Git guide for Rational Team Concert users

Correspondence of source code management architecture, concepts, terms

Andrew Freed co-authorJason Brown
delivered 2016 November 24

As a Rational Team Concert user, you know that Rational Team Concert is a robust and powerful integrated source code management system. However, it does not necessarily mean that Rational Team Concert is perfect for any software development project or team. This article is written for Rational Team Concert users who want to learn about Git like us and use Git for new projects. In this article, we compare the basic functions and enhancements of these two systems so that we can see the similarities and differences between Git and Rational Team Concert. Also, let's see what kind of source control pattern the general Git flow corresponds to in Rational Team Concert. Finally, I will point out some pitfalls I have to avoid in transitioning from Rational Team Concert to Git.

Basic functions and support

Both Rational Team Concert and Git provide basic support and advanced support for source code management, but the handling of source control on each system is somewhat different. For example, although both Git and Rational Team Concert support **distributed development** (that is, they can collaborate without requiring developers to be in the same location), Rational Team Concert users can connect to a central server and make changes While collaborating by sharing, Git users mainly work on local clones.

Basic functions include **code storage** to support team collaboration and version control . Within Rational Team Concert, developers create repository workspaces mirroring their local sandboxes on the server side. For each developer's repository workspace, the code is checked in by the automatic check-in function. This automatic check-in occurs every time the developer performs a save operation within the Eclipse IDE.

In Git, users store branches of code on their local machine and work on those branches. And when the work is completed, it pushes the completed code to the remote branch. Compared to using Rational Team Concert's automatic check-in function, the push to the remote branch is done discretely. Therefore, Git users generally do not push changes frequently like Rational Team Concert users.

Organizing the code is also a basic function of source code management. Rational Team Concert expresses the code the team is working on as a stream. Streams can be further divided into logical groups called components. You can use either streams or components to set access privileges.

Git does not have anything equivalent to the components of Rational Team Concert. As a matter of fact, Git's repository is fairly small and the team uses multiple repositories. In general, one repository is used for each software component, or for each group of related components.

IDE integration

Rational Team Concert has a fully integrated IDE that supports work item tracking, project planning, running builds, user management and process management. As an all-in-one solution, Rational Team Concert makes it easy to customize all these elements and integrate them into the workflow. In addition, you can integrate tools such as Rational Requirements Composer to centralize requirements and Rational Quality Manager to track tests. The features of Rational Team Concert are primarily driven by rich clients and there are web clients supported by the server installation environment. Some developers prefer to use the command line client to manage the source code, but most Rational Team Concert users are working within the Eclipse IDE.

Git itself has no function other than source code management. The size of the core Git binary is small, and it is intended that developers can efficiently manage source code from the command line interface. However, for developers who want to work with the graphical interface, Git can also use extensions such as EGit and TortoiseGit. Developers who require a repository front-end for project management can also choose GitHub or BitBucket. These two Git-based solutions provide features such as user management and problem tracking.

Comparison of source code management architecture

The main difference between Rational Team Concert and Git is that in Git, users work in a code repository hosted on their machine. Therefore, while Git can handle offline development scenarios, there are no scenarios for Rational Team Concert users offline development. This section describes the basic architecture and components of each source code management system and clarifies the differences between the two systems.

Figure 1 shows the architecture of Rational Team Concert's source control management system. This figure shows the Eclipse workspace on the local workstation and the repository workspace and stream on the remote server.

Rational Team Concert's SCM architecture

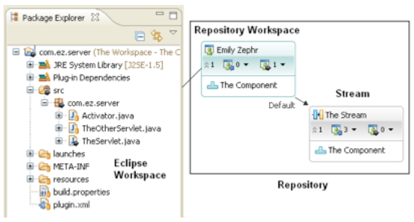
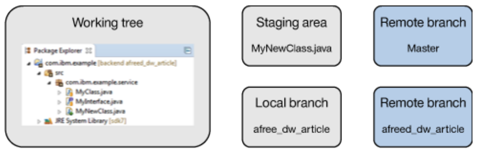


Figure 2 shows the architecture of Git corresponding to the above. This figure shows the working tree and the local branch in the local workstation and the remote branch on the remote server.

Git's SCM architecture



In Table 1, the meanings of the components of the two source control systems are clarified and compared.

