Git Manual Saved on: 2016-11-11

This document will guide through the most important information for using GIT in a project scope. It is a manual and process description in one document.

It does not make a difference if you are working for HMI, middleware or only on target or on simulation. All developers can use this version control system. The content of this document is based on GitStartPage.pdf being produced from ECN Robert Streitfeld. Many thanks to the team, giving us such good starting point. The Git based development is a highly collaborative from of SCM it solves several deficiencies seen in ClearCase and it is expected to continue even when other SCM systems like RTC become introduced. So it is worthwhile to start with Git today.

Please, read the complete document before starting with the short excurse in the practice chapter. It will refer back to the places in this document, to better memorize the steps. If you are newly starting using Git in g3g\_linux, please setup Git configuration as described in [https://inside-docupedia.bosch.com/.../Using+git](https://inside-docupedia.bosch.com/confluence/display/gen3generic/Using+git) and then execute the clone command as described in 3.3 (Load latest Software) This command will take time and give you enough waiting time to read the complete document.

**We wish you much fun with the new collaboration tool called Git.**

Yours sincerely,

The Administrators.

**Table of contents**

[1 Terminology 2](#_Toc466638235)

[1.1 Basic terms 2](#_Toc466638236)

[1.2 Advanced terms 3](#_Toc466638237)

[1.3 Administrators 3](#_Toc466638238)

[2 Git Basics 3](#_Toc466638239)

[2.1 All is local 3](#_Toc466638240)

[2.2 Snapshots, Not Differences 4](#_Toc466638241)

[2.3 Delete goes by “unlinking” a commit 5](#_Toc466638242)

[2.4 Some useful Git functions 5](#_Toc466638243)

[2.4.1 Git command line functions 5](#_Toc466638244)

[2.4.2 Git visual version tree with gitk 6](#_Toc466638245)

[2.4.3 Git visual staging area with git gui 7](#_Toc466638246)

[2.4.4 Visual difference tracking with git gui blame 8](#_Toc466638247)

[3 Software development in CM-AI HMI-Base 9](#_Toc466638248)

[3.1 Clone the initial Repository 9](#_Toc466638249)

[3.1.1 Filestorage for Repos 10](#_Toc466638250)

[3.1.2 Before further reading 10](#_Toc466638251)

[3.2 Using super repository and directory structure 10](#_Toc466638252)

[3.3 Load latest Software 11](#_Toc466638253)

[3.4 Branches 13](#_Toc466638254)

[3.4.1 Local branches 14](#_Toc466638255)

[3.4.2 Remote branches 14](#_Toc466638256)

[3.5 Remote branch types 15](#_Toc466638257)

[3.5.1 Main development branch 15](#_Toc466638258)

[3.5.2 User branches (e.g. usr/<userid>/branchname) 15](#_Toc466638259)

[3.5.3 Share (team) branches (e.g. share/branchname) 15](#_Toc466638260)

[3.5.4 Stabilization branches 16](#_Toc466638261)

[3.5.5 Integration branches \_import / \_export 16](#_Toc466638262)

[3.6 Exchange between local and remote branch 16](#_Toc466638263)

[3.6.1 Rebase branch: Loading the latest revision of the same branch from remote 16](#_Toc466638264)

[3.6.2 Merge changes between two different branches 18](#_Toc466638265)

[3.6.3 Sending the latest changes to a branch in remote 18](#_Toc466638266)

[3.6.4 Deliver changes on latest development branch in a super repository 19](#_Toc466638267)

[3.6.5 Removing remote and local branches 19](#_Toc466638268)

[3.6.6 Pull requests in stabilization phase 20](#_Toc466638269)

[3.6.7 Exchanging source changes as patch files 20](#_Toc466638270)

[4 CM-AI commit message guidelines 21](#_Toc466638271)

[5 Modifying the history (clubbing commits) 22](#_Toc466638272)

[5.1 Guidelines for commit contents 22](#_Toc466638273)

[5.2 Ways for clubbing commits 22](#_Toc466638274)

[6 Integration with ClearCase 23](#_Toc466638275)

[6.1 Synchronization from ClearCase to Git (import mode) 23](#_Toc466638276)

[6.2 Synchronization from Git to ClearCase (export mode) 24](#_Toc466638277)

[7 Why Git 25](#_Toc466638278)

[7.1 Popularity 25](#_Toc466638279)

[7.2 Who uses Git 25](#_Toc466638280)

[7.3 Git and ClearCase speed comparison 27](#_Toc466638281)

[8 Known problems and solutions 28](#_Toc466638282)

[9 Git Recommended Links 28](#_Toc466638283)

# Terminology

## Basic terms

|  |  |
| --- | --- |
| Git terminus | Explanation |
| Branch | Branches can be local or remote. They can be easily created, renamed or removed. |
| Checkout | Extracts a certain version of all files from the repository to your working files.  This is different from ClearCase. You just switch your currently used version and/or branch. |
| Clone | A clone is a complete copy of a repository containing the complete history. It contains all commits, tags and branches.  Usually you will start working on some project by making a clone of some official repository. |
| Commit | With commit, you create a new version containing the changes from the stage.  In contrast to ClearCase, you can commit a whole set of files at once, not a one by one check in. |
| Fetch | Transfer commits from a remote repository. Local branches are not changed. |
| Merge | Combine the changes done in any branch (local/remote) with the current branch. |
| Pull | Fetch + merge with one remote branch. Local branch will be modified. |
| Push | Transfer local commits to remote repository. |
| Rebase | Not the same as in ClearCase! Rebase means replaying modifications on top of a different commit. |
| Remote | A remote is a repository located in a different location either in the local file system, USB-stick, server, planet, etc. |
| Repository | In Git the project tree itself contains the repository, which is the complete project history, branches and tags. This is stored in the .git subdirectory in the project tree root.  Despite of what you may think, this will not fill up your HD as Git is extremely efficient in storing differences and compressing. |
| Super repository | Same as repository but containing links to submodules. The .git of submodules is cloned to .git/modules/sumoduleName/...  A super repository versions and maintains only single commit refs of each submodule. |
| Submodule | Same as a repository but positioned “inside” a super repository. The .git folder is in fact only a .git file back referencing into the super repository. |
| Staging | Prepare your next commit by selecting from the changes done on the work files. The set of changes selected is called stage. |
| Tag | Unique name that can be added to a commit for future reference. Git will check the uniqueness of the tag in a repository. |
| Unstaged | Working copy that contains changes which are not yet committed. |

## Advanced terms

|  |  |
| --- | --- |
| Git terminus | Explanation |
| Bare repository | Consists only of the content of the .git folder in a personal repository.  There are no working files available, thus branches cannot be checked out. This again enables pushing on all branches which is why bare repositories are used for publishing or in centralized environments. |
| Cherry picking | Allows to transfer changes done in one commit to any other branch. Ideal for bug fixes which need to be applied on several branches. For this reason, commits should contain only one change at a time. This can be achieved by splitting up the current changes in the working files into several commits with the help of staging |
| Squash | Combine several commits into one. |
| Stash | Something like an unnamed temporary branch, where changes can be put aside and applied back later on. |

See <http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html>

## Administrators

The admins of the git server will get a different documentation for maintenance.

See confluence wiki

[Git administration](https://inside-docupedia.bosch.com/confluence/display/cmaicfnavi/Git%2Badministration)

[Gitolite](https://inside-docupedia.bosch.com/confluence/display/cmaicfnavi/gitolite)

# Git Basics

Normally we recommend reading <http://git-scm.com/book>. However, some basics seem to be important and frequently asked by our developers. Please review also [https://inside-docupedia.bosch.com/.../Using+git](https://inside-docupedia.bosch.com/confluence/display/gen3generic/Using+git) before struggling around.

Always helpful is also git <cmd> --help.

