Git Extensions

U S E R M A N U A L

Henk Westhuis
Version 0.6

Index

1 GIT EXTENSIONS....4

1.1 Features....4
1.2 Video tutorials....4

2 GETTING STARTED....5

2.1 Install....5
2.2 Settings....6
2.3 Clone existing repository....8
2.4 Create new repository....9

3 BROWSE REPOSITORY....10

3.1 View commit log....10
3.2 Search history....11
3.3 Singe file history....12
3.4 Blame....13

4 COMMIT....14

4.1 Commit changes....14
4.2 Cherry pick commit....17
4.3 Revert commit....17
4.4 Stash changes....18

5 TAG....19

5.1 Create tag....19
5.2 Delete tag....19

6 BRANCHES....20

6.1 Create branch....20
6.2 Checkout branch....20
6.3 Merge branches....21
6.4 Rebase branch....22
6.5 Delete branch....23

7 PATCHES....24

7.1 Create patch....24
7.2 Apply patches....25

8 REMOTE FEATURES....26

8.1 Manage remote repositories....26
8.2 Create SSH key....27
8.2.1 PuTTY and github....27
8.2.2 OpenSSH and github....29
8.3 Pull changes....30
8.4 Push changes....33

9 MERGE CONFLICTS....34

9.1 Handle merge conflicts....34

10 SUBMODULES....36

10.1 Manage submodules....36
10.2 Add submodule....37

11 MAINTENANCE....38

11.1 Compress Git database....38
11.2 Verify Git database....38
11.3 Recover dangling objects....38

11.4 Fix user names....39
11.5 Ignore files....39
12 INTEGRATION....40
12.1 Visual Studio....40
12.2 Windows Explorer....42
13 COMMAND LINE....43
13.1 Git Extensions command line....43
GIT CHEAT SHEET....45

1 Git Extensions

Git Extensions is a toolkit aimed to make working with Git under Windows more intuitive. The shell extension will integrate in Windows Explorer and presents a context menu on files and directories. There is also a Visual Studio plug-in to use Git from Visual Studio. The source code of Git Extensions is located here: http://github.com/spdr870/gitextensions/tree/master

1.1 Features

- Windows Explorer integration for Git
- Visual Studio (2005/2008) plug-in for Git
- Feature rich user interface for Git
- Single installer installs Git, Git Extensions and the merge tool KDiff3
- 32bit and 64bit support

1.2 Video tutorials

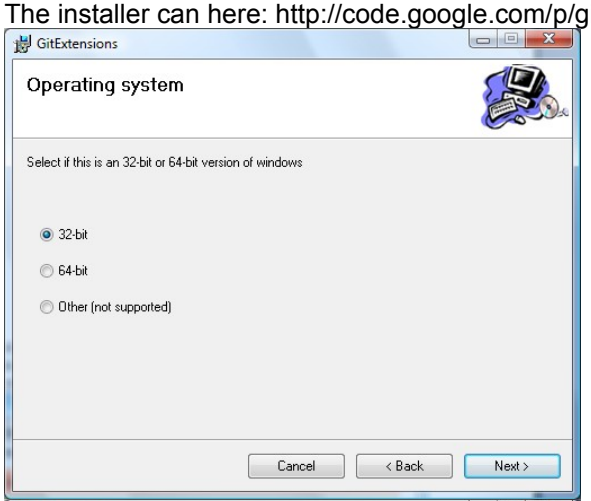
There are video tutorials for some basic functions on YouTube.

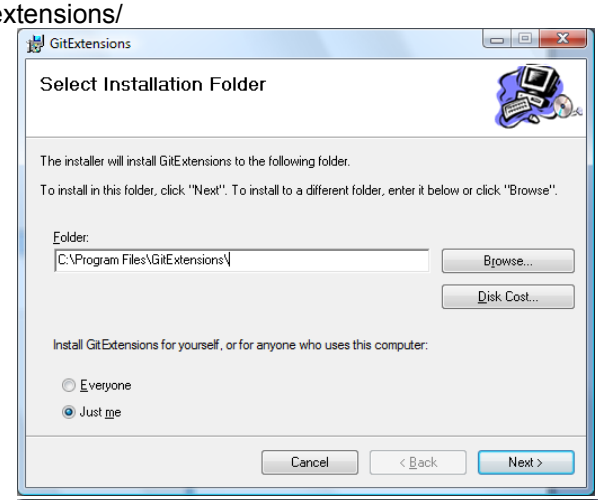
- 1 Clone - Git Extensions - http://www.youtube.com/watch?v=TlZXSkJGKF8
- 2 Commit changes - http://www.youtube.com/watch?v=B8uvje6X7lo
- 3 Push changes - http://www.youtube.com/watch?v=JByfXdbVAiE
- 4 Pull changes - http://www.youtube.com/watch?v=9g8gXPsi5Ko
- 5 Handle merge conflicts - http://www.youtube.com/watch?v=Kmc39RvuGM8

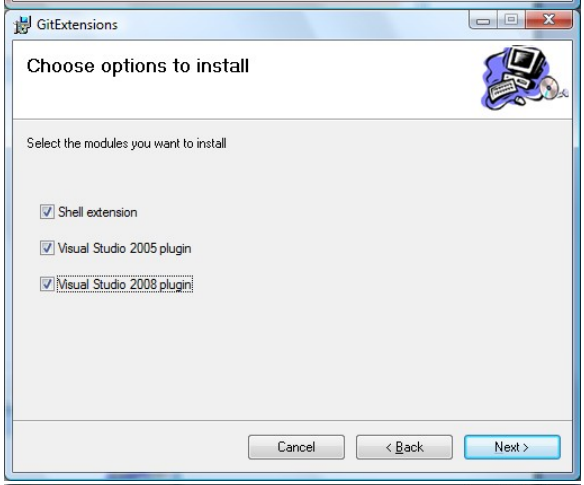
2 Getting started

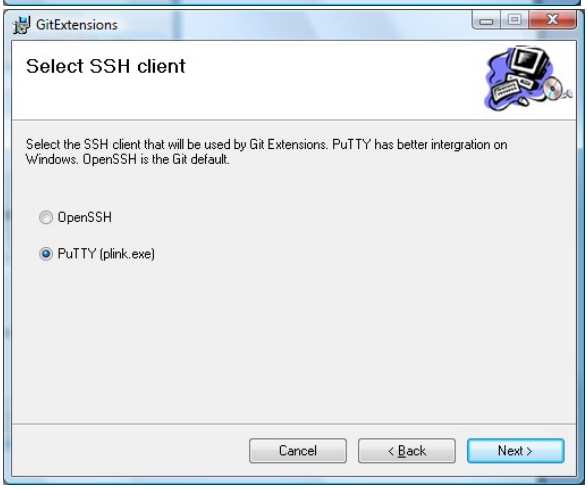
2.1 Install

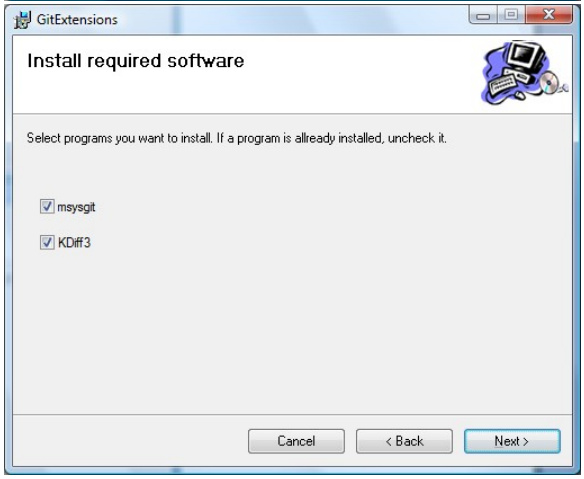
There is a single click installer that installs MSysGit, kdif3 and Git Extensions. You can choose to install the 32bit or 64bit version on the fist page of the installer.
The installer can here: http://code.google.com/p/gitextensions/





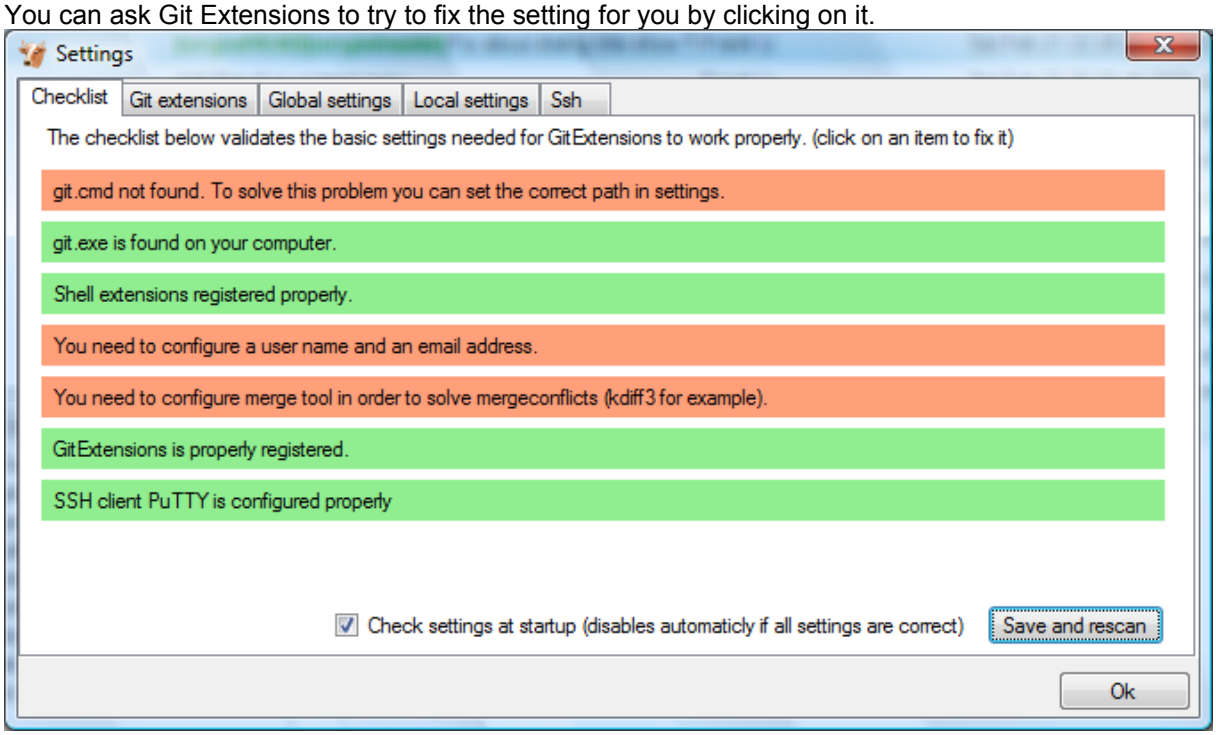






2.2 Settings

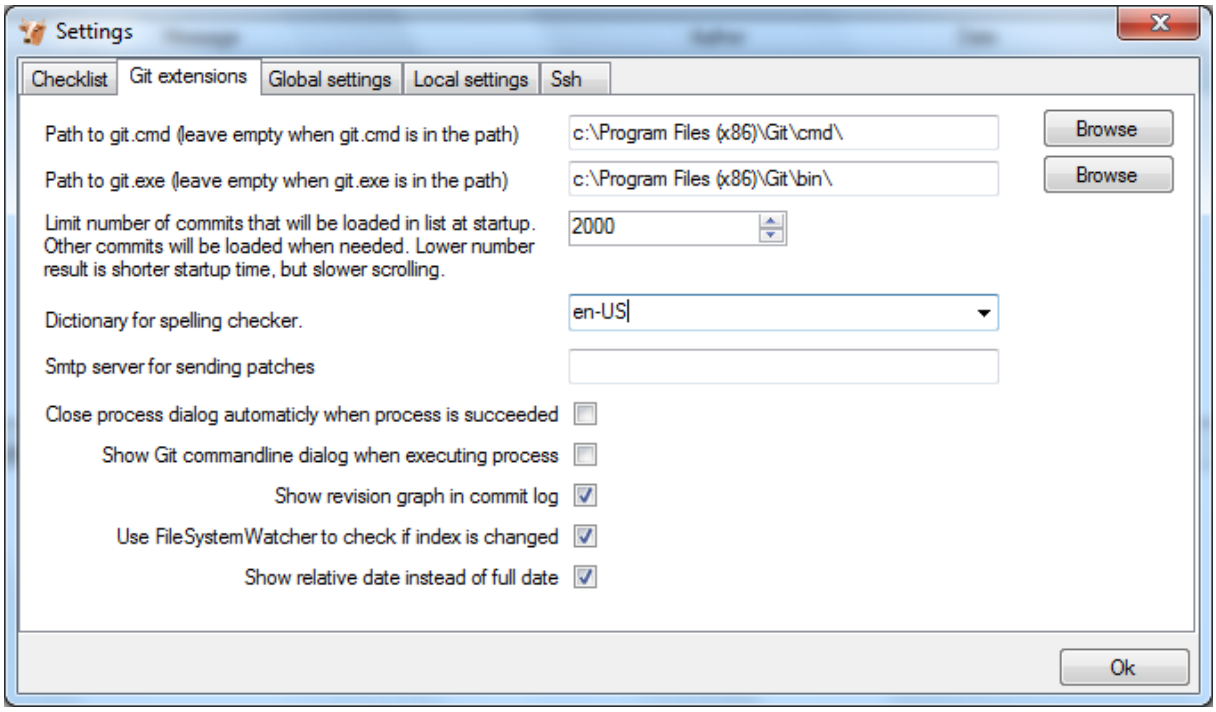
All settings will be verified when Git Extensions is started for the first time. If Git Extensions requires any settings to be changed the settings dialog will be shown. All incorrect settings will be marked red. You can ask Git Extensions to try to fix the setting for you by clicking on it.



All settings that are specific to Git Extensions will be stored in the Windows registry. The settings that are used by Git are stored in the configuration files of Git. The global settings are stored in a file called .gitconfig in the user directory. The local settings are stored in the .git\config file of the repository.

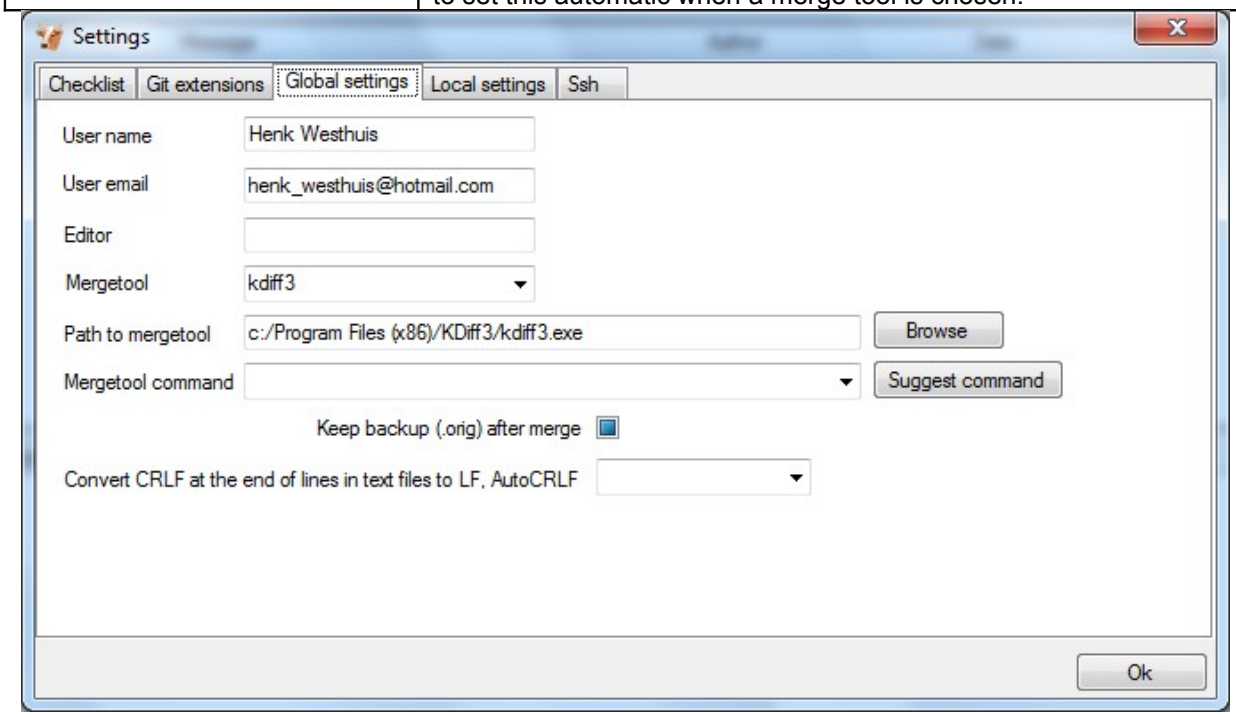
The 'Git Extension' tab contains all settings needed for Git Extension to run properly. The path to git.cmd and git.exe can be set here. This is only needed when these are not in the system path.

Path to git.cmd	Needed for Git Extensions to run Git commands
Path to git.exe	Only needed for a few optimized commands
Limit number of commits that will be loaded in list at start-up.	Git Extensions uses lazy loading to load the commit log. Lower this number to increase the start-up speed. Increase the number for faster scrolling. Turn of revision graph for optimal result!
Show revision graph in commit log.	Turn revision graph in commit log on/off.
Show Git command line dialog when executing process.	Turn this option on if you want to see the Git command line dialog when a process is executed.
Dictionary for spelling checker	Choose the dictionary to use for the spelling checker in the commit dialog.
Smtp server for sending patches	Smtp server to use for sending patches
Use FileSystemWatcher to check if index is changed	Using the FileSystemWatcher to check index state improves the performance is some cases. Turn this off if you experience refresh problems in commit log.
Show relative date instead of full date.	Show relative date, e.g. 2 weeks ago, instead of full date.

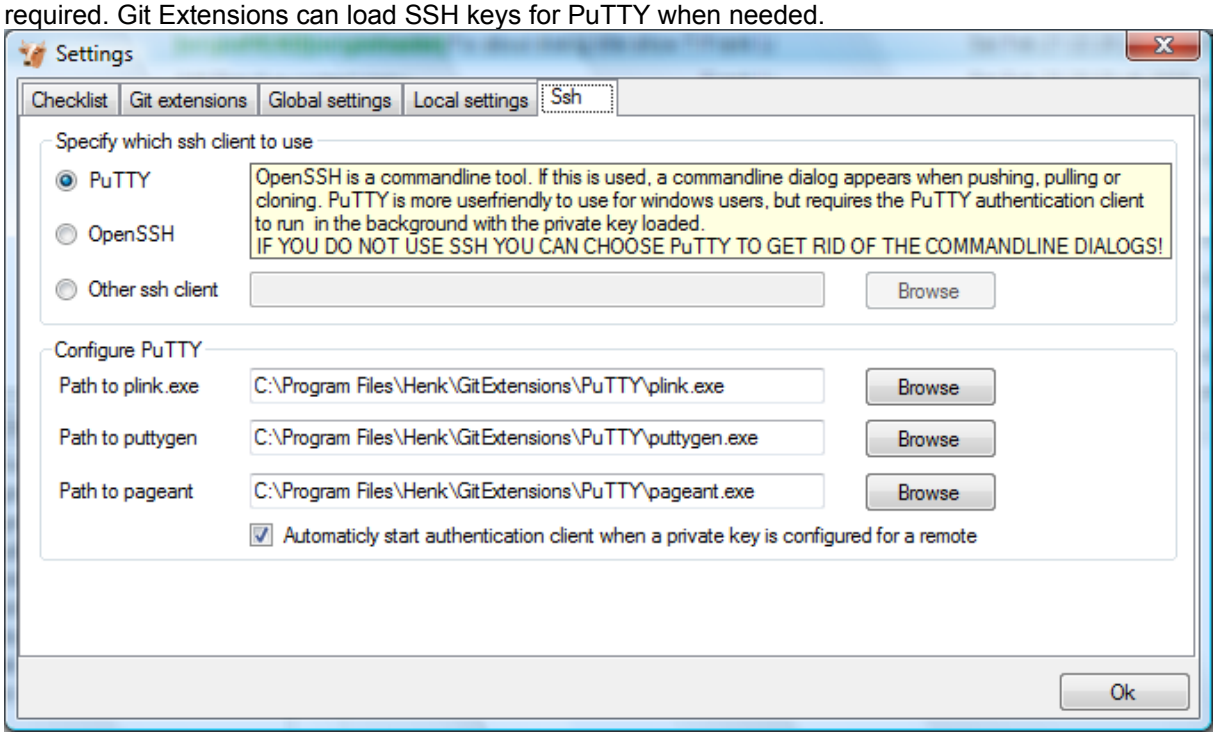


In the 'Global settings' tab some global Git settings can be set.

User name	User name shown in commits and patches
User email	User email shown in commits and patches
Editor	Editor that git.exe opens (e.g. for editing commit message). This is not used by Git Extensions, only when you call git.exe from the command line. By default Git will use the command line text editor vi.
Mergetool	Merge tool used to solve merge conflicts. Git Extensions will search for common merge tools on your system.
Path to mergetool	Path to merge tool. Git Extensions will search for common merge tools on your system.
Mergetool command	Command that Git uses to call the merge tool. Git Extensions will try to set this automatic when a merge tool is chosen.

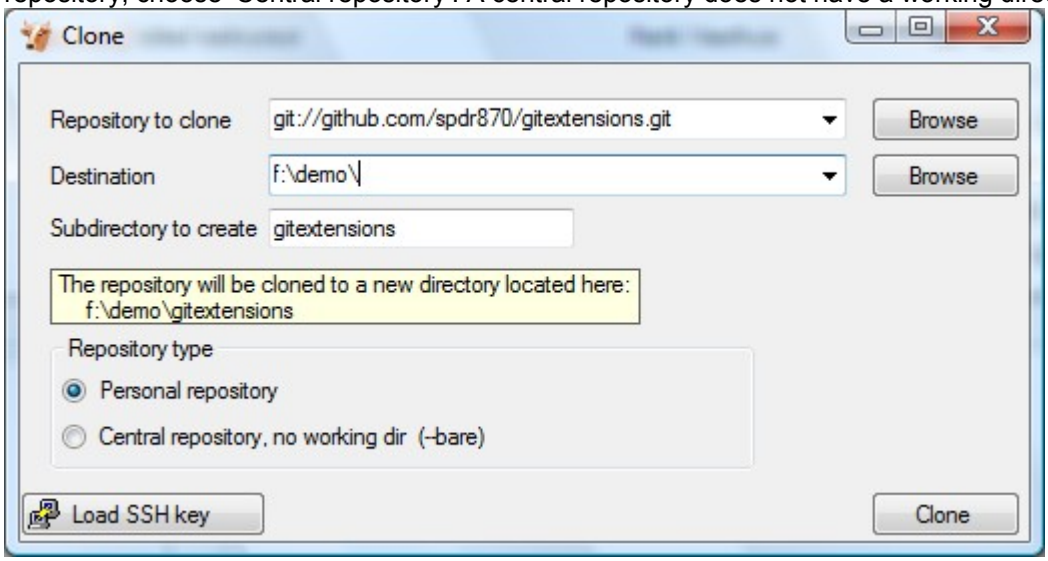


In the tab ‘SSH’ you can configure the SSH client you want Git to use. Git Extensions is optimized for PuTTY. Git Extensions will show command line dialogs if you do not use PuTTY and user input is required. Git Extensions can load SSH keys for PuTTY when needed.



