<https://github.com/atetrel/traceCleaner>

<http://www.vogella.com/tutorials/EclipseGit/article.html>

Git version control with Eclipse (EGit) - Tutorial

Lars Vogel

Version 3.9

Copyright © 2009, 2010, 2011, 2012, 2013, 2014, 2015 vogella GmbH

**Git with Eclipse (EGit)**

This tutorial describes the usage of EGit; an Eclipse plug-in to use the distributed version control system Git. This tutorial is based on Eclipse 4.5 (Mars).

Table des matières

[Lars Vogel 1](#_Toc431820021)

[1 Completely new to Git? 5](#_Toc431820022)

[2 What is Git for Eclipse (EGit)? 5](#_Toc431820023)

[3 The Eclipse workspace and Git repositories 5](#_Toc431820024)

[4 Command line Git 5](#_Toc431820025)

[5 Installation of Git support into Eclipse 5](#_Toc431820026)

[6 Setting up Git in Eclipse 6](#_Toc431820027)

[6.1 Interoperability of Git command line settings with the Eclipse IDE 6](#_Toc431820028)

[6.2 Git user settings in Eclipse 6](#_Toc431820029)

[6.3 Default clone location 7](#_Toc431820030)

[6.4 Configuring the toolbar and the menu for Git usage 7](#_Toc431820031)

[7 Exercise: Git configuration for the Eclipse IDE 9](#_Toc431820032)

[7.1 Validate your Git user settings 9](#_Toc431820033)

[7.2 Validate the default clone location 11](#_Toc431820034)

[7.3 Configure the toolbar with your preferred Git commands 11](#_Toc431820035)

[8 Configure files and directories to ignore 12](#_Toc431820036)

[8.1 Ignoring files and directories with a .gitignore file 12](#_Toc431820037)

[8.2 Global (cross-repository) .gitignore settings 12](#_Toc431820038)

[8.3 Local per-repository ignore rules 13](#_Toc431820039)

[9 Using the Git Repositories view 13](#_Toc431820040)

[9.1 Using the Git Repositories view 13](#_Toc431820041)

[9.2 Content in the Git Respositories view 14](#_Toc431820042)

[9.3 Open a commit 15](#_Toc431820043)

[9.4 Commands available in the Commit Viewer 16](#_Toc431820044)

[10 Git integration into the Package and the Project Explorer 17](#_Toc431820045)

[11 Using the Git Staging view 18](#_Toc431820046)

[12 Using the Git commit dialog 19](#_Toc431820047)

[13 Using the History view 19](#_Toc431820048)

[13.1 Purpose of the history view 19](#_Toc431820049)

[13.2 Review the repository history via the History view 20](#_Toc431820050)

[13.3 The History view filters 22](#_Toc431820051)

[13.4 Using search 23](#_Toc431820052)

[13.5 Showing details of a commit 24](#_Toc431820053)

[14 Working with one or multiple projects in a Git repository 26](#_Toc431820054)

[14.1 Workspace and projects 26](#_Toc431820055)

[14.2 Adding a new project to a Git repository 26](#_Toc431820056)

[15 Exercise: Working with a Git repository in Eclipse 28](#_Toc431820057)

[15.1 Target: Using Eclipse Git for a local repository 28](#_Toc431820058)

[15.2 Create a new Git repository via Eclipse 28](#_Toc431820059)

[15.3 Create .gitignore file 29](#_Toc431820060)

[15.4 Creating an Eclipse project 29](#_Toc431820061)

[15.5 Put project under version control 30](#_Toc431820062)

[15.6 Using the Git Staging view for the initial commit 32](#_Toc431820063)

[15.7 Using the Git Staging view for committing changes 33](#_Toc431820064)

[15.8 Commit more files 34](#_Toc431820065)

[15.9 Review your commit history via the History view 34](#_Toc431820066)

[15.10 Open an older version of a file via the History view 35](#_Toc431820067)

[16 Exercise: Sharing multiple projects into one Git repository 35](#_Toc431820068)

[16.1 Create a new Git repository 35](#_Toc431820069)

[16.2 Create two Java projects 35](#_Toc431820070)

[16.3 Share projects 35](#_Toc431820071)

[16.4 Validate that projects have been moved 36](#_Toc431820072)

[16.5 Create your initial commit 36](#_Toc431820073)

[17 Clone an existing repository 37](#_Toc431820074)

[18 Import projects from an existing repository 42](#_Toc431820075)

[19 Performing Git operations in Eclipse 46](#_Toc431820076)

[19.1 Pull, push and fetch 46](#_Toc431820077)

[19.2 Basic team operations 46](#_Toc431820078)

[19.3 Team operations available on the project 48](#_Toc431820079)

[19.4 Amending a commit 50](#_Toc431820080)

[20 Branching in Eclipse 50](#_Toc431820081)

[21 Starting a merge operation in Eclipse 50](#_Toc431820082)

[21.1 Merge 50](#_Toc431820083)

[21.2 Solving merge conflicts 52](#_Toc431820084)

[22 Rebasing a branch onto another branch via Eclipse 53](#_Toc431820085)

[23 Git reset and Git reflog 55](#_Toc431820086)

[23.1 git reset 55](#_Toc431820087)

[23.2 Finding "invisible" commits with the Reflog view 56](#_Toc431820088)

[24 Using git cherry-pick 56](#_Toc431820089)

[25 Creating patches 58](#_Toc431820090)

[26 Blame annotations 60](#_Toc431820091)

[27 Stash via the Git repository view 61](#_Toc431820092)

[28 Adjusting the history with interactive rebase 63](#_Toc431820093)

[28.1 Support for interactive rebase in Eclipse 63](#_Toc431820094)

[28.2 Actions available via the History view 63](#_Toc431820095)

[28.3 28.3. Using the Git Interactive Rebase view 64](#_Toc431820096)

[29 29. Using Eclipse Git with GitHub 69](#_Toc431820097)

[29.1 Clone project 69](#_Toc431820098)

[29.2 Push changes 69](#_Toc431820099)

[30 Eclipse support for SSH based authentication 70](#_Toc431820100)

[31 Eclipse integration with GitHub 70](#_Toc431820101)

[31.1 The Eclipse Mylyn plug-in 70](#_Toc431820102)

[31.2 GitHub issue integration 71](#_Toc431820103)

[31.3 Manage pull requests in Eclipse 72](#_Toc431820104)

[31.4 Import projects directly from GitHub 75](#_Toc431820105)

[31.5 More infos about the GitHub Mylyn integration 75](#_Toc431820106)

[32 32. Writing good commit messages 75](#_Toc431820107)

[32.1 Importance of Git commit messages 75](#_Toc431820108)

[32.2 Guidelines for useful commit messages 76](#_Toc431820109)

[32.3 Example message 76](#_Toc431820110)

[32.4 Good and bad example for a Git history 76](#_Toc431820111)

[33 Contributing to EGit - Getting the source code 77](#_Toc431820112)

[34 Git series 77](#_Toc431820113)

[35 Get the Book 77](#_Toc431820114)

[36 About this website 77](#_Toc431820115)

[37 37. Links and Literature 78](#_Toc431820116)

[37.1 EGit Resources 78](#_Toc431820117)

[37.2 vogella GmbH training and consulting support 78](#_Toc431820118)

# Completely new to Git?

In case you are completely new to the Git version control system you might want to read more about its concepts. For a detailed description of the Git concepts and different options please see the corresponding [Git book from Lars Vogel](http://www.vogella.com/books/git.html) or the [Git online tutorial](http://www.vogella.com/tutorials/Git/article.html).

# What is Git for Eclipse (EGit)?

The Eclipse IDE has excellent support for the Git version control system. This support is provided via a set of plug-ins (software component) from the EGit project.

The EGit functionality is based on the JGit library. JGit is a library which implements the Git functionality in Java.

# The Eclipse workspace and Git repositories

It is good practice to place your Git repositories outside the Eclipse workspace. This separates your Git repository from any additional meta-data which Eclipse might create. By default, Eclipse Git uses the git folder in the users home directory to clone new repositories. This default location can of course be adjusted, see [Section 6.3, “Default clone location”](http://www.vogella.com/tutorials/EclipseGit/article.html#egitconfiguration_defaultlocation) for more information on this.

# Command line Git

This tutorial describes the usage of EGit. If you want to learn about the usage of the Git command line, you can use the [Git Tutorial](http://www.vogella.com/tutorials/Git/article.html) as a reference.

This tutorial also explains the basic Git terminology, e.g., what is a commit, branch, etc.

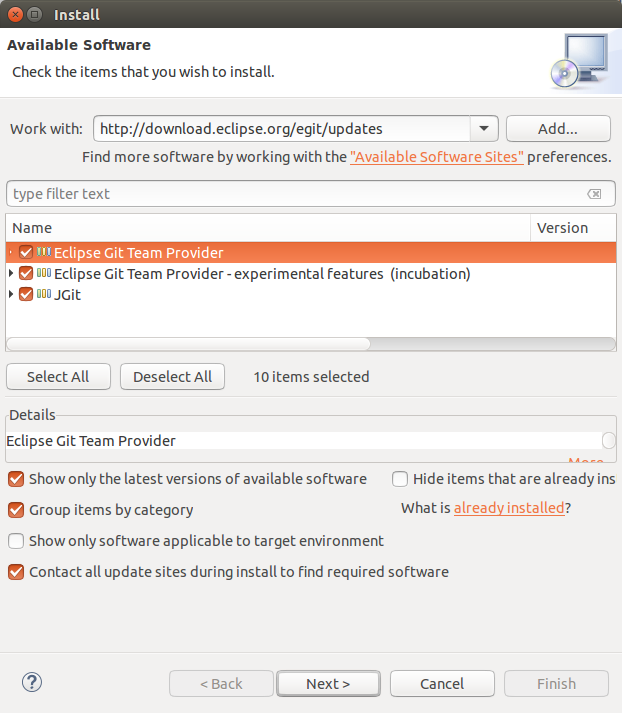
# Installation of Git support into Eclipse

Most Eclipse IDE downloads from Eclipse.org contain support for Git in their default configuration. In this case no additional installation is required.

If the Git functionality is missing in your Eclipse IDE installation, you can install it via the Eclipse installation manager. Start this manager via the Help → Install new Software... menu entry by using the following update site URL:

http://download.eclipse.org/egit/updates

The dialog to install the Eclipse Git team provider is depicted in the following screenshot.



# Setting up Git in Eclipse

## Interoperability of Git command line settings with the Eclipse IDE

The Git functionality in the Eclipse IDE uses the same configuration files as the Git command line tools. This makes it easier to use the Eclipse Git tooling and the command line tooling for Git interchangeable.

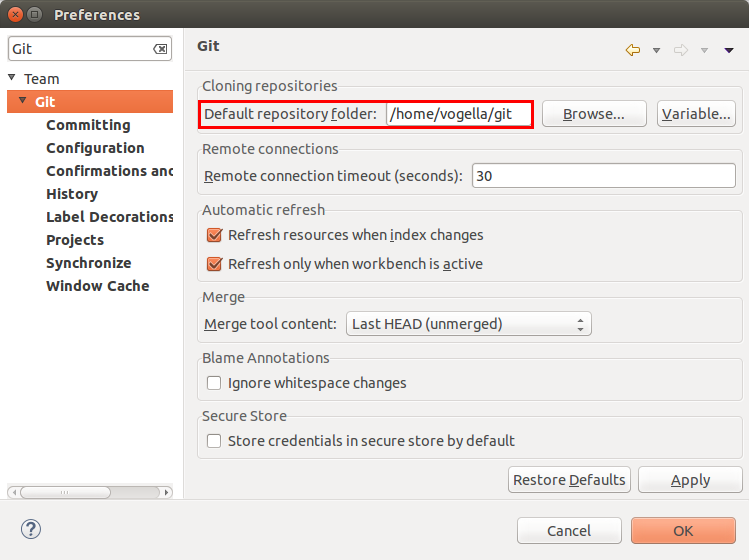
## Git user settings in Eclipse

To use Git you must configure your full name and email address. This information is used to fill the author and committer information of commits you create. These Git configuration settings can be adjusted via the Eclipse preference setting. Select Window → Preferences → Team → Git → Configuration to see the current configuration and to change it.

You can add entries to your Git configuration by pressing the Add Entries button on the Git Configuration preference page.

## Default clone location

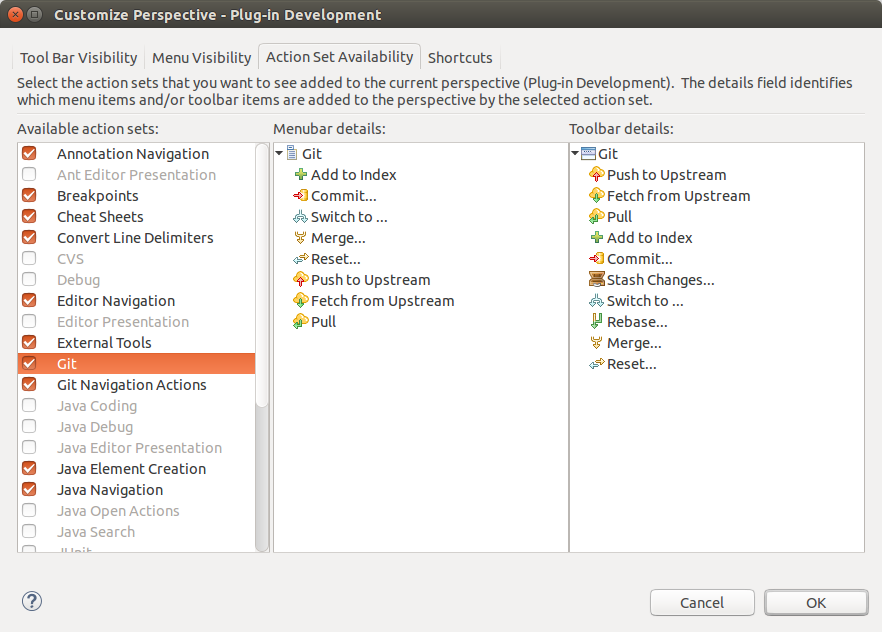
If you clone a new repository via Eclipse Git, it will create by default a new sub-folder for the new Git repository in a default directory. This default path can be configured via the Windows → Preferences → Team → Git entry in the Default Repository folder field.



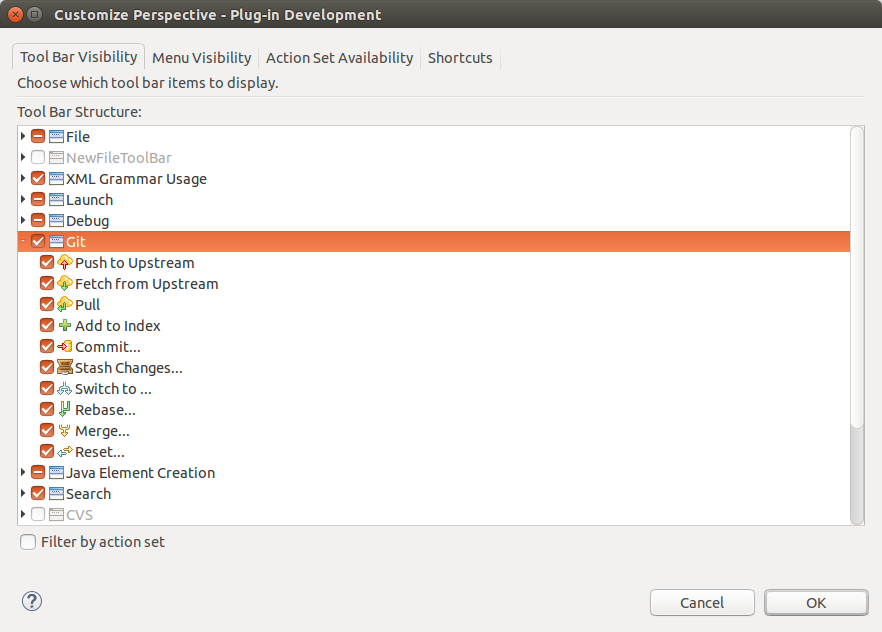
You can also use Eclipse configuration variables to define this path, e.g., if you want to store repositories in the folder "git" under the Eclipse workspace you may use ${workspace\_loc}/git.