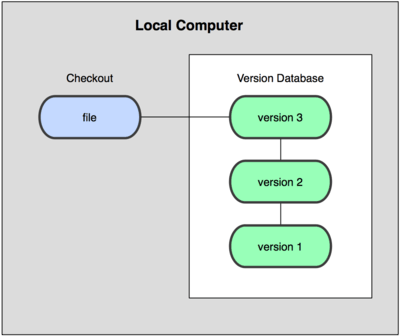
This chapter summarizes in short the content of the basics from git-scm.

## All is local

To deal with this issue, programmers long ago developed local Version Control Systems (VCS) that had a simple database that kept all the changes to files under revision control. The next major issue that people encounter is that they need to collaborate with developers on other systems.

In a Distributed Version Control System (DVCS; such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files: they fully mirror the repository. This means that each Git client user takes a snapshot of the full set of the files in the repository they are using and stores then on their computer.

Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Version control database is present on each local computer and on each server.

Git has 3 main states that your files can reside in:

* ***Modified:*** means that you have changed files, but not yet committed them to your version database.
* ***Staged:*** means that you have marked a modified file in its current version to go into your next commit snapshot.
* ***Committed:*** means the data is safely stored in your local version database.

This leads us to the three main sections of a Git project (figure left):

* Git directory
* Working directory
* Staging area.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.



From work to version

## Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time as illustrated below.



Other systems than Git tend to store data as changes to a base version of each file. Git doesn’t store the file again just a reference to the previous identical file it has already stored. Git thinks about its data more like in next Figure.



Git stores data as snapshots of the project over time. The mechanism that Git uses for this check summing is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

SHAID: 24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything by the hash value of its contents and not by file name.

## Delete goes by “unlinking” a commit

When a developer wants to undo something, typically, they will delete a file or directory. Git doesn’t work this way.

If you need to undo something, Git allows you to start a new version chain (branch) at the point of any past commit. Git does not forget the content that you undo: instead, the content you want to undo becomes part of the data in the whole linked version history.

So, users decide which versions (commits) are visible and all other commits remain in the database but are not visible to other users.

## Some useful Git functions

### Git command line functions

Git has many sub-commands but one of the most important is **git status.** This will provide the current health status of your view.

To push something into the version control you will perform three steps in sequence:

1. **git add** puts some files in a bucket called “staging area”. You can collect things by multiple calls.

2. **git commit** will pack your staging area together to become the next version. This next version contains a comment you provide and will be identifiable with a commit sha1 ID.

3. With **git push** you will send one or more commits since last sync onto remote server. After git push, other users can access your material.

If you work on branches read the chapter 3.4 - 3.6. **git checkout** is used to retrieve a specific version into your view and **git reset** to make modifications to the repository or reset the version information.

If you need a kind of a local mini backup (one level take away) you will use **git stash** **save** or **git stash pop**.

### Git visual version tree with gitk

*gitk* is a simple graphic tool on top of the command line. It visualizes versions (i.e. commits) and branches. This tool can switch between two basic modes:

1. All Files on one branch (here as example “**master**”) and

2. All Branches (which may be a bit overwhelming but helps to see “the others” (here several hmi\_... branches become visible).

|  |  |
| --- | --- |
| **All Files (One Branch)** | **All Branches** |
|  |  |

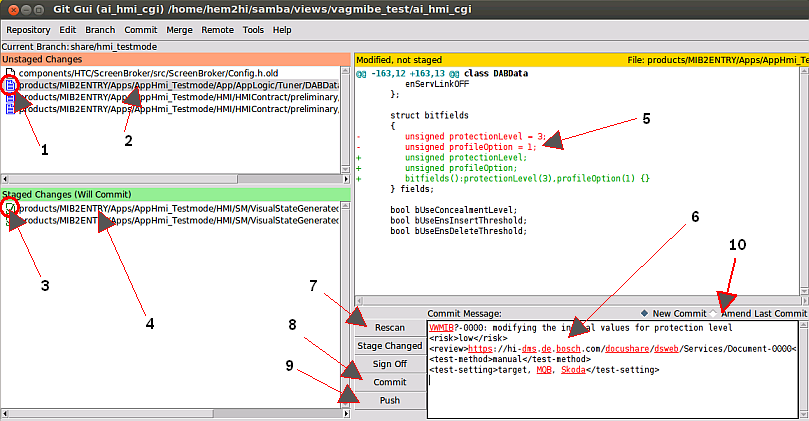
To create the **All Branches** view you have to make this settings in gitk only once:

|  |  |
| --- | --- |
| View -> New View... | Make settings according to this and press OK |
|  |  |

If you face a problem that switching to “All Branches” doesn’t work you can alternatively start gitk with parameter **gitk --all** to see all branches. Appending ‘**&**’ at the end of the command, **gitk –all&** will make the terminal functional while the gitk is displayed.

### Git visual staging area with git gui

A second tool helping to ease the use of git version control is: **“git gui”**. It can be started from out of gitk menu: File->”Start git gui” or directly from the command line with **git gui.** This displays a screen with all relevant information to bring your current modifications into the next git commit.



By clicking on (1) or (3) you can decide, which of the modifications made in your working copy shall be part of the upcoming commit. Whatever is listed in the green staged changes will be in the commit. The changes listed in the left upper area, Unstaged changes will not be committed (even if modified).

By clicking on (2) or (4) the current modification gets highlighted and the differences can be reviewed in the upper right window (5). Here the red lines are removed against pervious commit, the green lines are added.

With the buttons (7), (8) and (9) the commands git status (7), git commit (8) and git push (9) are getting executed. Surely you can also use stash but this is probably not so important at first.

With the button (7) **Rescan**, the command git status is getting executed.

With the button (8) **Commit**, the command git commit is getting executed.

With the button (9) **Push**, the command git push is getting executed.

In the text box (6) you can add a multiline commit message. Please, follow the rules in section 5 to format the template.

One additional interesting feature is the “**Amend Last Commit**” (10). This allows you to reopen the previous last commit you have already made once. It will put all previously made modifications back into the green staging area and rewrite the previously made commit message.

You can now “add” your new modifications to this previous commit and create a new one including all modifications. The new commit will then replace the amended commit.

Let’s assume you made a change and found something was missed. You now want to amend the previous commit:

|  |  |
| --- | --- |
|  |  |
|  |  |

The previously made commit will remain in your commit log, but it is not part of the current branch. So you and others can’t see it in the commit list anymore. And the list of versions looks nice with correct content.

### Visual difference tracking with git gui blame

One more interesting feature in the git gui tool is the file viewer called **git gui blame**. It displays what revision and author last modified each line of a file. This is helpful to find out when and by whom a modification was brought in is here visible. The file viewer displays the files changes history and how it changed over the time. Following command to start the tool from command line:

**git gui blame** directory/myFile.cpp

The following series of screens shows most important actions:

|  |  |
| --- | --- |
| Tool starts and reads the file history. White areas are potential to change still | One area clicked highlights in green all portions belonging to the commit. |
|  |  |
| Click on the commit link to move to the file’s status when this commit happened. | Click and hold in upper left corner to move back in version list. |
|  |  |

# Software development in CM-AI HMI-Base

This chapter describes how CM-AI HMI-Base is working with Git. Please, read the complete chapter first and do the practice afterwards.

Before continuing here - we assume you have made the [**Git-Setup**](https://inside-docupedia.bosch.com/confluence/display/gen3generic/Using+git#Usinggit-Setup)**.** *Git* is already installed in Ubuntu. Few configuration settings need to be adjusted. Please, execute these general settings:

git config --global user.email **<your.email@bosch.com>**

git config --global user.name **<userid>**

git config --global core.editor "gedit -w -s"

git config --global merge.tool kdiff3

git config --global diff.guitool kdiff3

git config --global color.ui true

git config --global pull.rebase true

git config --global core.bigFileThreshold 100m

Edit your local .bash\_aliases file and add one important alias, which will be used frequently when maintaining submodules:

# copy this script to your home folder

****

Add your required aliases for git usage into this file and concat it to the bash aliases.