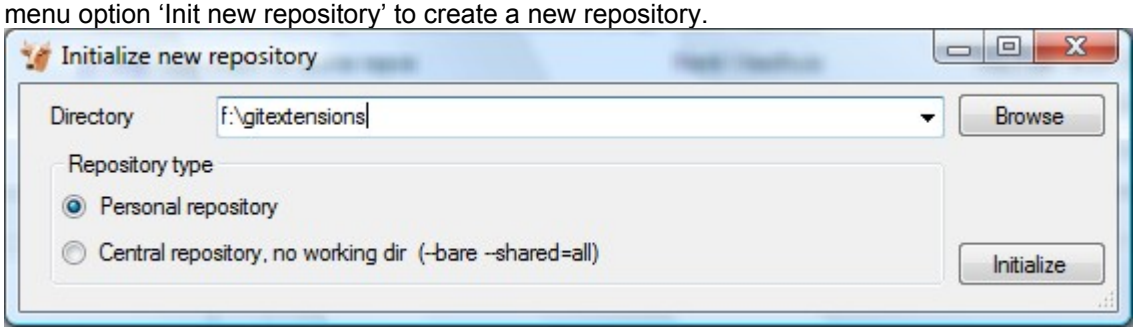
2.3 Clone existing repository

You can clone an existing repository using the ‘Clone’ menu option. You can choose the repository type to clone to. For personal use you need to choose ‘Personal repository’. For a central or public repository, choose ‘Central repository’. A central repository does not have a working directory.



2.4 Create new repository

When you do not want to work on an existing project, you can create your own repository. Choose the menu option ‘Init new repository’ to create a new repository.



You can choose to create a Personal repository or a Central repository. A personal repository looks the same as a normal working directory but has a directory named .git on root level containing the version history. This is the most common repository.

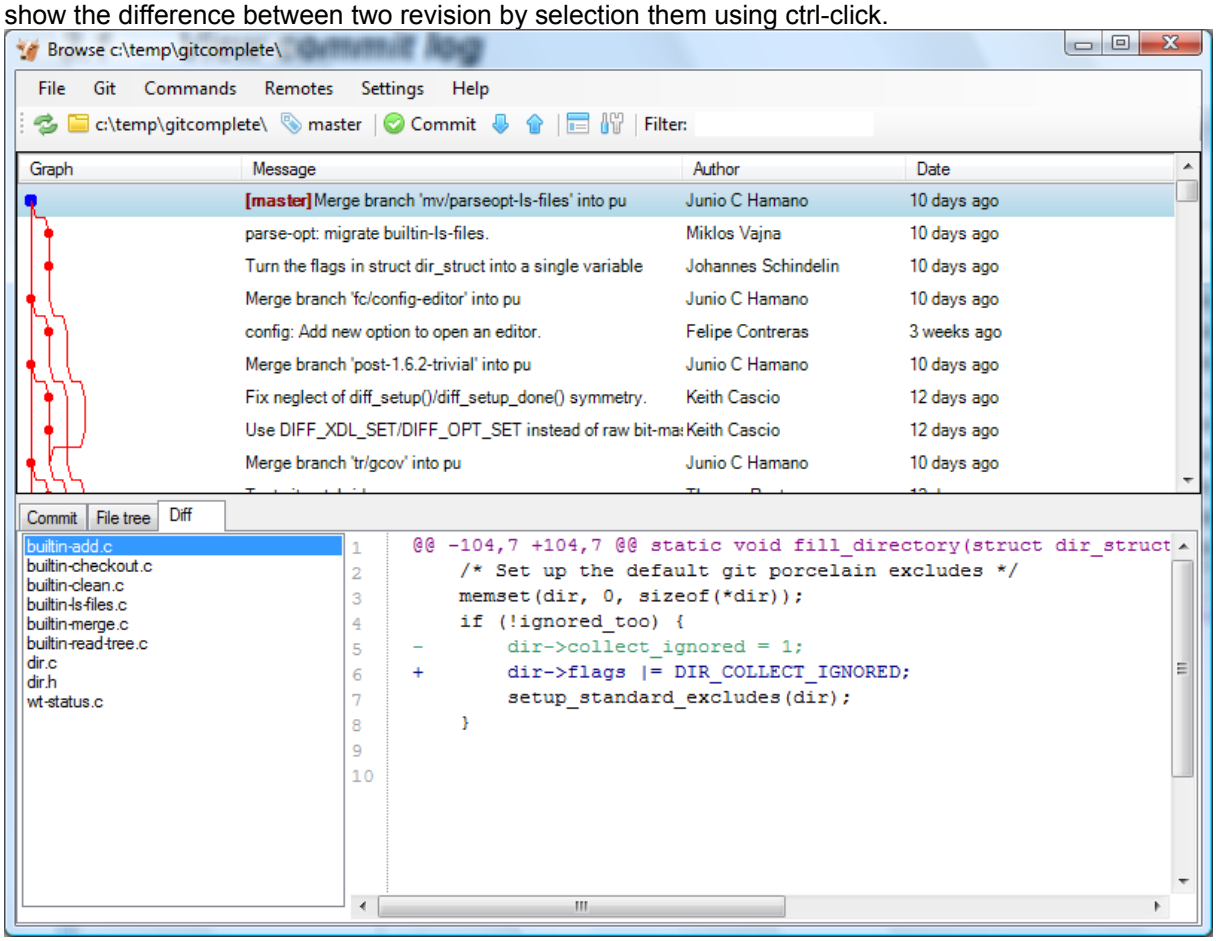
Central repositories only contain the version history. Because a central repository has no working directory you cannot checkout a revision in a central repository. It is also impossible to merge or pull changes is a central repository. This repository type can be used as a public repository where developers can push changes to or pull changes from.

3 Browse repository

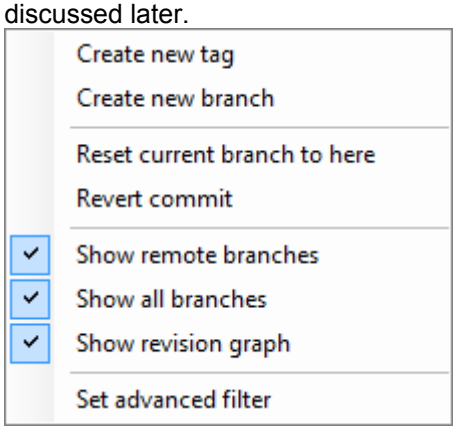
You can browse a repository by starting Git Extensions and select the repository to open. The main window contains the commit log. You could also open the ‘Browse’ window from the shell extensions and from the Visual Studio IDE.

3.1 View commit log

The full commit history can be browsed. There is a graph that shows branches and merges. You can show the difference between two revision by selection them using ctrl-click.

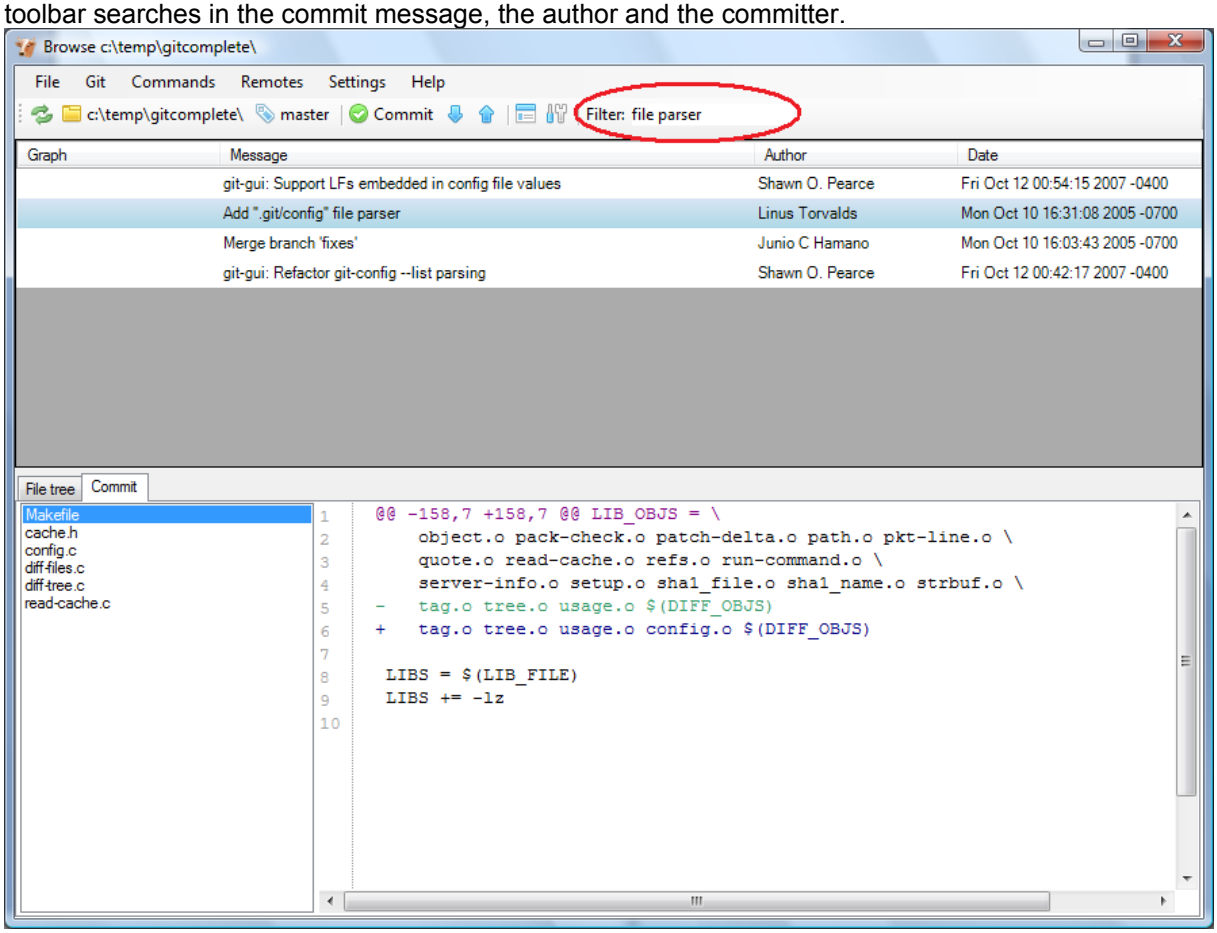


In the context menu of the commit log you can enable or disable the revision graph. You can also choose to only show the current branch instead of showing all branches. The other options will be discussed later.

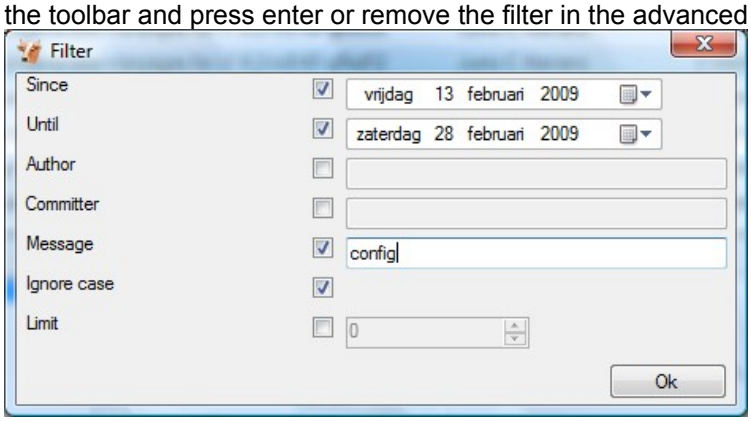


3.2 Search history

The history can be searched using regular expressions are basic search terms. The quick filter in the toolbar searches in the commit message, the author and the committer.

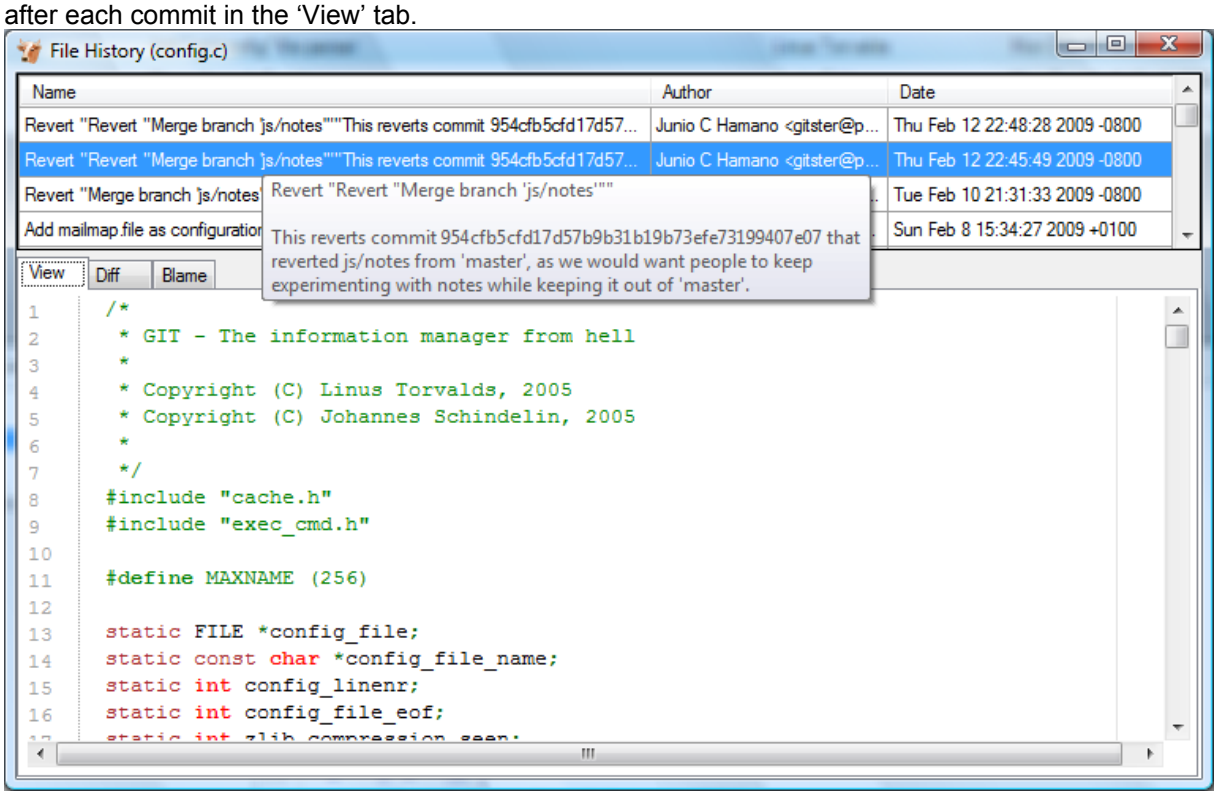


In the context menu of the commit log you can open the advanced filter dialog. The advanced filter dialog allows you to search for more specific commits. To remove the filter either remove the filter in the toolbar and press enter or remove the filter in the advanced filter dialog.

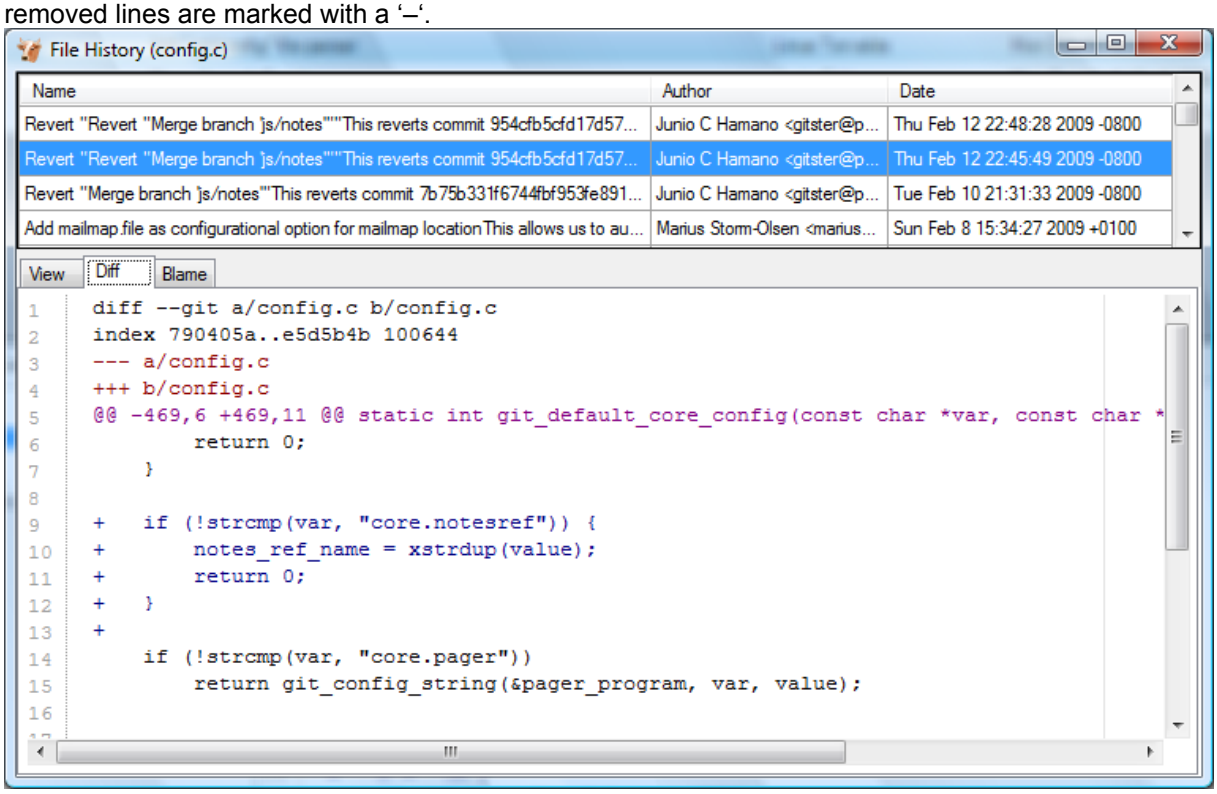


3.3 Singe file history

The single file history viewer shows all revisions of a single file. You can view the content of the file in after each commit in the ‘View’ tab.

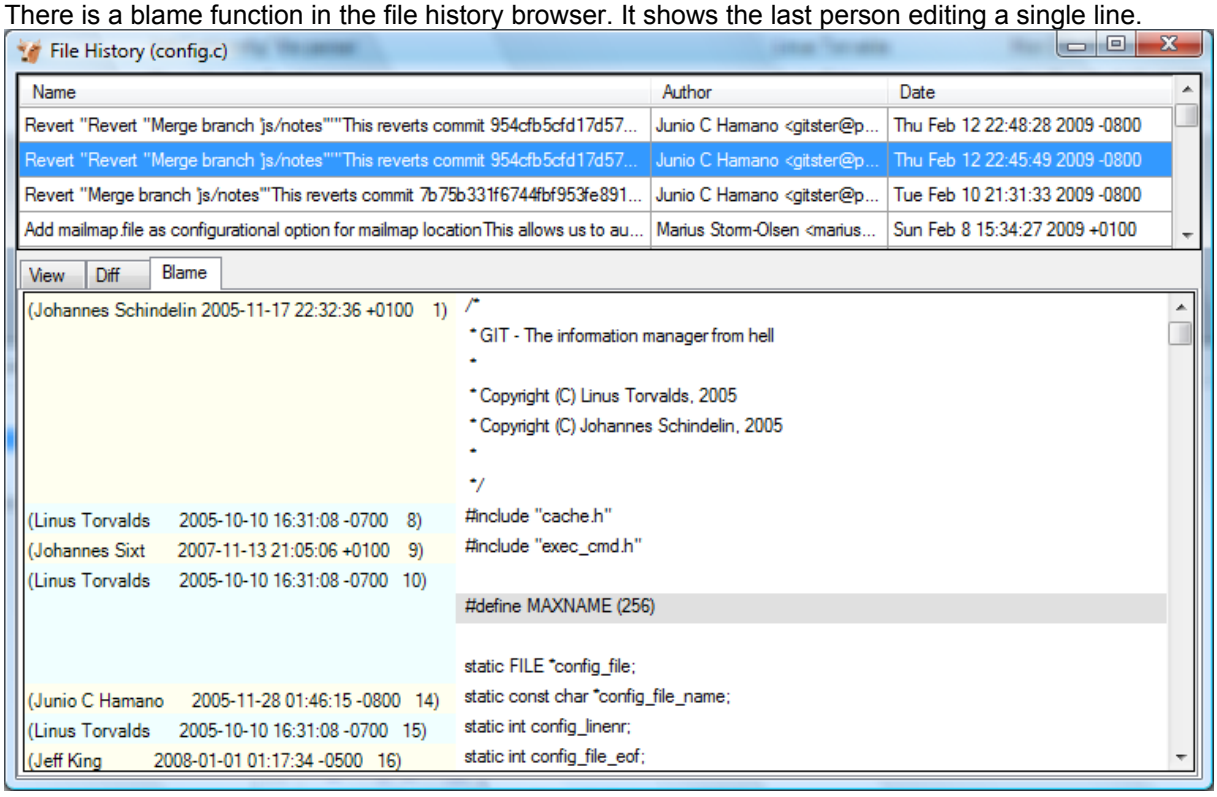


You can view the difference report from the commit in the ‘Diff’ tab. Added lines are marked with a ‘+’, removed lines are marked with a ‘–‘.