## Configuring the toolbar and the menu for Git usage

To simplify access to the common Git operations you can activate the Git toolbar. For this select Window+Perspective → Customize perspective... and check the Git and Git Navigation Actions entries in the Action Set Availability tab.



Afterwards you can configure which Git operations should be available via the Tool Bar Visibility or the Menu Visibility tab.

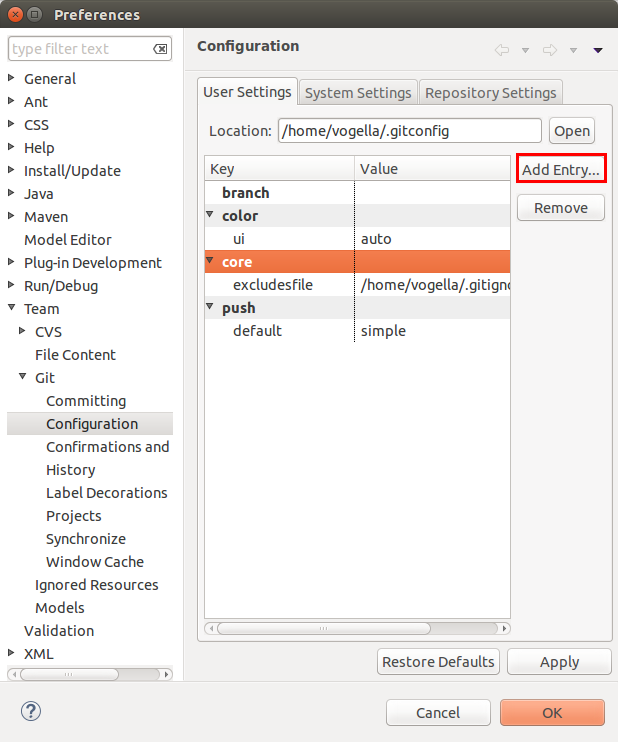


# Exercise: Git configuration for the Eclipse IDE

## Validate your Git user settings

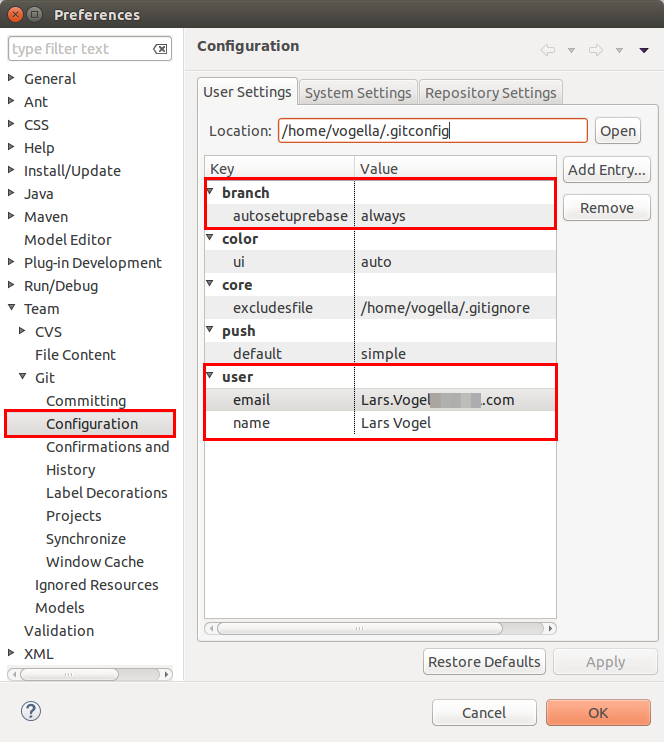
Select Window → Preferences → Team → Git → Configuration and add your full name and email setting to your user settings if this is not yet configured.

The user should be stored under the *user.name* key with the full name as value. If it is not available press the Add Entry... button. Repeat the procedure for your email address via the *user.email* key.



Unless you have a different opinion about rebase and merge commits, also set the *branch.autosetuprebase* parameter to *always*. This is common setting for Git repositories to avoid merge commits if you pull from a remote repository and have divergent changes and instead rebases your local branch on the remote branch it tracks.

After this exercise, the configuration should look similar to the following screenshot.



## Validate the default clone location

Use [Section 6.3, “Default clone location”](http://www.vogella.com/tutorials/EclipseGit/article.html#egitconfiguration_defaultlocation) to see where the Eclipse Git tooling will clone Git repositories to.

## Configure the toolbar with your preferred Git commands

Use [Section 6.4, “Configuring the toolbar and the menu for Git usage”](http://www.vogella.com/tutorials/EclipseGit/article.html#egitconfiguration_toolbar) to add your favorite Git commands to the toolbar.

# Configure files and directories to ignore

## Ignoring files and directories with a .gitignore file

Git can be configured to ignore certain files and directories for repository operations. This is configured via one or several .gitignore files. Typically, this file is located at the root of your Git repository but it can also be located in sub-directories. In the second case the defined rules are only valid for the sub-directory and below.

You can use certain wildcards in this file. *\** matches several characters. More patterns are possible and described under the following URL: [gitignore manpage](https://www.kernel.org/pub/software/scm/git/docs/gitignore.html)

For example, the following .gitignore file tells Git to ignore the bin and target directories and all files ending with a ~.

# ignore all bin directories

# matches "bin" in any subfolder

bin/

# ignore all target directories

target/

# ignore all files ending with ~

\*~

You can create the .gitignore file in the root directory of the working tree to make it specific for the Git repository.

Tip

The .gitignore file tells Git to ignore the specified files in Git commands. You can still add ignored files to the staging area of the Git repository by using the --force parameter, i.e. with the git add --force [paths] command.

This is useful if you want to add, for example, auto-generated binaries, but you need to have a fine control about the version which is added and want to exclude them from the normal workflow.

It is good practice to commit the local .gitignore file into the Git repository so that everyone who clones this repository have it.

## Global (cross-repository) .gitignore settings

You can also setup a global .gitignore file valid for all Git repositories via the core.excludesfile setting. The setup of this setting is demonstrated in the following code snippet.

# Create a ~/.gitignore in your user directory

cd ~/

touch .gitignore

# Exclude bin and .metadata directories

echo "bin" >> .gitignore

echo ".metadata" >> .gitignore

echo "\*~" >> .gitignore

echo "target/" >> .gitignore

# for Mac

echo ".DS\_Store" >> .gitignore

echo ".\_\*" >> .gitignore

# Configure Git to use this file

# as global .gitignore

git config --global core.excludesfile ~/.gitignore

The global .gitignore file is only locally available.

## Local per-repository ignore rules

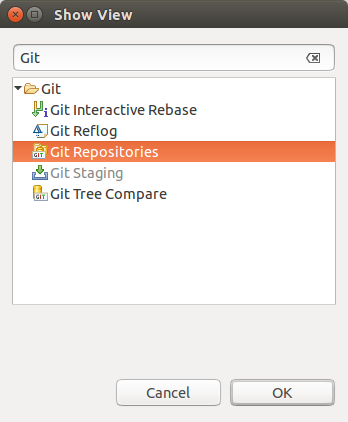
You can also create local per-repository rules by editing the .git/info/exclude file in your repository. These rules are not committed with the repository so they are not shared with others.

This allows you to exclude, for example, locally generated files.

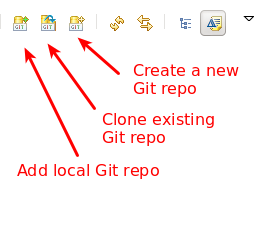
# Using the Git Repositories view

## Using the Git Repositories view

Eclipse Git provides the Git Repositories view which allows you to browse your repositories, add or initialize local repositories or clone remote repositories, checkout projects, manage your branches and much more. You can open this view via Window → Show View → Other... → Other... → Git → Git Repositories

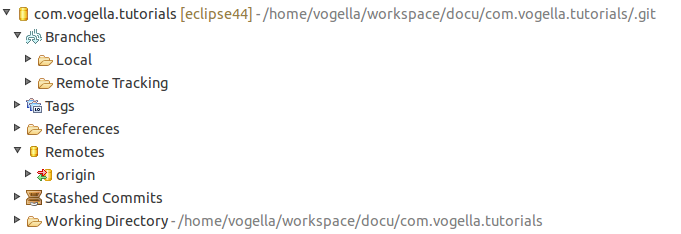


The toolbar entries allow you to add an existing local Git repository to the view, clone a Git repository and to create a new Git repository.

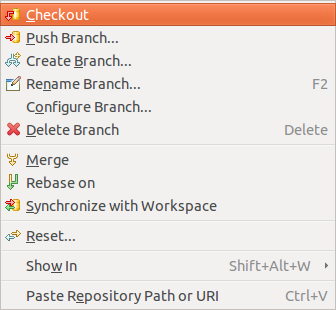


## Content in the Git Respositories view

The content area of the Git Repositories view shows the existing Git repositories and the important data of each repository. The following screenshot shows an example Git repository.

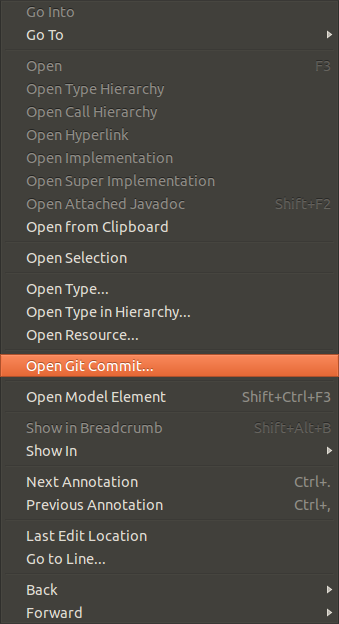


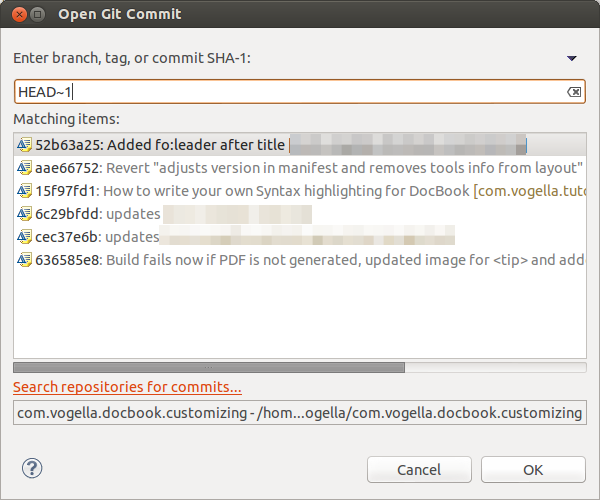
A right-click (context menu) on an element in the Git repositories view allows you to perform related Git operations. For example if you right-click on a branch you can checkout the branch or delete it.



## Open a commit

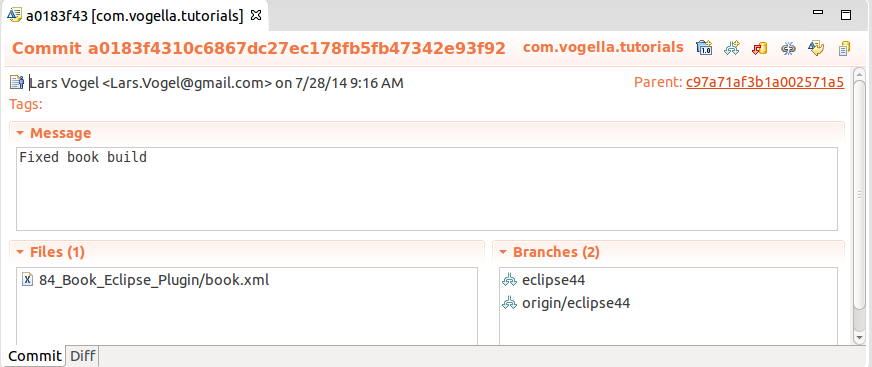
If you are in the Git repositories view you can open a commit via the main Eclipse menu. To do this select the Navigate → Open Git Commit menu entry.





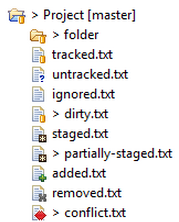
## Commands available in the Commit Viewer

If you open a commit you can create a tag or branch from it. You can also revert it, cherry pick it or check it out. You can also reveal it in the history view.



# Git integration into the Package and the Project Explorer

The Package Explorer view shows indicators on the files to show their status. The most important icon decorators are depicted in the following screenshot.



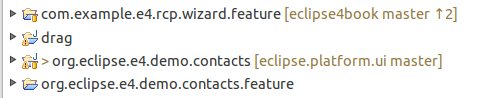
The file name describes the state of the file from the following table:

**Table 1. Git label decorations**

| **State** | **Description** |
| --- | --- |
| tracked | File is committed to the Git repository and has not changed. |
| untracked | File is neither staged nor committed. |
| ignored | File is flagged to be ignored by Git operations. |
| dirty | File has changed since the last commit. |
| staged | Changes in the file will be included in the next commit. |
| partially-staged | The resource has changes which are added to the index and additional unstaged changes in the working tree |
| added | Staged but not yet committed, i.e. snapshot of this file has been stored in the git database. This status is the same as the staged status, but the file wasn't under Git version control before. |
| removed | The resource is staged for removal from the Git repository. |
| conflict | A merge conflict exists for the file. |

A combination of the staged and dirty status means: some parts of the changed file have been staged while some are still unstaged. This can happen if you stage a file and then again modify the file before creating the next commit. You can also change the staged parts using the compare editor opened by double clicking files in the staging view.

On a project level the explorer view adds the information which Git repository is used to the project name. It also adds the number of commits that are different between local and remote tracking branch. This way you can quickly see if your local branch is ahead or behind the remote branch it is tracking.



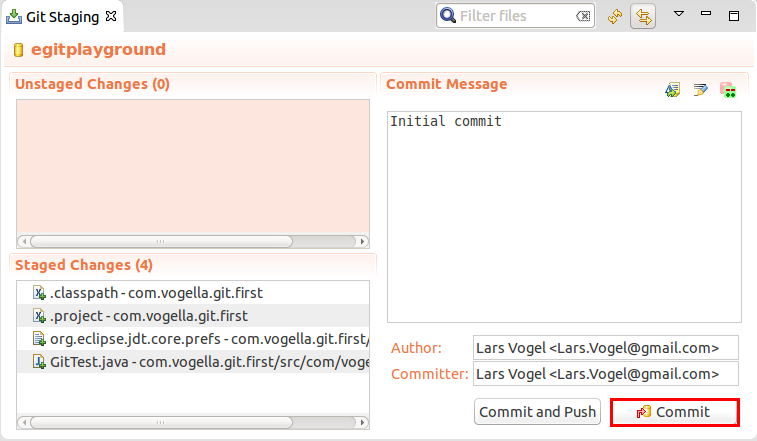
# Using the Git Staging view

Eclipse gives you several options to stage and commit your changes. The Git Staging view provides a convenient compact overview on all changes you have done compared to the current HEAD revision.

The Git Staging view is non-modal, you can switch between different repositories or even restart Eclipse without loosing a commit message and it allows incremental staging for changes.

Open the Git Staging view via the Window → Show View → Other... → Git → Git Staging menu.

In this view you select all files which have changed and drag them into the Staged Changes area. To commit the staged changes you write a descriptive commit message and press the Commit button which is highlighted in the following screenshot.