**gedit ~/.bash\_aliases**

# add this alias

**source ~/.bash\_git\_aliases**

Depending on your work location/region, set the alias for your gitserver to the closest mirror:

|  |  |  |
| --- | --- | --- |
| Hildesheim | **EMEA** | **git config --global url.”ssh://gitolite@si0vmc0298.de.bosch.com/”.insteadOf gitserver:** |
| Bangalore | **APAC** | **git config --global url.”ssh://gitolite@kor022l.apac.bosch.com/”.insteadOf gitserver:** |
| Coimbatore | **APAC** | **git config --global url.”ssh://gitolite@kor022l.apac.bosch.com/”.insteadOf gitserver:** |
| Suzhou | **CHINA** | **git config --global url.”ssh://gitolite@sgpvmc0232.apac.bosch.com/”.insteadOf gitserver:** |

Info: Apac server is KOR022**L**.apac.bosch.com

## Clone the initial Repository

To improve the understanding of git and to have the opportunity to practice git, first clone the **training repository** using the mentioned command. **training** repo is intended to make any kind of tryout. It’s cloned very quickly and can use this while reading.

**cd ~/samba/git**

**git clone gitserver:training**

The training repository is getting created into a new directory “training” now call in a second terminal.

### Filestorage for Repos

Our development environment is using a combination of windows and Linux. While the HMI Toolchain uses Windows tools, the remaining application software is built and executed in a Linux environment. The connection between this two different OSs can be done in two ways depending on the Development PCs configuration.

**case 1 :** If the main host is a Windows PC the Linux environment could be a Linux virtual machine guest [virtualbox.org].

**case 2 :** If the Dev PC is a native Linux machine, the Windows could be a virtual machine.

**case 3 :** Both machines could be separated PCs.

**case 4 :** If your setup is completely configured for Windows only development your setup is preferably using Cygwin or Mingw to execute scripts and build commands.

|  |  |
| --- | --- |
| Case 1 Win Host, Linux Guest | Use samba share for storing files in Linux file system |
| Case 2 Linux Host, Win Guest | Use vmshare for storing files in Linux system |
| Case 3 Two machines | Use samba share for storing files in Linux file system |
| Case 4 Only Win Host | Use Cygwin or mingw to execute scripting |

### Before further reading

Additionally, **“Clone the initial Repository” while reading further**; this clone will take a while and you can safely read the remaining chapters and do the real practice afterwards.

**cd ~/samba/views**

|  |  |
| --- | --- |
| AID HMI-Base | **git clone --recursive --branch=g3g\_int gitserver:g3g\_linux** |
| Nissan NCG3 | **git clone --recursive --branch=nissan\_ncg3\_int gitserver:nincg3** |
| CoC HMI | **git clone --recursive --branch=cochmi\_int gitserver:cochmi** |
| Chery | **git clone -–recursive --branch=chery\_m31t\_high\_int gitserver:chery** |
| PSA | **git clone -–recursive --branch=psa\_rcc\_int gitserver:psarcc** |
| Other repos | Check with the git administrators for the different repos available |

This downloads successively all submodules (vobs) one after the other of the repo.

After the clone has been completed, get into the git repositories root directory and start setting the environment. Throughout this document we use g3g\_linux repo for examples. Please, interpret the corresponding examples.

**cd ~/samba/views/g3g\_linux**

**./start**

**But let’s first continue reading.** Use the training repo to try out some commands.

## Using super repository and directory structure

The new git development environment contains all the former ClearCase-VOBs as its top top level directories. Each of the ClearCase-VOB is an own git repository. As these repositories are contained in a (root) super repository, they are called submodules.

**The super repository only stores references to certain commit hashes of the submodules**, a little similar to the config spec in ClearCase. This allows a hierarchical baselining:

**v1**

**v2**

**v3**

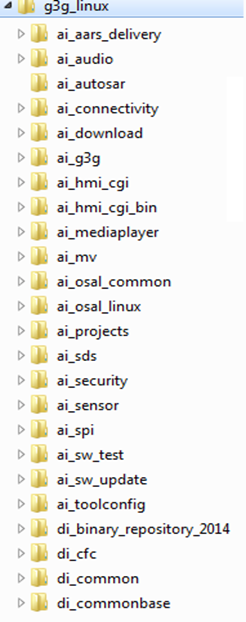
**v4**

**g3g\_linux**

**ai\_audio**

**ai\_hmi\_cgi**

**ai\_projects**



g3g\_int

g3g\_int

g3g\_int

g3g\_int

If a change is made as here e.g. in ai\_hmi\_cgi, first, a new commit on the branch in the corresponding submodule is created.

The second step is that an overall version is updated, on the root level. Both activities can be done independently and require some caution.

## Load latest Software

If you have already cloned the Git repository and want to **update to the latest overall version** of the super repository and you have cleaned up any untracked files from your working directory, you should issue the following commands:   
It is important to change the dir to the **Git root directory**! **~/samba/views/g3g\_linux/**

**cd ~/samba/views/g3g\_linux**

**git status**

# On branch g3g\_int

**nothing to commit** (working directory clean)

**git pull**

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (6/6), done.

remote: Total 6 (delta 3), reused 0 (delta 0)

Unpacking objects: 100% (6/6), done.

From hi-z8957.hi.de.bosch.com:g3g\_linux

0bc93c2..7490332 g3g\_int -> origin/g3g\_int

**Fetching** submodule ai\_hmi\_cgi

remote: Counting objects: 5925, done.

remote: Compressing objects: 100% (1677/1677), done.

remote: Total 5362 (delta 3646), reused 5341 (delta 3633)

Receiving objects: 100% (5362/5362), 51.44 MiB | 11.25 MiB/s, done.

Resolving deltas: 100% (3646/3646), completed with 306 local objects.

From hi-z8957.hi.de.bosch.com:ai\_hmi\_cgi

65dd95a..6c624c2 g3g\_int -> origin/g3g\_int

**Fetching** submodule ai\_projects

...

Now the super repository is on latest status and the submodules (vob) are fetched from remote, but still the submodules need to be updated to the latest status with a second command:

**git submodule update --init**

Submodule 'ai\_aars\_delivery' () registered for path 'ai\_aars\_delivery'

Submodule 'ai\_audio' () registered for path 'ai\_audio'

Submodule 'ai\_download' () registered for path 'ai\_download'

Submodule 'ai\_g3g' () registered for path 'ai\_g3g'

...

Submodule path 'ai\_g3g': checked out '0c145a5db8c32c07db7119d30b63ff46d4512853'

Submodule path 'ai\_hmi\_cgi': checked out 'c399049ef4938bd81bae7e5bb0ff7331f4d5e8a7'

Submodule path 'ai\_mediaplayer': checked out '5792054304bc698d306a64ccddf76d080363ee0e'

Submodule path 'ai\_mv': checked out '783c19e2a8d995ffe2e06a1fa4c9a631ddb9e6e9'

...

**Please close and reopen your build console to update to the latest environment settings.**

Now your complete repository is on the status as the remote super repository branch guides. BUT

**Be aware that your submodules (vobs) are checked out with explicit commits (not branches). Git calls this going to detached HEAD.**

Here is an example:

**cd ai\_mediaplayer**

**git status**

# Not currently on any branch.

nothing to commit (working directory clean)

...

You notice here the following effects:

If you are not having any branches in this submodule **all is fine and nothing needs to be done.**

**If you had a branch locally and you want to continue in this submodule** **you have to first checkout your branch in this submodule**.

In the following picture the important thing is your working copy (yellow ball) is already new but your branch reference still points to the old commit.

