DevOps Model RealTime Git Integration

Author: Anders Ek
HCL

This document describes how to handle models developed with DevOps Model RealTime using Git version management.

The document was last updated for Model RealTime 10.2 (integrating with EGit 4.8.1). Most screen shots were captured on the Windows platform.

Introduction

Git is one of the most popular source code management systems used today. In this document we will look at how to use Model RealTime together with Git. The focus is on model aspects and it assumes a basic knowledge of Git.
Good general purpose introductions to Git are for example:

- https://git-scm.com/book/en/v2/
  - A good general purpose on-line book about Git
- https://www.kernel.org/pub/software/scm/git/docs/gittutorial.html
  - A good getting started tutorial that describes basic work flows
- https://www.kernel.org/pub/software/scm/git/docs/user-manual.html
  - Git user manual

Model RealTime can be used with Git in two different modes:

- Using the command line Git interface described by the links above
- Using the Eclipse integration (EGit)

EGit is a special purpose rewrite of Git in Java that enables convenient access to Git features from within the Eclipse/Model RealTime graphical user interface. A good introduction to EGit is the EGit user guide available at http://wiki.eclipse.org/EGit/User_Guide.
The command line Git and the EGit Eclipse integration have different benefits and drawbacks:

- Command line Git is fast and efficient, but can be a daunting for new users. It also requires an environment specific configuration to work for model files.
- EGit is in most situations more convenient and easy to use than the command line interface since it is available directly in the Model RealTime modeling environment but has slower performance than the command line implementation.

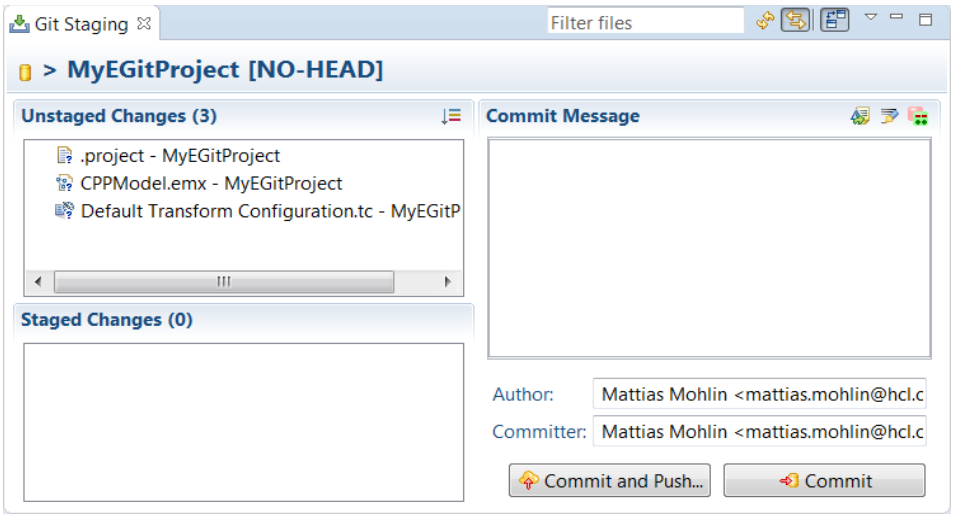
In this document we will look at both variants and it is also possible to use both in combination.

EGit Brief Overview

Before looking into different workflows in more detail we will take a brief tour of the most common EGit views you will use. There are three main views added or used by EGit:

- Git Staging View
- History View
- Git Repositories View

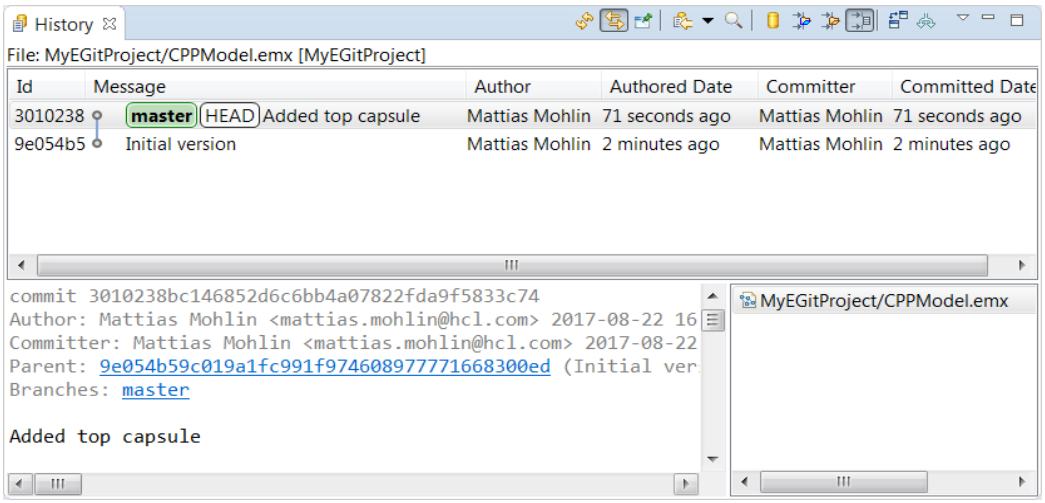
The first view you are likely to encounter is the Git Staging View:



You will use the Git Staging View to understand the state of your working tree, see what files have changed etc. and also to add your changes to the Git repository.

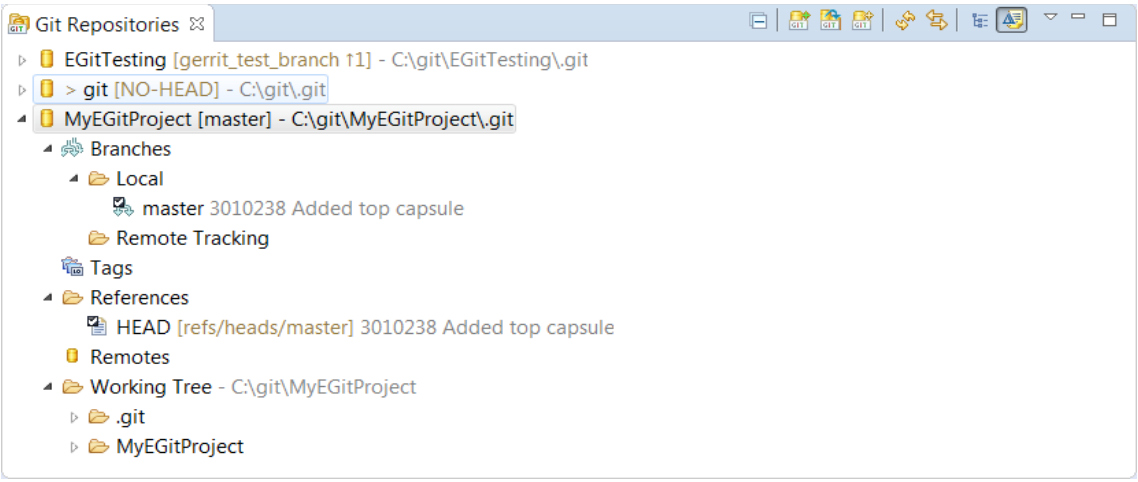
We will look in more detail into the different widgets and how to work with the view in the section Git Staging – Checking work done and committing the result.

The History view is not added by Egit, but is used also for other source code management integrations. However, EGit provides the contents of this view when you use Git.



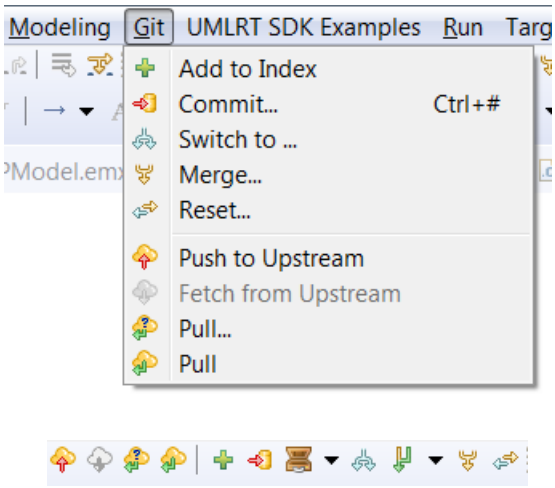
You will use the History view whenever you are interested in the previous actions done to your model. It provides information about who did what in a searchable graphical view sorted by date. We will look into this view more in the section Git History – Investigating Commits and Branches.

The third main view provided by EGit is the Git Repositories view.

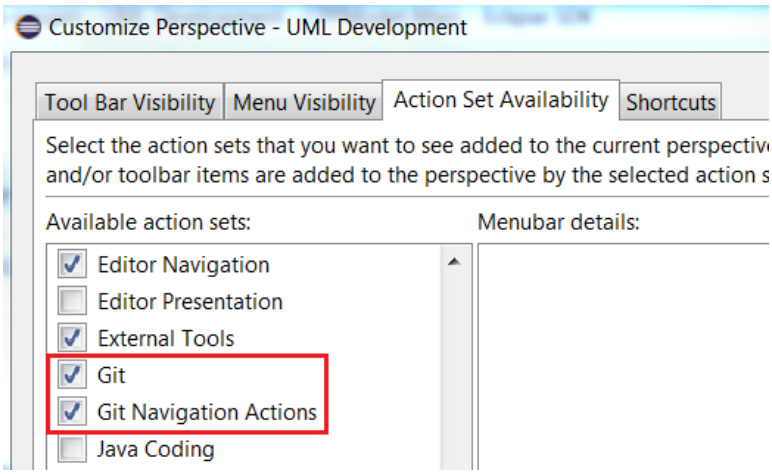


The Git Repositories view gives a summary of key aspects of the Git repositories you currently are working with inside Model RealTime. It provides information about the branches that exist (including which branch you currently are working on), what remote repositories you have configured and where in the file system each repository is located. We will encounter the Git Repositories view in many contexts in this document, including in the section Git History – Investigating Commits and Branches and in the section Checking Out and Creating Branches.

Git also provides a top-level menu and a toolbar.

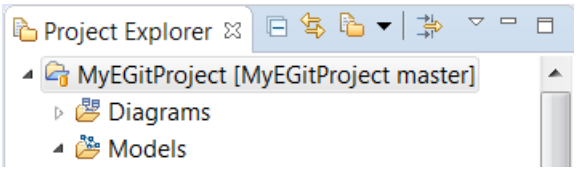


This menu and toolbar are not visible by default in the UML Development perspective. To make them visible invoke the command *Window – Perspective – Customize Perspective*. In the Action Set Availability tab mark Git and Git Navigation Actions.



The top-level Git menu and toolbar are strictly speaking not necessary, since all commands are available in the different EGit views and in context menus. However, in some situations they can be convenient.

You can tell if you have EGit installed as part of Model RealTime in a number of different ways. One very obvious way is that the Project Explorer by default annotates all projects stored in Git with the repository and current branch.



Another difference when you have EGit installed is that when you perform any Git action that will change the working tree contents, like for example when loading a new workspace or when switching branches in Git, then EGit will update the status information. This may typically take a couple of seconds and you can see the status bar (at the bottom of the Model RealTime window) indicating that something is happening. For example when checking out a branch you might see the following message:

Getting Started

Eclipse Projects and Git Repositories

Git is a distributed version management system, that relies on a local repository storing information about the different versions of files and directories. A Git repository is a directory containing a copy of all files and folders plus a special top level directory called ".git". This special directory contains the version information for all files and directories in the repository. There are several different schemes available when combining Git with Eclipse projects, but in general it is recommended to have the ".git" repository directory outside of the Eclipse project structure. A typical structure is thus:

- MyApplication: The root folder of the repository.
  - .git: The Git version information directory
  - Project1: First Eclipse project
  - Project2: Second Eclipse project

A structure similar to this is expected by most commands in EGit and will improve the performance of the tools.

Accessing a Remote Git Repository

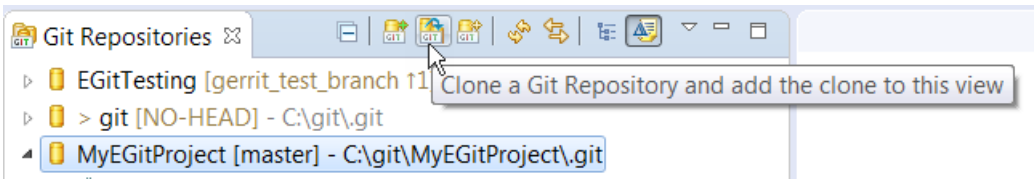
A common first step is often to get access to an existing remote Git repository. This can be done from the command line using the **clone** command:
`git clone <uri>` will create a local folder containing a Git managed file system including a .git folder. It is often a good idea to have one folder on the local file system containing all Git repositories.

An example structure is the following:

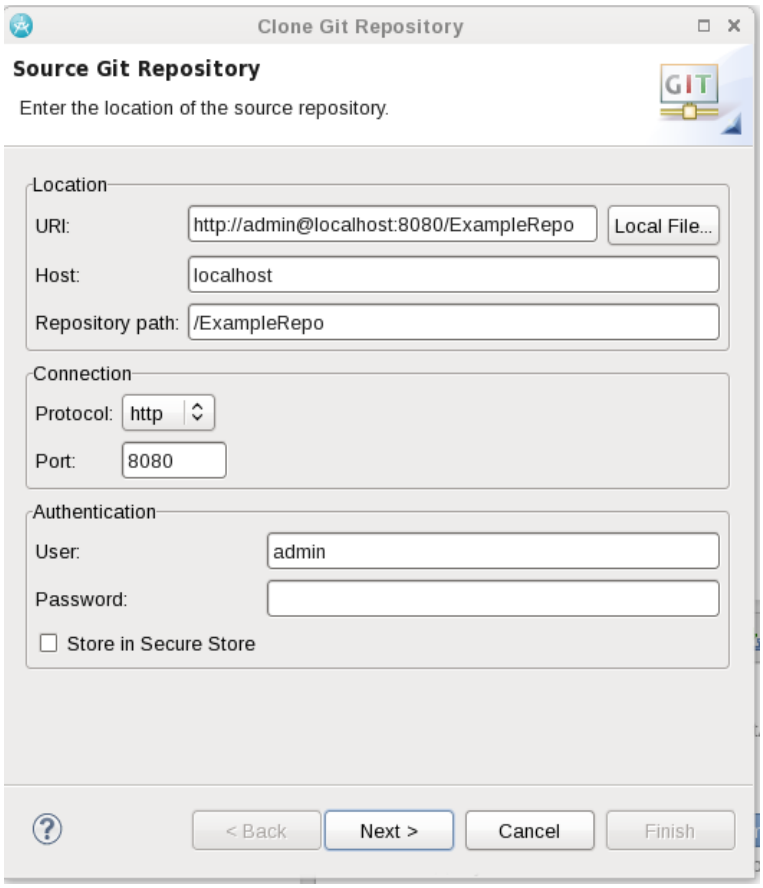
- /home/user/git
  - gitRepo1
  - gitRepo2

Whenever you need to access another remote Git repository just start a command shell in the directory corresponding to /home/user/git and do a `git clone <url>`.

It is also possible (and convenient) to clone the repository from inside Model RealTime. This can be done in the Git Repository view.



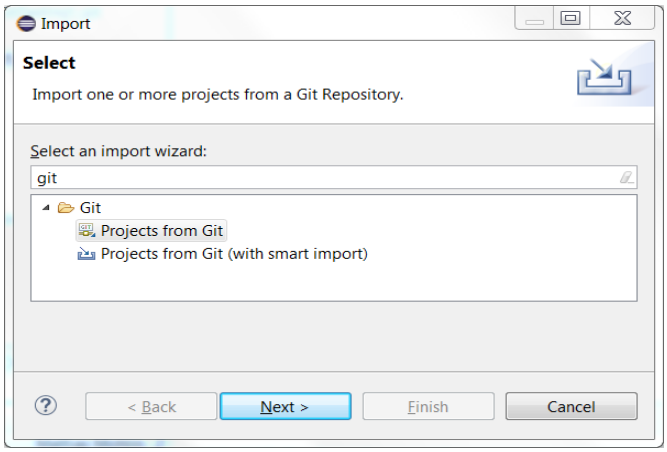
The Clone Git Repositories dialog that appears makes it possible to specify the source of the repository. The easiest way to fill in this dialog is to simply paste a URL into the top URI field. Then most of the other fields will be populated automatically. For example:

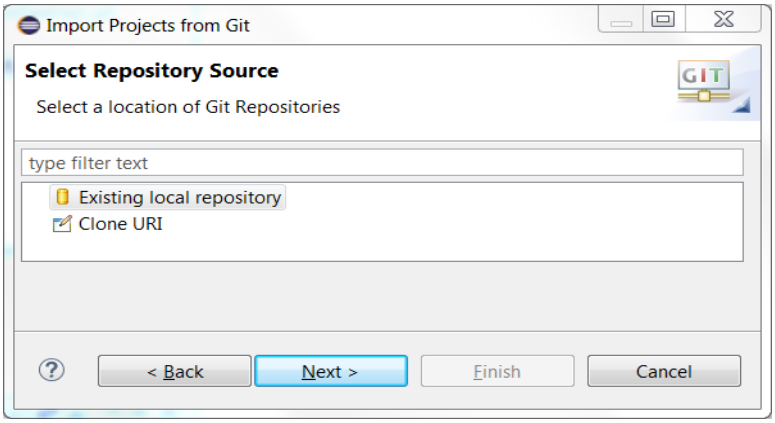


On the wizard pages that follow you can specify where in the local file system the repository will be located and also give a possibility to import any contained Eclipse projects into the current workspace.

