

IBM Rational Team Concert V4.0 Enterprise Extensions Build Administration Workshop



Acknowledgments and Disclaimers

© Copyright IBM Corporation 2013

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo, and ibm.com are 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 "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Session objectives

This workshop is a hands-on working session that will expose System z build administrators to many of the tasks that they might need to perform to migrate and maintain their source control and build infrastructure using Rational Team Concert.

The session is meant as a learning exercise, and can be done at the student's own pace.

At the end of this workshop, you should have an awareness and understanding of the following:

- Upfront planning and decisions involved in preparing for a migration
- Representation of host resources and build steps using RTC system definitions
- Configuration of build process using RTC build definitions
- Control of development hierarchy (Development->Test->QA->Production) through RTC promotion
- Release of changed applications to Test and Production environments through RTC deployment

Agenda

- **Overview**
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from Development to Production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Overview: The big picture

There are a number of different things that you will do during this workshop. The main areas of focus during this workshop include hands-on examples of:

- Creating **system definitions** to represent the host resources and steps involved in the build process
- Organizing mainframe source code into the required **zComponent project** structure
- Configuring **dependency build definitions** to compile and link changed mainframe applications
- Utilizing **promotion** to flow changes from Development to Production
- **Deploying** built applications to a runtime environment

Overview: Getting help

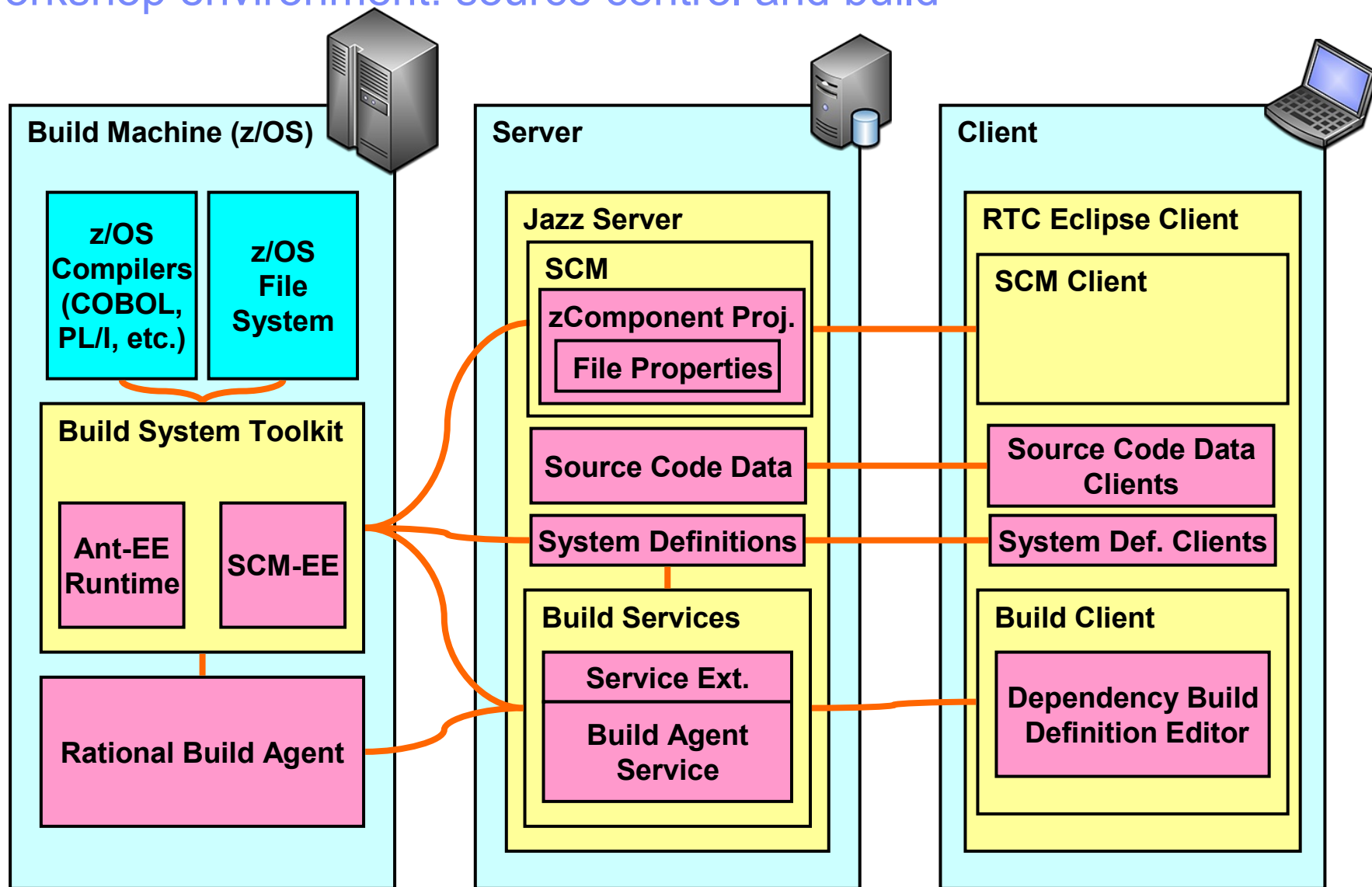
When you are working on your migration and build configuration in your own environment, you should be aware that you have help.

- Use the Information Center – it is the official owner’s manual for your CLM implementation
 - It’s at <http://publib.boulder.ibm.com/infocenter/clmhelp/v4r0m/index.jsp>
 - Infocenter versions will change with each release. Look at the “v4r0”. For the 3.0.1 release of CLM this part of the URL was “v3r0m1”.
- Use [Jazz.net](#)– the library has a wide array of videos and articles showing you how to configure and deploy your CLM solution
- Use the [Jazz forums](#) – post your questions on the forums, and search the forums for answers to your questions. You cannot be the only person in the world asking these questions, right?
- Check out the work items on Jazz.net. If something doesn’t work the way that you think it should, then enter a defect out on Jazz.net
- Follow the Enterprise Extensions blogs:
 - ryehle.wordpress.com
 - rtcee.wordpress.com

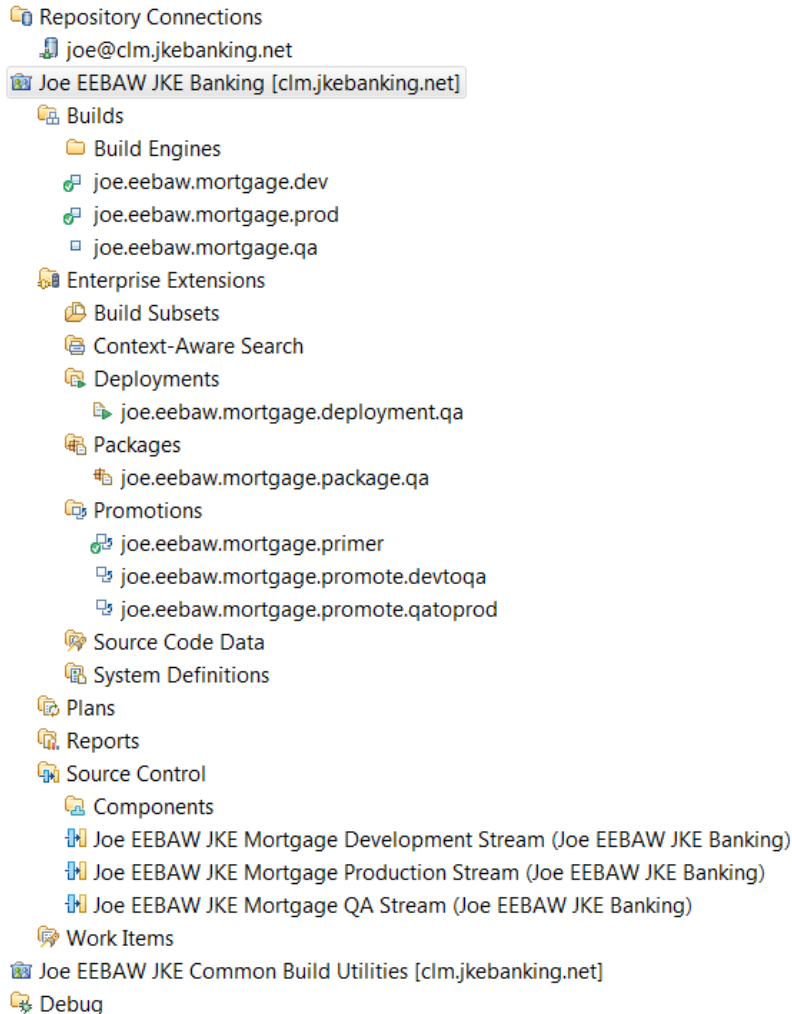
Agenda

- Overview
- **Installation and setup**
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from Development to Production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Workshop environment: source control and build



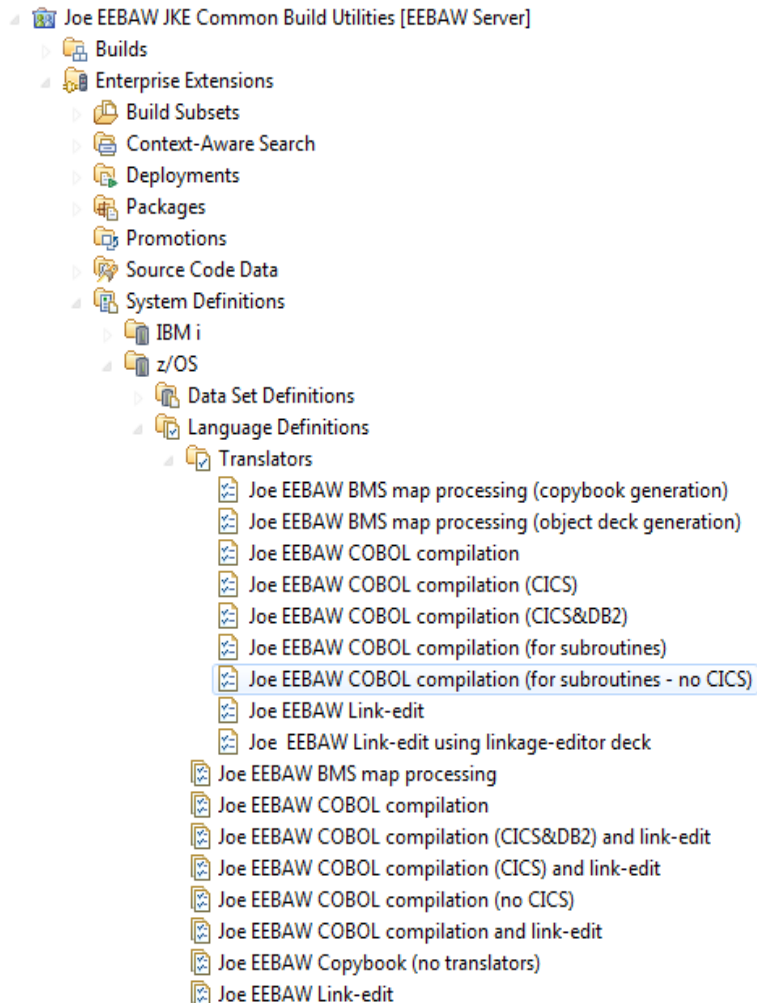
EEBAW JKE Banking project area



As part of the workshop setup, you will generate a new project area where you will perform the majority of the tasks throughout the workshop.

This EEBAW JKE Banking project area is where your mainframe developers have been successfully using work items and planning, and where (after your successful migration) they will be accessing and updating the mainframe application source code and building their changes.

EEBAW JKE Common Build Utilities project area



As part of the workshop you will also generate a project area for storing the Build Utilities.

This EEBAW JKE Common Build Utilities project area will serve as a central point for managing the system definitions that will be used for building and deploying the z/OS applications. System definitions are repository wide elements; so in your role of z/OS build administrator you will administer the needed elements in this central point, easing the configuration and maintenance of these resources.

