Git Extensions

U S E R M A N U A L

Henk Westhuis
Version 0.2

Index

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

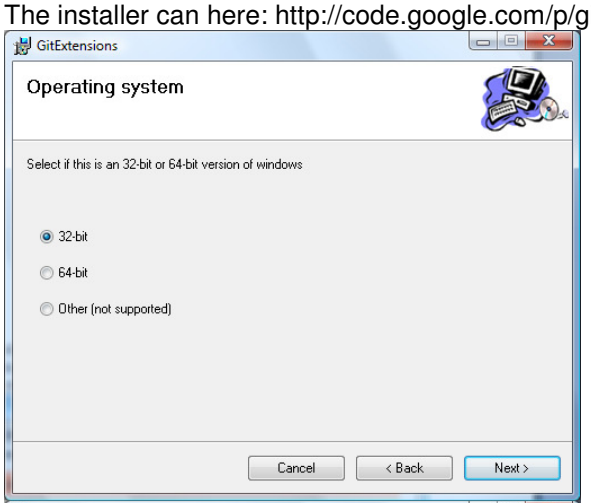
There are video tutorials for some basic functions on YouTube.

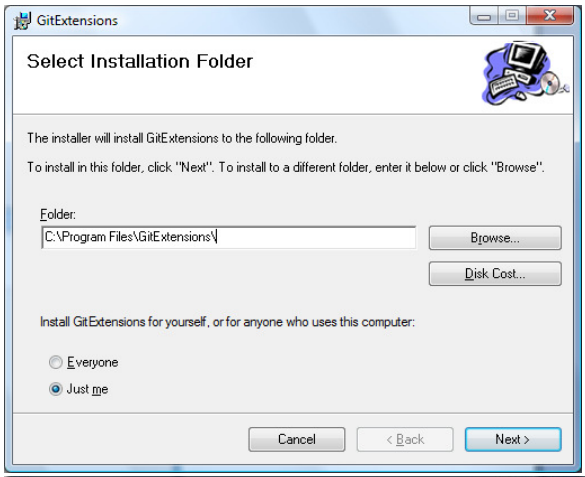
- 1 Clone - Git Extensions - http://www.youtube.com/watch?v=TIZXSkJGKF8
- 2 Commit changes - http://www.youtube.com/watch?v=B8uvje6X7lo
- 3 Push changes - http://www.youtube.com/watch?v=JByfXdbVAiE
- 4 Pull changes - http://www.youtube.com/watch?v=9g8gXPsi5Ko
- 5 Handle merge conflicts - http://www.youtube.com/watch?v=Kmc39RvuGM8

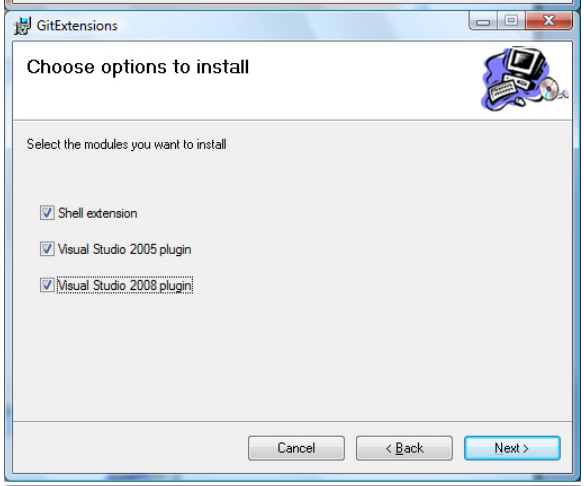
2 Getting started

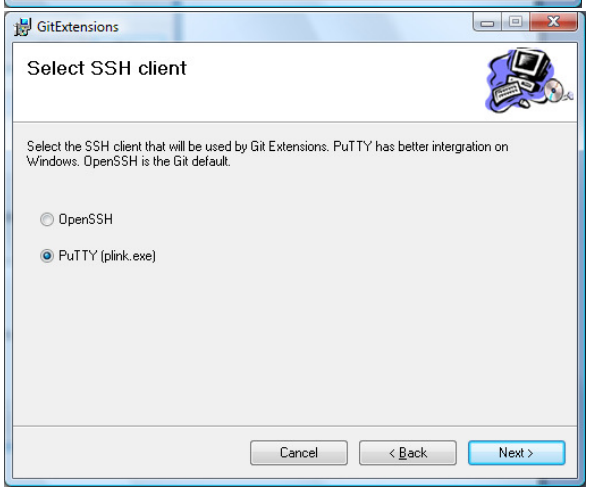
2.1 Install

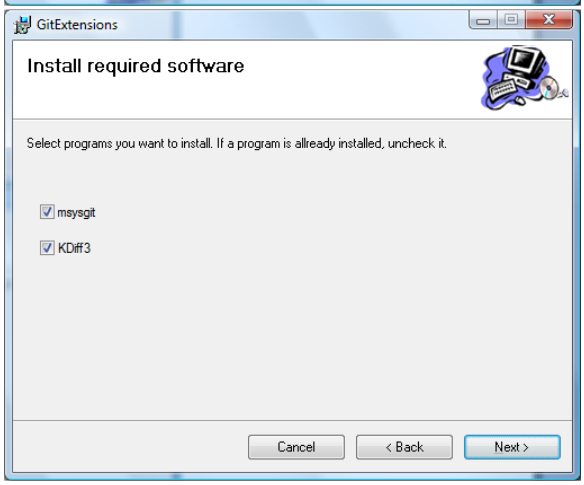
There is a single click installer that installs MSysGit, kdif3 and Git Extensions. You can choose to install the 32bit or 64bit version on the fist page of the installer.
The installer can here: http://code.google.com/p/gitextensions/