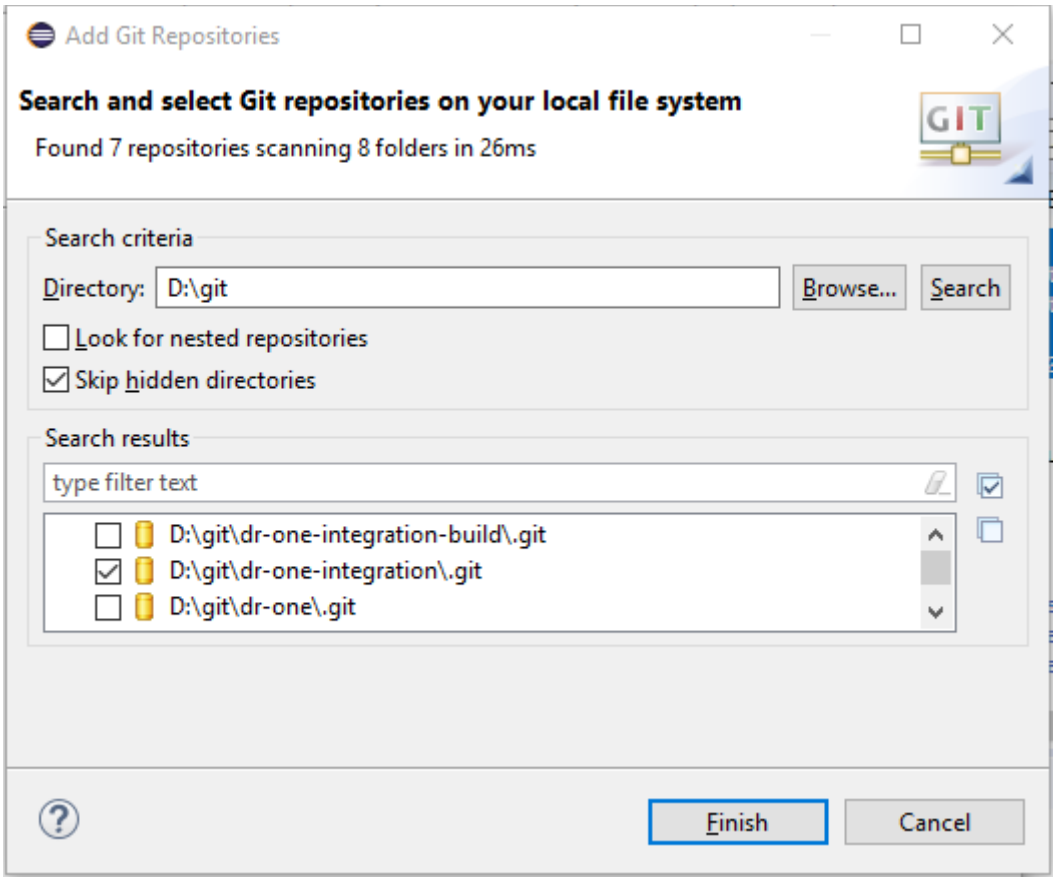
Importing an Existing Repository

To get access to an existing local Git repository from Model RealTime we can use the command *File - Import...* and choose ”Projects from Git”:

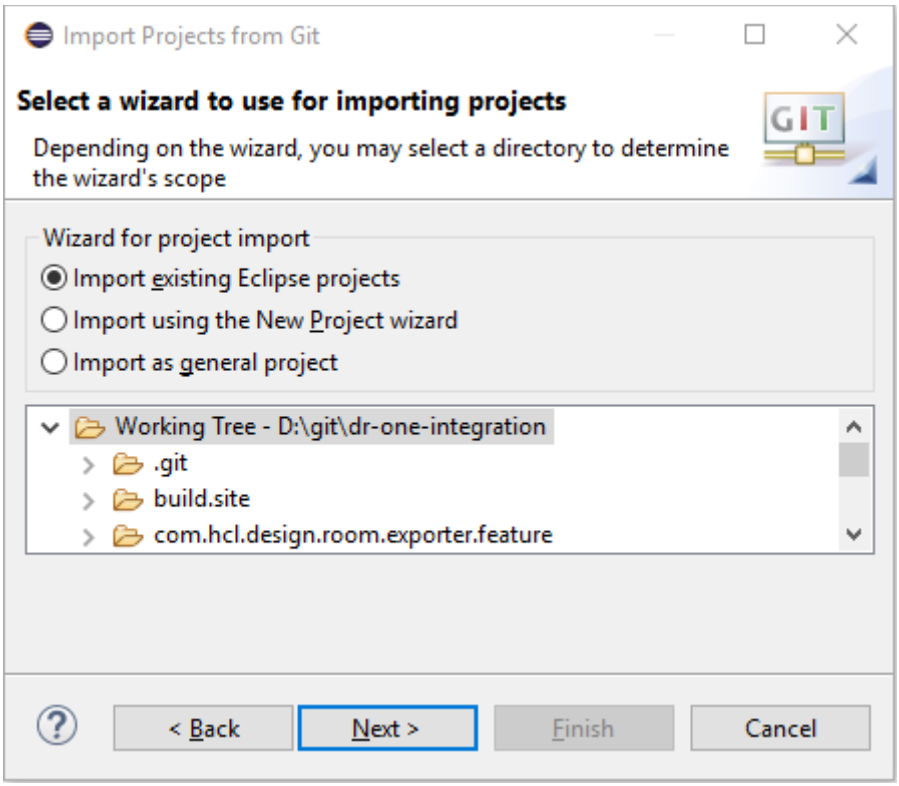




In the ”Import Projects from Git” dialog press the Add button, and enter the path to where you store your Git repositories.



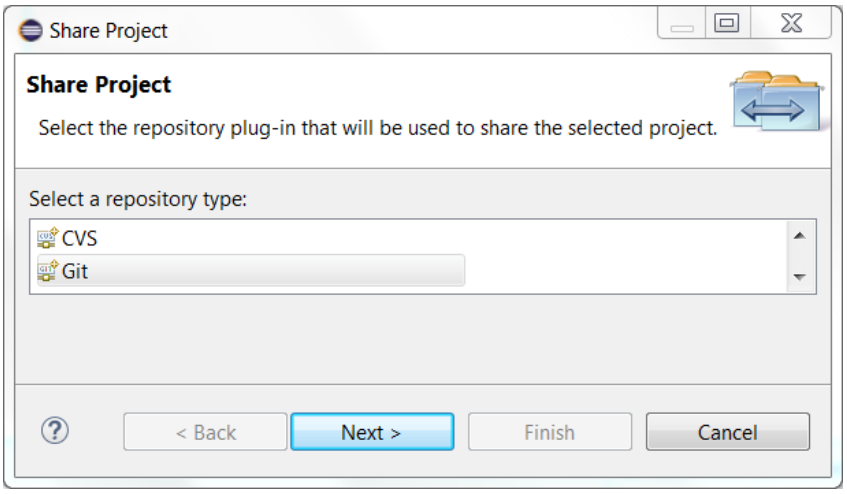
Select the Git repository you want to import and then press Finish. Then proceed to the next page in the import wizard where you can choose how you want to import the repository. If the Git repository contains one or many Eclipse projects you typically want to choose ”Import existing Eclipse projects”.



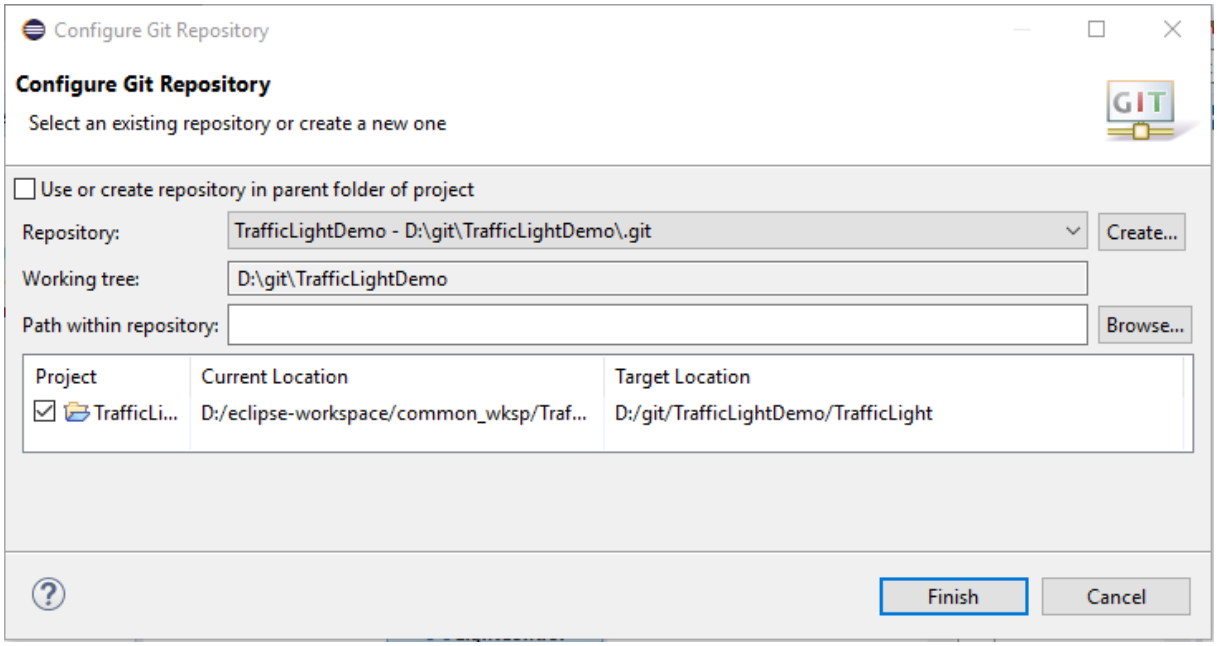
The final wizard page shows you the list of found Eclipse projects that will be imported. Press Finish to import them. Note that imported projects will be referenced from your workspace but will remain stored in the Git repository directory. This is important to remember since you sometimes may use Git commands from the command line and they need to be invoked in the Git repository in the file system.

Storing an Existing Project in Git

If you have an existing local Model RealTime project that you would like to store in a Git repository, the easiest way is to use the *Team - Share Project* command in the context menu of an Model RealTime project. If you have support for multiple versioning systems a dialog will ask you which one to use. Choose ”Git” as the repository type and press Next to configure the Git repository to use.



In the “Configure Git Repository” dialog choose an existing Git repository in the drop down menu, or create a new one by pressing the Create button. Note that it’s generally not recommended to mark the checkbox to create the repository in the parent folder of the project. See the tooltip of the checkbox for more information.



Assuming that you specify a new Git repository to be used for storing your project, the following will happen when you press the Finish button:

- A new Git repository will be created and initialized in the specified location
- Your project will be moved in the file system from your workspace location to the Git repository location
- Your project will be re-imported into Model RealTime from the new location

You are now ready to start managing the project files using Git and EGit features. Note that one thing that does NOT happen automatically is to commit an initial version of the files to Git. See the discussion on the Git Staging View below for more information on how to accomplish this.

Git Staging – Checking work done and committing the result

When working with Git you frequently want to check the current status of your changed files. Typically you want to know:

- What in the model has been changed?
- What was the change?
- What is added to the Git index, ready to be committed?

The **index** concept in Git can be a bit confusing for new users, but essentially it is used to manage the Git representation of what will be committed to the Git history. From a practical point of view a file (or directory) in a Git managed file system can be in three modes:

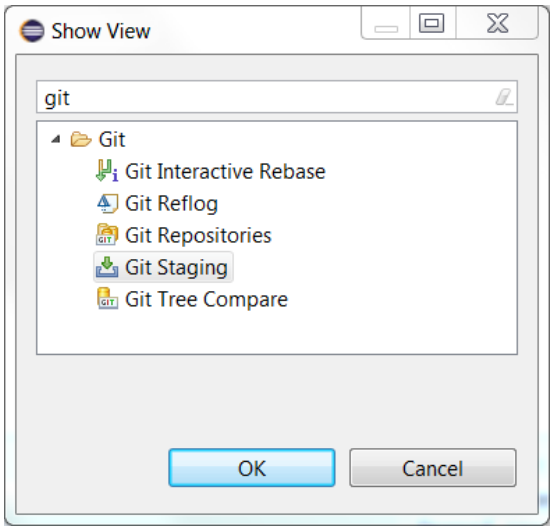
- Part of the local working directory. This is the version of the file that you can see and operate on using e.g. Model RealTime.

- Part of the Git index. This is nothing you typically will operate on, but is a temporary ”staging area” defining what will be committed to the Git history.
- Part of a **commit**. This forms the version history of the files and are from most practical points of view read-only.

If using Git from command line the most commonly used command in this context is the `git status` command. This will print information about what files in your local working directory have been added or modified. It will also tell you what changes you currently have added to the index. Similar information can be found within Model RealTime using the Git Staging View.

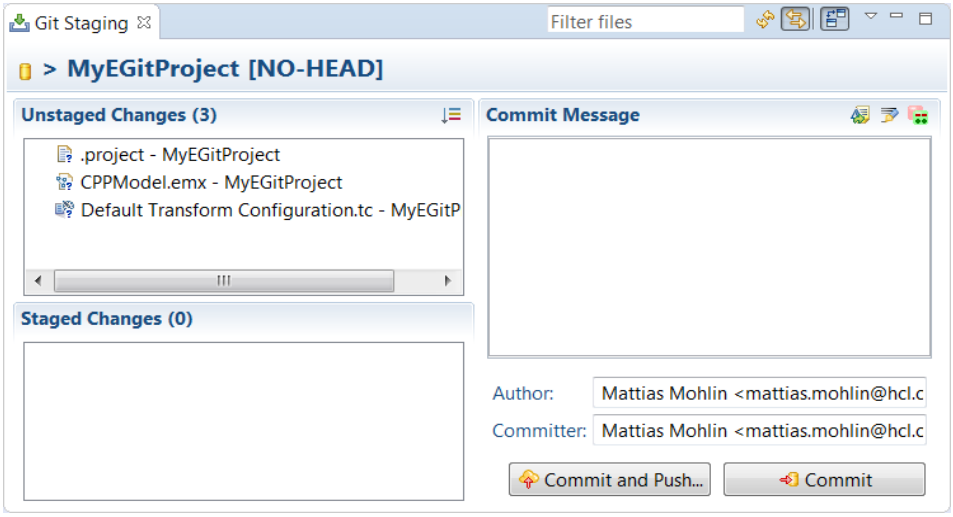
Git Staging View

You can access the Git Staging View using the *Window* - *Show View* - *Other...* command.



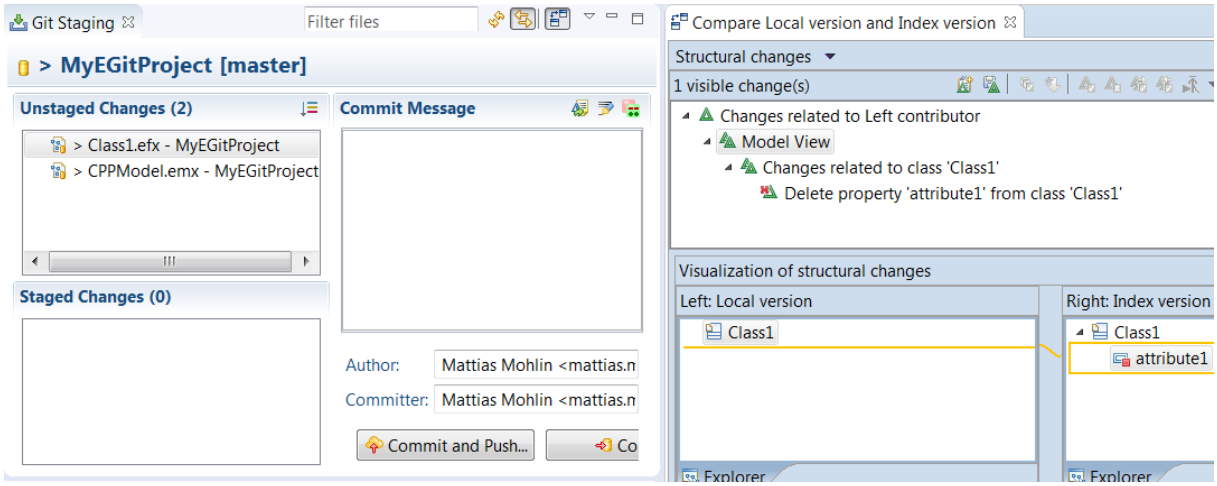
The Git Staging View has two main widgets:

- **Unstaged Changes**
  This is a list of files that have been changed in the local working directory
- **Staged Changes**
  This is the list of files that have been changed and added to the Git index. These files will be committed to the Git history when pressing the Commit button in the view.



Investigating Changes

To review the changes you can simply double click the file. If it is a model file the Compare editor will appear and show the changes.
See the document Comparing and Merging Models for more information about how to use the Compare editor.



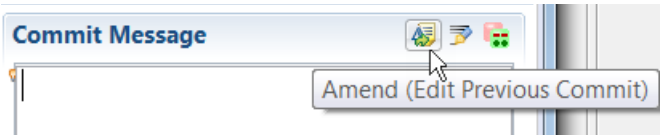
Committing Changes

To be able to commit your changes you first need to ”stage” them, i.e. add them to the Git index. This is done simply by dragging the files from the ”Unstaged Changes” list to the ”Staged Changes” list. If you change your mind you can easily ”unstage” them by dragging them in the other direction.

To commit the changes to Git simply press the Commit button, after describing the changes with a comment in the ”Commit Message” area.

Amending Changes to Previous Commit

Usually when committing your changes you will create a new item in the version history of the repository (such an item is usually called a **commit**). However, it is possible to instead modify the previous commit. You can do this by clicking the Amend button.



The visible result is that you get back the text in the Commit Message field for the previous commit. You can then edit the text so it reflects also the current changes you are about to commit. When you press the Commit button a new commit will not be added to the history. Instead, the previous commit will be amended to include your changes. This can be very useful if you forgot to include some changes in what was already committed, or if you committed some changes by mistake and want to revert them.

Git History – Investigating Commits and Branches

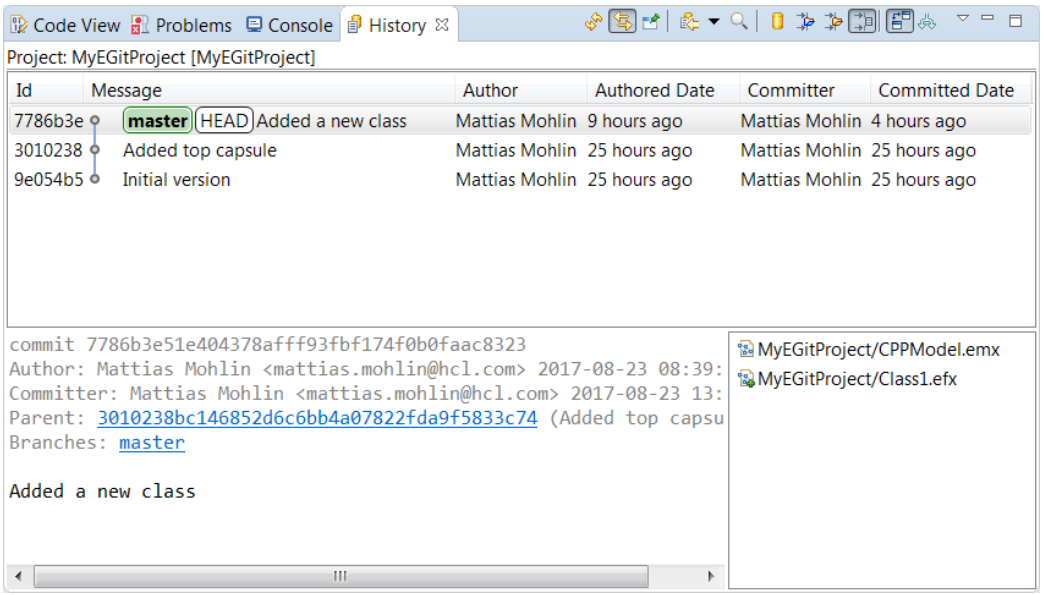
EGit uses the Eclipse History view to let you investigate the version history of a Git repository.

The History View

The simplest way to show the History view is to select a Git managed project in the Project Explorer and perform the context menu command *Team - Show in History*.

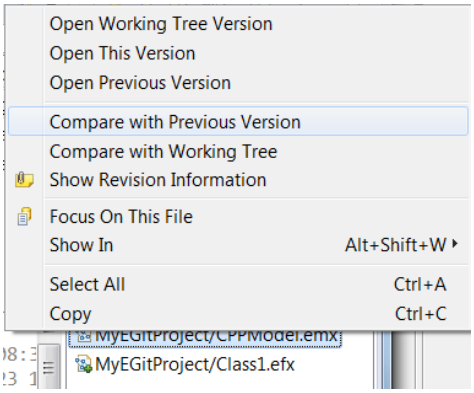
The History view has three main parts:

- The version history (top-most part)
- The file list (bottom right)
- The commit details (bottom left)



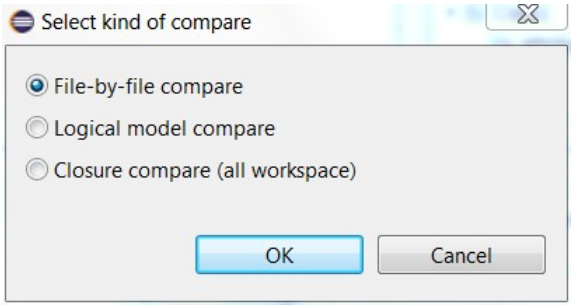
The basic workflow in the view is to select a specific commit in the version history and inspect the details using the file list and commit details parts.