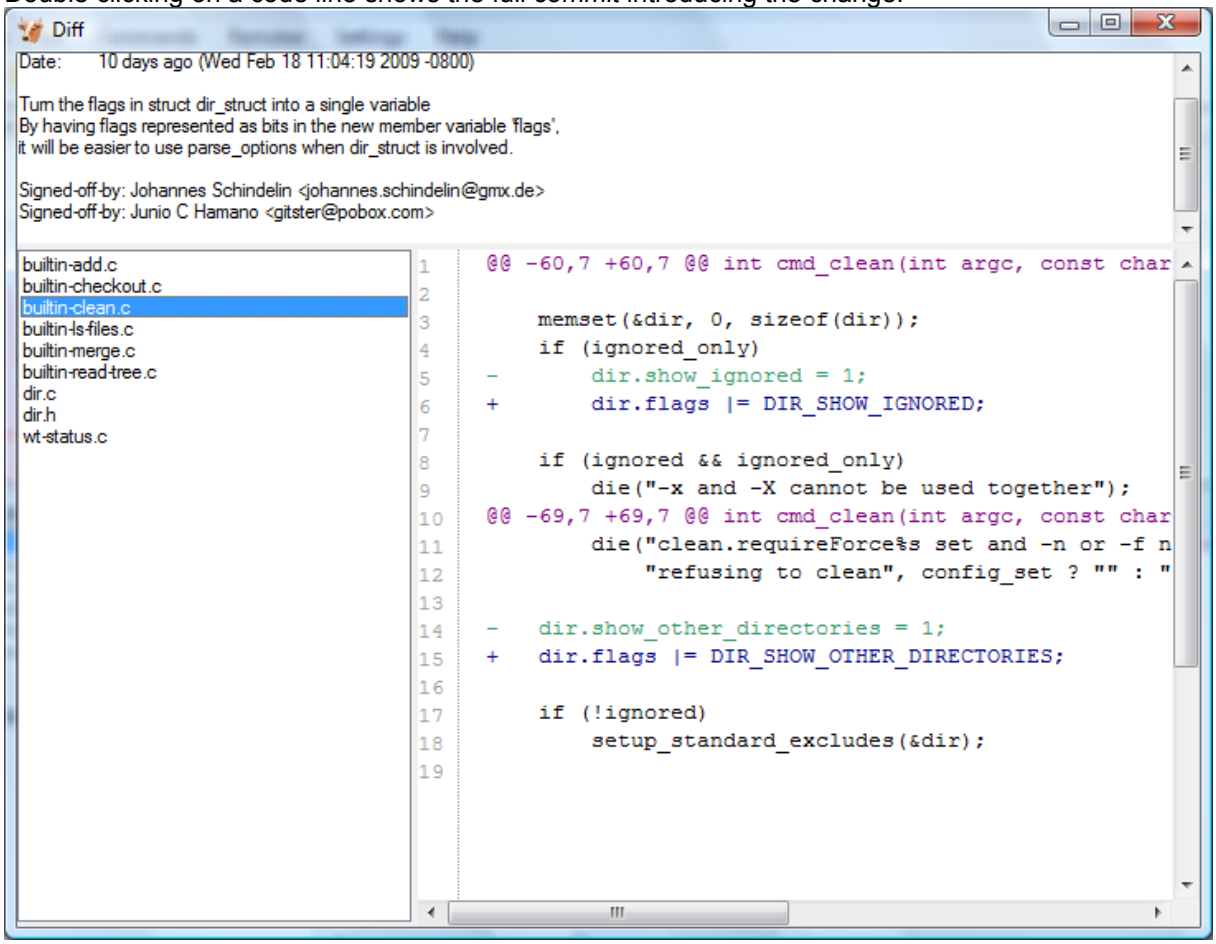


3.4 Blame

There is a blame function in the file history browser. It shows the last person editing a single line.



Double clicking on a code line shows the full commit introducing the change.



4 Commit

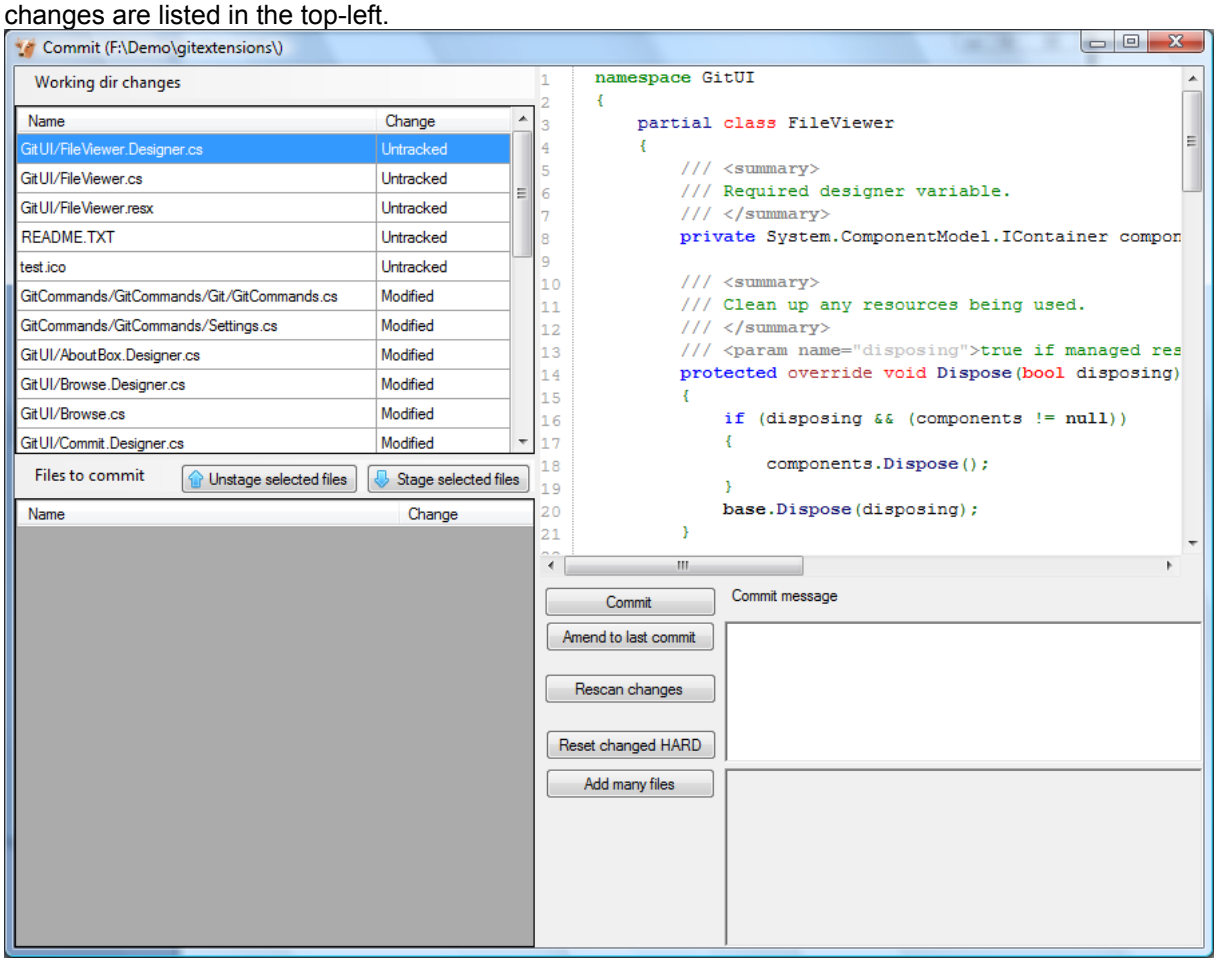
A commit is a set of changes with some extra information. Every commit contains the follow information:

- Changes
- Committer name and email
- Commit date
- Commit message
- Cryptographically strong SHA1 hash

Each commit creates a new revision of the source. Revisions are not tracked per file; each change creates a new revision of the complete source. Unlike most traditional source control management systems, revisions are not named using a revision number. Each revision is named using a SHA1, a 41 long characters cryptographically strong hash.

4.1 Commit changes

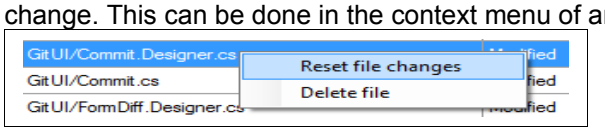
Changes can be committed to the local repository. Unlike most other source control management systems you do not need to checkout files before you start editing. You can just start editing files, and review all the changes you made in the commit dialog later. When you open de commit dialog, all changes are listed in the top-left.



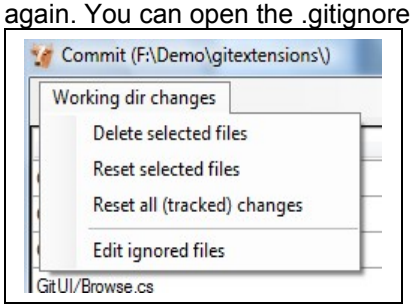
There are three kinds of changes:

Untracked	This file is not yet tracked by Git. This is probably a new file, or a file that has not been committed to Git before.
Modified	This file is modified since the last commit.
Deleted	This file has been deleted.

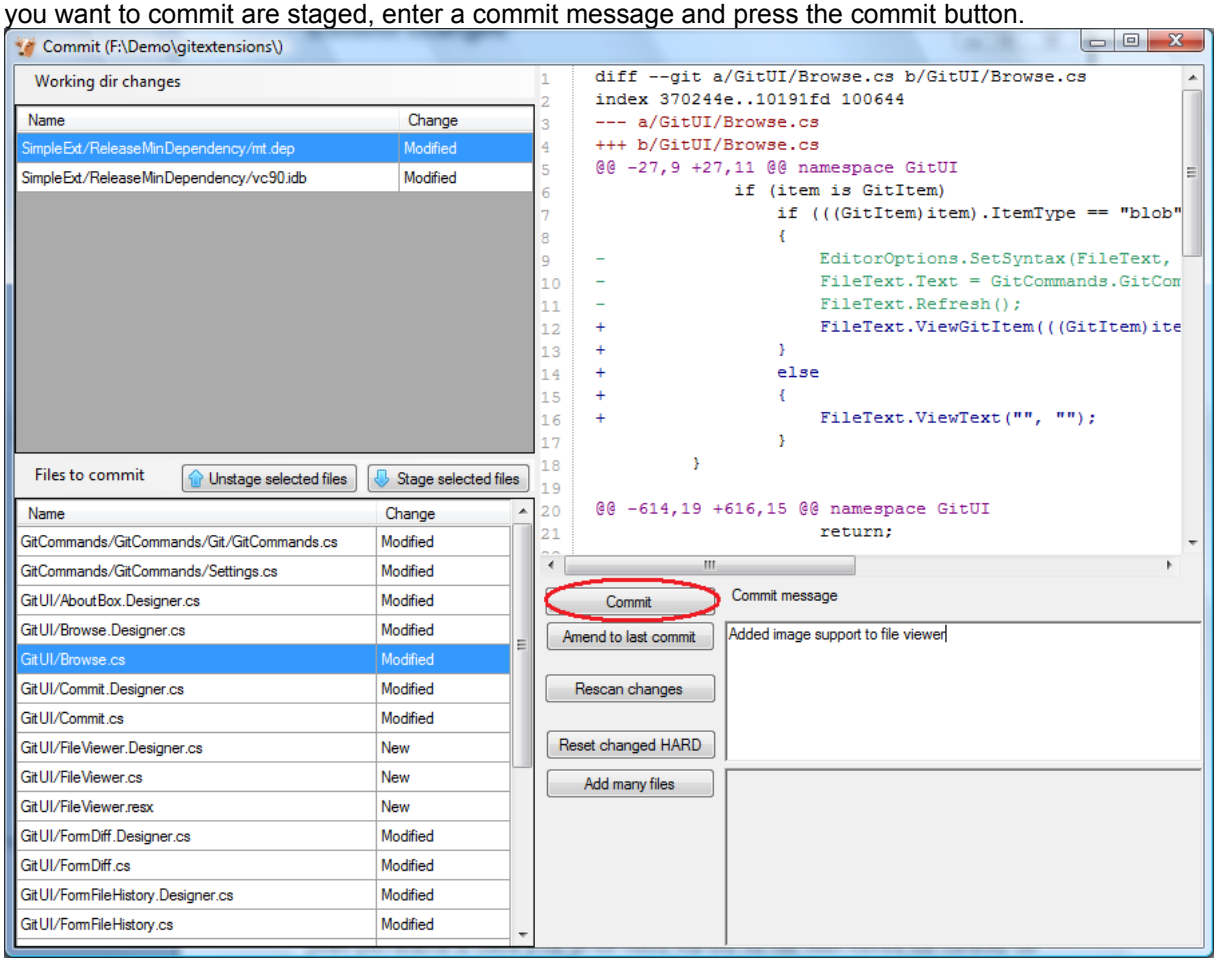
When you rename or move a file Git will notice that this file has been moved, but currently Git Extensions does not show this in the commit dialog. Occasionally you will need to undo the file change. This can be done in the context menu of any unstaged file.



During your initial commit there are probably lots of files you do not want to be tracked. You can ignore these files by not staging them, but they will show every time. You could also add them to the .gitignore file of you repository. Files that are in the .gitignore file will not show up in the commit dialog again. You can open the .gitignore editor from the menu 'Working dir changes'.

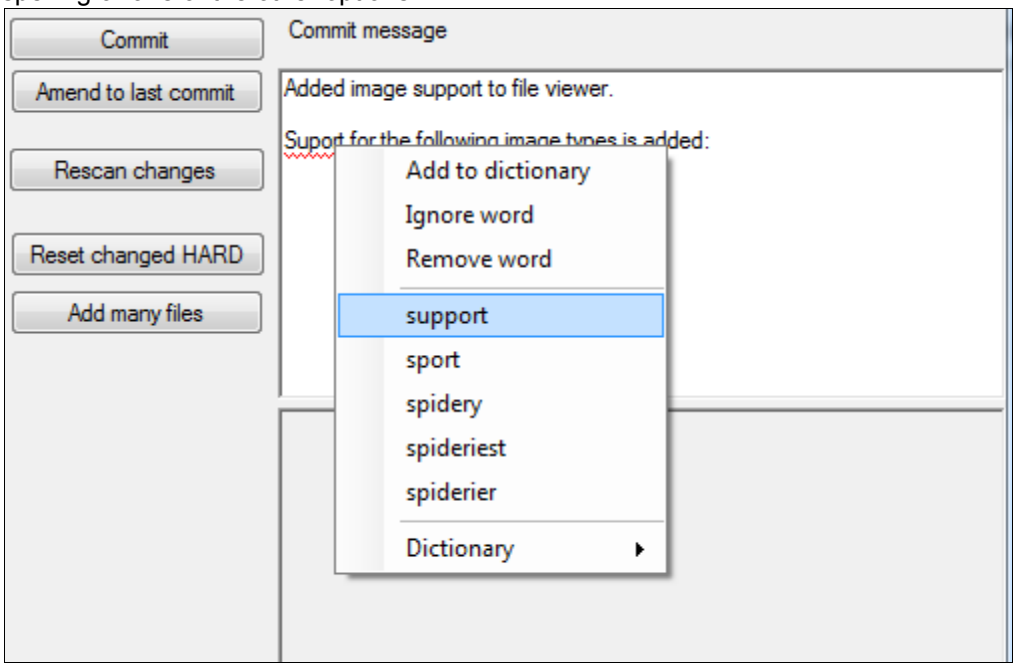


You need to stage the changes you want to commit by pressing the 'Stage selected files' button. You also need to stage deleted files because you stage the change and not the file. When all the changes you want to commit are staged, enter a commit message and press the commit button.

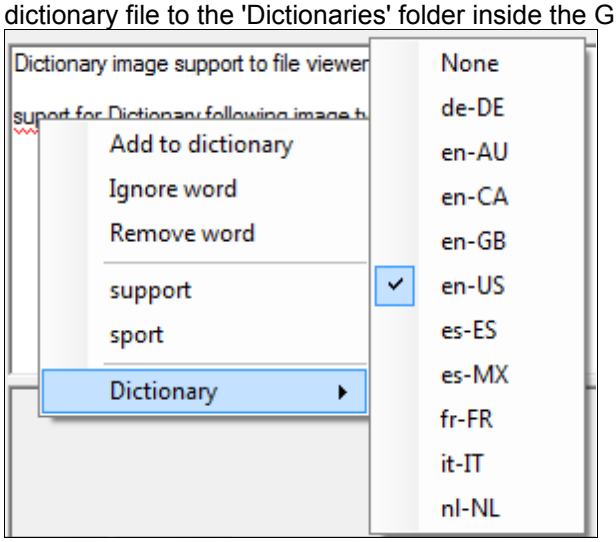


It is also possible to add files to you last commit using the 'Amend to last commit' button. This can be very useful when you forgot some changes. This function rewrites history; it deletes the last commit and commits it again including the added changes. Make sure you only use 'Amend to last commit' when the commit is not yet published to other developers.

There is a build in spelling checker that checks the commit message. Incorrect spelled words are underlined with a red wave line. By right-clicking on the misspelled word you can choose the correct spelling or one of the other options.

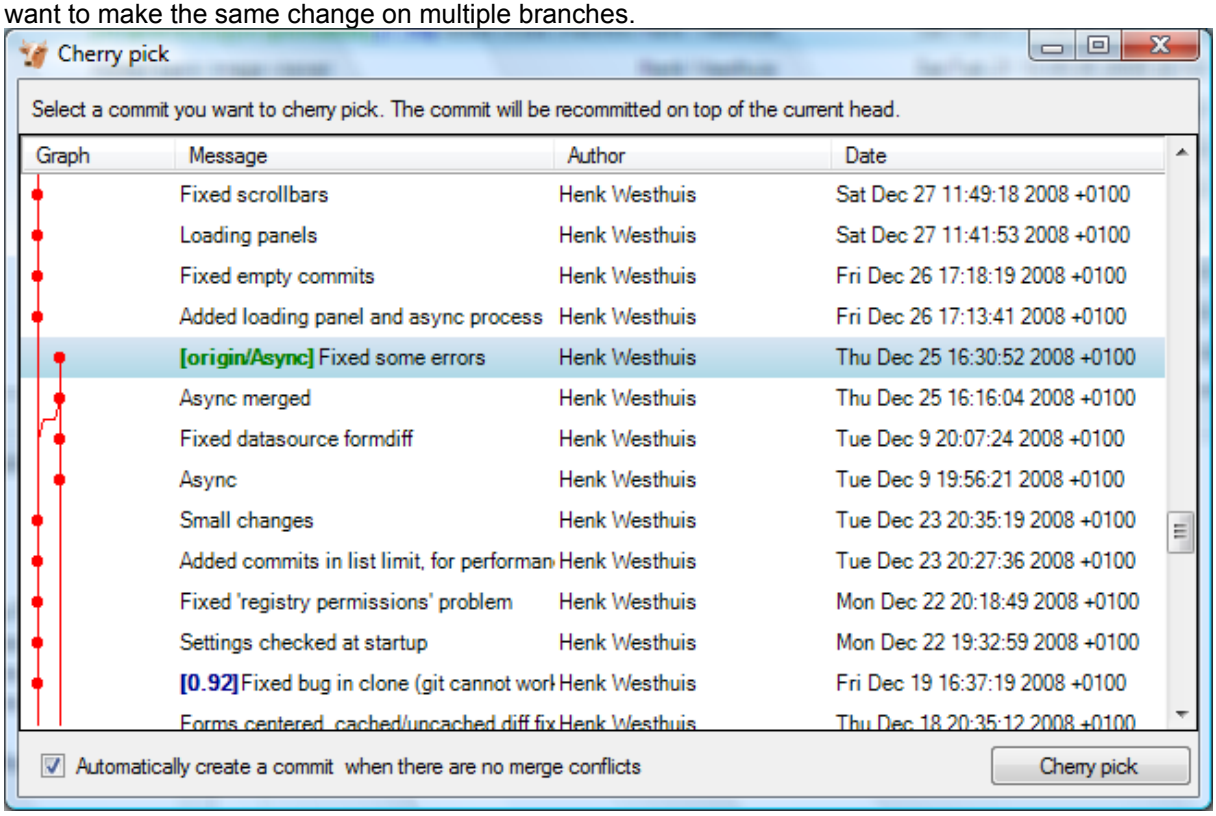


Git Extensions installs a number of dictionaries by default. You can choose another language in the context menu of the spelling checker or in the settings dialog. To add a new spelling dictionary add the dictionary file to the 'Dictionaries' folder inside the Git Extensions installation folder.



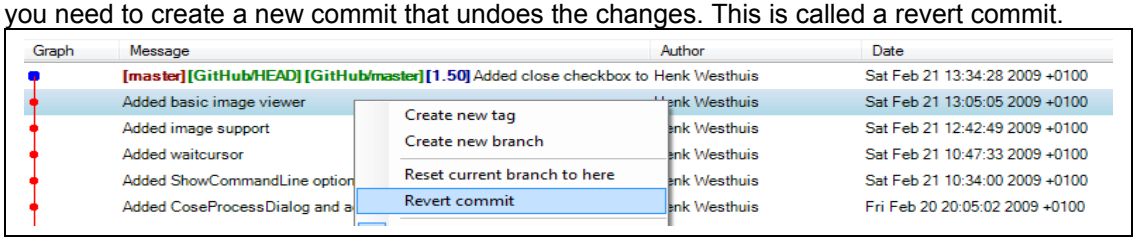
4.2 Cherry pick commit

A commit can be recommitted by using the cherry pick function. This can be very useful when you want to make the same change on multiple branches.