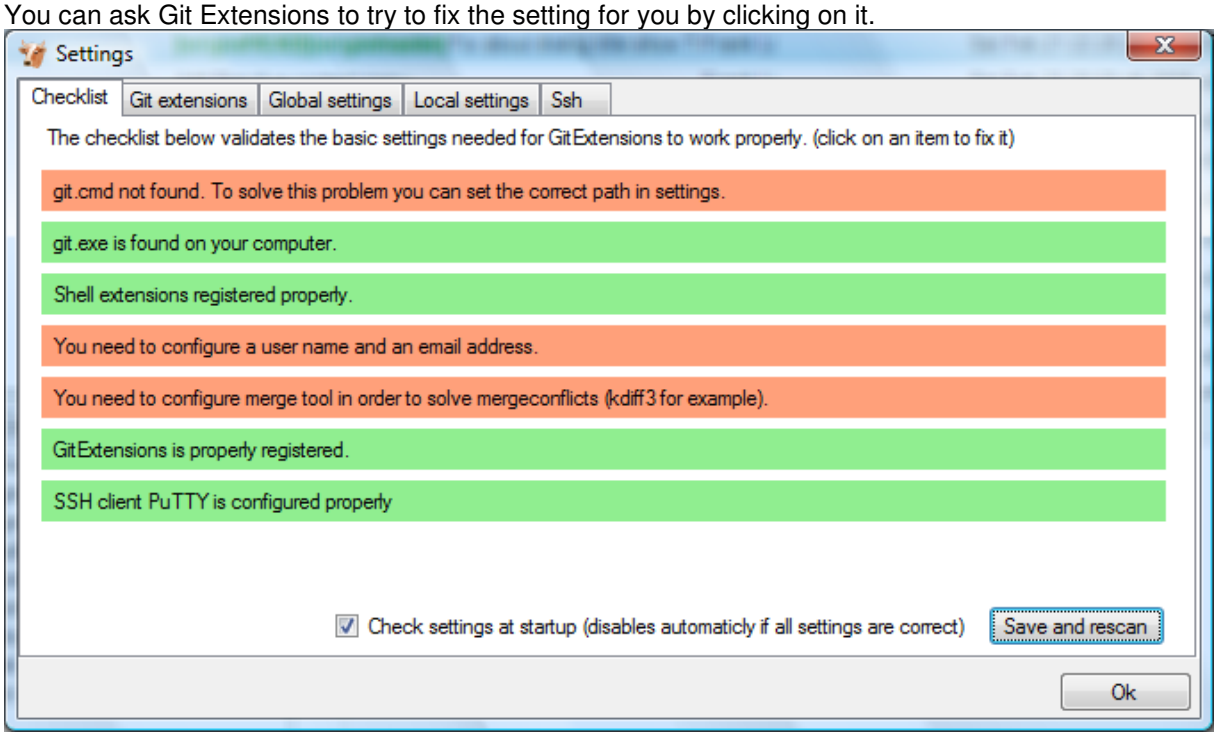






2.2 Settings

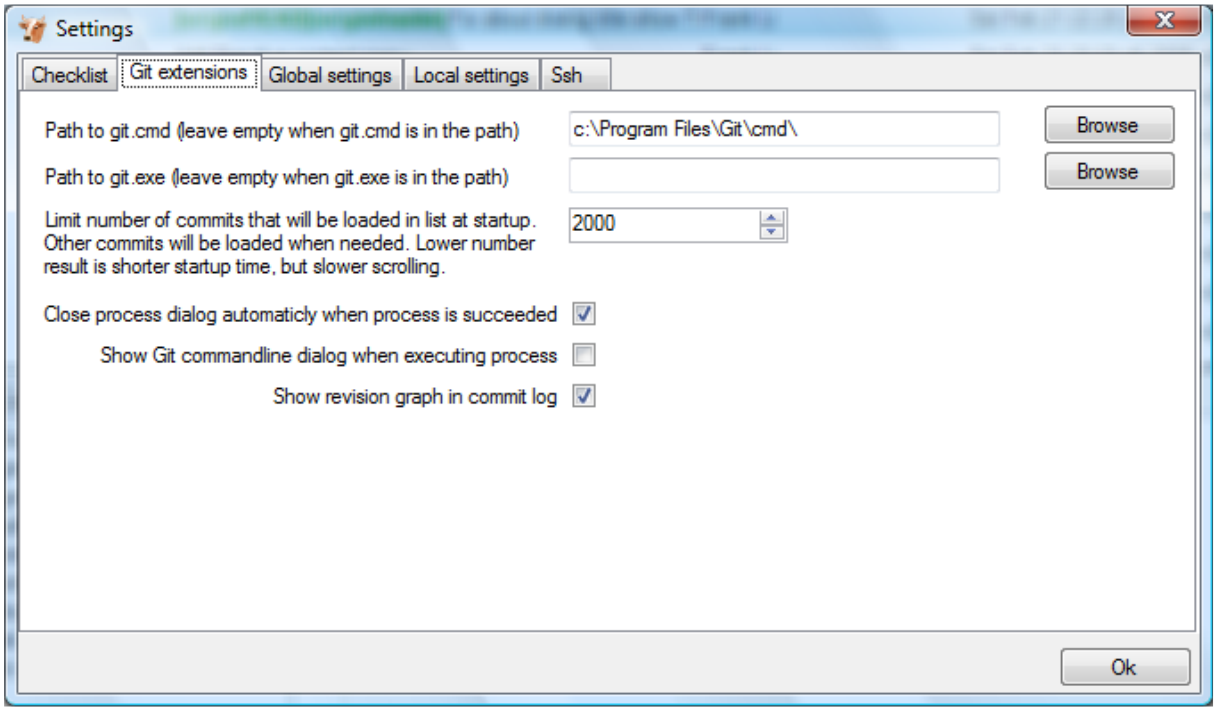
All settings will be verified when Git Extensions is started for the first time. If Git Extensions requires any settings to be changed the settings dialog will be shown. All incorrect settings will be marked red. You can ask Git Extensions to try to fix the setting for you by clicking on it.