Workshop preparation: Installation and Setup

Please complete the workshop installation and setup:

1. Install all product components not provided to you (document “EEBAW Installation Instructions.pdf”)
2. Deploy the workshop pre-configured process templates and create JKE Banking project area (document “EEBAW Labs.pdf”, “Workshop setup and preparation” section)

Agenda

- Overview
- Installation and setup
- **Lab 1: Planning your Rational Team Concert solution**
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from Development to Production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Planning your Rational Team Concert solution

Questions

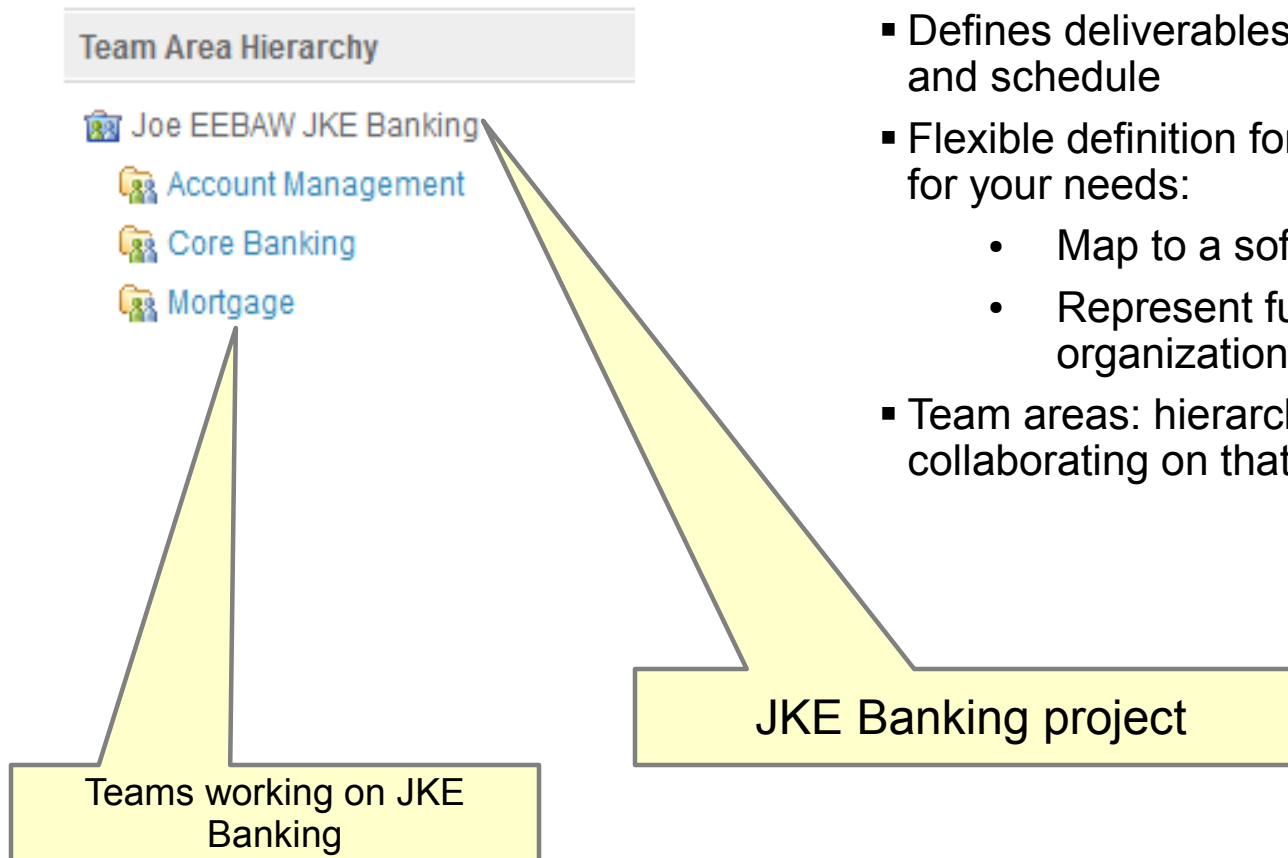
- How will you organize your people?
- How will you organize your source code?
- How many “levels” do you require to promote your changes from Development to Production?
- How will you build your applications? Where do they need to be tested and run?
- Who will be your early adopters? How will you pilot, and how will you grow the adoption?

RTC Concepts

- Project areas and team areas
- Components
- Streams
- Build
- Promotion
- Deployment

Project area and team areas

How will you organize your people?



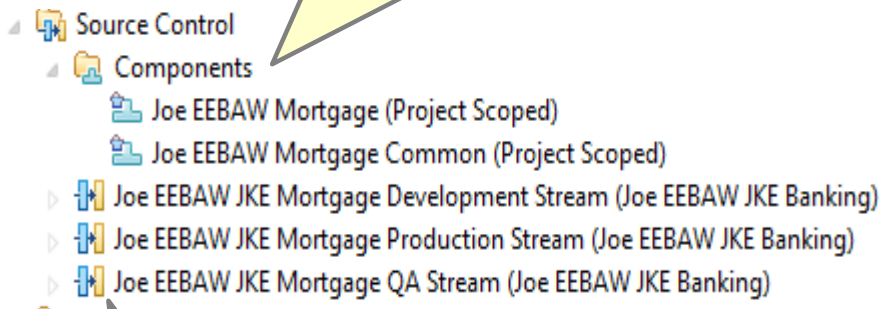
- A project area represents a software project effort
- Defines deliverables, team structure, process, and schedule
- Flexible definition for “what's a project area” for your needs:
 - Map to a software project
 - Represent functional areas within your organization
- Team areas: hierarchy of teams working and collaborating on that project

Components and streams

How will you organize your source code?

How many “levels” do you require to promote your changes from Development to Production?

Componentization of
JKE Mortgage application



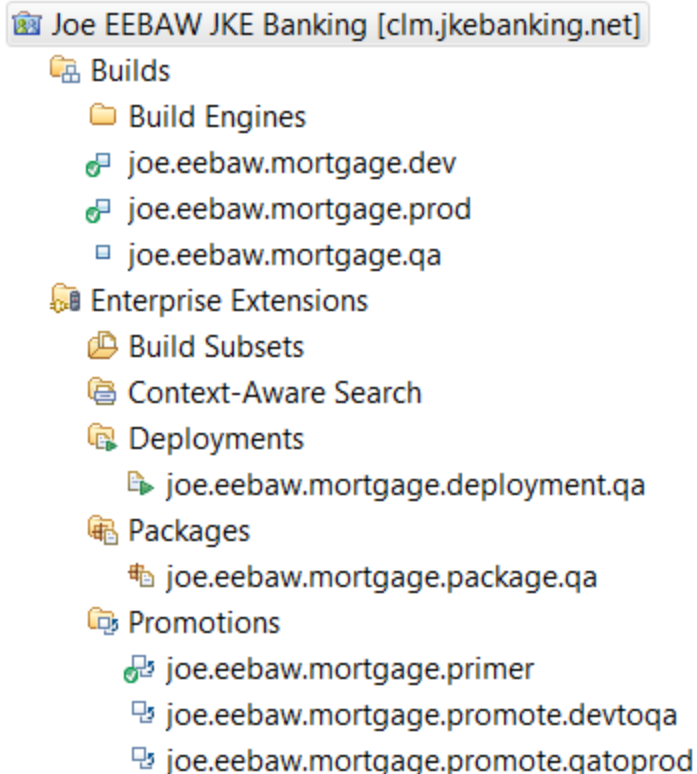
Streams for coding and building JKE
Mortgage application

- Components are SCM fundamental logical structure for organizing assets. A componentization effort for adopting SCM:
 - Think of the logical units of your application
 - Common and reusable units
 - Components built by specific teams
- Organize the development cycle stages in streams:
 - Organize your work (streams contain components)
 - Coordinate integration
 - Capture configuration points
 - Mirror your build and runtime environment levels

Build, promote, package and deploy

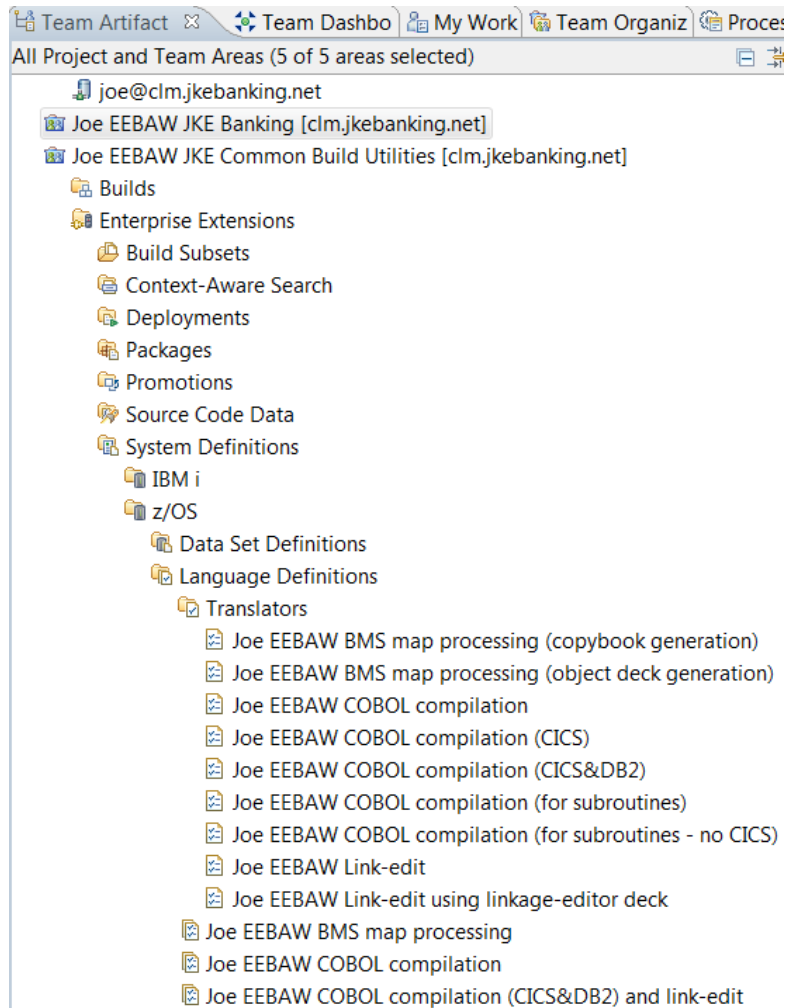
How will you build your applications?

Where do they need to be tested and run?



- Configure how your applications are built using **build definitions** that execute against a **build engine** (your z/OS)
 - **Dependency Build** features allows you to just build what's changed, saving time and resources.
- **Promote** changes throughout your defined development hierarchy
- **Package** your built artifacts
- Use **deployment definitions** to finally move your built application modules to the runtime environment
- All these elements leverage **System Definitions**

System definitions



- Capture and maintain supporting information required to build your mainframe applications
- Three types:
 - **Data set definitions** represent all MVS data sets involved in building your application from the source data sets (to be created and loaded with source from the repository) to the output data sets (generated by the build) and even the compiler itself
 - **Translators** represent a step in the build process, such as a compile or link-edit
 - **Language definitions** represent the complete ordered set of steps (translators) required to build a given source program

Lab 1

Please complete **Lab 1**

- Understand the basic set of information you need to consider from your existing environment in preparation for migration to Rational Team Concert
- Map the information you are gathering to some of the concepts in Rational Team Concert
- Create a project area to hold the system definitions to be used for building and deploying the applications enterprise wide

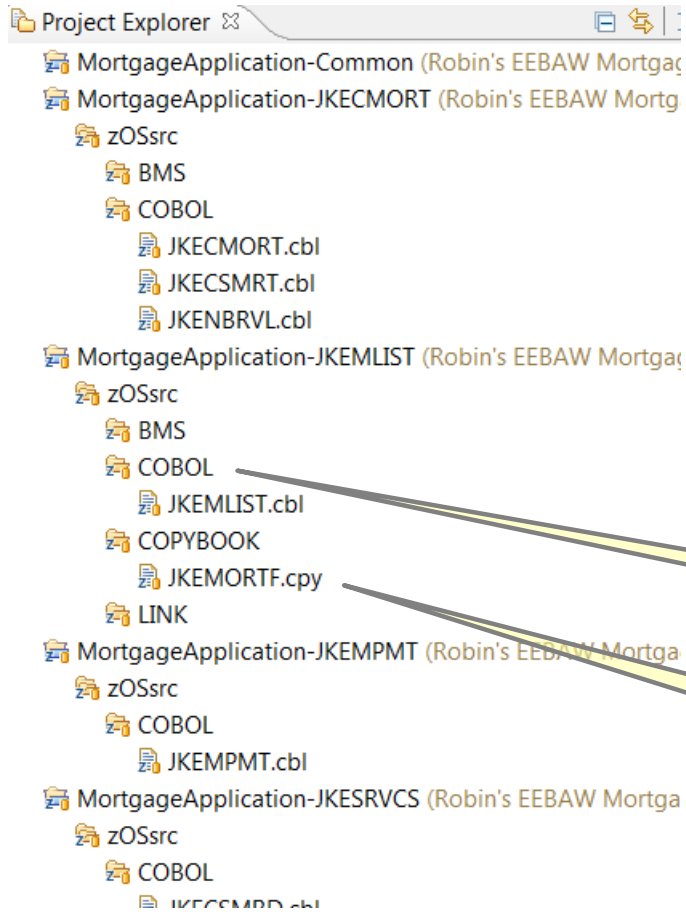
Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- **Lab 2: Sharing your source members in Rational Team Concert**
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from Development to Production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Migrating your source into the Jazz repository

- For the purpose of this lab, you will simply import existing zComponent projects into your eclipse workspace and share them
- In a real migration, you would utilize the **zimport** SCM command line tool (aka “mass import tool”) to import your PDS members directly into the repository
 - Automatically creates the proper zComponent project structure
 - Automatically creates a data set definition based on characteristics of data set on host
 - Automatically (optionally) associates language definitions with each member
- You can build a source code version history of your major releases by running a series of zimports with the same repository workspace
- zimport resources:
 - [System z mass import tool overview \(Information Center\)](#)
 - [Getting my MVS files into the RTC repository \(and getting them back out again\)](#)