To look at the changes for a specific file in the selected commit use the Compare commands available in the context menu.



”Compare with Previous Version” compares the selected version of the file with its previous version. Only that specific file is compared, which means that this is a so called file-by-file compare operation.

”Compare with Working Tree” compares the selected version of the file with its current version in the Git working tree (i.e. the version you see in your Model RealTime workspace). By default a dialog will appear to let you choose how to perform this comparison:



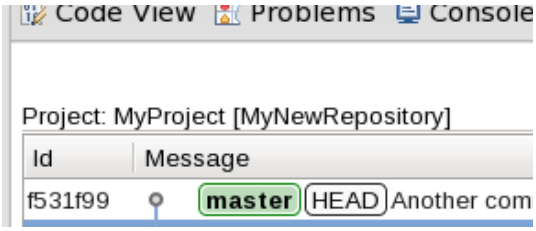
- **File-by-file compare** compares only the selected file.
- **Logical model compare** compares all the files that are part of the same logical model (i.e. one .emx file and zero to many .efx files).
- **Closure compare** compares all the files that are part of the workspace.

See the document Comparing and Merging Models for more information about the different kinds of compares.

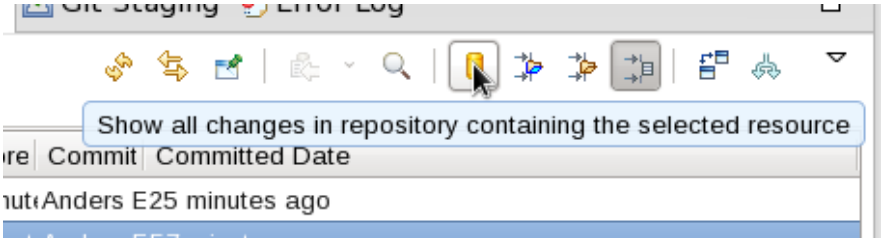
Filtering the History View

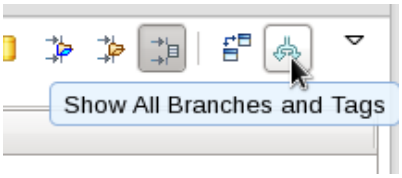
Several of the toolbar buttons in the History view are worth noticing, since you most likely will need to use them in your daily work.

If the version history does not show what you expect to see, the reason is usually that the filtering is too narrow. The basics of the filtering in the History view is the focus element. What you see in the view is always filtered with respect to the element shown in the top left part of the view.



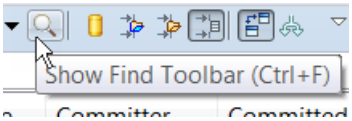
The filtering is controlled by buttons both with respect to repository elements and branches. To show as much as possible select both the repository scope button and the Show all Branches and Tags button.



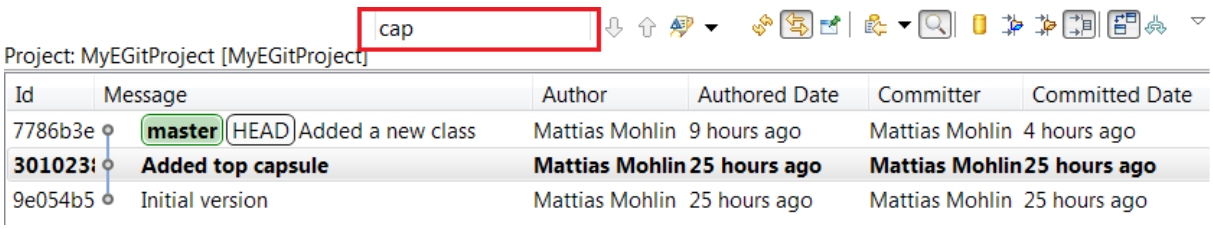


Searching the History View

Another feature of the History view is the search possibility. You can start this by clicking on the ”Show Find Toolbar” button.

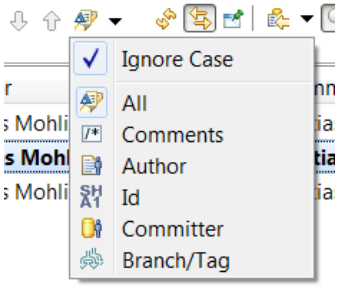


This makes the Find toolbar visible in the toolbar:



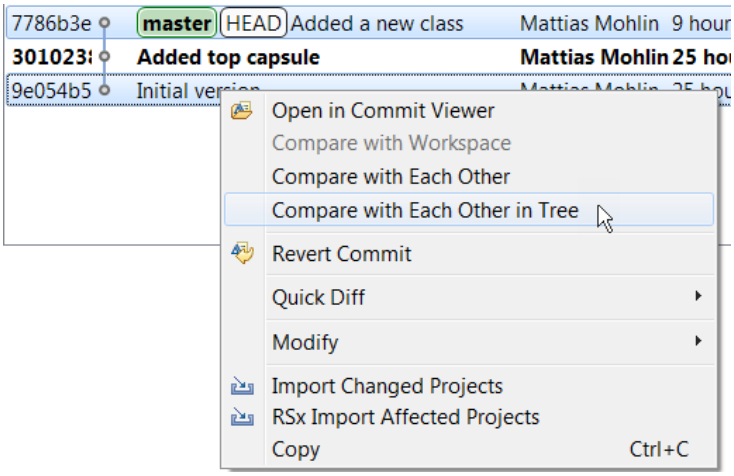
If you type something in the input box, a matching commit will be selected in the history and you can use the Next/Previous buttons to move to following or preceding matching commits.

The button menu next to the Next/Previous buttons allows you to define what columns in the commit history to search and if you want a case sensitive or case insensitive search.



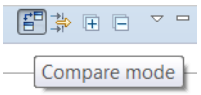
Comparing from History View – Git Tree Compare View

If you select two commits in the History view you will see a command ”Compare with Each Other in Tree” in the context menu.



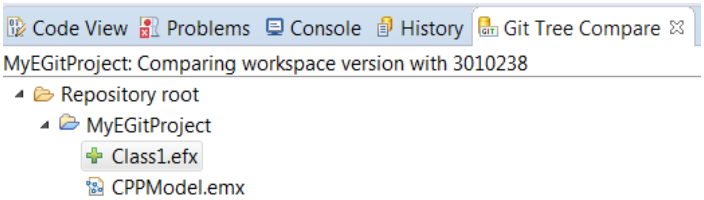
This command takes you to the Git Tree Compare View. This view shows the difference between the two selected commits in terms of what files (or directories) that have been added, removed or modified.

If you click on the ”Compare mode” button in the toolbar



you can then trigger compare sessions for the individual files by double-clicking on them. These compare sessions are always file-by-file.

However, if you just select one commit in the History view, and perform the context menu command ”Compare with Workspace”, the Git Tree Compare View will open in a mode where it shows the differences between the selected commit and what you currently have in the workspace.



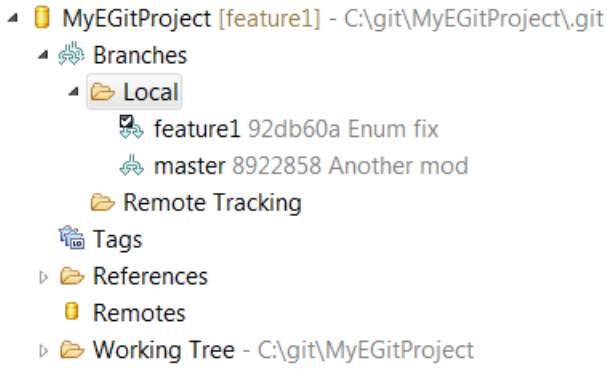
In this case double-clicking on a file will open the ”Select kind of compare” dialog that allows you to expand the compare operation to logical model or closure compare.

Using the Git Repositories View to Investigate Branches

If you only are interested in an overview of the available branches and not the details of the history, then the Git Repositories view is a good alternative to using the History view. It provides a very concise description of what branches exist in the Git repository and which one you have currently checked out.

You can show the Git Repositories view from the context menu of a Git managed project in the Project Explorer by the command *Team - Show in Repositories View*.

To see your local branches expand *Branches – Local*.



The branch with a small check-box is the branch you have currently checked out. It is also shown in brackets after the project name (both in this view and in the Project Explorer).

Checking Out and Creating Branches

To change the contents of your working directory to something from the Git history you need to check out a branch.

Check Out a Branch

It is possible to check out an existing branch both from command line and from the Model RealTime user interface.

From command line the steps are typically the following:

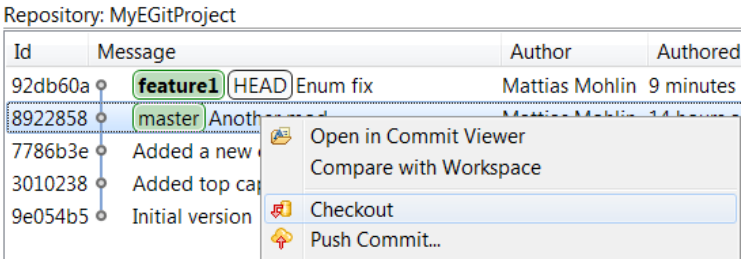
- `git branch`
  This command lists the available branches
- `git checkout <branch>`
  This command replaces the contents of the working directory with the version in the specified branch. It also tells Git to use that branch as the starting point for whatever Git operations you will perform, such as committing new or updated files.

Note! If you check out a branch from the command line you need to make sure that you refresh the Model RealTime workspace before working with the loaded models to avoid problems. You can refresh the workspace using the *File - Refresh Workspace* command. This will

reload any model files you have opened and make sure that the version from the new branch is loaded.

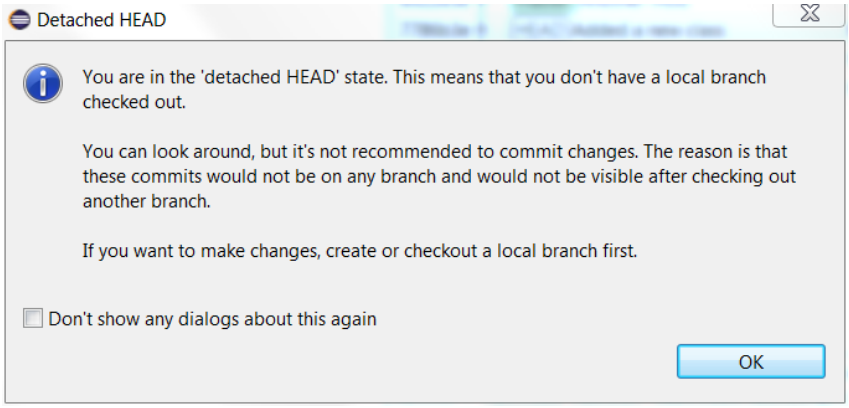
From the graphical user interface the most convenient way to checkout a branch is the History view. If you are looking for a specific commit you can use the Find Toolbar to search for it; otherwise just browse the history for what you are looking for.

The currently checked out branch is shown using bold characters and the current commit also has the indication ”HEAD”. To checkout a different branch, select a commit with a branch (shown with a green box in front of the commit message) and then use the command ”Checkout” that is available in the context menu of the History view.



This will replace the contents of the working directory with the contents of the selected branch and also inform Git that this is the current commit, as indicated by ”HEAD”.

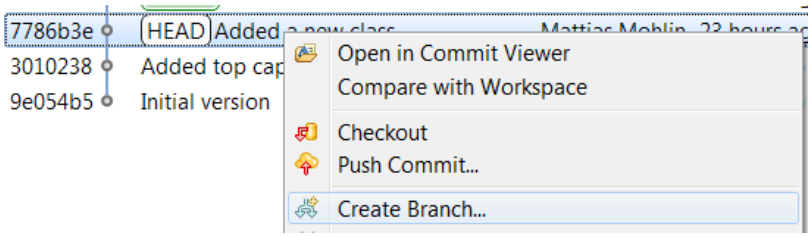
You can use the "Checkout" command also on commits that are not on a branch. Thereby you can view the contents of that commit. However, you should not modify the model in this state since there is no branch where your changes could be committed. A warning dialog will appear to inform about this:



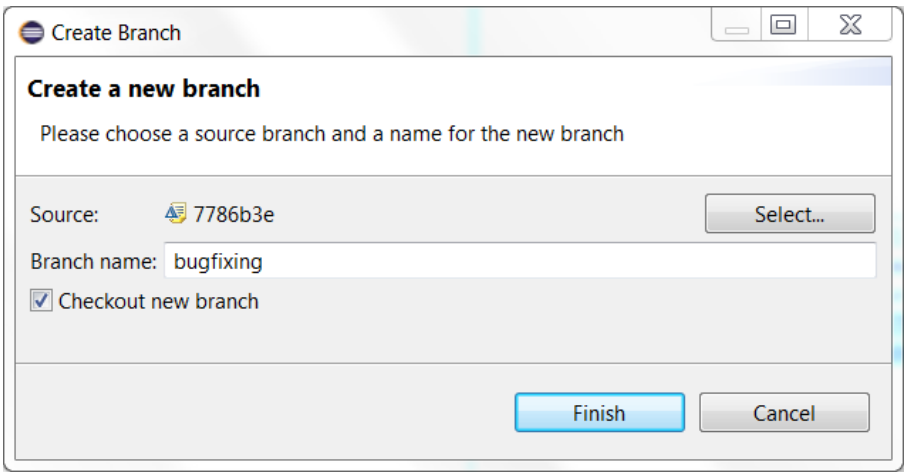
When you perform a "Checkout" from the user interface the workspace will automatically be refreshed and there is no need to use the *File - Refresh Workspace* command.

Create a New Branch

To create a new branch from the History view, select the commit from where you want the new branch to start, and then perform the context menu command ”Create Branch”.



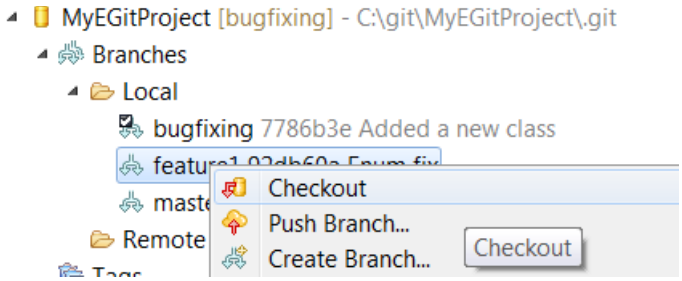
A dialog will prompt you for a name of the new branch. Leave the checkbox "Checkout new branch" marked if you want the new branch to be checked out.



From the command line you can accomplish the same thing by means of the command `git checkout -b <branch>`.

Using the Git Repositories View to Work with Branches

You can also checkout branches from the Git Repositories view. Right-click on a local branch and perform the "Checkout" command.



The context menu also contains commands for creating and deleting branches. Note that you cannot delete the branch that is currently checked out.

Merge and Rebase

A good workflow in Git is to do most work on specific branches (often called "feature branches") and not directly on the master branch. Thereby your changes are isolated from the changes of others until they are ready to be delivered to the master branch.

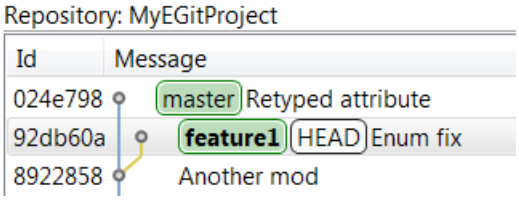
While you are working on a feature branch you often want to incorporate changes done on the master branch (or another branch to which you plan to deliver your changes when they are ready). This is done using the Merge and Rebase features of Git. Both these commands will result in you having all changes from the master branch available on your working branch. The main difference between Merge and Rebase is in the details of how the resulting Git history will look.

- If using Merge the changes on the master branch will be applied as one additional change after the changes you have done so far on your branch. In the history you will get a new commit with the changes from the master branch, but the branch otherwise looks the same. That is, your changes come before the changes from the master branch.
- If using Rebase the changes you have done on your branch will be re-applied to the master branch, after all other changes done on this branch. In the history it will appear as if your branch started at a new later point on the master branch (i.e. it was rebased). That is, your changes come after the changes from the master branch.

Both Merge and Rebase can be performed both from the Model RealTime user interface and from the command line.

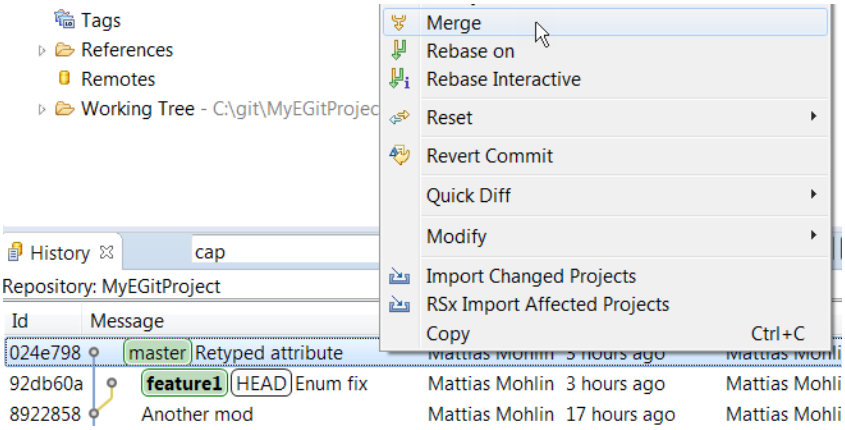
Merge

Let's look at a simple merge scenario. Assume that you are working on a feature branch "feature1" with the following simple history:

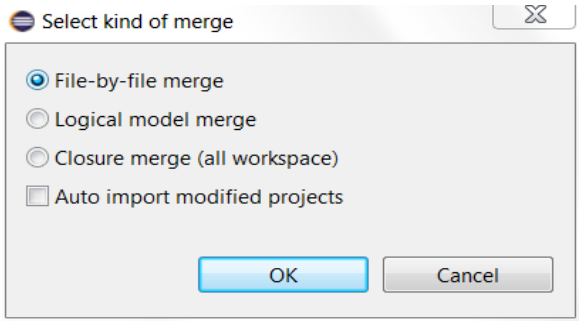


Here we can see that since we created the feature branch (on commit "Another mod") another commit ("Retyped attribute") has been delivered to the master branch. We want to incorporate that change on our feature branch so we can test that it works well with the changes made on the feature branch.

Let's first do this using the Merge command. Merge can be invoked from the History view. Make sure you have the target branch (feature1) checked out, then select the source branch (master) and perform the "Merge" command from the context menu.



If you haven't changed the preferences in *Team - RSx EGit Integration* for "Merge kind selection" you will now be prompted by a dialog about how to perform the merge.