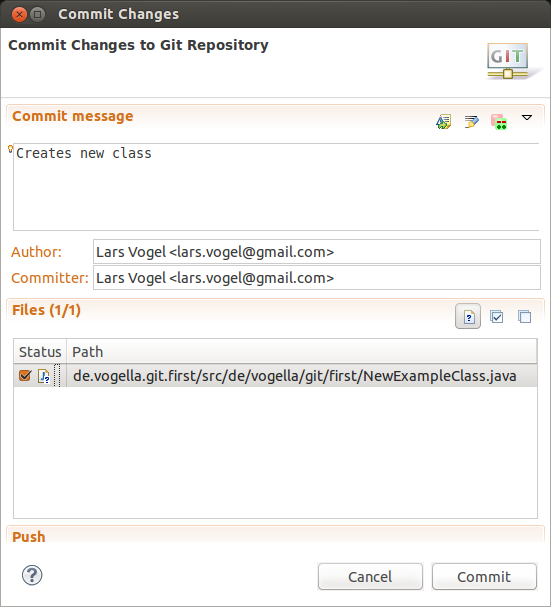


Perform these actions for your initial changes. Afterwards the first version of your Java project is under version control. If you don't experience any hardware error your data is now safely stored in your local Git repository and you can always restore your Eclipse project to this initial point.

# Using the Git commit dialog

You can also commit changes with the Git commit dialog. For this right-click on a resource and select Team → Commit from the context menu.

The dialog allows you to add changed and new files to the staging area and commit the changes.



Note

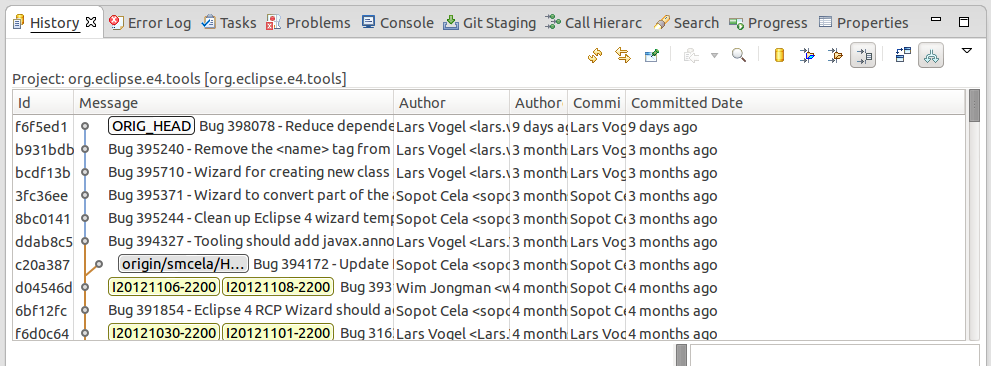
The commit dialog does not allow committing the removal of a file, since the commit dialog ignores the Git index.

# Using the History view

## Purpose of the history view

The History view allows you to analyze the history of your Git repository and to see to which commits the branches and tags points. This view displays author, date, commit message and the modified files of a commits.

This view is depicted in the following screenshot.



You can open this view via Window → Show View → Other... → Team → History. Alternative you can open the it via the repository node in the Git Repositories View. For this click on the Show In > History entry. Some views, e.g., in the Java EE-Perspective, do not have this shortcut, in this case use Team > Show in History).

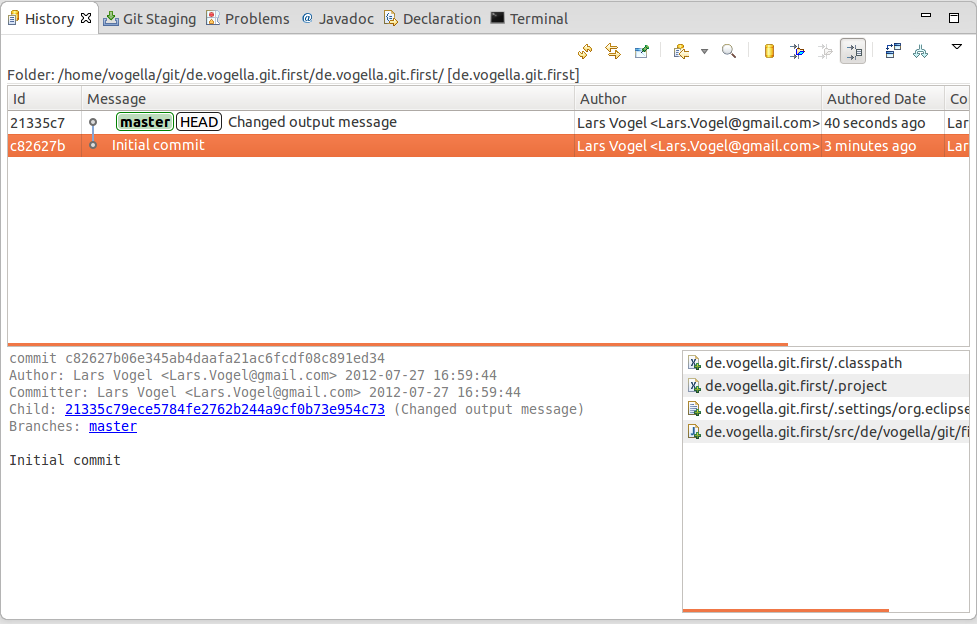
## Review the repository history via the History view

To see the history of a resource, select your project, a file or a folder, right-click on it and select the Show in → History context menu entry. Alternative you can use the Alt+Shift+W shortcut and select the History entry.

You can also configure the History view to display the history of the current selection. Select the highlighted button in the following screenshot for that.

Tracking active selection

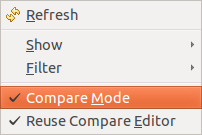
If you select a commit you see the commit message and the involved files.



Via right-click on an individual file you can compare this file with its ancestor (the commit before that) or with the current version in the workspace.

Tip

If the "compare mode" toggle is selected from the view menu of the History view you can also double click a file to compare it to the previous version.

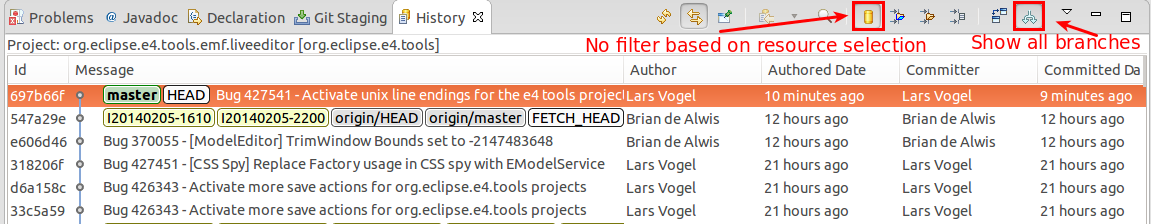


## The History view filters

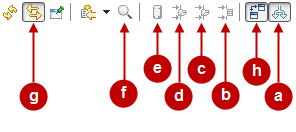
The History view has quite some options to configure which commits are displayed. Its toolbar allows you to customize which commits are displayed. By default, the History view filters the history based on the current selection and shows only the active branch.

If you work with several branches, e.g., because you are using Gerrit for code reviews, you typically want to see all branch information and remove the filter based on the resource.

The History view allows you to filter based on resources. See the tooltips of the toolbar for the meaning of the different filter options. In order to see all commits click the highlighted buttons with the Show all changes in this repository and Show all branches and tags tooltips.



The following listing gives an overview of the purpose of the different buttons.



Depending on your use case you may want to select the following option:

1. show only those commits which are reachable from the current branch. Hide all commits on other topic branches.
2. see only those commits which changed the selected resource (file, project, subfolder) or it’s children. E.g. display only those commits which touched the selected java file. The current selection is shown in the top right corner of the History view.
3. see only those commits which changed anything in the parent folder of the selected resource (file, project, subfolder) or it’s children. E.g. display only those commits which changed the same package as the selected java source.
4. see only those commits which changed anything in the same project as the selected resource or it’s children. Used when you are working in a repository which contains multiple projects.
5. don’t filter at all. Show all commits of the current repository

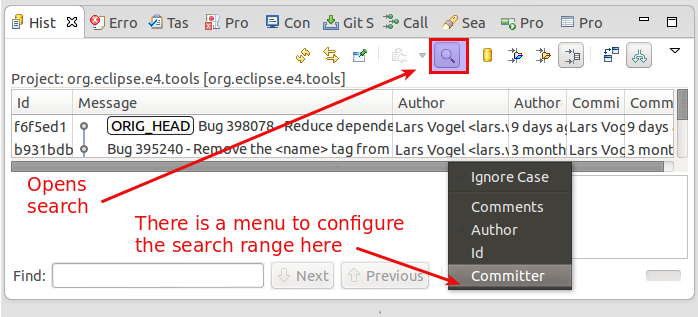
The options b., c. and d. are tied to the currently selected resource. Button g. allows that the history view automatically updates when you change the selection.

Tip

If you got lost with the different filters and the history doesn’t show what you expect, set it back to show everything. Therefore make sure that Show all branches and tags (a) is turned on and Show all changes in repository (e) is selected.

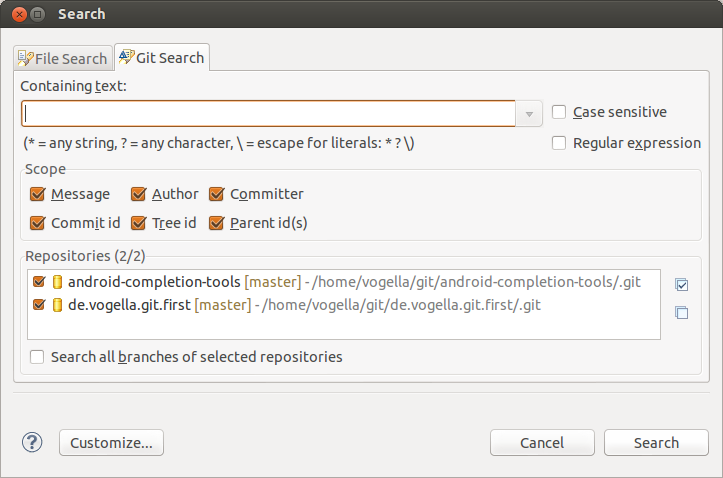
## Using search

You can also search for commits based on committer, author, ID or comment. For this turn on the Show Find toolbar (f) and type in a search string in the Find field. The commits fitting to your search are highlighted. You can combine this search with the filters explained above.



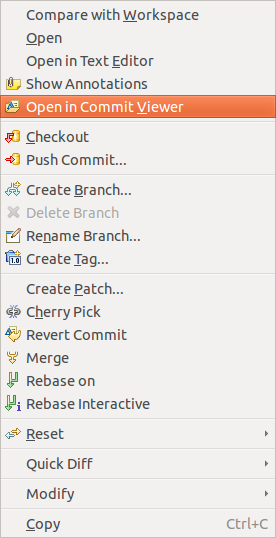
Note

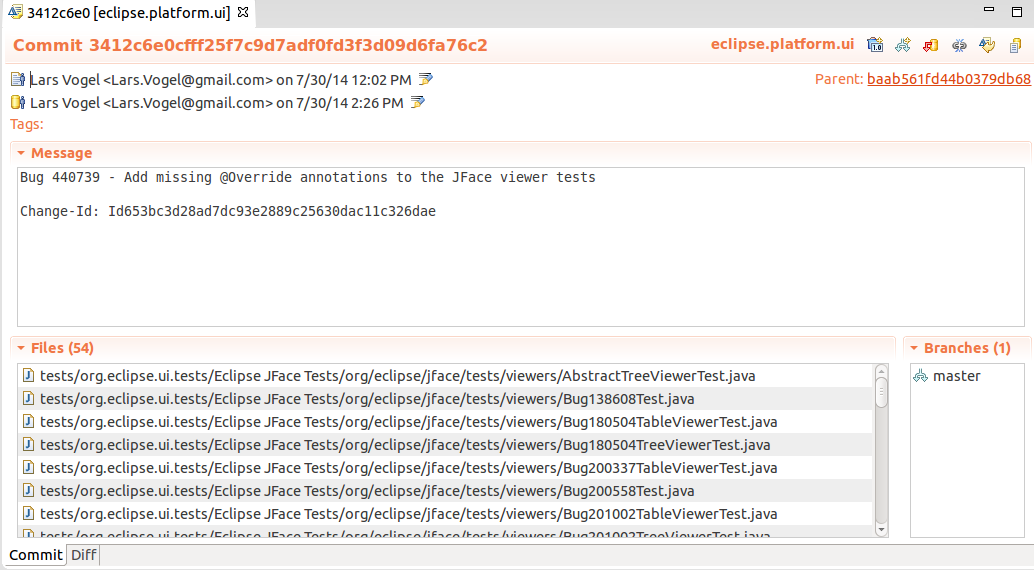
The Git Search available in the Search → Search menu is much more powerful and consumes less memory since it doesn't need to also display the history.



## Showing details of a commit

If you want to see more details about a commit, right-click it and select the Open in Commit Viewer entry.





# Working with one or multiple projects in a Git repository

## Workspace and projects

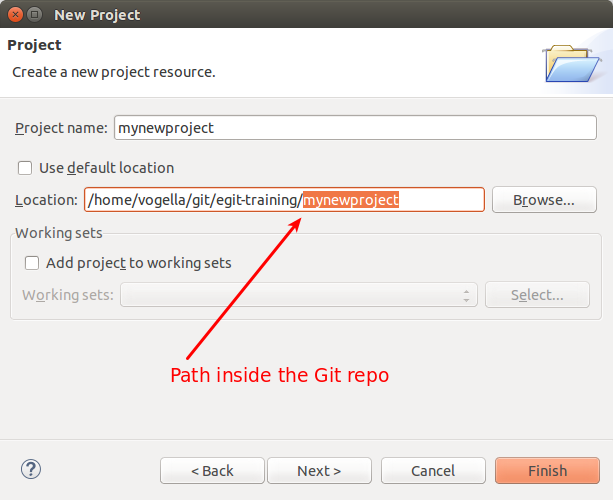
Eclipse allows working with projects that are not included in the root folder of the workspace.

Using this functionality your projects can be stored in the working tree of a Git repository.

## Adding a new project to a Git repository

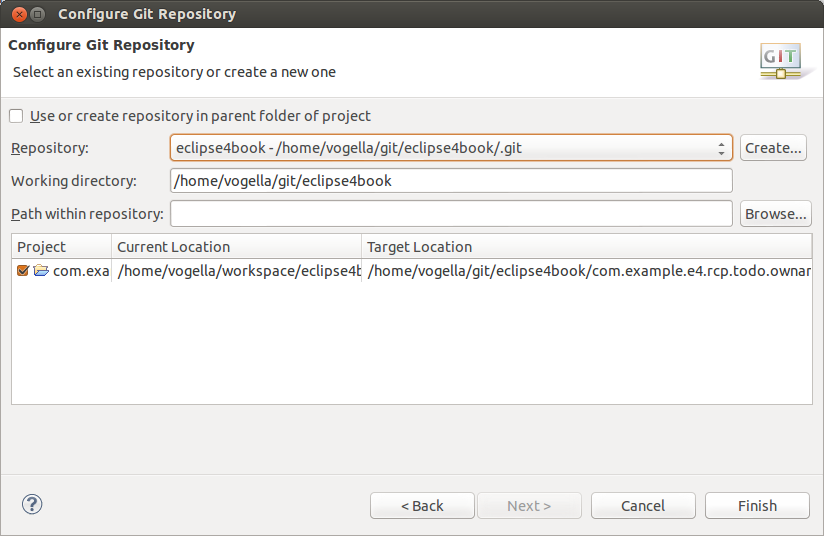
To can add Eclipse projects to an Git repository. If you do this the project is moved to the Git repository and linked to from the Eclipse workspace..

A simple way of adding a project to a Git repository is to specify the file location in the New Project wizard as depicted in the following screenshot.



If you add the Git repository to your Git repositories view, you can perform the Git team operations on the files of this repository.

To add a new Eclipse project to an existing Git repository, select the project, right-click on it and select Team → Share → Git and select the existing Git repository.



The Eclipse Git functionality moves the projects to the repository and imports the project automatically into your workspace.