zComponent project structure



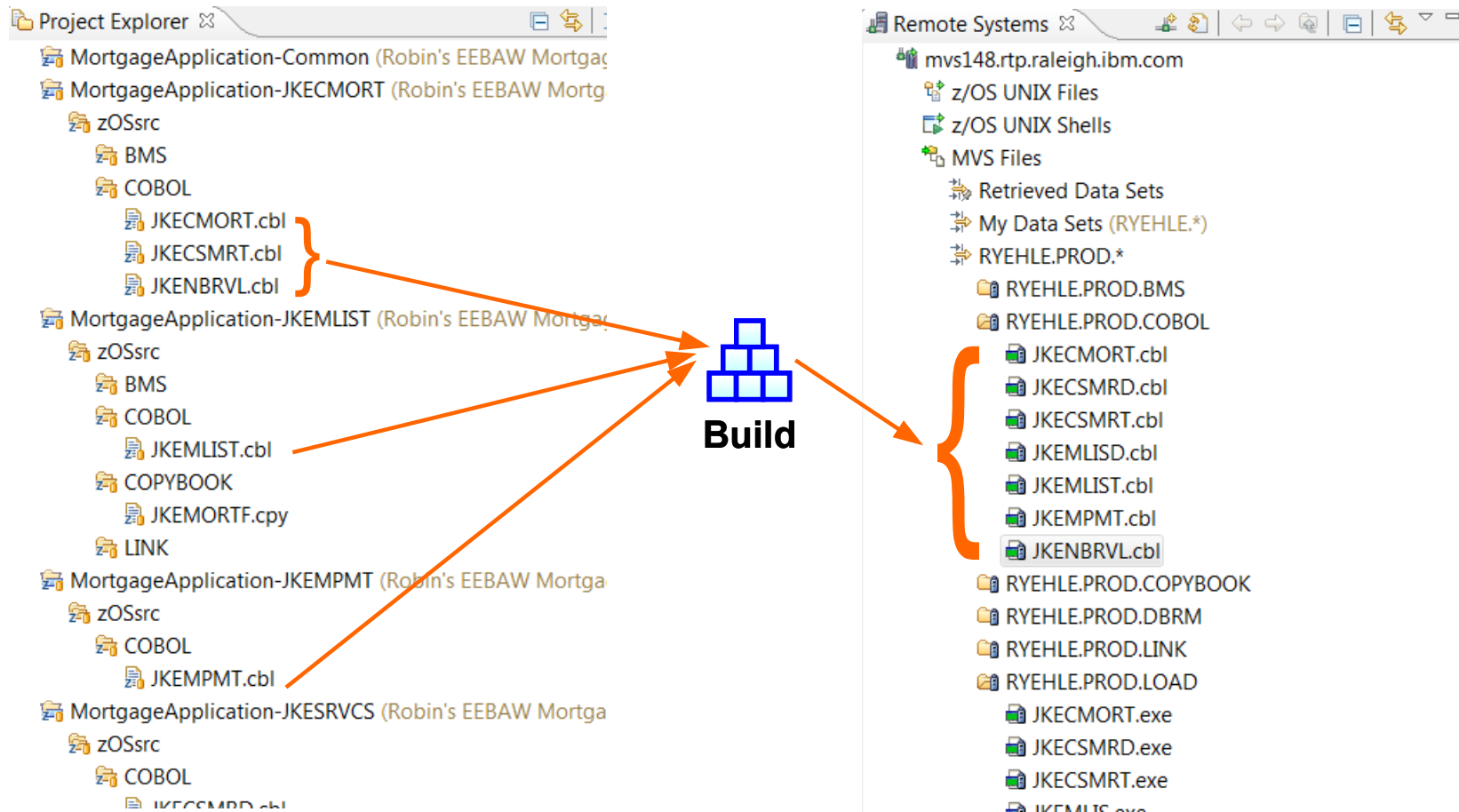
- Specialized eclipse project for loading and building z/OS artifacts on the host
- Rational Developer for System z local project
 - Appears in z/OS Projects view (RDz) automatically

zComponent Project

zFolder

zFile

zComponent project structure



Lab 2

Please complete **Lab 2**

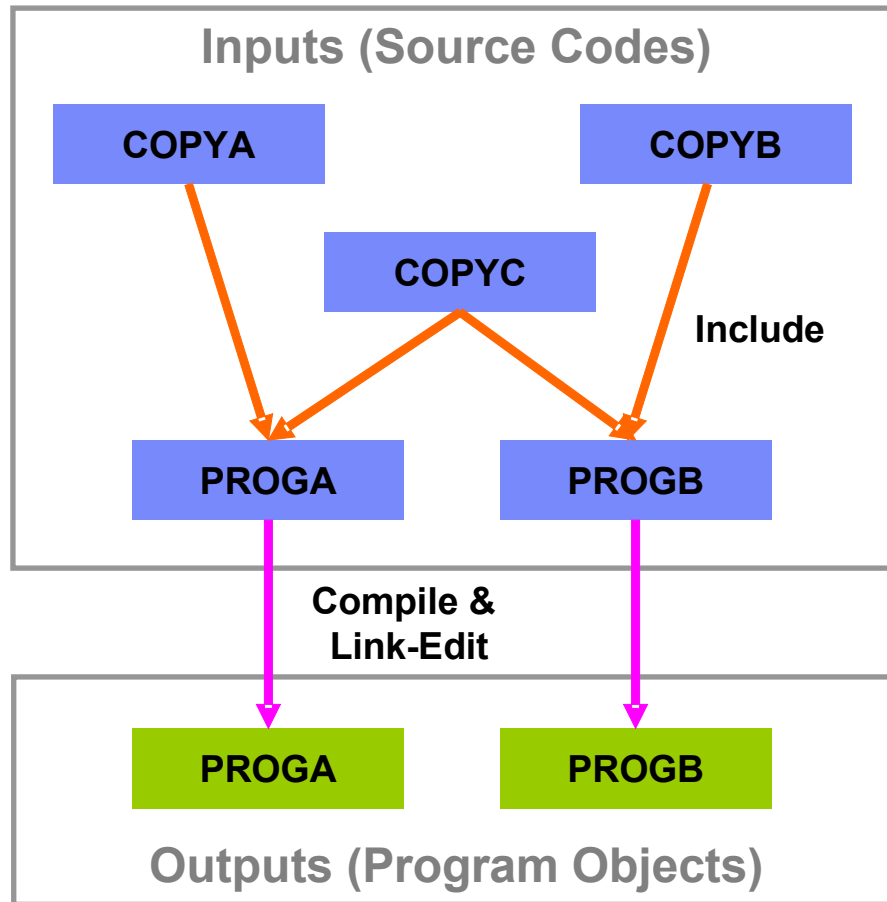
- Create a stream and component structure
- Import the sample mortgage application and explore the zComponent project structure
- Share the zComponent projects in the Rational Team Concert repository
- Associate the source with system definitions and deliver the projects

Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- **Lab 3: Migrating your build to Rational Team Concert**
- Lab 4: Promoting your changes from Development to Production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Dependency build

- Incremental build of modified and impacted sources



How does this work?

1. In the 1st build, PROGA and PROGB are built.
2. If no source code is changed, no outputs are re-built.
3. If PROGA or COPYA is changed, PROGA is rebuilt.
4. If COPYC is changed, both PROGA and PROGB are rebuilt.

Source code data collection

- Source code is scanned on a schedule and incrementally at build time
- Extension point available for contributing custom scanners
- Additional data can be added manually
- Streams individually toggled for scanning

Source Code Data Scanning

Stream Scanning Configuration

Select which project and team area streams to allow source code data scanning.

Streams:

- ☐ JKE Banking Integration Stream
- ☒ Mortgage Development Stream
- ☒ Mortgage Production Stream
- ☒ Mortgage QA Stream
- ☒ Mortgage Test Stream
- ☐ Production Stream
- ☐ QA Maintenance Stream

Source Code Data

Namespace: <http://www.ibm.com/xmlns/prod/rational/rtc/metadata/sourcefile/dependency/>

Jazz SCM Properties <<http://www.ibm.com/xmlns/prod/rational/rtc/metadata/sourcefile/scm/>>

Dependency Properties

The following properties were produced by the associated scanner and may be edited. To add properties or to create additional instances of existing properties use the Add button in the User-defined Data section.

dependencyLogical...	dependencyFileType	dependencyPath	dependencyRefere...
SQLCA	INCL	SYSLIB	SQL INCLUDE
JKENBRPM	INCL	SYSLIB	COPY
JKEMTCOM	INCL	SYSLIB	COPY
JKEMORT	INCL	SYSLIB	COPY

fileType:

language:

logicalName:

User-defined Data

User-defined data is not produced by the source code scanners and will persist even after resetting the scanner data.

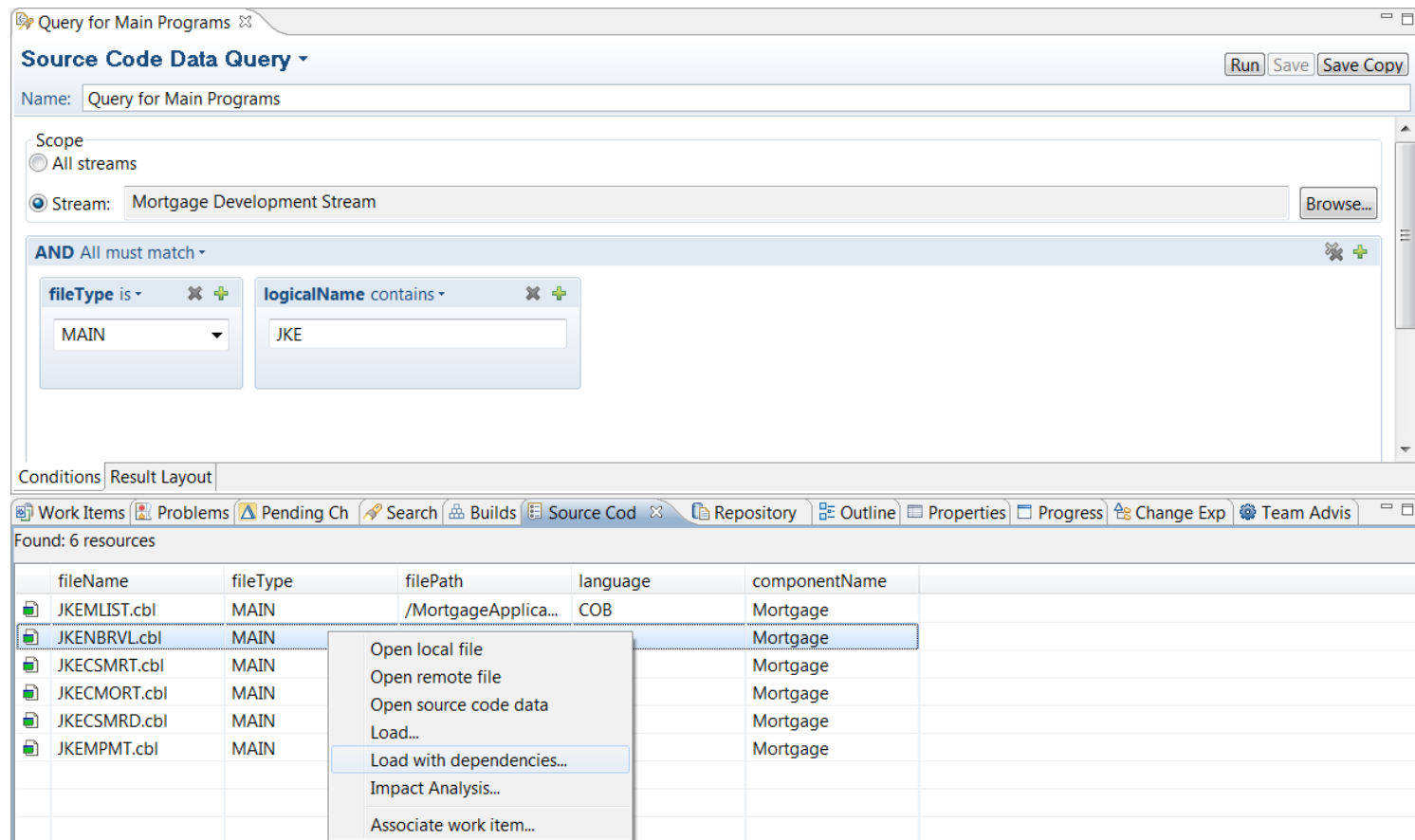
Fred: ☐ true ☒ false

Bob:

Overview

Source code data query

- Flexible query editor (similar to work item query editor)
- Query Source Code Data information for scanned and manual data
- Easily find your file in the stream/repository and sparse load (with dependencies)



Query for Main Programs

Source Code Data Query Run Save Save Copy

Name: Query for Main Programs

Scope

☐ All streams

☒ Stream: Mortgage Development Stream Browse...

AND All must match

fileType is

logicalName contains

Conditions Result Layout

Work Items Problems Pending Ch Search Builds Source Cod Repository Outline Properties Progress Change Exp Team Advis

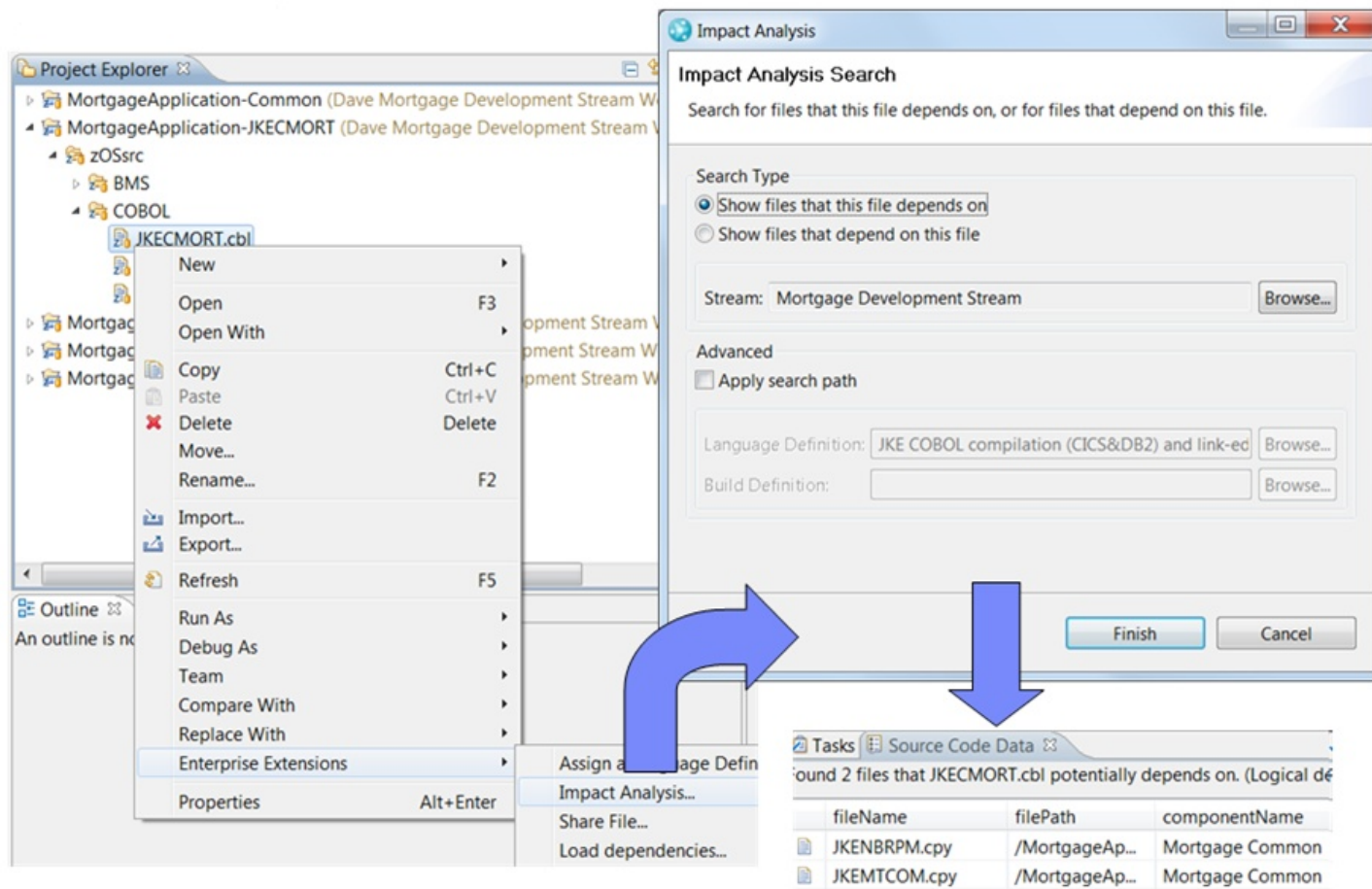
Found: 6 resources

fileName	fileType	filePath	language	componentName
JKEMLIST.cbl	MAIN	/MortgageApplica...	COB	Mortgage
JKENBRVL.cbl	MAIN			Mortgage
JKECSMRT.cbl	MAIN			Mortgage
JKECMORT.cbl	MAIN			Mortgage
JKECSMRD.cbl	MAIN			Mortgage
JKEMPMT.cbl	MAIN			Mortgage

Open local file
Open remote file
Open source code data
Load...
Load with dependencies...
Impact Analysis...
Associate work item...