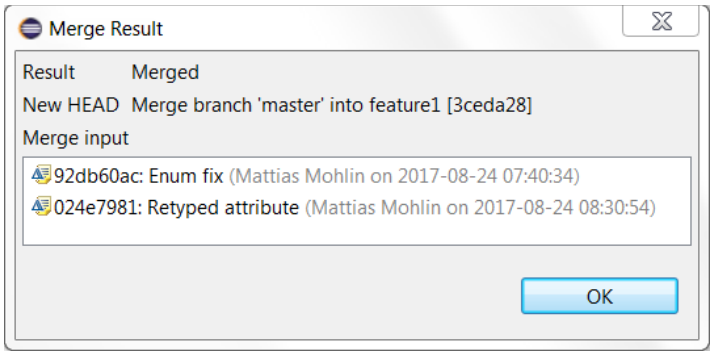
- **File-by-file merge** merges all files that need to be merged using one merge session for each file.
- **Logical model compare** groups files that are part of the same logical model, and merges these groups of files in one merge session per logical model.
- **Closure compare** merges all files in a single merge session.

In most cases file-by-file merge is sufficient, and it also has the best performance. However, if you have done refactorings then the other options can be considered.

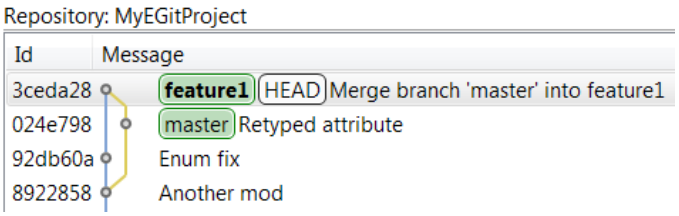
See the document Comparing and Merging Models for more information about the different kinds of merge.

The ”Auto import modified projects” is a useful feature if you only have a small part of the complete model loaded in your workspace. If you select this option all projects that are modified will be automatically imported into your workspace. This allows you to inspect the changes that were done after the merge is completed.

The merge operation itself is a silent operation that ends with a dialog showing the result.



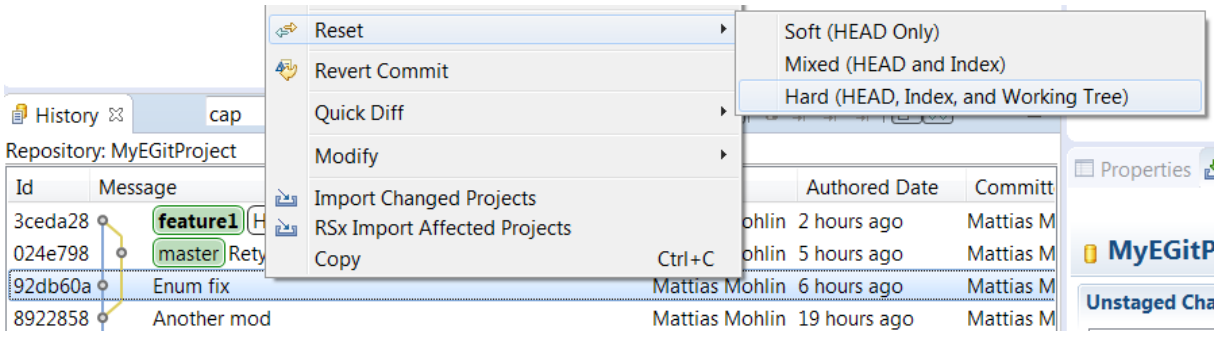
If there are no conflicts the new and changed files are committed and the History view now looks like this:



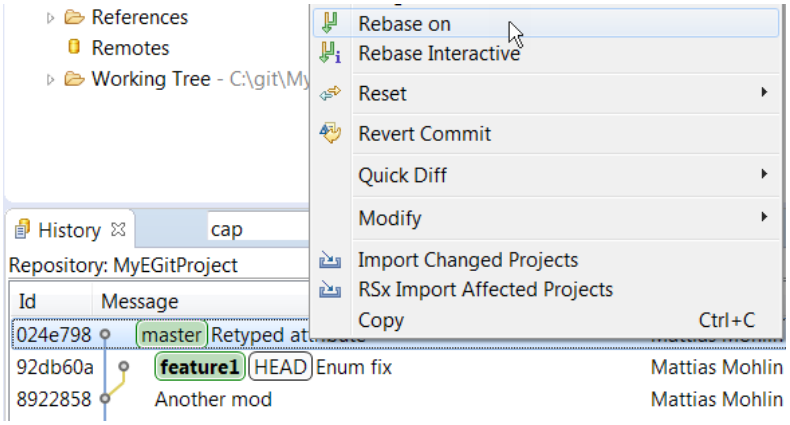
The same merge can also be done from command line using the command `git merge <source-branch>`. However, **do not merge from command line if Model RealTime has not been added to Git as a merge handler.** Doing command line merges without adding Model RealTime as a merge handler can often result in corrupted models. See section Integrating Model RealTime with Command Line Git for more information about how to integrate Model RealTime with Git command line merge. There you can also find the details about which merge mode that will be used in this case.

Rebase

Now let's use the Rebase command instead for introducing the change from the master branch onto the feature branch. If you have performed a Merge but realize that you should have done a Rebase instead, you can easily revert the merge by right-clicking on the previous commit in the History view and run the context menu command *Reset - Hard*.

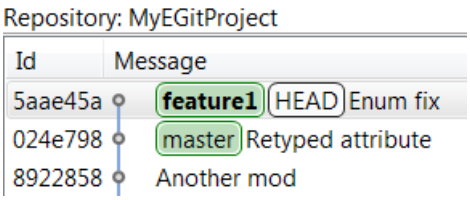


After resetting the merge commit we are back to the state prior to the merge, and can now instead use the Rebase command. With the target branch (feature1) checked out, select the source branch (master) and perform the "Rebase on" command in the context menu.



Just as when merging, you will be prompted to choose the merge type. After that a dialog appears with the rebase result.

The resulting history shows that instead of merging the changes from the master branch to the feature branch, the changes you did on the feature branch have been re-applied onto the changes from the master branch. The base of the feature branch is now the "Retyped attribute" commit, i.e. it was rebased from the "Another mod" commit which was its previous base.



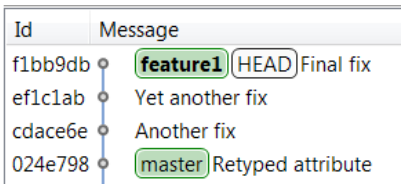
You can also perform rebase from the command line. The command to use is `git rebase <source-branch>` to rebase the currently checked out branch on the master branch. Since rebasing may require models to be merged, the same warning applies as for the Merge command. **Do not invoke Git rebase for Model RealTime model files without a proper integration into Git merge handling.**

Rebase is a possibility to create a nice and simple History view and is safe to use on local branches. **However, be very careful if using rebase on a branch that is shared with other team members (see Collaborating within a Team using Remote Branches). If someone else has downloaded the branch and have it in their local repository, Rebase will cause problems. So, if you work in a team and share branches, avoid rebase, or make sure to coordinate the rebase so everybody involved will clone a new copy of the repository afterwards.**

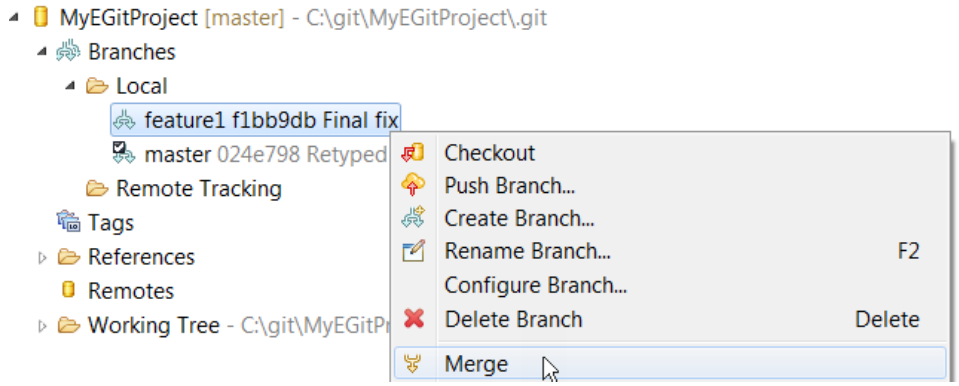
Rewrite History using Squash Merge

In a large organization, and as time goes by, it becomes increasingly important to have a reasonably simple version history with understandable commits. Often, the granularity of commits on a feature branch is smaller than what we would like to have on the master branch. To overcome this problem Git provides a possibility to rewrite history during merge and collapse several commits into one commit. This is done using Squash Merge and is typically done when merging changes from a feature branch to the master branch (i.e. when your feature is ready to be delivered).

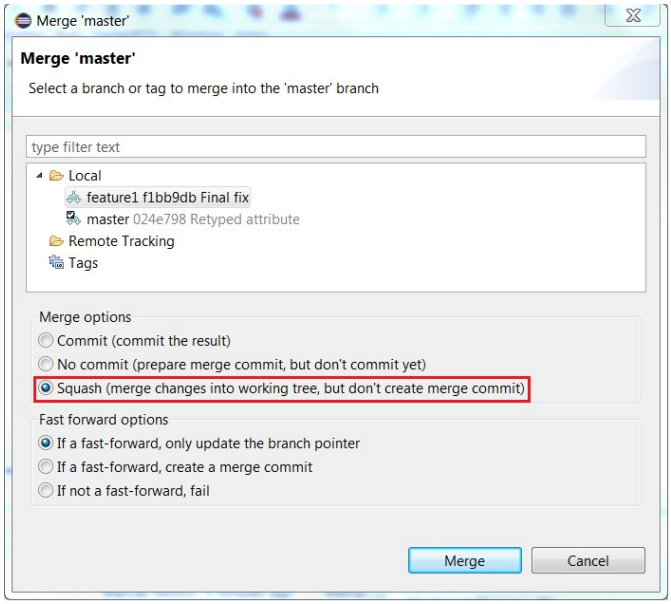
As an example, assume that we have implemented a feature on a ”feature1” branch using three commits, and feel that the feature is ready to be delivered to the master branch.



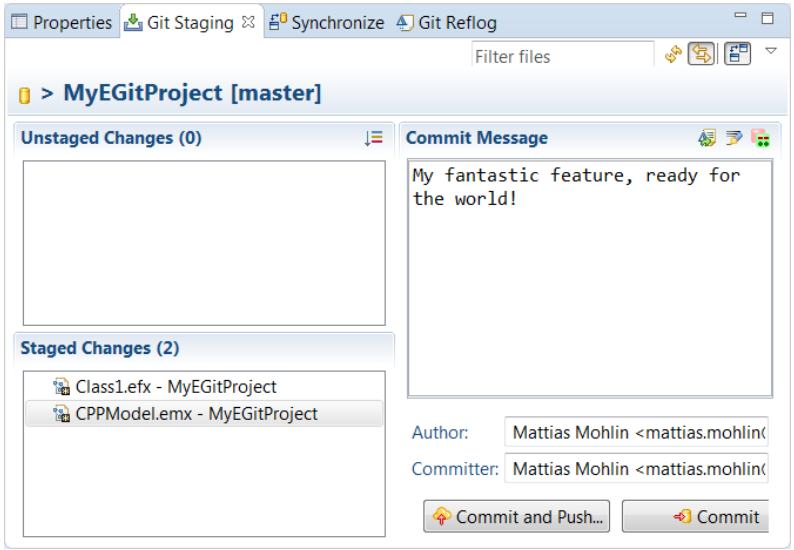
To merge our changes to the master branch as a single commit we can perform a merge from the Git Repositories view. First checkout the master branch, then select the feature branch and perform the Merge command from the context menu.



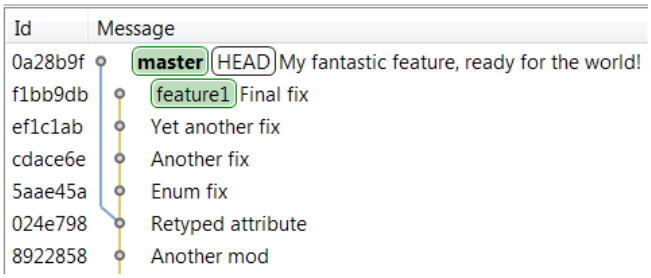
In the dialog that appears make sure to set the "Squash" merge option.



The result after the merge is that all changes from our feature branch are merged to the master branch, but the result is not committed. This give us an opportunity to commit the changes ourselves, using a suitable commit message that describes the changes as a whole.



This results in the following history:

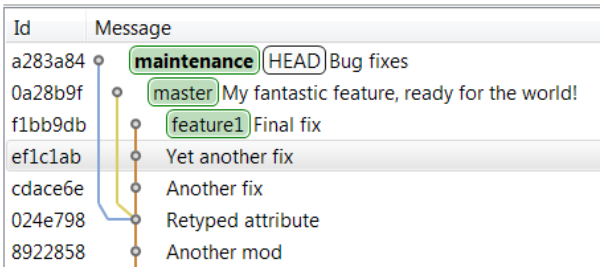


Note that others in your team will only see your changes as a single commit on the master branch, since they will not see your feature branch (unless it is explicitly pushed to the central repository and pulled by others).

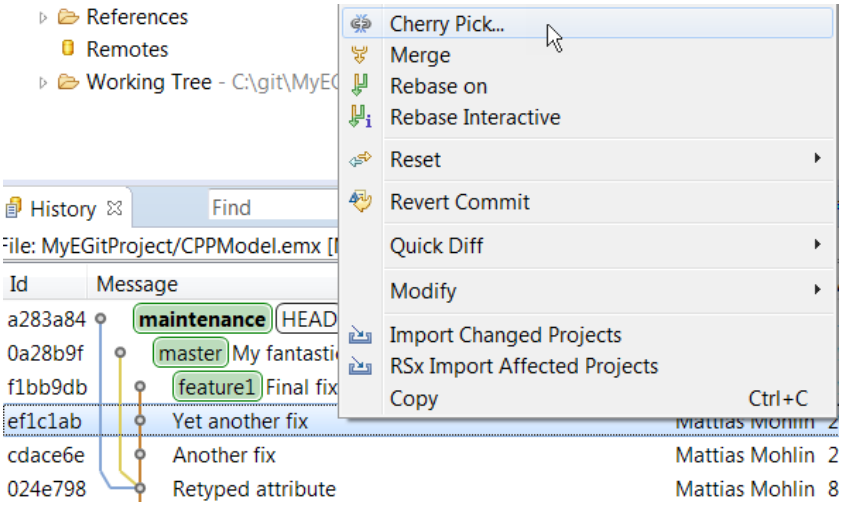
Merging a Specific Commit using Cherry Pick

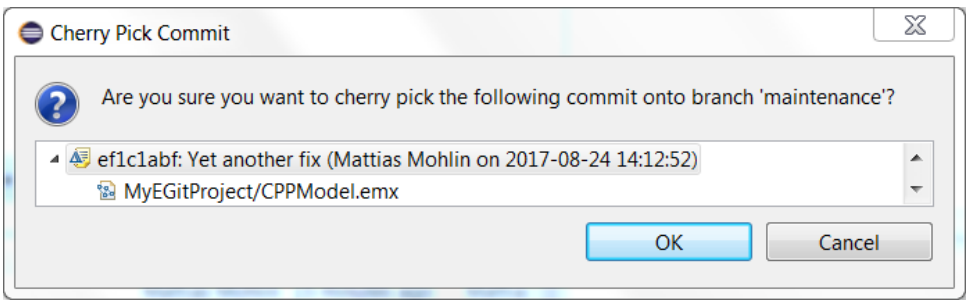
Sometimes you may want to merge only one specific commit from another branch onto your current branch, rather than all commits on that branch. Git supports this through the Cherry Pick command. In Model RealTime we can cherry pick a commit from the History view.

Assume we are working on a maintenance branch where we want to include a few important bug fixes. One of the bug fixes we would like to include happen to be already delivered on a feature branch "feature1" in a commit "Yet another fix".

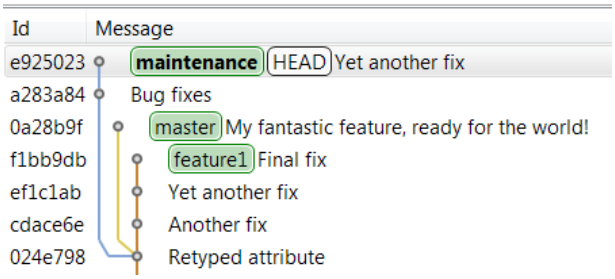


To cherry pick this commit into the maintenance branch we select the "Yet another fix" commit and perform the command "Cherry Pick" from the context menu.





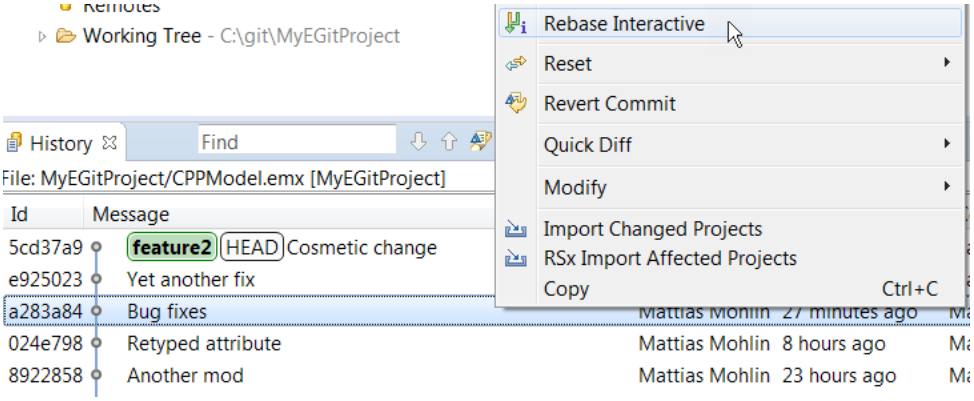
As for the other merge commands we will be prompted to choose the kind of merge to use. The selected commit will be merged into the maintenance branch, leading to the following history:



Interactive Rebase – Advanced Rewriting of History

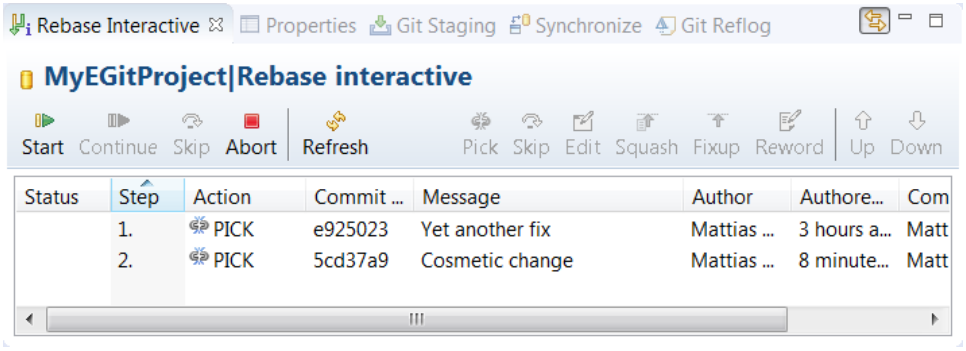
Interactive Rebase is an advanced Git command that combines a regular rebase with squash style merge and some aspects of cherry pick. You can use this as a convenient short-cut if you want to re-apply a series of commits (like in a regular rebase), but in the process also want to change the commit history (like a squash merge), selectively include or not include specific commits (slightly similar to a cherry pick) and even change some of the commits (like an amend commit operation).

You start an interactive rebase by checking out the branch you want to modify. Then use the History view to select the commit from where you want to change the commit history and perform the ”Rebase Interactive” command from the context menu.



Here we will perform an interactive rebase to rewrite the history of the feature2 branch. Just like a regular rebase, an interactive rebase may involve merging so you will be prompted about which kind of merge to use.

