



Build Manager's Guide

IBM Rational Synergy
Build Manager's Guide
Release 7.1

Before using this information, be sure to read the general information under [Notices](#) on page 115.

This edition applies to Version 7.1, Rational Synergy (product number 5724V66) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1992, 2009

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of contents

Chapter 1: Overview	1
Locating build management information	2
Build management road map	3
Preparation	3
Ongoing integration test cycle	3
System test cycle	3
Software release	3
Special considerations	3
Conventions	4
Command line interface	5
Option delimiter	5
Standards	5
Changes in Rational Synergy	6
Web mode and Traditional mode	6
Rational documentation	7
Rational Synergy Help systems	7
Readme	8
Contacting IBM Rational Software Support	9
Prerequisites	9
Submitting problems	9
Chapter 2: Prepare for build management	13
Before you start	14
UNIX build managers	14
All build managers	14
Setting up your environment	17
About the platform file	17
Setting up platforms	18
About releases	18
Modifying release values	20

About purposes and process rules	20
Using process rules for a new release	22
Insulated and collaborative development	23
Component development.	23
About parallel releases and platforms	24
About baselines.	24
Controlling the released project hierarchy.	25
About build management projects.	25
Creating the integration testing projects	26
Creating the system testing projects	28

Chapter 3: Build management basics 29

Before you start....	30
About builds.	31
Following build guidelines	31
Automating the build management process	33
Providing the application for testing	34
Using the build workflow	35
Integration test cycle	36
Update.	37
Show and resolve conflicts.	37
Build and test	37
Create a baseline	38
Working with a bad baseline	38
System test cycle	39
Update.	40
Show and resolve conflicts.	40
Build and test	40
Building with specific tasks	41
Releasing the software	42
Preparing for a new release	43
Mark a baseline for deletion.	44

Chapter 4: Update and conflicts **45**

Using the update operation	45
Updating with process rules	45
Updating manually	46
Update guidelines	47
Updating a project.	48
Working with selection rules	48
Update and baselines	49
Update with platform values	49
Reviewing update results	50
Diagnosing selection problems	54
Verifying the update properties	55
How conflict detection operates	57
How conflicts arise.	57
Detecting conflicts	58
Categories of conflicts	58
Conflicts and dependencies	59
Resolving conflicts	61

Chapter 5: How baselines work **63**

Working with baselines	63
Using a baseline.	64
Considering how to use a baseline	65
Creating a baseline	68
Publishing a baseline to developers	71
Baselines and the update process	72
Creating an incremental baseline	73
Removing unnecessary baselines	75

Chapter 6: How to share products	77
Sharing external projects	78
Preparing to use external projects	79
Creating an external project	80
Modifying the build process for multi-phased builds	82
Chapter 7: How to package an application	85
About installation areas and projects	86
Creating an installation project	87
Modifying the build process for installation projects	88
Chapter 8: Parallel releases	89
Creating a patch for a release	90
Setting the patch release	90
Including projects	90
Obtaining fixes from developers	91
Creating a release for a patch	91
Creating a patch	91
Using a parallel development environment	93
Building on parallel platforms	93
Setting up a parallel platform	94
Setting up parallel releases	95
Chapter 9: Project restructuring	97
Adding an existing project to your hierarchy	99
Cutting a project from your hierarchy	99
Deleting a project from your hierarchy	99
Converting a directory to a subproject	100
Adding a new project to an existing hierarchy	101

Chapter 10: Build management variations	103
Build management for UNIX and PC together	104
UNIX work areas with local files	105
Grouping projects	106
About the grouping project to be created.	106
Grouping projects versus project groupings.	106
Creating a grouping project	107
Creating a release for a patch.	107
Creating a patch	108
Creating a custom folder template query	109
Adding additional test phases	110
Appendix A: Convert to process rules	111
Process rules requirement.	111
Converting projects.	112
Converting to process rules for build managers.	112
Converting to process rules for developers	113
Appendix B: Notices	115
Trademarks	117
Terms and concepts	119
Index	129

1

Overview

The *Rational Synergy Build Manager's Guide* is intended for build managers. Build managers are the people who know how a software product fits together and how it is built. This document describes how to prepare for, perform, administer, and troubleshoot build management operations.

This document assumes that you are familiar with IBM® Rational® Synergy. The terms and concepts used, the methodology discussed, and the scenarios given in this document all assume that you have both a conceptual (task-based CM methodology) and practical (how to perform developer-level task-based CM operations) understanding of Rational Synergy. Additionally, this document describes how to perform build management tasks using a task-based CM methodology.

All steps in this document show how to complete an operation using the Rational Synergy GUI.

If you are **new to Rational Synergy**, the following Rational documents are prerequisites to this document:

- For terms, concepts, and methodology refer to the [Introduction to Rational Synergy](#).
- To complete a short tutorial in a pre-populated tutorial database, refer to the [Rational Synergy Tutorial](#).
- For a description of step procedures, refer to [Rational Synergy Help](#).
- For a description of commands and default settings, refer to [Rational Synergy Classic CLI Help](#).

The command line interface steps shown in this document use the Rational Synergy Classic CLI. The help for this CLI is called Rational Synergy Classic CLI Help.

There is also a CLI for Rational Synergy, with the help called Rational Synergy CLI Help, Web Mode. This document does **not** reference that help.

If you are **new to build management**, but are already familiar with Rational Synergy, this is the right document for you.

Locating build management information

All build management information is located in this document. Although this document is in HTML format, and HTML is a format that lends itself to random patterns of information access, read [Prepare for build management \(page 13\)](#) and [Build management basics \(page 29\)](#) in the order presented before reading other chapters and topics.

The topics in this document are ordered hierarchically, from the most basic information ([Prepare for build management \(page 13\)](#)) to the more complex ([Build management variations \(page 103\)](#)). The first time you read this document, do so in the order shown in the Table of Contents. After you are familiar with the document, you will want to save often-used topics in your Favorites folder. This will make it easier for you to find the information you need quickly.

Additionally, if your site uses IBM® Rational® Change, you can run predefined queries to gather information about the contents of Rational Synergy builds. For more information regarding Rational Change predefined build management queries, see "Querying for CM build information" in [Rational Change User Help](#).

Build management road map

The following items briefly describe the different operations you will need to complete to prepare for, then implement build management operations for your team. Each time you start a new project, you will complete these tasks.

Preparation

- [Setting up your environment \(page 17\)](#)
- [Creating the integration testing projects \(page 26\)](#)
- [Creating the system testing projects \(page 28\)](#)

Ongoing integration test cycle

- [Updating a project \(page 48\)](#)
- [Detecting conflicts \(page 58\)](#)
- [Build and test \(page 37\)](#)
- [Publishing a baseline to developers \(page 71\)](#)

System test cycle

- [Building with specific tasks \(page 41\)](#)
- [Updating a project \(page 48\)](#)
- [Detecting conflicts \(page 58\)](#)
- [Build and test \(page 37\)](#)

Software release

- [Releasing the software \(page 42\)](#)
- [Preparing for a new release \(page 43\)](#)

Special considerations

Several special topics are covered in later chapters. These topics go beyond the basics. Your site might or might not require them, but read about them so that you are aware of how to deal with special situations, if they occur at your site.

Conventions

The following describes the conventions used in this document.

Convention	Items	Examples
Bold	<ul style="list-style-type: none">— dialog box names and options— commands on menus and buttons— emphasis— icon names— registry keys (and uppercase)— toolbar button names	<ul style="list-style-type: none">— Start dialog box, Password field— Task box, Apply button— Do not change the path.— Query icon— HKEY_LOCAL_MACHINE— Current Task button
<i>Italic</i>	<ul style="list-style-type: none">— book titles— placeholders— roles and states— user input	<ul style="list-style-type: none">— <i>Build Manager's Guide</i>— Add your <i>picture.gif</i> file.— <i>build_mgr</i> and <i>shared</i>— Type <i>Yes</i>.
Courier	<ul style="list-style-type: none">— commands and options— code samples— file names— paths	<ul style="list-style-type: none">— <code>ccm start -h lego</code>— <code>void main ()</code>— The <code>foo.c</code> file.— <code>/user/local/ccm_docs</code>
UPPERCASE	<ul style="list-style-type: none">— environment variables and macros— registry keys (and bold)	<ul style="list-style-type: none">— <code>CCM_ADDR</code>— HKEY_LOCAL_MACHINE

Note A note contains important information about your Rational Synergy software.

Caution A caution indicates potential danger to the database, the database server, or some other integral piece of the Rational Synergy software or your system.

Command line interface

Rational Synergy supports the CLI under all supported platforms. You can execute any Rational Synergy command from the UNIX® shell or from the Windows® command prompt.

Option delimiter

By default, the Windows client supports the slash (/) option delimiter and the UNIX client supports the dash (-) option delimiter. Examples in this document are shown using both delimiters.

Standards

Instructions for editing text files are given using Notepad (Windows) or vi (UNIX) commands. Notepad and vi are the Rational Synergy default text editors. If you use a different text editor, substitute the appropriate commands.

Changes in Rational Synergy

Effective in the 7.0 release, the Rational Synergy interfaces, for both the graphical and command line, were enhanced to include build management operations. The other graphical interface is now referred to as Rational Synergy Classic.

When this document gives steps using the CLI, it uses the commands and default settings for the older Rational Synergy Classic CLI. If you want to use the new Rational Synergy CLI (Web mode), be sure to check the default settings for that interface, which are outlined in the [Rational Synergy CLI Help, Web mode](#).

Web mode and Traditional mode

Rational Synergy 7.0 improves wide area network (WAN) performance by introducing a new architecture where Rational Synergy clients communicate to a Web-based Rational Synergy server using the HTTP protocol. This architecture reduces the dependency on network latency by using parallel, asynchronous network communication between the client and server.

Rather than replacing the original network communication with this new technique, Rational Synergy 7.0 introduces the new technique as Web mode. The previous RFC architecture, which is referred to as Traditional mode, is still available for use.

Most developers and build managers can use Web mode. Users who need administration capabilities or other advanced features will need to use Traditional mode instead. Traditional mode behaves as it did in Synergy 6.5.

Users will need to use one of the Classic clients (CLI or GUI) for the following reasons:

- Most administrative operations
- Data migration

This document uses the Rational Synergy Classic CLI when referring to commands issued from the CLI.

See the *Readme* for detailed information regarding changes to Rational Synergy.

Rational documentation

The Rational documentation is available in HTML and PDF on the documentation DVD, and in PDF on the IBM® Rational® Info Center (<http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/index.jsp>). The documentation can be available to all users by mounting the DVD on a shared drive.

Note that information in the *Readme* takes precedence over information in the documentation or in any of the Rational Synergy Help systems. The most up-to-date version of the *Readme* is available to Rational Synergy users on the IBM Rational Info Center.

Rational Synergy Help systems

The following describes the Help systems that are available with different interfaces you might be using. The *Build Manager's Guide* references some of these interfaces and Help systems. However, when this document shows steps from the CLI, it refers to the Rational Synergy Classic CLI. See [Changes in Rational Synergy \(page 6\)](#) for details.

The Synergy Classic command line interface (CLI) Help for the 7.0 release of Rational Synergy contains all supported commands, except for some database administration commands, which are documented in the Administration Guides.

The Rational Synergy Classic graphical user interface (GUI) contains similar information in the Help system that opens from that interface. Although Help for the CLI is contained in that Help system, it is not the most current command line information because it does not include new commands and options. For the latest commands and options available in this release, start the Help while using the command line interface.

The most current help for the Rational Synergy Classic interface is help written for the 6.3 release. The Rational Synergy Classic GUI has had only minor changes since the 6.3 release. Check the *Readme* for information about those differences.

If you are not sure which help you are viewing, check the footer on the bottom of each help page, which identifies the release. In addition, help written for Rational Synergy Classic (both the CLI and GUI systems) has a light blue background; help for Rational Synergy has a white background.

Readme

The Rational Synergy *Readme* describes the new features in Rational Synergy, provides updates to the documentation, and contains sections on troubleshooting, contacting IBM Customer Support, and known errors. Look in the *Readme* for the latest updates to the installation documentation.

The *Readme* is an HTML document available on the product DVD and on the [Rational Synergy Information Center](#).

Information in the *Readme* takes precedence over information in the documentation or in the Help.

Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM® Rational® Software Support for assistance in resolving product issues.

Note If you are a heritage Telelogic customer, a single reference site for all support resources is located at <http://www.ibm.com/software/rational/support/telelogic/>

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from <http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html>

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html.
- For further assistance, contact your IBM representative.

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to <http://www.ibm.com/software/support/>.
- Be listed as an authorized caller in the service request tool.

Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

Severity	Description
1	The problem has a <i>critical</i> business impact: You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.
2	This problem has a <i>significant</i> business impact: The program is usable, but it is severely limited.
3	The problem has <i>some</i> business impact: The program is usable, but less significant features (not critical to operations) are unavailable.
4	The problem has <i>minimal</i> business impact: The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2. Describe your problem and gather background information. When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:
 - What software versions were you running when the problem occurred?
To determine the exact product name and version, use the option applicable to you:
 - Start the IBM Installation Manager and select **File > View Installed Packages**. Expand a package group and select a package to see the package name and version number.
 - Start your product, and click **Help > About** to see the offering name and version number.
 - What is your operating system and version number (including any service packs or patches)?
 - Do you have logs, traces, and messages that are related to the problem symptoms?
 - Can you recreate the problem? If so, what steps do you perform to recreate the problem?

- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
 - Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.
3. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:
- **Online:** Go to the IBM Rational Software Support Web site at <https://www.ibm.com/software/rational/support/> and in the Rational support task navigator, click **Open Service Request**. Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

For more information about opening a service request, go to <http://www.ibm.com/software/support/help.html>

You can also open an online service request using the IBM Support Assistant. For more information, go to <http://www.ibm.com/software/support/isa/faq.html>.
 - **By phone:** For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at <http://www.ibm.com/planetwide/> and click the name of your country or geographic region.
 - **Through your IBM Representative:** If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at <http://www.ibm.com/planetwide/>.

2

Prepare for build management

Build management is the process by which a software product for a company is built and managed.

When a company controls its code by using Rational Synergy, it gives the job of building and managing the software product to the build manager.

The build manager is the person responsible for managing the following processes:

- Create baselines from the initial version of software.
- Organize and refine the structure of your software.
- Set up build management projects for testing and staging.
- Set up and maintain process rules and folder templates.
- Collect software changes from developers, then build test areas.
- Run reports in Rational Change to find out features and tasks that are in or not in a build.
- Freeze software at important milestones, such as a customer release.
- Set up configuration information for the team (e.g., platform and release values).
- Make the latest changes available to developers.
- Delete baselines that are no longer needed.
- Recreate old software releases to identify problems and create fixes.

This list is a short summary of build management responsibilities. Each bulleted item carries with it specific operations, all of which are discussed in detail in this book.

Before you start...

Perform the following preparatory operations before moving on to the next section.

UNIX build managers

- Become a member of the *ccm_root* group.
All build managers and user *ccm_root* must be members of the *ccm_root* group on your UNIX network. Verify with your system administrator that this is the case.
- Set the group of the *build_workarea* directory to *ccm_root*. The *build_workarea* is the directory where you place all of your build management work areas. Set the permissions of the *build_workarea* directory so that they are readable and writable by the group.

```
su ccm_root
cd /shared_directory
mkdir build_workarea
chgrp ccm_root build_workarea
chmod 775 build_workarea
chmod g+s build_workarea
```

All build managers

- Select parallel notifications during update.
If you build from the Rational Synergy CLI, set up your environment so that you can see parallel notifications in your log file. This helps you to determine whether parallel versions from developers are being merged at check in, and alerts you to any possible configuration problems. Regardless of whether you build from the GUI or the CLI, use the **Detect Membership Conflicts** operation to detect parallel conflicts (as well as other conflicts).

In the [Options] section of your Rational Synergy Classic initialization file, set the following `reconfigure_parallel_check` option, then restart your session:

```
reconfigure_parallel_check = TRUE
```

When you set `reconfigure_parallel_check`, if a given set of update candidates have equal scoring and are parallel, you will receive a warning message in your `ccm_ui.log` similar to the following:

```
Warning: Parallel versions selected by selection
rules, latest create time will be used:
save.c-3
save.c-2.1.1
```

- Set your release properties to dictate how a team working on this release will use parallel versions to either allow both parallel check out and check in; allow parallel check out, but disallow parallel check in; or disallow both parallel check out and check in.
- Set up the build management work area
Set up a directory to be used for the build management work area. The directory should be on a shared drive or NFS-mounted partition where all users can access it.

Windows build managers: Be sure that all build managers have full control of the shared drive and mount the drive using the same drive letter.

Set the work area path template for shared access

Set up the work area path template to create the build management project work areas in a shared location so that other build managers can access them.

The work area path defaults to `C:\Documents and Settings\username\My Documents\Synergy\ccm_wa` (Windows users) or `home/ccm_wa` (UNIX users), where `home` is your home directory.

1. In the **Work** pane, click the **Work Area** tab.
2. Click the **Change Default Path** button.
The **Options** dialog box opens.
3. Set the **Base directory** field to the shared location where you want the build management projects to reside.

Both Windows and UNIX users should keep the **+Database name** check box set. This will cause the database name to be appended to the base directory.

You might need to reset the work area paths for existing build management projects to the shared location.

Optionally, you can divide your work area into release-specific or platform-specific work areas. This is useful to build managers to organize many projects into their own custom structure.

For example, say you have a **toolkit** project in each of two platforms that you build: **vista** and **winxp**. By setting a work area template to `N:\network_disk\Synergy\shared\ccm_wa\%release\%platform` (Windows users) or `home/ccm_wa/%release/%platform` (UNIX users), you would see the following work areas:

Windows users:

```
\user\local\shared\ccm_wa\shared\2.8\vista\toolkit  
\user\local\shared\ccm_wa\shared\2.5\winxp\toolkit
```

UNIX users:

```
/user/local/shared/ccm_wa/2.8/solaris/toolkit  
/user/local/shared/ccm_wa/2.5/linux/toolkit
```

Setting up your environment

Your environment needs to reflect the type of application you will build. The following sections describe how to set your environment:

- For a standard release
- To develop your application on more than one platform
- To develop your application for more than one release

About the platform file

Note If you need to develop your application on more than one platform, be sure to read this section and [About parallel releases and platforms \(page 24\)](#).

Platforms are used by the update operation. Update uses the platform value to update a project with the most appropriate objects.