Impact analysis

- What copybooks does my program depend on?
- What programs will be rebuilt if I change this copybook?
- Based on Source Code Data gathered information



Lab 3

Please complete **Lab 3**

- Review the translators created by the system definitions generator and create a new translator
- Review the language definitions created by the system definitions generator and create a new language definition
- Create a dependency build definition

Dependency build: advanced topics

- File loading
- Personal build
- Build subsets
- Full build of all programs
- Ignore changes
- Variable overrides
- Link-edit support
- DB2 BIND and CICS NEWCOPY
- Adding a custom scanner

File loading

- Minimum loading (default behavior): Only the programs being built and their dependencies are loaded on the build machine.
- Efficient loading: If a file has not changed since it was last loaded, it will not be loaded again.

File loading: advanced load options

▣ **Build Definition** ▾

ID: Project or Team Area:

Load Options

Specify file extraction details. Properties can be referenced using \${propertyName}.

Load directory:*

☐ Delete directory before loading

Resource prefix:*

☐ Load workspace to the load directory at the beginning of the build.

☐ Load workspace to the resource with the resource prefix at the beginning of the build.

☐ Create folders for components

If only some of the components in the workspace are to be built, exclude the ones that do not need to be loaded.

Components to exclude:

The components to exclude will be available as the build property "team.enterprise.scm.loadComponents".

Select load rule files describing how to load the build workspace.

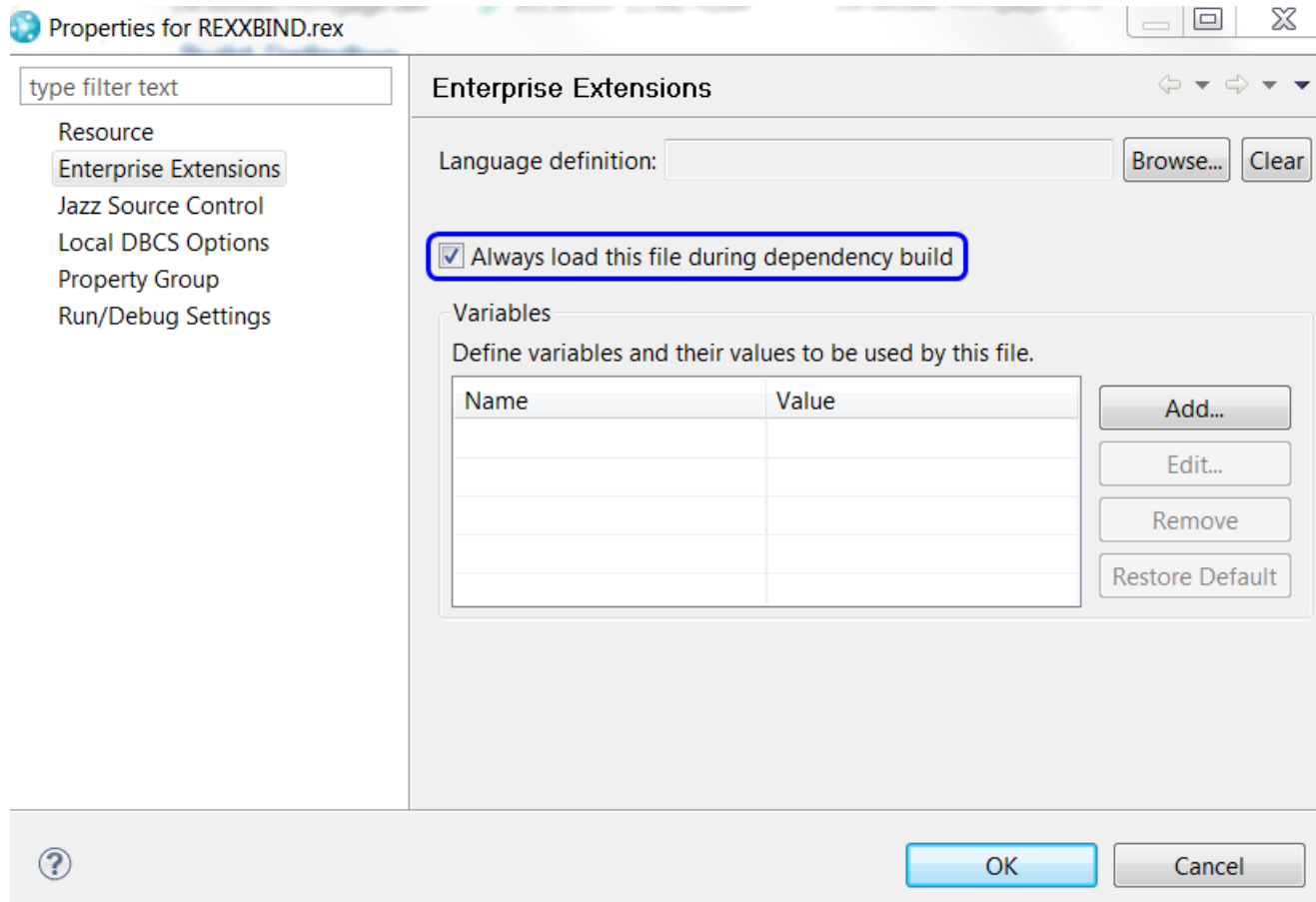
Load rules:

The load rules will be available as the build property "team.enterprise.scm.componentLoadRules".

- Options to load all USS files and all MVS files at the beginning of the build
- Load rules option allows you to specify a file that defines a subset of component files and folders for the build to load

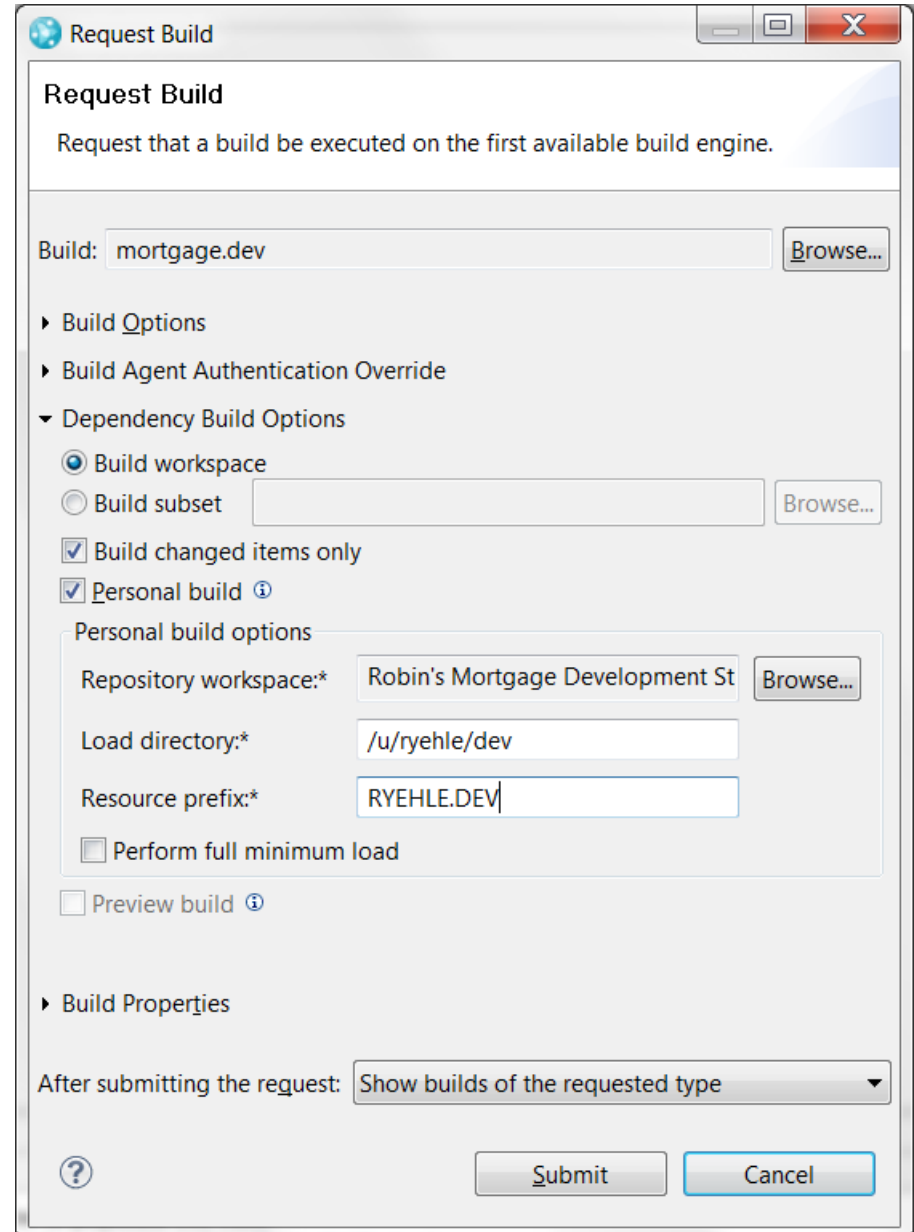
File loading: always load option

- You can mark individual MVS files to always be loaded during the build, regardless of if they are going to be built or are a dependency of a program being built (e.g., REXX scripts)



Personal build

- Build from files in developer's personal workspace
- Does not impact team build results
- Delta files are loaded to a personal sandbox (HLQ) and concatenated with team sandbox
- Full minimum load option loads all files needed to build modified and impacted programs to personal sandbox

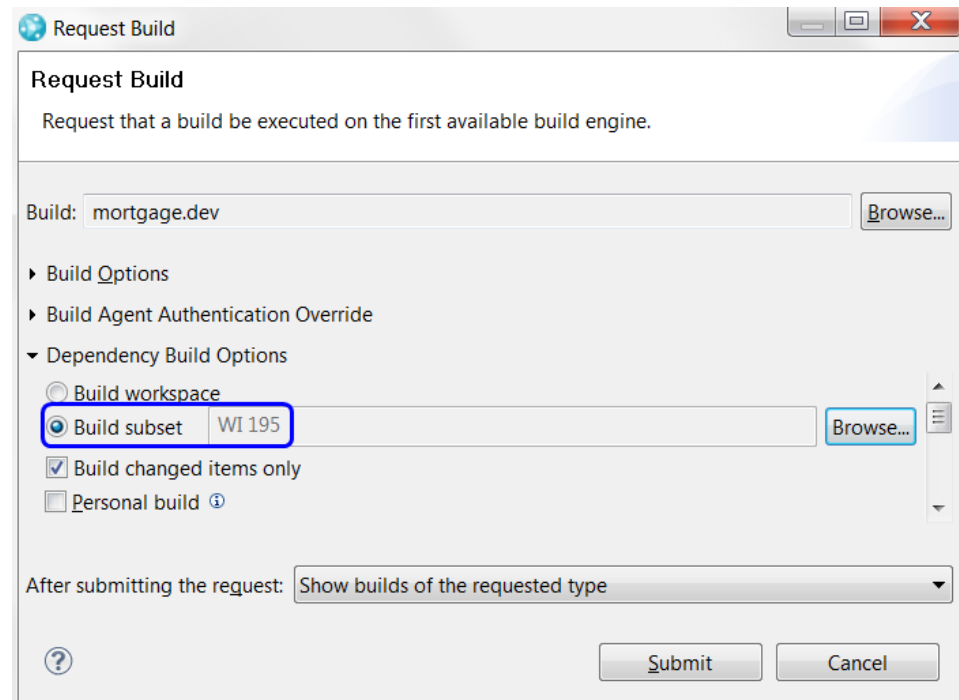
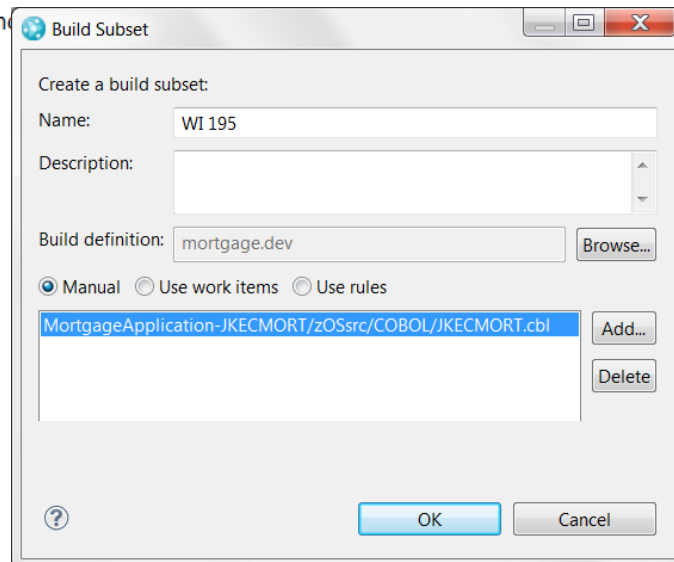
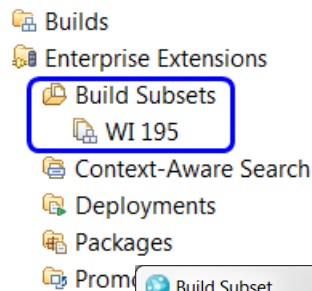