The interactive rebase is controlled from the Rebase Interactive view.

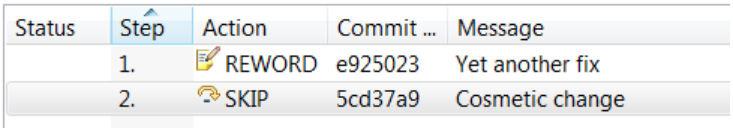


This view lists all the commits that follow the selected commit on the branch. For each of the commits you can now decide the action to perform. The following actions can be chosen:

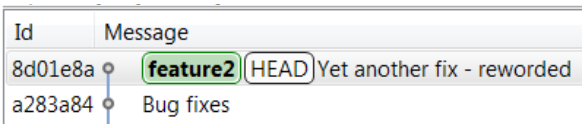
- **Pick**
  Keep the commit as it is. This is the default action.
- **Skip**
  Remove the commit from the history.
- **Edit**
  Make a pause before the commit so you can add additional changes to it.
- **Squash**
  Combine the commit with its predecessor. Also allow the commit message to be edited.
- **Fixup**
  Combine the commit with its predecessor, and use the predecessor's commit message.
- **Reword**
  Make a pause before the commit so you can edit its commit message.

When you have set the action for the commits you have defined the steps that will take place when you start the interactive rebase. If you want you can reorder the commits using the Up and Down buttons.

For example, assume we want to keep the "Yet another fix" commit but give it a better commit message, and that we want to skip the "Cosmetic change" commit. Then we can set the following actions for the commits:



Press the Start button to start the interactive rebase. All the steps in the view will be performed in the specified order. The resulting history will look like this:



For more details about interactive rebase take a look a the EGit user guide.

A word of warning: Avoid using interactive rebase on a branch that you have pushed to a remote repository and are sharing with other team members. Rewriting the history for a remote branch may cause serious problems for everybody that has pulled the branch to their local repository.

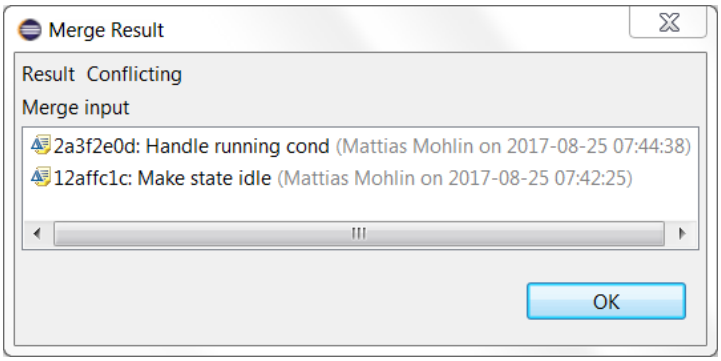
In the above example we used interactive rebase to rewrite the history of a single branch. Of course you can also select a commit on a different branch. In that case the interactive rebase will start by checking out that branch and then allow you to set the actions for all commits that are present on both the branches (after the common commit from where the two branches originate).

Resolving Conflicts

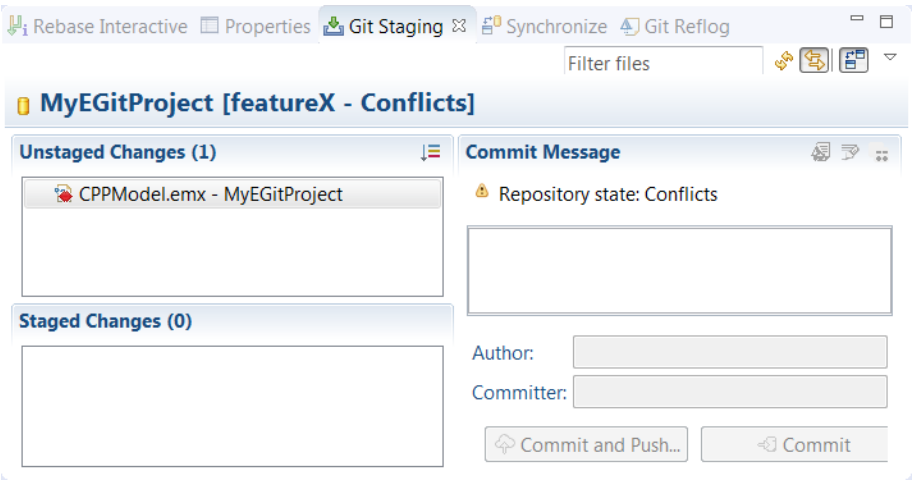
Unfortunately not all merge and rebase operations are as simple as the scenarios we looked at in Merge and Rebase. Whenever we perform a merge or rebase there is a chance that it leads to one or many merge conflicts. This chapter is about how to resolve such conflicts.

Merge Conflicts

If merge conflicts occur you will be notified by the information dialog that pops up after the merge or rebase operation has completed. Here is an example of what that dialog may look like when a merge led to a conflict.

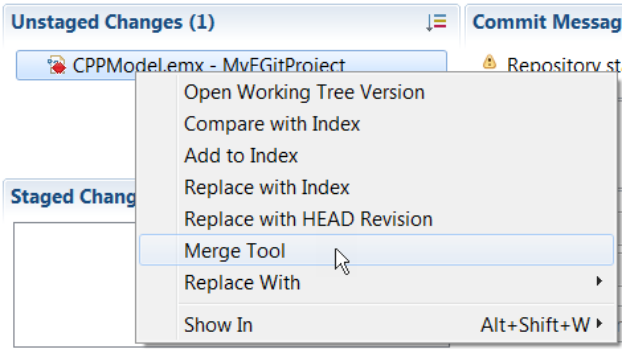


The view that is used for understanding and resolving the conflict is the Git Staging View. If you open this view when the repository is in a conflict state, you will see the text ”Repository state: Conflicts”. This tells you that the conflicts must be resolved before you can commit your changes. The Commit buttons will not be enabled until the conflicts have been resolved.

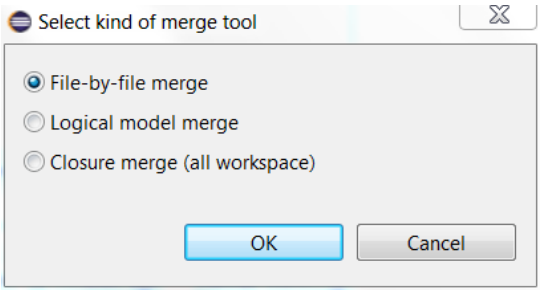


The files that contain the conflicting changes are marked with a small red icon, not only in this view but also in others such as the Project Explorer.

To resolve conflicts you use the Merge Tool command. Right-click on a file marked as conflicting and perform the context menu command Merge Tool.



Just like when merging or rebasing, a dialog will ask you what kind of merge tool to use.



In most cases it's recommended to use Closure merge since it leads to fewer merge sessions, and more accurate descriptions of the conflicting changes. This helps you understand how to best resolve the conflicts.
See the document Comparing and Merging Models for more information about the different kinds of merges.

The Merge Tool command invokes the interactive merge editor of Model RealTime.

See Comparing and Merging Models for information about how to use this merge editor in order to inspect and resolve the conflicts.

After the merge session has been completed, all modified files will by default be added to the Git index automatically. If you don't like this behavior you can disable the preference *Team - RSx EGit Integration - Automatically add resources to index after resolving conflicts*.

Note! For historic reasons the Model RealTime merge editor uses the term "commit" for completing a merge session. Hence, it has nothing to do with committing in Git. In particular, the modified files will not be automatically committed to Git when the merge session is "committed".

The Merge Tool command can also be used to resolve conflicts in non-model files. If you invoke the command on such a file, different interactive merge tools are invoked based on what kind of file it is.

From the command line you can invoke the merge tool using the command `git mergetool` or `git mergetool <file>`. If you invoke the command without a file argument Git will automatically invoke a merge driver for each file that is in conflict.

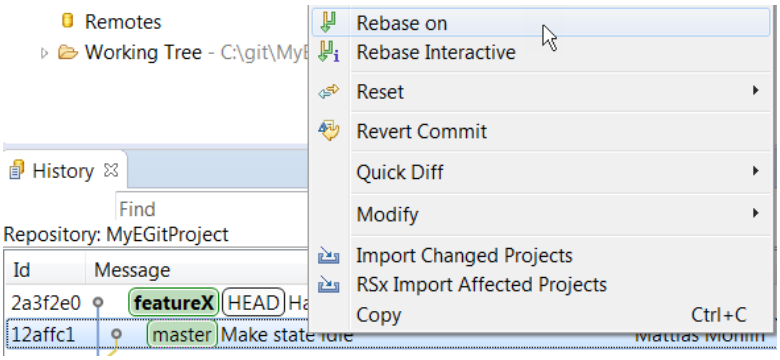
The same warning as for merge and rebase also applies to Git mergetool: **Do not use Git mergetool for model files if you do not have an integration with Model RealTime that triggers Model RealTime as the mergetool driver for model files.** This is likely to cause model corruption. See the section Integrating Model RealTime with Command Line Git for more information about how to integrate Model RealTime with Git command line merge.

Due to the complexity of creating a merge tool integration with good performance that supports closure merge, the currently recommended best practice is to use the Model RealTime graphical user interface and the Git Staging View to resolve merge conflicts.

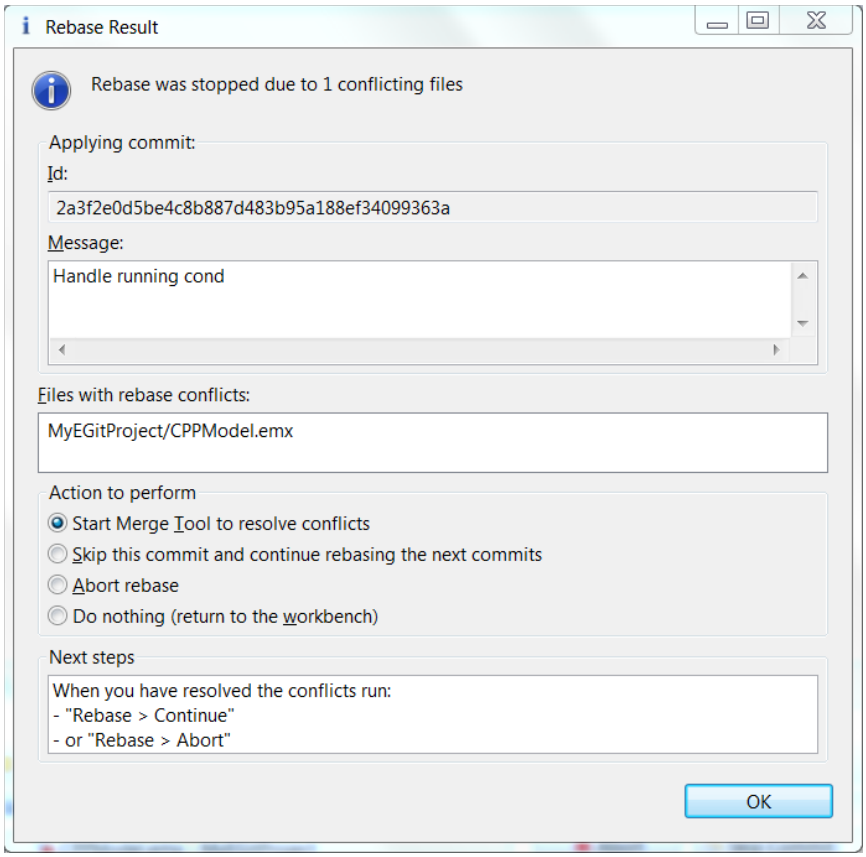
Rebase Conflicts

Since the Git rebase operation essentially is a sequence of merges, the handling of conflicts in a rebase scenario is very similar to the merge scenario. The main difference is that after resolving conflicts found during a rebase you will need to continue the rebase instead of only committing the merge result.

Let's use the Rebase command in the History view, to rebase a feature branch on the master branch.

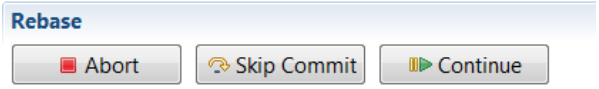


Instead of the simple results dialog we would get from a merge, we now get a more complex Rebase Result dialog describing the rebase state.

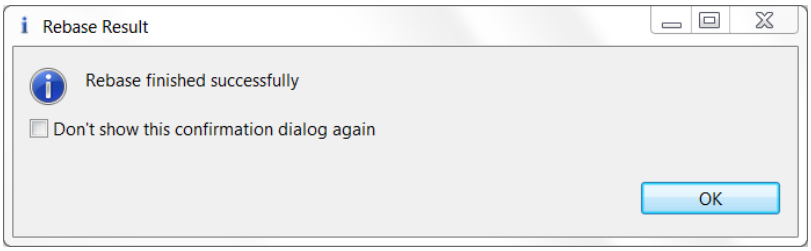


Here it is recommended to choose ”Do nothing (return to the workbench)” and again use the Git Staging View for performing the Merge Tool command. It is not recommended to directly start the Merge Tool from this dialog (the default choice).

When you have resolved the conflict in the merge editor, and completed the merge session, press the Continue button in the Git Staging View to continue.



Continue the process of resolving conflicts using the Git Staging view until you get a dialog indicating that the rebase was successfully completed.



Working with Remote Repositories

One of the key ideas behind Git is that it is a distributed version control system. From a practical point of view the main implication is that you easily can work locally with your repositories and changes. Almost everything we have described in this document so far has been based on having local repositories. However, in order to share your work with the rest of the world and collaborate on joint efforts you need to extend the scope and also look at remote repositories.
When you work with remote repositories Git supports a multitude of different workflows, from a simple scheme suitable for a small team to complex schemes for large distributed projects. You can find a good description of some possible alternatives in Distributed Git - Distributed Workflows. In addition, when using Git in complex projects it is commonly combined with supporting tools for managing the review of changes. A popular tool to manage reviews for Git repositories is Gerrit (https://www.gerritcodereview.com/). It is possible to combine Model RealTime with Gerrit, both using the default Gerrit web based interface and using the review support in Mylyn (http://www.eclipse.org/mylyn/). There is also a dedicated Eclipse plugin called EGerrit (https://www.eclipse.org/egerrit) that can be used.

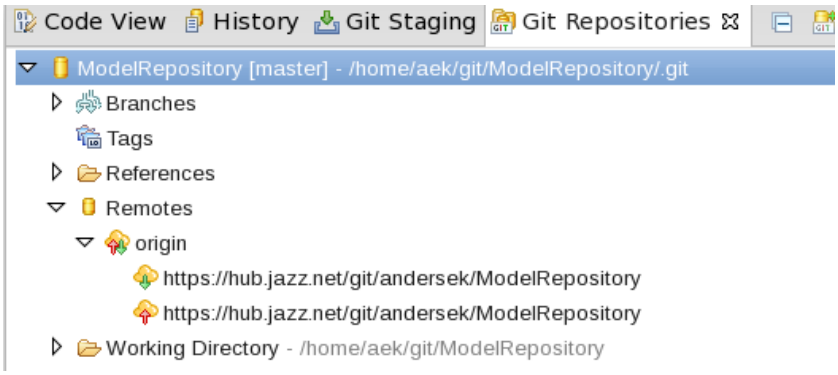
In this section and in Collaborating within a Team using Remote Branches we will focus on the simple scheme where you work directly with a remote Git repository, assuming that you have both direct read and write access to the remote repository.

A more advanced scheme is covered in Working with Remote Repositories using Gerrit Review Support where we focus on how to contribute your work to a shared repository managed using Gerrit.

Remote Repositories

If you followed the Getting Started instructions in Accessing a Remote Git Repository you most likely already have at least one remote repository configured. From the command line, you can use the `git remote -v` command to get information about what remote repositories are configured for your current local repository.

In the Model RealTime GUI this information is most easily seen in the Git Repositories view. To see what remote repositories are configured expand the ”Remotes” node in the outline. This is what the Git Repositories view could look like if you have cloned a remote repository to get started.



In the above image, we set up a remote connection called “origin” to a remote repository. The image also shows the web addresses (URLs) for pushing and pulling the data to and from this remote setup.

Remote Tracking Branches

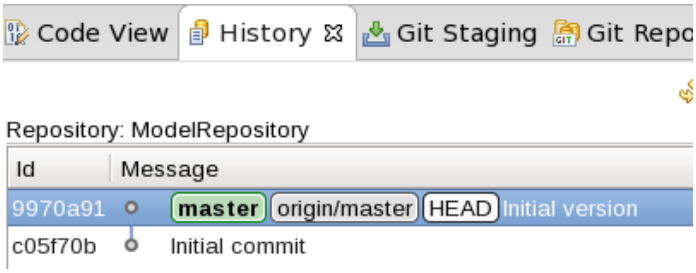
To ensure that local and remote repositories are not too tightly coupled Git introduces a special mechanism to locally keep track of the state of a remote repository. This is the concept of **remote tracking branches**.

In our example above we have a remote repository called ”origin”. To track the state of this remote repository we will in our local repository have one or more branches only intended for this purpose. These branches can be seen in most of the relevant views in Model RealTime. Let us first check the Git Repositories view. If you expand the ”Branches” node for your repository you will see that there now is another set of branches, shown in the ”Remote Tracking” part of the outline.

In our simple example we only have one remote repository named ”origin” and we can see that there is only one branch ”origin/master” that we track from this remote repository.

The remote tracking branches will be visible also in other views where branches are visible. For example the remote tracking branches are also visible in the History view.

This is what the simplest possible example can look like. This is the history of a newly created repository with one ”origin/master” remote tracking branch (and one local ”master” branch).

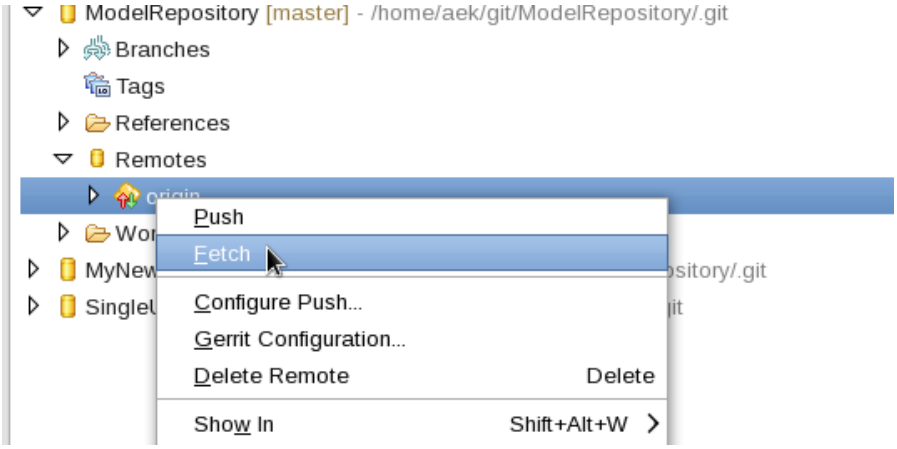


Note that the remote tracking branches are read-only from a local point of view. They will only be updated when you use various commands to synchronize the state of your local repository with the remote repository.