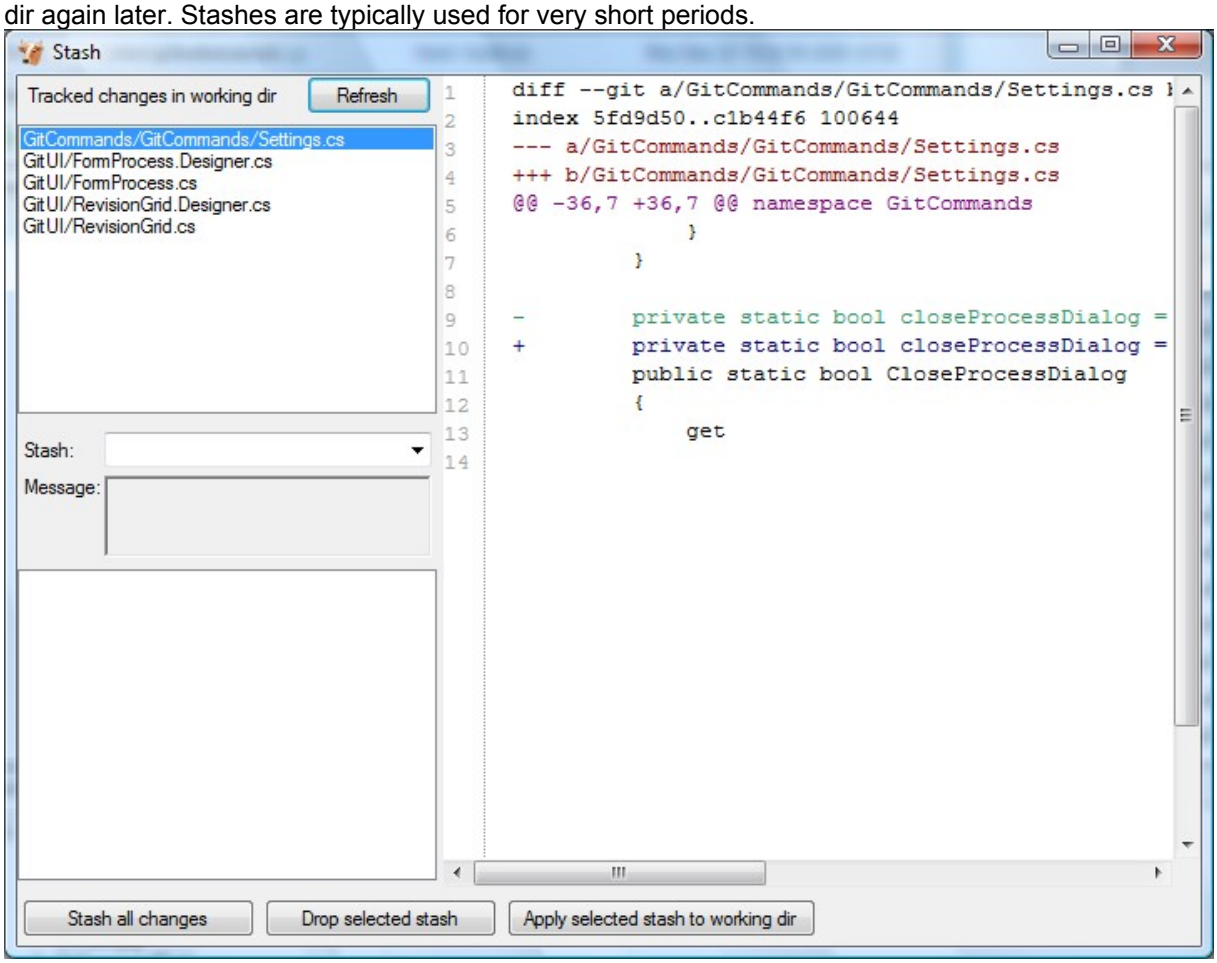
4.3 Revert commit

A commit cannot be deleted once it is published. If you need to undo the changes made in a commit, you need to create a new commit that undoes the changes. This is called a revert commit.

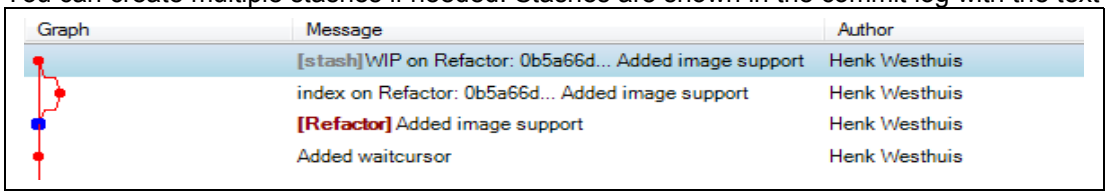


4.4 Stash changes

If there are local changes that you do not want to commit yet and not want to throw away either, you can temporarily stash them. This is useful when working on a feature and you need to start working on something else for a few hours. You can stash changes away and then reapply them to your working dir again later. Stashes are typically used for very short periods.



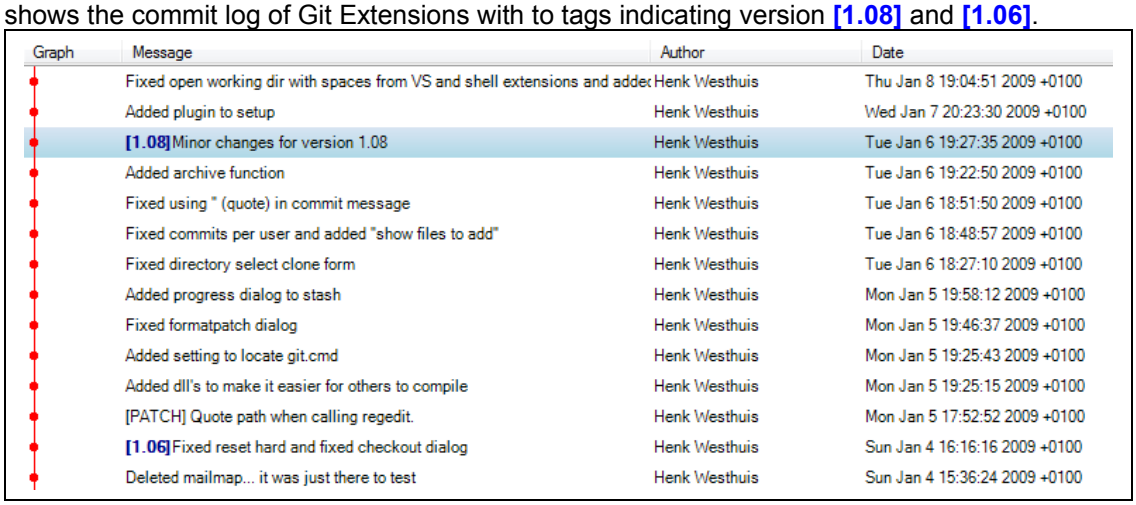
You can create multiple stashes if needed. Stashes are shown in the commit log with the text **[stash]**.



The stash is especially useful when pulling remote changes into a dirty working directory. If you want a more permanent stash, you should create a branch.

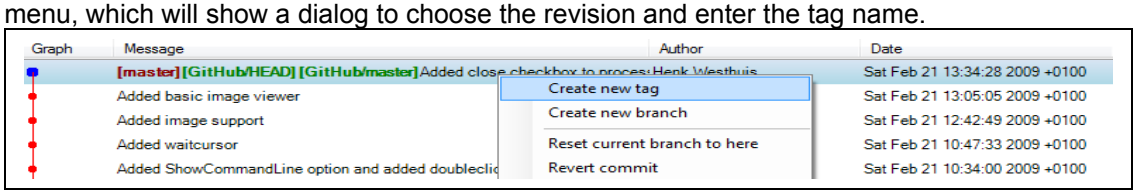
5 Tag

Tags are used to mark a specific version. Usually a tag will not be moved anymore. The image below shows the commit log of Git Extensions with to tags indicating version **[1.08]** and **[1.06]**.



5.1 Create tag

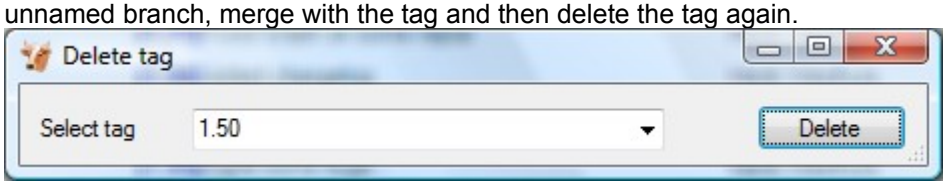
In Git Extensions you can tag a revision by choosing 'Create new tag' in the commit log context menu. A dialog will prompt for the name of the tag. You can also choose 'Create tag' from the 'Commands' menu, which will show a dialog to choose the revision and enter the tag name.



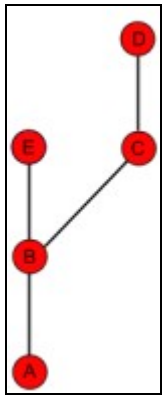
Once a tag is created, it cannot be moved again. You need to delete the tag and create it again to move it.

5.2 Delete tag

For some operation it is very useful to create tags for temporary usage. Git uses SHA1 hashes to name each commit. When you want to merge with an unnamed branch it is good practise to tag the unnamed branch, merge with the tag and then delete the tag again.



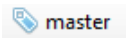
6 Branches



Branches are used to commit changes separate from other commits. It is very common to create a branch when you start working on a feature and you are not sure if this feature will be finished in time for the next release. The image on the right illustrates a branch created on top of commit B.

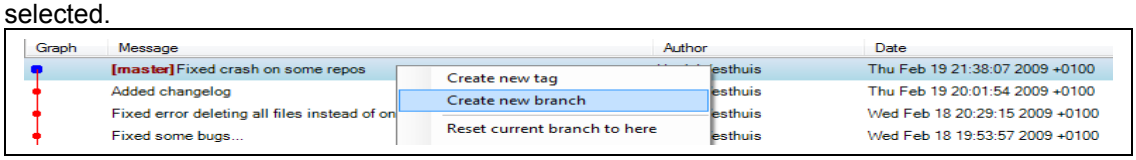
In Git branches are created very often. Creating a branch is very easy to do and it is recommended to create a branch very often. In fact, when you make a commit to a cloned repository you start a new branch. I will explain this in the pull chapter.

You can check on what branch you are working in the toolbar.

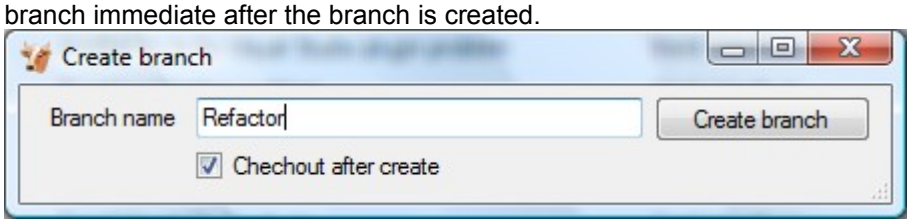


6.1 Create branch

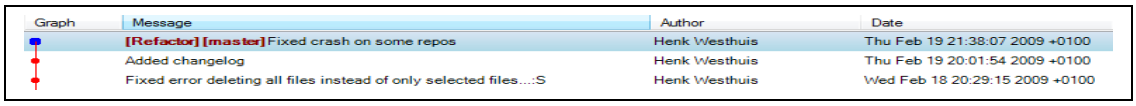
In Git Extensions there are multiple ways to create a new branch. In the image below I create a new branch from the context menu in the commit log. This will create a new branch on the revision that is selected.



I will create a new branch called ‘Refactor’. In this branch I can do whatever I want without considering others. In the ‘Create branch’ dialog there is a checkbox you can check if you want to checkout this branch immediate after the branch is created.



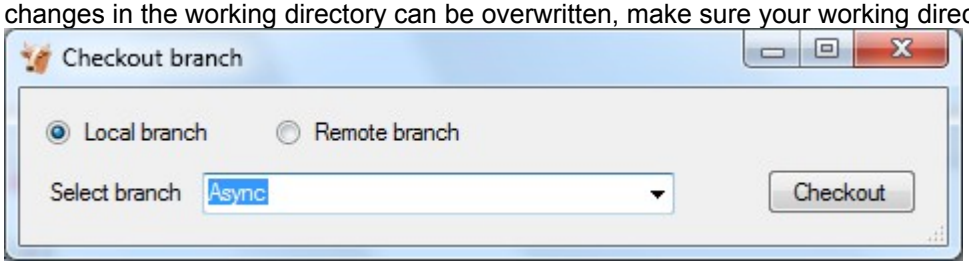
When the branch is created you will see the new branch **[Refactor]** in the commit log. If you chose to checkout this branch the next commit will be committed to the new branch.



Creating branches in Git requires only 41 bytes of space in the repository. Creating a new branch is very easy and is very fast. The complete work flow of Git is optimized for branching and merging.

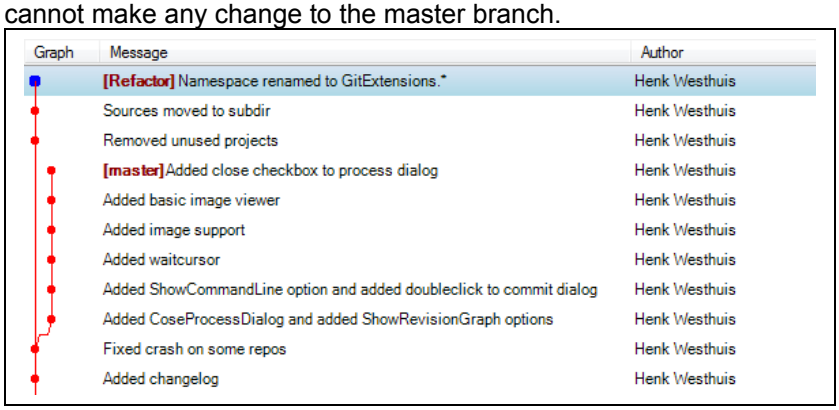
6.2 Checkout branch

You can switch from the current branch to another branch using the checkout command. Checkout a branch sets the current branch and updates all sources in the working directory. Uncommitted changes in the working directory can be overwritten, make sure your working directory is clean.

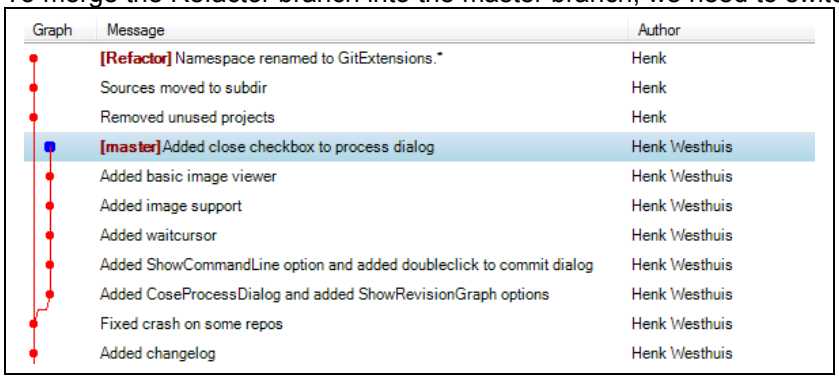


6.3 *Merge branches*

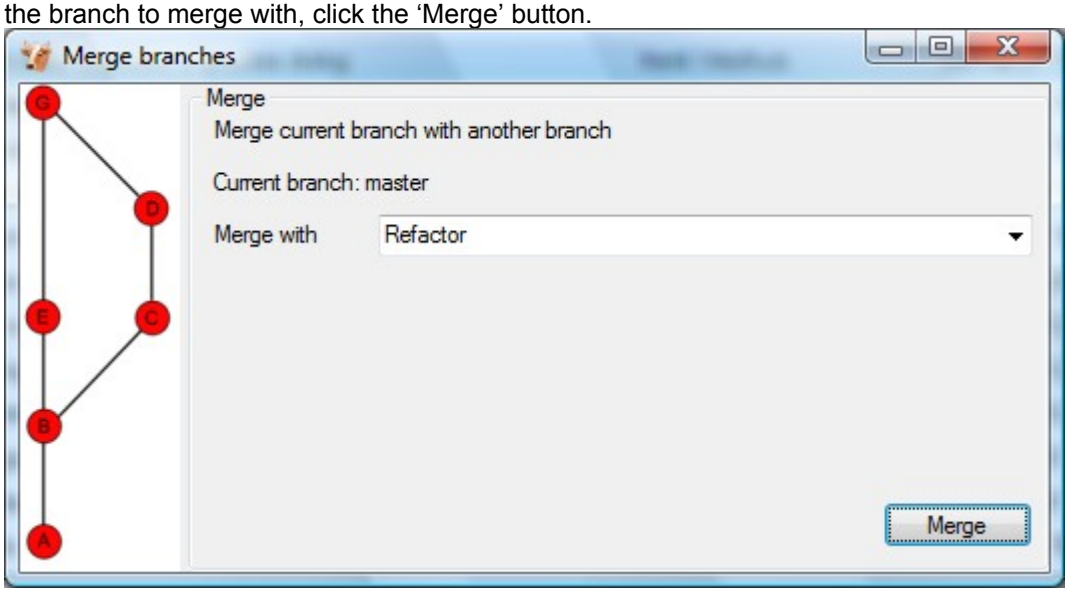
In the image below there are two branches, **[Refactor]** and **[master]**. We can merge the commits from the master branch into the Refactor. If we do this, the Refactor branch will be up to date with the master branch, but not the other way around. As long as we are working on the Refactor branch we cannot tough the master branch itself. We can merge the sources of master into our branch, but cannot make any change to the master branch.



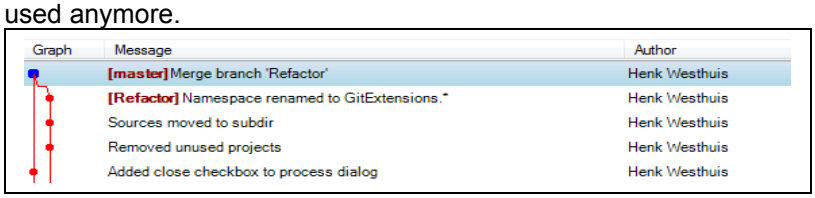
To merge the Refactor branch into the master branch, we need to switch to the master branch first.



Once we are on the master branch we can choose merge by choosing ‘Merge branches’ from the ‘Commands’ menu. In the merge dialog you can check the branch you are working on. After selected the branch to merge with, click the ‘Merge’ button.



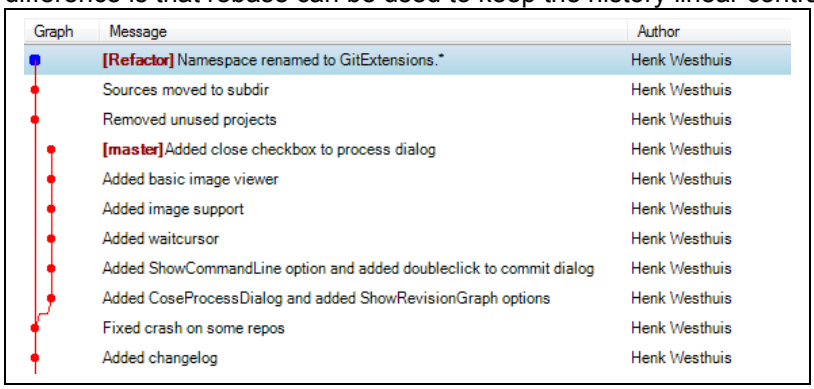
After the merge the commit log will show the new commit containing the merge. Notice that the Refactor branch is not changed by this merge. If you want to continue working on the Refactor branch you can merge the Refactor branch with master. You could also delete the Refactor branch if it is not used anymore.



When you need to merge with on unnamed branch you can use a tag to give it a temporary name.

6.4 Rebase branch

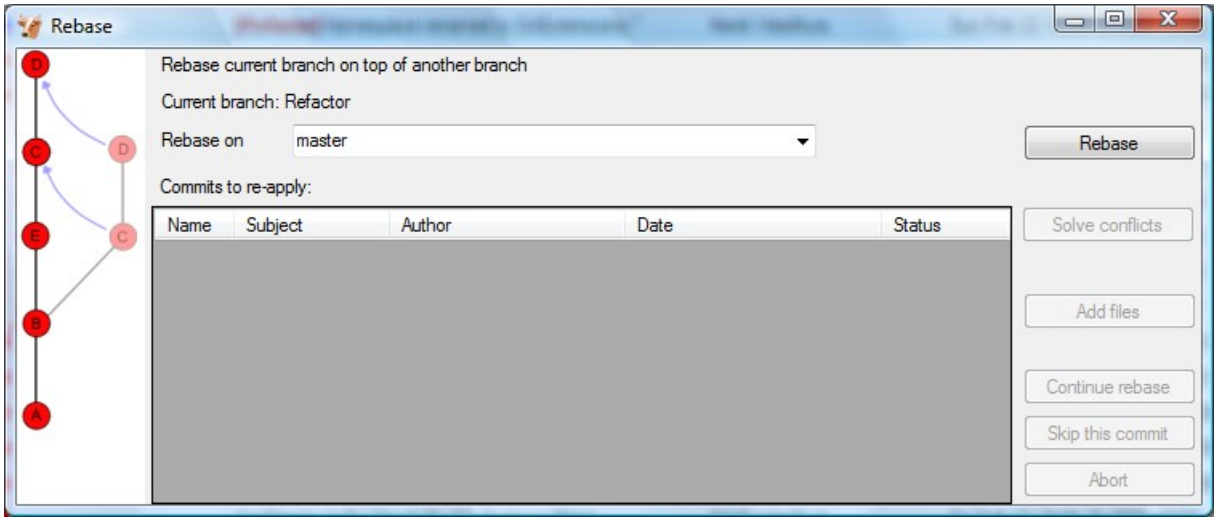
The rebase command is the most complex command in Git. The rebase command is very similar to the merge command. Both rebase and merge are used to get a branch up-to-date. The main difference is that rebase can be used to keep the history linear contrary to merges.



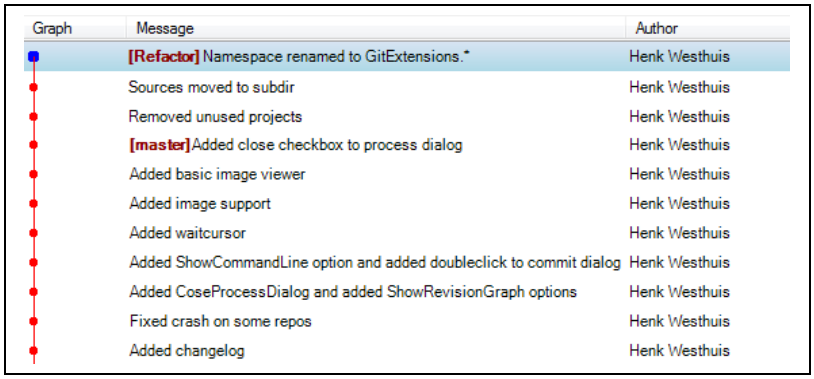
A rebase of Refactor on top of master will perform the following actions:

- All commits specific to the Refactor branch will be stashed in a temporary location
- The branch Refactor will be removed
- The branch Refactor will be recreated on the master branch
- All commits will be recommitted in the new Refactor branch

During a rebase merge conflicts can occur. You need to solve the merge conflicts for each commit that is rebased. The rebase function in Git Extensions will guide you through all steps needed for a successful rebase.



The image below shows the commit log after the rebase. Notice that the history is changed and is seems like the commits on the Refactor branch are created after the commits on the master branch.

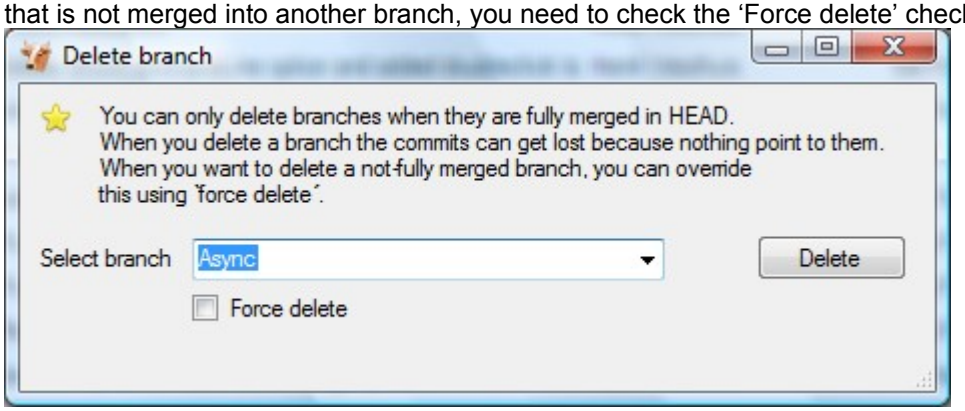


Because this function rewrites history you should only use this on branches that are not published to other repositories yet. When you rebase a branch that is already pushed it will be harder to pull or push to that remote. If you want to get a branch up-to-date that is already published you should merge.