You must specify platform names, followed by at least two spaces or tabs and a semicolon: `IBM-AIX` ;

You can define the values in the platform file to reflect the platforms on which your application needs to be built.

The platform file you will change is called `om_hosts.cfg`, and is located in `CCM_HOME\etc` (Windows server) or `$CCM_HOME/etc` (UNIX). This file applies to every session that uses that Rational Synergy installation.

Alternatively, you can use a different platform list for every database in the installation. To do so, copy the `om_hosts.cfg` file from `CCM_HOME\etc` (Windows server) or `$CCM_HOME/etc` (UNIX) to the **etc** directory under your database, then edit it to set up the platforms and hosts specific to that database.

Windows client users: Define a platform file for each database because developers typically will have their own Rational Synergy installation areas.

After you make the changes, you and all users of the database will need to restart your sessions if you want to view the new values. Use the **Properties** dialog box **Platform** list to view the new values.

Using platform values is optional. You only need to use platform values if you build different versions of your software on or for different hardware platforms or operating systems.

Setting up platforms

1. Log on as the Rational Synergy administrative user (Windows users) or as *ccm_root* (UNIX users).
2. Edit the *om_hosts.cfg* file, located in *CCM_HOME\etc* (Windows server) or *\$CCM_HOME/etc* (UNIX) or in the database.
3. Save and exit from the file.
4. Exit from the Rational Synergy administrative user (Windows users) or as *ccm_root* (UNIX users).

About releases

Note If you need to have more than one release, be sure to read this section and [About parallel releases and platforms \(page 24\)](#).

Note Set the release value for all projects and tasks in Rational Synergy.

Rational Synergy stores releases for your software application. A release enables you to mark projects, tasks, and folders for particular releases. It also helps you to keep track of which object versions were developed for each release.

Only build managers can create or modify releases. You can view releases in the **Release Properties** dialog box. Each Rational Synergy database has its own set of releases, although you can transfer them between databases by using Rational Synergy Distributed (DCM).

Releases contain settings that impact the process for a team, for example:

- You can define whether or not parallel development will be allowed in a release.
- The release defines the process for a team by identifying a set of process rules available for use for that release.

For example, a typical release can be any of the following. This example shows the release, which is created by the build manager. It is made up of the [component name \(page 122\)](#) and the [component release \(page 122\)](#). The release is what the users see.

Release	Component name	Component release
1.0		1.0
2.0		2.0
2.0_patch		2.0_patch
Synergy/7.0	Synergy	7.0
editor/2.0	editor	2.0
editor/2.1	editor	2.1

A release consists of an optional component name and release delimiter (slash, by default), and a component release. The component name might represent the name of an application or component, such as **Synergy** or **editor**. The component release identifies the specific release of that application or component.

Note that the component name is not a mandatory part of the release. In the first row in the table above, the **1.0** component name does not have a component, and Rational Synergy leaves it blank

When you create a new release, you can create it based on an existing release, and the new release inherits properties of that release automatically.

The release can be any text string up to 117-characters long. (The component name can contain a maximum of 85 characters; the component release can contain a maximum of 32 characters.) For example, a release could be **Integrations/telecom_patch**

Component names and component releases must not start with the following characters:

/ \ ' " : * ? [] @ % - + ~ space, tab

Second and subsequent characters cannot include the following:

/ \ ' " : * ? [] @ %

Note that the component name and component release can contain the version delimiter character (by default -) if it is not one of the restricted characters.

Whenever an object is checked out, Rational Synergy automatically will copy the release from the current task to the new object.

Modifying release values

Rational Synergy provides several ways to modify release values. The following procedures are described in Rational Synergy Help.

- [Creating or copying a release](#)

Create a release when you want to define a release that is not based on an existing release. Copy a release when you want to base your new release on an existing release.

- [Deleting an object \(release\)](#)

This general delete topic includes information specific to deleting a release from your database.

- [Modifying the properties of a release](#)

Use this procedure when you want to make changes to an existing release.

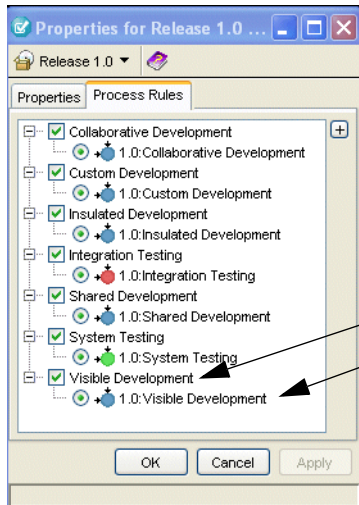
Note that Release 7.1 supports the automatic creation of component tasks ([component task \(page 122\)](#)) for baselines. The **Component Tasks** option is a checkbox on the **Release Properties** dialog box. See the Rational Synergy Help for details.

About purposes and process rules

Use purposes to set up multiple *prep*, *shared*, *working*, or *visible* versions of the same project for different uses, such as different levels of testing. Each project has a purpose. A project purpose defines the project state and ensures that it will select the right members when you update.

Typically, you do not need to modify the purposes. If you need to create purposes, see [Creating a purpose](#) in Rational Synergy Help.

Each release contains a list of process rules that are valid for that release. This lets you control the process for a team for a particular release, and lets teams working on different releases use different processes.

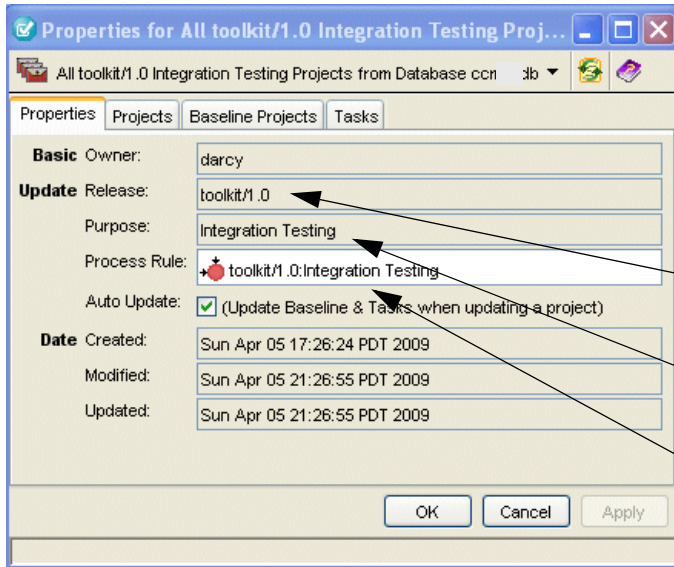


This figure shows the **Release Properties** dialog box displaying the process rules for release **1.0**.

This is a purpose.

This is a process rule.

A process rule specifies how a project will update. To use a process rule, a project must have a release and purpose set. The release and purpose combination for a project determines which process rule the project will use. There is one active process rule for each purpose for a release.



This figure shows a project grouping in the **Project Grouping Properties** dialog box called **All toolkit/1.0 Integration Testing Projects from Database ccm71db**.

The release for the project grouping is set to **toolkit/1.0**.

The purpose for the project grouping is set to **Integration Testing**.

This process rule defines how the baseline and tasks are selected.

If your site is new to Rational Synergy use, your database is set to use process rules by default.

If your site uses manual projects with Rational Synergy Classic, some existing templates might be set for projects to use manual update properties. If your team is converting to process rules during a release, follow the conversion process in [Converting projects \(page 112\)](#). After you finish the conversion, continue preparations by reading [About parallel releases and platforms \(page 24\)](#).

Using process rules for a new release

The standard purposes were renamed in Release 6.5. If your versions of the original standard purposes were modified in either 6.4 or 6.5, the upgrade to 7.1 might save a copy of your purpose, then rename it to the new standard purpose name, or it might create a new standard purpose. If you modified any of the standard purposes, review the purposes after the upgrade to ensure they meet your needs. If they do not, rather than modify any of the standard purposes, create new purposes.

Insulated and collaborative development

Insulated development occurs when developers do not want to receive changes made by other developers until after those changes have passed integration testing. Insulated development is a stable environment where the developer is insulated from other developers' ongoing changes.

Collaborative development occurs when developers want to receive the latest changes completed by other developers, rather than waiting until those changes have passed integration testing. Collaborative development enables developers to collaborate and share each others' changes as soon as the changes are completed.

When a developer checks out a project, he can choose the project's purpose (**Insulated Development** or **Collaborative Development**). A project's purpose can be changed in the **Properties** dialog box. The project's purpose will determine which tasks are added to the project when he updates it: **Insulated Development** indicates that the project selects the developer's tasks plus the most recent tested tasks, while **Collaborative Development** indicates that the project will select the developer's tasks plus all completed tasks, whether or not they have been tested.

Teams can choose to get each others' changes after they are tested or after they are checked in. The level of insulation is determined by which objects are selected when they update their projects. A process rule is a pattern that defines how projects created for a certain purpose will be updated, by automatically setting up the update properties for a project. For example, by default, Rational Synergy provides process rules for **Insulated Development** and **Collaborative Development**.

These process rules correspond to the purposes for **Insulated Development** and **Collaborative Development**.

Component development

Teams use component development to create applications from reusable parts. Developers can focus on the bigger solution they are trying to implement, rather than solving the same technical problems again and again. Additionally, teams can make products extensible by using Rational Synergy to assemble components into many different configurations that provide unique variants.

A component can be an individual file or project. Because file versions are reusable in Rational Synergy, they can be created or built in one project and used in another. For example, the `ccmscci.dll` library file might be built in the `ccmscci` project where its source code resides, but the same file can also be included as a member in both the `visual_studio_integration` and

rhapsody_integration projects. Each project can contain a different version of the file, depending on its needs.

Rational Synergy 7.1 supports the use of component development by allowing process rules to contain a folder or folder template for component development. Rational Synergy creates automatic and component tasks to support component development. However, these tasks are typically hidden from users because they are only used during component development.

You can create and add a folder or folder template that finds automatic and component tasks for use in your component development project. See [Adding a folder or folder template for component development to a process rule](#) in Rational Synergy Help for steps.

About parallel releases and platforms

A parallel release occurs when your company needs to develop more than one release of an application concurrently. For example, one team might be working on new features for release **toolkit/3.0** of the **toolkit** application while another team works on bug fixes for release **toolkit/2.1**. Likewise, you might be developing one release on multiple platforms.

To support parallel platforms or releases, you will create one integration build management project hierarchy and one system test build management project hierarchy for each unique release and platform combination you will build.

If you have not yet read [About the platform file \(page 17\)](#) and [About releases \(page 18\)](#), do so before going on.

About baselines

A baseline is a set of projects and tasks used to represent your data at a specific point in time. A baseline has many uses. When you perform an update, Rational Synergy uses a baseline as a starting point to look for new changes.

Creating a baseline for each **Integration Testing** and **System Testing** build enables testers and developers to refer back to the set of changes that were used to create the build. Typically, you will create a baseline for all projects in the same release and purpose. For example, you would create a baseline for each **Integration Testing** build using all **Integration Testing** projects for that release.

See [How baselines work \(page 63\)](#) for a thorough discussion.

Controlling the released project hierarchy

Before you can perform build management project activities, you will need to control the project hierarchy to be used as the starting point. Most sites use the last released version of their application. If your site already uses Rational Synergy, then you will want to create a baseline from the last project hierarchy released to your customers.

If you are a new user and have not migrated your source code into Rational Synergy yet, be sure to read the *CM Live!* book. This book discusses project structure and shows different ways in which source code can be migrated.

Note Be sure to create a baseline for your new project.

About build management projects

A build management project is a staging project used to build and test a test area or release. By default, build management projects support two levels of testing: integration testing and system testing. Therefore, when you create the build management projects for your application, you will usually create at least two versions of each project.

Note Your application probably will comprise many projects organized into a project hierarchy. When this document discusses a build management project (that is, the integration build management project), it refers to the entire project hierarchy, if you have one.

You can add more testing levels by adding a purpose and process rule for build management projects. For example, to add a performance testing level, create a purpose named **Performance Testing** for build management projects, create a process rule for the new purpose, then create a version of a build management project for it.

The first build management project, called the **integration testing project**, enables you to collect, build, and test the latest completed tasks checked in by developers. The members of this project are brought in through a query of all completed tasks.

See [Creating the integration testing projects \(page 26\)](#) to set up the integration testing projects.

The second build management project, called the **system testing project**, enables you to collect, build, and test the application in more detail, to reach an agreed-upon quality standard. The members of this project are brought in through a carefully controlled process.

See [Creating the system testing projects \(page 28\)](#) to set up the system testing project.

You might want to set build arguments for the integration testing projects or the system testing project. For example, you might want to build the integration test area with the debug flag or build the system test area with the optimize flag. For information on building, see [Following build guidelines \(page 31\)](#).

Note If you do not perform integration testing on all platforms, you do not need an integration build management project hierarchy for every platform.

Creating the integration testing projects

Copy a new project hierarchy from your baseline to build the integration testing area.

1. Find the baseline by using the **Query** dialog box.
2. Right-click over the baseline and select **Copy Projects**.

The **Copy Project** dialog box appears.


3. Set the **For Release** value to the appropriate release.

If an appropriate release is not in the **For Release** list, use **Refresh Choices**. If necessary, see [Creating or copying a release](#) (described in Rational Synergy Help).

4. Set the **For Purpose** choice to **Integration Testing**.
5. In the **Copy Subprojects** list, you will see the projects for the baseline.

If all of the locations you will use for build management work areas for these projects are not visible from the system you are running Rational Synergy on, copy the projects that will have visible work areas here.

For example if you develop your application on more than one platform, such as on Windows and UNIX, be sure to copy Windows projects separately from UNIX projects, using the appropriate Windows or UNIX machine. Additionally, if you develop your application for more than one release, be sure to create an integration testing project hierarchy for each release.

6. Change the new project version to a meaningful name by typing a new name in the **Versions** field. If the projects you are copying should have different versions, click  to see each unique version of the projects being copied.

Click **Use Default** and then type the new version for each entry.

The version should indicate the release and platform for a project hierarchy, and that it will be used for integration testing. For example, a version of **winxp_3.0_int** would be appropriate for an integration project on a Windows XP® platform with a release of 3.0.

7. Check the work area path and change it, if necessary.
8. By default, Rational Synergy updates all new projects after it copies the projects. If you do not want the new projects to be updated, clear the **Update new projects** check box.
9. Copy the projects by clicking **OK**.

The integration testing project hierarchy will be created.

Note Because you are using process rules, the update properties are already set up properly. The release and purpose of your project determine which process rule a project uses. As long as you have set the release and purpose on the project correctly, you do not need to do anything else to set the update properties.

Creating the system testing projects

Copy a new project hierarchy from your baseline. This project will be used to build the system testing area.

1. Find the baseline by using the **Query** dialog box.
2. Right-click over the baseline and select **Copy Projects**.


The **Copy Project** dialog box appears.

3. Set the **For Release** value to the appropriate release.

If an appropriate release is not in the **For Release** list, use **Refresh Choices**. If necessary, see [Creating or copying a release](#) (described in Rational Synergy Help).

4. Set the **For Purpose** choice to **System Testing**.
5. In the **Copy Subprojects** list, you will see the projects for the baseline.

For example if you develop your application on more than one platform, such as on Windows and UNIX, be sure to copy Windows projects separately from UNIX projects, using the appropriate Windows or UNIX machine. Additionally, if you develop your application for more than one release, be sure to create a system testing project hierarchy for each release.

6. Change the new project version to a meaningful name by typing a new name in the **Versions** field. If the projects you are copying should have different versions, click  to see each unique version of the projects being copied.

Click **Use Default** and then type the new version for each entry.

The version should indicate the release and platform for the project hierarchy, and that it will be used for system testing. For example, a version of **winxp_3.0_sys** would be appropriate for a system testing project on a Windows XP platform with a release of 3.0.

7. By default, Rational Synergy updates all new projects after it copies the projects. If you do not want the new projects to be updated, clear the **Update new projects** check box.
8. Check the work area path and change it, if necessary.
9. Copy the projects by clicking **OK**.

The system testing project hierarchy will be created.

3

Build management basics

The build process for a site is the process that the site undergoes to build test areas, find problems, and build a quality product. The build process is similar for all sites, no matter how large or small.

The build process consists of the following operations. You will perform these operations many times during the course of a product release. Each of the operations contains several steps. As you work through the steps for each operation, the build management process will take shape.

- **Update** the build management project hierarchy to collect the set of software to be built.

When you update a project hierarchy, you update the members of the hierarchy with the object versions associated with the tasks in the build management folders.

Information on update is in [Update guidelines \(page 47\)](#).

- **Show and resolve conflicts** to identify and correct potential configuration problems.

A conflict is a potential problem with a project. Keep in mind, however, that not every conflict is necessarily a bad thing. When you resolve a conflict, you correct the problem with your project configuration.

- **Build** your application with the latest completed tasks.

Information on builds is in [Following build guidelines \(page 31\)](#).

- **Provide the application** to be tested in a format such as a CD or an installation area from which users can run the application.

Information on providing applications is in [Providing the application for testing \(page 34\)](#).

Before you start...

Before you move on to the next sections, ensure that you have done the following:

- Set your project to use process rules (described in [Using process rules for a new release \(page 22\)](#)).
- Prepared an integration build management project hierarchy (described in [Creating the integration testing projects \(page 26\)](#)), which contains all the software to be built and tested, including all platforms and releases.
- Prepared a system test build management project hierarchy (described in [Creating the system testing projects \(page 28\)](#)), which contains all of the system test software to be built and tested, including all platforms and releases.
- Verified that your build mechanism works for these build management project hierarchies (described in [Following build guidelines \(page 31\)](#)).

About builds

The following two sections describe what you need to know when building, and the advantages of automating builds.

Following build guidelines

- Review the build output.

When you build the products, capture the output of the build process in a log file, then review the log carefully for errors or indications of problems in the build. (If your builds are automated through a batch file or script, you can direct the output into a log file.)

- Review new versions of makefiles.

Be sure to control the build environment carefully. Because the build environment is partially defined in your makefiles, you will need to review makefiles that are checked in by developers to verify that they will not adversely affect the build environment.

For example, a developer could check out a makefile and customize it to reference his personal test version of a library file, then accidentally check in the makefile. (That would cause the build manager's build to reference the developer's test library, and is likely to cause problems at some point.)

- Do not control Java™ class files.