Update from Remote Repositories: Fetch

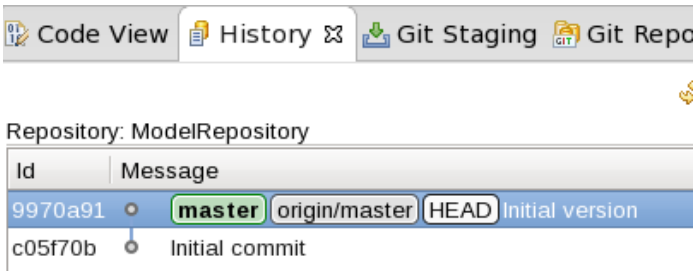
It is easy to update the local repository with changes done in the remote repository. The most basic way of doing this is using the ”Fetch” command. In our simple example from the previous section the command `git fetch origin` would retrieve any changes done in the remote repository called ”origin” and make them available in the remote tracking branch ”origin/master”.

In the EGit GUI the Fetch command is of course also available, for example if you select a remote repository in the Git Repositories view.

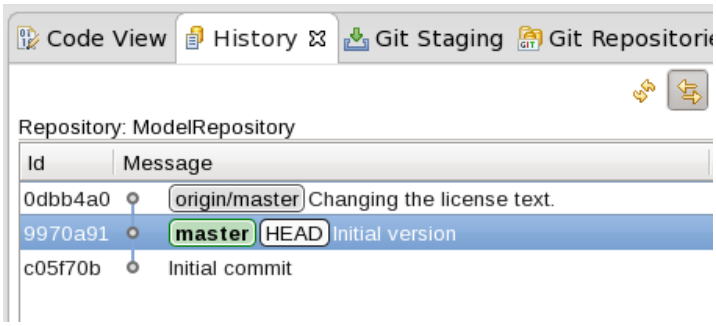


Note that the Fetch command only changes the remote tracking branches of your local repository. No local branches are updated.

Assume for example that someone has updated the remote repository we called ”origin” and we do a Fetch command. The history before the Fetch:



History after Fetch:

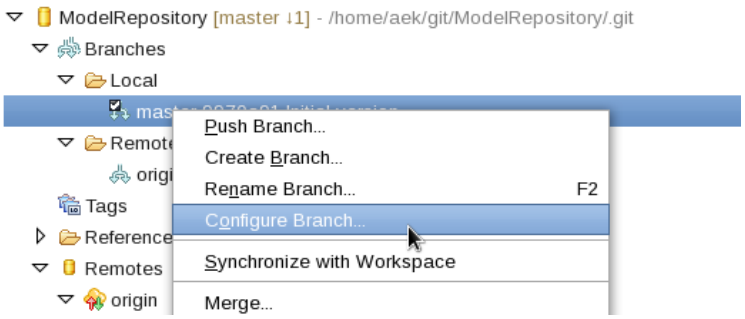


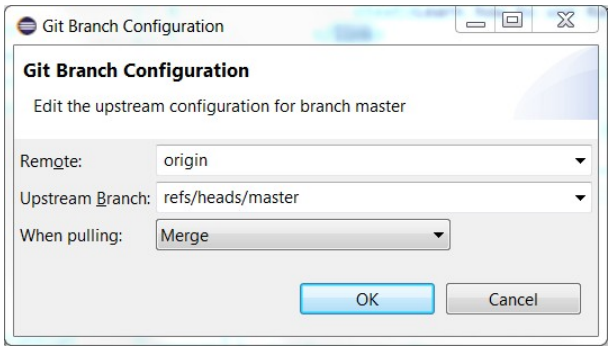
As you can see our local branch ”master” is still on the same commit (that also is what we have checked out, as seen by the ”HEAD” indication). However the ”origin/master” tracking branch indicates that there indeed was a change in the ”origin” remote repository. There was one commit done on the ”master” branch in this repository that we now also got locally in our ”origin/master” remote tracking branch.
In most cases we now want to merge this change into our own local ”master” branch. This can be done exactly as for any other branch using the Merge command (see Merge). However, as we will see in Upstream Configurations below, an easier way is to use the Pull command.

Upstream Configurations

To make it easier to incorporate changes from remote tracking branches to our local branches (and vice versa) it is possible to define an **upstream configuration** for a local branch. If you have used the Clone command to get a copy of a remote repository, this has already been done for you.

To check and update the upstream configuration for a local branch we can use the command Configure Branch from the Git Repositories view.





In this case we can see that the local ”master” branch is defined to have an upstream remote tracking branch that is the ”master” branch on the ”origin” remote server. (You can ignore the ”refs/heads/” part of the name for now.)

The upstream configuration is what enables us to use two commands to update our local repository from the remote and vice versa:

- The Pull command fetches the latest changes and automatically adds them to our local branch. The "When pulling" drop down menu in the Git Branch Configuration dialog allows you to configure how Pull should add the changes to our local branch. By default it will use Merge but you can also configure it to use Rebase.
- The Push command does the opposite. It takes our latest local changes, sends them to the remote repository, commits them and fetches back the changes to the remote tracking branch.

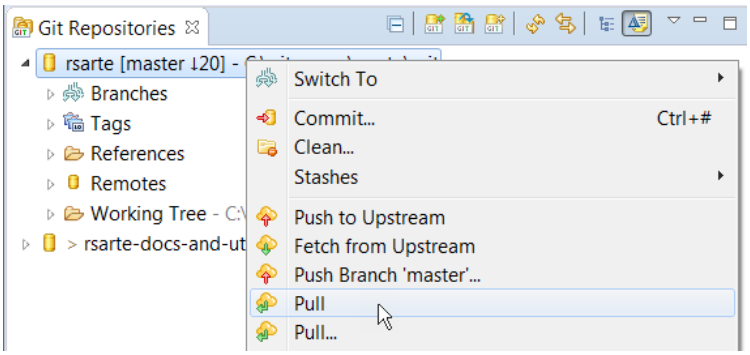
The two commands are described in the sections below.

Update from Remote Repositories: Pull

If you have an upstream configuration for a branch that identifies a remote tracking branch, and thus also identifies a remote repository, you can use the Pull command instead of doing a Fetch followed by a Merge. Pull will first do a Fetch from the remote repository to retrieve the remote changes and then automatically merge the changes (either using Merge or Rebase).

Note that the Pull command is based on the branch that is currently checked out and will change the files you currently have in the working tree similar to what the corresponding merge would have done. Often it can therefore be good to do a Fetch before invoking the Pull, just to be able to inspect what changes will be merged into your branch if you proceed with a Pull.

You can invoke the Pull command for example from the Git Repositories view on any local repository.



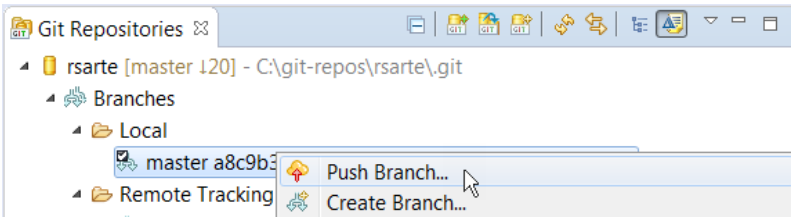
Use "Pull" to perform the Pull according to the upstream configuration for the branch, and use "Pull..." if you want to perform the Pull using different settings.

Updating Remote Repositories with Local Changes using Push

To be able to push your local changes in a simple fashion to a remote repository, you need to have both a remote tracking branch and an upstream configuration as discussed in the sections Remote Tracking Branches and Upstream Configuration. There are other, more advanced ways, to determine what and how to push, see for example The EGit Tutorial or the Git Reference for details. However, the most convenient mechanism is based on using remote tracking branches and the upstream configuration to handle the mapping and that is what we'll use now.

You can invoke the Push command from the command line using the command `git push origin master`. Here ”origin” indicates the remote repository and ”master” is the branch to push.

From Model RealTime you can do a Push from the Git Repositories view. Identify the branch you want to push and use the "Push Branch" command in the context menu.



Collaborating within a Team using Remote Branches

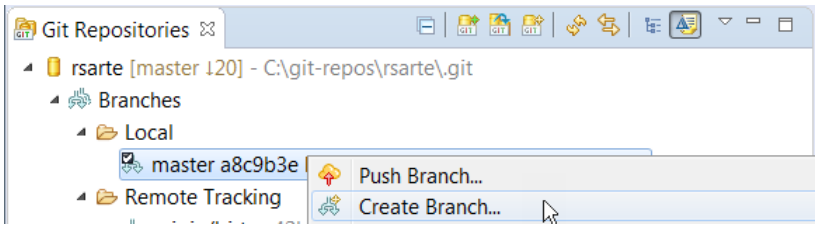
So far in the remote repository discussion we have used the ”master” branch in all examples. This is enough if there is only one team working on a project. All team members have write access to the ”master” branch and they use this branch to share the progress of the work. However, as soon as there is more than one team working on a project the Git remote branch concept is the recommended way for the team members to share work. A remote branch is simply any branch that is available on a remote server that all team members have access to. By using one or more specific branches for the team's work they can use the remote server to share progress while still not disturb the rest of the organization.

Creating a Remote Branch

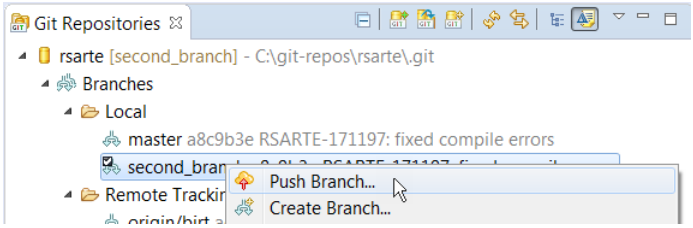
From the command line a remote branch based on the current master branch can be created using two Git commands:

- `git checkout -b mybranch master` to create the branch locally
- `git push -u origin HEAD:mybranch` to push the newly created branch to the remote repository (”origin”), giving it the same name as the local branch. The ”-u” flag makes Git configure a remote tracking branch and an upstream configuration automatically.

You can do the same thing from EGit. Create the branch as usual, for example using the ”Create branch...” command from the Git Repositories view.



Make sure to mark the checkbox "Configure upstream for push and pull" if you plan to use the Push and Pull commands. Then push the new branch to the remote repository.



Configuration of and Basic Operations for Remote Branches

Fortunately, working with a remote team branch is from most points of view no different than working with the master branch as we discussed in the section Working with Remote Repositories. You can configure the remote tracking branch and the upstream configuration the same way, except of course specifying the name of the team branch instead of ”master” in the relevant configuration dialogs.

When the configuration is done you can then use the Fetch, Pull and Push commands to update your local repository from the common repository and vice versa.

Working with Remote Repositories using Gerrit Review Support

In the sections Working with Remote Repositories and Configuration of and Basic Operations for Remote Team Branches we looked at a simple scheme where everybody can both read and write from a common repository without constraints. This works well for a small organization with a limited number of teams working on a reasonably small common project. However, when the number of teams grows and the complexity of the application increases, it's useful to

have some constraints on the process of pushing contents to the master branch in the common repository. For example, you may want to enforce code reviews before changes are delivered.

Git supports many different workflows to suit different teams and organizations. See for example Distributed Git - Distributed Workflows to learn more about some typical workflows. In many workflows there is often one main common repository that represents the official version of the application source code. Then there is a scheme put in place with a "guardian" that controls who can push changes to this common repository and how they can do so.

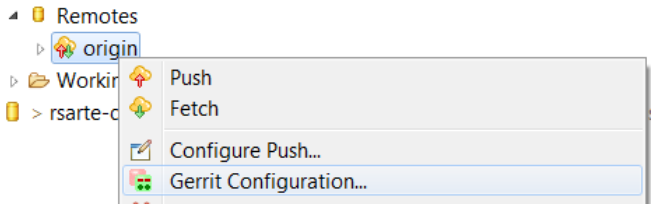
One common scheme that we will look at in this section is to use Gerrit as the guardian (https://www.gerritcodereview.com/). The details of the process that guards the master branch is configurable and is outside the scope of this document. Fortunately the overall workflow as seen from our Model RealTime developer's point of view is similar for most scenarios.

Cloning and Initiating a Repository from Gerrit

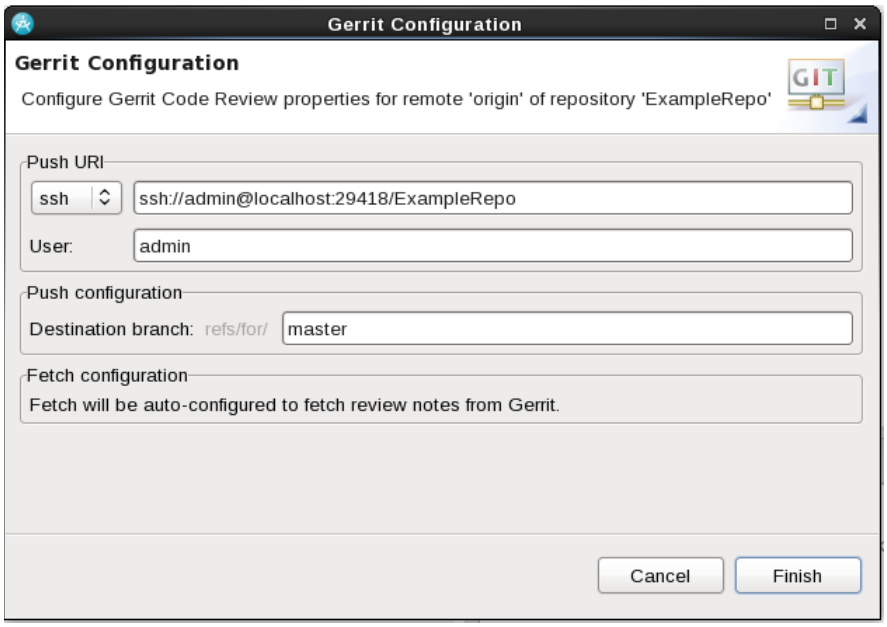
When using Gerrit the common main repository is usually hosted on the Gerrit server. Getting access to a repository from a Gerrit server is no different than from any other remote server. So the process is the same as is described in Accessing a Remote Git Repository The only aspect to be aware of is that you often need to use the ssh protocol to access the Gerrit repository. It is possible to change after cloning, but easiest to handle already from the beginning.

When you have cloned the repository you will need to tell Model RealTime that you want to enable Gerrit specific commands for the repository and identify the remote that is to be used for the Gerrit specific commands.

In some cases the Gerrit configuration will be set up automatically. You can check if it's already set-up by looking at the icon for the repository in the Git Repositories view. This is the case if the icon shows a small green and red decorator: 
In any case you can check and change the configuration using the Gerrit Configuration command that is available in the context menu of the Git Repositories view when a Remote is selected.



The Gerrit configuration makes it possible both to set up the access method and remote address of the Gerrit server as well as the review branch in the Gerrit repository to which you will push review requests.



When the repository is enabled for Gerrit you will find some extra features, in particular the ”Push to Gerrit” and ”Fetch from Gerrit” context menu commands that are available for models in the Project explorer and repositories in the Git Repositories View.

Gerrit Basics

The purpose of Gerrit is to enable some form of review of commits, for example code reviews. To accomplish this two key concepts are used:

- ”change ids”: A unique identifier that identifies a review request in Gerrit.
- ”for” branches: Branches in the Gerrit hosted repository that contains the commits to be reviewed.

In Model RealTime you will see the change ids at the end of commit messages. The recommended practice is to use a Git commit hook that automatically will add the change id when you commit a change in Model RealTime. This can be downloaded from the Git repository and we will assume that you use this hook. The result is that if you check your commits for example in the History view, you will see lines starting with ”Change-Id:” followed by a long identifier added to all your commit messages.

The ”for” branches will show up when you push changes to Gerrit. In Updating the Remote Repository with Your Local Changes using Push we used an upstream configuration to identify a remote tracking branch and used this to identify the branch on the remote server we wanted to push our changes to. We also used the same configuration for both fetch and push. When using Gerrit this is no longer the case. We still use a remote tracking branch for fetching changes, but when pushing we instead push to a special ”for” branch on the Gerrit server. The idea is that someone using Gerrit will review our changes (and potentially do other checks, like building and running tests) and if all is fine deliver the change we proposed. The change will then automatically be merged to the ”real” branch and we can retrieve the new version using a fetch. We will take a close look at the push and review support later in this chapter.

Basic Operations when using a Gerrit Based Repository

A Gerrit server is really no different than any other remote Git server. This means that the commands you will perform for understanding version history, investigating commits, committing local changes etc are all the same. The only difference is the ”change ids” that will appear in commit messages.

Also, fetching changes from remote repositories and working with a team on shared remote branches follows the same principles. The exception being of course the review process that will change the way you push your changes. But actually, depending on how your Gerrit server is configured, you can perhaps also continue to push as described previously in this document, not starting a review process with Gerrit. This depends on the policy used by your organization to handle contributions to the common main repository maintained by the Gerrit server.

Push for Review

When you want to push a commit you have created for review using Gerrit, you will as mentioned before push to a special branch identified by a ”for” prefix. In most cases the branch identification will look something like ”refs/for/mybranch” instead of the usual ”refs/head/mybranch” that we have seen before.

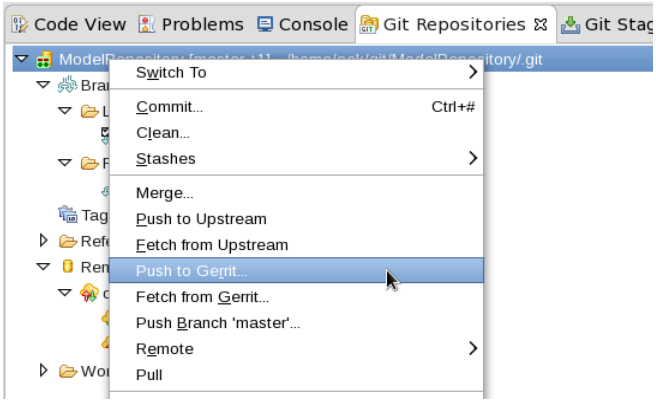
If doing this from command line you would use a command similar to the following:

```
git push HEAD:refs/for/mybranch
```

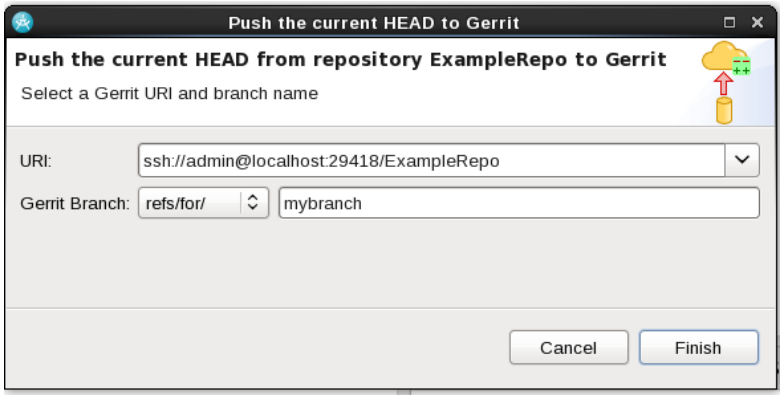
This particular command assumes that:

- ”mybranch” is the name of the branch you want to push to, and
- you have the corresponding local branch checked out so HEAD identifies the commit you want to push.