6.5 Delete branch

It is very common to create a lot of branches. You can delete branches when they are not needed anymore and you do not want to keep the work done in that branch. When you delete a branch that is not yet merged, all commits will be lost. When you delete a branch that is already merged with another branch, the merged commits will not be lost because they are also part of another branch.

You can delete a branch using ‘Delete branch’ in ‘Commands’ menu. If you want to delete a branch that is not merged into another branch, you need to check the ‘Force delete’ checkbox.



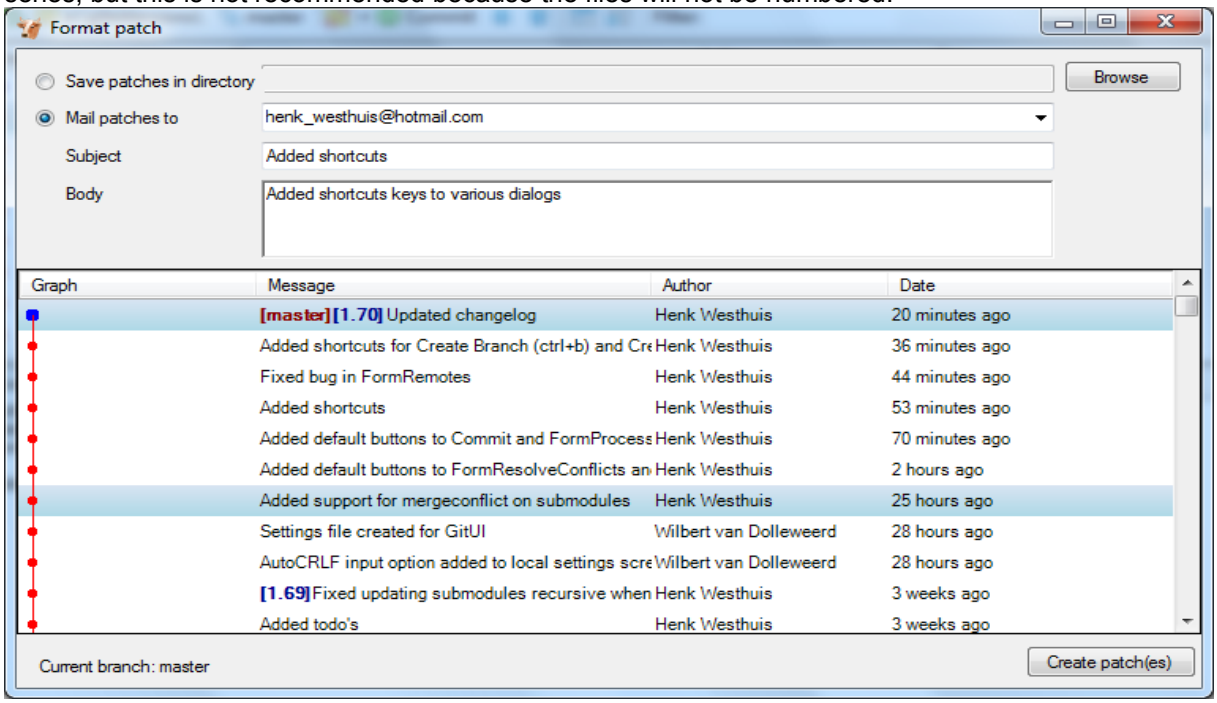
7 Patches

Every commit contains a change-set, a commit date, the committer name, the commit message and a cryptograph SHA1 hash. Local commits can be published by pushing it to a remote repository. To be able to push you need to have sufficient rights and you need to have access to the remote repository. When you cannot push directly you can create patches. Patches can be e-mailed to someone with access to the repository. Each patch contains an entire commit including the commit message and the SHA1.

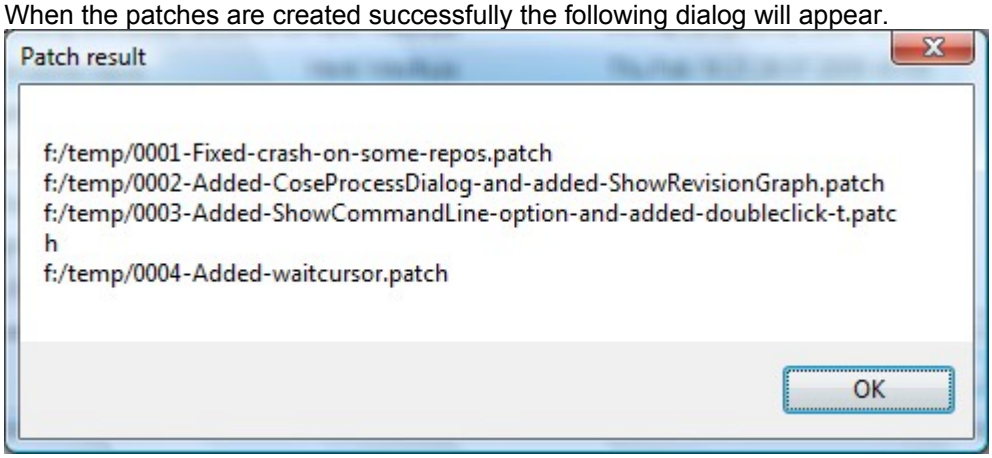
```
From 58c02ec4701c94c671a41e1e5d50c582e859851f Mon Sep 17 00:00:00 2001
From: Russell King <rmk@dyn-67.arm.linux.org.uk>
Date: Sun, 17 Apr 2005 15:40:46 +0100
Subject: [PATCH 000213/123824] [PATCH] ARM: h3600_irda_set_speed arguments

h3600_irda_set_speed() had the wrong type for the "speed" argument.
Fix this.

Signed-off-by: Russell King <rmk@arm.linux.org.uk>
---
 arch/arm/mach-sa1100/h3600.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/arch/arm/mach-sa1100/h3600.c b/arch/arm/mach-sa1100/h3600.c
index 9788d3a..84c8654 100644
--- a/arch/arm/mach-sa1100/h3600.c
+++ b/arch/arm/mach-sa1100/h3600.c
@@ -130,7 +130,7 @@ static int h3600_irda_set_power(struct device *dev, unsigned int state)
    return 0;
 }

-static void h3600_irda_set_speed(struct device *dev, int speed)
+static void h3600_irda_set_speed(struct device *dev, unsigned int speed)
 {
    if (speed < 4000000) {
        clr_h3600_egpio(IPAQ_EGPIO_IR_FSEL);
--
1.6.1.9.g97c34
```

7.1 Create patch

Format a single patch or patch series using the format patch dialog. You need to select the newest commit first and then select the oldest commit using ctrl-click. You can also select an interrupted patch series, but this is not recommended because the files will not be numbered.

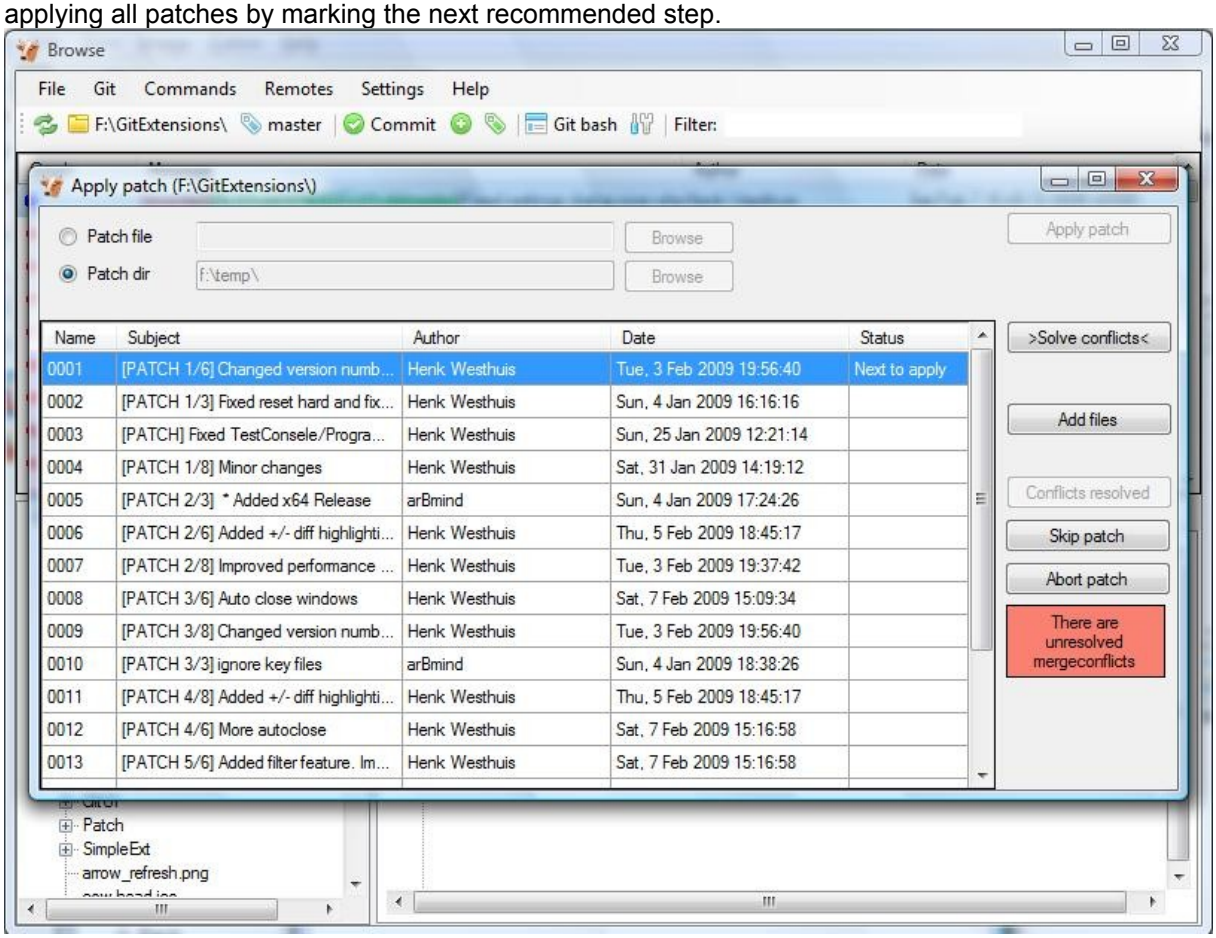


When the patches are created successfully the following dialog will appear.



7.2 Apply patches

It is possible to apply a single patch file or all patches in a directory. When there are merge conflicts applying the patch you need to resolve them before you can continue. Git Extensions will help you applying all patches by marking the next recommended step.

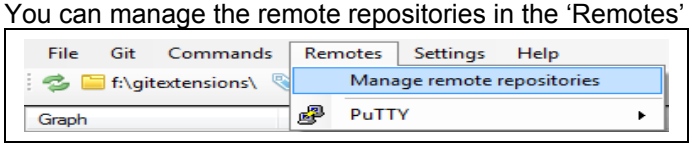


8 Remote features

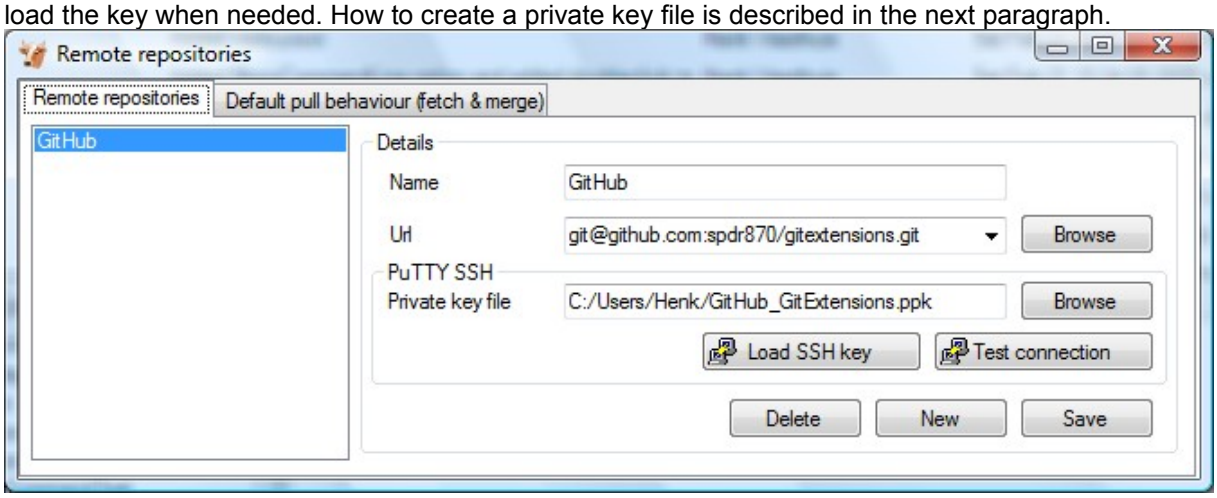
Git is a distributed source control management system. This means that all changes you make are local. When you commit changes, you only commit them to your local repository. To publish your local changes you need to push. In order to get changes committed by others, you need to pull.

8.1 Manage remote repositories

You can manage the remote repositories in the 'Remotes' menu.

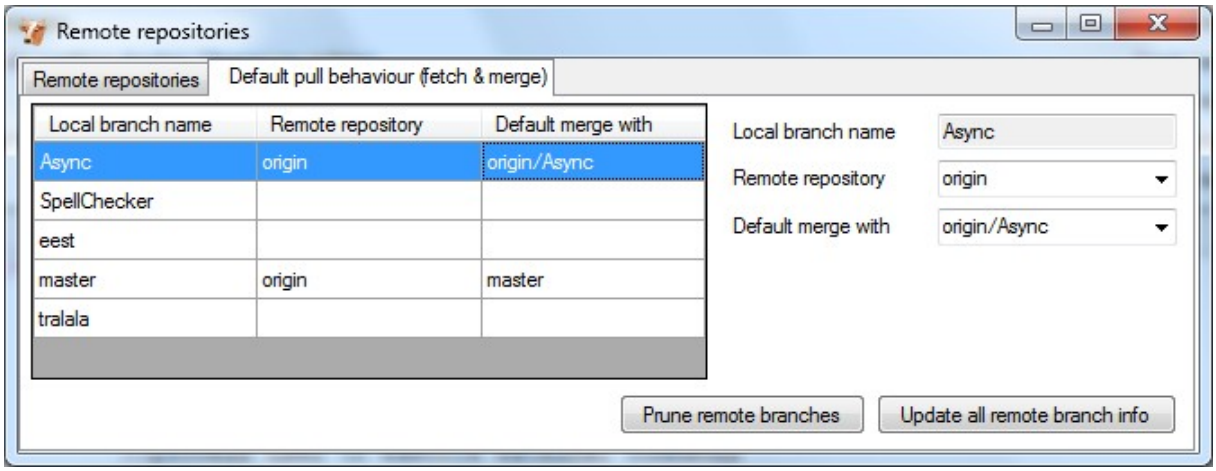


When you cloned your repository from a public repository, this remote is already configured. You can rename each remote for easy recognition. The default name after cloning a remote is 'origin'. If you use PuTTY as SSH client you can also enter the private key file for each remote. Git Extensions will load the key when needed. How to create a private key file is described in the next paragraph.



In the 'Default pull behaviour' tab you can configure the branches that need to be pulled and merged by default. If you configure this correctly you will not need to choose a branch when you pull or push. There are two buttons on this dialog:

Prune remote branches	Throw away remote branches that do not exist on the remote anymore
Update all remote branch info	Fetch all remote branch information



After cloning a repository you do not need to configure all remote branches manually. Instead you can checkout the remote branch and choose to create a local tracking branch.

8.2 Create SSH key

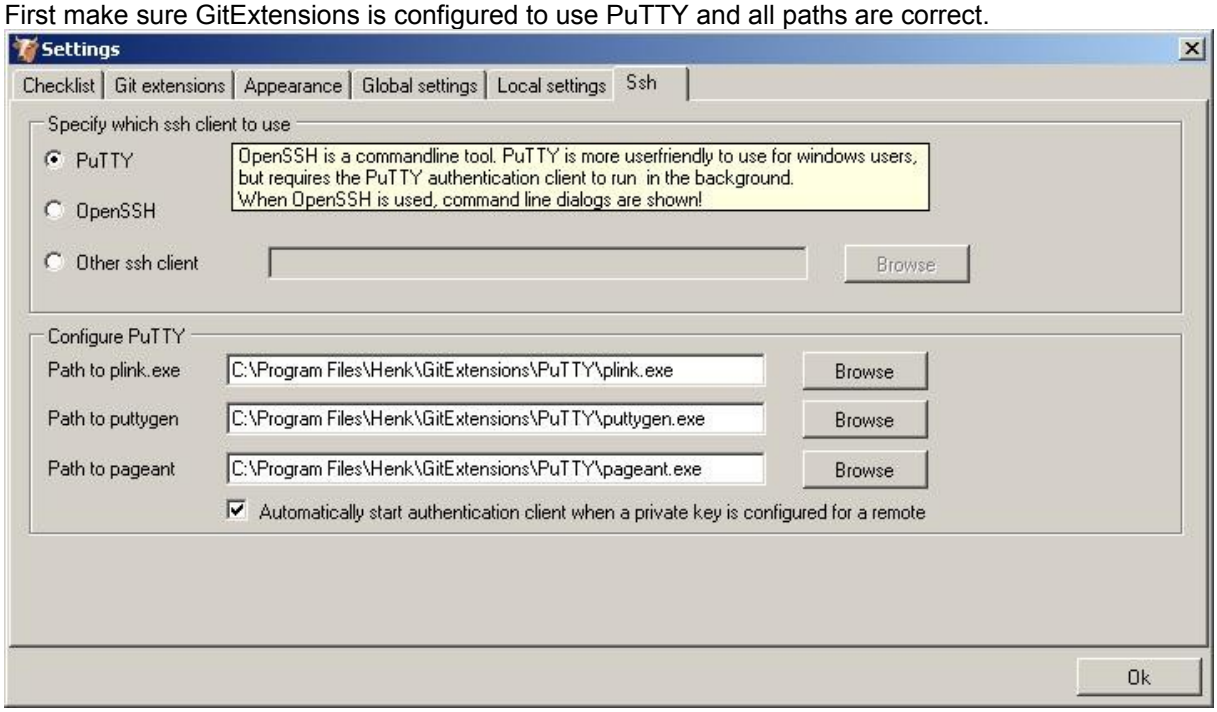
Git uses SSH for accessing private repositories. SSH uses a public/private key pair for authentication. This means you need to generate a private key and a public key. The private key is stored on your computer locally and the public key can be given to anyone. SSH will encrypt whatever you send using your secret private key. The receiver will then use the public key you send to decrypt the data.

This encryption will not protect the data itself but it protects the authenticity. Because the private key is only available to the sender, the receiver can be sure about the origin of the data. In practise the key pair is only used for the authentication process. The data itself will be encrypted using a key that is exchanged during this initial phase.

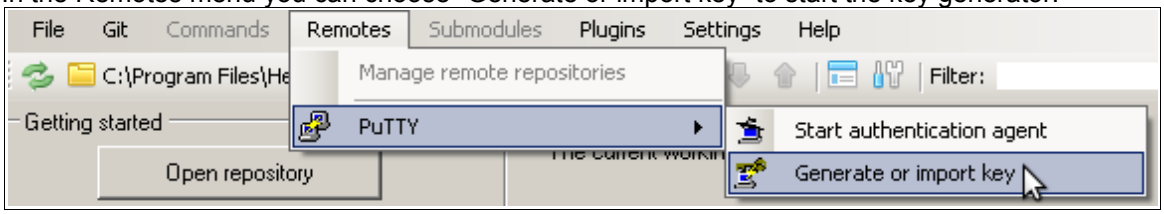
8.2.1 PuTTY and github

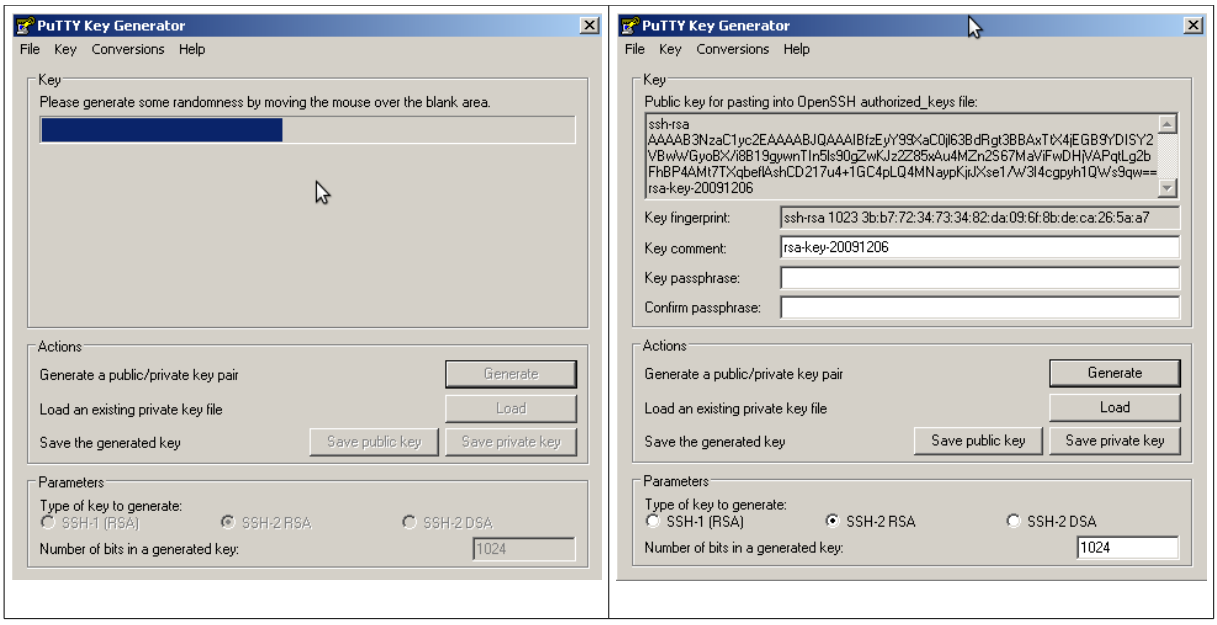
PuTTY is SSH client that for Windows that is a bit more user friendly then OpenSSH. Unfortunately PuTTY does not work with all servers. In this paragraph I will show how to generate a key for github using putty.