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 config 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.

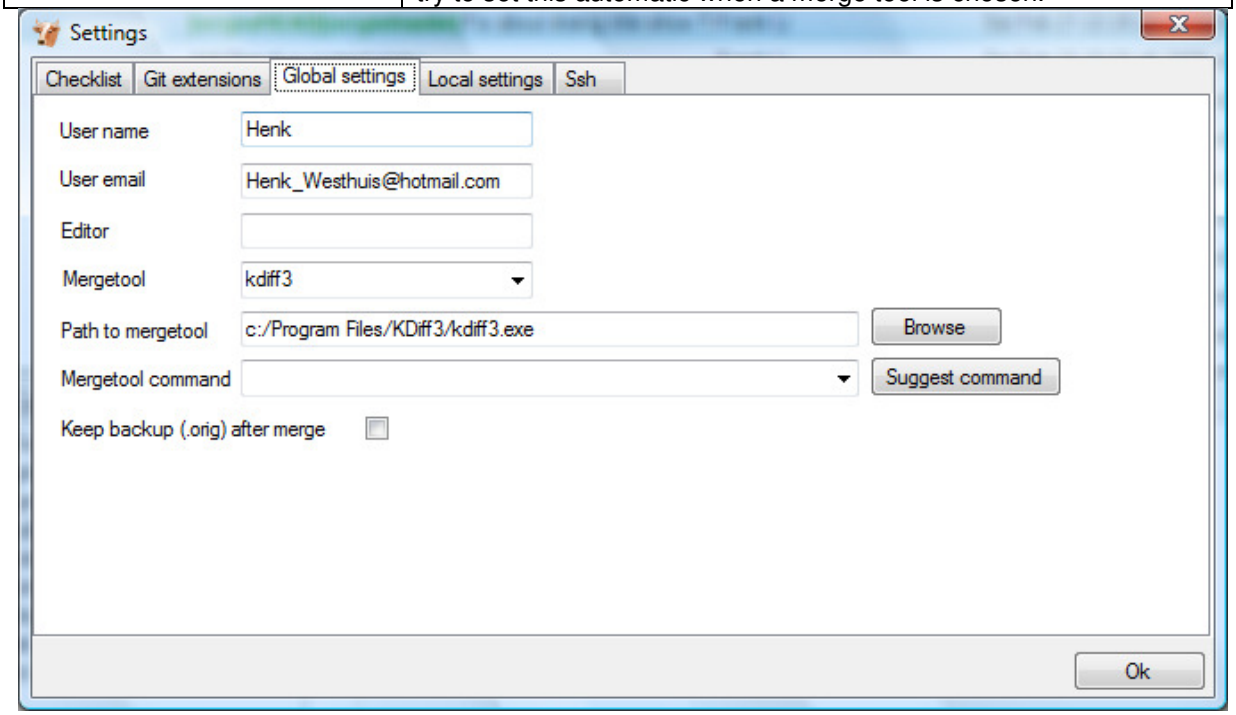
The 'Git Extension' tab contains all settings needed for Git Extension to run properly. The path to git.cmd and git.exe can be set here. This is only needed when these are not in the system path.

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.

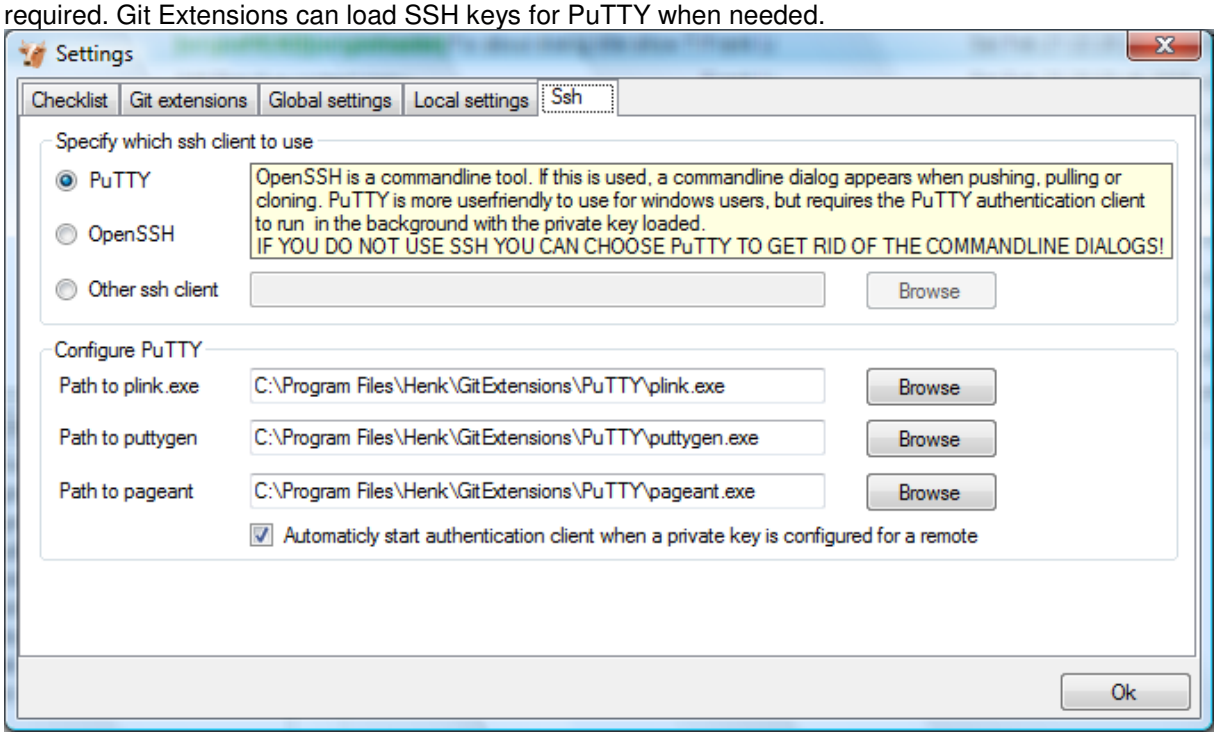


In the 'Global settings' tab some global Git settings can be set.