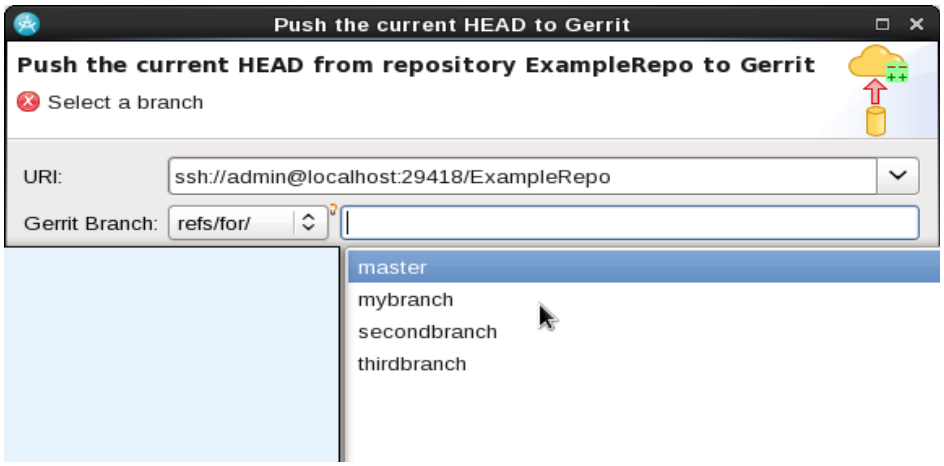
If using the Model RealTime user interface you will have a ”Push to Gerrit...” command available for repositories in the Git Repositories view. Also in this case you will need to have the local branch checked out so HEAD identifies the commit that you want to push for review.



This will pop a dialog where you can specify the push details.



It is worth noting that the ”Gerrit Branch” field supports name completion. If you press Control-Space in this field you can choose between the currently available branches.



After pressing the Finish button your commit becomes available on the Gerrit server for review.

Perform a Review using Mylyn

Mylyn (http://www.eclipse.org/mylyn/) is a task and application lifecycle management framework for Eclipse. From an Model RealTime point of view Mylyn has one very important feature: It enables reviewing Gerrit changes directly from within Model RealTime. Having to use the Gerrit web interface for comparing model files is not very convenient since you then have to look at the textual format for the model files. With Mylyn you can use the Model RealTime Compare editor for looking at the changes.

Mylyn contains many additional features and you can find a good description of both the review support and other features in the Mylyn User Guide. In this section we will only look at the basic steps you are likely to need when reviewing Gerrit change requests.

Installing and Configuring Mylyn

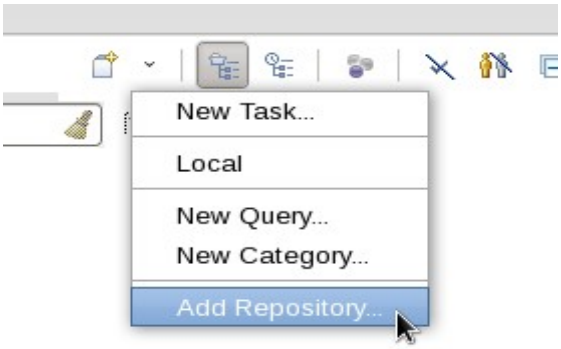
If Mylyn is not installed by default in your Eclipse you can download it from http://www.eclipse.org/mylyn/ and install it using *Help - Install New Software* in Model RealTime.

When installing it is recommended to install at least the following features:

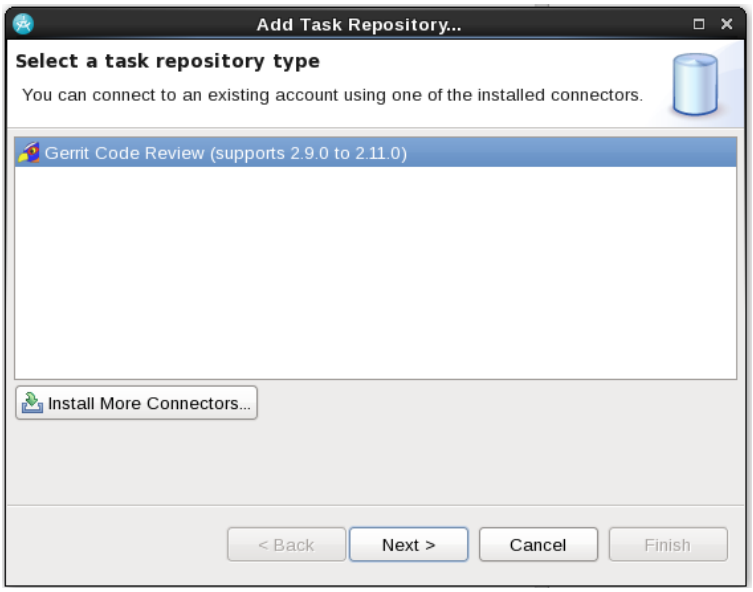
- "Mylyn Task List"
- "Mylyn Task-Focused Interface"
- "Mylyn Reviews Connector: Gerrit"
- "Mylyn Reviews Connector: Gerrit Dashboard"

Before you can use Mylyn for review you also need to connect it to your Gerrit server. This can be done from the ”Task List” view of Mylyn. A convenient way is to open the ”Planning” perspective in Eclipse. This is also a useful perspective for reviewing changes as we will see in the next section.

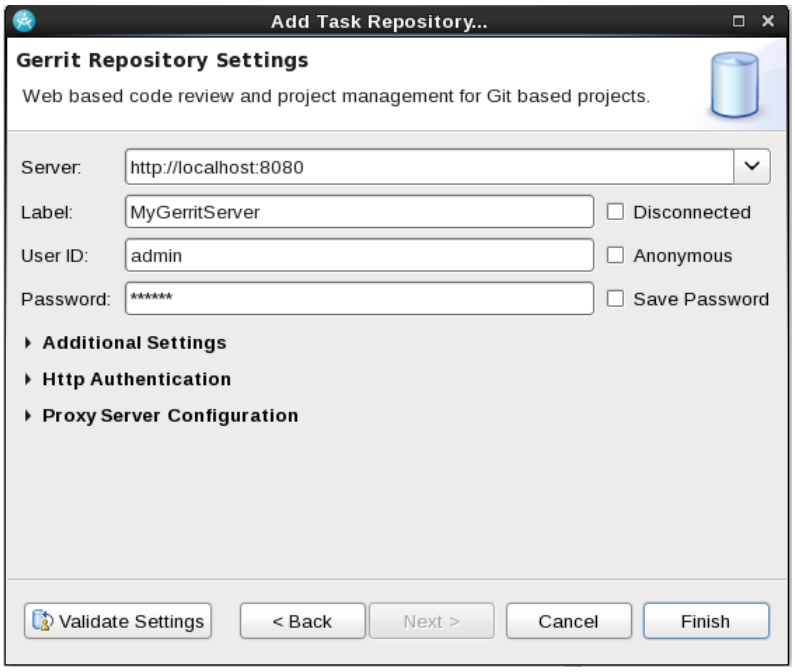
In the Task list you can use the ”Add Repository...” command to connect to Gerrit.



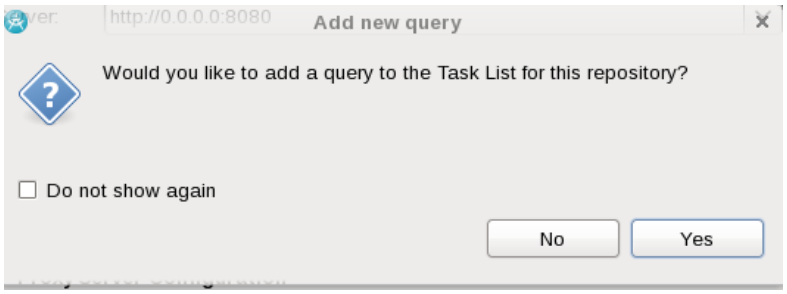
Choose the Gerrit connection if you have more than one Mylyn connection installed.



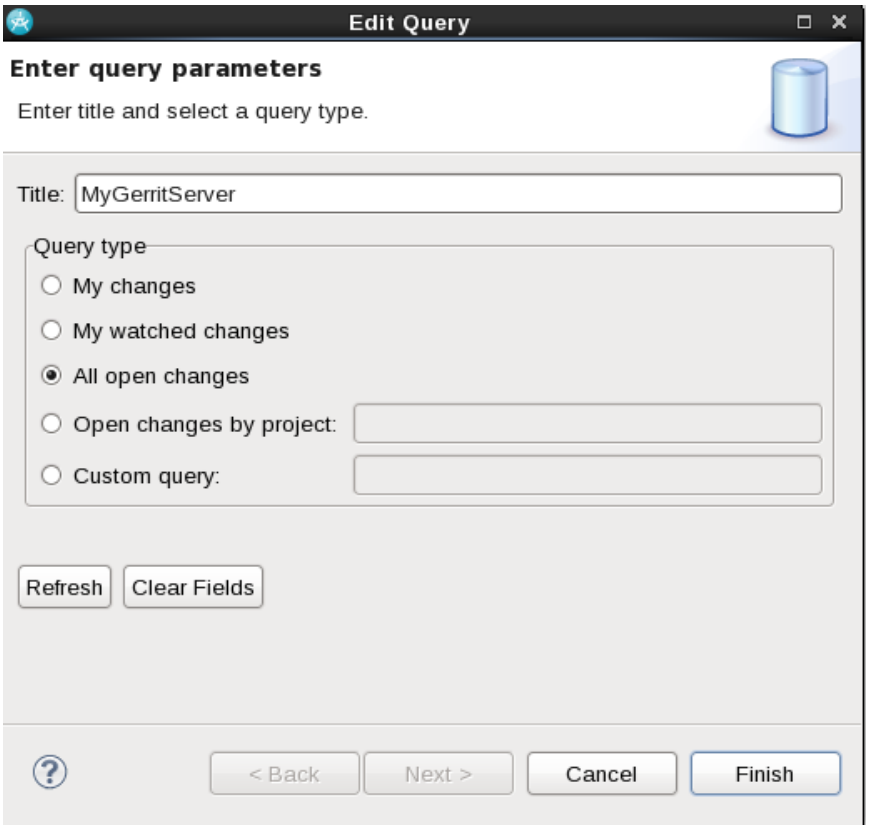
You need to specify the URL of your Gerrit server and your login credentials on the wizard pages that follows.



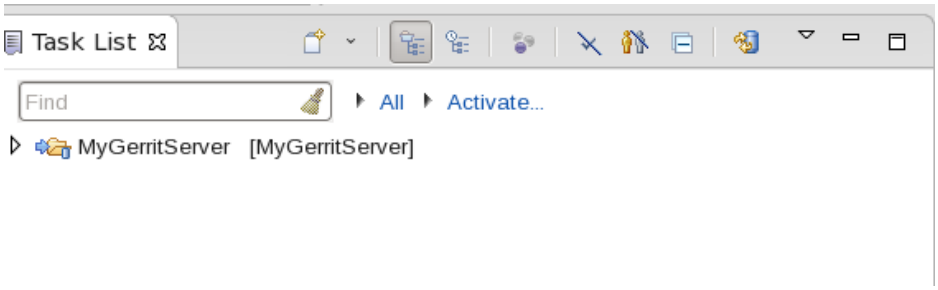
It also makes sense to choose to add a query automatically when prompted.



Select an appropriate Query type to set the scope of what you want to see.

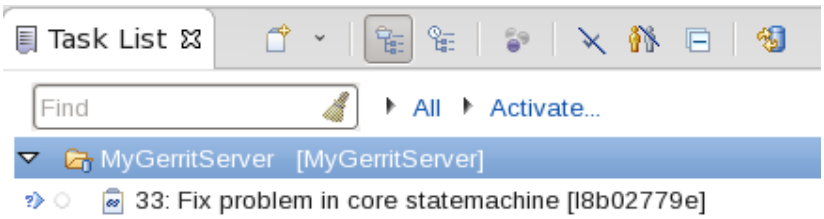


When done you will have a Task List view that is configured to show review requests from your Gerrit server.



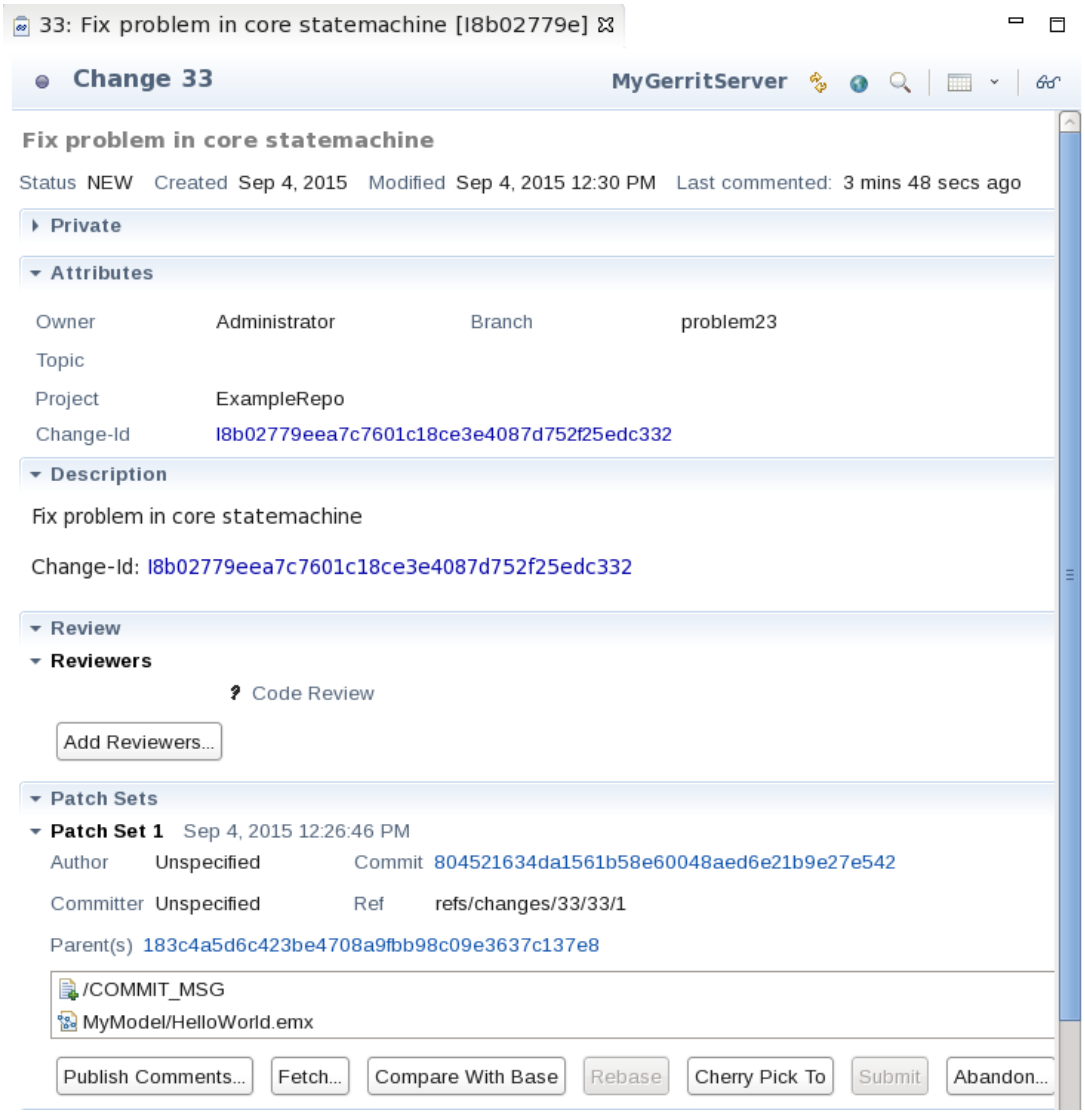
Reviewing a Change Request in Mylyn

When you want to review the changes assigned to you (or other changes), the first step is to look at the ”Task List” view again. The changes that match the query you created will now be visible in the query node in the list.

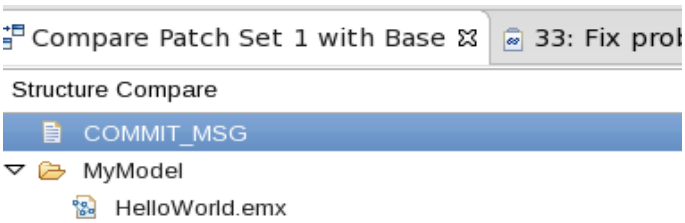


If you don't see what you expect, press ”F5” to refresh the list.

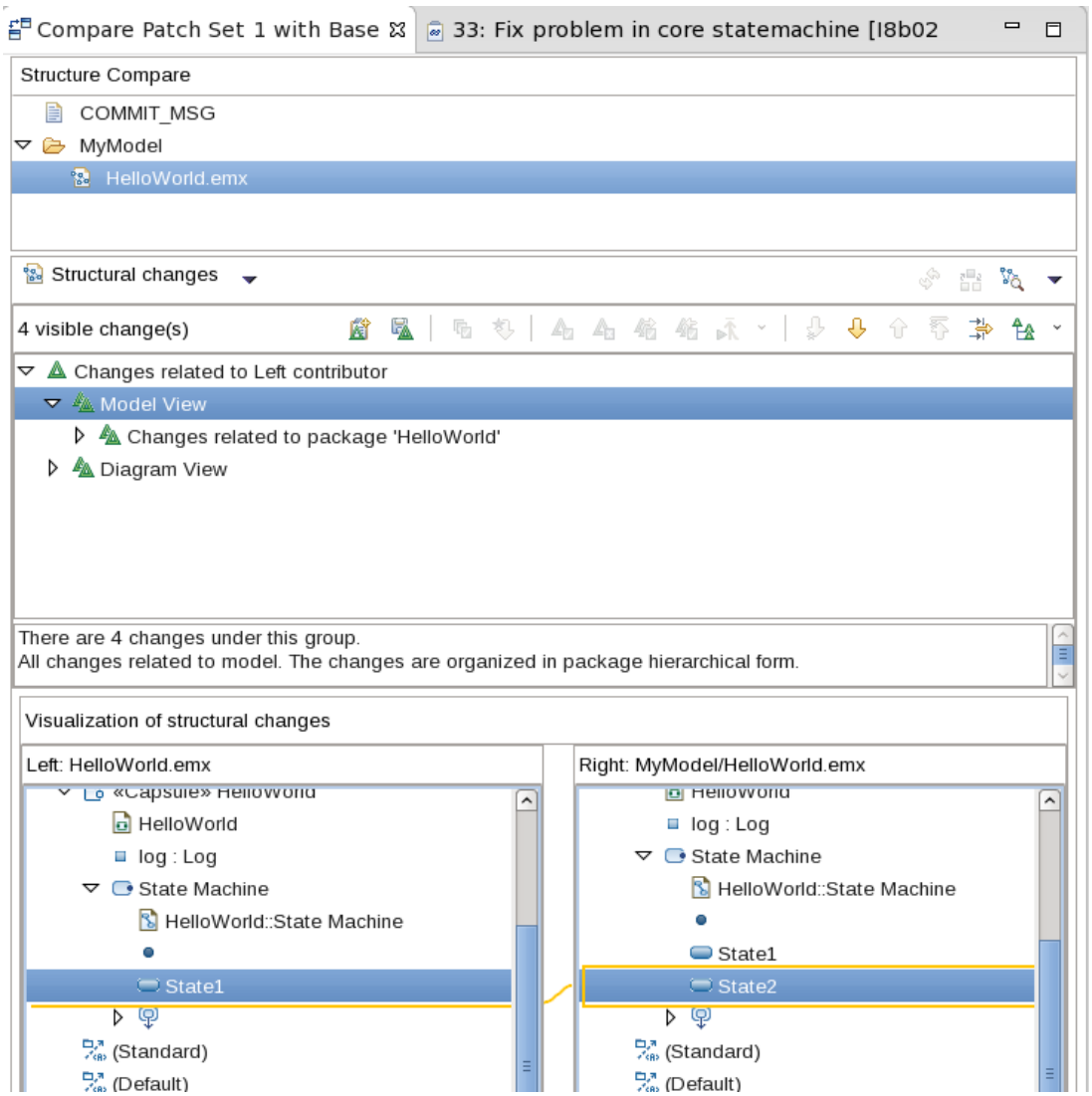
When double-clicking on one of the matches you will get the Mylyn view that contains all relevant information about the review request.



