicting version of a part is visible.

approval record. A status record on which an approver must give an opinion of the proposed part changes required to resolve a defect or implement a feature in a release.

approver. A user who has the authority to mark an approval record with accept, reject, or abstain within a specific release.

approver list. A list of user IDs attached to a release, representing the users who must review part changes that are required to resolve a defect or implement a feature in that release.

attribute. Information contained in a field that is accessible to the user. TeamConnectionenables family administrators to customize defect, feature, user, and part tables by adding new attributes.

authority. The right to access development objects and perform TeamConnection commands. See also *access list*, *base authority*, *explicit authority*, *granted authority*, *implicit authority*, *restricted authority*, and *superuser privilege*.

authority group. A group of TeamConnection actions that a member of the authority group is authorized to perform.

B

base authority. The set of actions granted to a user when a user ID is created within a TeamConnection family. See also *authority*. Contrast with *implicit authority* and *explicit authority*.

base name. The name assigned to the part outside of the TeamConnection server environment, excluding any directory names. See also *path name*.

base part tree. The base set of parts associated with a release, to which changes are applied over time. Each committed driver or workarea for a release updates the base part tree for that release.

build. The process used to create applications within TeamConnection.

build associate. A TeamConnection part that is not an input to or an output from a build. An example of such a part is a read.me file.

build cache. A directory that the build processor uses to enhance performance.

build dependent. A TeamConnection part that is needed for the compile operation to complete, but it will not be passed directly to the compiler. An example of this is an include file. See also *dependencies*.

builder. An object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers.

build event. An individual step in the build of an application, such as the compiling of hello.c into hello.obj.

build input. A TeamConnection part that will be used as input to the object being built.

build output. A TeamConnection part that will be generated output from a build, such as an .obj or .exe file.

build pool. A group of build servers that resides in an environment. The environment in which several build servers operate. Typically, several servers are set up for each environment that the enterprise develops applications for.

build scope. A collection of build events that implement a specific build request. See also *build event*.

build script. An executable or command file that specifies the steps that should occur during a build operation. This file can be a compiler, a linker, or the name of a .cmd file you have written.

build server. A program that invokes the tools, such as compilers and linkers, that construct an application.

build target. The name of the part at the top of the build tree which is the final output of a build. TeamConnection uses the build target to determine the scope of the build. See also *build tree*.

build tree. A graphical representation of the dependencies that the parts in an application have on one another. If you change the relationship of one part to another, the build tree changes accordingly.

C

change control process. The process of limiting and auditing changes to parts through the mechanism of checking parts in and out of a central, controlled, storage location. Change control for individual releases can be integrated with problem tracking by specifying a process for the release that includes the tracking subprocess.

check in. The return of a TeamConnection part to version control.

check out. The retrieval of a version of a part under TeamConnection control. In non-concurrent releases, the check out operation does not allow a second user to check out a part until the first user has checked it back in.

child component. Any component in a TeamConnection family, except the root component, that is created in reference to an existing component. The existing component is the parent component, and the new component is the child component. A parent component can have more than one child component, and a child component can have more than one parent component. See also *component* and *parent component*.

child part. Any part in a build tree that has a parent defined. A child part can be input, output, or dependent. See also *part* and *parent part*.

client. A functional unit that receives shared services from a server. Contrast with *server*.

collision record. A status record associated with a workarea or driver, a part, and one of the following:
- The workarea or driver's release
- Another workarea

TeamConnection generates a collision record when a user attempts to replace an older version of a part with a modified version, another user has already modified that part, and the first user's modification is not based on this latest version of the part.

command. A request to perform an operation or run a program from the command line interface. In TeamConnection, a command consists of the command name, one action flag, and zero or more attribute flags.

command line. (1) An area on the Tasks window or in the TeamConnection Commands window where a user can type TeamConnection commands. (2) An area on an operating system window where you can type TeamConnection commands.

committed version. The revision of a part that is visible from the release.

common part. A part that is shared by two or more releases, and the same version of the part is the current version for those releases.

comparison operator. An operator used in comparison expressions. Comparison operators used in TeamConnection are > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), = (equal to), and <> (different from).

component. A TeamConnection object that organizes project data into structured groups, and controls configuration management properties. Component owners can control access to data and notification of TeamConnection actions. Components exist in a parent-child hierarchy, with descendant components inheriting access and notification information from ancestor components. See also *access list* and *notification list*.

concurrent development. Several users can work on the same part at the same time. TeamConnection requires these users to reconcile their changes when they commit or integrate their workareas and drivers with the release. Contrast with *serial development*. See also *workarea*.

configurable field. A field that a family administrator can add to certain TeamConnection objects to customize the kind of information that TeamConnection stores in relation to those objects.

configuration management. The process of identifying, managing, and controlling software modules as they change over time.

connecting parts. The process of linking parts so that they are included in a build.

context. The current workarea or driver used for part operations.

corequisite workareas. Two or more workareas designated as corequisites by a user so that all workareas in the corequisite group must be included as members in the same driver, before that driver can be committed. If the driver process is not used in the release, then all corequisite workareas must be integrated by the same command. See also *prerequisite workareas*.

current version. The last visible modification of a part in a driver, release, or workarea.

current working directory. (1) The directory that is the starting point for relative path names. (2) The directory in which you are working.