When compiled, a Java source file can produce many class files, some of whose names are not fixed and known in advance. A class file for an anonymous inner class will have a name with a sequentially numbered suffix. If you attempt to control Java class files, the history of an object with such a name will not be meaningful because it could refer to different anonymous inner classes at different times. Furthermore, if a class is removed, the Java compiler will not delete a previously existing class file for that class, so you will have to remove the corresponding object from the project manually. If you attempt to work around this by deleting all class files before compiling, then all your controlled class file products will appear to change in every build.

Instead of controlling the class files, build a jar file (or ear or war file, as appropriate for your application) and control that.

- Control scripts and tools in Rational Synergy.

If you have scripts and tools to build your product, you should consider controlling these scripts and tools themselves in Rational Synergy. Keeping controlled copies of the right versions of the tools used to build each release

of your software makes it easier for you to rebuild older releases, and to build patches to older releases. Remember that you might also need to keep access to older machines and operating systems.

- **UNIX Users:**

Ensure that uncontrolled product files are writable by multiple build managers.

Most teams do not control all of their products within Rational Synergy. It is common to control top-level products, such as libraries and executables, but not to control intermediate products such as **.obj** files, since these files can be very big. If you control them, your database will grow quickly because many different copies of the product files will be created by different users. Uncontrolled products exist only in your work area.

Because products are uncontrolled, Rational Synergy does not set their owner or access permissions. Therefore, you need to take some extra steps to ensure that multiple build managers can update the uncontrolled products in the work area. If uncontrolled products are created with permissions that only the build manager who created the file can modify, another build manager will not be able to rebuild the project without first deleting the uncontrolled products.

To ensure that uncontrolled products are writable by multiple build managers, perform the following steps:

- Set each build manager's primary group to *ccm_root* so that new files are created in the *ccm_root* group.
- Set each build manager's *umask* so that new products are created with permissions that are writable by *owner* and *group*.
- Some platforms and shells will not set the file permissions correctly, even if you have set your *umask*. You might consider updating your makefiles to use the Korn shell (*ksh*), which does support the *umask*.

Alternatively, you can update your build scripts or makefiles either to delete intermediate products before building them, or to change the permissions of those files to set the group and allow group write access after building them.

Automating the build management process

Batch files or scripts are great ways to automate the update and build process.

The advantages of automation include:

- Your build process will be reproducible because it will be done the same way every time.
- Your build will be less error-prone because you will not have to remember all the details every time you build.
- You can schedule the process to run at times when you are not around or when the system is not being used heavily, such as at night.
- Others can take over the build process when you are away.
- You can write a batch file or script program to check the update and build logs automatically, and to notify you of words or patterns that indicate failure or possible problems. (Even if you automate this, you should still skim the logs.)

When you automate a build, be sure to log the output so you can diagnose any problems that occur.

Note When you automate builds, you will not be able to resolve conflicts before the build. If conflicts are found, you may need to run the build cycle (or parts of it) again.

Providing the application for testing

Some of the common ways to provide your application to customers include: DVD, CD-ROM, or an installation area from which users can run the application.

An **installation area** is a location in your file system where you can install a version of your software application so that users can run the application. Installation areas are used for testing, reproducing problems, or using your own application within your company. This area usually consists of the executables, libraries, batch files or scripts, and data and configuration files needed to run the application.

As a build manager, you probably will set up many installation areas for different uses. For example, you might have an installation area for a general release your team is working on, called **rel_int/3.2**. You might have another installation area for your SQE team to test the general release, and that area might be called **rel_sqe/3.2**. You might have another installation area for a service pack called **rel_sp/3.1**.

You can provide an application for testing in a variety of ways. See [Creating an installation project \(page 87\)](#) for a description of one common way to do this.

Using the build workflow

The build workflow is the process that a site undergoes to build test areas, find problems, and build a quality product. The build manager will need to do the following:

- Complete frequent integration test cycles
Each integration test cycle includes a cycle of the build process (update, show conflicts, resolve conflicts, build, and baseline).
- Complete several system test cycles
Each system test cycle includes a cycle of the build process (update, show conflicts, resolve conflicts, build, and baseline).
- Release the software
- Prepare for a new release

Each site will need to determine the frequency of the test cycles and the level of testing done based on answers to the following questions:

- What are the quality requirements of the product?
- How frequently is the product changing?
- How close to release is the product?
- How long does it take to test the product?

The answer to these questions will determine how frequently you complete integration and system test cycles. If the product is changing frequently, you will want to perform integration testing frequently. If you are near a release, you will probably be concentrating on system testing.

Note that you might choose to modify this workflow to work better for your organization.

Integration test cycle

The integration test cycle includes the following:

- [Update \(page 37\)](#)
- [Show and resolve conflicts \(page 37\)](#)
- [Build and test \(page 37\)](#)
- [Create a baseline \(page 38\)](#)

During the integration builds, all of the newly completed tasks from developers are gathered and built. They are gathered based on the integration testing process rule used by the integration testing project hierarchy.

The software is likely to have problems at this point, and may not even build successfully. The goal is to find problems right away, not to get a high quality installation area. This is because of the unstable nature of the software at this point in development.

The kinds of problems you might see during integration-level builds include:

- Parallel branches that have not been merged (you get one developer's changes but not the other developer's changes)
- A developer checked in only part of his changes, for example, a developer forgot to associate all necessary objects to complete a particular task.
- Two developers made incompatible changes, for example, they both added defines with the same name.
- A program fails to compile because of syntax errors, for example, the developer forgot to unit test.

Remember that the integration build area is not a stable environment because it contains the most recently completed tasks. Another reason is that the candidates change frequently as developers complete their tasks. (This is normal.)

The integration test cycle be short and frequent; this will help you to find problems as early in the development cycle as possible. Additionally, developers with insulated development projects do not bring in each others' changes until the tasks have passed integration testing.

The integration cycle works best if you can build and test every day, and make the newly tested tasks available to developers immediately after they pass testing.

Typically, the integration-level build cycle includes the following:

1. Developers continuously make changes and check them in by completing their tasks, without regard to the cycle. (The advantage is that the team is not interrupted or distracted by testing efforts.)
2. The build manager updates, shows conflicts, resolves conflicts, builds the hierarchy, and creates a new installation area or media to be tested. (Part of this can be automated and done as a nightly job.)
3. The build manager tests the resulting products with a short set of tests that verify that the product builds correctly and is usable. If defects are found, a member of the team creates tasks to fix the problems.
4. If no severe defects are found, the application is ready for use, for example, as a development test area. Note that this may not happen every day; some days severe defects will be found, and some days the build may not even succeed.
5. If the build manager finds no severe defects, he can continue by [Creating a baseline \(page 68\)](#). This makes the objects associated with the tasks in the baseline available to developers the next time they update their projects.

Now that you understand what you will be doing during the integration test cycle and why you need to complete these operations, you are ready to perform the operations.

Update

You are ready to update the build management project hierarchy. This is described in [Updating a project \(page 48\)](#).

Show and resolve conflicts

You now have updated your build management project hierarchy. You are ready to identify and resolve potential configuration problems **before** you build your application. This is described in [Detecting conflicts \(page 58\)](#).

Build and test

You have updated, dealt with conflicts, and are ready to build your application. Builds will vary greatly from site to site; use the following general guidelines when you build.

1. After your build completes, review the build logs.
2. Create an installation area or test media.

-
3. Run a short series of tests to perform testing.
 4. If the build fails, you can either add selected tasks and rebuild, or add fix tasks and begin the build process from update. See [Building with specific tasks \(page 41\)](#) for detailed instructions.

Create a baseline

You have now updated the integration testing projects, dealt with conflicts, built your products, and tested the application. You are ready to make the changes available to developers. See [Creating a baseline \(page 68\)](#).

For detailed baseline information, see [How baselines work \(page 63\)](#).

Working with a bad baseline

Sometimes a baseline has problems and developers should not use it. If this occurs, do one of the following:

- Create a new baseline that contains the fix, if there is one.

OR

- Roll back to the previous baseline.

See [Removing unnecessary baselines \(page 75\)](#), which gives the steps for marking a baseline for deletion and deleting a baseline using the Save Offline and Delete command. Marking a baseline for deletion will make it unavailable for developers. (After all developers have updated their projects, you can delete the bad baseline.)

System test cycle

The system test cycle includes the following:

- [Update \(page 40\)](#)
- [Show and resolve conflicts \(page 40\)](#)
- [Build and test \(page 40\)](#)
- [Building with specific tasks \(page 41\)](#)
- [Creating a baseline \(page 68\)](#)
- [Releasing the software \(page 42\)](#) (optional)
- [Preparing for a new release \(page 43\)](#) (optional)

The system test cycle enables you to do more in-depth testing on a specific set of tasks, insulated from ongoing changes by developers. The goal is to produce an installation area or to produce release media that meets a quality standard.

Because you can select which tasks to add to the system testing project, they are insulated from ongoing changes. This gives you the ability to build, fix, and test the software apart from ongoing changes until it meets an agreed-upon quality standard.

The system test area is more stable and easier to build because most of the integration problems will be resolved before the software is at the system test level.

The system test cycle usually is used in preparation for a milestone, such as a release. The frequency of the system test cycle and the level of testing will depend on several situations, such as:

- At the beginning of a release cycle, when many new features are being added, system testing may happen only infrequently because it takes longer and is harder to test, usually requiring the development of new test cases.

Also, since software development is still underway, the goal is to find defects and develop new tests, not to create a stable installation or release.

- After the development phase is over and the team is stabilizing the software and fixing defects, you may want to do system testing more frequently, perhaps once or twice per week.
- At the end of the development cycle, you will want to test every change, in anticipation that each iteration of the software may be the one that is released.

The following overview should give you an idea of what the system test build cycle includes:

1. Update the system test build management project hierarchy, show conflicts, resolve conflicts, then build. Because you are trying to obtain a very clean system test area, be sure to show and resolve conflicts carefully. You then will need to create a new system test installation area or media to be tested.
2. Test the resulting products. If defects are found, create tasks.
3. To fix the defects necessary to meet the quality standard for the system test area, do the following:
 - The project team decides which problems to fix.
 - Developers are assigned tasks to fix the problems.
 - Developers will set the new task as their current task and fix the problem.
 - Developers complete their current task. They might notify you when they complete their tasks.
 - The build manager adds the completed tasks to the system test folder (back to [step 1](#)).
4. When the system test area meets the agreed-upon quality standard, make it available for general use or release it to customers. See [Creating a baseline \(page 68\)](#) to learn how to contain the projects at this point in the release.

Update

You are ready to update the build management project hierarchy. This is described in [Updating a project \(page 48\)](#).

Show and resolve conflicts

You now have updated your build management project hierarchy. You are ready to identify and resolve potential configuration problems **before** you build your application. This is described in [Detecting conflicts \(page 58\)](#).

Build and test

This topic is discussed in [Build and test \(page 37\)](#).

Building with specific tasks

After testing is complete and if the quality of the software is approved, see [Releasing the software \(page 42\)](#).

If defects were found and your project team has determined the specific tasks they want you to build into the hierarchy, you need to add the approved tasks to the appropriate project grouping.

1. Right-click over the appropriate project grouping, and clear **Automatically Update Baseline & Tasks**.

This keeps the baseline and tasks from changing during an update operation. The baseline and tasks are a part of the project grouping; therefore, if you do not want a new baseline chosen and new tasks to come in automatically when you update a project, you must turn off the this option in the project grouping that the project is in.

2. Add approved tasks to the project grouping in any of the following ways:
 - Drag and drop the approved tasks onto the project grouping.
 - Right-click over the approved task and add it to the project grouping.
 - Use the **Tasks** tab in the **Project Grouping Properties** dialog box. See "Add a Task to a Change" in [Performing a Rebuild](#) in Rational Synergy Help for instructions.
3. Right-click over the appropriate project grouping, point to **Update**, and select **All Projects**.

The update operation will bring in the changes from the new task(s).
4. Show and resolve conflicts. (See [Resolving conflicts \(page 61\)](#) for details.)
5. Rebuild your product.
6. See [Creating a baseline \(page 68\)](#).

Once you are satisfied with the build quality, you can create a test baseline, which saves a copy of the build made available to SQE and enables developers to view the changes in the build, but does not publish or release the baseline for everyone's use.

Note that **Update** is still disabled for the appropriate project grouping. You can enable it when you are ready to accept tasks as specified in the process rule.

All the tasks that you want included in your next release or milestone are now in the project grouping. You can begin another build cycle with the approved fixes.

Releasing the software

As soon as your application has passed system testing, you are ready to release the baseline and all objects.

If you have already created a baseline, you can release it. If you need to, see [Creating a baseline \(page 68\)](#) to do so now. The following step shows you how to release a baseline.

- Select the system testing baseline(s), right-click over the baseline(s), and choose **Release**.

At this point, you might be ready to prepare your product for delivery to customers.

Additionally, you can rename the baseline or change the versions of the projects and products in the baseline, as you did when you published the baseline. You can perform these operations together from the CLI only.

1. Start Rational Synergy from the command prompt.

```
ccm start -h engine_host -d database_path -nogui
```

After the session starts, the Rational Synergy address (CCM_ADDR) is printed in your command window (Windows) or in the shell where you started the session (UNIX).

2. Set your role to *build_mgr*.

```
ccm set role build_mgr
```

3. Optionally change the name of the baseline or the versions on its projects.

Changing the version to a meaningful name is a good visual cue to reflect the purpose of the project. (See the discussion of version templates in step 2 in [Publishing a baseline to developers \(page 71\)](#).)

Additionally, you can use the baseline name to reflect the purpose of a release. Note that you can modify the baseline name after you create it.

```
ccm baseline -modify "7.1 Turn 10" -name "7.1 General Release" -versions  
-vt "%{platform:-}%{platform:+_}%{release}_GR"
```

4. Release the baseline.

```
ccm baseline -release_baseline "7.1 General Release"
```

Preparing for a new release

After you have released your application, you will be ready to start on the next release. You will need to complete the following operations:

- [Add new releases. \(page 43\)](#)
- [Update the release for all incomplete tasks. \(page 43\)](#)
- [Reuse the integration testing projects, if you have not created new ones already. \(page 44\)](#)
- [Creating the system testing projects \(page 28\)](#)
- [Tell developers to reuse their development projects. \(page 44\)](#)

1. Add new releases.

Add a new release (see [Modifying release values \(page 20\)](#)), select the process that will be used for the new release, and select the baseline release for the new release.

2. Check that the process rules are set up correctly for the new release.

3. Update the release for all incomplete tasks.

During the development cycle, some tasks are not included for the current release. One way of doing this is by not completing the task. To be able to bring in the task when it is completed during your new release, you will need to update the release value for the task so that update sees the task as a candidate to be included in your project configuration.

4. Update the release for all completed tasks not included in the release you just released. (Additionally, be sure to update the release values for change requests. You must use Rational Change to do this.)

Rational Change Users:

- Choose a query called **What is not in this build?** to run a query in Rational Change that shows change requests that are set to the current release, but are not in the build. This query prompts you for specific release information before performing the query.

OR

Rational Synergy Users:

- a. Start the **Query** dialog box.

Click **Find > Tasks**

In the **Query** dialog box, enter the following custom query. The following query returns all completed tasks for the given release

(*task_release*) that are not in the latest baseline with the given release (*baseline_release*) and purpose (*baseline_purpose*).

`is_available_task_of_release(task_release,
baseline_release, baseline_purpose)`

Further, the latest baseline is selected from all baselines in the *test_baseline*, *published_baseline* or *released* state. For a task to be considered to be in a baseline, it must be included in the update properties of all projects making up that baseline.

The results of the query shows tasks that were completed after the final build for the release, and were incorrectly left tagged for the release. (Tag these tasks for the new release.)

- b. If you need to add tasks that should be in the release to the appropriate project grouping, see [Adding tasks to a project grouping](#) in Rational Synergy Help for instructions.
5. Reuse the integration testing projects, if you have not created new ones already.
 - a. Right-click on the top-level integration testing project and select **Properties**.

The **Properties** dialog box appears.

- b. If you put the release in the version, be sure to change the version.
 - c. Save the changes by clicking **OK**.
- If the project contains subprojects, the release will be copied automatically to the subprojects.
6. Reuse the system testing project, if you have not created new ones already.
 7. Tell developers to reuse their development projects.

Developers will need to change the release value for the project hierarchy.

Mark a baseline for deletion

Once you are ready to prepare for a new release, you will want to clean up baselines that are no longer needed. During the course of a release, you will have created several baselines, especially during the integration testing phase, and will want to clear those that are no longer necessary to avoid cluttering the database.

See [Removing unnecessary baselines \(page 75\)](#) for steps.

4

Update and conflicts

Using the update operation

The update operation updates your project with the latest set of changes that satisfy the update properties. The update properties resolve to a baseline project and a set of tasks. Update computes the baseline project and the set of tasks differently, depending on whether you update projects manually or by using a process rule. These are distinct processes and will be discussed separately.

In both cases, once update computes the baseline project and tasks, it performs the following steps:

1. Builds a list of candidates based on the baseline project and tasks.
 - a. Each member of the baseline project is a candidate.
 - b. Each object version associated with each task is a candidate.

This step is performed once for each **project**.

2. After the candidates have been collected, a simple set of selection rules is used to select the best object version for each [directory entry \(page 123\)](#).

This step is performed once for each **directory entry** in the project.

3. Update uses the selected object version in that directory entry.

This step is performed once for each **directory entry** in the project, if the selection has changed.

Updating with process rules

If the project uses a process rule, the baseline project and tasks are computed from the baseline and tasks on the project's project grouping. A project grouping's baseline consists of a collection of baseline projects. The tasks on the project grouping are exactly the set of tasks used by update. The baseline project used by update is the project in the project grouping's baseline that matches the project being updated. A baseline project is considered matching if it has the same name and instance as the project being updated, and the same platform, if the project has a platform.

The project grouping's baseline and tasks are computed as follows:

1. If the project grouping has auto-update off, the baseline and tasks that were previously computed and saved on the project grouping are used.

But, if the process rule specifies that baseline projects are selected using the **Latest projects** method, then the project grouping will not have a baseline, and the baseline projects will be recomputed in the same way as if auto-update were on.

2. If the project grouping has auto-update on, the baseline and tasks are computed as follows:
 - a. The baseline is computed according to the process rule for the project grouping. The process rule specifies rules that are used to determine which baseline to use: **Latest baseline**, **Baseline specified on process rule**, **Latest projects**, or **Baseline specified on project grouping**. When you update a project, update identifies a baseline project from the baseline selected from the process rule's baseline selection rules.