# Exercise: Working with a Git repository in Eclipse

## Target: Using Eclipse Git for a local repository

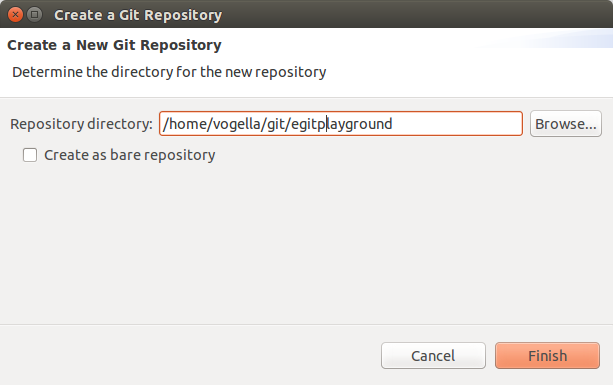
The following section explains how to create a local Git repository for one project with Eclipse. This allows you to keep track of your changes in the project and allows you to revert to another state at a later point in time.

## Create a new Git repository via Eclipse

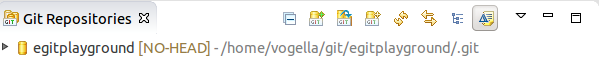
Open the Git Repositories view and select the Create a new Git repository and add it to this view option.



Select a new directory outside of your workspace. By convention this directory should be a subdirectory in the git folder of the users home directory.



If you press the Finish button this dialog creates the directory and adds a reference to the new Git repository to the Git Repositories view.



## Create .gitignore file

You want to configure Git to ignore the generated bin folder with the class files. Create for this purpose a .gitignore file in your Git repository with the following content. Unfortunately Eclipse Git does not allow to create a file directly in the repository. You have to do this step via the command line or via

bin

All files and directories which apply to the pattern described in this file will be ignored by Git. In this example all files in the bin are be ignored.

You can also create a local .gitignore file by right-clicking on a resource (file or folder) and by selecting the Team → Ignore context menu entry. Excluding individual files can be time consuming. Preferably you define a pattern in your .gitignore file in the root directory of the repository.

Note

You can also configure Eclipse to automatically ignore derived resources, e.g., class files via the Window → Preferences → Team → Git → Projects → Automatically ignore derived resources .. setting.

## Creating an Eclipse project

Create a new Java project called com.vogella.git.first in Eclipse. Create the com.vogella.git.first package and the following class.

package com.vogella.git.first;

public class GitTest {

public static void main(String[] args) {

System.out.println("Git is fun");

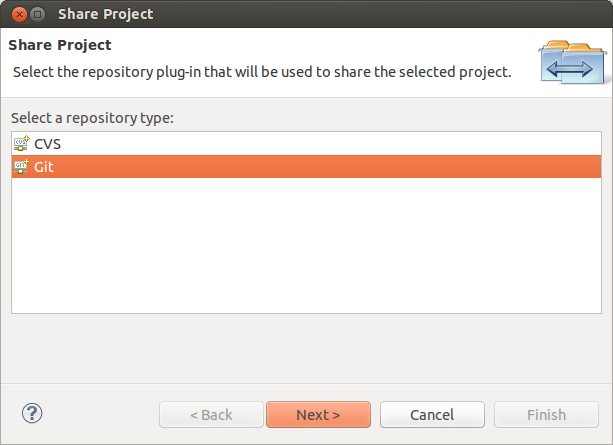
}

}

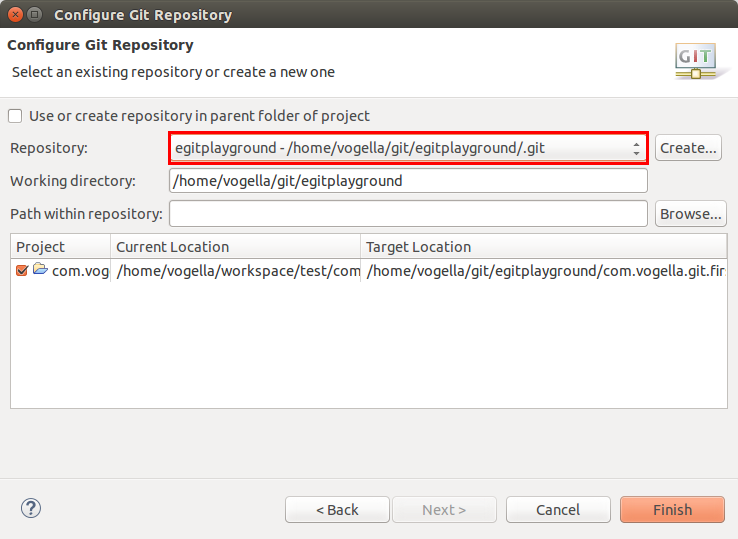
## Put project under version control

To put your new project under version control with Git, right-click on your project, select Team → Share Project → Git.

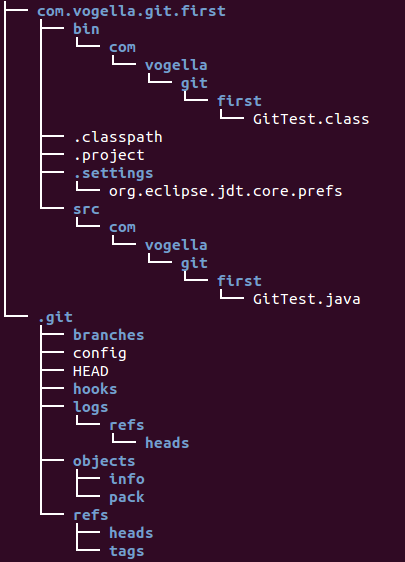
Depending on your installation you may have to select that you want to use Git as a version control system.



On the next dialog select your existing Git repository from the drop-down list and press the Finish button.

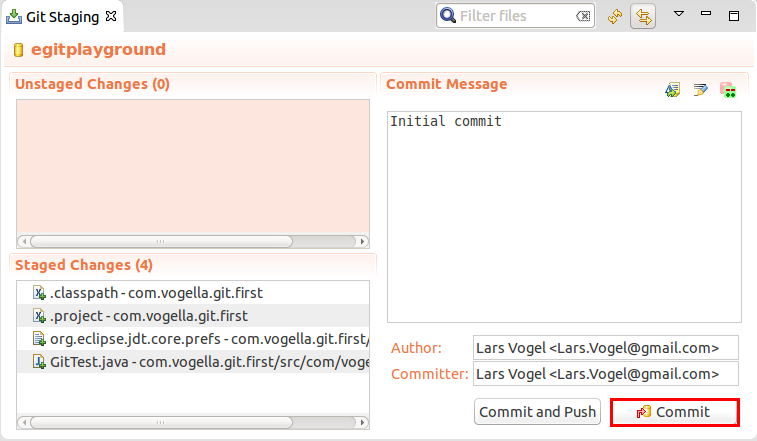


This moves the project to your Git repository. The Git repository is in this case directly stored in the specified folder in a .git folder. The following screenshot shows the generated directory structure. The .git directory contains the Git repository, the other directories contain the files of the working tree.



## Using the Git Staging view for the initial commit

Using the Git Staging view to drag all files into the Staged Changes area, write a meaningful commit message and press the commit button.



## Using the Git Staging view for committing changes

Change the System.out.println message in your GitTest class.

package com.vogella.git.first;

public class GitTest {

public static void main(String[] args) {

System.out.println("Git is cool");

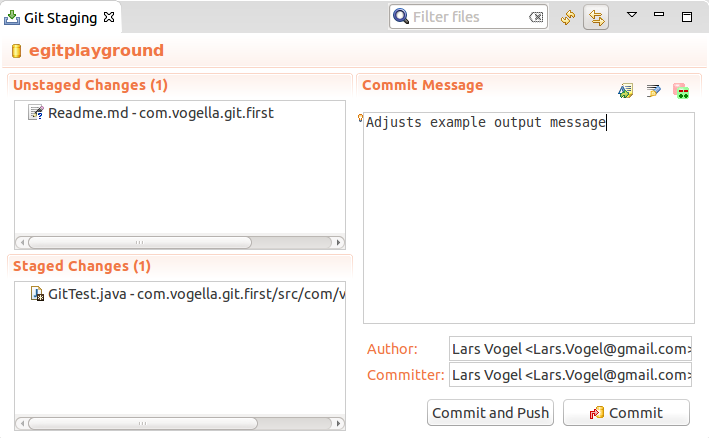
}

}

Also create a new file called Readme.md

In this example you should commit the changes of the GitTest class but not add and commit the Readme.md file to the Git repository.

Using the Git Staging view drag only the GitTest class into the Staged Changes area, write a meaningful commit message and press the commit button.



This change is now also stored in your local Git repository. The Readme.md file is neither staged nor committed to the Git repository.

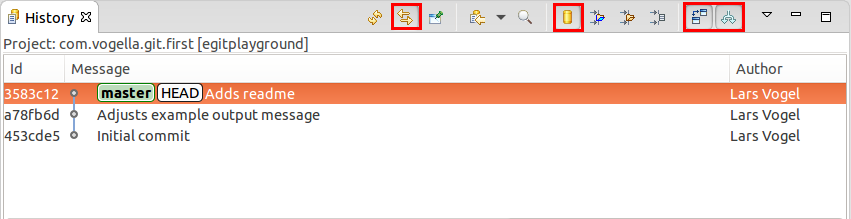
## Commit more files

Commit the Readme.md file. By now you should know that you have to stage the file and commit it.

## Review your commit history via the History view

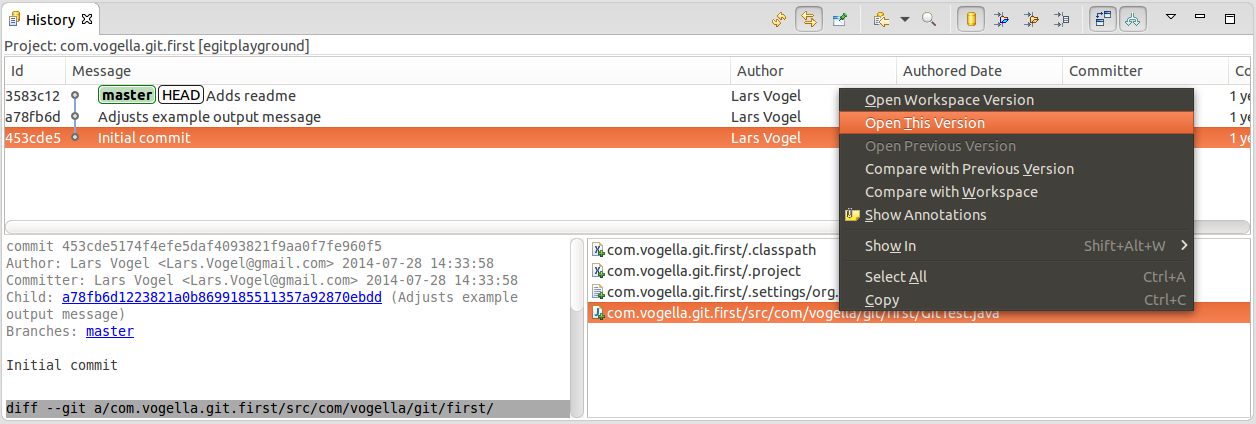
Open the History view to browse the commit history of your repository. See [???](http://www.vogella.com/tutorials/EclipseGit/article.html) for information how to use this view. Review which files were included in your individual commits.

In the History view click all toggle buttons as shown in the screenshot (Link with Editor and Selection, Show all changes in repository containing the selected resources, Compare Mode, Show all Branches and Tags)



## Open an older version of a file via the History view

Open the version for the first commit of GitTest via the History view.



# Exercise: Sharing multiple projects into one Git repository

## Create a new Git repository

Create a new local Git repository outside your workspace, called gitmultiexample. Use the description of [Section 15.2, “Create a new Git repository via Eclipse”](http://www.vogella.com/tutorials/EclipseGit/article.html#usingegit_create_repository) for this task.

## Create two Java projects

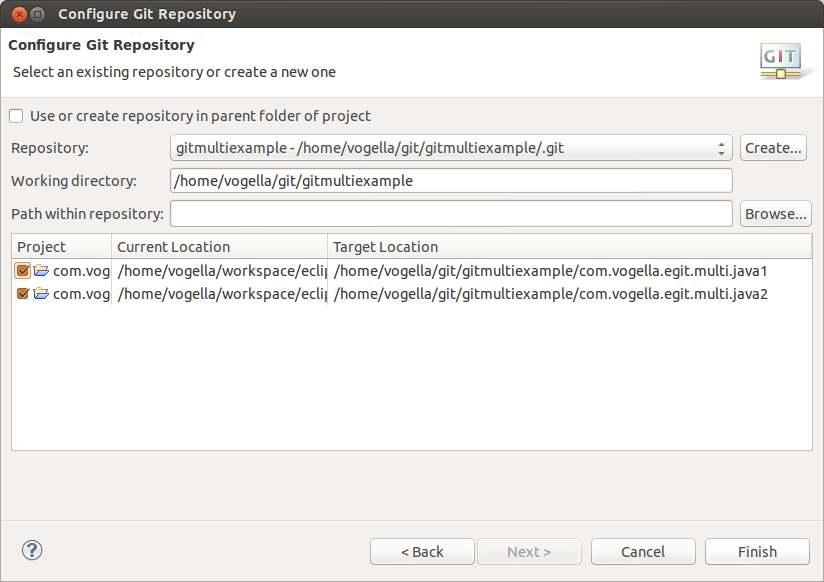
Create two Java projects called com.vogella.egit.multi.java1 and com.vogella.egit.multi.java2.

Create at least one Java class in each project.

## Share projects

Afterwards select both projects, right-click on them and select Team → Share Project... → Git. If asked by the Eclipse IDE, select that you want to use Git.

Select your new Git repository outside and include both projects into this repository.



Press the Finish button.

## Validate that projects have been moved

Check your workspace directory and your Git repository directory via a file explorer. You will see that the projects have been moved from there original location to the Git repository.

## Create your initial commit

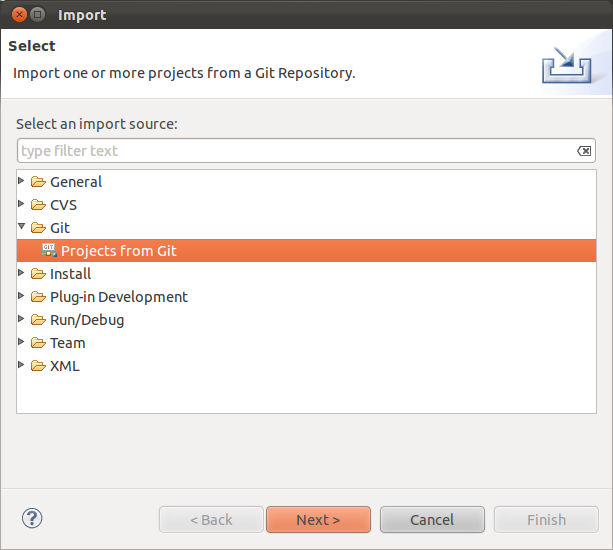
Now commit all files in the two projects to your Git repository.



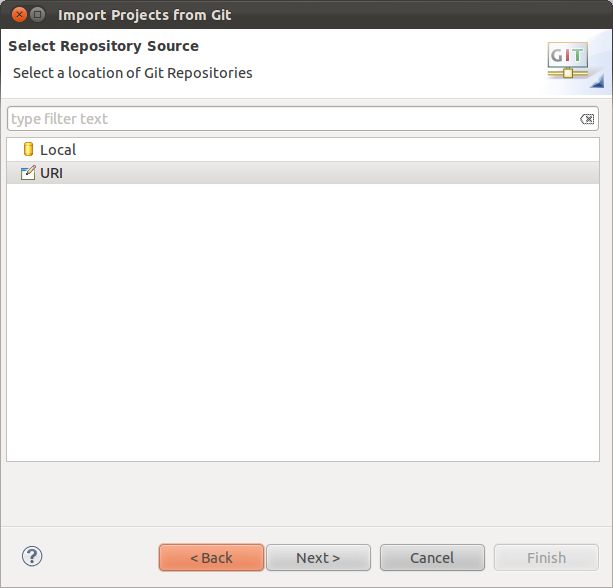
# Clone an existing repository

Eclipse allows you to clone an existing Git repository and to import existing projects from this repository into your Eclipse workspace by using a wizard.