Components that compose the architecture of Rational Team Concert and Git, respectively

Comparison of architecture and deliverables

Rational Team Concert	Git	Note
Repository	Repository	Within either system, the master code store is called the repository. Within Rational Team Concert, only copies of the complete repository are kept on the server. Within Git, the user clones the entire repository locally.
sandbox	Working tree	Local files checked out by developers and local changes that have not yet been checked in / committed
stream	branch	The work flow of shared code is called a stream within Rational Team Concert and it is called a branch within Git. A simple flow consists of only one integrated stream or branch. Within Git, this branch is named Master by default.
Repository Workspace	Feature branch (remote)	The repository workspace of Rational Team Concert is essentially an individual developer branch stored on the central server. Git users use branches much wider than that. For example, if you create a branch to support only one feature, merge that function into the integrated branch, then delete the original branch, and so on. Git has a local branch and a remote branch and uses "local" branches to "track" remote branches.
Not applicable	Feature branch (local)	Git supports offline development scenarios more flexibly than Rational Team Concert. Git users commit changes to arbitrary branches by a local commit operation. Then, when you work online, push the committed changes to the remote branch.
component	Not applicable	Rational Team Concert supports logical isolation of code within streams by means of components. Git does not have a concept corresponding to a component. Git users tend to divide development into multiple repositories.
Baseline / snapshot	tag	Baselines, snapshots, and tags mark the known level of code and will be the base for creating branches later. As an example, these artifacts are created to create "release 1.0".
Change set (active)	Staged change	It is a variable set consisting of one or more changes to the deliverables.
Change set (completed)	commit	It is an immutable set consisting of one or more changes to the deliverables.

Comparison of commands and concepts

The commands used by Rational Team Concert and Git are very similar. The main difference is that while source code management within Rational Team Concert takes the form of centralized management, within Git it is distributed and managed in server repositories and local clones.

In Figure 3, the Rational Team Concert user checks in the code and transfers the code from the Eclipse workspace to the repository workspace. Next, transfer the code from the repository workspace to the remote stream using a delivery operation. Finally, the approval operation transfers both to the repository workspace and to the Eclipse workspace.

Command flow within Rational Team Concert

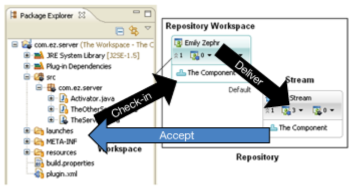


Figure 4 shows the multiple steps required to move code to a shared team branch within Git. In this scenario, the developer adds changes in the working tree to the staging area, commits the changes to the local branch, then pushes the changes to the remote branch and finally merges it into the team branch To do. Then the change is transferred from the team branch to the working tree by the pull operation.

Command flow within Git

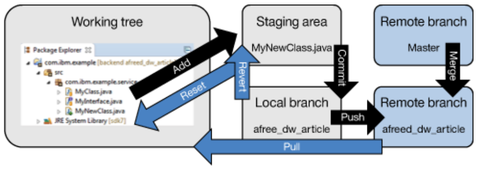


Table 2 clarifies the meanings of the major commands of Rational Team Concert and Git and compares them.

Key commands of Rational Team Concert and Git

Comparison of commands between Rational Team Concert and Git

Rational Team Concert	Git	Note
check-in	add to	Make local working copy changes ready for integration into the upstream. Within Rational Team Concert, this action causes the changes to be sent to the server for storage. In Git, changes are sent to the local staging area.
delivery	Push / Merge	Rational Team Concert users will integrate their repository workspace code into the stream. Delivery will be blocked if merge is required. Git users usually submit the code to the remote branch (feature branch) by push operation first. Within another operation, merge this branch into the main integration branch (aka master branch). If manual intervention is required for merging, Git will block the merge until the user resolves the conflict.
Display annotation addition / history	log	Both systems allow you to fully track changes to files and changes within the commit period.
Load	Replication	Download the source from the master repository to your local workstation. With Rational Team Concert, you can partially load the repository.
Approval	pull	Download the changes from the active branch on the master repository to the local work area. In Rational Team Concert's approval operation, all changes are pulled from the stream the user is working on. Because Git pull operation pulls changes from one branch to another branch (eg remote branch to local branch), Git pull operation directly corresponds to Rational Team Concert approval operation.
Not applicable	fetch	Download changes from all the remote branches the user is tracking to the user's local repository clone. The fetch operation **does not** update the user's local branch to the state of the downloading remote branch until the user pulls from that branch .
Lock / Unlock	Not applicable	Locking prevents other users from changing the file. This mechanism is not in Git.
Discard (active change set)	Unstage	Within Rational Team Concert, this command deletes changes from the repository workspace and the local sandbox. Within Git, remove changes only from the local branch, leaving the working tree intact.
Discard (completed change set)	reset	Updates the local state as if the given change was never downloaded. Within Rational Team Concert, the discard operation will be blocked if the discard operation results in a gap in the state. Git has several options for the reset operation. Reset-hard is roughly equivalent to the discard operation of Rational Team Concert.
Status (pending change)	status	Both tools allow you to compare local and remote states in different ways.

