**Eclipse Git Tutorial**

*This tutorial describes the usage of the Eclipse IDE to perform Git operations.*

## [**1. Git support for Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#git-support-for-eclipse)

The Eclipse IDE provides support for the Git version control system. You can easily perform the necessary Git commands like staging, commit, merge, rebase, pull and push via the Eclipse IDE.

|  |  |
| --- | --- |
|  | *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 [Default clone location](https://www.vogella.com/tutorials/EclipseGit/article.html#egitconfiguration_defaultlocation) for more information on this. |

### [**1.1. Command line Git**](https://www.vogella.com/tutorials/EclipseGit/article.html#command-line-git)

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

### [**1.2. Installation of Git support into Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#installation-of-git-support-into-eclipse)

Most Eclipse IDE distributions from Eclipse.org already contain support for Git. In this case no additional installation is required.

If the Git tooling is not available, you can install it via the Eclipse installation manager. Select the **Help**  **Install new Software…​** menu entry. Enter one of the following update site URLs:

# Use this update site to get the latest release

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

# use this update site to get the night build

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

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

## [**2. Exercise: Git user configuration for the Eclipse IDE**](https://www.vogella.com/tutorials/EclipseGit/article.html#eclipsegit_userconfiguration)

If you have already configured Git for the command line, no additional setup is required in the Eclipse IDE and you can skip this exercise.

|  |  |
| --- | --- |
|  | *Interoperability of Git command line settings with the Eclipse IDE*  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. |

### [**2.1. Ensure user and email is configured for Git**](https://www.vogella.com/tutorials/EclipseGit/article.html#ensure-user-and-email-is-configured-for-git)

In your Eclipse IDE, select the **Window**  **Preferences**  **Team**  **Git**  **Configuration** entry. Ensure that your full name and email is available in the user settings. As the Eclipse IDE uses the same settings as the Git command line, this might already be done.

For the user the user.name key is used, for the email the user.email key is used.

If these keys are not available press the Add Entry…​ button and add them. Repeat the procedure for your email address via the user.email key.

### [**2.2. Configure Git to rebase during pull operations**](https://www.vogella.com/tutorials/EclipseGit/article.html#configure-git-to-rebase-during-pull-operations)

To use rebase during the pull operation set the pull.rebase parameter to true. It avoids 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.

This is recommended setting by the author of this text.

### [**2.3. Validate the setup**](https://www.vogella.com/tutorials/EclipseGit/article.html#validate-the-setup)

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

### [**2.4. Configuring the proxy settings**](https://www.vogella.com/tutorials/EclipseGit/article.html#configuring-the-proxy-settings)

If you are using a proxy server, you can configure it via **Windows**  **Preferences**  **General**  **Network Connection**.

## [**3. Exercise: Working with a local Git repository in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#firstgit_with_eclipse)

The following exercise explains how to create a new local Git repository via the Eclipse IDE.

### [**3.1. Create a new Git repository via Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#create-a-new-git-repository-via-eclipse)

Open the Git Repositories view via the **Window**  **Show View**  **Other…​**  **Other…​**  **Git**  **Git Repositories** menu entry. From the toolbar, select the Create a new Git repository and add it to this view entry.

This opens a dialog which allows you to specify the directory for the new Git repository. Select a new directory outside of your workspace. By convention, this directory is a subdirectory in the git folder of the users home directory.

Press the Create button. Now the Git repository is created and a reference to it is added to the Git Repositories view.

### [**3.2. Create .gitignore file**](https://www.vogella.com/tutorials/EclipseGit/article.html#create-gitignore-file)

All files and directories which apply to the pattern described in a top-level .gitignore file are ignored by Git.

For our example, Git should ignore the bin folder which contains the generated class files. Eclipse Git does not allow to create a file directly in the top-level folder of your repository. You have to **do this step outside of the Eclipse IDE**, either via the command line or via your system project explorer.