First make sure GitExtensions is configured to use PuTTY and all paths are correct.



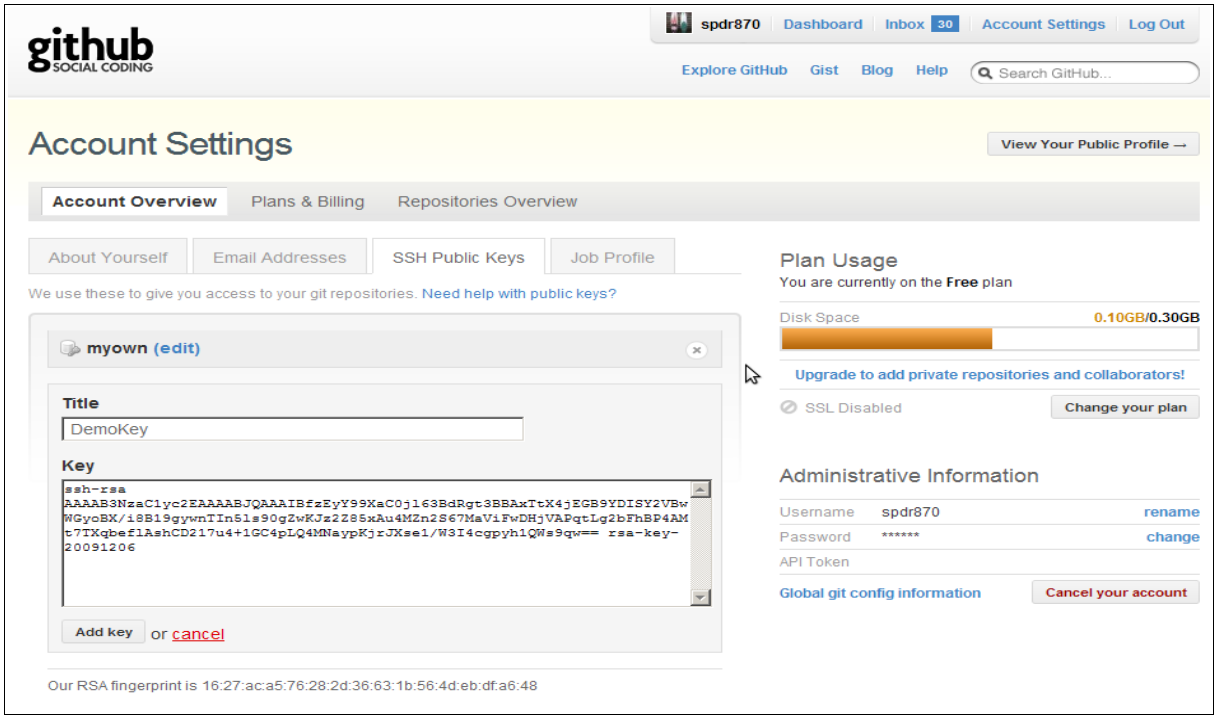
In the Remotes menu you can choose “Generate or import key” to start the key generator.



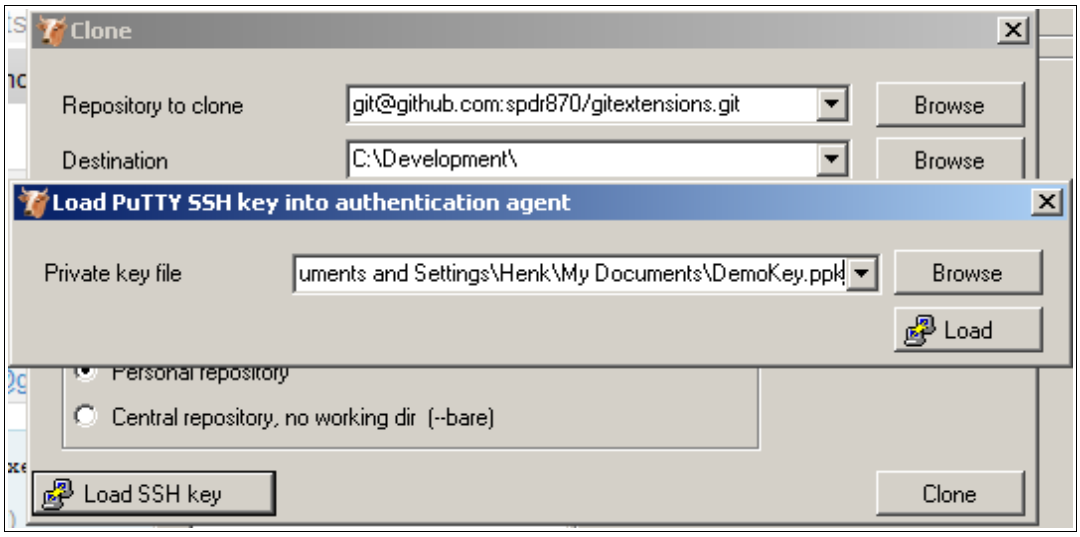


PuTTY will ask you to move the mouse around to generate a more random key. When the key is generated you can save the public and the private key in a file. You can choose to protect the private key with a password but this is not necessary.

Now you have a key pair you need to give github the public key. This can be done in “Account Settings” in the tab “SSH Public Keys”. You can add multiple keys here, but you only need one key for all repositories.



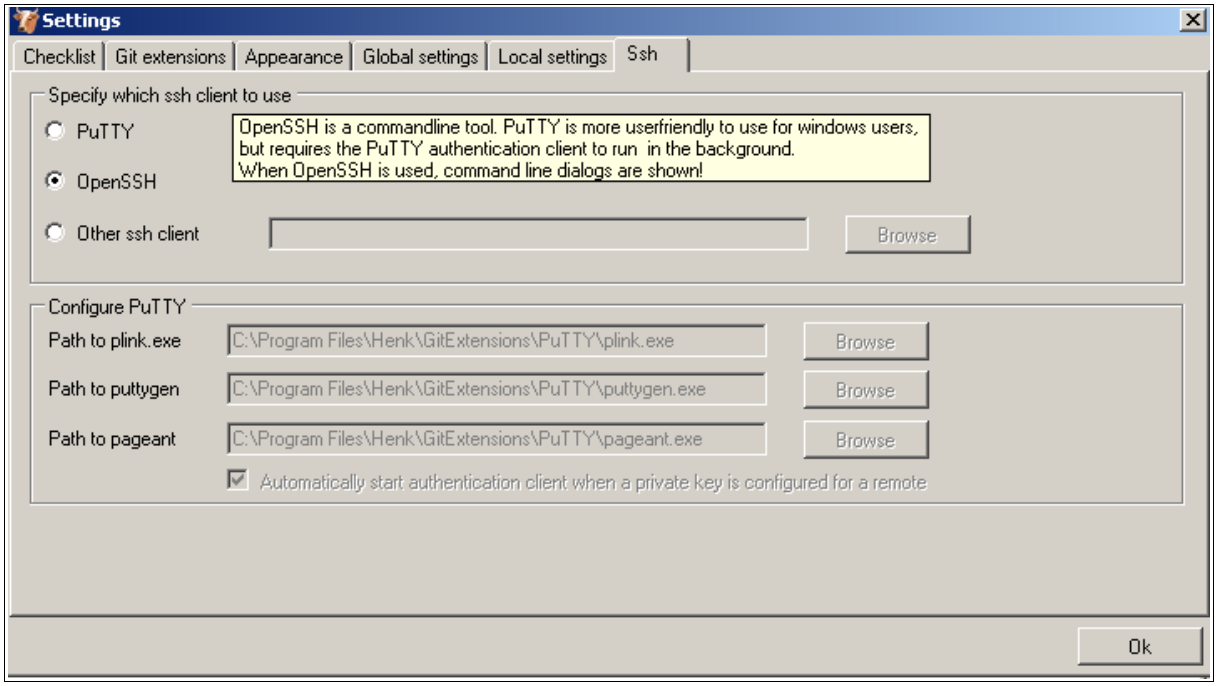
After telling github what public key to use to decrypt, you need to tell GitExtensions what private key to use to encrypt. In the clone dialog there is a “Load SSH key” button to load the private key into the PuTTY authentication agent. This can also be done manually by starting the PuTTY authentication agent and choose “add key” in the context menu in the system tray.



GitExtensions can load the private keys automatically for you when communicating with a remote. You need to configure the private key for the remote. This is done in the “Manage remote repositories” dialog.

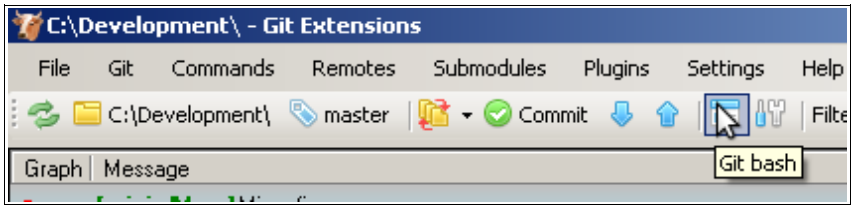
8.2.2 OpenSSH and github

When you choose to use OpenSSH you need to configure GitExtensions as shown in the screenshot below.

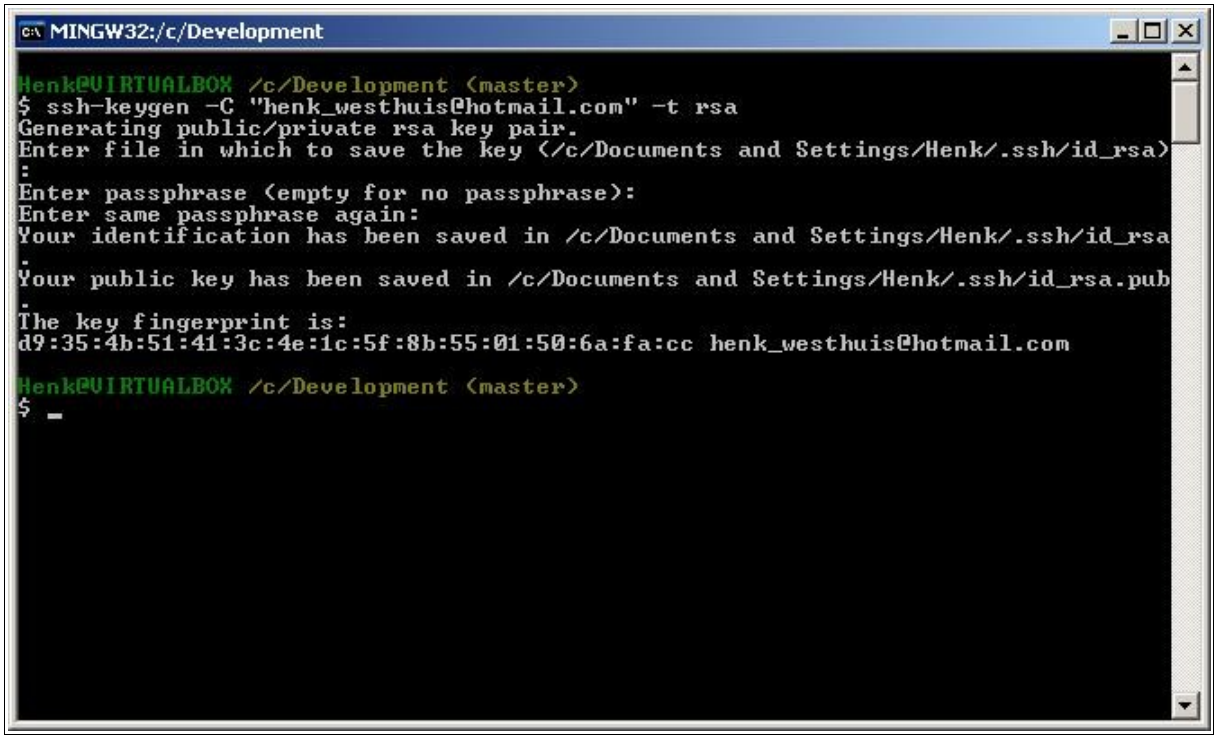


OpenSSH is the best SSH client there is but it lacks Windows support. Therefore it is slightly more complex to use. Another drawback is that GitExtensions cannot control OpenSSH and needs to show the command line dialogs when OpenSSH might be used. GitExtensions will show the command line window for every command that might require a SSH connection. For this reason PuTTY is the prefered SSH client in GitExtensions.

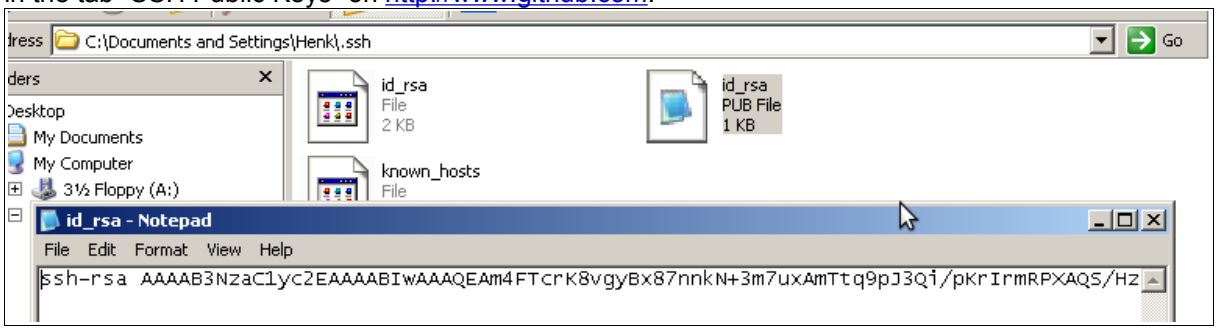
To generate a key pair in OpenSSH you need to go to the command line. I recommend to use the git bash because the path to OpenSSH is already set.



Type the following command: **ssh-keygen -C "your@email.com" -t rsa**
Use the same email address as the email address used in git. You will be asked where if you want to protect the private key with a password. This is not necessary. By default the public and private keys are stored in "c:\Documents and Settings\[User]\.ssh\" or "c:\Users\[user]\.ssh\".

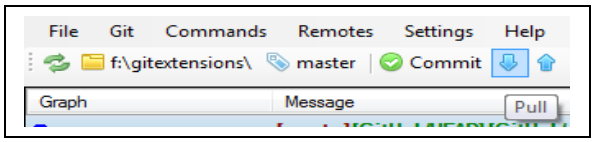


You do not need to tell GitExtensions about the private key because OpenSSH will load it for you. Now open the public key using notepad and copy the key to github. This can be done in "Account Settings" in the tab "SSH Public Keys" on http://www.github.com.



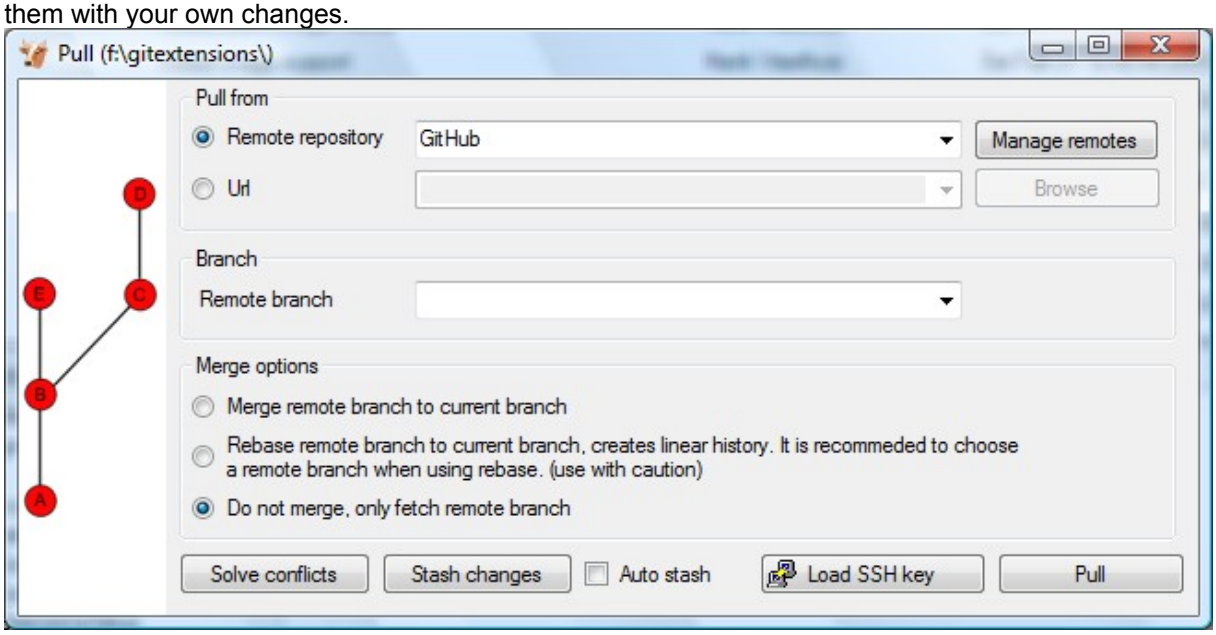
8.3 Pull changes

You can get remote changes using the pull function. Before you can pull remote changes you need to make sure there are no uncommitted changes in your local repository. If you have uncommitted changes you should commit them or stash them during the pull. You can read about how to use the stash in the Stash chapter.

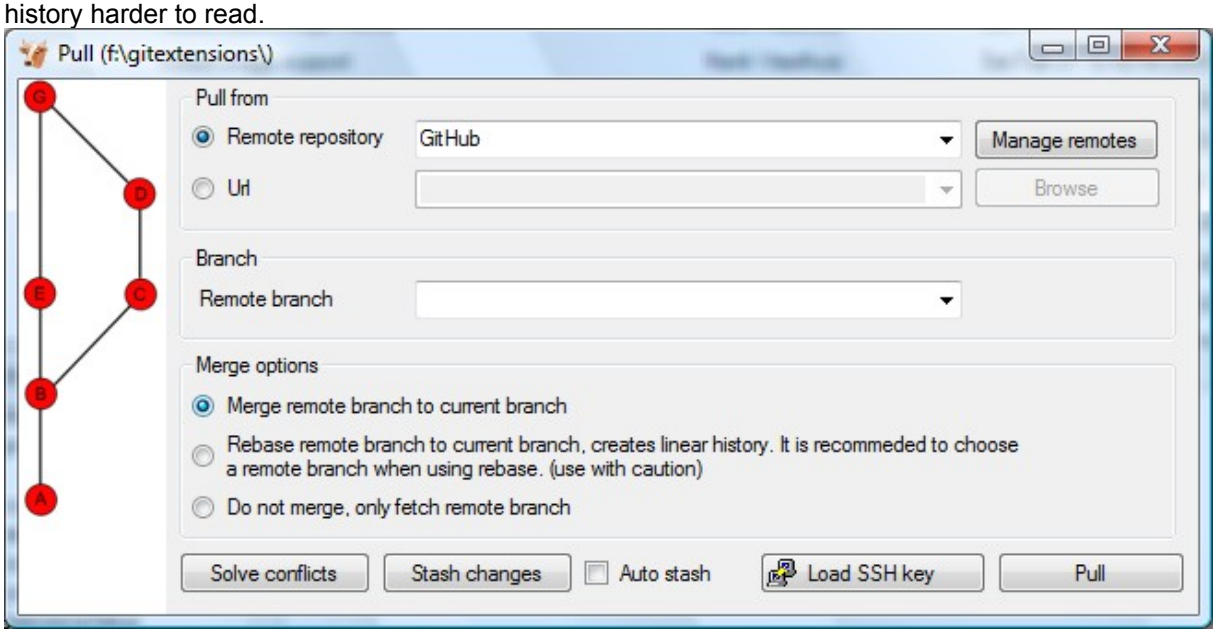


In order to get your personal repository up-to-date, you need to fetch changes from a remote repository. You can do this using the 'Pull' dialog. When the dialog starts the default remote for the current branch is set. You can choose another remote or enter a custom url if you like. When the remote branches configured correctly, you do not need to choose a remote branch.

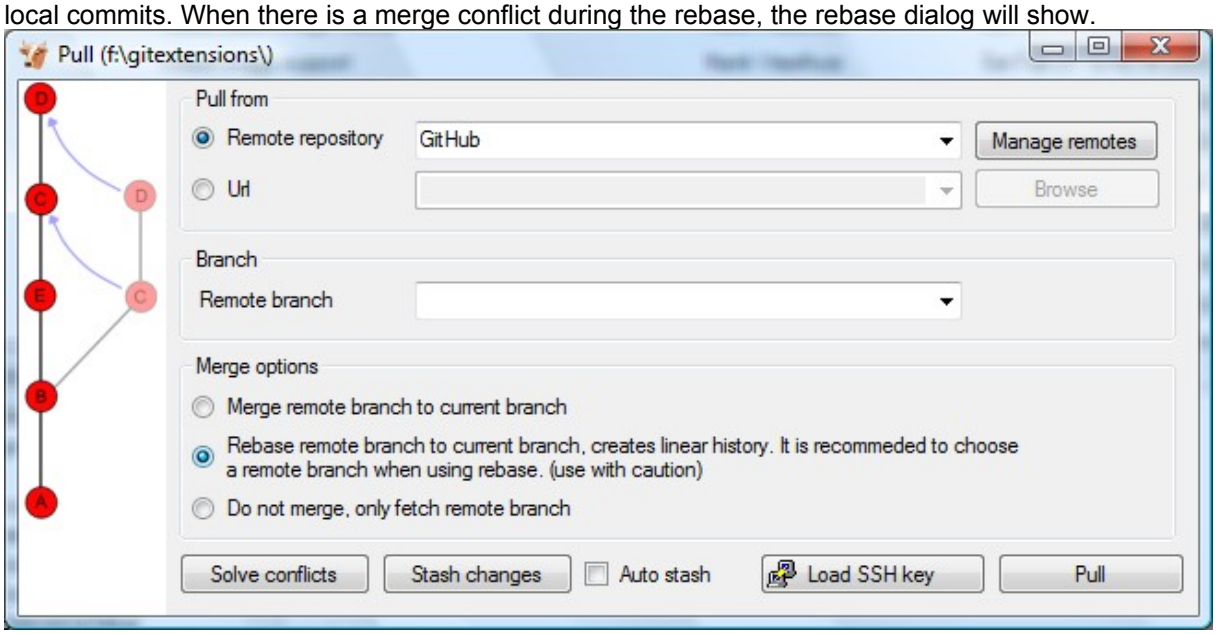
If you just fetch the commits from the remote repository and you already committed some changes to your local repository, the commits will be in a different branch. In the pull dialog this is illustrated in the image on the left. This can be useful when you want to review the changes before you want to merge them with your own changes.