D

daemon. A program that runs unattended to perform a standard service. Some daemons are triggered automatically to perform their task; others operate periodically.

database. A collection of data that can be accessed and operated upon by a data processing system for a specific purpose.

default. A value that is used when an alternative is not specified by the user.

default query. A database search, defined for a specific TeamConnection window, that is issued each time that TeamConnection window is opened. See also *search*.

defect. A TeamConnection object used to formally report a problem. The user who opens a defect is the defect originator.

delete. If you delete a development object, such as a part or a user ID, any reference to that object is removed from TeamConnection. Certain objects can be deleted only if certain criteria are met. Most objects that are deleted can be re-created.

delta part tree. A directory structure representing only the parts that were changed in a specified place.

dependencies. In TeamConnection builds there are two types of dependencies:

- **automatic**. These are build dependencies that a parser identifies.
- **manual**. These are build dependencies that a user explicitly identifies in a build tree.

See also *build dependent*.

descendant. If you descendant a development object, such as, a part or a user ID, any reference to that object is removed from TeamConnection. Certain objects can be descendant only if certain criteria are met. Most objects that are descendants can be re-created.

disconnecting parts. The process of unlinking parts so that they are not included in a build.

driver. A collection of workareas that represent a set of changed parts within a release. Drivers are only associated with releases whose processes include the track and driver subprocesses.

driver member. A workarea that is added to a driver.

E

end user. See *user*.

environment. (1) A user-defined testing domain for a particular release. (2) A defect field, in which case it is the environment where the problem occurred. (3) The string that matches a build server with a build event.

environment list. A TeamConnection object used to specify environments in which a release should be tested. A list of environment-user ID pairs attached to a release, representing the user responsible for testing each environment. Only one tester can be identified for an environment.

explicit authority. The ability to perform an action against a TeamConnection object because you have been granted the authority to perform that action. Contrast with *base authority* and *implicit authority*.

extract. A TeamConnection action you can perform on a builder, part, driver or release builder. An extraction results in copying the specified builder, part, or parts contained in the driver or release to a client workstation.

F

family. A logical organization of related data. A single TeamConnection server can support multiple families. The data in one family cannot be accessed from another family.

family administrator. A user who is responsible for all nonsystem-related tasks for one or more TeamConnection families, such as planning, configuring, and maintaining the TeamConnection environment and managing user access to those families.

family server. A workstation running the TeamConnection server software.

FAT. See *file allocation table*.

feature. A TeamConnection object used to formally request and record information about a functional addition or enhancement. The user who opens a feature is the feature originator.

file. A collection of data that is stored by the TeamConnection server and retrieved by a path name. Any text or binary file used in a development project can be created as a TeamConnection file. Examples include source code, executable programs, documentation, and test cases.

file allocation table (FAT). The DOS-, OS/2-, Windows 95-, and Windows NT-compatible file system that manages input, output, and storage of files on your system. File names can be up to 8 characters long, followed by a file extension that can be up to 3 characters.

fix record. A status record that is associated with a workarea and that is used to monitor the phases of change within each component that is affected by a defect or feature for a specific release.

freeze. The freeze action saves changed parts to the workarea. Thus, TeamConnectiontakes a snapshot of the workarea, including all of the current versions of parts visible from that workarea, and saves this image of the system. The user can always come back to this stage of development in the workarea. Note, however, that a freeze action does not make the changes visible to the other people working in the release.

Compare with *refresh*.

full part tree. A directory structure representing a complete set of active parts associated with the release.

G

Gather. A tool to organize files for distribution into a specified directory structure. This tool can be used as a prelude to further distribution, such as using CD-ROM or through electronic means like NetView DM/2. It can also be used by itself for distributing file copies to network-attached file systems.

GID. A number which uniquely identifies a file's group to a UNIX system.

granted authority. If an authority is granted on an access list, then it applies for all objects managed by this component and any of its descendants for which the authority is not restricted. See also *access list*, *authority*, and *inheritance*. Contrast with *restricted authority*.

graphical user interface (GUI). A type of computer interface consisting of a visual metaphor of a real-world scene, often as a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device.

GUI. Graphical user interface.