Rational Team Concert	Git	Note
Change flow target	check out	For example, instead of mainstream development, switch to another development flow for the purpose of working on the maintenance branch.
Create a snapshot	Create tag	Mark the state of the repository using a known ID (for example "Release 1.0").
Create repository workspace / stream	Create branch	Create a new development flow / integration point.
Interruption	Stash	Reset the repository workspace and working tree to the state they were in before the interrupted change was made. Within Rational Team Concert, suspended change sets are stored on the server, so you can include the change set in another repository workspace owned by the same user. Within Git, stash operations apply only locally.
Restart	Git stash applied	Interruption / operation opposite to stashing. This command applies suspended or stashed changes to the user's work area.
reverse	return	Create a "reverse" change set with that change. In effect, we will undo those changes.

About Git's scenario

The usage pattern of each of Rational Team Concert and Git reflects the fundamental difference between policies and methods for source code management. While developers tend to uniformly use Rational Team Concert for all development scenarios (for predictability and uniformity), Git users can follow a variety of usage patterns and use I will adapt the pattern. The difficult aspect of transitioning to Git is to determine the pattern (flow in Git terminology) that is appropriate for the requirements of the project.

The most commonly used Git patterns are GitHub Flow and Git Flow, so let's take a look at the features of these two patterns here. Given how each pattern will be implemented within Rational Team Concert, it should be helpful to use that pattern in Git.

GitHub Flow

GitHub Flow is a source control pattern assuming a branch with a relatively short lifetime. The branch in this pattern represents the development of functions and other small unit of work. When the unit of work is completed and all the tests pass, the owner of the branch merges the branch into the master branch and deletes the branch associated with the unit of work.

If we were to implement this pattern in Rational Team Concert, we would first create a new "feature" stream based on the integration stream. After that, we will merge one or more repository workspaces into this feature stream until development is complete. Once developed, we will merge one of these repository workspaces into the integration stream and submit all the changes at once (as stated above, Rational Team Concert will submit directly between streams It is not allowed). Both Rational Team Concert and Git will give you a warning if the code can not be merged because of merge conflicts. In that case, you may be prompted to resolve the merge.

Git Flow

Git Flow is based on " A successful Git branching model " by Mr. Vincent Driessen . Just like GitHub Flow, this pattern uses a short-lived feature branch that branches off from the shared development branch. However, as a big difference with GitHub Flow, Git Flow uses a separate branch for release candidates and does not frequently submit deliverables to the master branch.

To fit Git Flow to Rational Team Concert, you will use the method outlined in this linked article " How to keep your streams flowing smoothly in Rational Team Concert ". With this approach, the team sets up an integration stream to merge daily development artifacts, just as GitHub Flow does. Then we use the 'release candidate' build process to promote from the integration stream to the release candidate stream. When release candidates are approved, the team will set up a similar build process and promote release candidates from the release candidate stream to the master stream. In that case, we recommend using snapshot labels like "Release 1.0".

With Rational Team Concert you can easily create streams from any specific snapshot so you can reduce this pattern within Rational Team Concert. Rational Team Concert only needs to carefully manage the snapshot of the stream at the point where you usually create a new branch within Git's flow.

Migration from Rational Team Concert to Git

In order to switch to the new development tool, usually some adjustment is necessary. In order to facilitate the transition from Rational Team Concert to Git, it is useful to change the process as follows.

- **Use branches that will survive for a short period of time** . The Git flow is most effective for short term survival feature branches that contain small unit of work. The stream of Rational Team Concert tends to persist for a long time, and the unit of work submitted by developers is usually large. Using short-term focused branches makes it much easier to review code using pull requests, but makes it easier for other developers to merge code. Please make the Git branch focused so that you do not submit too much code at once.
- **Commit frequently** . Within Rational Team Concert, changes submitted will be distributed to all concerned parties. This tends to keep the code for a long time as developers try to submit changes after confirming that they are completely ready. With Git's policy, we recommend that you create branches frequently, even for simple functions. Until you merge them into the master branch, Git's commit is generally done to your own branch, and the committed code will not affect other users. Commit frequently and leave an audit trail of the work you did. By doing so, you will be informed of the details of the update content, so it will be a more effective code review. In addition, team members can see all remote branches, which makes it easier for team members to understand what they are doing and results in synergies between teams.
- **Use multiple branches** . Do not be afraid to start multiple branches at once. Within Rational Team Concert, we link one work area (sandbox) to only one repository workspace, but in Git you can easily switch sandboxes to several branches. This helps to keep the size of the branch small and narrow its focus. Please use multiple branches simultaneously to process multiple areas of code. In that case, if you submit the code, the code review will be focused on specific tasks per branch.

Summary

This article was written for developers familiar with Rational Team Concert who want to learn more about Git. In order to successfully adapt Rational Team Concert's workflow to Git, you need to change your way of thinking about code management and distribution. The goal in this article was to allow developers thinking about migrating to Git to clearly understand the differences in policy, architecture, and command structure between the two source management systems, so that each To find out more about the features unique to the tool, please refer to the following references.

Downloadable resources