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.

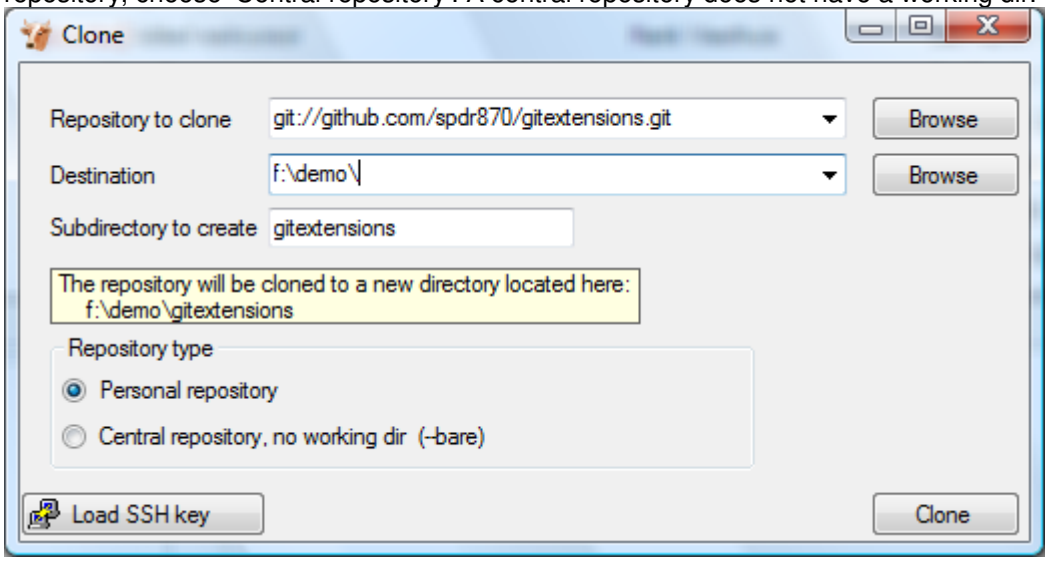


In the tab ‘SSH’ you can configure the SSH client you want Git to use. Git Extensions is optimized for PuTTY. Git Extensions will show command line dialogs if you do not use PuTTY and user input is required. Git Extensions can load SSH keys for PuTTY when needed.



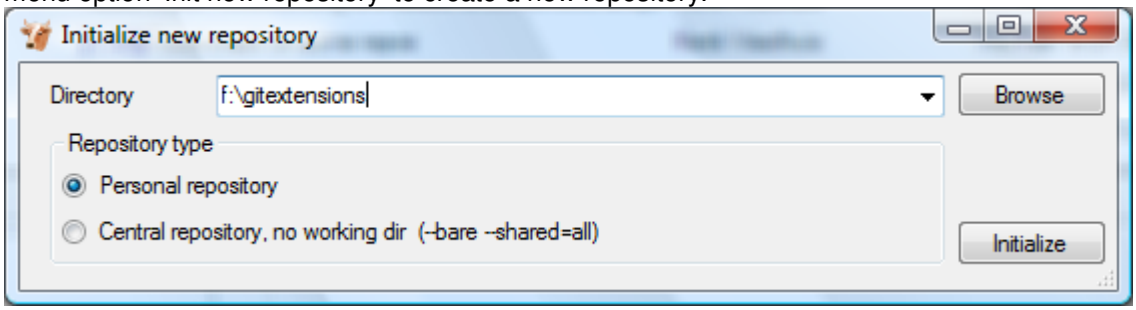
2.3 Clone existing repository

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 dir.



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.