The image shows a 'Request Build' dialog box with the following fields and options:

- Build:** mortgage.dev (with a 'Browse...' button)
- Build Options:**
 - Build Agent Authentication Override
 - Dependency Build Options
 - ☒ Build workspace
 - ☐ Build subset (with a 'Browse...' button)
 - ☒ Build changed items only
 - ☒ Personal build ⓘ
 - Personal build options:
 - Repository workspace:* Robin's Mortgage Development St (with a 'Browse...' button)
 - Load directory:* /u/ryehle/dev
 - Resource prefix:* RYEHLE.DEV
 - ☐ Perform full minimum load
 - ☐ Preview build ⓘ
- Build Properties**
- After submitting the request:** Show builds of the requested type (dropdown menu)
- Buttons:** ? (help), Submit, Cancel

Build subsets

- Specify a subset of the buildable files in a workspace to be considered for build
- Specific to a build definition
- Select build subset in build definition or build request



Full build of all programs

- Option available to build all programs in workspace or subset, rather than just changed and impacted programs
- Option can be specified in build definition or build request
- Permission required to specify full build on build request

The image shows two screenshots from an IBM Rational Team Concert interface. The left screenshot shows the 'Build Definition' tab for a project named 'mortgage.dev'. Under 'Define build target', the 'Build workspace' radio button is selected. The 'Build changed items only' checkbox is checked and highlighted with a red rectangle. The 'Advanced' section shows the 'Conditional build' checkbox checked. The right screenshot shows the 'Request Build' dialog box. The 'Build' field is set to 'mortgage.dev'. Under 'Dependency Build Options', the 'Build workspace' radio button is selected, and the 'Build changed items only' checkbox is checked and highlighted with a red rectangle. The 'Personal build' checkbox is unchecked. The 'After submitting the request' dropdown is set to 'Show builds of the requested type'. The 'Submit' and 'Cancel' buttons are at the bottom right.

Build Definition

ID: mortgage.dev

General | Dependency Options

Define build target

Choose to build all programs or a custom subset of buildable artifacts.

☒ Build workspace

☐ Build subset

☒ Build changed items only

Advanced

Specify advanced build options for dependency building.

☒ Conditional build

Request Build

Request that a build be executed on the first available build engine.

Build: mortgage.dev

Build Options

Build Agent Authentication Override

Dependency Build Options

☒ Build workspace

☐ Build subset

☒ Build changed items only

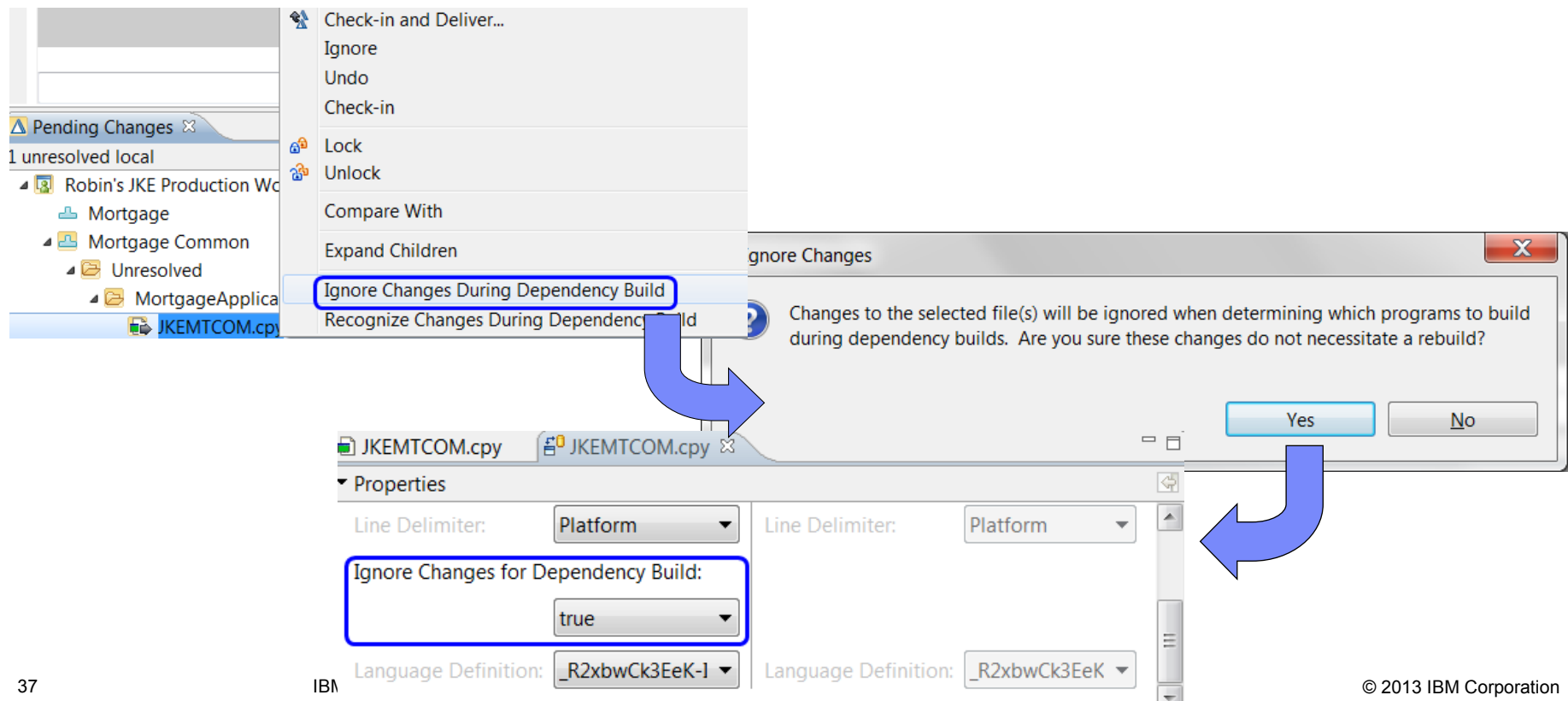
☐ Personal build

After submitting the request: Show builds of the requested type

Submit Cancel

Ignore changes

- Flag changes as non-impacting to prevent unnecessary rebuilds
- Optional permissions checks (configured as Deliver Phase 2 server precondition) to prevent unauthorized users from flagging changes as non-impacting
- Optional precondition on deliver (configured as Deliver client precondition) to prevent users from accidentally flagging an impacting change as non-impacting (users overrule failure on intentional non-impacting changes)



Variable overrides

- Variables can be used in default options field for called program translators and in the command field for ISPF/TSO command translators
- Default value for variable specified in the translator
- Overrides specified on individual files or in build definitions
- File level trumps build level override

The screenshot displays the 'Translator' configuration window for a PL/I compilation. The 'Name' field is set to 'PL/I compilation'. Under the 'Call Method' section, 'Called program' is selected. The 'Data set definition' is 'IBMZPLI'. The 'Default options' field contains '&LISTING OBJECT', with '&LISTING' highlighted by a blue box. The 'DD names list' is empty. Below, 'ISPF command or exec' and 'TSO command or exec' are both unselected. The 'Maximum return code' is set to '0'. A 'Variables' table at the bottom shows a variable named 'LISTING' with a value of 'LIST', where 'LISTING' is highlighted by a blue box.

In the background, the 'Explorer' view shows a project structure with files like 'DATAVARs.inc', 'GLOBAL.inc', and 'IBMJSO1.pli'. The 'Properties for IBMJSO1.pli' window is also visible, showing 'Enterprise Extensions' as a resource. The 'Enterprise Extensions' window shows the 'Language definition' as 'PL/I' and a table of variables with 'LISTING' set to 'NOLIST', where 'LISTING' is highlighted by a blue box.

Name	Value
LISTING	LIST

Name	Value
LISTING	NOLIST

Link-edit support: one object deck per load module

- Input of link-edit step is output of compilation step
- If a temporary data set is used for the object decks, mark the DD allocation as “Keep”

Language Definition Save

Name: Joe EEBAW COBOL compilation (CICS&DB2) and link-edit

General
Specify a language for this language definition. Optionally, you can specify file extensions to automatically associate files with this language definition.

Language: COBOL

File extensions:

Translators
Specify and order translators for this language definition. Translators run in this order during the build.

Joe EEBAW COBOL compilation (CICS&DB2) Add...

Joe EEBAW Link-edit Edit...

Translator Save

Name: Joe EEBAW COBOL compilation (CICS&DB2)

DD allocations:

DD Name	Data Set Definition	Member	Output Member Name	Keep	Output	Publish
SYSIN	<INPUT>	no		no	no	no
SYSLIN	Joe EEBAW Temporary file (object deck)	no		yes	no	no

Translator Save

Name: Joe EEBAW Link-edit


DD allocations:


DD Name	Data Set Definition	Member	Output Member Name	Keep	Output	Publish
SYSLIN	Joe EEBAW Temporary file (object deck)	no		no	no	no
SYSMOD	Joe EEBAW Program objects	yes	Same as input	no	yes	no

Link-edit support: several object decks per load module (I)

- When a program calls subprograms through static calls, the statically called subprograms need to be linked in the same load module as the main program. The linker, when using the autocall option, is able to find automatically the statically called object decks and include them in the load module by himself. Extra steps are necessary to cause the main program to be link-edited by the dependency build when a subprogram is changed.

1) Add a new dependency type to the language definition and associate it with the link translator


Language Definition

 Save

Name: Joe EEBAW COBOL compilation (CICS&DB2) and link-edit

Dependency Types

Specify the dependency types supported by this language definition. In addition, specify the translators that resolve each type.

Dependency type	Translators
COPY	Joe EEBAW COBOL compilation (CICS&DB2)
++INCLUDE	Joe EEBAW COBOL compilation (CICS&DB2)
SQL INCLUDE	Joe EEBAW COBOL compilation (CICS&DB2)
MACRO	Joe EEBAW COBOL compilation (CICS&DB2)
PROC	Joe EEBAW COBOL compilation (CICS&DB2)
LINK	Joe EEBAW Link-edit

Add...
 Edit...
 Remove
 Add All

General Scanners

Link-edit support: several object decks per load module (II)

2) Add source code data to each main program to indicate the object decks they depend on

Source Code Data



Namespace:

fileType:

language:

▼ User-defined Data

User-defined data is not produced by the source code scanners and will persist even after resetting the scanner data.

dependency



Add

Remove

dependencyLogical...	dependencyFileType	dependencyPath	dependencyReferenceType
JKENBRVL	OBJ	SYSLIB	LINK

Link-edit support: using a linkage card (I)

- When you have specific linkage instructions for a given program, then you should define an explicit link card as a member under SCM control. This link card will reference a dedicated link-edit language definition.
- You will need to manually specify the dependencies of your link file: all the object decks produced by programs under SCM control that are explicitly or implicitly needed to build the load module.
- **You implemented an example of this in Lab 3.

1) Add a dedicated link-edit language definition with a custom Link dependency type

Language Definition Save

Name: Joe EEBAW Link-edit

General
Specify a language for this language definition. Optionally, you can specify file extensions to automatically associate files with this language definition.
Language: Link edit

Translators
Specify and order translators for this language definition. Translators run in this order during the build.
Joe EEBAW Link-edit using linkage-editor deck Add...

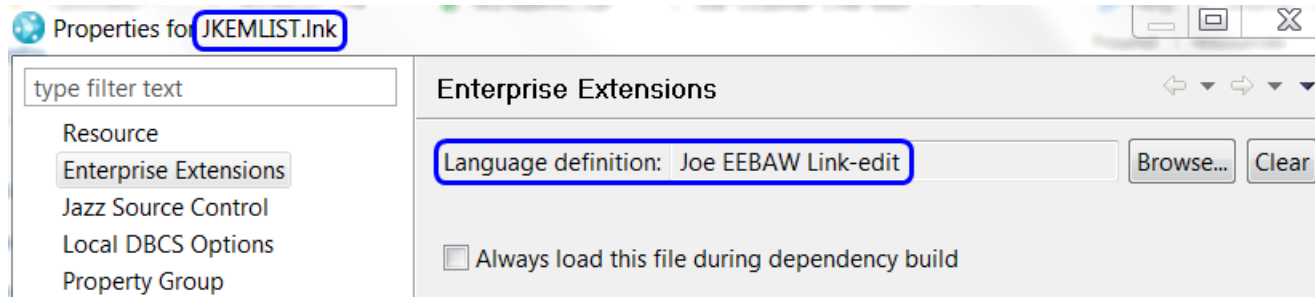
Dependency Types
Specify the dependency types supported by this language definition. In addition, specify the translators that resolve each type.

Dependency type	Translators	
Link	Joe EEBAW Link-edit using linkage-editor deck	Add...