To get the local g3g\_int into place call:

**git checkout -B g3g\_int HEAD**

Switched to and reset branch 'g3g\_int'

|  |  |
| --- | --- |
| after **git submodule update --init** | after **git checkout -B g3g\_int HEAD** |
|  |  |

If you have forgotten to update your local branch reference and you continue with a another commit your changes go as detached HEAD into the repository. From this space you will not be able to send this to the remote server. So you will have to put the change back on your intended branch.

**git add testfile5**

**git commit -m "test5"**

**[detached HEAD a66a5b4]** test5

0 files changed

create mode 100644 testfile5

**git status**

# Not currently on any branch.

nothing to commit (working directory clean)

**git branch**

\* (no branch)

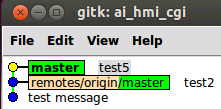
g3g\_int

|  |  |
| --- | --- |
| view “All files...” (no branch, nothing bold!?!) | view “All Branches” (no yellow point!?!) |
|  |  |
| View „All files“ (even local g3g\_int is wrong) | view “All Branches” (even local branch not visible) |
|  |  |

To get back onto your branch you will have to issue the same command:

**git checkout -B g3g\_int HEAD**

Now the changes are kept unchanged in your working copy and the git status is back on your previously intended branch:



The previously made commit is now hidden in your Git data base. If you want to find back the first detached commit you made you can use **git log** with **-g** or alternatively with **--all** :

**git log -g --grep=test5**

commit a66a5b4458b151c916a2ec809a596173db23d496

Reflog: HEAD@{1} (hem2hi <Matthias.Hessling@de.bosch.com>)

Reflog message: reset: moving to a66a5b4

Author: hem2hi <Matthias.Hessling@de.bosch.com>

Date: Fri Oct 11 19:42:29 2013 +0200

test5

So don’t worry! Nothing gets lost... You can repeat the steps by just again checking out the commit hash directly.

**git reset --hard a66a5b4**

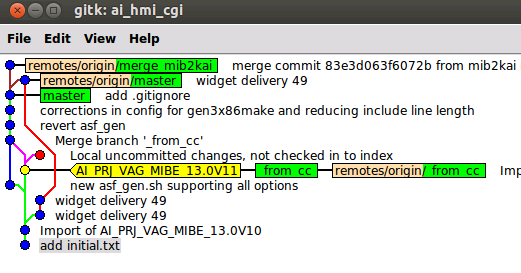
HEAD is now at a66a5b4 test5

## Branches

In contrast to *ClearCase*, branches are very lightweight in git. They are only pointers to a specific commit. There are two types of branches:

1. local branches (here shown as pure green label)
2. remote branches (starting with remotes/origin/ in light brown followed by the branch name in green).

Many pictures currently show a branch called **master**. This is the **default branch name in git** and was used before the branching concept was established for us. Please, realize that **all master branches** are in fact **\_int** branches.



### Local branches

Can be modified (moved forward) by a commit operation. This is where the actual work is done.

These branches are only temporary, which means that, after release to a remote branch, it can and should be deleted. Usually they have the same name as their remote counterparts, but they don't need to be the same:

**git branch test**

# or alternatively

**git checkout -b test**

**git branch**

\_import

g3g\_int

**\*** test

In case branch exists already somewhere and you want to move it to a different position use the capital letter “B”:

**git checkout -B test HEAD**

To delete local braches use following command

**git branch -d test**

Deleted branch test (was c399049).

You can only delete branches if they are not checked out.

Clean up local branches before releasing them (see below). See also some workflow background in chapter 4 “Modifying the history (clubbing commits)”.

If you want to move your local branch work to a remote delivery branch you do this by:

1. Rebase to the remote branch (read further in chapter 3.6.4)
2. Soft reset to remote branch (read further in chapter 3.6.6)
3. Checkin the changes in one commit (read further in chapter 2.4.3)

The above three steps be easier to understand after you have read the corresponding chapters.

### Remote branches

In the Git repository we make use of a specific naming conventions to ensure a good synchronization from and to ClearCase environment. To understand the strategy on branches between Git and ClearCase, the concept of ClearCase is explained briefly. Afterwards it will be easy to understand the concept in Git too.

The integration **branches in ClearCase** are based on a certain naming convention.

Whenever a certain vob contains a feature branch (one or more) those will be named with **<name>\_int**.

When the vob contains stabilization branches (one or more) the notation will be extended to **<name>\_<year>.<[1..n]>\_int**.

In a given view to be synchronized per files only one of these branches can be selected. A view is reflected by a certain **config\_spec** summarizing that entire branch configuration. There will normally be a base labeling / branch which are sometimes patched with some particular labels for selecting some view files to be in a different version. Only the complete label stack per each VOB reflects the exact version description of this view.

Supposed this view is to be synchronized, we want now a developer to be able to select a certain version of this view and having the exact same sources as in ClearCase. For the first step to realize this we make reuse of the same branch naming strategy as in ClearCase in the **super repository** (root folder) using the branch name of the main project providing the config\_spec for import/export.

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| **View / Function** | **Branch names in ClearCase ai\_projects** | **Branch names in other vobs  e.g. di\_middleware\_server** | **Branch names in Git applied in all sub- modules the same** |
| Main feature branch for  Nissan NCG3 (A-IVI) | ai\_prj\_nissan\_ncg3\_int | di\_midw\_g3g\_int, di\_middleware\_nissan\_int, di\_midw\_g3g\_audio\_int, ... | nissan\_ncg3\_int |
| Stabilization branch 1 for  Nissan NCG3 (A-IVI) | ai\_prj\_ nissan\_ncg3\_14.1\_int | di\_midw\_g3g\_int, di\_middleware\_nissan\_14.1\_int, di\_midw\_g3g\_audio\_14.3\_int, ... | nissan\_ncg3\_14.1\_stabi |
| Main feature branch for AID | ai\_g3g\_int | di\_midw\_g3g\_int, dia\_common\_gen3\_int, di\_midw\_g3g\_fea\_reception\_preint, ... | g3g\_int |
| Stabilization branch 1 in AID | ai\_g3g\_14.01\_int | di\_midw\_g3g\_int, dia\_common\_gen3\_int, di\_midw\_g3g\_fea\_reception\_preint, ... | g3g\_14.01\_stabi |

This basically means that individual branches in the different vobs are not visible in the sub modules. To still allow knowing about the roots of the imported label stack this one and the load rules are applied to the commit message on the **\_import** branch which contains the exact copy being imported to Git, so that this could be taken to get the original status in ClearCase. The import process is taking the ai\_projects config\_spec and cut this into pieces per VOB (resp. sub module).

When a given submodule is exported to ClearCase, always the last version to be exported is being committed to an **\_export** branch. This allows detecting patch differences during the synchronization process. See chapter Integration with ClearCase for further information.

## Remote branch types

### Main development branch

The main development branch in each **submodule** (VOB) is “origin/**<pj\_name>\_int”.** This is also the branch being used to export the new software to clearcase. All team members have write access to this branch and are supposed to directly work on this branch.

**If it's not on origin/<pj\_name>\_int, it's not released!**

Main development branches can be identified by the extension **\_int**. These branches are used to export the changes back to clearcase.

A commit to \_int always have to be tested and functional before it becomes pushed to remote!

### User branches (e.g. usr/<userid>/branchname)

These are personal branches pushed to the server mainly for backup reasons. They are readable to other project developers, so they can be used for collaboration to some extent. The user branches shall always start with **“<userid>/”** this helps to better separate the namespace. The slash inside a branch name is a simple text and is just part of the branch name.

Please, use this method to save (backup) your work to the Git server!

**git checkout –b usr/hem2hi/mybranch origin/usr/hem2hi/mybranch**

**git push origin usr/hem2hi/mybranch**

### Share (team) branches (e.g. share/branchname)

All team members in the project have full write access to share branches. The intention of share or team branches is to have a kind of pre-integration step, where teams are collaborating on. A share branch has to have a maintainer (team lead / pre integrator) that executes the delivery to the upstream \_int branch.