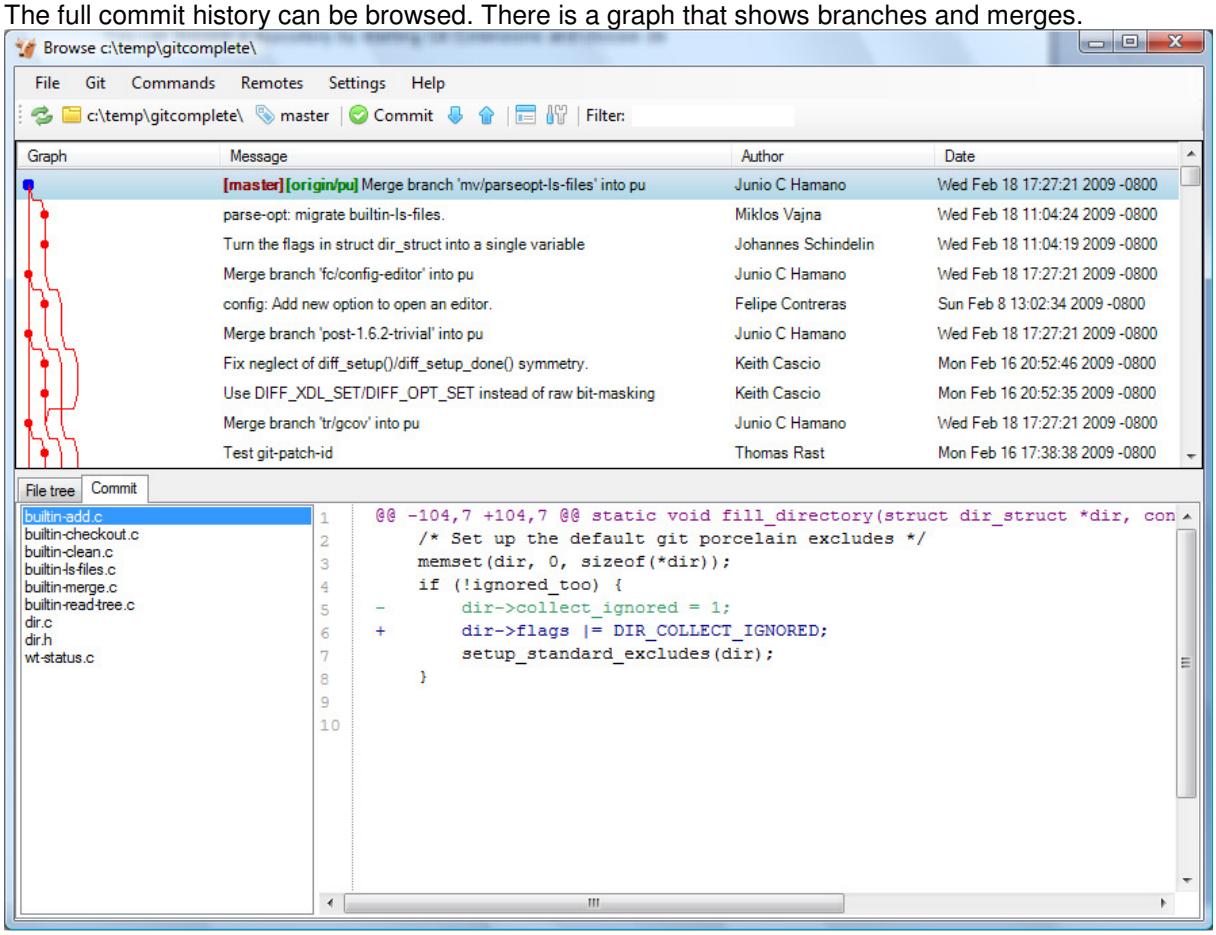


3 Browse repository

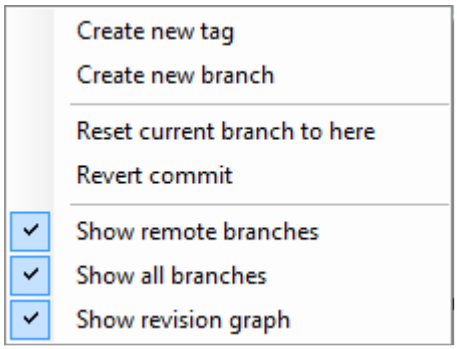
You can browse a repository by starting Git Extensions and select the repository to open. The main window contains the commit log. You can also open the 'Browse' window from the shell extensions and from the Visual Studio IDE.

3.1 View commit log

The full commit history can be browsed. There is a graph that shows branches and merges.

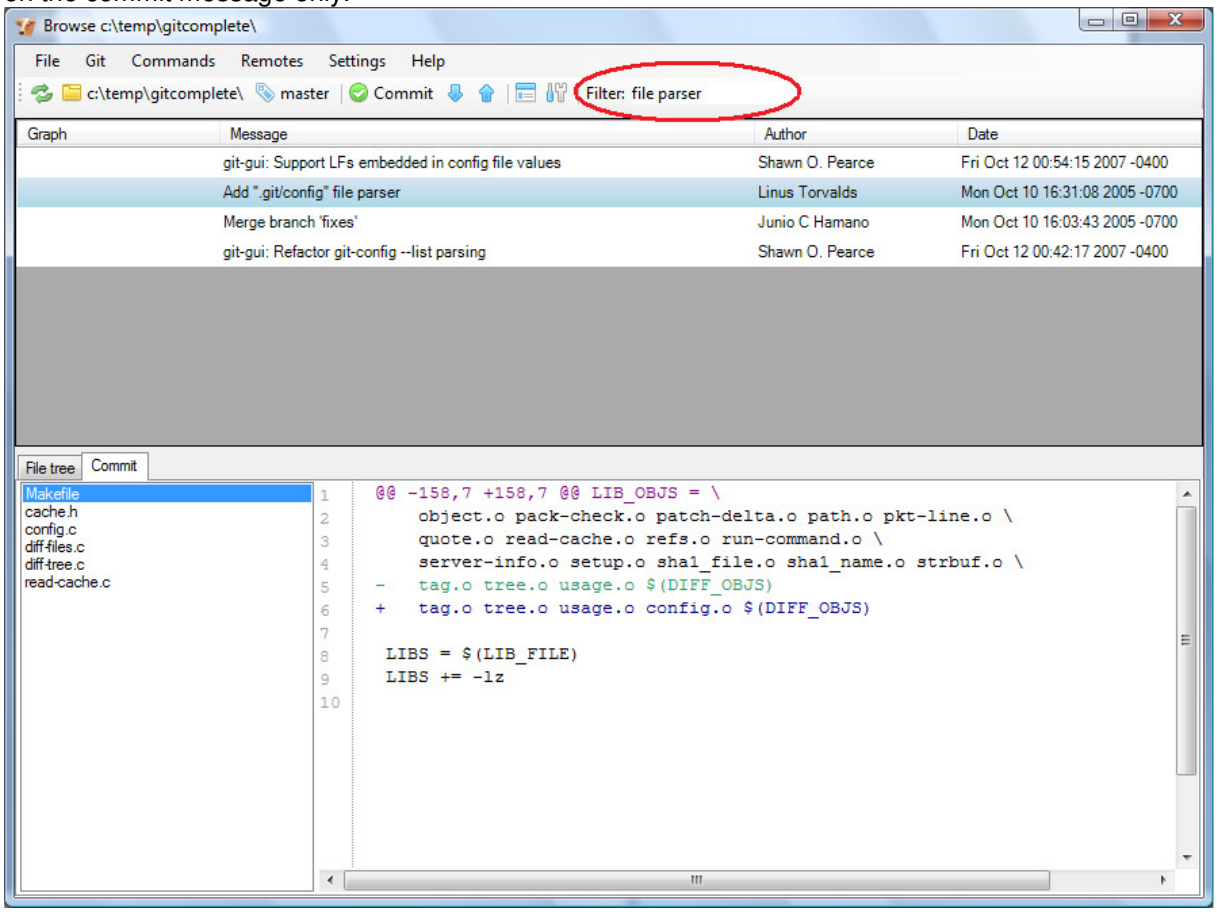


In the context menu of the commit log you can enable or disable the revision graph. You can also choose to only show the current branch instead of showing all branches. The other options will be discussed later.