Select File → Import → Git → Project from Git.



Select URI in the next dialog.



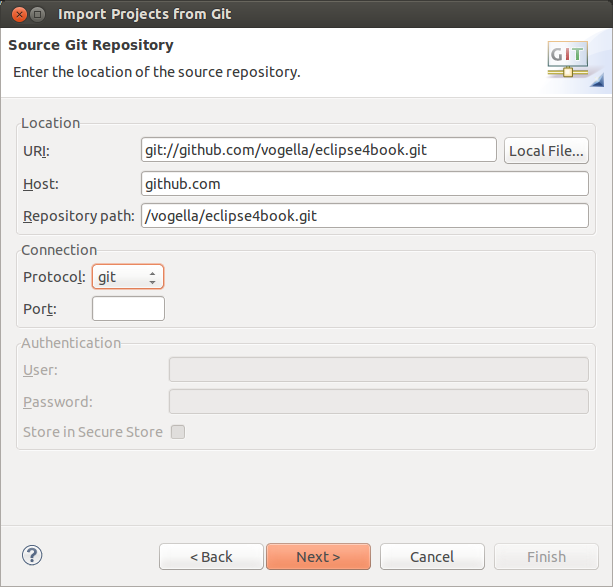
Enter the URL to your Git repository. Git supports several protocols, e.g. *git://*, *ssh://* and *https://*. You only have to paste the URL to the first line of the dialog, the rest will be filled out automatically.

Note

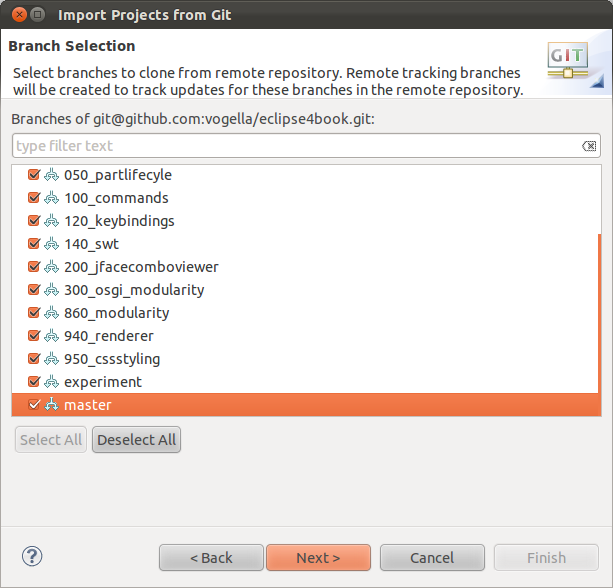
Some proxy servers block the *git://* and *ssh://* protocols. If you face issues, please try to use the *https://* or *http://* protocol.

For example the following URI can be used to clone the example projects of the Eclipse 4 application development book: git://github.com/vogella/eclipse4book.git

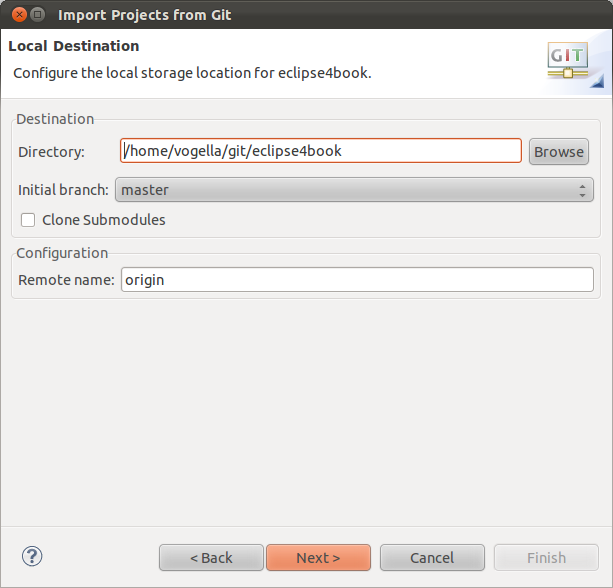
The above links uses the git protocol, alternatively you can also use the http protocol: http://github.com/vogella/eclipse4book.git



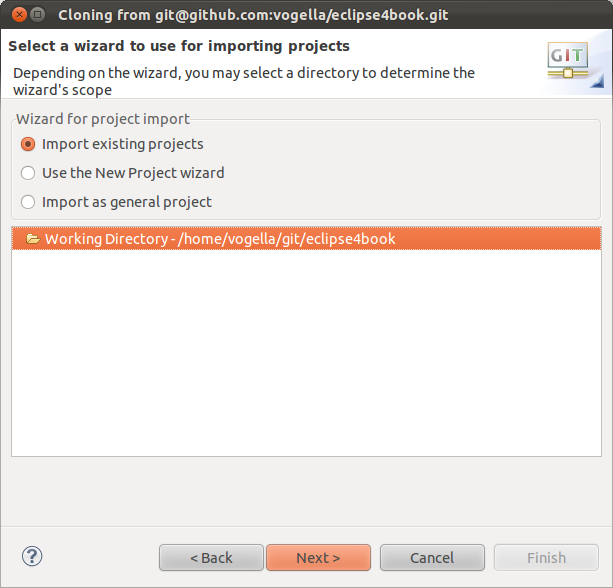
After pressing the Next button the system will allow you to import the existing branches. You should select at least master as this is typically the main development branch.



The next dialog allows you to specify where the repository should be copied to and which local branch should be created initially.



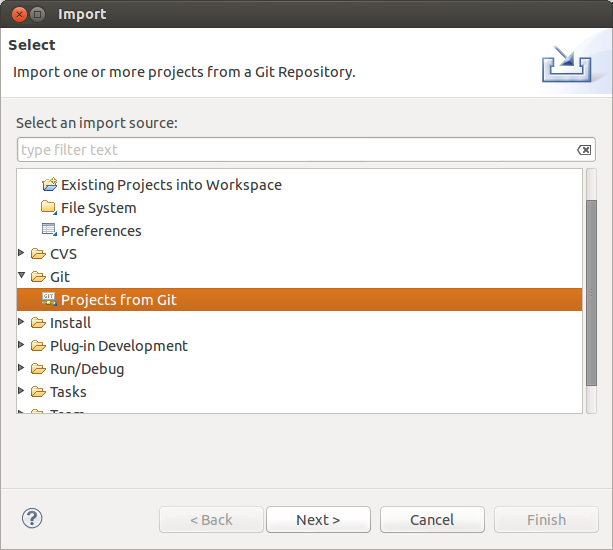
After the Git repository is cloned, EGit opens an additional import dialog which allows importing the Eclipse projects from the Git repository.



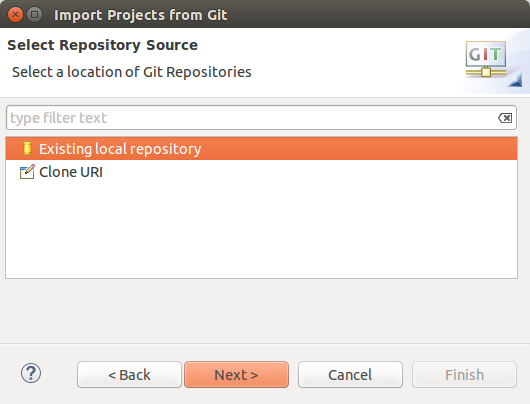
Once this dialog is completed, you have checked out (cloned) the projects into a local Git repository and you can use Git operation on these projects.

# Import projects from an existing repository

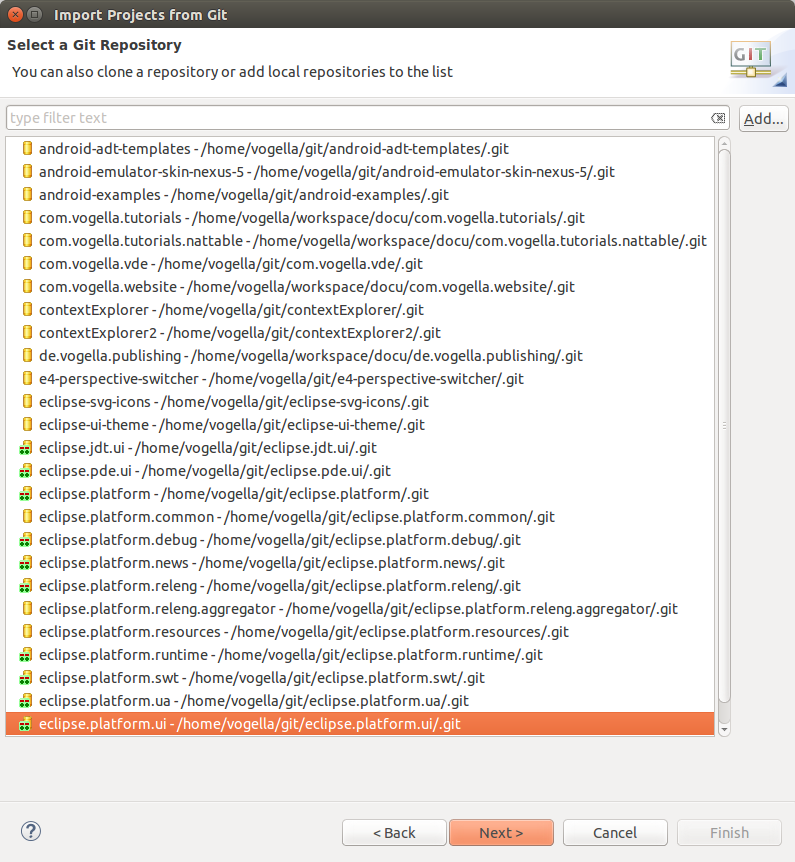
If you have already an existing Git repository you can add it to Eclipse and import the Eclipse projects into your workspace via the Select File → Import → Git → Project from Git menu entry.



Select Local if you want to important from a local repository or Clone URL if you first want to clone the repository.



The following screenshot shows several local repositories. To import the project contained in one of them, select one entries and press the Next button. To add a new local repository to this dialog (and the Git repositories view) use the Add... button.

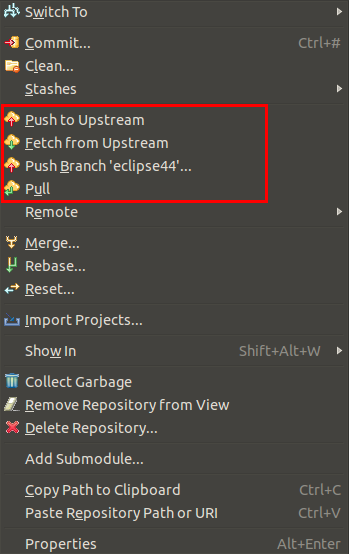


The wizard allows you to import existing projects. After this import the Eclipse IDE makes the projects available and is aware that these projects are part of a Git repository.

# Performing Git operations in Eclipse

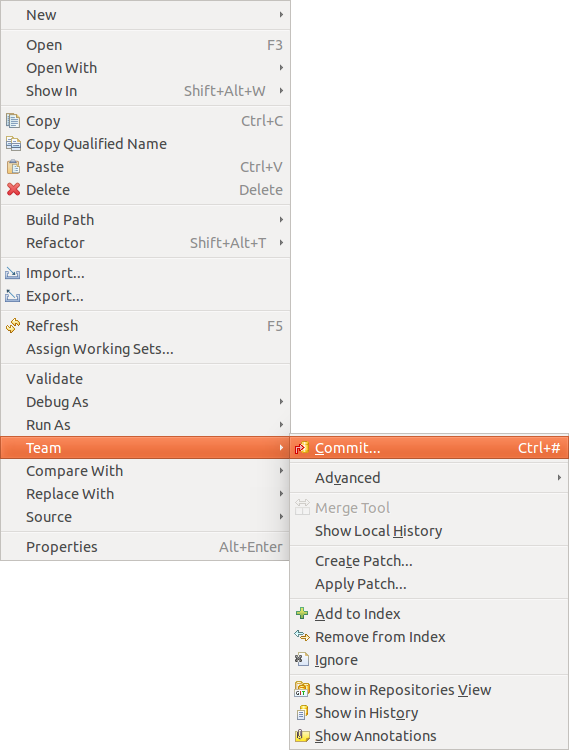
## Pull, push and fetch

You can use the Git Repositories view to pull, push and fetch to remote repositories. Right click on your repository and select the appropriated operation.



## Basic team operations

Once you have placed a project under version control you can start using team operations on your project. The team operations are available via right-click on your project or file.



Tip

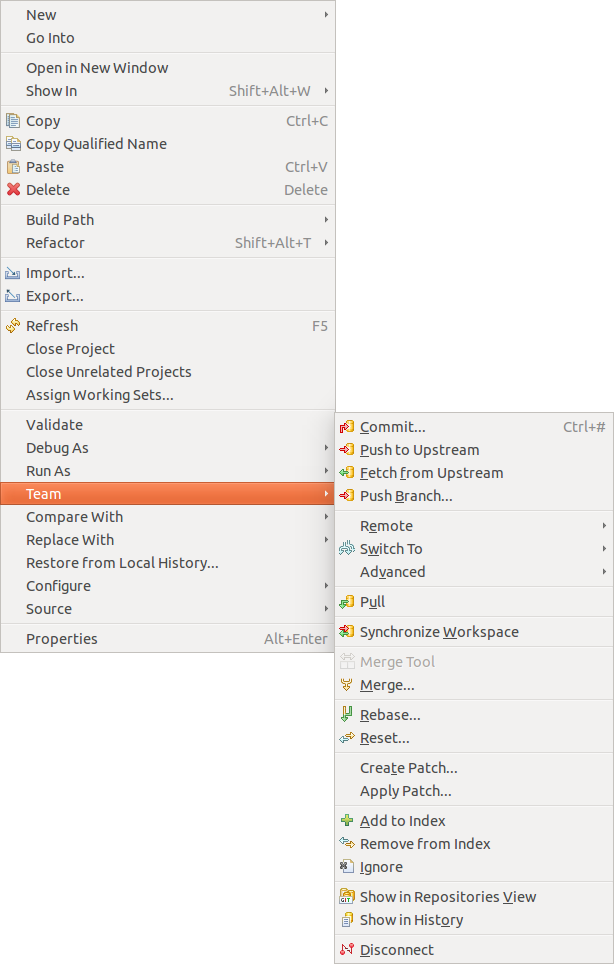
The Team menu is also available from the context menu of an opened editor.

The most important operations are described in the following list. Select:

* Team → Add to index, to add the selected resource(s) to the Git index
* Team → Commit, to open the commit dialog to create a new commit
* Team → Create Patch..., to create a patch
* Team → Apply Patch..., to apply a patch to your file system
* Team → Ignore, to add a file to the .gitignore file
* Team → Show in History, to display the history of the selected resources(s)

## Team operations available on the project

If you select a project you can use additional team operations from the context menu.

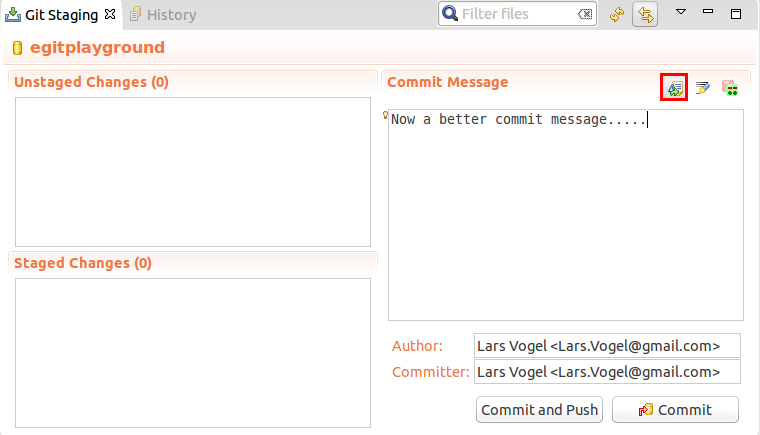


* Team → Pull to pull in changes from your remote Git repository
* Team → Fetch to fetch the current state from the remote repository
* Team → Switch To to checkout existing or create new branches
* Team → Push to push changes to your remote Git repository
* Team → Tag to create and manage tags.