When you choose to merge the remote branch after fetching the changes a branch will be created, and will be merged you're your commit. Doing this creates a lot of branches and merges, making the history harder to read.



Instead of merging the fetched commits with your local commits, you can also choose to rebase your commits on top of the fetched commits. This is illustrated on the left in the image below. A rebase will first undo your local commits (c and d), then fetch the remote commits (e) and finally recommit your local commits. When there is a merge conflict during the rebase, the rebase dialog will show.

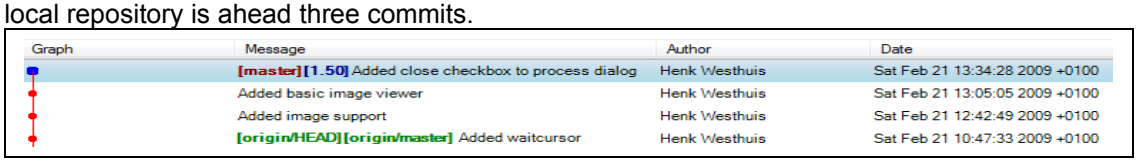


Next to the pull button there are some buttons that can be useful:

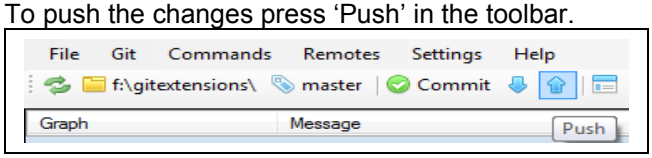
Solve conflicts	When there are merge conflicts, you can solve them by pressing this button.
Stash changes	When the working dir contains uncommitted changes, you need to stash them before pulling.
Auto stash	Check this checkbox if you want to stash before pulling. The stash will be reapplied after pulling.
Load SSH key	This button is only available when you use PuTTY as SSH client. You can press this button to load the key configured for the remote. If no key is set, a dialog will prompt for the key.

8.4 Push changes

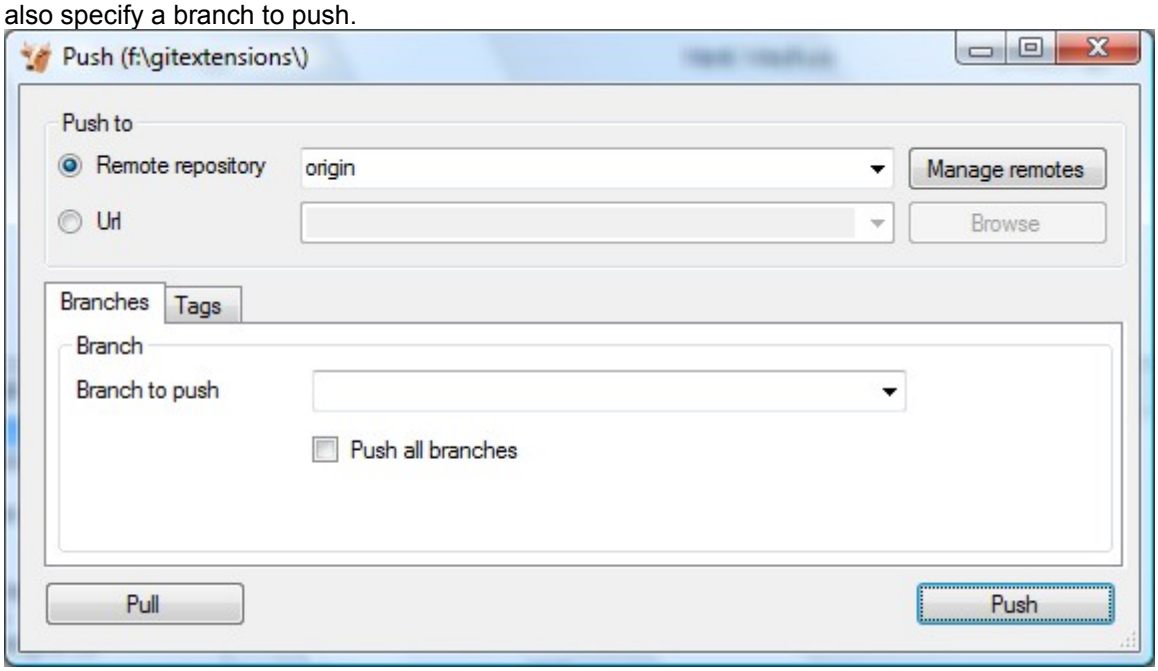
In the browse window you can check if there are local commits that are not pushed to a remote repository yet. In the image below the green labels mark the position of the master branch on the remote repository. The red label marks the position of the master branch on the local repository. The local repository is ahead three commits.



To push the changes press 'Push' in the toolbar.



The push dialog allows you to choose the remote repository to push to. The remote repository is set to the remote of the current branch. You can choose another remote or choose a url to push to. You can also specify a branch to push.

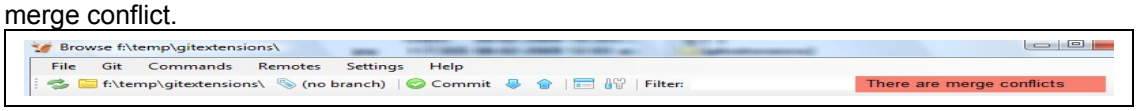


Tags are not pushed to the remote repository. If you want to push a tag you need to open the `Tags` tab in the dialog. You can choose to push a singe tag or all tags. No commits will be pushed when the `Tags` tab is selected, only tags.

You can not merge your changes in the remote repository. Merging must be done locally. This means that you cannot push your changes before the commits are merged locally. In practice you need to pull before you can push most of the times.

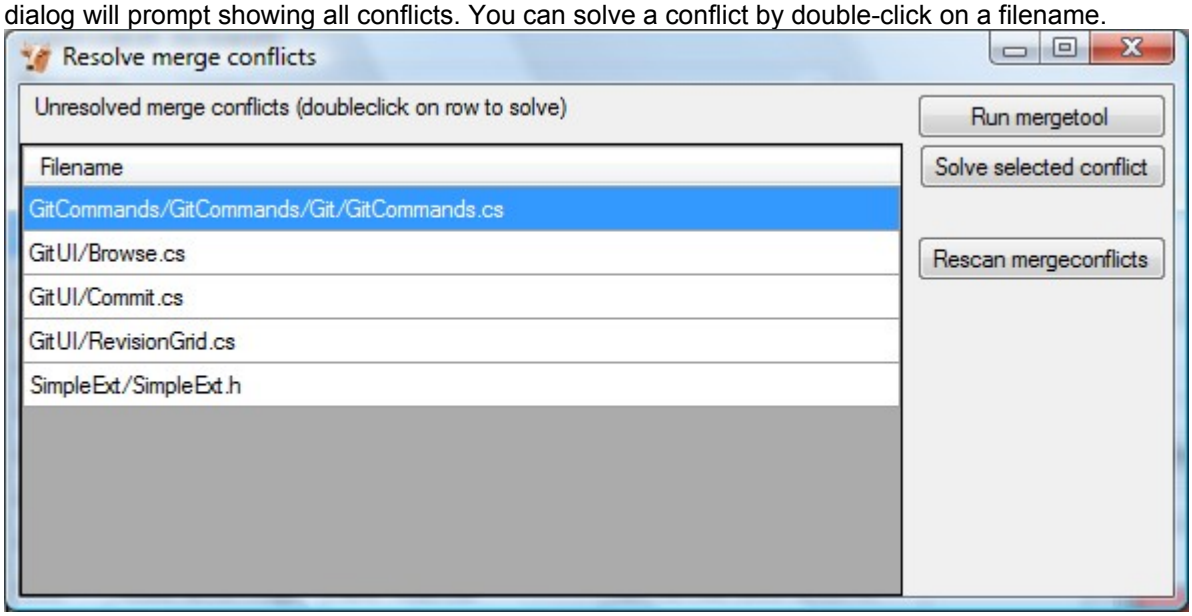
9 Merge conflicts

When merging branches or commits you can get merge conflicts. Git will try to resolve these, but some conflicts need to be resolved manually. Git Extensions will show warnings when there is a merge conflict.



9.1 Handle merge conflicts

To solve merge conflicts just click on a warning or open the merge conflict dialog from the menu. A dialog will prompt showing all conflicts. You can solve a conflict by double-click on a filename.



There are three kinds of conflicts:

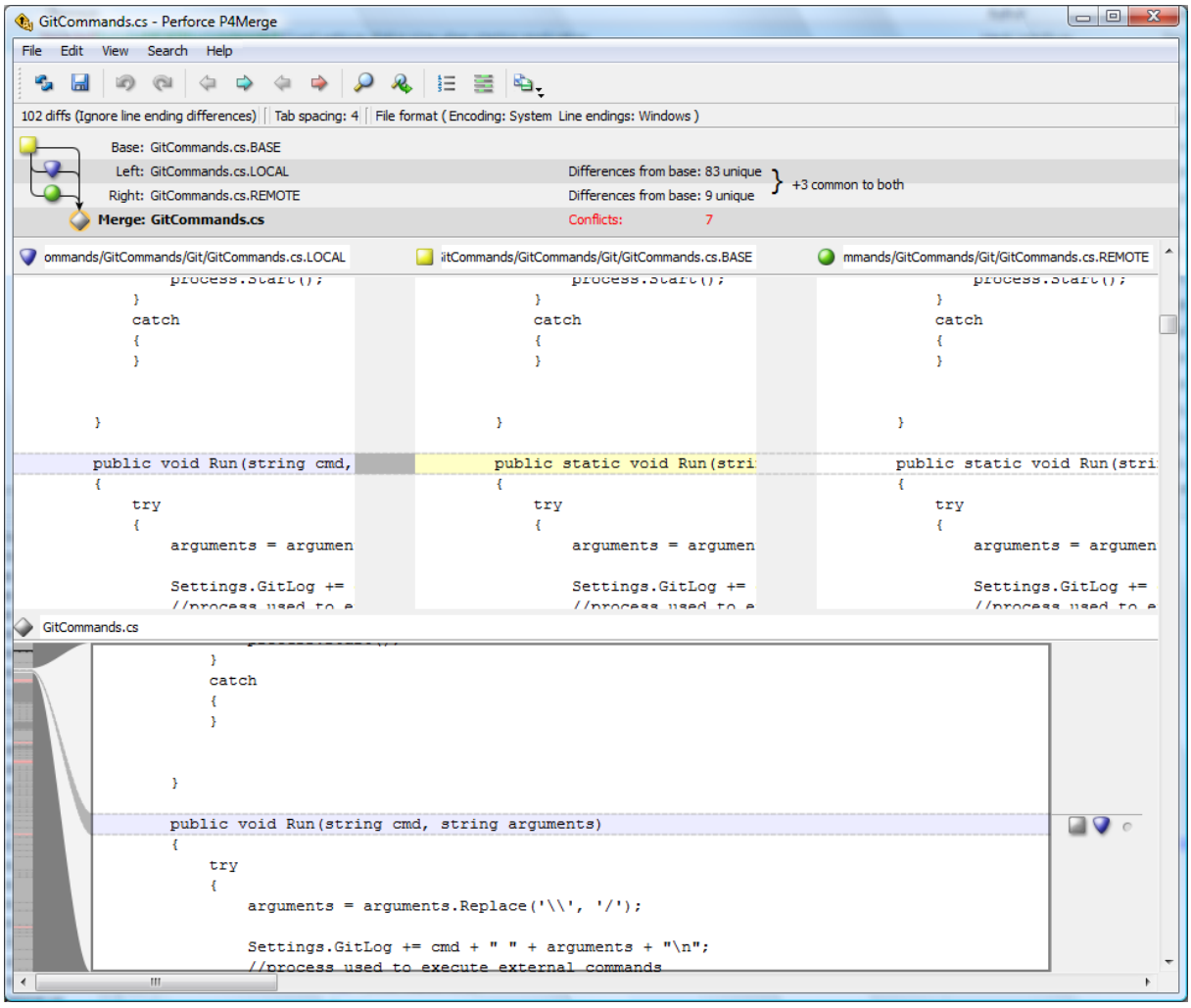
File deleted and changed	Use modified or deleted file?
File deleted and created	Use created or deleted file?
File changed both locally and remotely	Start merge tool.

If the file is deleted in one commit and changed in another commit, a dialog will ask to keep the modified file or delete the file. When there is a conflicting change the merge tool will be started. You can configure the tool you want to use for merge conflicts. The image below shows Perforce P4Merge a free to use merge tool. Git Extensions is packaged with KDiff3, an open source merge tool.

In the merge tool you will see four versions of the same file:

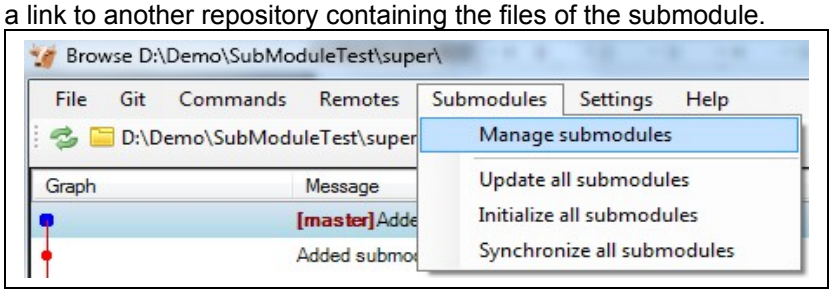
Base	The latest version of the file that exist in both repositories
Local	The latest local version of the file
Remote	The latest remote version of the file
Merged	The result of the merge

When you are in the middle of a merge the file named local represents your file. When you are in the middle of a rebase the file named remote represents your file. This can be confusing, so double check if you are in doubt.


GitCommands.cs - Perforce P4Merge
File Edit View Search Help
102 diffs (Ignore line ending differences) Tab spacing: 4 File format (Encoding: System Line endings: Windows)
Base: GitCommands.cs.BASE
Left: GitCommands.cs.LOCAL
Right: GitCommands.cs.REMOTE
Merge: GitCommands.cs
Differences from base: 83 unique
Differences from base: 9 unique
+3 common to both
Conflicts: 7
ommands/GitCommands/Git/GitCommands.cs.LOCAL
itCommands/GitCommands/Git/GitCommands.cs.BASE
mmands/GitCommands/Git/GitCommands.cs.REMOTE
public void Run(string cmd,
public static void Run(stri
public static void Run(stri
GitCommands.cs
public void Run(string cmd, string arguments)
{
try
{
arguments = arguments.Replace('\\', '/');
Settings.GitLog += cmd + " " + arguments + "\n";
//process used to execute external commands

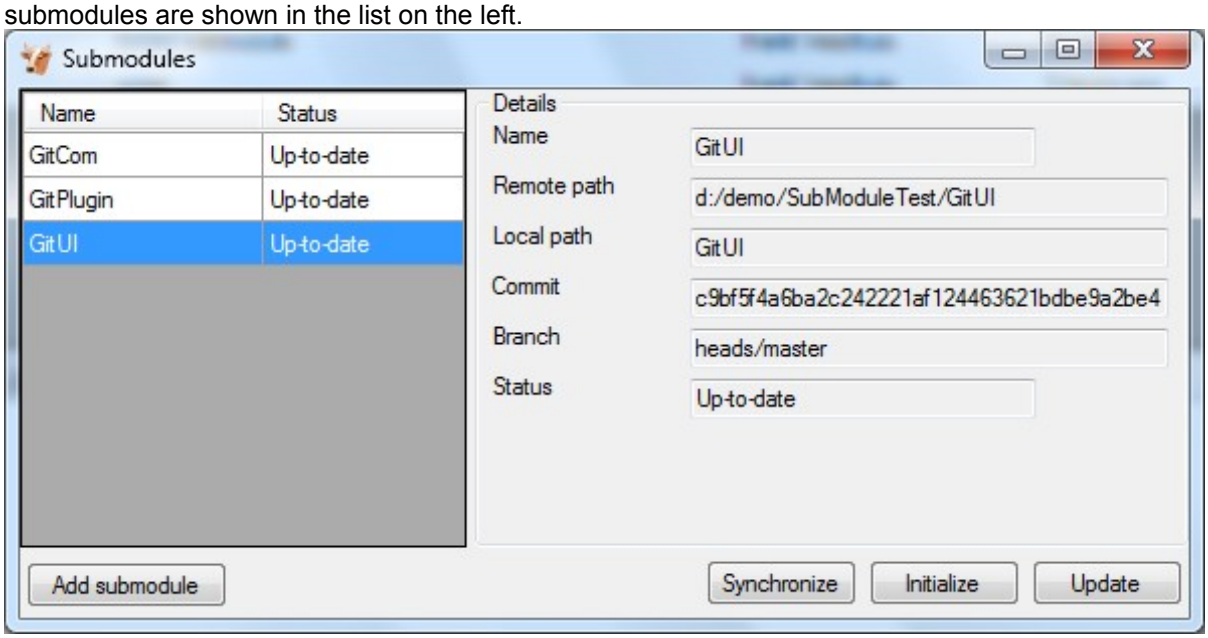
10 Submodules

Large projects can be split into smaller parts using submodules. A submodule contains the name, url and revision of another repository. To create a submodule in an existing git repository you need to add a link to another repository containing the files of the submodule.



10.1 Manage submodules

The current state of the submodules can be viewed with the 'Manage submodules' function. All submodules are shown in the list on the left.

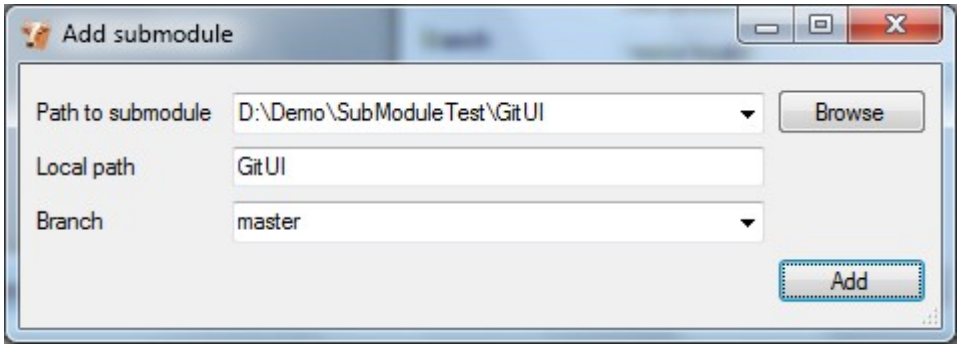


Add submodule	Add a new submodule to the repository
Synchronize	Synchronizes the remote URL configuration setting to the value specified in .gitmodules for the selected submodule.
Initialize	Initialize the selected submodules, i.e. register each submodule name and url found in .gitmodules into .git/config. The submodule will also be updated.
Update	Update the registered submodules, i.e. clone missing submodules and checkout the commit specified in the index of the containing repository.

10.2 Add submodule

To add a new submodule choose ‘Add submodule’ in the ‘Manage submodules’ dialog.

Path to submodule	Path to the remote repository to use as submodule.
Local path	Local path to this submodule, relative to the root of the current repository.
Branch	Branch to track

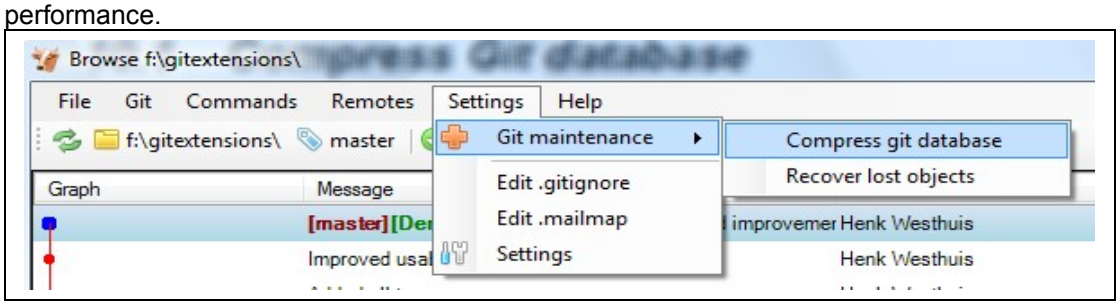


11 Maintenance

In this chapter some of the functions to maintain a repository are discussed.

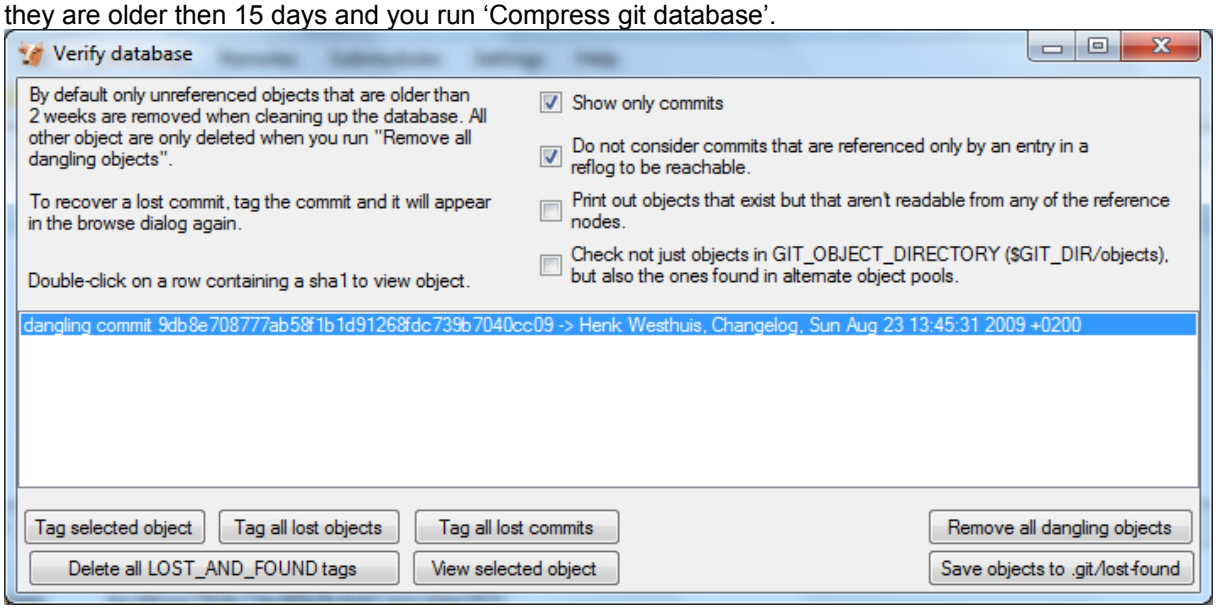
11.1 Compress Git database

Git will create a lot of files. You can run the 'Compress git database' to pack all small files building up a repository into one big file. Git will also garbage collect all unused objects that are older then 15 days. When a database is fragmented into a many small files compressing the database can increase performance.