Create a .gitignore file in the top-level folder of your Git repository. It should contain the following content.

bin

|  |  |
| --- | --- |
|  | Recent versions of MS Windows decided to prevent you from renaming a file in the file explorer without using a file extension. Create a file in Notepad or Editor (new name for Notepad) and select Save-As. Ensure you have removed the .txt extension. |

### [**3.3. Creating an Eclipse project**](https://www.vogella.com/tutorials/EclipseGit/article.html#creating-an-eclipse-project)

Create a new Java project called com.vogella.git.first via the **File**  **New**  **Other…​**  **Java Project** menu entry.

|  |  |
| --- | --- |
|  | If you are using Java 9 or higher, you will be asked to create a module-info.java. The following dialog is not shown if you use Java 8 or lower. |

Right-click on the source folder and select **New**  **Package** and create the com.vogella.git.first package.

Right-click on the package folder and select **New**  **Class** and create the following class.

**package** com**.**vogella**.**git**.**first**;**

**public** **class** **GitTest** **{**

**public** **static** **void** main**(**String**[]** args**)** **{**

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

**}**

**}**

### [**3.4. Put project under version control**](https://www.vogella.com/tutorials/EclipseGit/article.html#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**.

|  |  |
| --- | --- |
|  | If another version control system is installed you have to select that you want to use Git as a version control system.  Git repository creation wizard - Step 1 |

Afterwards, select your existing Git repository from the drop-down list and press the Finish button.

Press the Finish button. Your project is now moved to your Git repository.

The following screenshot shows the directory in the Git Repository view. The .git directory contains the Git repository, the other directories contain the files of the working tree.

### [**3.5. Using the Git Staging view for the initial commit**](https://www.vogella.com/tutorials/EclipseGit/article.html#using-the-git-staging-view-for-the-initial-commit)

Open the Git Staging view, via **Window**  **Show View**  **Other…​**  **Git**  **Git Staging**. In this view drag all files into the Staged Changes area, write a meaningful commit message and press the commit button.

### [**3.6. Doing a change and committing it**](https://www.vogella.com/tutorials/EclipseGit/article.html#doing-a-change-and-committing-it)

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.adoc.

Commit the changes of the GitTest class but do not add and commit the Readme.adoc file to the Git repository.

In 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.adoc file is neither staged nor committed to the Git repository.

### [**3.7. Commit more files**](https://www.vogella.com/tutorials/EclipseGit/article.html#commit-more-files)

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

### [**3.8. Review your commit history via the History view**](https://www.vogella.com/tutorials/EclipseGit/article.html#review-your-commit-history-via-the-history-view)

Open the History view via the **Window**  **Show View**  **Other…​**  **Team**  **History** menu entry.

Use it to 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
* Show all Branches and Tags

### [**3.9. Open an older version with the current version of a file via the History view**](https://www.vogella.com/tutorials/EclipseGit/article.html#open-an-older-version-with-the-current-version-of-a-file-via-the-history-view)

We want to see the version of the GitTest.java file as it was in the first commit. Select the first commit in the history view, find the file selected in the commit and select **Open this version** via the context menu.

### [**3.10. Add more projects to your Git repository**](https://www.vogella.com/tutorials/EclipseGit/article.html#add-more-projects-to-your-git-repository)

You can of course have multiple projects in a Git repository. To validate that, create two more Java projects called com.vogella.egit.multi.java1 and com.vogella.egit.multi.java2. Create at least one Java class in each project.

Afterwards select the new projects, right-click on them and select **Team**  **Share Project…​**  **Git**.

Select your Git repository in the following dialog and add both projects to this repository. Press the Finish button.

### [**3.11. Validate the project move and commit changes**](https://www.vogella.com/tutorials/EclipseGit/article.html#validate-the-project-move-and-commit-changes)

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

The changes have not yet been committed. Now commit all files in the two projects to your Git repository.

## [**4. Exercise: Clone an existing repository**](https://www.vogella.com/tutorials/EclipseGit/article.html#exercise-clone-an-existing-repository)

In this exercise you will clone a Git repository and import the existing projects from this repository into your workspace.

For this, select **File**  **Import…​|Git|Projects from Git**.

Select Clone URI in the next dialog.