If the change is pushed to a share branch, it is explicitly published for the team and shall be functional as expected by the team.

**git push origin share/the\_team\_name**

For workflow using share branches and also the situation when changes in two submodules refer Chapter 3.6.2.

### Stabilization branches

When a project becomes stabe then the changes delivered to it have to undergo a more qualified procedure in terms of review process, testing process and importance filtering. This is done with branches called **<pj\_name>\_<numbering>\_stabi.**

With this typeof branches the team can only deliver modifications via **pull request** method (no direct write access, see chapter 3.6.6).

A defined maintainer (integrator) will then do the takeover of the modifications (commits) provided on a side branch (e.g. user branch or \_int branch).

In this project mode, the commits are tested on this side branch and then the commit id (or a patch file from the same) is sent via mail to the integrator. In the next PCCB, the integrator decides which commits become integrated into the stable branch. As this process is much slower than normal Git SCM the stabi branch will be used only in a very later stage of the project.

### Integration branches \_import / \_export

Additionally there is one branch called <pj\_name>\_int**\_import** . This is the clearcase import branch. See chapter 6.1 for further explanation. These branches can be used to collaborate with ClearCase users. If you want to know your changes against a given ClearCase version you can simply do a **git diff** for the files as in ClearCase. This can be done in two ways:

1. via branch name on the latest status:

**git diff g3g\_int origin/g3g\_int -- path/to/files/to/diff\***

2. via given import labels from project spec

**git diff HEAD AI\_PRJ\_G3G\_LINUX\_14.0F41 -- path/to/files/to/diff\***

## Exchange between local and remote branch

### Rebase branch: Loading the latest revision of the same branch from remote

If you want to load the latest revision from a branch which you are already working on, then we assume you have this branch previously checked out. Also it is required to have all local modifications checked in, by committing those to the repository (or stashing it; read git-scm page for details) While the work is still in progress and a feature branch needs to be brought up to date with the remote branch, we use rebase – as opposed to merge – not to pollute the history with spurious merges. [e.g. [atlas](http://blogs.atlassian.com/2013/10/git-team-workflows-merge-or-rebase/)]. To ensure that this concept is generally followed the **PULL** command executes automatically a **Fetch + Rebase**.

|  |  |
| --- | --- |
| Changes made but others continued the work. | **git merge origin/g3g\_int** ends in extra commit just to rewind. |
|  |  |

Instead of merge, make use of git rebase. This reduces the merge commits to only those where you deliver changes. (next chapter)

|  |  |
| --- | --- |
| Changes made but others continued the work. | **git rebase origin/g3g\_int** ends with repeating your commits on top of latest revision from remote. |
|  | When you press Ctrl-F5 (reload) the dead end commit disappears. |

Resolving conflicts is basically the same as for merging. The only difference is the final step after resolving the merge conflict goes with   
**git rebase --continue** instead of git commit.

**git fetch**

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From hi-z8957.hi.de.bosch.com:training

1f05aa9..17b8851 g3g\_int -> origin/g3g\_int

**git rebase origin/g3g\_int**

First, rewinding head to replay your work on top of it...

Applying: changing mul.cpp

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging mul.cpp

CONFLICT (content): Merge conflict in mul.cpp

Failed to merge in the changes.

Patch failed at 0001 changing mul.cpp

**When you have resolved this problem run "git rebase --continue".**

If you would prefer to skip this patch, instead run "git rebase --skip".

To check out the original branch and stop rebasing run "git rebase --abort".

**git mergetool**

Merging:

mul.cpp

Normal merge conflict for 'mul.cpp':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (kdiff3):

**git rebase --continue**

Applying: changing mul.cpp

If you want to revert to your previously made commit you can use **git log with -g** :

**git log -g --grep=test5**

commit **a66a5b4458b151c916a2ec809a596173db23d496**

Reflog: HEAD@{1} (hem2hi <Matthias.Hessling@de.bosch.com>)

Reflog message: reset: moving to a66a5b4

Author: hem2hi <Matthias.Hessling@de.bosch.com>

Date: Fri Oct 11 19:42:29 2013 +0200

test5

So don’t worry! If a commit is made, Nothing gets lost... You can repeat the steps by just again checking out the commit hash directly.

**git reset --hard a66a5b4458b151c916a2ec809a596173db23d496**

HEAD is now at a66a5b4 test5

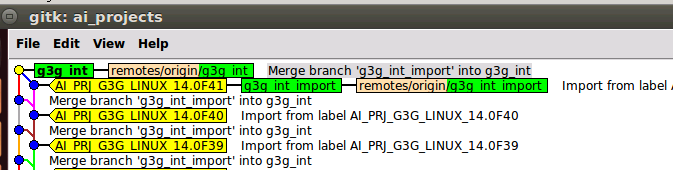
### Merge changes between two different branches

If you are working on a **share branch** we assume you have **committed and pushed** the changes to this branch already. Your git status displays all is clean.

**git status**

nothing to commit (working directory clean)

You now want to move the current status from the share branch to the \_int branch. In this case the merge commit is intended, as it helps to detect when the delivery did take place.



Do a checkout **on the delivery branch** which you want to provide your modifications in.

git checkout g3g\_int

Ensure you are on the latest commit of this branch doing a pull.

git pull origin g3g\_int

Do the merge from your branch to the delivery branch and solve merge conflicts.

git merge share/my\_team

If you have merge conflicts solve them via merge tool

git mergetool

Finally use git gui to review the modifications and writing the commit message.

git gui

**Check if software still builds and Test your feature is working fine.**

Now push the changes:

git push origin g3g\_int

To continue working on the same branch, we have to **reset the local work branch** pointer to the same commit which we just created and pushed. Because the commit history breaks here, intentionally we need to force the push. This action is accounted to the branch maintainer only.

git checkout -B share/my\_team g3g\_int

git push --force origin share/my\_team

**Important**:

**Do not pull first the delivery branch (e.g. g3g\_int) to the local branch.**

This cause all commits from your share branch becoming copied to the delivery branch, which we don’t want.

When we change to root directory, we will see the git status now telling new commits in submodules.

### Sending the latest changes to a branch in remote

Remote branches can be modified only by a push operation. This is valid for root as well as for the submodules (VOBS). You must check that no-one requires a remote branch before you delete it.

**git push origin g3g\_int**

Counting objects: 6, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (5/5), 439 bytes, done.

Total 5 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (5/5), done.

To hi-z8957.hi.de.bosch.com:di\_cfc

dd2be91..79e5cd0 g3g\_int -> g3g\_int

### Deliver changes on latest development branch in a super repository

Updating only the branch of the submodule (VOB) you are currently working on, might not be sufficient if there are any dependencies to other submodules. If you want to connect the currently selected commits in your super repository to an overall version as shown in chapter 3.2, you must perform a **two-step-push** to get the release done.

In case you are adding changes to other submodules as ai\_hmi\_cgi you have to indicate your dependencies to the corresponding clearcase label! Otherwise the integrator will fail to integrate hmi into the main project.

1. Push changes in one (or more) **submodule**(s)   
Repeat these steps for all the submodules in which you have made the changes.

git push origin g3g\_int

2. Push a new version pointer on **root** (super repository)

git status

# on branch g3g\_int

...

# modified: ai\_hmi\_cgi **(new commits)**

# modified: ai\_projects **(new commits)**

git add ai\_hmi\_cgi ai\_projects

git commit –m “CMG3G-1234: changes finalized for new feature”

git push origin g3g\_int

### Removing remote and local branches

Branches (especially user branches) are intended to be cleaned up regularly. Branches that no longer needed any more are to be cleaned up by the owner / maintainer of the branch. Removing a branch is done with two commands in series:

1. Remove local copy

2. Remove remote copy

**git branch -r -d origin/branchname**

Deleted remote branch origin/branchname (was 79e5cd0).