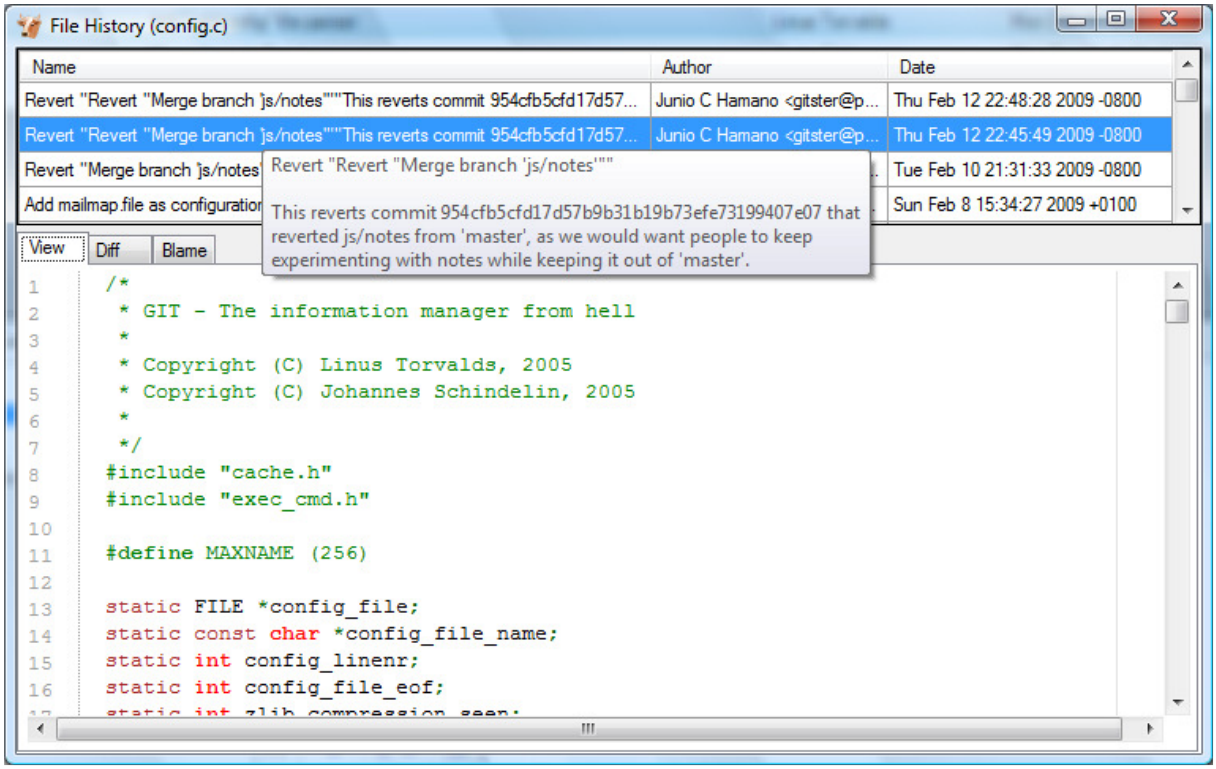
3.2 Search history

The history can be searched using regular expressions are basic search terms. There will be filtered on the commit message only.