Enter the URL to your Git repository which you want to clone. You can use the following example URI.

git://github.com/vogella/eclipse4book.git

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

|  |  |
| --- | --- |
|  | Git supports several protocols, e.g. git://, ssh:// and https://. You can paste the clone URL to the first line of the dialog, the rest of the dialog is filled based on this data.  Some proxy servers block the git:// and ssh:// prThe above link uses the git protocol, alternatively you can also use the http protocol: <http://github.com/vogella/eclipse4book.gitotocols>. If you face issues, please try to use the https:// or http:// protocol. |

After pressing the Next button the system will allow you to import the existing branches. Select at least the master 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, you can import existing projects.

Once this dialog is completed, you have clone the remote repository into a local Git repository. You can use Git operation on these projects.

|  |  |
| --- | --- |
|  | The project may not compile, as you may miss pre-requisites. For this exercises, this can be ignored, the purpose of it was to learn how to clone a repository. |

## [**5. Exercise: Import projects from an existing repository**](https://www.vogella.com/tutorials/EclipseGit/article.html#exercise-import-projects-from-an-existing-repository)

If you have already an existing Git repository you can add it to the Git Repostory view. Afterwards you can import the projects into your workspace via the **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 multiple 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 specify the projects to import. After the import the Eclipse IDE is aware that these projects are part of a Git repository.

## [**6. Exercise: Using interactive rebase in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#eclipsegit_interactiverebase)

Git allows to adjust the local commit history via the interactive rebase functionality. Eclipse provides support for simplified versions of this as well as support for full interactive rebase operations. This includes changing the order of commits or combining, removing and adjusting commits.

### [**6.1. Simple interactive rebase operations available via the History view**](https://www.vogella.com/tutorials/EclipseGit/article.html#simple-interactive-rebase-operations-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. Select afterwards the **Modify**  **Squash** menu entry from the context menu.

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

### [**6.2. Performing a full interactive rebase via Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#performing-a-full-interactive-rebase-via-eclipse)

To start the full interactive rebase open the History view and click **Rebase Interactive** on the context menu. Select the last 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. The Git Interactive Rebase view allow you perform the full interactive rebase functionality.

It shows the rebase plan populated with the commits to be modified. They are sorted in topological order of the sequence in which 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 following actions are available.

| *Table 1. 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.

|  |  |
| --- | --- |
|  | 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. |

## [**7. Additional configuration options for Git**](https://www.vogella.com/tutorials/EclipseGit/article.html#additional-configuration-options-for-git)

### [**7.1. Git user settings in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**7.2. Default clone location**](https://www.vogella.com/tutorials/EclipseGit/article.html#egitconfiguration_defaultlocation)

If you clone a new repository via the Eclipse IDE, it will be cloned by default to a new sub-folder 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 the path, e.g., if you want to store repositories in the folder "git" under the Eclipse workspace you may use ${workspace\_loc}/git. |

### [**7.3. Configuring the toolbar and the menu for Git usage**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**7.4. Eclipse support for SSH based authentication**](https://www.vogella.com/tutorials/EclipseGit/article.html#eclipse-support-for-ssh-based-authentication)

You can create an SSH key pair in Eclipse for SSH based communication. This can be done via **Window**  **Preferences**  **General**  **Network Connection**  **SSH2**.

## [**8. Details on the Git views**](https://www.vogella.com/tutorials/EclipseGit/article.html#details-on-the-git-views)

### [**8.1. Using the Git Repositories view**](https://www.vogella.com/tutorials/EclipseGit/article.html#using-the-git-repositories-view)

The Git Repositories view allows you to browse, add. initialize or clone repositories. It also allows to import projects, manage your branches and much more. You can open this view via **Window**  **Show View**  **Other…​**  **Other…​**  **Git**  **Git Repositories**

The toolbar allow you to:

* add an existing local repository to the view
* clone a repository
* create a new repository

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 entry.

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.

### [**8.2. Using the Git Staging view**](https://www.vogella.com/tutorials/EclipseGit/article.html#using-the-git-staging-view)

The Git Staging view allows staging and committing as well as reverting changes.