If the process rule specifies that baseline projects are selected using the **Latest projects** method, then the project grouping will not have a baseline, and each project will have a baseline project that is the latest matching static project for the release and purpose specified in the process rule.
 - b. The tasks are computed from the folders and tasks specified on the instantiated process rule for the project grouping.
 - * For each query-based folder, update re-evaluates the query to update the tasks in the folder, and then makes a list of the tasks in all of the folders.
 - * If the project grouping has a baseline, the tasks in the baseline are subtracted from this computed set of tasks.
 - * If any tasks have been manually added or removed from the project grouping, those tasks are added or removed from the computed set of tasks.

Updating manually

Use the custom purpose if you need to select your baseline and tasks manually. When you use the custom purpose, you can choose a baseline directly on the project grouping. You can then add tasks manually by selecting the appropriate tasks, right-clicking, and choosing **Add to Project Grouping**.

Update guidelines

The following are guidelines for build managers to consider during update operations:

- Update all projects to be built.

You will need to update the entire build management project hierarchy in one operation. This ensures that you build consistent versions into your application. If one project is not updated, the products built for it may not be compatible with the rest of the products you build.

If building the project hierarchy in one operation is not feasible, see [Modifying the build process for multi-phased builds \(page 82\)](#) for a multi-phased approach.

Note Do not use the **Use Version** operation as a substitute for update. Sometimes you might need to use a particular version for a quick test; however, only a full update enables you to receive a complete, consistent project configuration, reducing the probability of getting partial changes.

Updating a project

At this point in your application development, you are ready to update the build management project hierarchy.

1. Right-click over the appropriate project, and point to **Update** and choose **Members and Subprojects**.
2. When the update is complete, be sure to review the results.

If you had any problems during the update operation, you might need to read more extensive output. See [Diagnosing selection problems \(page 54\)](#).

Working with selection rules

When you update a project or directory, the update operation considers the candidates available for each object in the project, and compares their properties with the properties of the project, such as the platform, to select the most qualified candidate.

The selection rules analyze each candidate object version.

- If the properties for the object version are incompatible with the project, it is considered ineligible and will **never** be selected.

For example, if user *joe* has an object version in the *working* state, and user *tim* performs an update, user *joe's* *working* state object version will never be included in user *tom's* project configuration.

- The object version receives scores, based on points, for its characteristics. The points are cumulative. For example:

A *working* object version receives five points for its status.

An object version whose platform matches that of the project platform receives eight points.

After all candidate object versions have been analyzed, update selects the one with the highest score. (The ineligible candidates are not considered.)

If multiple candidates are tied for the highest score, update selects the one with the latest creation time.

Note Regardless of whether you build from the Rational Synergy GUI or CLI, use the **Detect Membership Conflicts** operation to detect parallel conflicts (as well as other conflicts). The **Membership Conflicts** dialog box will display parallel messages whenever multiple candidates are parallel.

This generally means that a developer forgot to merge his changes, or a selection property was not set to the right value on one of the parallel branches.

For help with conflict detection, see [Detecting conflicts \(page 58\)](#).

Update and baselines

When a project is updated based on process rules, update re-evaluates which [baseline project \(page 121\)](#) it should use. For more information on baselines and updating, see [Baselines and the update process \(page 72\)](#).

Sometimes, it is not possible or practical for you to create full baselines that include all projects for a given release and prep purpose. In these situations, see [Creating an incremental baseline \(page 73\)](#).

Update with platform values

When you update a project that has a platform value set, update prefers candidates with matching platform values. It will never select a candidate with a platform value that does **not** match, but it could select a candidate that does not have a platform value set.

During update, the properties of the candidates are compared with the properties of the project. The platform values are compared as follows:

- If both the project and the candidates have platform values set and the platform values do not match, update **never** selects the candidate.
- If the platform values match, the candidate is **preferred**; that candidate receives eight points.
- If neither the project nor the candidate have a platform, the candidate is **preferred**; that candidate receives eight points.

The **platform** property is used primarily on projects and products. Source code usually is written so the same file can be built on different platforms (for example, using `#ifdef`'s); therefore, individual source files typically do not need **platform** properties set.

Reviewing update results

- Review the update results to be aware of problems.

During an update, the output is written to the session log file and the **Messages** dialog box. However, reading the update results in this log file can be cumbersome because all other messages also are written to this file.

You can redirect the `ccm_client.log` (user interface log) file so that it points to a location other than your Windows profile directory (Windows users) or `ccmlog` directory (UNIX users). This is done by setting the `ccm.user.properties` key in your `ccm.user.properties` file as follows:

1. Open your properties file.

For Windows users, the file is called `ccm.user.properties` and is located in your Windows profile directory.

For UNIX users, the file is called `.ccm.user.properties` and is located in your home directory.

* To redirect a log file to `C:\cmsynergy\synint\bob`, do the following:

```
user.default logfile=
C:\\cmsynergy\\synint\\bob\\ccm_client.log
```

* To rename a log file (typically when working in multiple databases), for a database named `int`, do the following:

```
user.default logfile=
C:\\cmsynergy\\bob\\ccm_client_int.log
```

2. Save and exit the file.

Note that when using the `ccm.user.properties` key, you must use the complete path and file name, as in the example above.

Additionally, users must enter Windows paths using double backslashes.

Read the update results after every update/build cycle to look for problems. At the end of the update messages, Rational Synergy writes a summary in the **Messages** dialog box or output log describing the success or failure of an update; however, make a habit of reviewing the logs to read detailed reports of update failures.

Additionally, a successful build does not always mean that the software is configured correctly. Reviewing the update results is a good way to find errors in the configuration: the wrong version of an object or project,

changes that were not merged, incorrect selection property settings, and so on. Here are some things to look for.

- Check for parallel versions.

If you build from the Rational Synergy CLI and you set the `reconfigure_parallel_check` option, if a given set of update candidates contains parallel scoring, you will receive a warning message similar to the following in your Rational Synergy Classic `ccm_ui.log`:

```
Warning: Parallel versions selected by selection
rules, latest create time will be used:
save.c-3
save.c-2.1.1
```

Check the history of the object called out in the warning message to see if parallel versions that should have been merged were not. If so, your build is missing part of a change, and you should notify the developers involved that they need to merge the parallel versions.

For information on how to check for parallel versions when building from Rational Synergy or the Rational Synergy CLI, see [Select parallel notifications during update. \(page 14\)](#)

- Check for replaced subprojects.

Your build management project hierarchy must stay together as a unit. If update properties (release, platform, etc.) are set incorrectly, update could select a different version of a subproject. Check for messages such as:

```
Subproject editor-int_3.0 replaces editor-int_2.1
under toolkit-2:dir:1
```

If you find any messages about replaced projects, investigate the project versions; check their update properties to verify that they are correct.

- Check for empty directory entries.

By default, Rational Synergy leaves a directory entry empty if there are no candidates for it. If you find any, you might need to look for reasons for the occurrence, such as a task with a wrong release value. Look for messages, such as the following:

```
2 directory entries were left empty because they had no
candidates.
```

Empty directory entries are not always an error. For example, directory entries might be left empty if you build products on multiple platforms within a single directory. A shared library might be named `mylibrary.so` on Solaris™ and `mylibrary.dll` on Windows. If you control both

products in the same directory, but use that directory in two parallel projects for the two platforms, you will get an empty directory entry for the Solaris library in the Windows project, and vice versa.

- Check for replaced makefiles.

A developer could customize a makefile with settings specific to his environment, then accidentally check in the custom makefile. If any makefiles are replaced during the update process, review the new versions of the makefiles to ensure that the changes are appropriate for your build environment. Look for messages, such as the following:

```
'makefile-6:makefile:3' replaces 'makefile-5:makefile:3' under 'editor-2:dir:1'
```

- Check for work area conflicts.

If your project has conflicts, you will need to use **Sync Work Area** to resolve work area conflicts, then update again. Look for messages, such as the following:

```
Unable to update membership of project
ccm_client,td_7.1 with
InteractiveProcessCreator.java,21:java:J#1 due to work
area conflicts.
```

- Check for older object versions that replaced newer object versions.

If an older object version replaces a newer one during the update process, perform a show conflicts operation to help pare down possible problems. Look for messages, such as the following:

```
'foo.c-2:csrc:3' replaces 'foo.c-3:csrc:3' under
'toolkit-4:dir:1'
```

Additionally, look at the task that the newer object version is associated with and look at the process rules for the project. Comparing these might tell why the older version replaced the newer version; it should show you some difference in the process rules that is causing the task to add the older object version.

If you see any of these problems and cannot figure out what happened, perform another update with verbose messages set to receive more detailed update results in your output log. (Set **Show verbose messages** in the **Options** dialog box **Actions** tab, in the **Update** option.)

The following sections describe update in greater detail:

- [Updating a project \(page 48\)](#)
- [Working with selection rules \(page 48\)](#)
- [Update with platform values \(page 49\)](#)
- [Diagnosing selection problems \(page 54\)](#)
- [Verifying the update properties \(page 55\)](#)

Diagnosing selection problems

Sometimes you need to figure out why a certain object version was or was not selected during an update. Sometimes an occurrence in the build management project hierarchies makes this necessary; other times you might need to help a developer with a problem in his development project.

Check the following items in the order given:

1. Check the process rules for the project to be sure that:
 - the task queries are correct on the folder template.
 - the correct folders or folder templates are a part of the process rule. (You might need to update the process rule if the folders or folder templates for the project are incorrect.)
 - the baseline is set.
2. Run the update operation, using the `verbose` option. The `verbose` option generates extra information about the candidates that are analyzed. It shows the scores for each candidate, and why they received the scores. Troubleshoot using the given information.

Note You can perform a verbose update on **just** the directory with the questionable object version. This is faster than running the verbose update from the top-level project.

3. Check the project grouping properties.
 - If you turned off **Auto Update**, be sure that you turned it back on.
 - If you removed a task temporarily, check to be sure you added it back.
 - If you manually added a task, be sure you want to keep it.
 - Check that the baseline project is set.
4. Compare process rules to ensure they are set correctly.
 - a. Right-click over the project or project grouping and choose **Process Rule Properties**.
 - b. In the **Process Rule Properties** dialog box, click the object menu (located in the upper left corner) and choose **Compare with Process Rule from Process**.
5. Right-click over the project you are having problems with and choose **Properties**. Verify that the baseline project is appropriate.
6. If you were using a [generic process rule \(page 123\)](#) that uses the **Baseline specified on process rule** setting, then you added it to a release, you must specify a baseline for the process rule. If you do not, the project grouping

using that process rule will not get a baseline and will not update correctly. You can specify a baseline in the **Process Rules Properties** dialog box. (See [Modifying the properties of a process rule](#) in Rational Synergy Help.)

If you have tried the items above, but still have selection problems, try the following:

- Compare process rules for two releases.
If your projects updated correctly in the last release, but they are not updating properly in your new release, compare the process rules between the two releases.
See [Comparing two like objects](#) in Rational Synergy Help.
- Compare the process rule for the current release to the process rule from the process.
- Compare update properties for two projects.
If two projects for the same purpose and release update differently, you can compare their update properties. This is useful for a developer who is setting his update properties manually. He can see if the folders in his update properties differ from those based on templates.
See [Changing the update properties for a project](#) in Rational Synergy Help.
- Compare two folder templates or two folders.
Compare folder templates or folders to ensure your query is correct or to find out which members differ between two folders.
See [Comparing two like objects](#) in Rational Synergy Help.

Verifying the update properties

If you have performed a verbose update and can find no problems in the messages, but your project is not configured correctly, use the following list to check that the update properties are set correctly for your software release:

1. Start the appropriate **Project Grouping Properties** dialog box.
 - Verify that the release is set correctly.
 - Verify that the purpose is correct.
 - Verify that the process rule is set correctly.
 - Verify that the correct [baseline project \(page 121\)](#) is set. (Be sure that the baseline contains a version of the project you are updating.)

This is very important! If you do not have a baseline project and the project you have is not new, your project will be incomplete.

2. Look at the tasks in the project grouping properties. Be sure the tasks you expect are there.
 - Look at the folder properties by double-clicking the folder template. In the **Folder Template Properties** dialog box, click the **Folder Properties** tab.
 - Verify that the release is set correctly on the query for the folder, if it is a query-based folder.
3. If the update properties are correct, look at the folder templates. (Right-click over the project grouping and choose **Process Rules Properties**, then click the **Tasks** tab.)
 - Look at the folder properties by double-clicking the folder template. In the **Folder Template Properties** dialog box, click the **Folder Properties** tab.
 - Verify that the release is set correctly on the query for the folder, if it is a query-based folder.

Alternatively, see [Comparing two like objects](#) to find out information about folders, folder templates, process rules, and project groupings.

How conflict detection operates

A conflict is a potential issue with your configuration. Conflict detection is a way to check that your project contains the configuration you requested. Conflict detection identifies when only part of a change is included in your configuration. It helps ensure that if you include a particular change (as defined by a task), it includes all of the change.

Note Remember, these are **potential** issues. Not every conflict is a bad thing. Whether you want to be notified about certain conflicts will depend on how your software development team works.

For example, after you update your project, then perform a show conflicts operation, you will receive a conflict warning if you have an object that is associated with multiple tasks. If your team often rewrites a program to fix multiple problems in that program, then an object that is associated with multiple tasks will not be a problem to your organization. You can turn off such conflict notifications so that you are informed of only those that you will want to resolve.

Use conflict detection to perform an operation that will tell you if your configuration is missing part of a change or includes an unexpected change.

The next several sections will show you how to find some of these conflicts, what causes these potential issues, how to resolve them, if necessary, and how to tell Rational Synergy what type of conflicts you want to be notified about.

How conflicts arise

A conflict arises when there is a difference between the update properties for the project and the project members. The relationships Rational Synergy uses to detect conflicts include:

- Changes that belong together (because they are associated with a task).
- Changes that include other changes (because they are predecessors).
- Tasks you have specified to be in your project in the update properties.

For example, Rational Synergy detects a conflict when an object is a member of your project, but is not associated with any tasks in the update properties for the project. Alternatively, Rational Synergy detects a conflict when an object is associated with a task specified in your update properties for the project, but the object is not in your project (directly or as a predecessor of another object).

Update your projects immediately before showing the conflicts for a project or projects. If the update properties for a project change after you update it or you manually update its members, conflicts will be shown for discrepancies between the update properties for the project and the project members. Therefore, showing conflicts immediately after updating the update properties for the project minimizes the chance of additional conflicts.

Detecting conflicts

The conflict detection operation shows you the conflicts for a project hierarchy. Additionally, you can perform deep conflict detection, which is a deep analysis of project members that considers tasks and objects that are in or came before the baseline. (See [Performing deep conflict detection](#) in Rational Synergy Help.)

- Right-click over the project for which you want to show conflicts, select **Detect Membership Conflicts**, and choose **Projects and Subprojects**.

A progress indicator appears while Rational Synergy analyzes your project. When the analysis is complete, the **Membership Conflicts** dialog box displays the conflicts for a project. If no conflicts are found, a message displays in the main window status bar indicating that no conflicts were detected.

Now you are ready to resolve conflicts. If you are not sure what the conflicts and dependencies mean, read the next two sections, [Categories of conflicts \(page 58\)](#) and [Conflicts and dependencies \(page 59\)](#). If you already know about conflicts and dependencies, you are ready to resolve the conflicts in your project, which is described in [Resolving conflicts \(page 61\)](#).

Many options in the **Membership Conflicts** dialog box can help you sift through conflicts. For a discussion of those options, see [Resolving membership conflicts in a project or project grouping](#) in Rational Synergy Help.

Categories of conflicts

Two primary categories of conflicts exist. These categories include:

- Changes that are in your project, but not in your update properties.
For example, if you use a new object version without adding its task to your update properties, the object will show as a conflict.

- Changes that are in your update properties, but not in your project.
For example, if the update properties for your project contain two tasks that are associated with parallel versions of the same object, the version that is not a member of your project will show as a conflict.
Because of the association between a task and selected object versions, Rational Synergy shows conflicts as a task or an individual object.

Conflicts and dependencies

In Rational Synergy, an object version does not stand alone. It includes all of the changes for predecessors; each successive version is checked out from the earlier version and is based on the contents of the earlier version.

Dependency relationships is a **key concept** in understanding conflicts. Let's take a look at a dependency relationship: Suppose that **bar.c-1** is associated with task 12, **bar.c-2** is associated with task 25, **bar.c-3** is associated with task 37, and **bar.c-4** is associated with task 48. Therefore, **bar.c-4** contains not just the change from task 48, but also the changes from tasks 37, 25, and 12.

Now let's see how dependencies affect your project's configuration. If your project contains **bar.c-3**, does it contain task-37? Yes. But it also contains tasks 25 and 12 because **bar.c-3** includes its predecessors' changes. What if task 37 is in your project's update properties, but task 25 is not? Then it meets the definition of a conflict: a change that is in your project, but not in your update properties.

So far, the example of **bar.c** is one-dimensional. When you consider that each task can be associated with other object versions, dependencies become much more complex. For example, say task 37 (associated with **bar.c-3**) is also associated with **foo.c-6**. If **bar.c-3** or one of its successors is included in your project, then **foo.c-6** or one of its successors should also be included in your project. Furthermore, it means that the tasks associated with the predecessors of **foo.c-6** are included in your project, so their other associated objects will be included, too. Rational Synergy analyzes all of the history and task relationships to determine which changes are included, and which should be included based on dependencies.

Your project is based on another project, called its [baseline project \(page 121\)](#). The baseline project contains all the changes to earlier versions of its member objects. Rational Synergy needs to look for conflicts in only the differences between the current project and the baseline project. Therefore, conflict analysis looks at each of the project's members only as far back as the version that is in the baseline project.

Conflict terminology

When Rational Synergy shows conflicts in your project, each type of conflict has a name. This section describes the terminology for each type of conflict.

Changes that are directly included in your project are called explicit changes. Changes that are indirectly included are called implicit changes.

Let's build on that. Changes that are in your update properties are known as explicitly specified because you have explicitly specified those tasks to be included in your project. Changes that are not explicitly specified in your update properties but are required because other changes depend on or include them are known as implicitly required.

Changes whose source code is in your project are known as included. If an included change is not explicitly specified, it is implicitly included. Implicitly included changes are included in your project because other changes in your project depend on or include them.

Conflict message definitions

The conflict message definitions are the conflict messages you might see when you run conflict detection. Note that some display by default, while others do not. The messages are described in Rational Synergy Help, in [Resolving membership conflicts in a project or project grouping](#).

If your team requires different default display settings, users in the `ccm_admin` role can change them. This is described in [conflict parameters](#) in Rational Synergy CLI Help.