H

high-performance file system (HPFS). In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the existence of multiple, active file systems on a single personal computer, with the capacity of multiple and different storage devices. File names used with HPFS can have as many as 254 characters.

host. A host node, host computer, or host system.

host list. A list associated with each TeamConnection user ID that indicates the client machine that can access the family server and act on behalf of the user. The family server uses the list to authenticate the identity of a client machine when the family server receives a command. Each entry consists of a login, a host name, and a TeamConnection user ID.

host name. The identifier associated with the host computer.

HPFS. See *high-performance file system*.

I

implicit authority. The ability to perform an action on a TeamConnection object without being granted explicit authority. This authority is automatically granted through inheritance or object ownership. Contrast with *base authority* and *explicit authority*.

import. To bring in data. In TeamConnection, to bring selected items into a field from a matching TeamConnection object window.

inheritance. The passing of configuration management properties from parent to child component. The configuration management properties that are inherited are access and notification. Inheritance within each TeamConnection family or component hierarchy is cumulative.

integrated problem tracking. The process of integrating problem tracking with change control to track all reported defects, all proposed features, and all subsequent changes to parts. See also *change control*.

interest group. The list of actions that trigger notification to the user IDs associated with those actions listed in the notification list.

J

job queue. A queue of build scopes. One job queue exists for each TeamConnection family.

L

local version ID. In collision records, the database ID of the version of the current workarea.

lock. An action that prevents editing access to a part stored in the TeamConnectiondevelopment environment so that only one user can change a part at a time.

login. The name that identifies a user on a multi-user system, such as AIX or HP-UX, Solaris, or Windows NT. In OS/2 and Windows 95, the login value is obtained from the TC_USER environment variable.

M

map. The process of reassigning the meaning of an object.

metadata. In databases, data that describe data objects.

N

name server. In TCP/IP, a server program that supplies name-to-address translation by mapping domain names to Internet addresses.

National Language Support (NLS). The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

Network File System (NFS). The Network File System is a program that enables you to share files with other computers in networks over a variety of machine types and operating systems.

notification list. An object that enables component owners to configure notification. A list attached to a component that pairs a list of user IDs and a list of interest groups. It designates the users and the corresponding notification interest that they are being granted for all objects managed by this component or any of its descendants.

notification server. A server that sends notification messages to the client.

NTFS. NT file system.

NVBridge. A tool for automatic electronic distribution of TeamConnection software deliverables within a NetView DM/2 network.

O

operator. A symbol that represents an operation to be done. See also *comparison operators*.

originator. The user who opens a defect or feature and is responsible for verifying the outcome of the defect or feature on a verification record. This responsibility can be reassigned.

owner. The user who is responsible for a TeamConnection object within a TeamConnection family, either because the user created the object or was assigned ownership of the object.

P

parent component. All components in each TeamConnection family, except the root component, are created in reference to an existing component. The existing component is the parent component. See also *child component* and *component*.

parent part. Any part in a build tree that has a child defined. See also *part* and *child part*.

parser. A tool that can read a source file and report back a list of dependencies of that source file. It frees a developer from knowing the dependencies one part has on other parts to ensure a complete build is performed.

part. A collection of data that is stored by the family server and retrieved by a path name. They include text objects, binary objects, and modeled objects. These parts can be stored by the user or the tool, or they can be generated from other parts, such as when a linker generates an executable file.

path name. The name of the part under TeamConnection control. A path name can be a directory structure and a base name or just a base name. It must be unique within each release. See also *base name*.

pool. See *build pool*.

pop-up menu. A menu that, when requested, appears next to the object it is associated with.

prerequisite workareas. If a part is changed to resolve more than one defect or feature, the workarea referenced by the first change is a prerequisite of the workarea referenced by later changes. A workarea is a prerequisite to another workarea if:

- Part changes are checked in, but not committed, for the first workarea.
- One or more of the same parts are checked out, changed, and checked in again for the second workarea.

problem tracking. The process of tracking all reported defects through to resolution and all proposed features through to implementation.

process. A combination of TeamConnection subprocesses, configured by the family administrator, that controls the general movement of TeamConnection objects (defects, features, workareas, and drivers) from state to state within a component or release. See also *subprocess* and *state*.

Q

query. A request for information from a database, for example, a search for all defects that are in the open state. See also *default query* and *search*.

R

raw format. Information retrieved on the report command that has the vertical bar delimiter separating field information, and each line of output corresponds to one database record.

refresh. This TeamConnection action updates a workarea with any changes from the release, and it also freezes the workarea, if it is not already frozen.

relative path name. The name of a directory or a part expressed as a sequence of directories followed by a part name, beginning from the current directory.

release. A TeamConnection object defined by a user that contains all the parts that must be built, tested, and distributed as a single entity.

restricted authority. The limitation on a user's ability to perform certain actions at a specific component. Authority can be restricted by the superuser, the component owner, or a user with AccessRestrict authority. See also *authority*.

root component. The initial component that is created when a TeamConnection family is configured. All components in a TeamConnection family are descendants of the root component. Only the root component has no parent component. See also *component, child component*, and *parent component*.