# AND remote delete by pushing with **a colon** in front of the name

**git push origin :branchname**

To git@hi-z8957.hi.de.bosch.com:g3g\_linux

- [deleted] branchname

Users do have certain rights to delete branches! If you face an **error while deleting** your branch, please contact the **administrators** for clarification.

**cd ~/bosch/g3g\_linux**

**git status**

# On branch share/my\_team

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

# (commit or discard the untracked or modified content in submodules)

#

# modified: ai\_hmi\_cgi **(new commits)**

# modified: ai\_projects **(new commits)**

#

We repeat the same procedure as on the submodules as last step.

**git add ai\_hmi\_cgi**

**git add ai\_projects**

After this we **commit the changes with git gui** and push it to the server. In the meantime some other changes have been pushed to the share branch we will have to go back and first merge the changes in the sub module! Otherwise, our current commit would point to earlier versions in some submodules.

### Pull requests in stabilization phase

The \_stabi branch requires that modifications are taken as change from a different branch or tag. The developer creates a usr branch on basis of the \_stabi branch and does the required modification. Before the developer pushes the changes to the remote repo, squashes all commits to one single commit before the push to remote. With this a single reference commit id, it can be used by the integrator to apply this back on the \_stabi branch with a **cherry pick** (this preserves the originator of the commit).

### Exchanging source changes as patch files

To provide someone with a single modification you can use the patch command. This patch file contains all required modifications including date, time and author of the modification, which helps later to identify the originator.

To create a patch use: (see --help for creating patches from ranges)

**git format –patch**

To integrate the patch file use:

**git apply --patch <patchfilename>**

# CM-AI commit message guidelines

Please follow these guidelines whenever you release commits.

Local commits can differ, but must be edited according to the rules for release.

1. **Rules for all lines**

a. Useful comment, not only “bug fixed” but make sure the content follows the chapter 4.1 Guidelines for commit contents

b. Explain the result of the commit not the state before

c. Write your commit message in the present tense: "Fix bug" and not "Fixed bug"

1. **First line should be short (50 chars or less) summary**

Always use references to work package if available (examples for the first line):

a. MMS or JIRA tickets

"MMS-288843: Fix speech problem after dataset change"

b. JIRA Feature

"VWMIBE-22929: Add POI list handling to the POI Filter"   
c. Change request:

"CR-201169: Store only location cache content in TIMA's persistent memory.   
d. Roadmap

"RM-11074: Implemented Export/Import for address-book widget"

1. **Blank line separating the summary from the body is critical (unless you omit the body entirely)**
2. **Body if necessary contains more detailed explanatory text, if necessary. Wrap it to 72 characters.**
3. **Further paragraphs come after blank lines.**

A template for a commit message shall be like this. The content of the tags shall be selected according to your current commit situation. The definition is as follows: (please remove the text, inside of the tags not valid for you).

**VWMIB?-0000: write what you have done in the nutshell.**

**<risk>low/medium/high</risk>**

**<dependencies>dep. to other changes</dependencies>**

**<review>https://hi-dms.de.bosch.com/docushare/dsweb/Services/Document-0000</review>**

**<test-method>manual/automated</test-method>**

**<test-setting>target/LSIM, MQB/PQ, Skoda, VW, Seat</test-setting>**

**<test-result>OK/NOT\_OK/NO\_TEST</test-result>**

# Modifying the history (clubbing commits)

## Guidelines for commit contents

**Commit often on your local repository**

Don't forget, a commit is local in the first place. Therefore you can/should record as many steps as possible of your work. This gives higher flexibility during development. You can go back and forth between your changes, use it as an undo history, or try different implementation paths. It can also help to find which change has inserted a bug and revert that commit.

Your commits only get published by a push operation to a public server.

**Be sure you have a story worth telling when you push**

Don't push all commits being understandable for the rest of the world. To combine many commits to only one commit for the delivery, will be explained in this chapter.

**Keep changes as atomic as possible**

Developers usually get carried away while writing code for example, start some feature, fix some bugs, do some refactoring, continue working on the feature...etc

Developers should keep changes that affect functionality separate from those that just affect code layout, indentation, whitespace, filenames etc. Refactoring should never be a part of a feature commit but a separate one. This makes merging of functional code much easier later on.

**Keep changes as complete as possible**

Half a bug fix is not worth publishing, nor is a non-functioning feature part.

If a local commit doesn't even compile, you should not publish it. Combine all necessary commits that are needed to make a compliable and useful commit.

**Prepare for cherry pick**

A very good reason for having atomic commits is the cherry pick feature. Git allows you to pick the changes from a single commit and apply them to some other branch. Of course if the commit contains more than just the bug fix, things get more complicated.

## Ways for clubbing commits

**Using the Git stage**

Here the staging area (aka Index) comes into the game. It allows you to select only some of your changes and mark them as relevant for the next commit. This is done by adding those changes to the staging area with "git add". This allows to easily separate bug fixes, features, etc. See chapter 2.4.3 for using the git gui to better understand how to use the stage.

**Amend the last commit**

Stage the changes you want to add to the last commit. Use the command.

**git --commit --amend**

Or better, use the **git gui** to better understand what you bring together. See chapter 2.4.3 for using git gui.

**Squash several commits into only one commit to push?**

This is the fastest and easiest way to go to combine (or sum up) a number of your commits into one, and to rewrite a new commit message for upcoming delivery**. Only do this on user branches! Never on integration or other shared branches!**

For example, you want to squash the last **3** commits together into one with a new commit message. Then make sure that your local branch is clean, i.e., no uncommitted changes lying around in your branch, and call

**git reset --soft HEAD~3**

You can also replace HEAD~3 by the commit ID of the commit before the last three. This will set the branch back to the specified commit, but will leave all the changes of the last 3 commits in your staging area. (If you want to squash any number of commits, just change the number accordingly.) Now you can issue

**git commit**

This will then create a new commit with all the changes and can give it a new commit message. Then push this commit to the remote repository using

**git push origin <branch\_name>**

This will force the remote branch to rewrite the history, as you have done locally (the usual behavior is just to fast-forward to the new HEAD commit, but this is obviously not possible here).

**Squash several commits into only one commit using rebase**

With **git rebase** command you can squash a given number of commits to only one or more:

**git rebase –i HEAD~<count\_of\_commits>**

An editor will open listing the number of commits to be combined as below:

**pick e59aac3 script execution**

**pick b2bf1f0 replacing key for Vishal Raj (RBEI/ECV) vhr7kor**

**# Rebase 09b20a6..b2bf1f0 onto 09b20a6**

**#**

**# Commands:**

**# p, pick = use commit**

**# r, reword = use commit, but edit the commit message**

**# e, edit = use commit, but stop for amending**

**# s, squash = use commit, but meld into previous commit**

**# f, fixup = like "squash", but discard this commit's log message**

**# x, exec = run command (the rest of the line) using shell**

**#**

**# If you remove a line here THAT COMMIT WILL BE LOST.**

**# However, if you remove everything, the rebase will be aborted.**

**#**

Except for one commit, replace “pick” to “squash” for all the commits. Save and close the editor. This combines all the commits to one single commit.

An editor will open listing the commit messages of all the commits. Edit or rephrase the new commit message for the commit being created. Save and close this editor also. Then push the commit to the server.

**git push origin <branchname>**

**Note:** Commits already pushed to the server cannot be squashed (combined).

# Integration with ClearCase

In chapter 3, remote branches, the basic concept of naming the branches has been described already. This chapter describes how the integration from and to Git works. For synchronization between GIT and clearcase it is required to have an exact copy of sources from ClearCase in Git and also to have an exact copy of Git in ClearCase for the particular cases.