This view presents which files you have touched and which files will be included in the next commit. Unstaged Changes lists those changes which you have done locally but which you have not yet added to the staging area. Staged Changes list those changes which you already have added to the staging area. You can drag and drop files from one area to the other. To commit the staged changes you write your commit message and press the Commit button which is highlighted in the following screenshot.

You can switch between different repositories or even restart Eclipse without losing a commit message and it allows incremental staging for changes.

You can open the Git Staging view via the **Window**  **Show View**  **Other…​**  **Git**  **Git Staging** menu.

### [**8.3. Git integration into the Package and the Project Explorer**](https://www.vogella.com/tutorials/EclipseGit/article.html#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 2. 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.

### [**8.4. Using the History view for viewing the Git history**](https://www.vogella.com/tutorials/EclipseGit/article.html#using-the-history-view-for-viewing-the-git-history)

#### [8.4.1. Purpose of the history view](https://www.vogella.com/tutorials/EclipseGit/article.html#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**. Alternatively you can open 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**.

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.

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.

|  |  |
| --- | --- |
|  | 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. |

#### [8.4.2. The History view filters](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

|  |  |
| --- | --- |
|  | 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. |

#### [8.4.3. Using search](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

#### [8.4.4. Showing details of a commit](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**8.5. Commit Viewer**](https://www.vogella.com/tutorials/EclipseGit/article.html#commit-viewer)

The Eclipse IDE allows to view the content of a commit. For example,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.

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.

## [**9. Performing Git operations in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#performing-git-operations-in-eclipse)

### [**9.1. Pull, push and fetch**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**9.2. Basic team operations**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

|  |  |
| --- | --- |
|  | 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)

### [**9.3. Team operations available on the project**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**9.4. Amending a commit**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**9.5. Creating and switching branches in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#creating-and-switching-branches-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.

### [**9.6. Starting a merge operation in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#starting-a-merge-operation-in-eclipse)

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.

### [**9.7. Rebasing a branch onto another branch**](https://www.vogella.com/tutorials/EclipseGit/article.html#rebasing-a-branch-onto-another-branch)

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.

|  |  |
| --- | --- |
|  | 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**.

### [**9.8. Solving conflicts created by merge, rebase or other operations**](https://www.vogella.com/tutorials/EclipseGit/article.html#solving-conflicts-created-by-merge-rebase-or-other-operations)

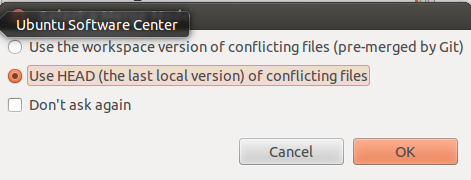
If during a Git operation, two changes are conflicting, you have to solve these conflicts manually. Eclipse highlights the affected files in the Package Explorer and Project Explorer view.

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.

Eclipse also allows to show the common ancestor of both commits to make the merge easier. Press the Hide/Show Ancestor Pane button for that. This is demonstrated by the following screenshots.

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**.

### [**9.9. Git reset and Git reflog**](https://www.vogella.com/tutorials/EclipseGit/article.html#git-reset-and-git-reflog)

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.

### [**9.10. Finding "invisible" commits with the Reflog view**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**9.11. Selecting individual commits via git cherry-pick**](https://www.vogella.com/tutorials/EclipseGit/article.html#selecting-individual-commits-via-git-cherry-pick)

In the History view, you can cherry-pick a commit via the context menu. Cherry-pick allows to move selected changes describes by a commit to another branch. You can also use it in combination with a Git reset to perform a simplified interactive rebase.

Lets assume the following situation, in which you would like to remove the "Bad commit" from the history described by the master branch.

You could start with a hard reset of the branch to origin/master. This will move the master branch pointer to the commit described by origin/master. The Good and the bad commit are not reachable anymore by the master branch.

To re-apply the changes by the "Good commit", cherry-pick the good commit.

This results in a history without the bad commit.