S

search. To scan one or more data elements of a set in a database to find elements that have certain properties.

serial development. While a user has parts checked out from a workarea, no one else on the team can check out the part. The user develops new material without interacting with other developers on the project. TeamConnection provides the opportunity to hold the part until the user is sure that it integrates with the rest of the application. Thus, the lock is not released until the workarea as a whole is committed. Contrast with *concurrent development*. See also *workarea*.

server. A workstation that performs a service for another workstation.

shadow. A collection of parts in a filesystem that reflects the contents of a TeamConnection workarea, driver, or release.

shared part. A part that is contained in two or more releases.

shell script. A series of commands combined in a file that carry out a function when the file is run.

SID. The name of a version of a driver, release, or workarea.

sizing record. A status record created for each component-release pair affected by a proposed defect or feature. The sizing record owner must indicate whether the defect or feature affects the specified component-release pair and the approximate amount of work needed to resolve the defect or implement the feature within the specified component-release pair.

stanza format. Data output generated by the Report command in which each database record is a stanza. Each stanza line consists of a field and its corresponding values.

state. workareas, drivers, features, and defects move through various states during their life cycles. The state of an object determines the actions that can be performed on it. See also *process* and *subprocess*.

subprocess. TeamConnection subprocesses govern the state changes for TeamConnection objects. The design, size, review (DSR) and verify subprocesses are configured for component processes. The track, approve, fix, driver, and test subprocesses are configured for release processes. See also *process* and *state*.

superuser. This privilege lets a user perform any action available in the TeamConnectionfamily.

system administrator. A user who is responsible for all system-related tasks involving the TeamConnection server, such as installing, maintaining, and backing up the TeamConnectionserver and the database it uses.

T

task list. The list of tasks displayed in the Tasks window. The user can customize this list to issue requests for information from the server. Tasks can be added, modified, or deleted from the lists.

TCP/IP. Transmission Control Protocol/Internet Protocol.

TeamConnection client. A workstation that connects to the TeamConnection server by a TCP/IP connection and that is running the TeamConnection client software.

TeamConnection part. A part that is stored by the TeamConnection server and retrieved by a path name, release, type, and workarea. See also *part*, *common part*, and *type*.

TeamConnection superuser. See *superuser*.

tester. A user responsible for testing the resolution of a defect or the implementation of a feature for a specific driver of a release and recording the results on a test record.

test record. A status record used to record the outcome of an environment test performed for a resolved defect or an implemented feature in a specific driver of a release.

track subprocess. An attribute of a TeamConnection release process that specifies that the change control process for that release will be integrated with the problem tracking process.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communications protocols that support peer-to-peer connectivity functions for both local and wide area networks.

type. All parts that are created through the TeamConnection GUI or on the command line will show up in reports with the type of TCPart as the part type. The TeamConnectionGUI and command line can only check in, check out, and extract parts of the type TCPart.

U

user exit. A user exit allows TeamConnection to call a user-defined program during the processing of TeamConnection transactions. User exits provide a means by which users can specify additional actions that should be performed before completing or proceeding with a TeamConnection action.

user ID. The identifier assigned by the system administrator to each TeamConnection user.

V

verification record. A status record that the originator of a defect or a feature must mark before the defect or feature can move to the closed state. Originators use verification records to verify the resolution or implementation of the defect or feature they opened.

version. (1) A specific view of a driver, release, or workarea. (2) A revision of a part.

version control. The storage of multiple versions of a single part along with information about each version.

view. An alternative and temporary representation of data from one or more tables.

W

workarea. An object in TeamConnection that you create and associate with a release. When the workarea is created, you see the most current view of the release and all the parts that it contains. You can check out the parts in the workarea, make modifications, and check them back into the workarea. You can also test the modifications without integrating them. Other users are not aware of the changes that you make in the workarea until you integrate the workarea to the release. While you work on files in a workarea, you do not see subsequent part changes in the release until you integrate or refresh your workarea.

working part. The checked-out version of a TeamConnection part.

Y

year 2000 ready. IBM VisualAge TeamConnection Enterprise Server is Year 2000 ready. When used in accordance with its associated documentation, TeamConnection is capable of correctly processing, providing and/or receiving date data within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software and firmware) used with the product properly exchange accurate date data with it.

Index

Readers' Comments — We'd Like to Hear from You

IBM VisualAge TeamConnection Enterprise Server
User's Guide
Version 3.0

Publication No. SC34-4499-04

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC34-4499-04

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department G7IA / Bldg 062
P.O. Box 12195
Research Triangle Park, NC
27709-2195

Fold and Tape **Please do not staple** Fold and Tape

SC34-4499-04

Cut or Fold
Along Line

Part Number: CT66GNA

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

CT66GNA

SC34-4499-04