The synchronization – based on the fact that label stacks can be treated in importing from ClearCase – requires that, for each VOB / submodule, the decision has to be made as to whether it is in **import or export mode**. A change of code against the selected direction is not automatically reflected. A developer will have to make the modification manually on the opposite side.

## Synchronization from ClearCase to Git (import mode)

A vob configured for import mode will fetch the label stack and the load rules from the config\_spec, reflecting the view in ClearCase and import these sources to a branch called **<name>\_[<year>.<[1..n]>\_]\_int\_import.** This branch reflects the exact copy from ClearCase.

In a second step these changes are merged onto the corresponding **\_int** branch. This additional merge allows a modification in import VOBs in Git which stays there until the modification gets manually reverted or the same modification is reflected in ClearCase too. The merge process is closely reviewed by the integrator, who also takes care of monitoring the differences between \_import and \_int branch.

A change merged to the **\_int** branch can help to make precursor modifications being expected in the next import. E.g. if a middleware interface is expected to be extended, the HMI developer can simply merge the middleware label to the \_int branch expected with next main integration and develop the HMI changes even before the changes in the same integration cycle.

## Synchronization from Git to ClearCase (export mode)

The export will be controlled by a configuration list mapping of one **\_export** branch following the same naming convention as above and one sub module in Git to one ClearCase **import** branch called ...**\_import**.

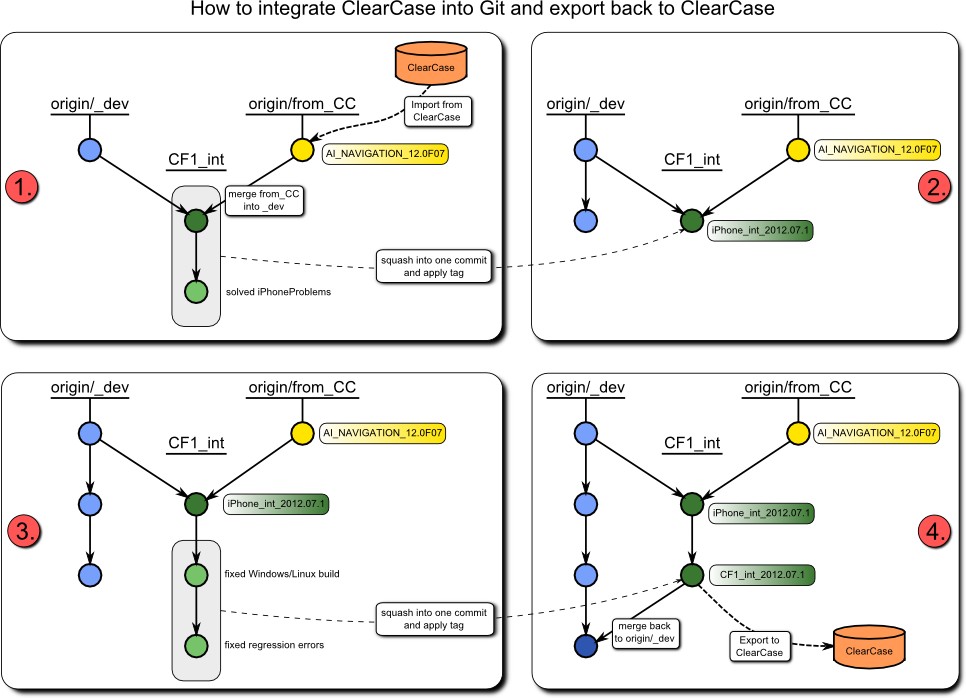
The first step in the export will now be a **forced update** from a given commit for the **\_int** branch to the corresponding **\_export** branch. This means the **\_export** branch always maintains the last time exported version. This branch now gets copied to ClearCase. In the next step an Integrator will integrate this branch as single input to a ClearCase integration branch.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub module / vob to be exported** | **Branch in Git** | **Branch in ClearCase** | **Integrated in ClearCase to** |
| ai\_hmi\_cgi | g3g\_int\_export | ai\_g3g\_from\_git | ai\_hmi\_cgi\_int |
| ai\_hmi\_cgi | g3g\_14.01\_int\_export | ai\_g3g\_14.01\_from\_git | ai\_hmi\_cgi\_14.1\_int |

The integration synchronizates in a similar way to that CF1 is doing (see following picture).

Only the explicitly named sub modules like ai\_hmi\_cgi are exported by the integration. Whenever changes to other vobs are done the corresponding developers have to do normal devReleases into clearcase. All pure import vobs will normally not be exported. Changes between **\_int** branches and ClearCase are just thought as precursors for upcoming expected changes, similar to patch labels in ClearCase.

The following figures show graphically the process of synchronization as it is used in CM-AI/ECN (formerly called CF1).



# Why Git

* Incredible speed on all operations: pull, push, merge, diff, checkout (have a look at speed comparison)
* Collaboration/preintegration
* Instant diff between all files of any two versions in history
* Great merge (merge through rename), easy and fast Tech Talk: Linus Torvalds on Git:

... one of my main statistics is that the kernel is actually one of the biggest open source projects. We have

22,000 files. We've used Git for two years. During those two years, we have averaged 4.5 merges a day, every single day. That's not something you do with something where merging is hard.

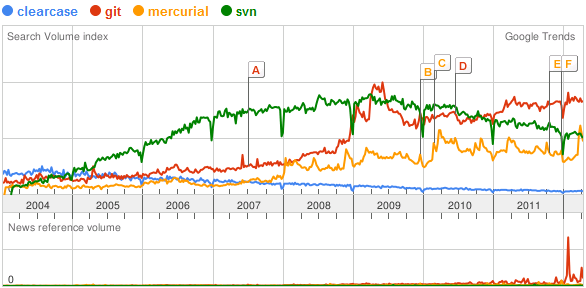
... and I get unhappy if a merge takes more than 5 seconds.

* Cherry picking
* Staging
* Offline repository with complete history and all branches
* Platform independent
* Search even inside changes from history
* Work flow much more simple and non-interrupting

[The thing about Git](http://tomayko.com/writings/the-thing-about-git):

Git means never having to say, "you should have"

## Popularity



From [Google Trends](http://www.google.de/trends/?q=clearcase%2Cgit%2Cmercurial%2Csvn&amp;date=all&amp;geo=all&amp;ctab=0&amp;sort=2&amp;sa=N) on 16.04.2012 Projects registered at Black Duck analytic service:

<http://www.ohloh.net/repositories/compare>

## Who uses Git

Bosch uses Git:

* iPhone development
* ADIT CF3
* CF3: Git server
* SCMPlan\_Gen3.doc
* Cross Divisional Group - Software, Methods, Tools / Engineering Base Software CDG-SMT: Git
* CM-PS (Professional Systems) CM-AI/EAR-H
* CM-AI/PJ-CB (for internal development)

MontaVista delivers linux-based platforms using Git.

GENIVI Alliance uses Git [https://collab.genivi.org/wiki/display/genivi/GENIVI+Git](https://collab.genivi.org/wiki/display/genivi/GENIVI%2BGit).

Other major companies and projects using Git:

Rational Team Concert and Git



The Rational® Adapter for Git allows you to link Rational Team Concert™ change requests and Git commits or Gerrit changes. [Integrating other SCM Systems with Rational Team Concert 2.0: Git](https://jazz.net/library/article/194#Git)

[Using the Rational OSLC Adapter for Git with Rational Team Concert 4.0](https://jazz.net/library/article/1209)

[Rational Adapter for Git](http://pic.dhe.ibm.com/infocenter/rliahelp/v1/index.jsp?topic=%2Fcom.ibm.rational.rlia.git.doc%2Ftopics%2Fc_git_use_ovw.html)

External links:

[Maturity model for source control (SCMM)](http://www.lucasward.net/2010/02/maturity-model-for-source-control-scmm.html) [Why Git is better than X](http://whygitisbetterthanx.com/)

[Git vs Mercurial: Why Git?](http://blogs.atlassian.com/2012/03/git-vs-mercurial-why-git/)

## Git and ClearCase speed comparison

Comparison of Speed for different operations performed in Git and ClearCase

The tests have been done on a Win7 64-bit with 4GB RAM Intel Core2 Quad 2.4 GHz.