From an Model RealTime point of view the most interesting possibility is that you can directly access the contents of the change set. If you click on the ”Compare with Base” button you will get a structure compare view that shows all changed files.



Double-clicking one of the model files will show you the differences in the Model RealTime compare editor.



You can now add comments to the review and take any action that is necessary and the information will be pushed back to the Gerrit review server exactly as if you would have accessed it from the web interface.

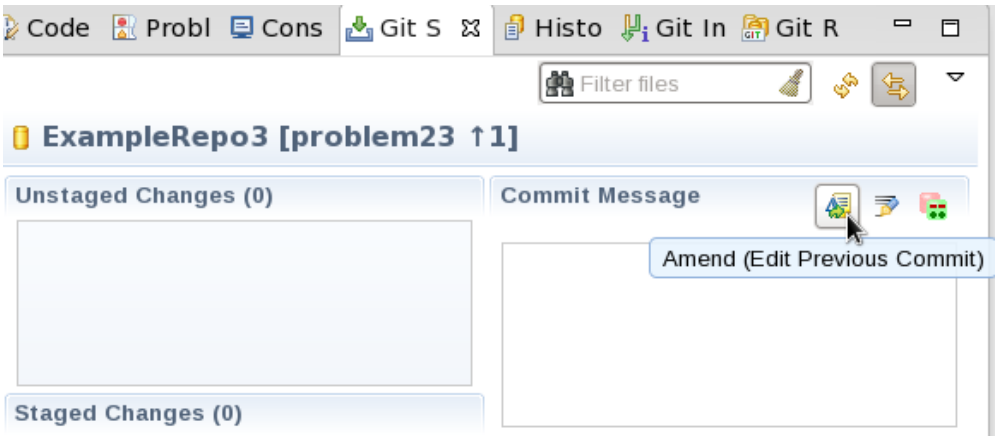
Using Amend Commit to Fix Review Comments

If you have pushed a commit for review to Gerrit, and the reviewer had comments that you would like to fix, then the Git support for amending commits is very handy.

Amending a commit in Git essentially means to make edits to the changes that already have been committed. So it gives you a possibility to re-do some changes.

This can be done using the Model RealTime user interface in the following way:

- Checkout the commit that you want to change
- Open the Git Staging view
- Click on the ”Amend Commit” button



You now have the freedom to make any modifications you like to fix the review comments. When done, stage the changes as usual.

You can now push the commit to Gerrit again as described in Push for Review.
Remember that to make this work in a Gerrit context it is important that you do not change the Change-Id line in the commit comments. You want to make sure that the Change-Id is left as it was to ensure that your fixes will be added to the same review request in Git, instead of creating a new review request.

Other Common Scenarios – Stashing, Resetting and Tagging

In this section we'll go through a few more common actions you likely will need to know about: Stashing, resetting and tagging

Stashing – Temporarily Saving Your Changes

The purpose of the Git stashing feature is to make it possible to save changes you've done in your working tree but that you for some reason don't want to commit to the repository at this time. The typical example is where you are half-way through a change and don't want to com-

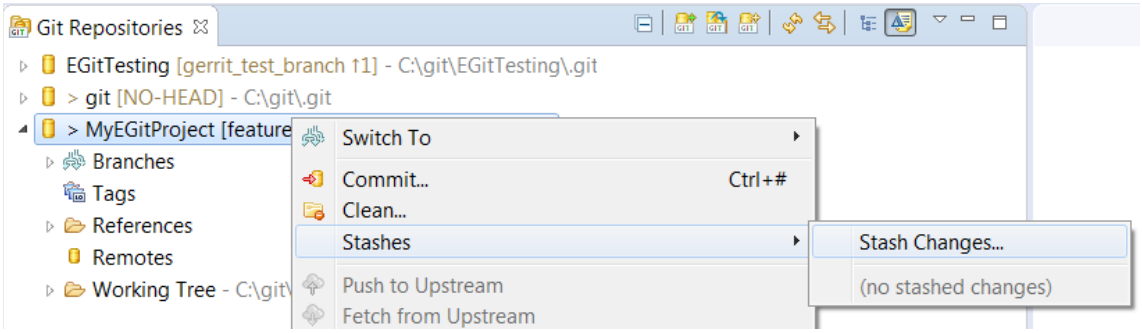
mit it when you get some more urgent task you need to handle. Stashing enables you to store them in a safe place and get them back later. Stashing can thus be seen as a quick alternative to creating a temporary branch and commit the changes on that branch.

The mechanism has two parts:

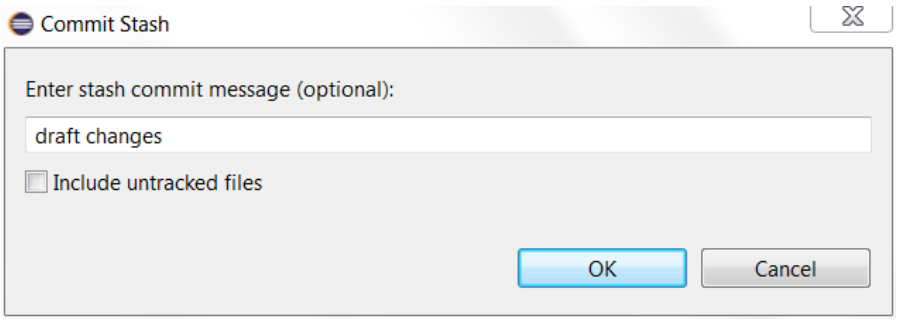
- Save your changes: **stash** them
- Get your changes back: **unstash** them

For command line handling of stashes, and more information about stashing see for example https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning. In this section we'll focus on the Model RealTime user interface actions.

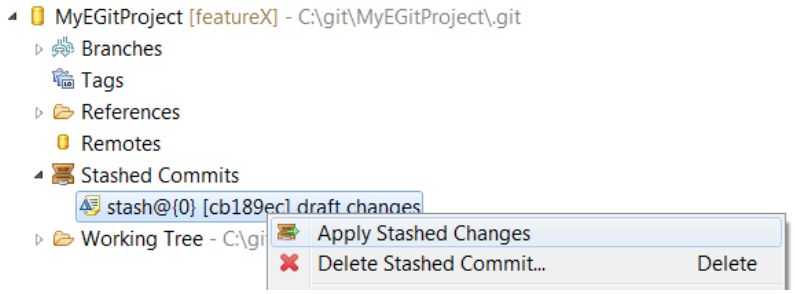
To stash your current changes you can use the “Stashes - Stash Changes” command that is available in the context menu of a repository in the Git Repositories view. This saves both the modified files in your working tree and the files you have added to the Git index.



If you want you can add a short description of your changes before storing them.



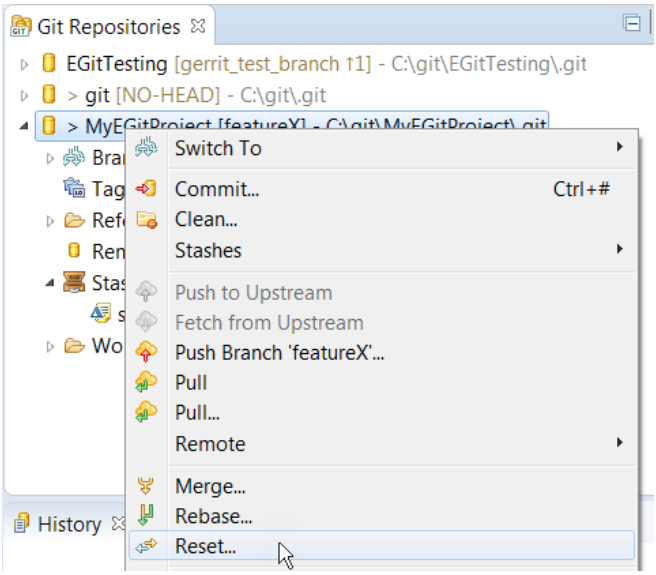
The stashed changes appear in the Git Repositories view. To get your changes back to the working tree, right click on the "stash" and perform the "Apply Stashed Changes" command.



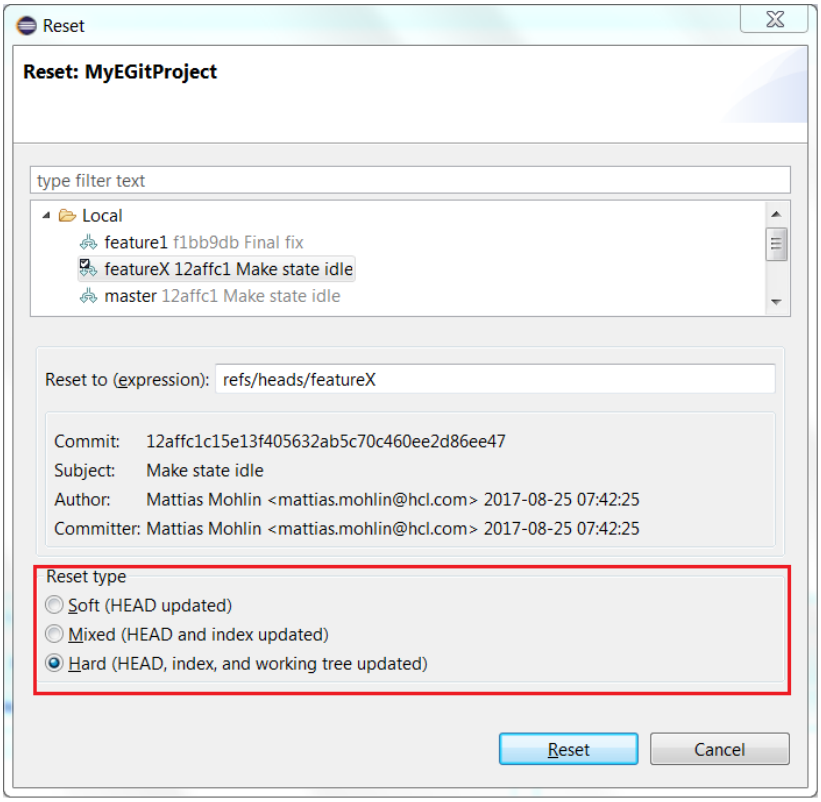
Note that applying stashed changes do not automatically delete the stash. Perform the command "Delete Stashed Commit" when you no longer need the stash.

Reset – Reverting Your Changes

Sometimes it's useful to discard all your current changes in the working tree and revert to the last stable version of the history. Git provides the “Reset” command for doing this. You can perform “Reset” for example from the context menu of a repository in the Git Repositories view.



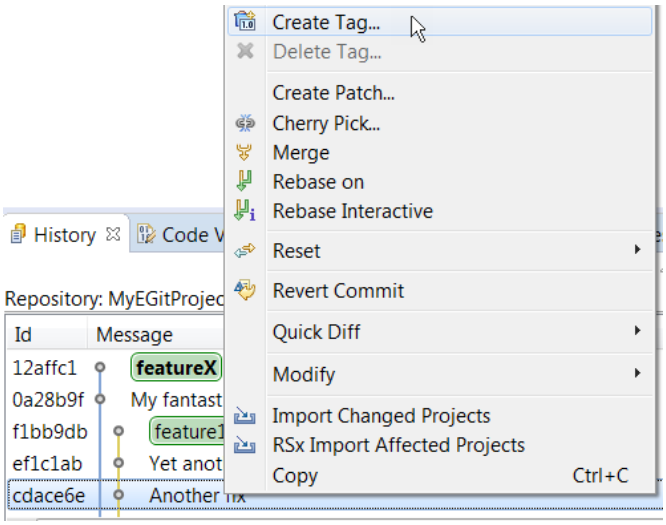
There are three different variants of Reset: Soft, Mixed and Hard. The first two variants, Soft and Mixed, do not actually revert your working tree changes. See http://git-scm.com/docs/git-reset for details when these variants can be useful. The Hard Reset is what you most often want to use as it will revert all your current changes to the working tree.



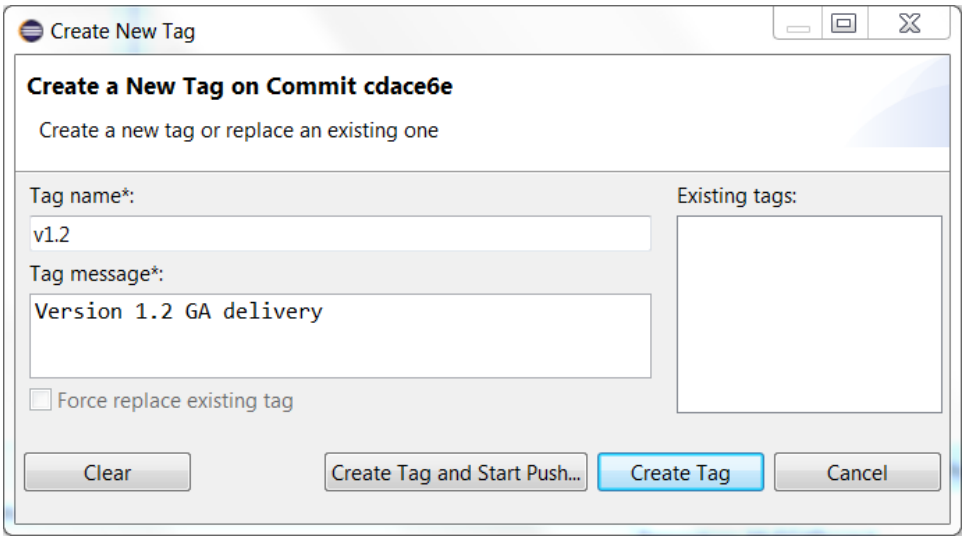
Tagging – Marking Important Commits

A tag is a label you can put on a specific commit. It is typically used to mark important commits, for example to represent a release or other external delivery.

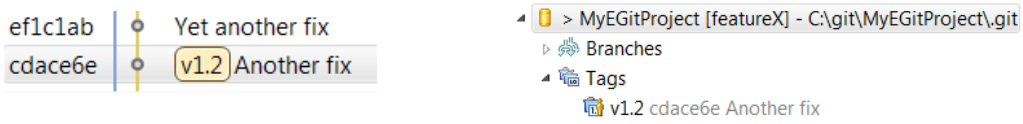
You can create a tag for any commit in the History view using the ”Create Tag” command in the context menu.



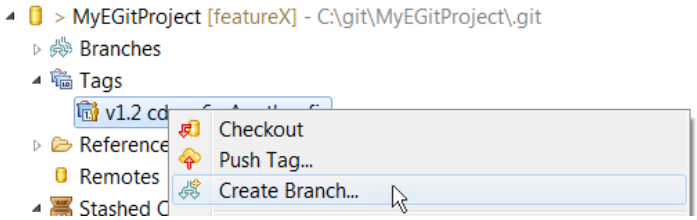
You will be prompted for the name of the tag, and can also write a brief description of what it means.



Tags are shown in the History view as a label in front of the commit message. They are also shown in the Tags folder in the Git Repositories view.



From there you can also easily create a new branch based on a tag.



Integration with Command Line Git

In this section we will look at what is necessary in order to make command line Git commands that trigger a merge operation to work for Model RealTime models. Note that this is a necessary precondition to safely work with Model RealTime models from the Git command line. Without the merge integration described below, Git merge commands may trigger a default text based merge that easily can create corrupt models.

We will also cover some other aspects of integrating Model RealTime with a Git customization using hooks.

Invocation from Command Line

The integration of Model RealTime into Git command line handling is primarily done using the Model RealTime command line tool for compare/merge. This is implemented by a Java application called `cmcmdline.jar` that is available in the Model RealTime installation. It's

located in the plugin folder for the com.ibm.xtools.comparemerge.team plugin. For example (the part in boldface depends on the installed version of Model RealTime):

```
<install-dir>\plugins\
com.ibm.xtools.comparemerge.team_7.60.100.v20180618_1038\utm
```

The details of this command line application is given in the chapter ”Command Line Tool for Compare/Merge” in ”Comparing and Merging Models”. See also the reference guides for the command line tools available from the Model RealTime Compare/Merge documentation page. In this document we will only cover what is needed for Git integration.

Merge Integration

Git allows the definition of ”merge drivers” in order to make it possible to integrate tools that use a special-purpose storage format (like Model RealTime does) with the Git commands. A description on how this is done can be found in http://git-scm.com/docs/gitattributes. See in particular the section ”Defining a custom merge driver”.

The following is needed:

- A definition of the merge drivers in the Git configuration (e.g. in `.git/config` in your repository)
- A mapping of the Model RealTime file extensions to the defined drivers using Git attributes (e.g. in the `.git/info/attributes` file in your repository)

Let's look at an example config definition defining merge for the three most common types of files handled by Model RealTime (.emx, .emf and .tc). As you can see the only difference between the drivers is that the extension is passed to the command line tool using the ”-ext” argument for emx/efx files. This is shown in bold below. The installation specific path to `cmcmdline.jar` that you need to modify (see above) are shown in bold and underlined.

```
[merge "rtistemx"]
         name = Model RealTime EMX merge driver
         driver = java -cp /path-to-cmcmdline-tool/cmcmdline.jar com.ibm.xtool-
s.comparemerge.cmcmdline.CMTool merge -ext=emx -ancestor %O -left %A -right %B
-out %A
[merge "rtistefx"]
         name = Model RealTime EFX merge driver
         driver = java -cp /path-to-cmcmdline-tool/cmcmdline.jar com.ibm.xtool-
s.comparemerge.cmcmdline.CMTool merge -ext=efx -ancestor %O -left %A -right %B
-out %A
[merge "rtisttc"]
         name = Model RealTime TC merge driver
         driver = java -cp /path-to-cmcmdline-tool/cmcmdline.jar com.ibm.xtool-
s.comparemerge.cmcmdline.CMTool merge -ancestor %O -left %A -right %B -out %A
```

An example definition from a Git attributes file:

```
*.emx merge=rtistemx
*.efx merge=rtistefx
*.tc merge=rtisttc
```

When these (or similar) definitions are done, all command line Git commands that trigger a merge Invocation will correctly handle Model RealTime model files. Note however that this

will only trigger a file-by-file merge. To invoke a logical model merge or a closure merge either use the Model RealTime graphical user interface or call the command line application `cmcmdline.jar` directly as is describe in section ”Command Line Tool for Compare/Merge” in
”Comparing and Merging Models”.

Also note that this integration only handles ”merge”, not the conflict resolution that requires a ”merge tool” integration. Currently we recommend the usage of the graphical user interface in Model RealTime to handle conflict resolution.

Git Hooks

Git allows various hooks to be associated with various actions. In general this works fine together with Model RealTime, but there is one exception: Scripts that require user interaction, for example asking for confirmation of some action, will NOT work from the Model RealTime
user interface. From a users point of view it will appear as if the tool hangs when this happens. A typically example is a pre-commit hook that asks for user input. This will for example be triggered when pressing the ”Commit” button in the Git Staging View and the tool will appear to hang.