Resolving conflicts

When you are ready to resolve conflicts, keep the following in mind:

- Update to bring project members in sync with the update properties. If you do not understand why an object version was selected, perform a verbose update.
- Gather information about the conflict:
 - Look at the project to see which version of the object is being used.
 - Look at the object history to see the relationship between the object in conflict and the object that is being used.
 - Look at which tasks are associated with the objects you are interested in.
- For each implicitly included or required object:
 - Consider whether its task should be added to the update properties for the project. If it should be, find out why it is not being included already: Is the release value for the task incorrect?
 - If this task for the object should not be added to the update properties for the project, look at its successors, and consider whether their tasks should be removed from the update properties for the project. If they should be, find out why they were included: Are the release values set incorrectly?
- If you update the update properties for the project or task release values, remember to update again.
- For each parallel version, consider whether it needs to be merged. (If so, you can create and assign a new task.)
- After resolving as many conflicts as possible, run conflict detection again. Often, clearing up one conflict resolves many others, since a conflict can have a cascading effect because of dependency relationships.

Remember that each team has its own unique process and method for keeping track of changes. This will impact what your team considers to be a conflict. One team might consider a specific conflict as part of their methodology, and that team would turn off that particular conflict. Another team might view that same conflict as a problem to be corrected immediately. Be sure that your team agrees on the kind of conflicts that should be addressed as early in your development process as possible.

5

How baselines work

Working with baselines

A baseline is a set of projects and tasks used to represent your data at a specific point in time. A baseline has many uses. When you perform an update, Rational Synergy uses a baseline as a starting point to look for new changes. You can also compare two baselines to see what changes have been made relative to a particular build. If you use Rational Change, you can use baselines to generate change request reports.

Typically, a build manager creates a baseline; a developer does not need to create a baseline because he does not make his builds available to other users.

You might find it useful to create a baseline as soon as you perform a build. You can create a baseline and make it available to the test group without making it available to all developers. This is called a test baseline. Making a test baseline as soon as you build saves a representation of the build in Rational Synergy in case it is needed later to create a fix for that particular build.

Creating a baseline for each **Integration Testing** and **System Testing** build enables testers and developers to refer back to the set of changes that were used to create the build. Typically, you'll create a baseline for all projects in the same release and purpose. For example, you would create a baseline for each **Integration Testing** build using all **Integration Testing** projects for that release.

Baselines also improve performance for update operations. An update that uses baselines only needs to analyze the tasks that were added since the last baseline, rather than all tasks for the entire release.

Using a baseline

When you create a baseline, you choose a list of projects to be included in the baseline. Be sure to include **all** related projects in your baseline so that you have a complete set for reference.

Any projects that are already in a static state are included without further change. For each project that is not static (for example, a build management project), the following actions occur when the baseline is created:

- A new version of the project is copied. This version has no work area, which makes the operation as fast as possible.
- If multiple projects are copied within a hierarchy, they are used in a single hierarchy that is a copy of the original hierarchy.
- The project history is updated to show the baseline inserted **before** the build management project that it was copied from.
- The new projects are checked into a static state.

The original build management projects and their work areas are left unchanged. The benefit is that they will continue to be rebuilt incrementally. If the build management projects were checked in and new prep versions checked out from them, they would be entirely rebuilt because uncontrolled intermediate products would not be in the newly checked out projects' work areas.

After creating a baseline, you can make work areas available for other users by turning on work area maintenance for selected projects in the baseline project, which will write it out to the work area. Typically, you will want to do this for absolute subprojects if developers reuse the static subprojects rather than checking out their own versions. The following commands are examples of how to query the database, then turn on work areas:

```
ccm query "is_project_in_baseline_of
(baseline('20070203')) and name match '*_ext_x' and
platform='UNIX'"
ccm wa -wa @
```

Considering how to use a baseline

Synergy uses a baseline as a snapshot of projects and tasks at a particular point. Before you create a baseline, you will need to consider how it will be used. The update operation uses the baseline as a way of saying “start from here.” Therefore, if you use the standard process rules to set up the baseline to contain projects from multiple components or not to contain all the projects for a release, then Synergy will not be able to use your baseline to update properly. This methodology is supported, but you must customize process rules for it to work. The next two examples show how to set up a baseline correctly, while the last example shows a baseline that was set up incorrectly. Using baselines correctly can improve performance when you update your projects.

The baseline is selected by its release; therefore, it is important to have projects with a consistent release in the baseline. The projects in the baseline are used as baseline projects for the projects in a project grouping. In the following table, notice that every project in the project grouping maps to a project in the baseline. This example shows a complete baseline.

Baseline for CM/7.0 build 1234	← Baseline	Project Grouping for CM/7.1 Integration Testing
cm_top-CM/7.0	← Baseline project	cm_top-CM/7.1
cm_gui-CM/7.0	← Baseline project	cm_gui-CM/7.1
cm_api-CM/7.0	← Baseline project	cm_api-CM/7.1
cm_platform-CM/7.0	← Baseline project	cm_platform-CM/7.1

In the following table, notice that two separate baselines are needed for a hierarchy with mixed components. This example shows correct baselines.

Baseline for CM/7.0 build 1234	← Baseline	Project Grouping for CM/7.0 Integration Testing
cm_top-CM/7.0	← Baseline project	cm_top-CM/7.1
cm_gui-CM/7.0	← Baseline project	cm_gui-CM/7.1
cm_api-CM/7.0	← Baseline project	cm_api-CM/7.1
cm_platform-CM/7.0	← Baseline project	cm_platform-CM/7.2
Baseline for TC/5.1 build 5678	← Baseline	Project Grouping for TC 5.2 Integration Testing
change_api-TC/5.1	← Baseline project	change_api-TC/5.2

If you create a single baseline for projects with mixed components, the projects will not be able to find a baseline project correctly if you are using standard process rules. In the following example, notice that a subproject from the TC/5.1 release (in *Italics* in the table below) is included in a baseline for the CM/7.0 release.

Synergy will not be able to use this project as a baseline project because the baseline it is a member of (for example, **Baseline for CM/7.0 build 1234**) will not be selected by any project grouping (for example, **Project Grouping for TC 5.2 Integration Testing**) that would use this as a baseline project. It will also result in tasks for both releases being included in the baseline.

If all of the tasks in the baseline are not used by all of the projects in the baseline, the tasks will not be subtracted by the project grouping during the update. Optimally, new project members selected by the update operation are gathered in this way:

- Synergy gathers all tasks specified by your process rules.
- Synergy subtracts all tasks from your baseline.

If you have a task in your baseline that was not used by one or more projects in your baseline, Synergy realizes this and will not subtract it when calculating tasks to use for update.

In the following table, **change_api-TC/5.1** in the first column is from another component and should not be in the CM/7.0 baseline. Because it is there, all the tasks associated with it are also part of the baseline and none of the tasks from either CM/7.0 or TC/5.1 are used by all of the projects in the baseline; therefore, none of the tasks are subtracted. This will hamper performance.

- Update selects candidates for each member of a project based on the corresponding object in the baseline project and the tasks in the project grouping properties. The candidates considered are the object in the baseline project plus any new versions of that object associated with the tasks collected, as explained above.

In the following table, **missing** in the first column shows that a baseline for **Project Grouping for TC 5.2 Integration Testing** cannot be found. Because there is no baseline, there will not be a baseline project for **change_api-TC/5.1**. This means that you will not get the members that you expect when you update the project.

This example shows a baseline that will not work with standard process rules.

Baseline for CM/7.0 build 1234	← Baseline	Project Grouping for CM/7.1
cm_top-CM/7.0	← Baseline project	cm_top-CM/7.1
cm_gui-CM/7.0	← Baseline project	cm_gui-CM/7.1
cm_api-CM/7.0	← Baseline project	cm_api-CM/7.1
cm_platform-CM/7.0	← Baseline project	cm_platform-CM/7.1
<i>change_api-TC/5.1</i>		
missing	← Baseline	Project Grouping for TC 5.2 Integration Testing
none	← Baseline project	change_api-TC/5.2

Creating a baseline

At this point, you have typically updated the integration or system testing projects, dealt with conflicts, and built your products. You are now ready to create a baseline to save a copy of this build for future reference. When you create the baseline, you can choose to publish it and make changes available to developers immediately or wait to publish the baseline until the build has had more testing.

1. Right-click over a project hierarchy or project grouping and select **Create Baseline**.

You can create a baseline from a build management project grouping or from a project that is static or is a build management project.

The **Create Baseline** dialog box appears.

Note When you create a baseline, you'll choose a list of projects to be included in the baseline. Be sure to include **all** related projects in your baseline so that you have a complete set for reference.

2. Modify the properties of the baseline, as required.

- a. Type a name in the **Name** field.

This is the name of a baseline. It uniquely identifies this baseline within this database. By default, Rational Synergy names baselines with the creation date; for example, 20090309 stands for March 9 2009.

However, you can modify it. The following are restricted characters and should not be used in the name: / \ ' " : ? * [] @ - #

Note that if you are creating baselines in more than one database and you are using Rational Change to generate reports on these builds, use the same baseline name in each of the databases. This allows you to generate a build report that has a related baseline in more than one database.

- b. Confirm that your release is correct.

A release is a property that identifies a baseline that is specific to a particular release.

- c. Confirm that your purpose is correct.

A project purpose specifies which projects were used to create it, for example, **Integration Testing**. The process rules use the purpose of the baseline to select the appropriate baseline during an update operation.

- d. Optionally, you can set the build identifier by typing it in the **Build** field.

The build property shows the build identifier (letters, numbers, or combination) associated with a baseline. The build identifier can contain a maximum of 64 characters.

Note that if you are creating baselines in more than one database and you are using Rational Change to generate reports on these builds, use the same baseline build identifier in each of the databases. This allows you to generate a build report that has a related baseline in more than one database.

- e. Type a description in the **Description** field to describe the baseline you are creating.

- 3. Change the projects included in the baseline.

If the **Always include all subprojects regardless of release** option is set in the **Create Baseline** action in the **Options** dialog box, then all projects for a project hierarchy, regardless of the release value, will be available. If this option is not set, all *prep* subprojects are used. Only the static subprojects with a component that matches that of the top-level project are used.

See [Changing baseline creation options](#) in Rational Synergy Help for details on setting this option.

- a. To add individual projects, click the **Add Project** button.

The **Select Project** dialog box appears, which works the same as the **Query** dialog box. By default, projects display in the selection set field based on the release and purpose for the project, for example, **toolkit/2.0** and **System Testing**. Additionally, you can define a query to find the project that you want to add to the baseline.

- b. To add all projects from a project grouping, click the **Add Project Grouping** button.

The **Select Project Grouping** dialog box appears, which works the same as the **Query** dialog box. Adding a project grouping enables you to add projects associated with a project grouping, which is helpful if you are creating an [incremental baseline \(page 124\)](#). Additionally, you can define a query to find the project grouping that you want to add to the baseline.

- c. To add all projects from an existing baseline, click the **Add Baseline** button.

The **Select Baseline** dialog box appears, which works the same as the **Query** dialog box. Adding a baseline enables you to add the projects associated with a baseline, which is helpful if you are creating an [incremental baseline \(page 124\)](#). Additionally, you can define a query to find the baseline that you want to add.

- d. Click **OK**.

4. Publish the baseline to developers upon creation.

If you are creating a [test baseline \(page 127\)](#) for limited availability, skip this step and proceed to [step 5](#).

This option is cleared by default, but you'll want to set it. As soon as you publish the baseline, developers will be able to update their projects to bring in the most recent changes from the baseline.

5. Create the baseline by clicking **OK**.

Publishing a baseline to developers

You have now updated the system testing projects, dealt with conflicts, built your products, and tested the application. You are ready to make the changes available to others. You will do this by publishing the test baseline.

As soon as you publish the baseline, it will be available as a baseline for selection during an update. The process rules ensure that projects will use the latest baseline.

1. Select the test baseline(s) that you want to transition to system testing, right-click over the baseline(s), and choose **Publish**.
2. Create work areas for your baseline projects, if necessary.

If you are creating an external project, for example, for product sharing, be sure to create a work area that is visible to everyone. For a discussion of baseline projects, see [Using a baseline \(page 64\)](#), bottom paragraph.

- a. Right-click over the system testing project(s) whose work area settings you want to modify, and choose **Properties**.

The **Properties** dialog box appears.

- b. Click the **Work Area** tab.
- c. Select the **On/Off** option.

Selecting this option causes Rational Synergy to maintain a copy of the project in your work area and the project to sync automatically when you click the **Apply** button.

- d. Click **Apply** to save the changes.

3. Inform developers that they can update their projects to bring in changes.

You have now published a baseline for the system testing projects. The published baseline makes the build available to others. As soon as users update their projects, they can start using the new baseline.

Baselines and the update process

When a project is updated based on process rules, update re-evaluates which [baseline project \(page 121\)](#) it should use. It checks the process rule to find the baseline, then checks the baseline to identify a version of the project. If multiple versions of that project exist in the baseline, update compares the platform values to select the project that has a matching or compatible platform value.

For example, when the **editor-bob** project is updated, update checks the process rule and finds that the latest baseline is selected. It then identifies the latest baseline matching the template criteria, for example, **20070115**. It then checks baseline **20070115** for a version of the **editor** project to use as a baseline project.

If no version of a project is found in the baseline, that project is updated without a baseline project. This means that object versions that are not associated with any tasks for the current release are not considered candidates, and are not selected when you update the project. Some directory entries might be left empty in this case.

Therefore, it is important to include all projects in your application in the baseline, even those that have not changed since the last baseline. (Static projects can be reused in multiple baselines.)

Creating an incremental baseline

If it is not possible or practical for you to make a copy of a project in the baseline for build management, you can use the incremental baseline approach.

For example, if you have projects **proj1** through **proj100**, and you want to create a new baseline with just **proj1-int** and **proj2-int** (for example, because **proj3** through **proj100** haven't changed), then you can create your new baseline from your most recent baseline, plus these two projects.

1. Select the two projects you want to include in the incremental baseline, right-click over one, and select **Create Baseline**.

You can create a baseline from a build management project grouping or from a project that is static or is a build management project.

The **Create Baseline** dialog box appears.

2. Modify the properties of the baseline, as required.

- a. Type a name in the **Name** field.

This is the name of the baseline. It uniquely identifies this baseline within this database. By default, Rational Synergy names baselines with the creation date; for example, 20070309 stands for March 9 2007. However, you can modify it. The following are restricted characters and should not be used in the name: / \ ' " : ? * [] @ - #

Note that if you are creating baselines in more than one database and you are using Rational Change to generate reports on these builds, use the same baseline name in each of the databases. This allows you to generate a build report that has a related baseline in more than one database.

- b. Confirm that your release is correct.

A release is a property that identifies a baseline that is specific to a particular release.

- c. Confirm that your purpose is correct.

A project purpose defines what it is used for, for example, **Integration Testing**. When you change your baseline purpose, Rational Synergy uses different selection criteria when you update the project or project grouping.

- d. Optionally, you can set the build identifier by typing it in the **Build** field.

The build property shows the build identifier (letters, numbers, or combination) associated with a baseline. The build identifier can contain a maximum of 64 characters.

Note that if you are creating baselines in more than one database and you are using Rational Change to generate reports on these builds, use the same baseline build identifier in each of the databases. This allows you to generate a build report that has a related baseline in more than one database.

- e. Type a description in the **Description** field to describe the baseline you are creating.

3. Change the projects included in the baseline.

Add the most recent baseline by clicking the **Add Baseline** button.

You do not have to remove the older versions of the build management projects you just added, but doing so will give you a more accurate representation of the members of your baseline.

The **Select Baseline** dialog box appears, which works the same as the **Query** dialog box. By default, projects display in the **Included Projects** field based on the release and purpose for a project, for example, **toolkit/2.0** and **System Testing**. Additionally, you can define a query to find the project that you want to add to the baseline.

4. Publish the baseline to developers upon creation.

This option is cleared by default, but you'll need to set it. As soon as you create the baseline, developers will be able to update their projects to bring in the most recent changes that have passed integration testing.

5. Create the baseline by clicking **OK**.

Removing unnecessary baselines


To remove an unnecessary baseline, first mark it for deletion. Once marked, you can set up the Save Offline and Delete command to automatically delete baselines that are marked for deletion when they are no longer used.

The following steps show you how to mark a baseline for deletion. (The [soad Command](#), described in Rational Synergy CLI Help, enables you or the CM administrator to delete the baselines you have marked.)

1. Find the baselines that you want to remove.

Find > Baselines

The **Query** dialog box appears.

2. Set the query criteria and then run the query.
 - a. Set **For Release** to the appropriate release.
 - b. Click the plus sign to add another property to the query.
 - c. Set **With Purpose** to **Integration Testing**.
 - d. Run the query by clicking .
3. Mark the unnecessary baseline for deletion.

Choose all baselines that you want deleted, then right-click and select **Delete**.

The **Delete** dialog box displays the selected baselines. Click **Delete**. The selected baselines are now marked for deletion.

Note that an update operation will not select a baseline that is marked for deletion. If your team uses project groupings, and any of the project groupings are using marked baselines, they will choose different baselines during the update operation.

The CM administrator can set up the Save Offline and Delete command to automatically delete baselines that are marked for deletion when they are no longer used. For details, see the [soad Command](#) in Rational Synergy CLI Help.

Additionally at this point, your site might decide to inactivate the release you have just finished. Use the [ccm clean up command](#), described in Rational Synergy CLI Help, to clean out the process rules and any old releases from the release.

There is not much overhead associated with retaining these extra releases and process rules; clean up is for convenience. Cleaning out the releases and the

process rules makes it easier to find the information you need in the corresponding dialog boxes.

Additionally, you cannot delete a baseline if a project grouping uses that baseline, or if a process rule uses the baseline. If you try to delete a baseline and its checked-in projects and products, and one or more of its associated projects or products is a member of a project that is not part of the baseline, the delete baseline operation will succeed; however, those projects or products will not be deleted. Furthermore, if one or more projects that are in the baseline are members of another baseline, or are baseline projects, the delete operation will be successful, but those projects will not be deleted.

Note that you will need to remove obsolete project hierarchies, and then empty project groupings, before you can delete old baselines. However, empty project groupings may still carry important information about tasks the owner explicitly added to or removed from that grouping.