Some operations in CC are much faster on dynamic branches, which cannot be used for compiling for performance reasons. A

switch between dynamic and snapshot view is required for optimal performance. The time of the work flow is not measured, only time between user interactions, e.g. creating a branch in ClearCase requires a lot of user steps, Git requires only the name.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | CC [min]  snapshot | CC [min] dynamic | Git [min] | Comment |
| View ai\_navigation | 21:10 | 00:08 | 06:27 | Views are not exactly the same. The Git view contains 50%  more work files.  Nr. of files in view CC: 15.327 files, 2.078 folders Nr. of files in view Git: 23.707 files, 2.591 folders Total size of view CC: 1,62 GB and Git: 2,81 GB  The Views differ in the kind of binaries they contain. The repository contains 13 versions of binaries and 16 integration versions of navigation sources.  The Size of the complete Git repository itself (.git folder) is only 1.15 GB. |
| Switch to different version | 21:10 |  | 00:40 | Changes in over 1600 files. Only 00:12 without binaries. |
| Rebase a branch |  | 01:51 | 00:01 | Rebase without merge candidates (rebased to previous label). |
| Merge two branches (no files to merge) |  | 01:11 | 00:01 | Empty merge with base branch. |
| Merge two branches (~ 420 files to merge) | 25:30 |  | 00:52 | Trivial merges. ClearCase 17 min for merge only, rest for check in of 422 files. |
| Show version tree of a file | 00:02 | 00:02 | 00:01 |  |
| Show differences between two  SW-Versions | - |  | 00:01 | Not easily done in ClearCase. Difference of two dynamic views with difftool required.  Git instantly shows all differences between any two selected  versions. |
| Create a branch without switching | 00:28 |  | 00:01 |  |
| Find modified files in ai\_navigation | 01:20 |  | 00:02 | Changed 3 files in different folders |
| See differences of modified files  (only 3 files) | 00:10 |  | 00:01 | In ClearCase each file has to be compared individually, which takes at least 2-3 secs for every file.  Git shows all the differences instantly |
| Check in modified files (only 3 files) | 00:03 |  | 00:01 |  |
| Create a label of changes (only  3 files) | 05:28 | 01:25 | 00:01 |  |
| Create a label of many changes  (~420 files) | 10:00 |  | 00:01 |  |

# Known problems and solutions

Please read [https://inside-docupedia.bosch.com/.../Using+git](https://inside-docupedia.bosch.com/confluence/display/gen3generic/Using+git) when you face problems. If you don’t find the answer there, and online resources don’t help. Then please contact the administrators to get answers.

DO NOT WAIT OR DO WORK AROUND - OTHERS MIGHT HAVE EXPERIENCED THE SAME PROLEM!

# Git Recommended Links

Cheatsheets and short references

[cheatsheet\_Spickzettel.pdf](https://inside-docupedia.bosch.com/confluence/download/attachments/145575508/cheatsheet_Spickzettel.pdf?version=1&modificationDate=1340610065000&api=v2)

Git Pro useful diagrams [cheat Git](http://cheat.errtheblog.com/s/git)

Books

Real books about Git in CF1 internal library

138. Git - kurz und gut, Sven Riedel

161. Pro Git, Scott Chacon

162. Git Verteilte Versionsverwaltung für Code und Dokumente Valentin Haenel, Julius Plenz

169. - 171. GIT- Dezentrale Versionsverwaltung im Team - Grundlagen und Workflows, Rene Preißel, B. Stachmann

Git tutorials

[Online hands on introductory tutorial for Git on command line](http://try.github.com/levels/1/challenges/1)

<http://www.ralfebert.de/tutorials/git/>

<http://www.ralfebert.de/blog/tools/git_screencast/>

[Top 10 Git Tutorials for Beginners](http://sixrevisions.com/resources/git-tutorials-beginners/)

Git Extensions Video Tutorials

1. [Clone - Git Extensions](http://www.youtube.com/watch?v=TlZXSkJGKF8) 1:47 min

2. [Commit changes](http://www.youtube.com/watch?v=B8uvje6X7lo) 1:29 min

3. [Push changes](http://www.youtube.com/watch?v=JByfXdbVAiE) 1:07 min

4. [Pull changes](http://www.youtube.com/watch?v=9g8gXPsi5Ko) 1:00 min

5. [Handle merge conflicts](http://www.youtube.com/watch?v=Kmc39RvuGM8) 5:49 min

Git explained

[Introduction to Git with Scott Chacon of GitHub](http://www.youtube.com/watch?v=ZDR433b0HJY)

This talk introduces the Git Version Control System by looking at what Git is doing when you run the commands you need to do basic version control with it.

[Git from the bottom up](http://ftp.newartisans.com/pub/git.from.bottom.up.pdf)

Very nice pdf which helps understanding Git.

[A visual Git reference](http://marklodato.github.com/visual-git-guide/index-en.html)

This page gives brief, visual reference for the most common commands in Git.

Once you know a bit about how Git works, this site may solidify your understanding.

Excellent Diagrams. 

[Git ready](http://gitready.com/)

Very good HowTos with categorization for beginner, intermediate and advanced users.

[Git Immersion](http://gitimmersion.com/index.html)

Git Immersion is a guided tour that walks through the fundamentals of Git. Command line only.

[Git Data Transport Commands](http://osteele.com/archives/2008/05/my-git-workflow)

Very good explanation of the data model.

Excellent Diagrams.

[Pro Git Book](http://progit.org/book/)

Same book we have in the library as print out. Can be viewed a html or pdf.

An extract of the most useful diagrams can be found here Git Pro useful diagrams.

[Pro Git reset explained](http://progit.org/2011/07/11/reset.html)

If you really want to understand what happens during reset soft, mixed, hard. It also explains differences to checkout.

Excellent Diagrams. 

[AlBlue’s Blog](http://alblue.bandlem.com/Tag/git/)

Good explanations on various Git topics

[Understanding the Git Workflow](http://sandofsky.com/blog/git-workflow.html)

Also explains, why you should not fight Git's defaults like using "--no-ff"

If you don’t understand the motivation behind Git’s design, you’re in for a world of hurt.

With enough flags you can force Git to act the way you think it should instead of the way it wants to. But that’s like using a screwdriver like a hammer;

it gets the job done, but it’s done poorly, takes longer, and damages the screwdriver.

[Git-Hilfen](http://weinimo.de/Git-Hilfen)

German only. Weiterführende Lösungsansätze und Tricks rund um Git.

[Git Home Documentation](http://git-scm.com/documentation)

The official Git documentation with a lot of references to tutorials and other helpful links.

Internal Links

Common CM-AI CF usecases and processes

Git @ Bosch

[CDG-SMT: Git](https://inside-docupedia.bosch.com/confluence/display/CDGSMT/Git) (Cross Divisional Group - Software, Methods, Tools / Engineering Base Software) [Embedded Linux Wiki: Git an introduction](https://inside-docupedia.bosch.com/confluence/display/Embreallin/Dojo%2B5%2B-%2BGIT%2Ban%2Bintroduction) has some presentation videos

[CF3: Git server](http://fe0vm142.de.bosch.com/cgi-bin/gitweb.cgi)

[SCMPlan\_Gen3.doc](https://hi-dms.de.bosch.com/docushare/dsweb/Services/Document-423673)

Bluepedia:Linux ADIT/CF3 buildtooling for TRITON