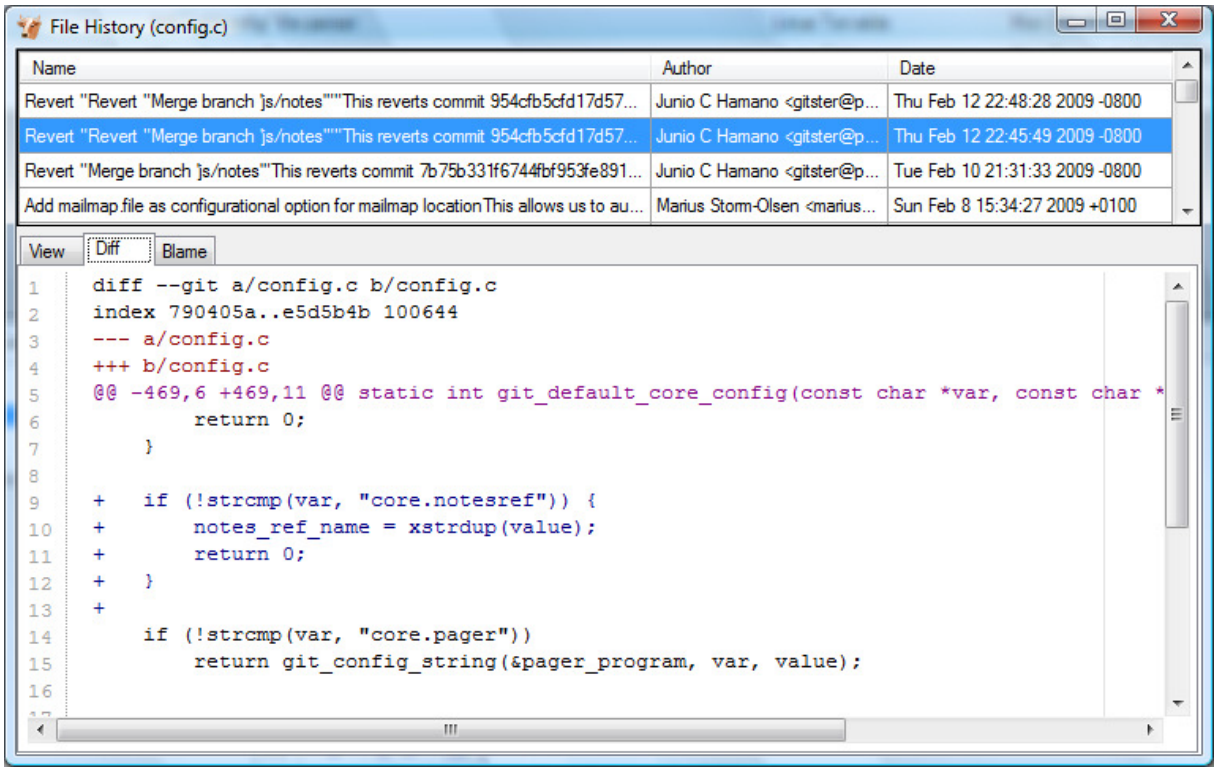


3.3 Singe file history

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.

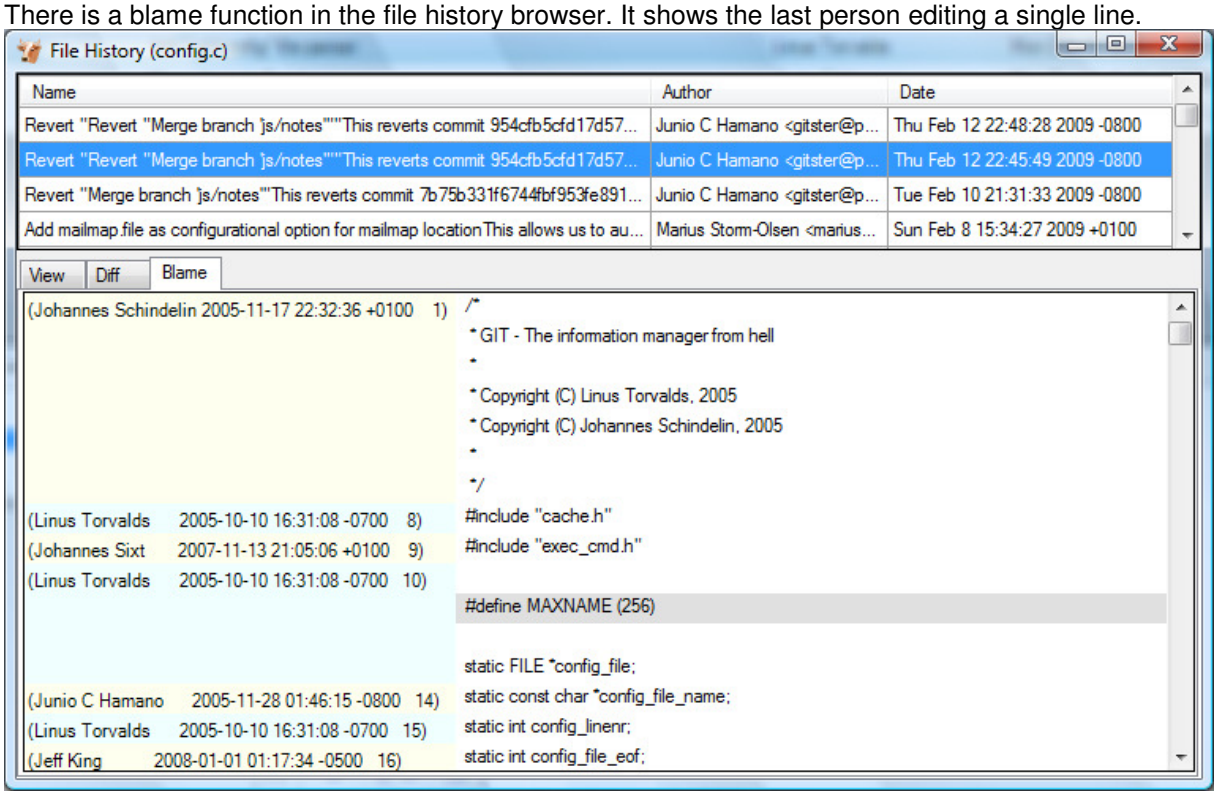


You can view the difference report from the commit in the ‘Diff’ tab. Added lines are marked with a ‘+’, removed lines are marked with a ‘–‘.