To find and remove empty project groupings that also have no explicitly added or removed tasks, use the following commands:

```
ccm set role ccm_admin
ccm query -t project_grouping
"is_no_project_grouping() and
has_no_added_task_in_pg() and
has_no_removed_task_in_pg()"
ccm delete @
```

6

How to share products

Product sharing is the practice of letting developers share products that were built by the build manager. For example, Joe is working on the **toolkit.exe** executable. He has a *working* version of **toolkit.exe** because he is making changes to it. In the office next door, Mary is making changes to the **guilib.lib** library. Although the **toolkit.exe** executable is linking with the **guilib.lib** library, Joe is not making changes to **guilib.lib** and does not need a working version of it.

Joe can use the version of **guilib.lib** built by the build manager. After the integration testing project is built and passes testing, the build manager will create a baseline, which checks in copies of the *prep* products to make them available to developers. When Joe is ready to get Mary's latest changes to **guilib.lib**, he will update his development project to bring in the latest tested product file.

Why is this something your site might want to use?

This process cuts the amount of extra work developers need to complete because they do not need to build products that they do not modify.

You can share products by packaging them into external projects ([Sharing external projects \(page 78\)](#)) that can be shared.

Sharing external projects

The following section describes one way to share external projects. Although there are many ways in which to share external projects, the following is how Rational Synergy developers do this.

External projects save time for developers because a developer does not need to copy and update the projects that he does not change. External projects also help to promote modular code and information hiding, both considered best practices for software development.

However, the build manager must do some extra work to manage external projects, including restructuring projects, updating makefiles, and updating build scripts that automate the build process. Also, developers who work on both the lower-level project (e.g., the library) and the project that depends on it (e.g., the executable) must manage their own *working* versions of the external project and both source projects.

If you want to use external projects, read the following sections:

- [Preparing to use external projects \(page 79\)](#)
- [Creating an external project \(page 80\)](#)
- [Modifying the build process for multi-phased builds \(page 82\)](#)

Preparing to use external projects

External projects make products and associated objects developed in one project (such as a library and the header files for the functions in it) available to another project. Normally you would add the project as a subproject so that you could get the header file. An external project is one way to set up your project structure to minimize the number and size of development projects needed by developers.

For example, Joe's *working* version of his **toolkit.exe** executable is linked with the **guilib.lib** library. Joe is not making changes to **guilib.lib** and does not need a working version of it, but his executable is linked to the library, so he needs it to do his work. When Joe is ready to get the latest changes to **guilib.lib**, he will ensure that the **guilib** project is a member of his development project, then update it to bring in the newly built product file.

Based on this example, imagine if Joe was using products from ten or fifteen different projects, none of which needed to change. He would need development projects for each project and this could mean extra time when he updates to bring in the objects he needs. An external project can remedy this situation.

An external project contains **only** the products and header files needed to use those products. (It is called an "external project" because it makes those files available to other projects, but it is external to the project where those files are actually developed.)

Because external projects are counterparts to other projects, it is useful to give them similar names. For example, the external project corresponding to the **guilib** project might be named **guilib_ext**. (It is also useful to keep the same structure so that makefiles can be modified easily to reference the external project rather than the original project.) Note that adding a subproject is not necessary with some programs, such as Java, where you do not need the header files and can add the library alone.

Now the build manager can create a baseline to check in the external projects and the products after they pass integration testing, and developers whose code references those products can share the external projects. They do not need working versions of any projects that they do not personally need to modify.

Developers will share the products most recently checked in by the build manager when they update. If your site does not use baselines, then you should check in the products and external tasks after they pass integration testing. If you do use external projects, you will need to create work areas that are visible to everyone for the external baseline projects.

Note The use of external projects is optional. Consider the product sharing needs of your team.

Creating an external project

Be sure that you already have build management project hierarchies. Creating an integration testing project is described in [Creating the integration testing projects \(page 26\)](#). Creating a system testing project is described in [Creating the system testing projects \(page 28\)](#).

1. Create a task to create an external project, leave yourself as the resolver.
The newly created task is set as the current task.
2. Copy a project to be the external project.
 - a. Create the new project.
 - b. Set the properties for the new project.
 - c. Click **OK**.
3. Add the product object(s) (and any other objects) to your external project by using drag and drop or copy and paste.
4. Add the new external project to your original project.
 - a. View the appropriate build management project in the **Work** pane.
 - b. Cut the original subprojects from your original project. (You will need to do this if you are creating an external project from an existing subproject.)

If you have a **guilib** project, and you created a **guilib_ext** external project, you will need to replace each occurrence of the **guilib** project with **guilib_ext**. You will need to do this for each external project.

For example, Joe would add **guilib_ext** to his **toolkit** project, and remove the **guilib** project from his **toolkit** project. Moreover, he could delete his copy of the **guilib** project if he never needed to build that subproject.
 - c. Add your new external project to your original project. (If the new external project is a subproject, you can skip this step.)

Refer to [step 3](#) above.
5. Repeat [step 4](#) for each project that should include the external project.

Note Modify your build scripts and makefiles to properly reference the new external project(s).
6. Complete the current task.
7. Perform an integration test cycle on your integration testing projects.

8. Create a baseline from the project hierarchy.
9. Copy a system test build management project from the external baseline project.
10. Perform a system testing build and test cycle on your system testing project.

Modifying the build process for multi-phased builds

The regular build process is described in [Build management basics \(page 29\)](#).

The modified build process for external projects is a multi-phase build. Project hierarchies that contain external projects cannot be completely updated right away because the new products you want to select may not exist until the hierarchy is partially built. (If the products are in a non-writable state before the build, new versions will be checked out. Those new versions need to be selected into the external project after the build is complete.) Portions of the hierarchy need to be updated and built in phases.

To prevent selecting new tasks, clear auto-update on the project grouping after the first update, then set auto-update in the middle of this multi-phase build on the project grouping after the last update. The following example shows you how to do this:

1. Refresh and freeze your project grouping.
 - a. Update the query folder by refreshing the project grouping.

Right-click over the project grouping and point to **Update**, then select **Baseline & Tasks**.
 - b. Disable auto-update for the project grouping.

Right-click over the project grouping and clear **Automatically Update Baseline & Tasks**.
2. Update all low-level projects (such as library projects) whose products will be used in external projects.
3. Show and resolve conflicts.
4. Build all products to be used in external projects (i.e., all the projects updated in [step 2](#)).
5. Update all projects not updated in [step 2](#), including external projects.

If you are building installation projects, exclude those now. You will update them later in [Modifying the build process for installation projects \(page 88\)](#).

Note All the products that were built in [step 4](#) should be selected in the external projects by this update.
6. Show and resolve conflicts.
7. Build all remaining products (i.e., the projects updated in [step 5](#)).
8. Prepare an installation area or a CD.
9. Test the software.

10. Create a baseline to check in products and external projects.
11. After the last update, right-click over the project grouping and set **Automatically Update Baseline & Tasks**.

First the low-level projects, such as libraries, are updated and built. Next the external projects and the high-level projects that use them, such as executables, are updated, then the high-level projects can be built.



7

How to package an application

Application packaging is a way to release your application to any customer, whether they are internal customers (e.g., SQE) or external customers (e.g., CIA).

Each site packages their application according to what makes the most sense for their software. Some typical ways to deliver software include:

- On DVD
- On CD-ROM
- On your company Web site

A company will usually opt to deliver its product on CD-ROM or from the company Web site. One way to set up the data to be packaged is to create an installation area from an installation project.

About installation areas and projects

If you control your product files in Rational Synergy, you can create a project that has the structure of an installation area, called an **installation project**.

You can create an installation project whose structure matches that of your installation area. You then can use the work area for the installation project to create an installation image (Windows) or a set of tar files (UNIX) and place it on a CD for testing or for the release of your software.

For example, if you needed to release software internally for testing, you would start with an integration testing project, then create an installation project with all of the products and objects that need to be available to testers. Last, you would add the installation project to the integration testing project hierarchy. When development is at a point where the team needs testing performed by an SQE group, you would build your system testing project, update your installation project, then create an installation image (Windows) or a set of tar files (UNIX) and place it on a CD.

The next section, [Creating an installation project \(page 87\)](#), shows you how to create an installation project.

Creating an installation project

Be sure that you already have the appropriate build management project hierarchies. Creating an integration testing project is described in [Creating the integration testing projects \(page 26\)](#). Creating a system testing project is described in [Creating the system testing projects \(page 28\)](#).

Note The use of installation projects is optional. Consider the needs of your team before creating them.

1. Create a task to create an installation project, and leave yourself as the resolver.
The newly created task is set as the current task.
2. Copy a project to be the installation project.
 - a. Create the new project.
 - b. Set the properties for the new project.
 - c. Click **OK**.
3. Create directories in the new project to match your installation area.
4. Add the product objects (and any other objects that are part of your deliverable) to your installation project using drag and drop or copy and paste.
5. View the appropriate build management project in the **Work** pane.
6. Add your new installation project to the build management project hierarchy.
Refer to [step 4](#) above.
7. Extend your build scripts and makefiles to construct the installation image.
8. Complete the current task.
9. Perform an integration test cycle on your integration testing projects.
10. Copy a system testing project from the baseline installation project.
11. Perform a system testing test cycle on your system testing project.

Note Complete this process for each platform, checking out integration and system testing project for each platform.

Modifying the build process for installation projects

The regular build process is described in [Build management basics \(page 29\)](#).

This is another phase in the multi-phase build approach. Project hierarchies that contain installation projects cannot be completely updated right away because the new products you want to select may not exist until the hierarchy is partially built. (If the products are in a non-writable state before the build, new versions will be checked out. Those new versions need to be selected into the installation project after the build is complete.) Portions of the hierarchy need to be updated and built in phases.

To prevent selecting new tasks, clear auto-refresh on the project grouping after the first update, then set auto-refresh again on the project grouping after the last update.

1. Update each installation project to select the newly built products.
2. Show and resolve conflicts.
3. Prepare an installation area or a CD from the work area of each installation project.

First the projects that produce products are updated and built. Then the installation projects are updated to get the products that were built.

8

Parallel releases

The following parallel releases are discussed in this chapter:

- [Creating a patch for a release \(page 90\)](#)
- [Using a parallel development environment \(page 93\)](#)

Creating a patch for a release

Rational defines a patch to be any release in which you rebuild portions of your application with one or more fixes.

To create a patch, you will need to be able to reproduce and rebuild the released software version. This underlines the importance of creating baselines for the projects used to build your application. Baselines create a snapshot of what was built. When you need to reproduce or rebuild that released project, as you will in the case of a patch, it will be available for your use.

Setting the patch release

A patch release can be small (a single patch) or large (a service pack including several patches). Regardless of size, you will need to set a release for the patch.

A single patch might have a name like **toolkit/3.0patch**, while a service pack-size release might have a name like **toolkit/2.0sp1**.

Identify which purposes will be used for the patch release. Typically you will need only **Collaborative Development** and **System Testing**.

Including projects

When you create a patch, you need to include new versions of the projects to be rebuilt. The following paragraphs describe what to include in your patch project:

- If a developer is fixing a defect in an executable, the developer would change the code in the project that rebuilds the executable; therefore, your patch would include only a version of the projects that contain changed code used to build the executable.
- If the defect was in a library, the developer would change the code, you would rebuild the library, then rebuild every executable that links with it. Therefore, you would need projects for the library and all of the executables that link with it, and external projects, if you use them.

Often, the developer assigned to fix the bug can tell you which projects need to be rebuilt.

Note You can set up [grouping project \(page 124\)](#)s to organize your patches in the Rational Synergy database. See [Grouping projects \(page 106\)](#) for more information.

If you are using the installation image to build your patch, you will need the installation project.

Obtaining fixes from developers

You can obtain fixes from developers in the following way:

1. The developer creates the patch projects, fixes the bug, unit-tests the fix, then completes the task.
2. The build manager then copies those projects. He updates his copies to get the developer's tasks.

Creating a release for a patch

The following process shows you high-level steps for creating a release for a patch. Many of the low-level operations you need to perform are discussed in earlier sections of this book, and you can use the links provided here or your own bookmarks to go to those sections.

This process assumes that you have set release values **consistently** on all of your projects. If you have not, this process will **not** work.

1. Create a release for the patch. (For steps, see [Creating or copying a release](#) in the Rational Synergy Help.)

Typically, you will copy the release you are patching.

2. Set the process rule you will use for the patch in the **Create Release** dialog box.

Typically, you will need the **Collaborative Development** and **System Testing** process rules.

Creating a patch

The following process shows you high-level steps for creating a patch.

1. Create a task (or several tasks) and assign it to the appropriate developer(s) for the patch release.
2. Give the project information and new release values to the developers.

Developers will need to copy a new development project from the baseline release, set their current task, diagnose and fix the problem, unit test, then complete the current task.

Note Developers who are fixing problems must set the release value on their projects **before** checking out files. They will also need to set the current task so that objects they check out will be associated with the correct task automatically. If they forget these steps, they will have to perform several manual steps to ensure that the patch includes the correct objects.

Also, developers should **not** complete any other work in the patch projects. This is so that other checked out versions are not tagged with the patch release value and included in the patch accidentally.

3. Create a system testing project for each project being patched.

You can follow the same steps to create the system testing project **with the following exception:**

Copy only the projects to be included in the patch.

See [Creating the system testing projects \(page 28\)](#).

Note You do not need an integration *prep* patch project.

4. See [Building with specific tasks \(page 41\)](#).

5. Set up a test area.

To set up a test area for the patch, you can install a copy of the released software to an installation area (e.g., **patch_test_1.2**), then copy the products built for the patch into the area. (Ideally, you should install the patch the same way your customers will install it.)

If you have a special utility for installing patches, use it instead.

6. System testing proceeds; if defects are found, start the cycle again at [step 1](#) (however, do not repeat [step 2](#) and [step 3](#)).

7. When the patch passes your quality standards, make it available to customers.

Using a parallel development environment

A parallel development environment occurs when your company decides to ship a product on more than one platform (e.g., one for Windows and one for UNIX) or when your company needs to ship more than one release of a product (e.g., a main product release and a patch release).

For a discussion on the parallel platform environment, see [Building on parallel platforms \(page 93\)](#); for information on how to set up this environment, see [Setting up a parallel platform \(page 94\)](#). For a discussion on the parallel release environment, see [Setting up parallel releases \(page 95\)](#).

Building on parallel platforms

If you need to build your software for different platforms, use the platform property to create a version of each project for each platform. These projects are called variant projects. The variant projects share most of the same source members, but you can set different build arguments and save different resultant products in each variant project. However, it is not necessary to mark individual tasks for specific platforms, or to set up folders by platform. A single task can contain files changed for all platforms. When parallel versions occur, each project will select the object versions matching its platform.

For example, to build the **toolkit** project for Windows and HP-UX®, you would copy two different versions of the project hierarchies, but set each of their **platform** properties to the appropriate value. (Note that you must already have set the platform values in the **om_hosts.cfg** file, which is discussed in [About the platform file \(page 17\)](#).)

You can give these projects meaningful versions, such as **sp1_win32_2.0** or **hp_2.0**. This is a great way to name your projects to identify them at a glance.

Caution If you have added a platform attribute to an object, be careful before removing it from future versions. For example, in release 1 of your product you have two parallel versions of a file, version `win_1` with the platform value `x86` and version `sol_1` with platform value `sparc`. In release 2, you decide to merge these two parallel files to form the cross-platform version 2, and you clear the platform attribute on version 2. Because Rational Synergy prefers matching platform values, a project with platform value `x86` will still pick up the version `win_1`, and not your merged version 2. To fix this, you could remove the platform attributes from the old `win_1` and `sol_1` versions, but then you might be unable to build patches to that older release. A better fix would be to

change the name of the merged object, so the older versions would no longer be candidates.

Products are also platform-specific. You will need to check out parallel branches of each product for each platform, then set the platform values appropriately.

Note Users could build the same product for different platforms by using the same project and changing the **platform** property, make macros, and work area for that platform before building.

This is **not** a good way for users to perform builds.

Build managers need to be able to reproduce the products they build. If you are constantly changing the configuration back and forth to build different platforms, you will not be able to see how the product was built. This will make it very hard for you to track down problems, test fixes, or preserve the software when it reaches a milestone.

Also, this method requires you to force a rebuild every time you change the platform.

For more information on how update for parallel platform works, see [Working with selection rules \(page 48\)](#).

Setting up a parallel platform

This operation shows how to copy a new project hierarchy for a parallel platform, from an existing project hierarchy.

1. Set up the new platform values you will need in the **om_hosts.cfg** file.
This operation is described in [About the platform file \(page 17\)](#).
2. Copy a new project hierarchy from the existing project hierarchy.
Remember to set the platform value and to give the project version a meaningful name. Ensure that the version and platform are used in the work area path to make work areas for parallel projects distinct.
3. Verify that your new project hierarchy builds successfully.
You may need to change the makefiles and project macros.
4. Baseline your new project hierarchy.
This operation is described in [Publishing a baseline to developers \(page 71\)](#).

Now you can copy the build management projects from this baseline.

Setting up parallel releases

Sometimes companies develop parallel releases of an application concurrently. For example, one team might be working on new features for release **toolkit/3.0** of the **toolkit** application while another team works on bug fixes for release **toolkit/2.1**.

Since you will need to build your application for more than one release, you will need to create a different project version for each developing release.

For example, your team is working on a new feature release called **toolkit/3.0**, and is concurrently working on a bug fix release called **toolkit/2.1**; you want the **toolkit/3.0** release to include the bug fixes from release **toolkit/2.1** but you do not want the **toolkit/2.1** release to include the features from **toolkit/3.0**.

In this case, you would modify the process rule for release **toolkit/3.0** by adding the **All Completed Tasks for Release toolkit/2.1** folder (not the folder template) to the **toolkit/3.0** integration testing process rule so that the **toolkit/3.0** integration testing projects pick up tasks from both releases. You would need to make similar changes for the other process rules in the **toolkit/3.0** release.

Note that this does not remove the need for developers to merge parallel changes. If Joe makes a change to a file in **toolkit/3.0**, then Mary makes a change to the same file for **toolkit/2.1**, those changes will be parallel. (Mary's change will be chosen for **toolkit/3.0** because it is newer.) The two versions must be merged into a new version for **toolkit/3.0**.



9

Project restructuring

Project restructuring is the act of rearranging your integration or system testing project members by turning existing directories into projects, or adding or removing projects from the hierarchy.

Many reasons exist for why sites decide to restructure projects; the following are a few:

- The direction of your product has changed and you need to remove subprojects from the hierarchy.
- A project has grown too large, and you want to split it into smaller parts.
- Your team added lots of new functionality to your product, and you need to add subprojects to the hierarchy.
- A different team now is responsible for part of the software, and you want to move it into a separate project.
- Your team decided to wait and include an invasive, disruptive change to your product in the next release, and you need to unuse a subproject from the hierarchy.
- You want to add an external project.
- You want to add an installation project.