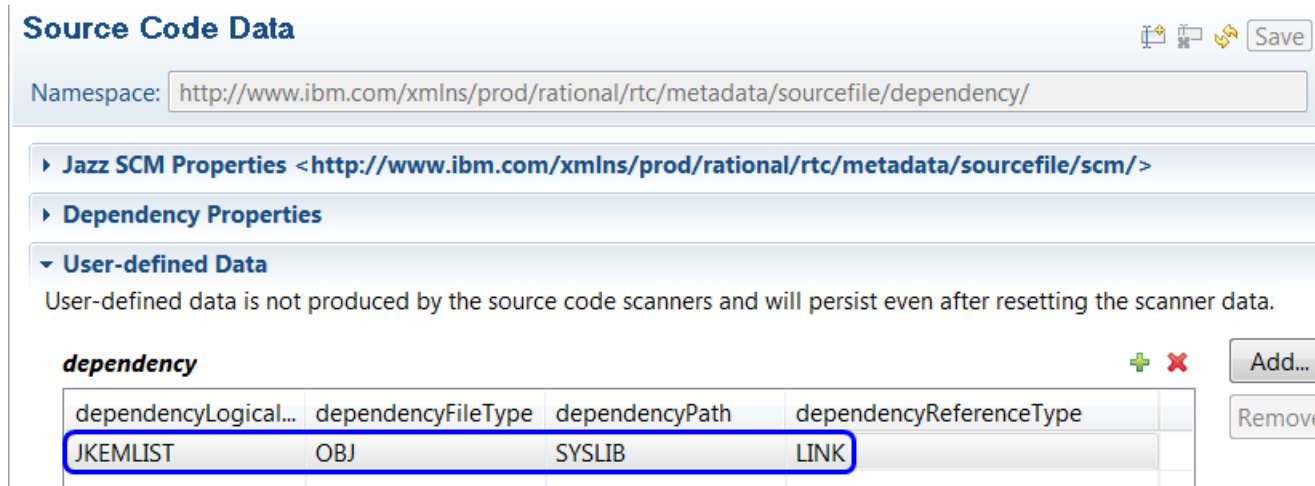
General | Scanners

Link-edit support: using a linkage card (II)

2) Assign the link-edit language definition to the link card



3) Add dependencies on the necessary object decks to the link card source code data



Link-edit support: references

- For more details on these and other link-edit scenarios, visit <https://jazz.net/wiki/bin/view/Main/LinkEditScenarios>
- To watch the improvements under investigation for link-edit support, subscribe to [145844: \[CCM\] Dependency build - Enhance link edit support](#)

DB2 BIND and CICS NEWCOPY

- Performed through the execution of custom REXX during or after a build process or as a post-promote or post-deploy command.
- These slides reference DB2 BIND, but the concepts are the same for performing a CICS NEWCOPY
- Option #1: DB2 BIND as a step in the build process
 - You can perform a DB2 bind as a step in your dependency build. The translator that calls the bind REXX script is integrated into the language definitions that include SQL pre-processing.

Translator

Name: Joe DB2 BIND

☒ TSO command or exec

Command/member: EXEC '\${team.enterprise.scm.resourcePrefix}.REXX(REXXBIND)' '\${db2.package} \${team.enterprise.scm.resourcePrefix}.DBRM@{source.member}'

Maximum return code: 0

Language Definition

Name: Joe EEBAW COBOL compilation (CICS&DB2) and link-edit

General

Specify a language for this language definition. Optionally, you can specify file extensions to automatically associate files with this language definition.

Language: COBOL

File extensions:

Separate extensions with a comma; for example, "cbl,cob".

Translators

Specify and order translators for this language definition. Translators run in this order during the build.

Joe EEBAW COBOL compilation (CICS&DB2)

Joe EEBAW Link-edit

Joe DB2 BIND

Add...

Edit...

Remove

DB2 BIND and CICS NEWCOPY, cont

- Option #2: DB2 BIND as a post-deploy command
 - If you are not building your programs directly into your runtime environment, you need to package and deploy the programs after you build them. As a result, you need to perform a bind on the programs in their target environment. Configure this bind as a post-deploy command.
 - Requires an additional REXX script to parse deltaDeployed.xml and determine what DBRM modules you need to bind

Deployment Definition ▾

ID: Project or Team Area:

Deploy pre-command:

Deploy post-command:

Rollback pre-command:

Rollback post-command:

Overview | Properties | z/OS Deployment | **Ant**

- Additional options: Perform BIND as a post-build or post-promote command. Gather the DBRM members you need to bind from artifacts of the build and promotion, generatedOutputs.properties and promotionInfo.xml.
- For a step by step guide on configuring option #1 or #2, see [Performing a DB2 bind with Rational Team Concert 4.0](#)

Adding a custom scanner

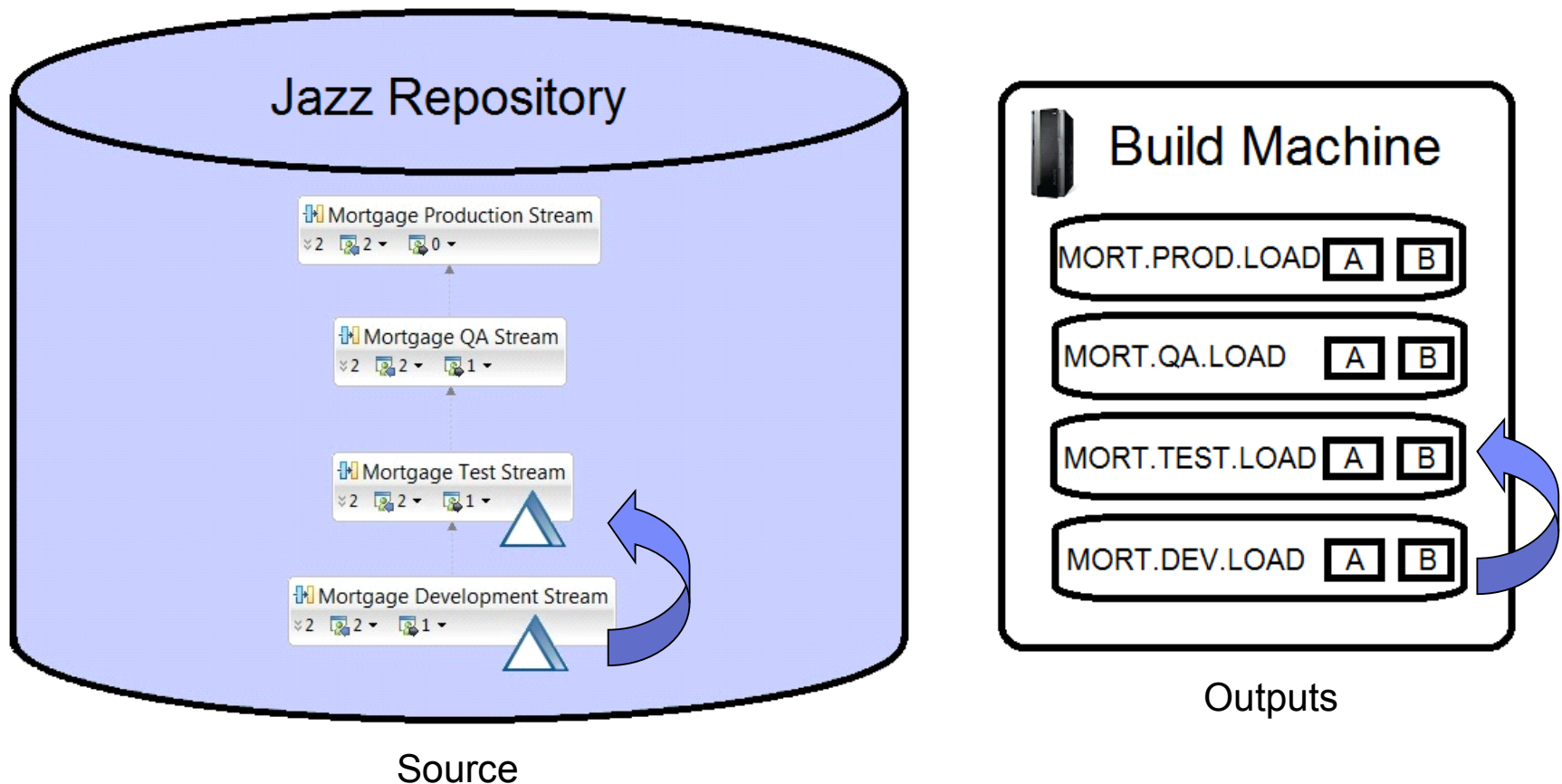
- Out-of-the-box Default Scanner for System z supports:
 - COBOL
 - PLI
 - Assembler
 - CA Easytrieve
- You can contribute your own scanners by implementing an eclipse extension
 - Reference: [Source Code Data Scanner API](#)
- Custom scanners will run in the same manner as the default scanner:
 - On a schedule (configurable), all selected streams are scanned
 - At the start of a build, the stream the build repository workspace flows with will be scanned
 - Full re-scans can be forced from the Source Code Data node in the Team Artifacts view (permission required)

Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- **Lab 4: Promoting your changes from Development to Production**
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

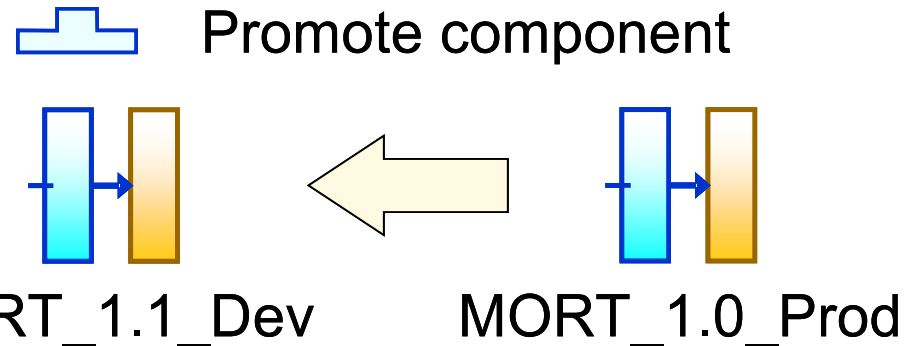
Promotion

- Purpose: Flow source code changes and build outputs through the development hierarchy
- Two styles: Component-based and work item-based

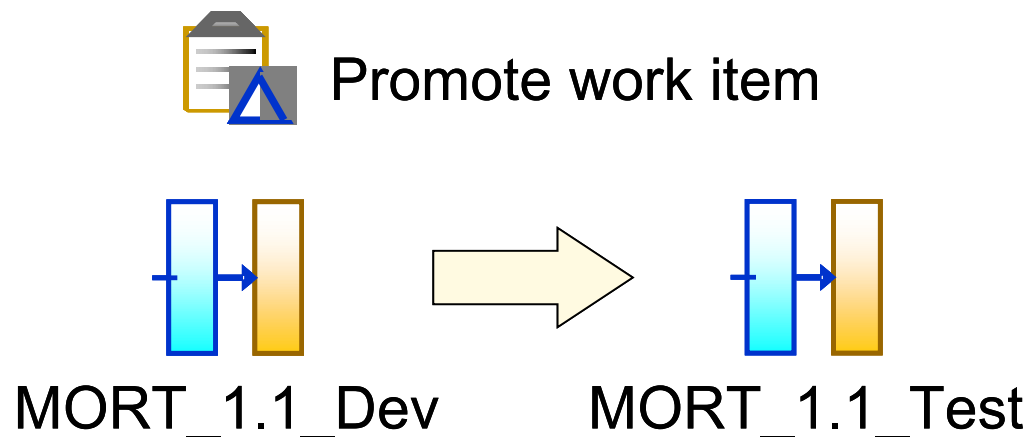


Promotion best practices

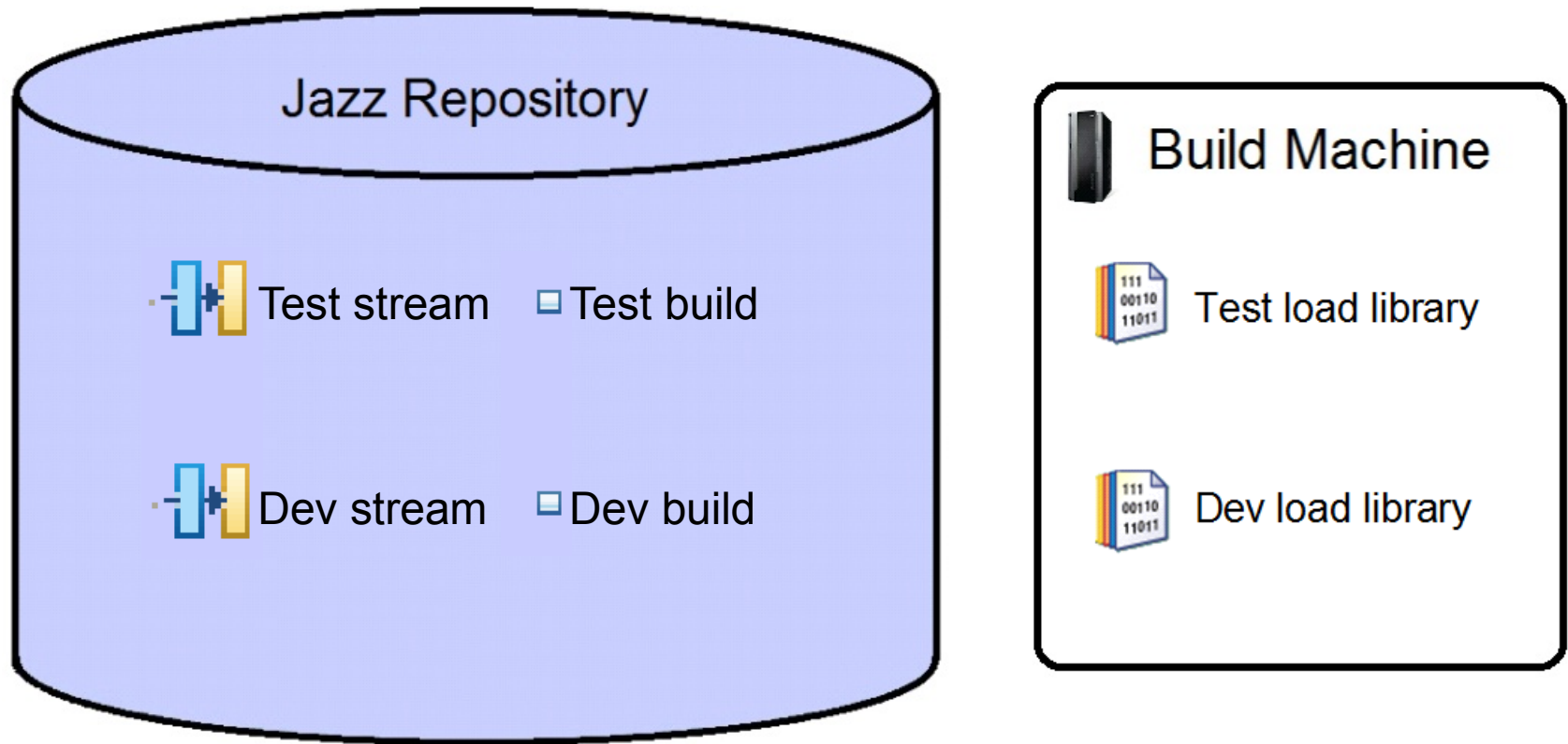
- Prime the development streams from production using component promotion