## Amending a commit

Git amend allows adjusting the last commit. For example you can change the commit message or add another modification.

The Git Staging view allows you to perform the Git amend command via the highlighted button in the following screenshot.



# Branching in Eclipse

Right-click your project and select Team → Branch to create new branches or to switch between existing branches.

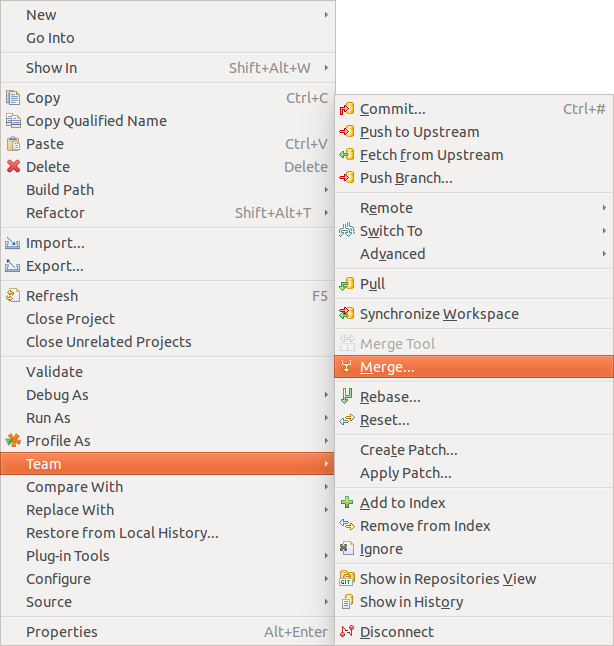
You can also switch branches in the History view or the Git repositories view.

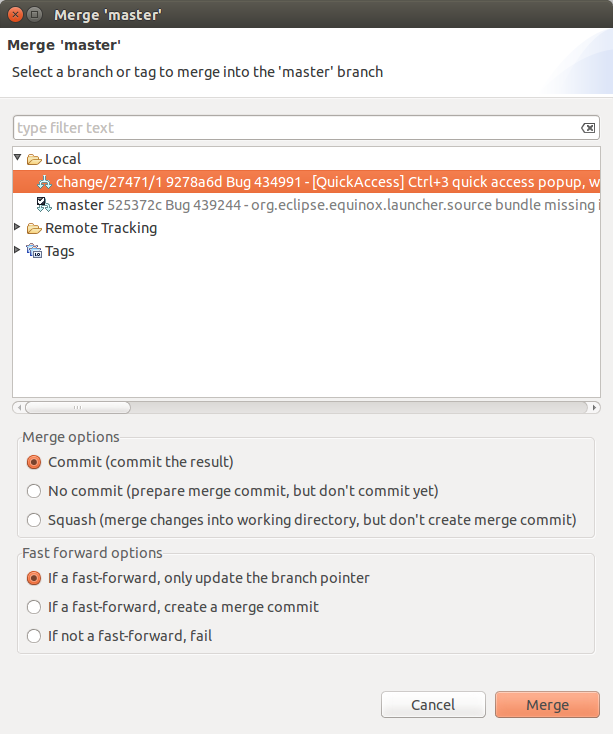
# Starting a merge operation in Eclipse

## Merge

Eclipse supports merging of branches to add the changes committed on one branch into another branch.

Checkout the branch into which you want to merge the changes into and select your project and Team → Merge to start the merge dialog.





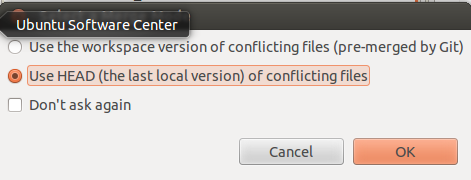
## Solving merge conflicts

If during a Git operation, two changes are conflicting, you have to solve these conflicts manually. Eclipse Git highlights the affected files in the Package Explorer or Project Explorer view.

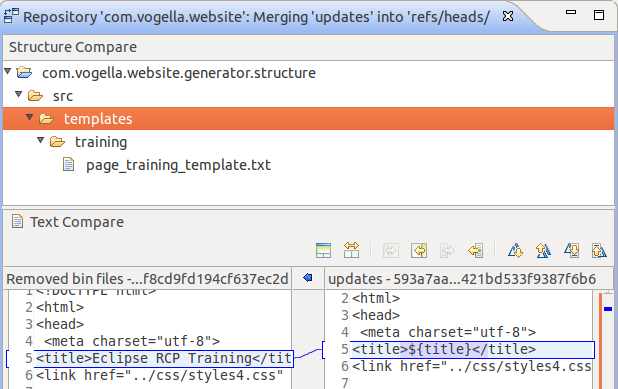
Eclipse Git supports the resolution of these merge conflicts. To trigger this via the explorer views, right-click on a file with merge conflicts and select Team → Merge Tool.

You can also use the Git staging view to find the conflicting files. In large projects this is usually faster than navigating the Package Explorer or Project Explorer view.

This opens a dialog, asking you which merge mode you would like to use. The easiest way to see the conflicting changes is to use the Use HEAD (the last local version) of conflicting files as merge mode. This way you see the original changes on the left side and the conflicting and non-conflicting changes on the right side.



You can manually edit the text on the left side or use the Copy current change from right to left button to copy the changes from right to left.

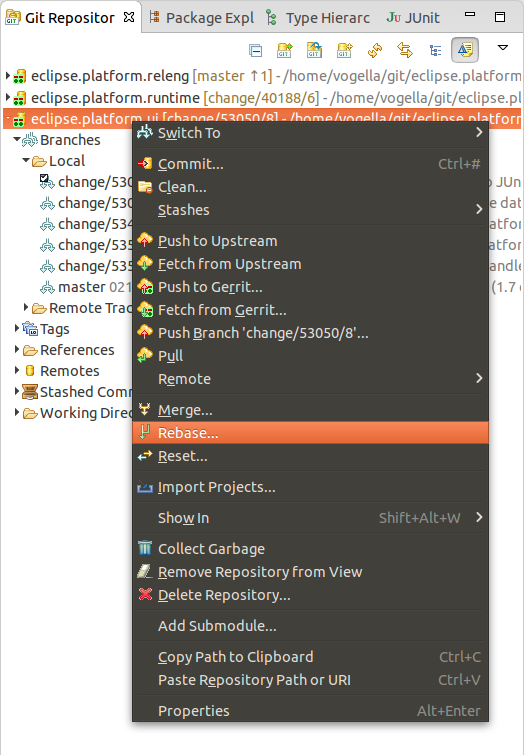


Copy changes from right to left

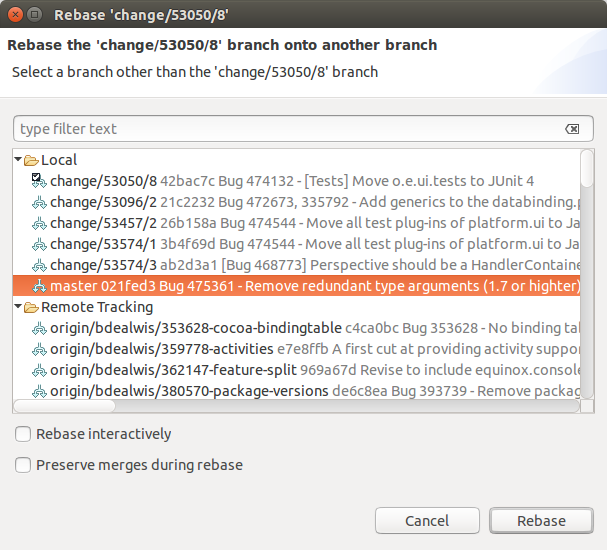
Once you have manually merged the changes, select Team → Add from the context menu of the resource to mark the conflicts as resolved and commit the merge commit via Team → Commit.

# Rebasing a branch onto another branch via Eclipse

The Git Repositories view allows you to rebase your currently checkout branch onto another branch. Right-click on a repository node and select Rebase as depicted in the following screenshot.



In the following dialog you can select the branch onto which you want to rebase.



Tip

You can also select the branch to rebase onto from the Branches node of the tree directly.

If the rebase was successful a dialog is shown. You have to resolve rebase conflicts if they occur. After resolving them, select Rebase → Continue.

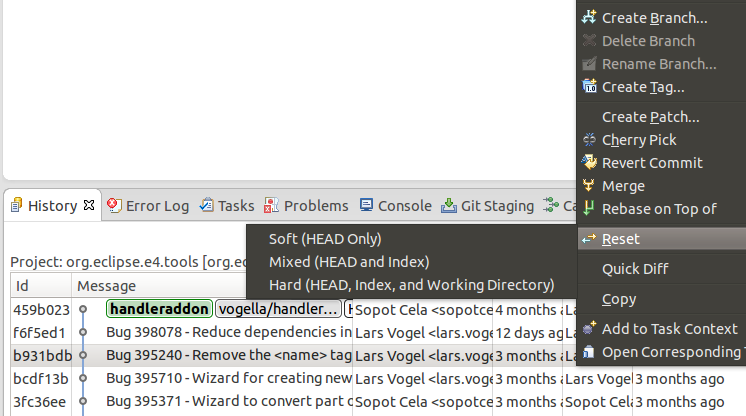
If you want to skip the conflicting commit and continue with the rebase operation use Rebase → Skip.

To cancel the rebase operation select Rebase → Abort.

# Git reset and Git reflog

## git reset

The History view allows you to reset your current branch to a commit. Right-click on a certain commit and select Reset and the reset mode you would like to use.



## Finding "invisible" commits with the Reflog view

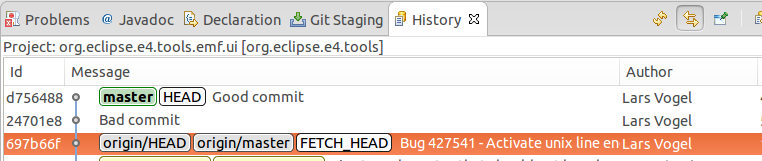
Commits are not visible in the Git history if they can't be reached from a branch or tag. This might happen during a reset, commit amend or rebase operation. By default, such invisible commits are removed after two weeks by the Git system.

The Git Reflog view keeps track of the movements of the HEAD pointer and the movements of each branch. This view allows you to find a commit again, e.g., if you used the git reset --hard command to remove certain commits.

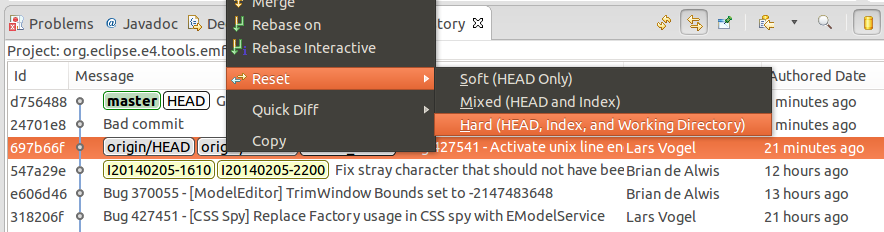
# Using git cherry-pick

In the History view, you can cherry-pick a commit via the context menu.

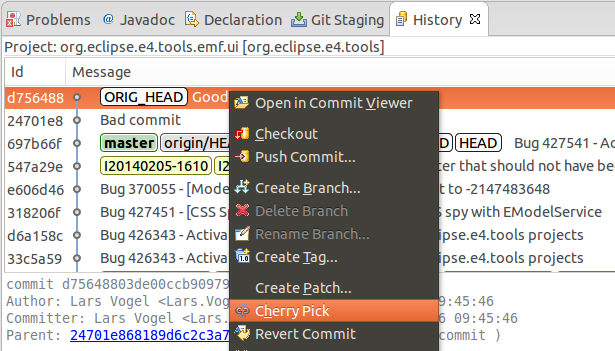
A combination of git reset and git cherry-pick allows you to move the changes done in a commit to anther branch. Assume you have a bad commit which you would like to remove from the history of branch followed by a good commit. This situation is depicted in the following screenshot.

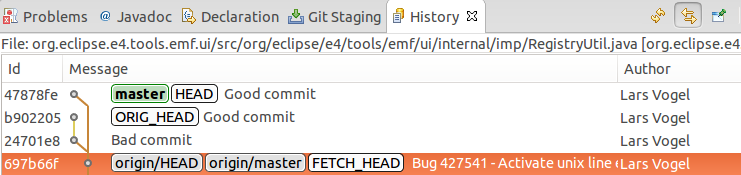


For this you would make a hard reset on the commit of origin/master.



Afterwards you can cherry-pick the good commit.





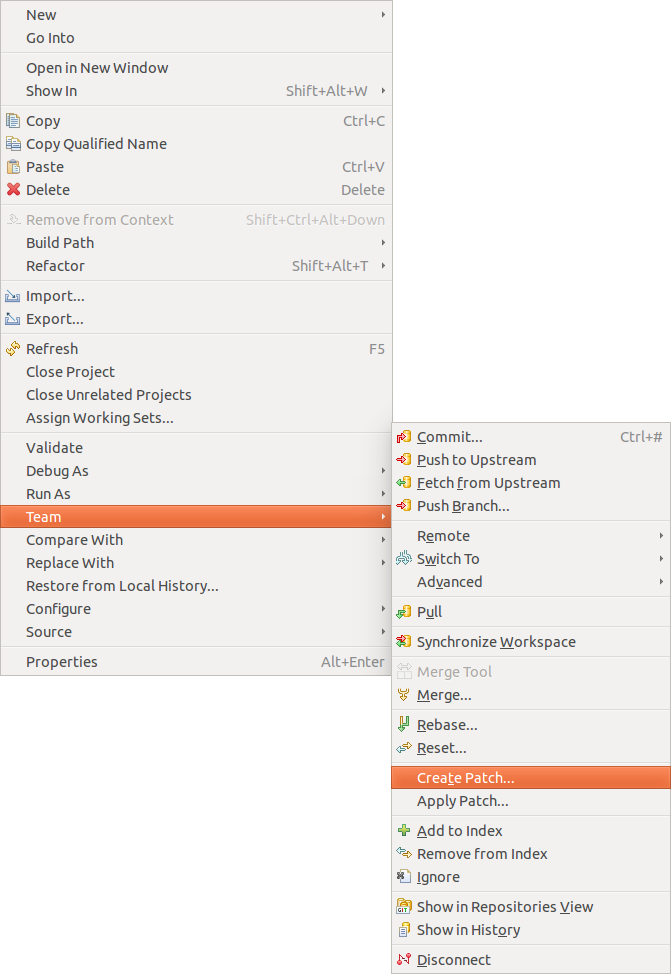
This results in a history without the bad commit.

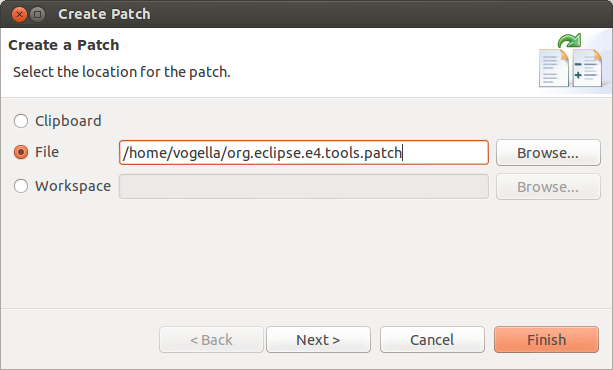
Note

You can do the same with interactive rebase. See [???](http://www.vogella.com/tutorials/EclipseGit/article.html).

# Creating patches

To create a patch for a set of changes with Eclipse Git, select the resources for which you want to create a patch in the Package Explorer view, right click and select Team → Create Patch.