Whenever you restructure a project, you will need to change the makefiles, the build process, and all automated jobs to reflect the changes you have made.

You will need to apply the changes to both the integration testing project hierarchy and the system testing project hierarchy. Update the integration testing project hierarchy first, then apply the change to your system testing project hierarchy by checking out any new projects, then updating to bring in the changes.

Additionally, when you restructure a project, you will need to perform an update and also rebuild the project hierarchy to ensure the integrity of your application. For the integration testing project, your usual short test suite should suffice. For the system testing project, your SQE team will probably need to retest your application.

The operations in this chapter include:

- [Adding an existing project to your hierarchy \(page 99\)](#)
- [Cutting a project from your hierarchy \(page 99\)](#)
- [Deleting a project from your hierarchy \(page 99\)](#)

-
- [Converting a directory to a subproject \(page 100\)](#)
 - [Adding a new project to an existing hierarchy \(page 101\)](#)

Note When you restructure projects, update the integration testing projects and perform an integration test cycle first to find and fix any problems. Your changes will be selected into your system test projects automatically during the system test cycle.

Adding an existing project to your hierarchy

1. Create a task and leave yourself as the resolver.
The new task is set as the current task.
2. View the project where you want to add the existing project.
3. Use drag and drop or copy and paste to add the existing project to the current project.
If you do not remember the name of the project you want to add, use the **Query** dialog box to find the project.
4. Complete the current task.

Cutting a project from your hierarchy

1. Create a task and leave yourself as the resolver.
The new task is set as the current task.
2. View the parent project that contains the subproject you want to cut.
3. Right-click over the subproject and select **Cut**.
This cuts the subproject from the project, but does not remove it from the database.
4. Complete the current task.

Deleting a project from your hierarchy

Note The delete operation permanently deletes the project from the database.

If you do not want the project in your hierarchy, but you still want it in your database, see [Cutting a project from your hierarchy \(page 99\)](#).

1. View the project you want to delete.
2. Right-click over the project to be deleted, and select **Delete**.
Be sure to select the appropriate scope for the deletion (project, project and members, etc.).

Converting a directory to a subproject

You can perform this operation from the CLI only.

1. Start Rational Synergy from the command prompt.

```
ccm start -h engine_hostname -d database_path -nogui
```

After the session starts, the Rational Synergy address (CCM_ADDR) is printed in your command window (Windows) or in the shell where you started the session (UNIX).

2. Set your role to *build_mgr*.

```
ccm set role build_mgr
```

3. Create a task, assign it to yourself, and set it as the default.

```
ccm task -create -synopsis "string" -default
```

4. Change to the directory in your work area above the directory you want to convert to a project.

5. Create a project, specifying the directory as its root.

```
ccm create -type project -root existing_dir -version int  
-release release -purpose "Integration Testing"
```

6. Create the platform for the subproject, if necessary.

```
ccm attr -create platform -type string -value platform  
-project project_spec
```

7. Use the `ccm unuse` command to unuse the directory.

8. Add the new integration testing project to your integration testing project hierarchy.

```
ccm use -p project_name delimiter version
```

9. **Windows users:** If you use absolute subprojects, change the makefiles, the build process, and all automated jobs to reflect the changes you have made.

If you use relative subprojects, no changes are necessary.

10. Complete the current task.

```
ccm task -complete default
```

11. Perform an integration test cycle, then [Creating a baseline \(page 68\)](#).

12. Copy a system testing project from the new project.

This operation is discussed in [Creating the system testing projects \(page 28\)](#).

13. Update your top-level system testing project, rebuild your application, then run through your test suite.

Note Repeat this process for each platform, checking out integration and system testing project for each platform.

14. Exit from the Rational Synergy CLI.

```
ccm stop
```

Adding a new project to an existing hierarchy

Steps a and b below need to occur if a developer creates the new project to be added to your existing hierarchy. Developers performing this operation need to have the *component_developer* role set in Rational Synergy Classic.

1. Be sure that the developer does the following:
 - a. The developer must complete his current task.
 - b. The developer must check in his new project.
2. Copy an integration testing project from the project you just checked in. Be sure to set the version, purpose, platform and release.
3. Add the new integration testing project to the integration testing project hierarchy.

If the developer already added the project to a directory in the hierarchy and checked in that directory, update the integration testing project hierarchy to ensure that Rational Synergy selects the new directory and includes the new project. Update is described in [Updating a project \(page 48\)](#).

Alternatively, if the developer did not add the new project to the hierarchy, you will need to add the project to the integration testing project hierarchy. (This operation is described in [Adding an existing project to your hierarchy \(page 99\)](#).) Additionally, you will need to create a task, then complete the task when the change is complete.

If you get empty directory entries in your new project, it could be because some objects are not associated with a task for this release.

4. Create an external project, if needed. Note that you need to add one build management project for integration and one for system testing project.

This operation is discussed in [Creating an external project \(page 80\)](#).

-
5. Create versions for parallel platforms, if needed. Note that you need to add one for integration and one for system testing project.

This operation is discussed in [About the platform file \(page 17\)](#).

6. If the new project applies to multiple releases, create parallel release versions of it. Note that you need to add one for integration and one for system testing project.

This operation is discussed in [Setting up parallel releases \(page 95\)](#).

7. Change the makefiles, the build process, and all automated jobs to reflect the changes you have made.

8. Complete any tasks you have used for restructuring.

9. See [Publishing a baseline to developers \(page 71\)](#).

10. Update your integration testing project hierarchy, rebuild your application, then run through your test suite.

11. Create another baseline.

12. Create a corresponding system testing project for each new integration testing project.

This operation is discussed in [Creating the system testing projects \(page 28\)](#).

13. Update your system testing project hierarchy, rebuild your application, then run through your test suite.

14. Perform system testing.

Note Complete this process for each platform, checking out integration and system testing project for each platform.

10

Build management variations

All companies do not build their applications in exactly the same way. Each company has its own special needs. Company Q might be very new and small with one product offering, while Company V might be very large, offering several products on different platforms.

The Rational Synergy task-based methodology includes the use of different features in Rational Synergy. Some of the features build managers will use are not included in detail in this book because they are not required. The features do, however, make the build manager's job much easier by automating operations and by keeping developers from performing operations that are complex.

The following information is intended to give direction to those companies for whom the standard methodology, given in the previous chapters, does not apply completely. If your company falls into this category, you will use a variant of the methodology to perform your build management duties. The following variants are discussed:

- [Build management for UNIX and PC together \(page 104\)](#)
- [UNIX work areas with local files \(page 105\)](#)
- [Grouping projects \(page 106\)](#)
- [Creating a custom folder template query \(page 109\)](#)

Build management for UNIX and PC together

If your application runs on both UNIX and the PC, be sure to consider the following before you attempt to build:

- Work area visibility

A Rational Synergy session on UNIX probably cannot see your work areas on the PC, and a session on the PC cannot see your work areas on UNIX.

- Makefile formats

Most sites probably have existing parallel makefiles for building on UNIX and the PC; if so, you can continue using your makefiles on both platforms. If not, do the following:

- a. Set up parallel versions of the project, one for each platform.
- b. Set up parallel versions of the makefile, setting each **platform** property for the makefile to the appropriate value.

When you update the parallel versions of the project, each will bring in the appropriate makefile for its platform.

- Automation

When you automate your update/build process, you will need to automate the Windows and UNIX jobs separately with shell scripts or batch files, or use a cross-platform scripting solution, such as Perl or Cygwin.

UNIX work areas with local files

By default, UNIX work areas contain symbolic links into a secure directory structure that contains the controlled files. However, you can set up your UNIX work areas with local file copies, if necessary. Consider the following advantages and disadvantages before changing the default:

Advantages

- You can disconnect your local machine and use your work areas.
This option is most useful to developers.
- Build times may be faster if all files are local rather than accessed through a file server such as NFS.

For large software teams, the performance associated with building across a file server can make this type of build prohibitive. This feature enables you to have your files on the local disk when you build, bypassing the expense of accessing files through a file server.

Disadvantages

- Regular operations will be a little slower.
You can use local copies to speed local builds, but other daily operations, such as check out, check in, and update, will be a little slower. This is because they need to copy the file back and forth across the network between your database and work area.
- Multiple copies of shared files can present problems if accessed outside of Rational Synergy.

With local copies, if a developer has the same *working* object version in two projects, there are two copies of that file; each work area has its own copy. The copies of the file have no knowledge of each other.

During Rational Synergy operations, Rational Synergy keeps all copies of the file up-to-date in all visible work areas. (A visible work area is one in which the Rational Synergy client performing the operation can see the file system location for the work area. As long as the Rational Synergy client can see all of the work areas that a file is in, it will update each of the work areas with any changes during Rational Synergy operations.) However, if you work directly in your work area (for example, in FrameMaker™), your change is made to the one file in which you are working.

Note If you make changes to multiple file copies from multiple work areas without synchronizing or accessing the files through Rational Synergy between the changes, you will

create work area conflicts that will not be resolved without your interaction (i.e., by using the **Sync** option to detect work area conflicts for projects and subprojects).

Grouping projects

A grouping project specifically groups projects. For example, a good use for a grouping project is to contain the different platforms for a software application. If you have all of your projects structured into one big hierarchy, you can check out new versions of all projects by using the **Copy Project** dialog box **Subprojects** list box options.

Note Grouping projects are optional. They are useful because they make it much easier to check in a set of projects or check out a new set of projects when they are all grouped in a hierarchy.

About the grouping project to be created

As an example, assume you want to group different platforms of a project called **toolkit_top-3.0**. It will group **toolkit-win** and **toolkit-unix** together.

To set up the **toolkit_top-3.0** grouping project, you need to set up uniquely named projects.

Two projects with the same name cannot be subprojects within the same parent project. For example, **toolkit-win** and **toolkit-unix** cannot be grouped together in the same project.

To group projects with the same name, create an extra level of projects with unique names. A single project with a work area cannot contain subprojects whose work areas are not visible to the parent project. Therefore, you need to turn off work areas, which is described in [Creating a grouping project \(page 107\)](#) in [step 3](#).

Grouping projects versus project groupings

Grouping projects are not the same as project groupings. Project groupings are created automatically by Rational Synergy to organize all projects belonging to the same release and purpose. Once a project is checked in, however, it is no longer a member of any project grouping.

Grouping projects are created by you and are completely under your control.

Creating a grouping project

This example creates a project called **toolkit_top-3.0** to represent all of the **toolkit** projects for different platforms. Be sure to read [About the grouping project to be created \(page 106\)](#) before you start this operation.

1. Create new projects with unique names for each platform.
Be sure to set each platform value for the project, if it will contain projects for a specific platform.
2. For each of the new platform projects, add the existing projects for that platform as subprojects. (Use drag and drop to do this.)
If you do not remember the name of the project you want to add, use the **Query** dialog box to find the project, then use drag and drop to add it.
3. Create the top-level grouping project and turn off work area maintenance for the new grouping project.

- a. Create a new project.

Task > New > Project

- b. Right-click over the project you just created and choose **Properties**. In the **Work Area** tab, turn off work area maintenance. Save the change by clicking **OK**.

Note Do not set a platform value. This project will contain subprojects with different platform values. If you set a platform value for the top-level grouping project, only the subprojects with the same platform value will be included after you update the top-level grouping project.

- c. Add each of the new platform projects as members of the top-level grouping project.

Right-click over the root directory, point to **Create Member**, then choose **Subproject**. Type the new subproject name in the **Create Subproject** dialog box. If the subproject exists, drag and drop it under the top-level grouping project.

Creating a release for a patch

The following process shows you high-level steps for creating a release for a patch. Many of the low-level operations you need to perform are discussed in earlier sections of this book, and you can use the links provided here or your own bookmarks to go to those sections.

This process assumes that you have set release values **consistently** on all of your projects. If you have not, this process will **not** work.

-
1. Create a release for the patch. (For steps, see [Creating or copying a release](#) in the Rational Synergy Help.)

Typically, you will copy the release you are patching.

2. Set the process rule you will use for the patch in the **Create Release** dialog box.

Typically, you will need the **Collaborative Development** and **System Testing** process rules.

Creating a patch

The following process shows you high-level steps for creating a patch.

1. Create a task (or several tasks) and assign it to the appropriate developer(s) for the patch release.
2. Give the project information and new release values to the developers.

Developers will need to copy a new development project from the baseline release, set their current task, diagnose and fix the problem, unit test, then complete the current task.

Note Be sure that developers who are fixing problems set the release value on their projects **before** checking out files. They also will need to set the current task so that objects they check out will be associated with the correct task automatically. If they forget these steps, they will have to perform several manual steps to ensure that the correct objects are included in the patch.

Also, developers should **not** complete any other work in the patch projects. This is so that other checked out versions are not tagged with the patch release value and included in the patch accidentally.

3. Create a system testing project for each project being patched.

You can follow the same steps to create the system testing project **with the following exception:**

Copy only the projects to be included in the patch.

Creating a system testing project is discussed in [Creating the system testing projects \(page 28\)](#).

Note You do not need an integration *prep* patch project.

4. See [Building with specific tasks \(page 41\)](#).
5. Set up a test area.

To set up a test area for the patch, you can install a copy of the released software to an installation area (e.g., **patch_test_1.2**), then copy the products built for the patch into the area. (Ideally, you should install the patch the same way your customers will install it.)

If you have a special utility for installing patches, use it instead.
6. System testing proceeds; if defects are found, start the cycle again at [step 1](#) (however, do not repeat [step 2](#) and [step 3](#)).
7. When the patch passes your quality standards, make it available to customers.
8. Check in your new projects and create a baseline that contains the new projects as well as the other projects in the hierarchy.

Creating a custom folder template query

A build manager might need to create a folder template whose folders gather tasks with a particular property.

In this scenario, your team is developing two parallel releases at the same time. You want to set up a completed tasks folder to collect changes from both releases (in this scenario, release **toolkit/2.1** and release **toolkit/3.0**).

1. Start the **Query** dialog box to find a folder template you want to copy.

Find > Folder Templates

The **Query** dialog box appears.
2. Select the folder template you want to customize, then click **Copy Folder Template**.

The **Copy Folder Template** dialog box appears.
3. Set the following custom query:
 - a. Set the first list to **In State** and set the state to **completed**.
 - b. Set the **For Release** list to the appropriate release.
 - c. Add another query clause for another release, if necessary.
 - d. Set the **Modifiable in Database** list to the appropriate database name.
4. Save the settings by clicking **OK**.

Built in folder templates cannot be customized.

Adding additional test phases

To add additional test phases, you will need to create a process rule and purpose to represent the phase.

You can add test phases at any point in a release.

1. Create a process rule. ([Setting up a process rule](#) is described in Rational Synergy Help.)
2. If you need to create a purpose, see [Creating a purpose](#) in Rational Synergy Help.
3. Edit the release that will use the new process rule.

Right-click over the release and choose **Properties**.

In the **Process Rules** tab, click **Add Process Rule**, select the new process rule, then click **OK**.

Now you are ready to copy projects for your new purpose.

Appendix A: Convert to process rules

Existing customers will find this chapter useful. Additionally, if your site is new, but started out not using process rules and now wants to transition to them, this is the right chapter for you.

New customers can skip this chapter and follow the instructions in [Prepare for build management \(page 13\)](#) and [Build management basics \(page 29\)](#). Process rule usage is described for new users in those chapters.

Process rules requirement

The build management methodology uses purposes, process rules, and folder templates. All documented methodology discussions are centered around these features.

Support for updating projects manually is an obsolete feature. All sites should use process rules now.

Converting projects

A site can convert to process rule use at any time. Converting at the start of a release is easiest because you can ensure that the team is using process rules consistently.

The following sections describe the conversion process for build managers and developers.

Converting to process rules for build managers

Before users convert their projects to use process rules, the build manager should do the following. This operation can be done from the Rational Synergy CLI.

1. Start Rational Synergy from the command prompt.

```
ccm start -h engine_host -d database_path -nogui
```

After the session starts, the Rational Synergy address (CCM_ADDR) is printed in your command window (Windows) or in the shell where you started the session (UNIX).

2. Set your role to *build_mgr*.

```
ccm set role build_mgr
```

3. Query for all process rules for the new release.

You will run a command on the selection set (query output).

```
ccm query -type process_rule "release='new_release' "  
where 'new_release' is the new release name.
```

4. Specify that you want new projects to use process rules by default.

```
ccm process_rule -modify -default @
```

If your site is converting from manual updating, you will need to set processes and process rules for active projects. Rational Synergy contains default processes and process rules that might fit your needs, or you might need to create custom processes and process rules.

5. Inform developers that they can convert their projects to use process rules, as described in [Converting to process rules for developers \(page 113\)](#).
6. Exit from the Rational Synergy CLI.

```
ccm stop
```

Converting to process rules for developers

When developers are ready to start using process rules, they will need to convert their existing projects. If you are starting a new release, developers can copy new projects instead of converting their old ones.

To convert existing projects to use process rules, perform the following steps. (Note that before anyone can use this procedure, the build manager must have set up process rules for the current release. Refer to [Converting to process rules for build managers \(page 112\)](#).)

1. Use the **Project Properties** dialog box to change each project's release setting. To change the release value for all projects in a hierarchy, perform the following:
 - a. Right-click on the top-level project and select **Properties**.
The **Project Properties** dialog box appears.
 - b. Select the new release value in the **Release** list.
If the project has subprojects, the release will be changed on all subprojects automatically.
 - c. Be sure the **Purpose** is set to **Insulated Development**, **Collaborative Development**, or **Custom Development**. If you use **Custom Development**, you will need to select your baseline. (See [Selecting a new baseline](#) in Rational Synergy Help for instructions.)
 - d. Save the changes.
2. Start Rational Synergy from the command prompt.

```
ccm start -h engine_host -d database_path -nogui
```

After the session starts, the Rational Synergy address (CCM_ADDR) is printed in your command window (Windows) or in the shell where you started the session (UNIX).
3. Query for all process rules for the new release.
You will run a command on the selection set (query output).

```
ccm query -type process_rule "release='new_release' "
```

where *new_release* is the new release name.
4. Specify that you want new projects to use process rules by default.

```
ccm process_rule -modify -default @
```
5. Exit from the Rational Synergy CLI.

```
ccm stop
```

If you linked to this chapter from the "Prepare for Build Management" chapter, and you want to pick up the flow of operations again, the next section for you to read is [About parallel releases and platforms \(page 24\)](#).