- Work item promote changes from development to test



Ingredients for promotion



Lab 4

Please complete **Lab 4**

- Create the development and QA streams and dependency builds
- Configure a “primer” promotion definition for seeding lower level streams and builds from the Production level
- Prime QA and Development using component promotion
- Configure promotions from Development to QA and from QA to Production

Promotion: advanced topics

- Work-item promotion: preconditions and follow-up actions
- Work-item promotion: promoting unmodified programs
- Work-item promotion: evaluating build outputs for promotion
- Work-item promotion: considerations for ignored changes

Work-item promotion: preconditions and follow-up actions

- Easily build process around work-item promotion with out of the box preconditions and follow-up actions, or add your own custom process extensions
 - Pre-condition:
 - Require Work Item States: Ensure work items are in a specified state before they are promoted. You could require specific approvals before allowing a work item to be saved in this state. You also could then query on the required state to find your work items ready to be promoted.
 - Follow-up actions:
 - Add Work Item Comment
 - Add Work Item Tags
 - Modify Work Item State
- The screenshot shows a configuration window for the "Promote Work Items (server)" operation. The title bar reads "Promote Work Items (server)". Below the title bar, there is a section labeled "Reports" which contains a table with several columns. The first column has a dropdown menu currently set to "Reports". To the right of the table, there is a blue button with a person icon. At the bottom of the window, a status message states: "The Promote Work Items operation is executed whenever an attempt is made to promote a selection of work items."

Promote Work Items (server)

Reports

The Promote Work Items operation is executed whenever an attempt is made to promote a selection of work items.

☒ Preconditions and follow-up actions are configured for this operation

☐ Final (ignore customization of this operation in child areas)

Preconditions (1 available):

Require Work Item States

Add...

Remove

Up

Down

Name: Require Work Item States ☐ Fail if not installed

Description:

Verifies that a work item can only be promoted if it is in one of the configured states.

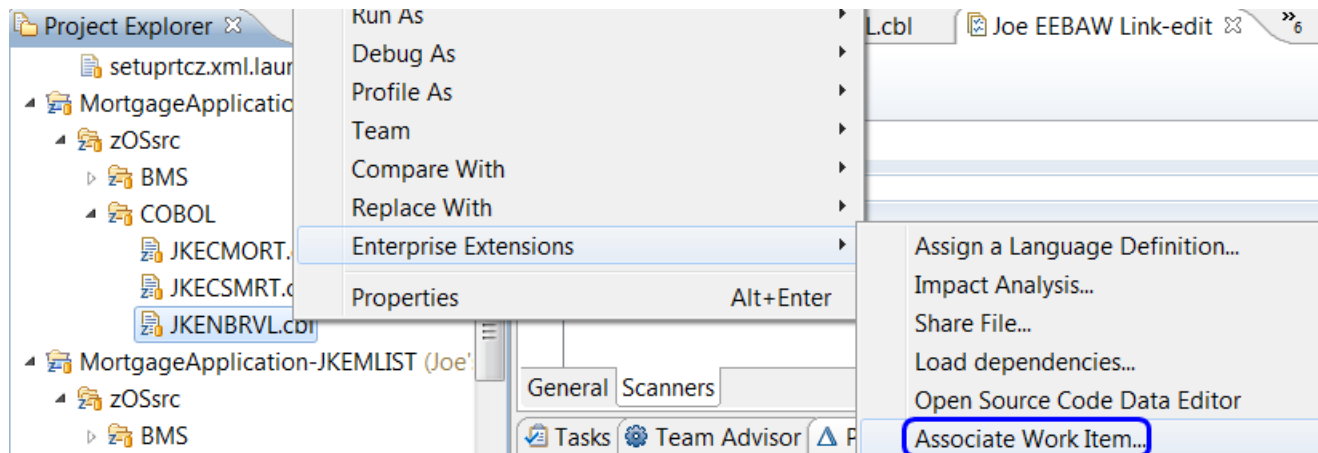
Build Definitions: joe.eebaw.mortgage.promote.

Category or Type	States
<input checked="" type="checkbox"/> Defect	Resolved
<input checked="" type="checkbox"/> Task	Done

Edit...

Work item promotion: promoting unmodified programs

- Only programs contained in promoted change sets are candidates for output promotion
- To promote the output of unmodified programs, associate the program directly with the work item being promoted. Directly associated buildable programs are candidates for output promotion.



Work-item promotion: evaluating build outputs for promotion

- Change sets must have been delivered to the source stream to be promoted to target stream
- Buildable files must be included in one of the selected work items' change sets or be directly associated to one of the selected work items for their outputs to be promoted
- For each buildable file to be promoted, perform build map validation:
 - Verify that the buildable file has an associated build map. If not, this buildable file has not been built successfully.
 - Verify that the buildable file has been built at the level being promoted.
 - If you are promoting an un-built change to a buildable file, the promotion will fail.
 - E.g. program is built in state 1, you deliver state 2, and then attempt to work item promote state 2
 - If you have changed and built the program since the change you are promoting, the promotion will fail.
 - E.g. program is built in state 2 and you attempt to work item promote state 1
 - For each internal (i.e. stored in the repository) input file in the build map, verify that the state of the input file in the build maps matches what the state of the input file in the Target stream will be when the promoted change sets are delivered
 - E.g., if A.cbl was built with state 3 of X.cpy, X.cpy must be at state 3 once the promoted change sets are delivered to the Target stream

Work-item promotion: considerations for ignored changes

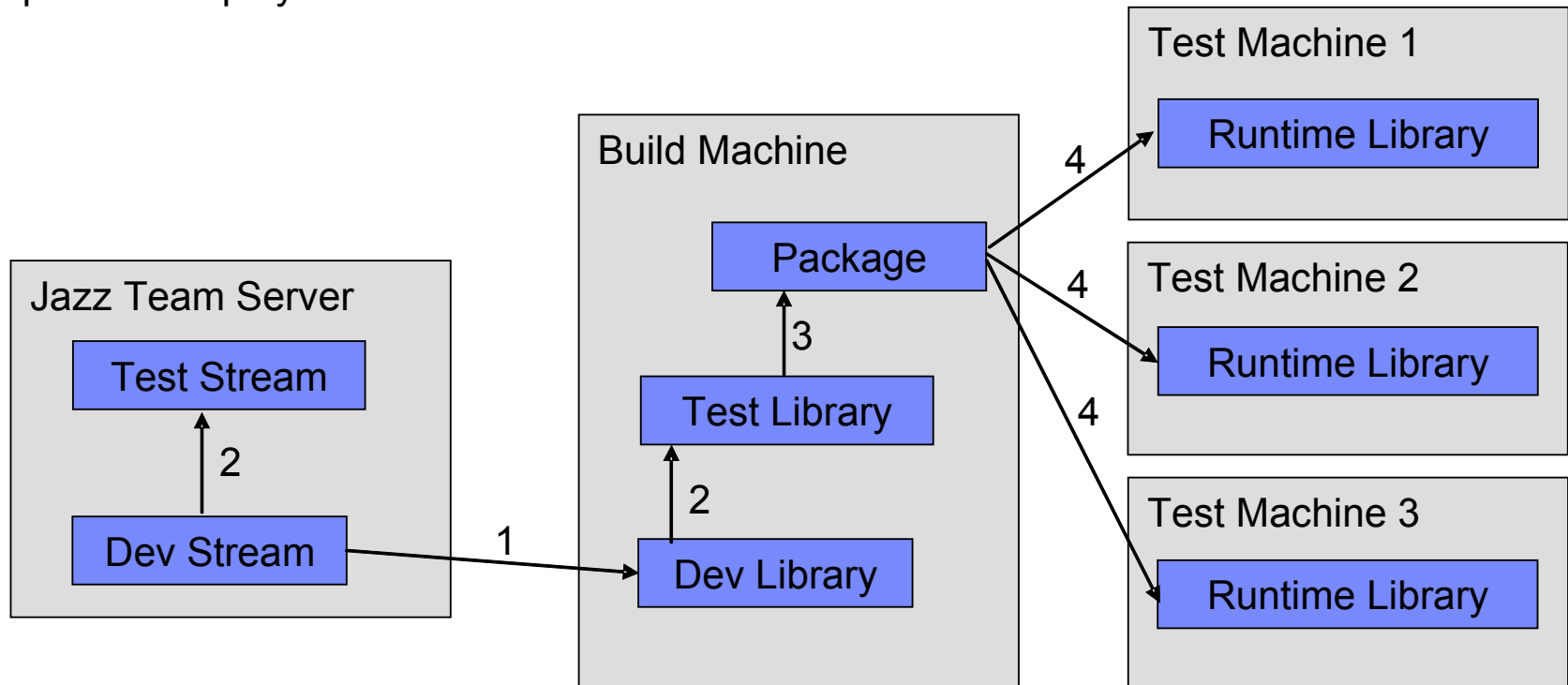
- Changes to files can be marked as “non-impacting” and as such will not cause programs to be rebuilt during a dependency build
- Option added in 3.0.1.2 to not fail the validation of build maps if the file whose state doesn't match the state at the target stream (post-source code promotion) is flagged as non-impacting
 - If A.cbl is built with a state of X.cpy that has NOT been promoted to the target stream, A.cbl's output can still be promoted if X.cpy's change is marked as non-impacting
 - If A.cbl is built with a back-level state of X.cpy, but the state of X.cpy in the target stream is marked as non-impacting, A.cbl's output can still be promoted

Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from development to production
- **Lab 5: Packaging and deploying your application**
- Lab 6: Performing an end-to-end verification of your development lifecycle
- Migration advanced topics

Deployment: big picture

- Dependency build runs on build machine. Source is loaded from Dev Stream and outputs are built in Dev Library.
- Promotion build runs on build machine. Source is promoted from Dev Stream to Test Stream and build outputs are copied from Dev Library to Test Library.
- Package build runs on build machine. Test Library build outputs are archived in a package.
- Deploy build runs on various test machines. Package is loaded to test machine and build outputs are deployed to runtime libraries.



Deployment: overview

- Three step process:
 - **Package:** on the build machine, zip the programs you built or promoted (creates a zip file on USS containing MVS data set members)
 - **Load:** copy or FTP the zip file to a USS directory on the machine where the programs will be deployed
 - **Deploy:** restore the built programs from the zip file to the runtime environment (MVS data sets)
- Specifying the package contents:
 - **Shiplist based:** list the resources you want to include in the zip
 - **Work item based:** choose the work items you want to package, and the outputs generated from change sets associated with those work items will be included in the zip.
 - Can be supplemented with a shiplist
- Additional options:
 - Restore mapping table: allows you to change the name of the MVS data set you deploy to on the target machine
 - Rollback: capability to do an n-1 rollback to previous state
 - Pre and post commands: exit points for custom processing before and after package, load, and deploy

Lab 5

Please complete **Lab 5**

- Create a definition to package your build outputs at QA level
- Perform the configuration steps to be able to deploy the package contents in the QA runtime environment
- Review additional options and how these concepts would be extended to deploy in Production environment

Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from development to production
- Lab 5: Packaging and deploying your application
- **Lab 6: Performing an end-to-end verification of your development lifecycle**
- Migration advanced topics

Lab 6

Please complete **Lab 6**

- Create a change at Development level and build it
- Promote the changed source and built outputs and deploy to QA

Agenda

- Overview
- Installation and setup
- Lab 1: Planning your Rational Team Concert solution
- Lab 2: Sharing your source members in Rational Team Concert
- Lab 3: Migrating your build to Rational Team Concert
- Lab 4: Promoting your changes from development to production
- Lab 5: Packaging and deploying your application
- Lab 6: Performing an end-to-end verification of your development lifecycle
- **Migration advanced topics**

Migration advanced topics

- Avoiding the initial full build
- Code pages and line delimiters
- Handling non-roundtrippable characters
- Build agent security
- Cross-platform build
- Saving your listings and finding the right one
- Determining what changes are in your environment

Avoiding the initial full build

- First dependency build request will build all programs. Allowing this full build to occur to prove out that everything is properly configured and can be built is a best practice.
 - Reality says you often can not or refuse to perform this full build for a number of reasons.
- With any of the following approaches, it's essential that you test out the full cycle: build at production, component promote down, deliver various types of changes - e.g., a main program, a copybook, a BMS map, an ignored change, etc - at development, build the changes, and work item promote the changes up on a small subset of your programs to ensure that your solution works for your environment and situation, before importing and building your full collection of programs.