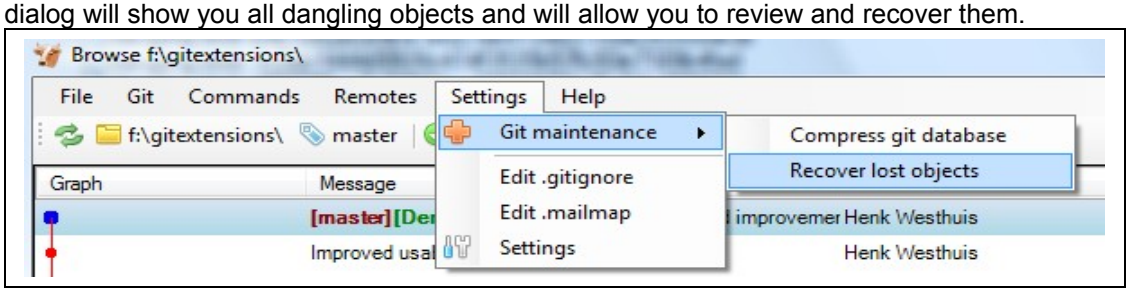
11.2 Verify Git database

Normally Git will not delete files right away when you remove something from your repository. The reason for this is that you can restore deleted items if you need to. Git will delete removed items when they are older then 15 days and you run 'Compress git database'.



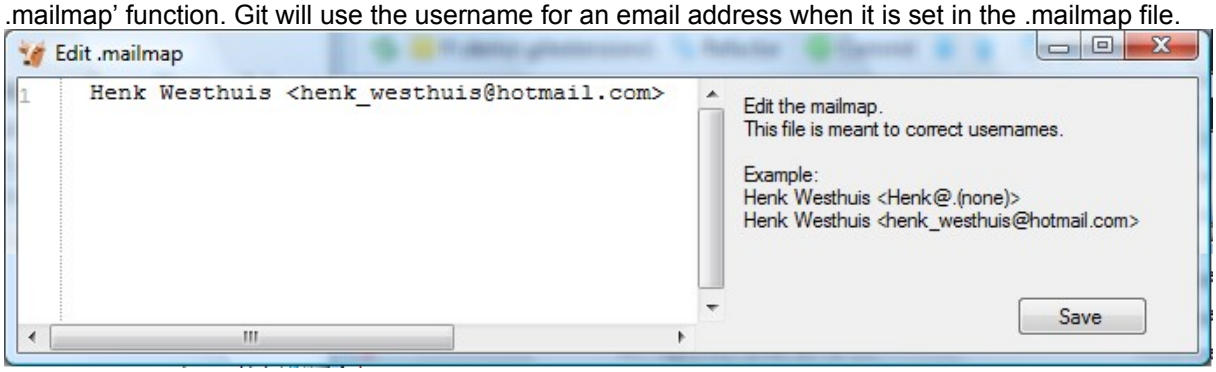
11.3 Recover dangling objects

If you accidently deleted a commit you can try to recover it using the `Recover lost objects´ function. A dialog will show you all dangling objects and will allow you to review and recover them.



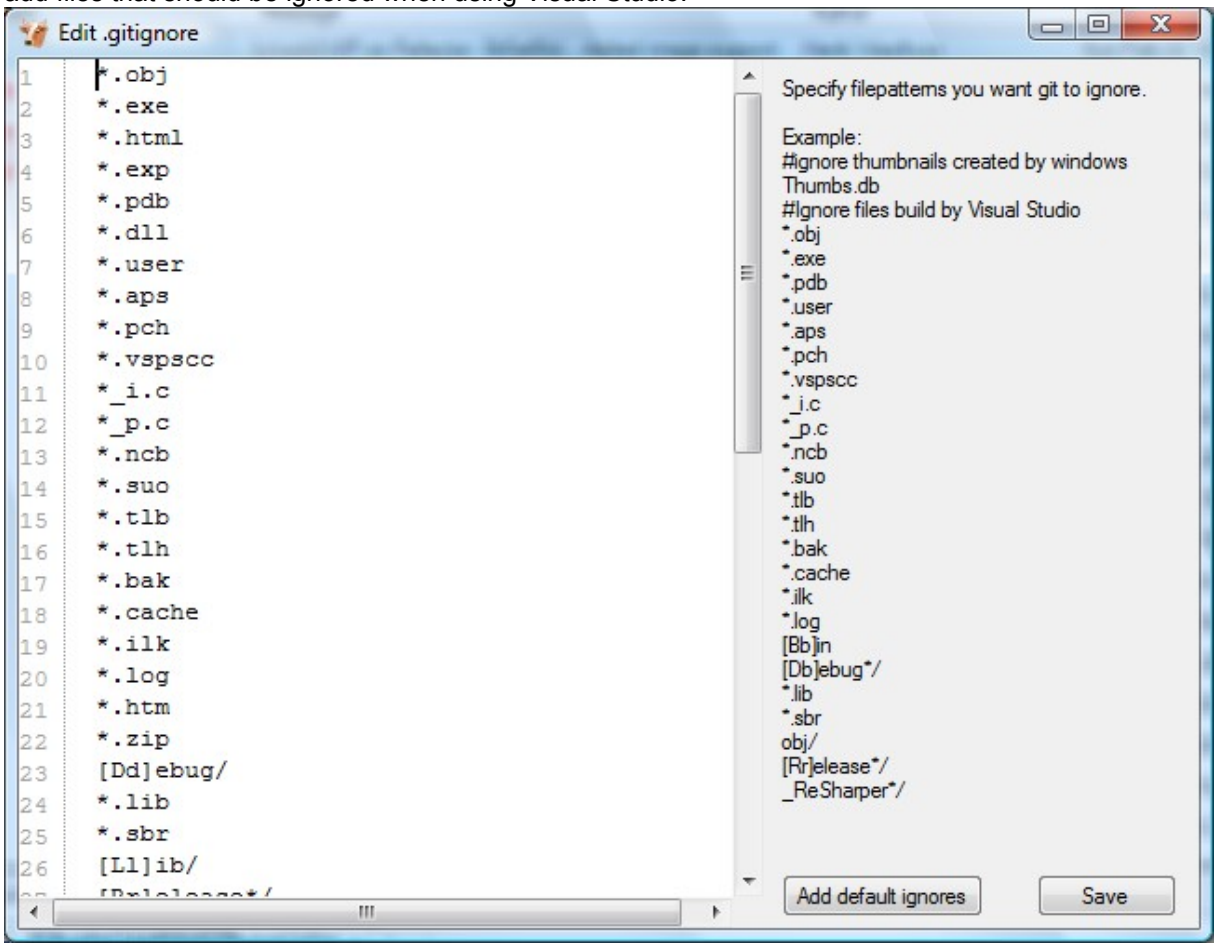
11.4 Fix user names

When someone accidentally committed using a wrong username this can be fixed using the ‘Edit .mailmap’ function. Git will use the username for an email address when it is set in the .mailmap file.



11.5 Ignore files

Git will track all files that are in the working directory. Normally you do not want to exclude all files that are created by the compiler. You can add files that should be ignored to the .gitignore file. You can use wildcards and regular expressions. All entries are case sensitive. The button ‘Add default ignores’ will add files that should be ignored when using Visual Studio.



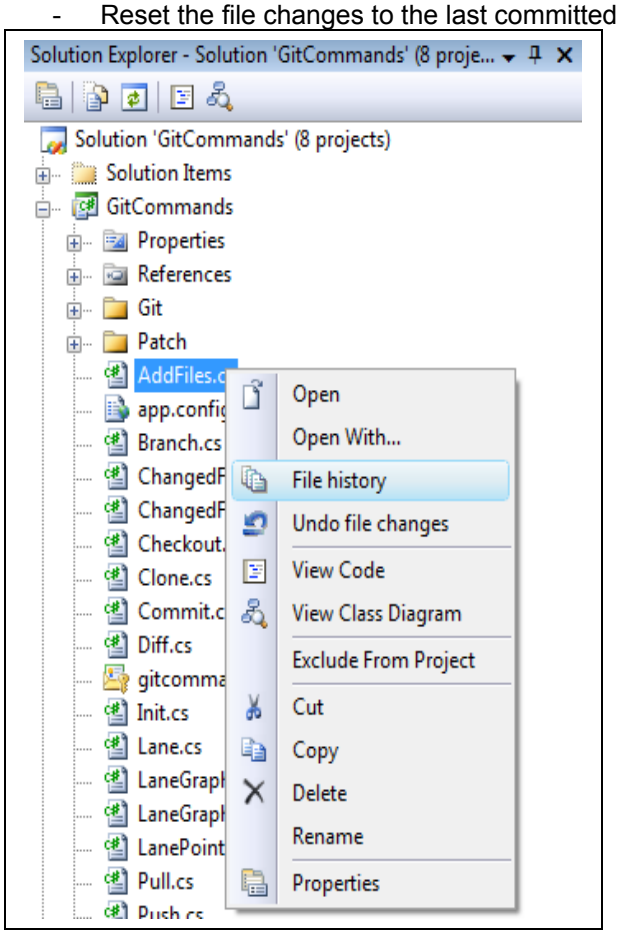
12 Integration

During installation you can choose to install the Visual Studio plug-in and shell extensions.

12.1 Visual Studio

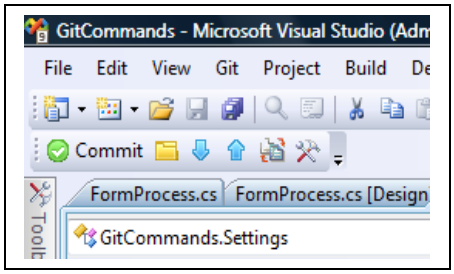
There are two options in the context menu on files.

- View the file history by choosing the ‘File history’ option.
- Reset the file changes to the last committed revision.

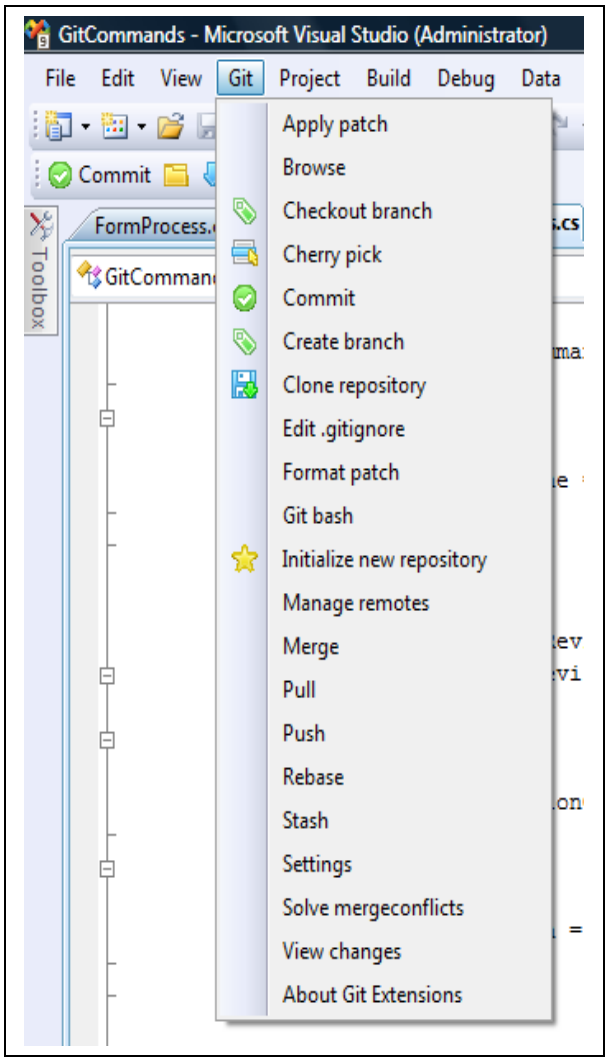


A Git Extensions toolbar allows you to perform the most common actions.

	Commit
	Browse
	Pull
	Push
	Stash changes
	Settings

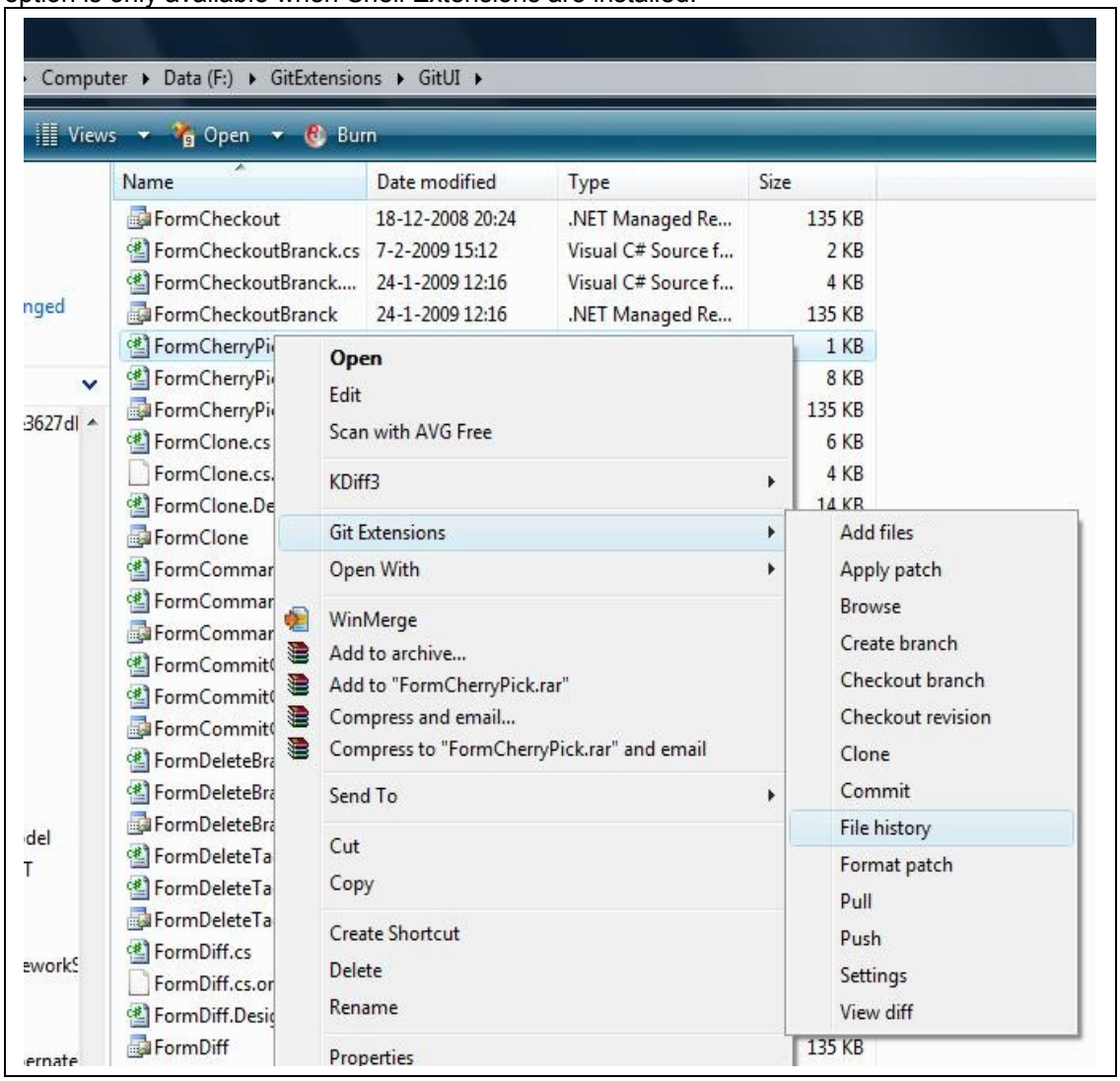


Almost all function can be started from the ‘Git’ menu in Visual Studio.


GitCommands - Microsoft Visual Studio (Administrator)
File
Edit
View
Git
Project
Build
Debug
Data
Commit
Toolbox
Apply patch
Browse
Checkout branch
Cherry pick
Commit
Create branch
Clone repository
Edit .gitignore
Format patch
Git bash
Initialize new repository
Manage remotes
Merge
Pull
Push
Rebase
Stash
Settings
Solve mergeconflicts
View changes
About Git Extensions

12.2 Windows Explorer

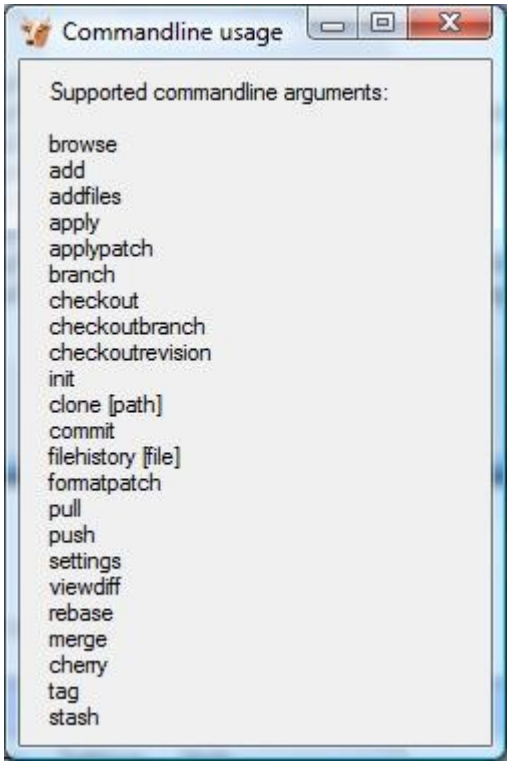
The common commands can be started from Windows Explorer using the shell extensions. This option is only available when Shell Extensions are installed.

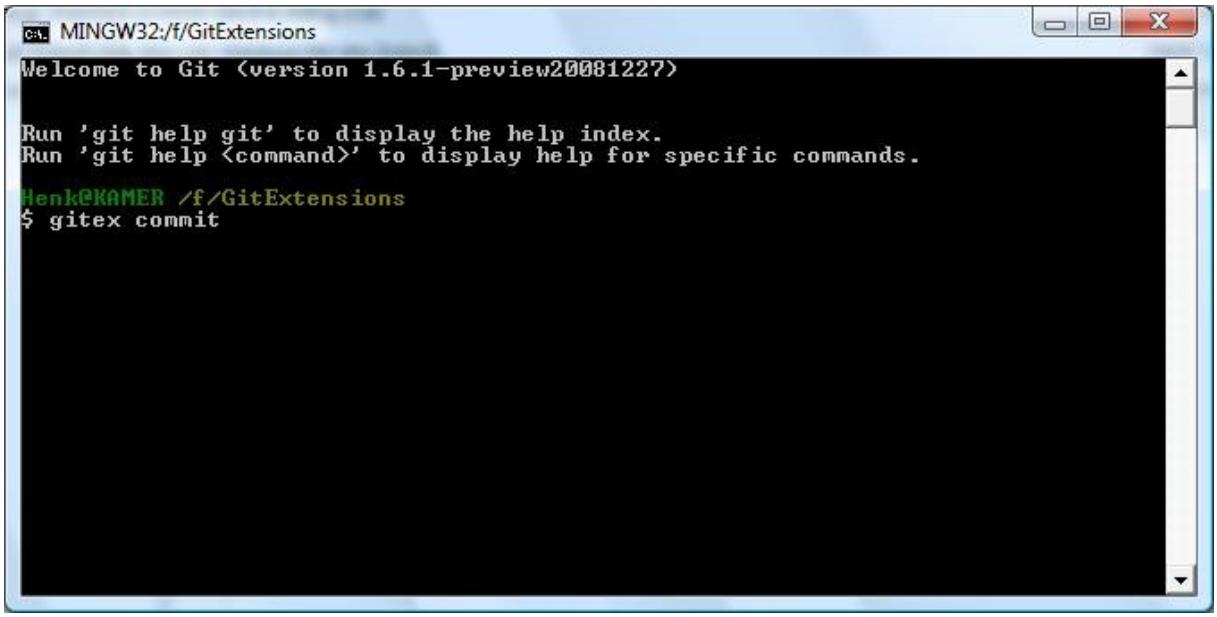


13 Command line

13.1 Git Extensions command line

Most features can be started from the command line. It is recommended to add gitex.cmd to the path when using from the command line.





The following functions are available from the command line:

- Add files
- Apply patches
- Archive
- Create branches/tags
- Checkout branches/tags/revisions
- Merge branches
- Rebase
- Commit changes
- Stash changes, with multiple stash support
- Clone repositories
- Push
- Pull
- SSH support
- All Git remote features
- Init new repository
- Single file history
- Full development history
- Difference reports
- Browsing repository
- Support for external mergetools

Git Cheat Sheet

Create new repository
```
$ git init
```
Create shared repository
```
$ git init --bare --shared=all
```

Clone repository
```
$ git clone c:/demo1 c:/demo2
```

Checkout branch
```
$ git checkout <name>
```
Create branch
```
$ git branch <name>
```
Delete branch
```
$ git branch -d <name>
```
Merge branch (from the branch to merge into)
```
$ git merge PDC
```
Solve conflicts (add --tool=kdiff3 if no mergetool is specified)
```
$ git mergetool
$ git commit
```

Create tag
```
$ git tag <name>
```

Add files/changes (. for all files)
```
$ git add .
```
Commit added files/changes (--amend to amend to last commit)
```
$ git commit –m “Enter commit message”
```

Discard changes
```
$ git reset --hard
```

Create patch (-M = detect renames –C = detect copies)
```
$ git format-patch –M –C origin
```

Apply patch without merging
```
$ git apply c:/patch/0001-employee.patch
```
Merge patch
```
$ git am --3way --signoff c:/patch/0001-employee.patch
```
Solve conflicts (add --tool=kdiff3 if no mergetool is specified)
```
$ git mergetool

$ git am --3way --resolved
```

Stash changes
```
$ git stash
```
Apply stashed changes
```
$ git stash apply
```
Pull changes (add --rebase to rebase instead of merge)
```
$ git pull c:/demo1 master
```
Solve conflicts (add --tool=kdiff3 if no mergetool is specified)
```
$ git mergetool
$ git commit
```
Push changes (in branch $ git push c:/demo1 master master:<new>)
```
$ git push c:/demo1
```

Blame
```
$ git blame –M –w <filename>
```

Help
```
$ git <command> --help
```

Default names	
master	: default branch
origin	: default upstream repository
HEAD	: current branch
HEAD^	: parent of HEAD
HEAD-4	: the great-great grandparent of HEAD