### [**9.12. Comparing and replacing files based on the Git history**](https://www.vogella.com/tutorials/EclipseGit/article.html#comparing-and-replacing-files-based-on-the-git-history)

#### [9.12.1. Comparing files](https://www.vogella.com/tutorials/EclipseGit/article.html#comparing-files)

Eclipse Git allows you to compare a selected file, folder or project with another commit or with the Git index. Use **Team**  **Compare with** to open the menu. Afterwards select with what you want to compare. The HEAD Revision selection is depicted in the following screenshot.

#### [9.12.2. Replacing files](https://www.vogella.com/tutorials/EclipseGit/article.html#replacing-files)

The **Team**  **Replace with** menu entry allows you to replace the current selection with the version contained in the selected commit or the Git index.

## [**10. See Git information line by line (aka git blame)**](https://www.vogella.com/tutorials/EclipseGit/article.html#see-git-information-line-by-line-aka-git-blame)

Eclipse allows to display the information which commit and person change a line. To enable this, right-click on your file and select **Team**  **Show Revision Information**.

|  |  |
| --- | --- |
|  | This action is also available via the line ruler in most editors. |

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

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

### [**10.1. Stash via the Git repository view**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**10.2. Creating patches**](https://www.vogella.com/tutorials/EclipseGit/article.html#creating-patches)

To create a patch for a set of changes with Eclipse, select the resources for which you want to create a patch in the Package Explorer view. Now, 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.

## [**11. Using Eclipse Git with GitHub**](https://www.vogella.com/tutorials/EclipseGit/article.html#using-eclipse-git-with-github)

### [**11.1. Clone project**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**11.2. Push changes**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

## [**12. Eclipse integration with GitHub**](https://www.vogella.com/tutorials/EclipseGit/article.html#eclipse-integration-with-github)

### [**12.1. The Eclipse Mylyn plug-in**](https://www.vogella.com/tutorials/EclipseGit/article.html#the-eclipse-mylyn-plug-in)

Eclipse 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.

### [**12.2. GitHub issue integration**](https://www.vogella.com/tutorials/EclipseGit/article.html#github-issue-integration)

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

### [**12.3. Manage pull requests in Eclipse**](https://www.vogella.com/tutorials/EclipseGit/article.html#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.

### [**12.4. Import projects directly from GitHub**](https://www.vogella.com/tutorials/EclipseGit/article.html#import-projects-directly-from-github)

You can also import now directly projects from GitHub repositories.

### [**12.5. More infos about the GitHub Mylyn integration**](https://www.vogella.com/tutorials/EclipseGit/article.html#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

## [**13. Writing good commit messages**](https://www.vogella.com/tutorials/EclipseGit/article.html#writing-good-commit-messages)

### [**13.1. Importance of Git commit messages**](https://www.vogella.com/tutorials/EclipseGit/article.html#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. It should help the user to understand the history of the repository.

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

### [**13.2. Guidelines for useful commit messages**](https://www.vogella.com/tutorials/EclipseGit/article.html#guidelines-for-useful-commit-messages)

A commit message should have a header and a body. The header should be less than 50 with a maximum of 72 characters. The body should wrap its text at 72. The body is separated from the header by an empty line.

This ensures that the commit message is displayed well on the command line or in graphical tools.

The body describes 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. This data is 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. 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'. Or it may link to a bug tracking system, e.g., 'Bug: 1234'.

### [**13.3. Example message**](https://www.vogella.com/tutorials/EclipseGit/article.html#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

### [**13.4. Good and bad example for a Git history**](https://www.vogella.com/tutorials/EclipseGit/article.html#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 Adds search and filter to the model editor tree

9a84a8a Missing DynamicMenuContribution in child selector

952e014 Fixes spelling error in Toolbar/Add child

71eeea9 Adds option to import model elements from legacy RCP

123672c New Application wizard is missing dependencies

97cdb9a 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.

## [**14. Contributing to EGit - Getting the source code**](https://www.vogella.com/tutorials/EclipseGit/article.html#contributing-to-egit-getting-the-source-code)

This support is provided by the EGit project via a set of plug-ins (software component).

Eclipse uses the JGit library to perform the Git commands. JGit is a library which implements the Git functionality in Java.

EGit is 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.

Vv