Avoiding the initial full build, continued

- **Issue #1:** I don't want a whole new set of production-level modules, when my current production modules are already tested and proven.
- **Response:** No problem! Simply perform the production dependency build to prove out your build setup and generate your build maps, and then throw away all of the build outputs and replace them with your current production modules. This is actually the recommended migration path. You will simply need to use the “Skip timestamp check when build outputs are promoted” option when you are component promoting down (but don't skip it when you work item promote back up). Also ensure that your dependency builds are configured to trust build outputs. This is the default behavior, and allows the dependency build to assume that the outputs on the build machine are the same outputs that were generated by a previous build. When this option is turned off, the dependency build checks for the presence of the build outputs and confirms that the timestamp on each output matches the timestamp in the build map. A non-existent build output or a mismatched timestamp will cause the program to be rebuilt.

Avoiding the initial full build, continued

- **Issue #2: Some of my programs need changes before they can be built, and it's not feasible to do all of that work up-front before the migration.**
- **Workaround:** Assign your unbuildable programs a language definition with no translator. RTC will not consider these programs buildable and they will be ignored during dependency build. When you are ready to update the programs, assign them a proper language definition at that time. You can also use NO language definition on your unbuildable program if you're not using default language definitions (i.e. language definitions assigned based on file extension). In this case, the file will also not be scanned.

Avoiding the initial full build, continued

- **Issue #3: All those copies of outputs at each level in my hierarchy are just taking up space. I don't want them there.**
- **Workaround:** You can modify the promotion script to promote the build maps but not copy the outputs themselves. Again, ensure that trust build outputs is true (default) in your dependency build definitions. If you are building at a level where you don't have outputs, ensure that your production libraries are included in the SYSLIB in your translators.
 - Follow these steps to utilize this workaround:
 1. Copy generatedBuild.xml from a promotion result to the host.
 2. In the Promotion definition, on the z/OS Promotion tab, choose "Use an existing build file".
 3. Specify the build file you created in step 1.
 4. For build targets, specify "init, startFinalizeBuildMaps, runFinalizeBuildMaps" without the quotes.

Avoiding the initial full build, continued

- **Issue #4: I refuse to build all of my programs. That's ridiculous and way too expensive.**
- **Workaround:** Seed the dependency build by creating IEFBR14 translators. This will give you the build maps you need without actually building anything. Then switch to real translators. There is a major caveat here: Indirect dependencies are not handled automatically until the depending program is actually built. For example, if you have a BMS map that generates a copybook that is included by a COBOL program, the dependency of the COBOL program on the BMS map is not discovered until the COBOL program actually goes through a real build. If you can accept this limitation, one approach to this workaround is as follows:
 1. Create two sets of translators: your real translators, and one IEFBR14 translator per original translator to make sure there are no issues with SYSLIBs changing when you switch from IEFBR14 to real translators.
 2. Use build properties in your language definition to specify translators, and set those properties in the build definition.
 3. Request the build with the properties pointing to the IEFBR14 translators. Everything “builds” but no outputs are generated.
 4. Change all of the translator properties in the build definition to point at the real translators.
 5. Request another build and see that nothing is built.

This approach again requires that we trust build outputs so we don't rebuild based on none of the load modules listed in the build maps actually existing.

Code pages and line delimiters

- Files are typically stored in the Jazz repository in UTF-8; files on z/OS are typically stored in EBCDIC.
- When we zimport, we convert from EBCDIC to UTF-8. When we load the files back out (zload, build, etc), we convert from UTF-8 to EBCDIC.
- To avoid the conversion on zimport, specify -b for binary. In that case, we set the line delimiter to none and do no conversion. Otherwise, we convert and set the line delimiter to platform. We determine which encoding to use for the conversion based on the ZLANG environment variable. If ZLANG is set, use it. If ZLANG is unset, use the system's default encoding.
- Files with line delimiter of none are not converted when loaded back out. Otherwise:
 - To determine which encoding to use when loading to MVS, we:
 - Check the mvsCodePage versionable property of the file. If present and non-blank, it is used.
 - If no mvsCodePage, default to the value of the ZLANG environment variable, if ZLANG is set.
 - If the mvsCodePage property is not present, and ZLANG is unset, default to using the system's default encoding.
 - To determine which encoding to use when loading to USS, we:
 - Check the mvsCodePage versionable property of the file. If present, and non-blank, it is used.
 - If the mvsCodePage property is present but blank:
 - If ZLANG is set, use the value of ZLANG.
 - If ZLANG is unset, default to the system's default encoding.
 - If the mvsCodePage property is not present, load the file as it is in the repository — no conversion is done.

Code pages and line delimiters, continued

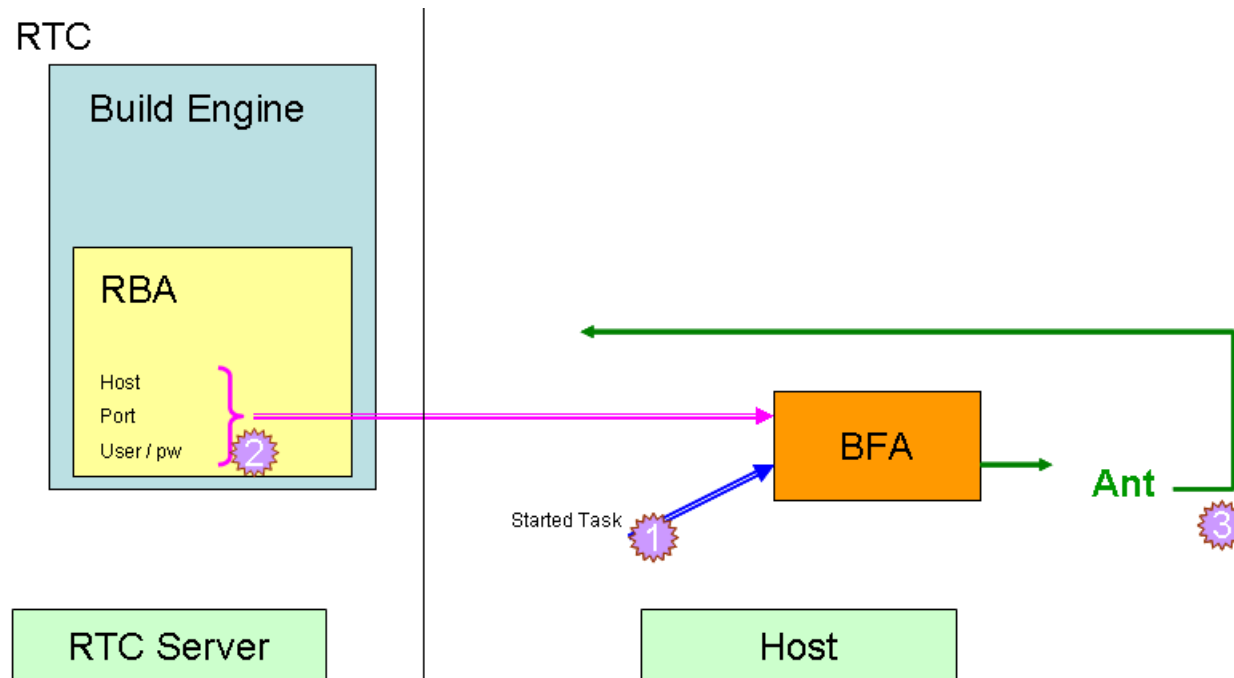
- Be aware of your client code page settings and potential unwanted side effects
- Ex 1: You create a file containing square brackets from your emulator configured with code page IBM-037. When the zimport on this file is performed, the default system setting of code page IBM-1047 is used. Because the encodings of square brackets do not match between IBM-037 and IBM-1047, these “variant” characters are corrupted in the process.
 - Workaround: Either change the square brackets to match the system's default encoding, or use the ZLANG environment variable to alter the codepage used by zimport.
- Ex 2: Windows eclipse clients have a default text file encoding of Cp1252. If your project and/or file is not set specifically to a UTF-8 encoding (and you do not change your default text file encoding in your client), you will not be able to save changes to your file if it contains characters that are not mapped in Cp1252. In 4.0.2, zComponent projects will be created with a project-level encoding of UTF-8 during zimport ([201615: zimport should set project-level encoding of UTF-8 on zComponent projects it creates](#)).
 - Workaround: Manually set UTF-8 as the text file encoding at either the project or Eclipse workspace level.

Handling non-roundtrippable characters




- “Non-roundtrippable” refers to characters that cannot be converted from EBCDIC to UTF-8 (to store in the SCM) and back to EBCDIC (to load back to MVS for build or edit).
- For example, your source files may include strings containing control characters such as a Carriage Return (0x'0D' in EBCDIC) for printing or any other embedded hex code. Your zimport will fail in this case if you do not use binary mode, because the perceived line delimiter in the middle of the line is inconsistent with the rest of the file (recall the MVS files are record length-based and do not contain line breaks at the end of each line). Other special characters may not break zimport but will still get mangled when roundtripped.
- Limitations on files containing non-roundtrippable characters:
 - Must be zimported using the binary option
 - Must be viewed and edited using the ISPF client. Eclipse client cannot handle these files.
 - RDz can be used to edit files directly on MVS after they are loaded using the ISPF client or zload.
 - Compare and merge capability is supported only in the ISPF client or using RDz.

Build agent security: three user IDs involved

- 1 TSO user that launches the BFA on the host
- 2 TSO user defined in the Build Engine and used to call the BFA to process a build request
- 3 Jazz user used by RTC Ant tasks launched by the BFA to connect back to the RTC server



Build agent security: considerations for user IDs

-  **1 TSO user that launches the BFA on the host**
 - Can be UID 0 (super-user) or regular user (with magic_login settings)
 - Has consequences on authority under which build runs (see next slide)
-  **2 TSO user defined in the Build Engine and used to call the BFA to process a build request**
 - Password should not expire (or process needs to be considered)
-  **3 Jazz user used by RTC Ant tasks launched by the BFA to connect back to the RTC server**
 - Configured with environment variables in the shell script startbfa.sh used to start the BFA
 - JAZZ_USER
 - JAZZ_PASSWORD_FILE
 - Password is encrypted using the BLZBPASS JCL provided (see info center). Requires running the JBE on z/OS.
 - Password file should not be accessible by all even though it is encrypted. Only prevents casual observation.
 - Must be defined as a member of the project area to have permission to save build result and save build engine (configured in process using Roles) and read access to the build workspace
 - Must have JazzUser repository permissions
 - Must have a sufficient Client Access License assigned (Build System at minimum)

Build agent security: under what ID does the build run?

- Reference: <https://jazz.net/wiki/bin/view/Main/ZosBuildAgentSec>

1

TSO user that launches the BFA on the host

2

TSO user defined in the Build Engine and used to call the BFA to process a build request

- The TSO user ID under which the build runs is dependent on several factors:
 - If user 1 is a superuser:
 - Build will run under authority of user 2 (regardless of if user 2 is a superuser or non-superuser)
 - Build agent authorization overrides (on the build request) can be used to allow users to submit under their own authority (this can be forced by specifying a dummy ID for user 2)
 - If user 1 is a non-superuser:
 - Configure the magic_login option in the build agent and specify user 2 (i.e. build engine definition user must match magic_login user).
 - Build will run under authority of user 1. User 2 is used only to authenticate.
 - Build agent authorization overrides are not possible.
 - If you do not configure magic_login, no authentication is performed and builds will run with the authorization of user 1. This will only work if you do not specify a password on the build agent/engine definition. You can not override.
- The user ID under which the build runs must have authority to:
 - Write to the data set HLQ specified in the build definition
 - Write to the load directory (USS folder) specified in the build definition
 - Write to the USS folder SCM_WORK/SCM configured in startbfa.sh

Cross-platform build

- One approach:
 - One RTC build definition that uses the Jazz Build Engine to build the distributed portion of your application
 - One RTC build definition that uses the Rational Build Agent to build the mainframe portion of your application
 - Depending on the complexity of your build, either:
 - Use a third RTC build to kick off the distributed and mainframe builds above
 - Ensure this build is serviced by a different engine than the definitions above
 - Reference: [A simple build chaining scenario](#)
 - Or, for more complex scenarios, use Build Forge to coordinate the distributed and mainframe builds above
- **Note:** You can configure your dependency build repository workspace to only include the components containing host-based source code

Saving your listings and finding the right one

- Listings are published with your dependency build results
 - Configuration options: Don't publish, publish only on error, zip logs (100 per zip)
- Build results and listings can be pruned to only save the latest good listing for each program
 - Reference: [Custom build result pruner](#)
- To locate your listing:
 - Find your file in the stream (Repository Files view) and right-click>Show History
 - Open the work item associated with the most recent change
 - In the Links tab, go to Included in Builds and open the build result for the level in which you build (e.g., if you compile in Dev and promote to QA, you would check the Dev build)
 - Find your listing on the Logs tab of the build result
 - Note: This approach only works when you modify the program itself. For programs that are rebuilt due to a changed dependency, you would need to manually have associated the program with the work item that changed the dependency.

Determining what changes are in your environment

- By having a stream that represents each level of your hierarchy (Development > Test > QA > Production), it is possible to know what changes are in your environment by simply checking for the change sets in the corresponding stream
 - Ex 1: You want to know if change X is in Production. If X was ready for Production, you would have promoted it from QA to Production and deployed it to your Production runtime environment. Therefore, you can simply search for the change set in the Production stream.
 - Ex 2: You want to know what changes in QA have not gone to Production yet. You can simply compare the two streams and get the complete listing of change sets that are in QA but not in Production.
 - Note there will of course be a window between the time you promote and the time you deploy that a change will be in the target stream but the built outputs will not yet be in the runtime environment.

Questions



Grazie धन्यवाद *Merci* ありがとうございます *Obrigado* 多谢
ITALIAN HINDI FRENCH JAPANESE BRAZILIAN PORTUGUESE SIMPLIFIED CHINESE

Thank You

多謝 Gracias Спасибо நன்றி ஸபரீம் *Danke* شكراً
TRADITIONAL CHINESE SPANISH RUSSIAN TAMIL THAI GERMAN ARABIC

We appreciate your feedback.
Please fill out the survey form in order to improve this educational event.