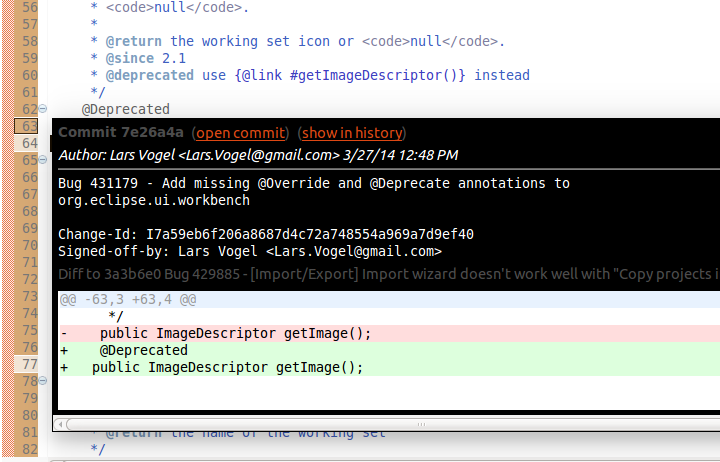


The resulting file can be used to get applied to another Git repository, via Team → Apply Patch. You can also apply the patch on a system where Git isn't installed at all, i.e., you don't need a Git repository to apply a patch.

# Blame annotations

The Git tooling in Eclipse allows displaying annotations to see which line was last changed by whom and which commit. To enable this, right-click on your file and select Team → Show Annotations.

Afterwards you can place the mouse on the left side of the editor and a popup dialog shows the commit information and the change applied by the shown commit.

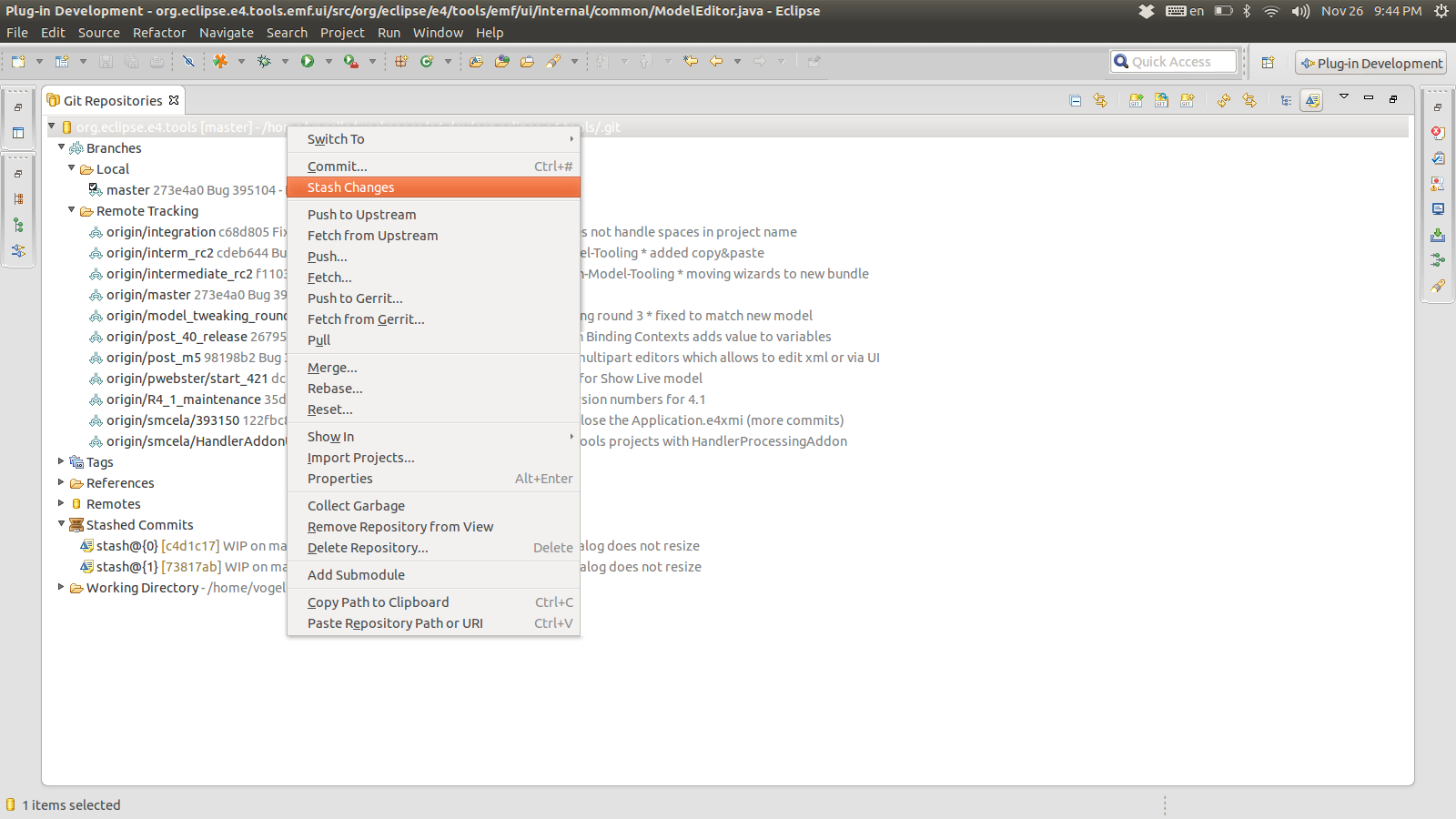


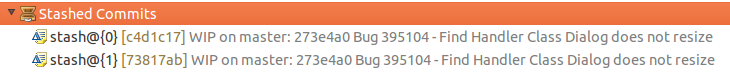
Tip

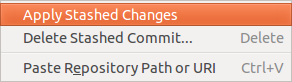
To ignore whitespace changes in the Git blame annotations in Eclipse, select Window → Preferences → Team → Git and select Ignore whitespace changes.

# Stash via the Git repository view

The git stash command is available in the Git repositories view. Right-click on your Git repository and select Stash Changes.







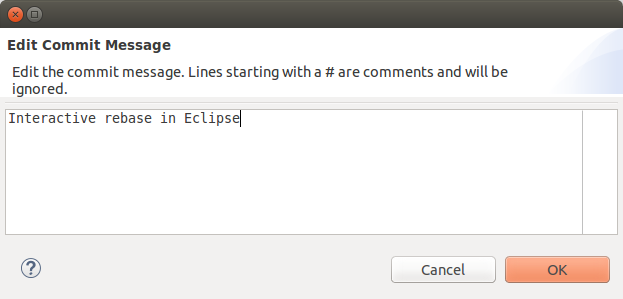
# Adjusting the history with interactive rebase

## Support for interactive rebase in Eclipse

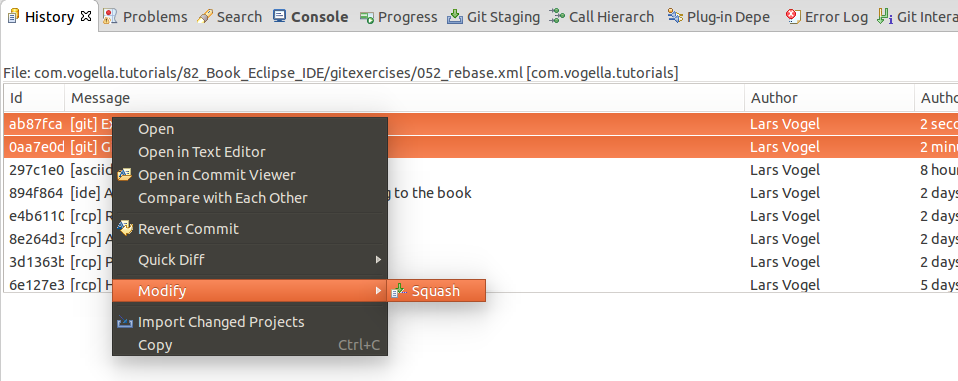
Git allows to adjust the existing commit history the interactive rebase functionality. Eclipse supports this functionality rebase via the History view and the Git Interactive Rebase view.

## Actions available via the History view

To reword a commit, right-click on it in the History view and select Modify → Reword to change the commit message.



You can squash several commits by selecting them in the History view and by using the Modify → Squash menu entry from the context menu.

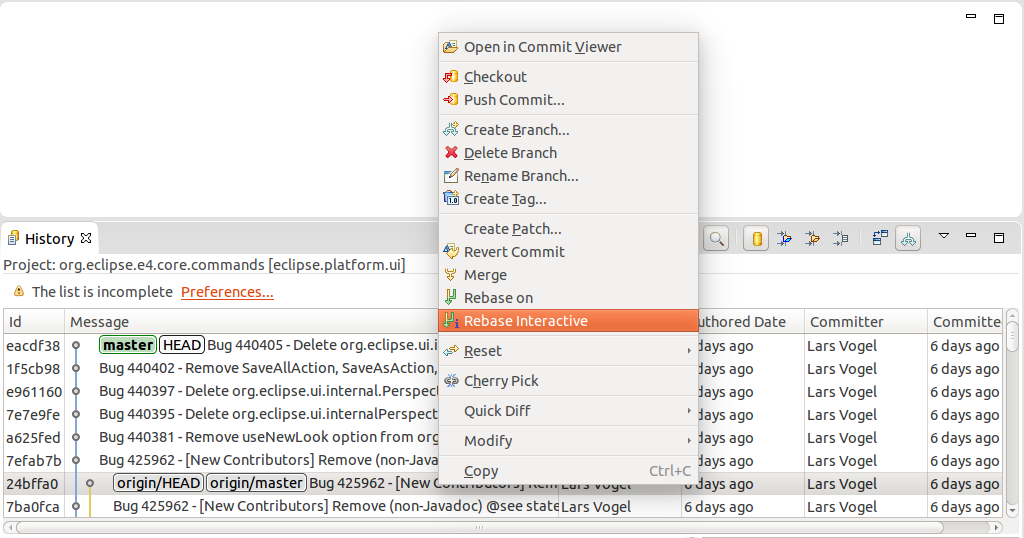


The above options are simplified ways to do an interactive rebase.

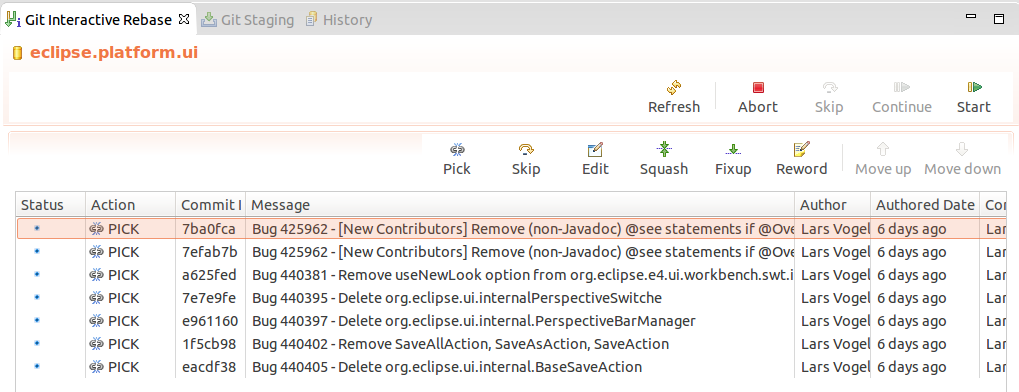
## 28.3. Using the Git Interactive Rebase view

The Git Interactive Rebase view allow you to change the order of commits, combine or remove commits, adjust commits and change the commit message.

To start the full interactive rebase open the History view and click Rebase Interactive on the context menu of the commit preceding the oldest commit you want to rewrite. Often this is the one origin/master points to.



This opens the Git Interactive Rebase view. It shows the rebase plan populated with the commits to be modified in topological order in the sequence they will be processed. This order is the reverse order which you see via the git log command or in the History view. The initial action for all commits is "Pick".

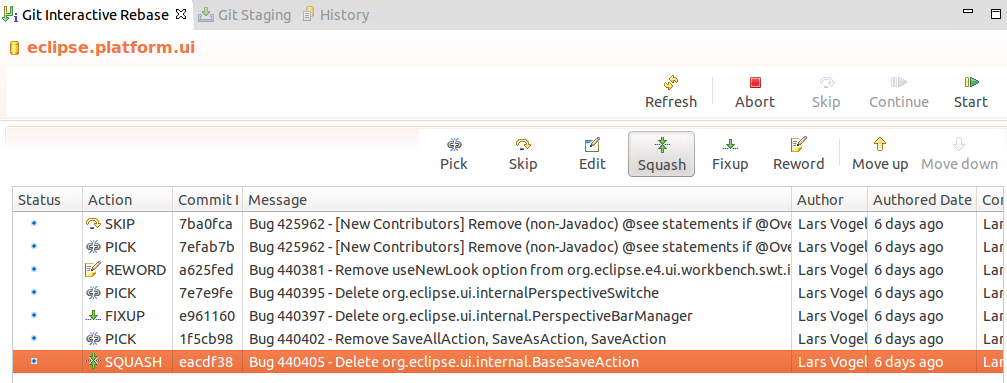


The Eclipse Git tooling supports the following actions.

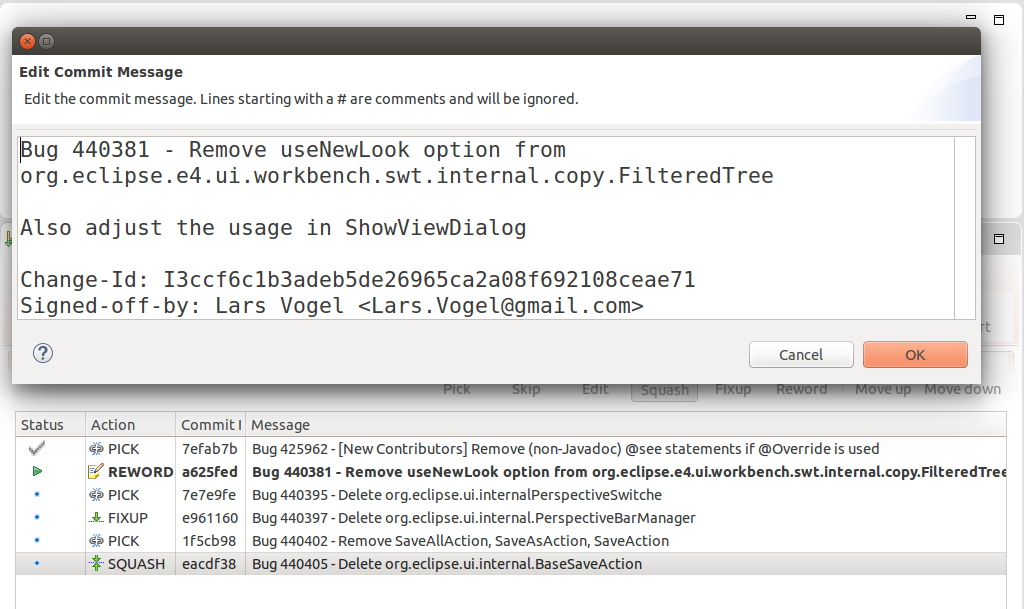
**Table 2. Interactive rebase actions**

| **Action** | **Description** |
| --- | --- |
| pick | includes the selected commit, moving pick entries enables reordering of commits |
| skip | removes a commit |
| edit | amends the commit |
| squash | combines the changes of the commit with the previous commit and combines their commit messages |
| fixup | squashes the changes of a commit into the previous commit discarding the squashed commit's message |
| reword | similar to pick but allows modifying the commit message |

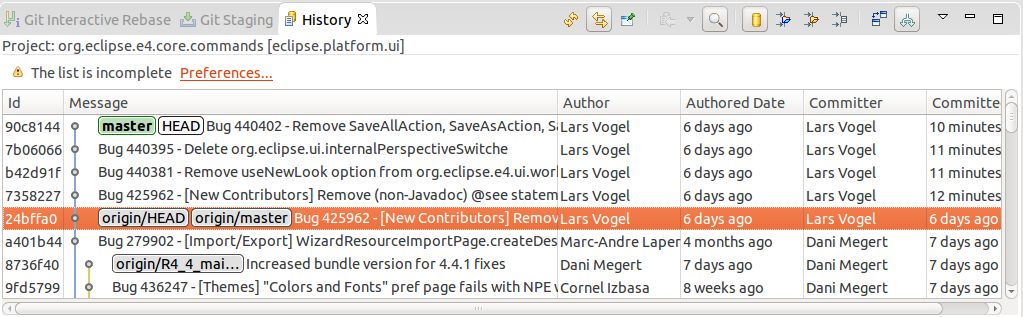
Use this view to finalize the rebase plan. For example you can reorder commits with the arrow buttons and select the rebase action you want to apply to the commit. The following screenshot demonstrates a possible selection.