Appendix B: Notices

© Copyright 1992, 2009

U.S. Government Users Restricted Rights - Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT,

MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

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

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

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

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

Intellectual Property Dept. for Rational® Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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

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

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

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos and Solaris are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Terms and concepts

- [baseline \(page 121\)](#)
- [baseline project \(page 121\)](#)
- [breadcrumbs \(page 121\)](#)
- [build \(page 121\)](#)
- [build manager \(page 122\)](#)
- [change request \(page 122\)](#)
- [check in \(page 122\)](#)
- [check out \(page 122\)](#)
- [common ancestor \(page 122\)](#)
- [compare \(page 122\)](#)
- [component task \(page 122\)](#)
- [complete a task \(page 122\)](#)
- [component name \(page 122\)](#)
- [component release \(page 122\)](#)
- [conflict \(page 122\)](#)
- [controlled product \(page 122\)](#)
- [copy project \(page 123\)](#)
- [current task \(page 123\)](#)
- [database \(page 123\)](#)
- [DCM \(page 123\)](#)
- default task, see [current task \(page 123\)](#)
- [delete \(page 123\)](#)
- [difference \(page 123\)](#)
- [directory \(page 123\)](#)
- [directory entry \(page 123\)](#)
- [file \(page 123\)](#)
- [folder \(page 123\)](#)
- [generic process rule \(page 123\)](#)

-
- [grouping project \(page 124\)](#)
 - [history \(page 124\)](#)
 - [incremental baseline \(page 124\)](#)
 - [inline differences \(page 124\)](#)
 - [instance \(page 124\)](#)
 - [merge \(page 124\)](#)
 - [merge conflict \(page 124\)](#)
 - [object \(page 125\)](#)
 - [process \(page 125\)](#)
 - [process rule \(page 125\)](#)
 - [parallel conflicts \(page 125\)](#)
 - [parallel version \(page 125\)](#)
 - [product \(page 125\)](#)
 - [product task \(page 125\)](#)
 - [project \(page 125\)](#)
 - [update properties for a project \(page 126\)](#)
 - [project task \(page 126\)](#)
 - [project grouping \(page 126\)](#)
 - [properties \(page 126\)](#)
 - [purpose \(page 126\)](#)
 - reconcile, see [sync \(page 127\)](#)
 - reconfigure, see [update \(page 127\)](#)
 - [process rule \(page 125\)](#)
 - [regular expressions \(page 126\)](#)
 - [release \(page 126\)](#)
 - [release \(page 126\)](#)
 - [remove \(page 127\)](#)
 - [sync \(page 127\)](#)
 - [Rational Change \(page 127\)](#)
 - [task \(page 127\)](#)

-
- [task-based methodology \(page 127\)](#)
 - [test baseline \(page 127\)](#)
 - [Traditional mode \(page 127\)](#)
 - [type \(page 127\)](#)
 - [update \(page 127\)](#)
 - [update properties \(page 128\)](#)
 - [process rule \(page 125\)](#)
 - [use a version \(page 128\)](#)
 - [version \(page 128\)](#)
 - [Web mode \(page 128\)](#)
 - [work area \(page 128\)](#)
 - [work area conflict \(page 128\)](#)

baseline	This is a snapshot of a set of projects and tasks at a point in time. May be used as the starting point for further development; may be compared to other baselines for reference.
baseline project	The project version on which you base your project is called its baseline project. For example, the baseline project for the editor-2.0 project would be editor-1.0 . When you check out a new version of a project, its baseline project is set automatically. A baseline is made up of baseline projects.
breadcrumbs	Breadcrumbs are a navigation technique that give users a way of tracking where they are in a document. Breadcrumbs typically appear horizontally across the top of a web page, usually below any title bars or headers. They provide a trail for the user to follow back to the entry point of a document.
build	A build is the execution of commands for a target in a makefile. A build creates a product, such as a library, executable, or relocatable object.

build manager	The build manager is a user who gathers and builds changes for a development team.
change request	A change request is a request for a change created in Rational Change.
check in	The check in operation preserves one or more files and makes them available to other users.
check out	The check out operation creates a new version of a file that can be modified by the user who checked it out.
common ancestor	The common ancestor is the most recent predecessor of the files being merged.
compare	The compare operation shows the differences between the contents of two objects.
complete a task	You can complete a task when all work on the task is finished. This causes all objects associated with the task to be checked in and available to the build manager to build the product.
component name	The component name is an optional part of a release. It might represent the name of an application or component, such as Synergy or editor .
component release	The component release is a part of a release. It identifies the specific release of that application or component.
component task	A component task is a task that collects projects or products from a baseline. It is always in the <i>component_task</i> state.
conflict	A conflict signifies that your work area is out of sync or an object has parallel versions.
controlled product	A controlled product is a file that is built or generated. A controlled product can be any type of

	object except a project or directory, but the most common controlled products are executables and libraries.
copy project	The copy project operation copies a project for your use. You will need to make a copy of a project to modify its content.
current task	The current task is the task that you are currently working on.
database	The Rational Synergy database is a data repository that stores all of your controlled data, including source and data files, their properties, and their relationships to one another.
DCM	Use Distributed Change Management to transfer data between multiple databases, enabling multi-site development.
delete	The delete operation removes an object from the Rational Synergy database.
difference	A difference is one variance between two files that have been compared or merged. Two files may have more than one difference.
directory	A Rational Synergy directory keeps track of which files belong in it.
directory entry	For each file that belongs in a directory, the directory has a place holder, called a directory entry. The directory entry identifies the file that belongs there, but not the version of the file.
file	A file is a collection of data or information.
folder	A named grouping of tasks.
generic process rule	A process rule specifies how projects for a particular purpose will choose new members during an update.

	<p>It does this by specifying how the system will find a baseline to use as a starting point and which tasks the system should use to find new members. A generic process rule is a process rule that is not part of a release.</p>
grouping project	<p>A project used to create logical groups of projects where a work area is not necessary or possible.</p>
history	<p>The history operation displays all the versions of a file, directory, or project, and the relationships between versions.</p>
incremental baseline	<p>A baseline that differs from a previous baseline by a subset of the projects. For example, if you have projects proj1 through proj100, and you want to create a new baseline with just proj1-int and proj2-int (because proj3 through proj100 haven't changed), then you can create your new baseline from your most recent baseline, plus these two projects.</p>
inline differences	<p>Inline differences show the individual modified line and character differences of two compared files.</p>
instance	<p>An instance is a property of an object. It is used to distinguish between multiple objects with the same name and type, but that are not versions of each other.</p>
merge	<p>The merge feature enables you to combine information from two parallel versions of a file. When you merge two files, a third file is created. The third file contains the blended information from both files.</p>
merge conflict	<p>A merge conflict is an inconsistency between two modified files, where both were modified on the same line but in different ways.</p>

object	Object is a general term that describes different types of data you can store in a Rational database, including files, directories, projects, tasks, and change requests.
parallel conflicts	A parallel conflict occurs when one or more parallel versions are checked out, but have not been merged.
parallel version	Parallel versions occur when two or more versions are checked out from a single file.
process	A process is a collection of process rules that define how the projects for a release are updated. For example, a release might contain a purpose called Integration Testing. Within the Integration Testing purpose, the build manager might have three process rules: Hotlist Testing, Integration Testing, and Resolved CRs. The Integration Testing purpose is flexible; it can be used to build an Integration Testing area or to perform testing in hotlist mode, depending on which process rule the build manager sets the purpose to have. Developers do not set processes or process rules; they only set purposes on projects.
process rule	A process rule contains patterns that define how projects will be updated; they specify rules for determining the baseline plus a set of tasks and folders to be used when someone updates a project. (In previous releases, "update" was called "reconfigure," "update template" was called "reconfigure template," and "process rule" was called "update template.")
product	A product is a file that is built by processing other files. Some examples of products include a .class file, .jar file, and .exe file.
product task	A task automatically created by Rational Synergy to manage a product.
project	A project is a logical grouping of selected versions of files and directories, arranged in a specific structure.

project task	A task automatically created by Rational Synergy to manage a project.
update properties for a project	The update properties for a project are the baseline and tasks on the project grouping for the project.
project grouping	Rational Synergy groups projects by purpose and release, for example, My 3.0 Collaborative Projects . This is called a project grouping. A project grouping holds the tasks and baseline used when you update a project, keeping the update properties consistent for all projects in a project grouping.
properties	The properties of an object, also known as its properties, enable you to track a variety of information. Examples of properties are name, version, and release.
purpose	A project purpose defines what it is used for, for example, Insulated Development, Integration Testing, System Testing. When you change your project purpose, Rational Synergy uses different selection criteria when you update the project.
reconcile	See <i>sync</i>
regular expressions	Regular expressions are strings of characters that define patterns used to search for matching text.
release	A release is a property that identifies a project or task that is specific to a particular release of your application.
release	A release consists of an optional component name and release delimiter, and a component release. The component name might represent the name of an application or component, such as Synergy or editor . The component release identifies the specific release of that application or component. Synergy/7.0 is an example of a release.

remove	The remove operation takes an object out of a directory or project, but does not delete it from the database.
sync	The sync operation creates a work area for one or more projects, compares the files in the work area to the database, and enables you to reconcile differences between the work area and the database.
task	A task is a to-do list item that is assigned to a user. A task also tracks the files that were modified to complete it.
Traditional mode	This is the classic architecture, which uses RFC.
task-based methodology	Task-based methodology enables a development organization to track changes to a software application using tasks, rather than individual files, as the basic unit of work.
Rational Change	Rational Change is a change request management system that is web-based and integrated with Rational Synergy. This document describes its usage with Rational Synergy, which is relevant only if you use Rational Change.
test baseline	A test baseline is one that is not yet ready to be made available to everyone. Typically, after the test baseline has passed SQE testing, then it is ready to be published for developers to use.
type	A type is the class of data contained in the object. The type defines the behavior or characteristics of an object. Examples of types are java, library, executable, and HTML.
update	You can perform an update operation to update your project or directory with recent versions checked in by other users. (In previous releases, "update" was called "reconfigure.")

update properties	These are properties that a project uses to decide which object versions to select when someone updates a project. (In previous releases, "update" was called "reconfigure," and "update properties" was called "reconfigure properties.")
use a version	You can use a different version of a file or directory in your project. You may perform the use operation during unit testing, when you want to go back to a previous version of a file in your project.
version	A version is a specific variation of a file, directory, or project.
Web mode	Rational Synergy 7.0 introduces a new architecture where Rational Synergy clients communicate to a Web-based Rational Synergy server using the HTTP protocol.
work area	The work area is a location in the file system that contains your copies of files, organized by projects.
work area conflict	A work area conflict is an inconsistency between the contents of your work area and the database.

Index

A

- application packaging
 - defined, 85
 - typical mediums, 85
- automation in builds, 33

B

- baselines
 - and update process, 72
 - bad, example, 66
 - changes to developers, 68
 - changes unavailable to developers, 38
 - complete, example, 65
 - correct, example, 65
 - create, 68
 - defined, 63
 - delete, 75
 - how they work, 63
 - included in baseline, change, 70
 - incremental, how to create, 73
 - mark for deletion, 75
 - marked for deletion, and update, 75
 - methodology, 68
 - partial, how to create, 73
 - problems when not set, 55
 - project, problems when not set, 55
 - publish, 71
 - query database for, 64
 - roll back to previous version, 38
 - Save Offline and Delete, 75
 - test, described, 68
 - test, methodology, 68
 - turn on work area maintenance, 64
- build management
 - road map, 3
 - UNIX and PC, 104
 - work area set up, 15
- build management project

- defined, 25
- steps to set up, 25, 26
- system testing
 - project, create, 28

- build property, 69
- builds
 - arguments, for int test, 26
 - automation, 33
 - make available to others, 71

C

- ccm_root group, when to become member, 14
- cleanup, releases and process rules, 75
- collaborative development, defined, 23
- compare, defined, 122
- component
 - name, defined, 18
 - release, defined, 18
- config, project (troubleshooting problems in), 55
- conflicts
 - and dependencies, 59
 - categories of, 58
 - defined, 29, 122
 - detection, discussed, 57
 - how detected, 57
 - messages, defined, 60
 - resolve, 61
 - show, GUI, 58
- convert to process rules use
 - developer procedure, 113
 - when to, 112
- copies, multiple (and shared files), 105
- custom folder template query, 109

D

- default text editor used in this document, 5
- delete, defined, 123
- deleting
 - baselines, 75

- baselines, mark, 75
- project groupings, empty, 76
- project hierarchies, obsolete, 76
- dependency relationships
 - defined, 59
 - example, 59
- development
 - collaborative, defined, 23
 - insulated, defined, 23
- directory entries, empty, 51
- documentation, available, 7

E

- editor, text (used in this document), 5
- empty directory entries, 51
- external project
 - contents, 79
 - create, 80
 - defined, 79
 - example, 79
 - why to use, 78

F

- files
 - platform, discussed, 17
 - shared, and multiple copies, 105
- folder
 - add test phases (why and how to), 110
 - external projects, freeze (CLI), 82
 - external projects, thaw (CLI), 83
- folder template query, custom, 109
- freeze, external projects folder, CLI, 82

G

- grouping project
 - create, 107
 - defined, 106
 - discussed, 106
 - unique project names, 106
- guidelines, update, 47

H

- hierarchies, project, remove, 76
- history, defined, 124

I

- IBM Customer Support, 9
- inactivate a release, 75
- incremental baselines, how to create, 73
- installation area, defined, 34
- installation project
 - create (GUI), 87
 - defined, 86
- insulated development, defined, 23
- integration testing project
 - build cycle activities, 37
 - create, 26
 - reuse, 44

L

- local files and UNIX work areas, 105
- log, update, 50

M

- merge, defined, 124
- messages, conflicts (defined), 60
- methodology
 - baselines, 68
 - test baselines, 68
 - variant, defined, 103
- multiple copies of shared files, 105

N

- name property, 68
- new release, what to do, 43
- NT Servers and platform file, 17

O

- om_hosts.cfg, platform file location, 17

P

- packaging applications
 - defined, 85
 - typical mediums, 85
- parallel
 - environment, platforms, 93
 - notifications, when to turn on, 14
 - release, defined, 24
 - releases and platforms, defined, 24
- parallel versions, not merged, 51
- partial baselines, how to create, 73
- patch
 - and release values, 91, 107
 - defined, 90
 - fixes, obtain from developers, 91
 - release, 90
 - release, create, 91, 107
 - what to include in, 90
- platform file
 - and NT Servers, 17
 - and update, 17
 - location, 17
- platform values, when to use, 17
- platforms
 - parallel releases, defined, 24
 - set up, 18
 - ship multiple, 93
- process rules
 - and purposes, defined, 20
 - cleanup, 75
 - convert, developer procedure, 113
 - delete, when and how to, 75
 - new release, build manager, 112
 - new release, developer, 113
 - use for new release, 112
 - when to convert to use, 112
- product sharing
 - defined, 77
 - example, 77
- project config problems, troubleshooting, 55

- project groupings
 - defined, 106
 - empty, remove, 76
 - included in baseline, change, 69
- project hierarchy
 - add existing proj to (GUI), 99
 - create subproj from dir (CLI), 100
 - delete existing proj from (GUI), 99
- projects
 - build management, defined, 25
 - build management, steps to set up, 25, 26
 - defined, 125
 - external, contents, 79
 - external, create, 80
 - external, defined, 79
 - external, example, 79
 - external, why to use, 78
 - hierarchies, remove, 76
 - included in baseline, change, 69, 74
 - installation, create (GUI), 87
 - installation, defined, 86
 - integration prep, defined, 25
 - integration testing, create, 26
 - restructuring, discussed, 97
 - system test prep, defined, 25
 - system testing, create, 28
 - variant, 93
- properties
 - build, 69
 - name, 68
- properties, defined, 126
- publish baselines, 71
- purposes and process rules, defined, 20

Q

- query, create for custom folder template, 109

R

- release

- characters allowed, 19
- defined, 18
- discussed, 18
- example, 18
- for patch, 90
- name creation, 19
- name, character restrictions, 19
- name, text string length, 19
- names, examples, 18
- new release, 43
- new, what to do, 43
- parallel (defined), 24
- patch name, 90
- patch, create, 91, 107
- release software, 42
- ship multiple, 93
- software, 42
- values, important for patches, 91, 107
- values, modify, 20
- release value, update for incomplete tasks (GUI), 43
- releases, inactivate, 75
- removing, see also, delete
- replaced subprojects, 51
- resolve conflicts, 61
- restructure a project, discussed, 97
- road map, build management, 3
- rules, process
 - convert (developer procedure), 113
 - delete, when and how to, 75
 - when to convert to use, 112

S

- Save Offline and Delete, and baselines, 75
- shared files in multiple copies, 105
- shared location for build mngmt work
 - areas, 15
- sharing products
 - discussion, 77
 - example, 77
- ship

- multiple platforms, 93
- multiple releases, 93
- show conflicts, GUI, 58
- subprojects, replaced, 51
- Synergy documentation, 7
- system test
 - area, fix defects in, 40
 - cycle, discussed, 39
 - cycle, example, 40
 - cycle, test levels, 39
- system testing, create project, 28

T

- tasks, defined, 127
- templates
 - folder, custom query, 109
 - set work area path, 15
- test level, for system test cycle, 39
- testing project
 - integration, create, 26
 - integration, reuse, 44
- text editor used in this document, 5
- thaw external projects folder, CLI, 83
- trademarks, 117
- troubleshooting
 - conflict resolution, 61
 - proj config problems, 55
 - selection problems, 54
 - update properties, 54

U

- UNIX work areas and local files, 105
- update
 - and baselines, 72
 - and baselines marked for deletion, 75
 - guidelines, 47
 - int prep proj hierarchy, 48
 - steps during operation, 45
 - verbose, 52
 - verbose, dir level vs proj level, 54
- update log

- and verbose option, 52
- empty directory entries, 51
- parallel versions not merged, 50
- replaced subprojects, 51

use a version, defined, 128

V

- values, platform (when to use), 17
- values, release
 - important for patches, 91, 107
 - update for incomplete tasks (GUI), 43
- variant
 - methodology defined, 103
 - projects, 93
- verbose update
 - dir level vs proj level, 54
 - option, 52
- versions, parallel, not merged, 51

W

- work areas
 - build management, set up, 15
 - defined, 128
 - path template, set, 15
 - set for specific release or platform, 15
 - UNIX and local files, 105
- workflow, defined, 35