When the rebase plan is finalized, click the Start button to start the interactive rebase command. Eclipse Git processes the plan. It stops at all commits with an action which needs user feedback, for example the reword action which requires entering the new commit message. The dialog for changing the commit message is depicted in the following screenshot.



Here is the result of the rebase operation displayed in the History view.



Tip

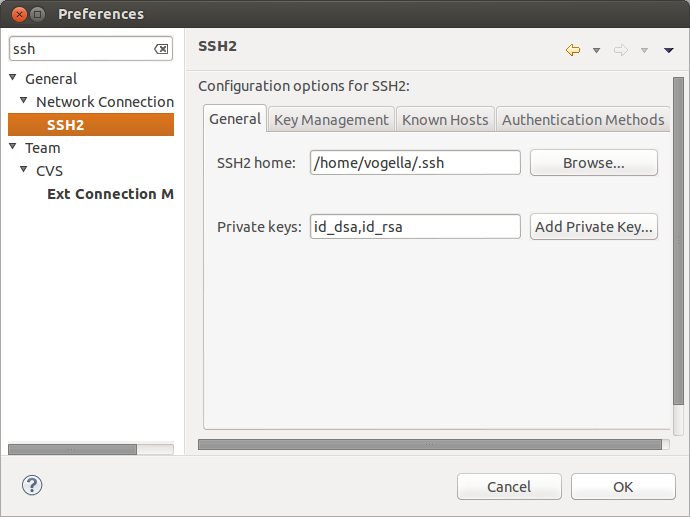
If something goes wrong during the rebase operation you can select Abort in order to stop the rebase operation and roll back to the starting point.

# 29. Using Eclipse Git with GitHub

## Clone project

Copy the URL from GitHub and select in Eclipse from the menu the File → Import → Git → Projects from Git

Eclipse fills out most of the fields based on the URL in the clipboard. Enter your user and password to be able to push to GitHub. Alternative you can also use an SSH key. You can configure Eclipse to know your SSH via the Window → Preferences → General → Network Connection → SSH2 preference setting. This setting is depicted in the following screenshot.



## Push changes

After you made changes and committed them to your local repository, you can select Team → Push to upstream on the project folder, to push your changes to your GitHub. This requires write access to the GitHub repository.

# Eclipse support for SSH based authentication

The Eclipse IDE allows you to create an SSH key pair for SSH based communication via Window → Preferences → General → Network Connection → SSH2.

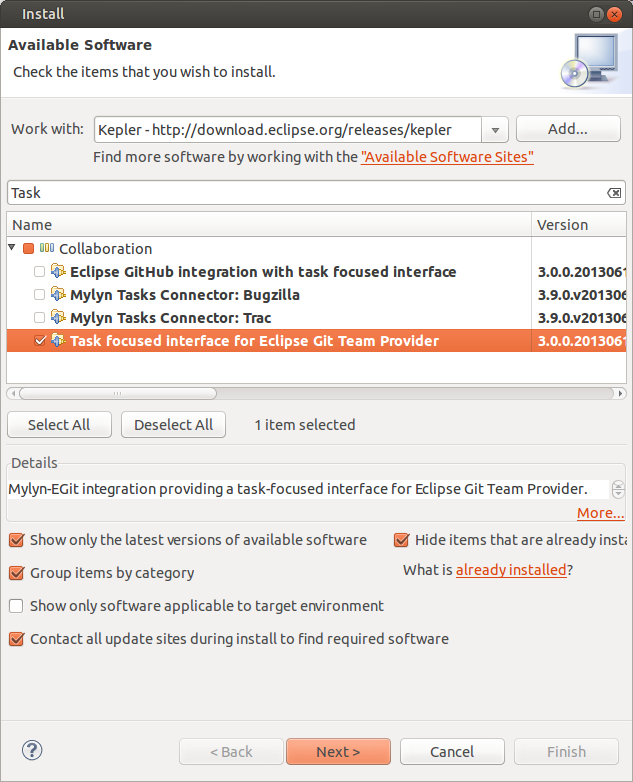
# Eclipse integration with GitHub

## The Eclipse Mylyn plug-in

Eclipse Mylyn provides task integration for GitHub issues, GitHub pull and Gist (short text snippets) into the Eclipse IDE.

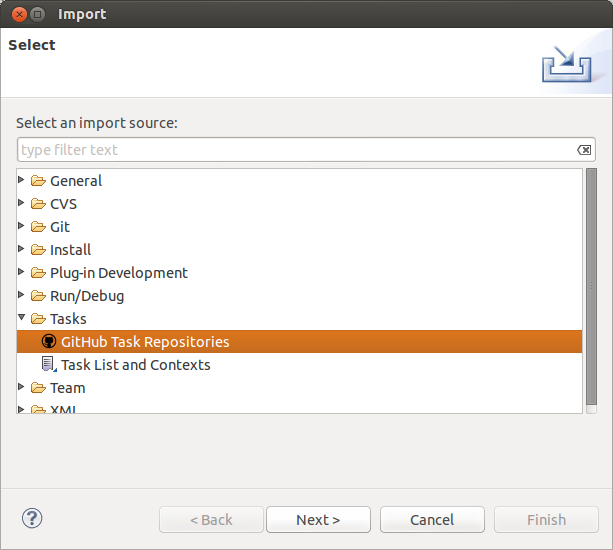
There is a GitHub connector for Mylyn available, please see [GitHub Mylyn User Guide](http://wiki.eclipse.org/EGit/GitHub/UserGuide) for details.

You install it via Help → Install new Software and the update site of your release.



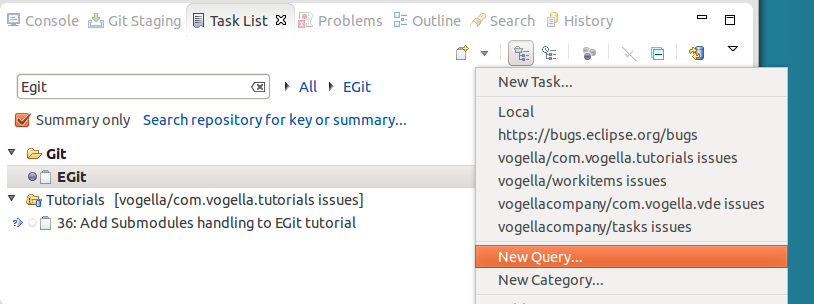
## GitHub issue integration

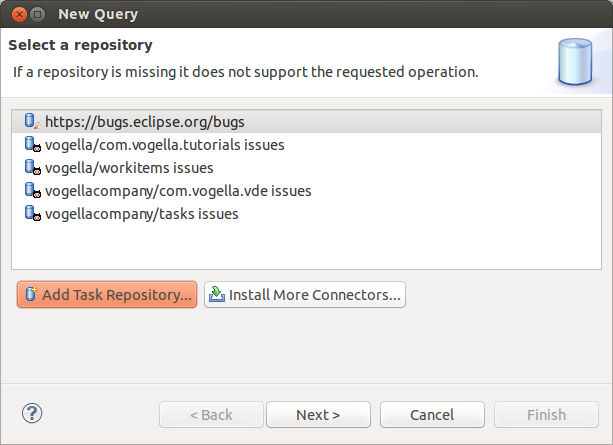
You can integrate your GitHub issues into Eclipse via File → Import... → Task → GitHub Task Repositories and by following the wizard.

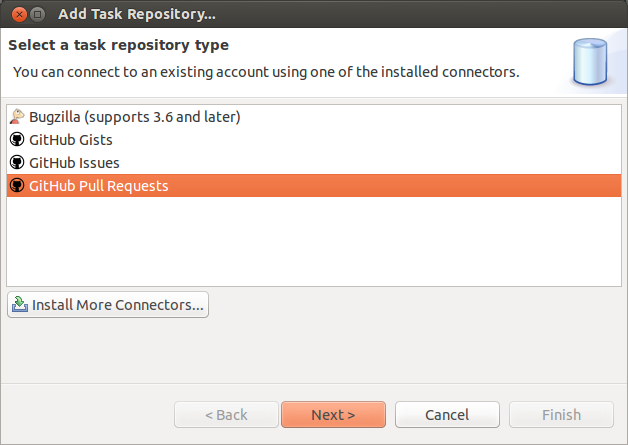


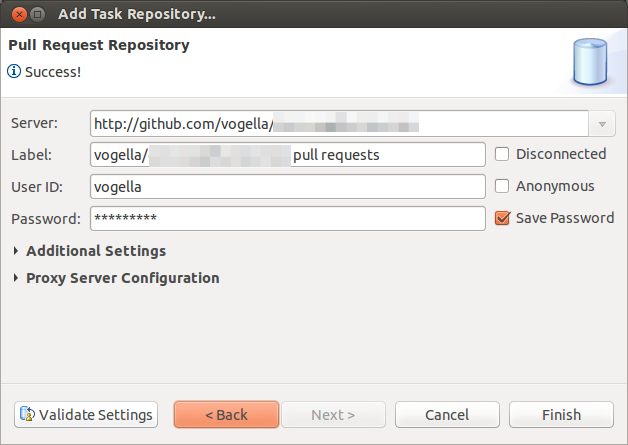
## Manage pull requests in Eclipse

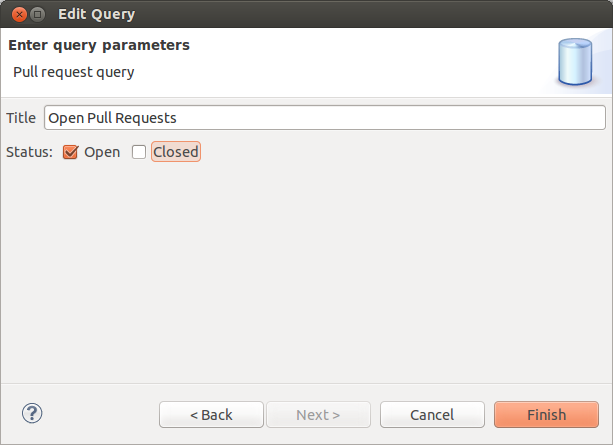
You can integrate your pull requests at GitHub into Eclipse by creating a new query from the Task List view. This is demonstrated via the following screenshots.









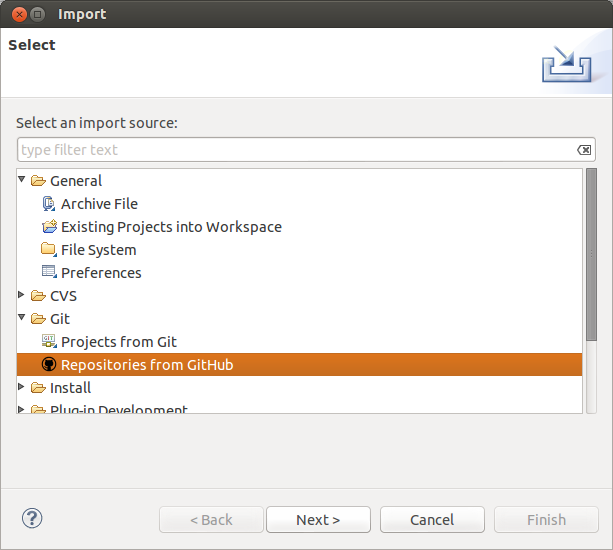


Note

Unfortunately the GitHub connect does currently not support that you merge the pull request.

## Import projects directly from GitHub

You can also import now directly projects from GitHub repositories.



## More infos about the GitHub Mylyn integration

For a detailed description of the Mylyn and EGit integration please see the following webpage.

http://wiki.eclipse.org/EGit/GitHub/UserGuide

# 32. Writing good commit messages

## Importance of Git commit messages

A commit adds a new version to the repository. This version is described by a commit message.

The commit message describes the changes recorded in a commit and helps the user to understand the history of the files contained in a Git repository.

A commit message should therefore be descriptive and informative without repeating the code changes.

## Guidelines for useful commit messages

A commit message should have a header and a body. The header should be less than 50 characters and the body should wrap its text at 72 so that the commit message is displayed well on the command line or in graphical tools displaying the history. The body should be separated from the header by an empty line.

The body should mainly describe the reason why the change was made. The changes in the file can be reviewed with the help of Git.

The commit message should be in present tense, e.g., "Adds better error handling" instead of "Added better error handling".

The last paragraph can also contain metadata as key-value pairs, also referred to as the commit message footer. This metadata can be used to trigger a certain behavior. For example the Gerrit code review system uses the *Change-Id* key followed by a change-id, which does not change across different versions of the same code review. This changed id is used to identify to which review the message belongs.

The commit message footer can also have e.g., 'Signed-off-by' and may be used to link to a bug tracking system, e.g., 'Bug: 1234'.

## Example message

The following can serve as an example for a commit message.

Short summary (less than 50 characters)

Detailed explanation, if required, line break at around 72 characters

more stuff to describe...

Fixes: bug #8009

Change-Id: I26b5f96ccb7b2293dc9b7a5cba0760294afba9fd

## Good and bad example for a Git history

The following listing shows the output of the git log --oneline command of a Git repository with bad commit messages. The first value in each line is the shortened SHA-1, the second the commit message. This history is not useful.

21a8456 update

29f4219 update

016c696 update

29bc541 update

740a130 initial commit

The next listing shows the history of another Git repository in which better commit messages have been used. This history already gives a good overview about the activities.

7455823 Bug 391086 - Search and filter the model editor tree.

9a84a8a Bug 404207 - Missing DynamicMenuContribution in child selector

952e014 Bug 404187 - Spelling error in Toolbar/Add child

71eeea9 Bug 402875 - Importing model elements from legacy RCP

123672c Bug 403679 - New Application wizard is missing dependencies

97cdb9a Bug 388635 - Creates an id for handlers

The above example also adds the corresponding bug number to the commit message. Some teams (like the Eclipse platform team) use this approach, others prefer to add the bug number to the commit messages.

# Contributing to EGit - Getting the source code

EGit is self-hosted on <git://git.eclipse.org>. See [EGit contributor guide](http://wiki.eclipse.org/EGit/Contributor_Guide) for a description how to work with the EGit and JGit source.

# Git series

This tutorial is part of a series about the Git version control system. See the other tutorials for more information.

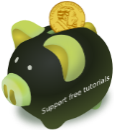
* [Introduction to Git](http://www.vogella.com/tutorials/Git/article.html)
* [Hosting Git repositories at GitHub, Bitbucket or on your own server](http://www.vogella.com/tutorials/GitHosting/article.html)
* [Typical workflows with Git](http://www.vogella.com/tutorials/GitWorkflows/article.html)
* [EGit - Teamprovider for Eclipse](http://www.vogella.com/tutorials/EclipseGit/article.html)

# Get the Book

This tutorial is part of a book available as [paper print](http://www.vogella.com/books/git.html) and electronic form for your [Kindle](http://www.vogella.com/books/git.html).

# About this website

[Support free content](http://www.vogella.com/support.html)

[[](http://www.vogella.com/support.html)](http://www.vogella.com/support.html)

[Questions and discussion](http://www.vogella.com/contact.html)

[[Questions and discussion](http://www.vogella.com/contact.html)](http://www.vogella.com/contact.html)

[Tutorial & code license](http://www.vogella.com/license.html)

[[License](http://www.vogella.com/license.html)](http://www.vogella.com/license.html)

[Get source code](http://www.vogella.com/code/index.html)

[[Source Code](http://www.vogella.com/code/index.html)](http://www.vogella.com/code/index.html)

# 37. Links and Literature

## EGit Resources

[EGit User Guide](http://wiki.eclipse.org/EGit/User_Guide)

[EGit contributor guide](http://wiki.eclipse.org/EGit/Contributor_Guide)

## vogella GmbH training and consulting support