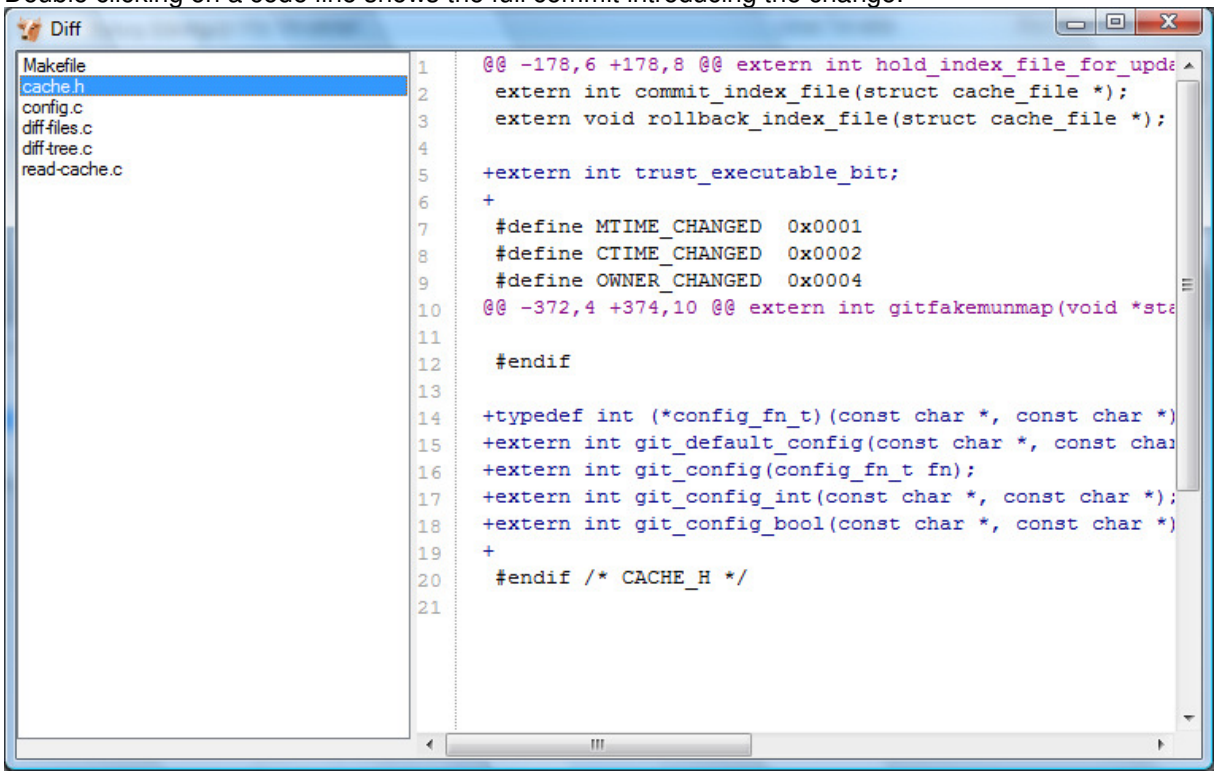


3.4 Blame

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

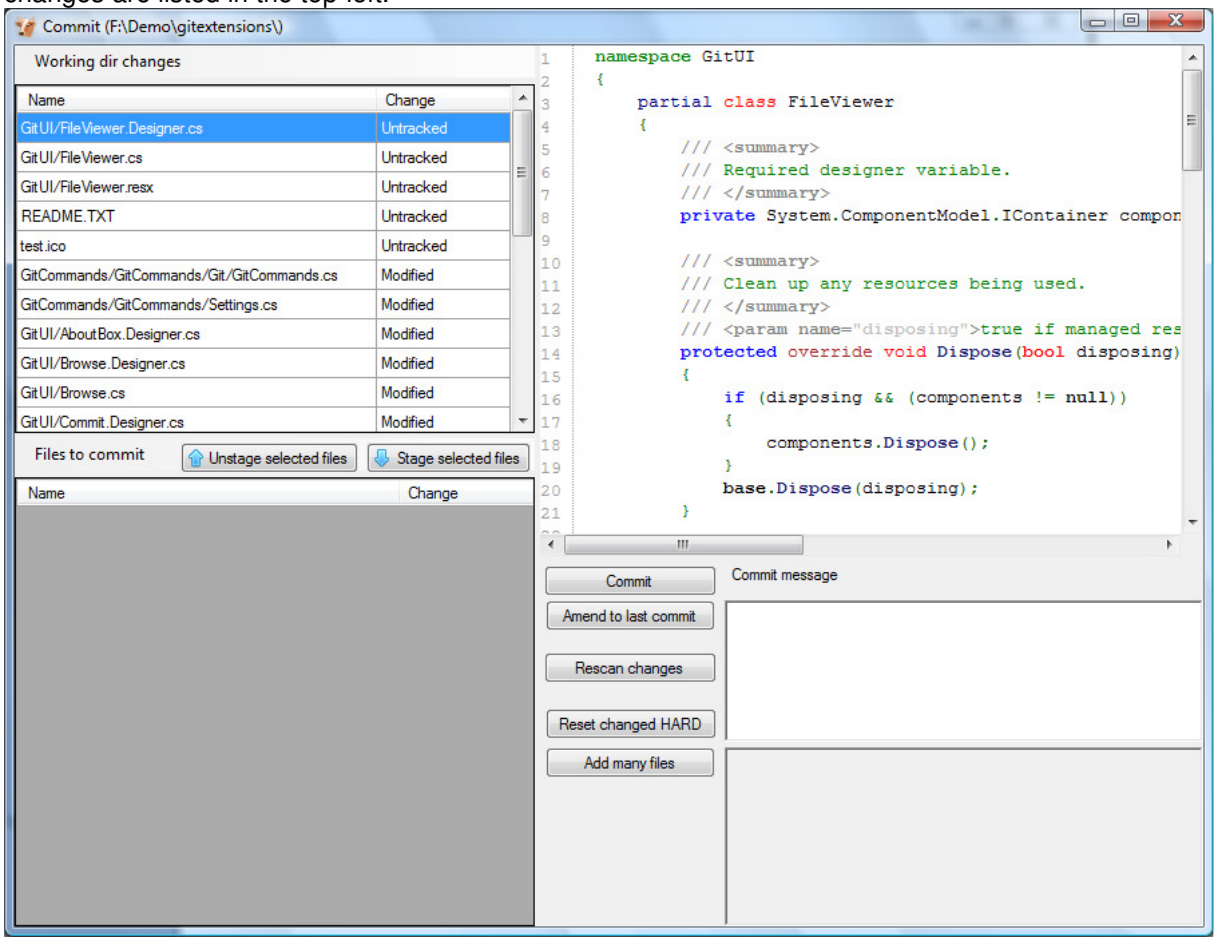
A commit is a set of changes with some extra information. Every commit contains the follow information:

- 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.

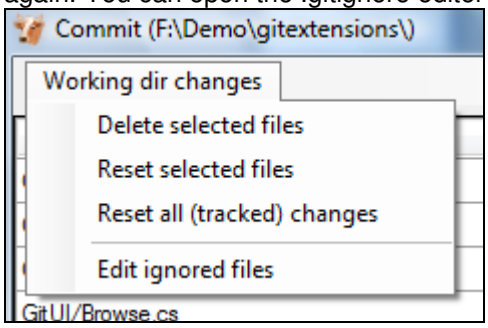


There are three kinds of changes:

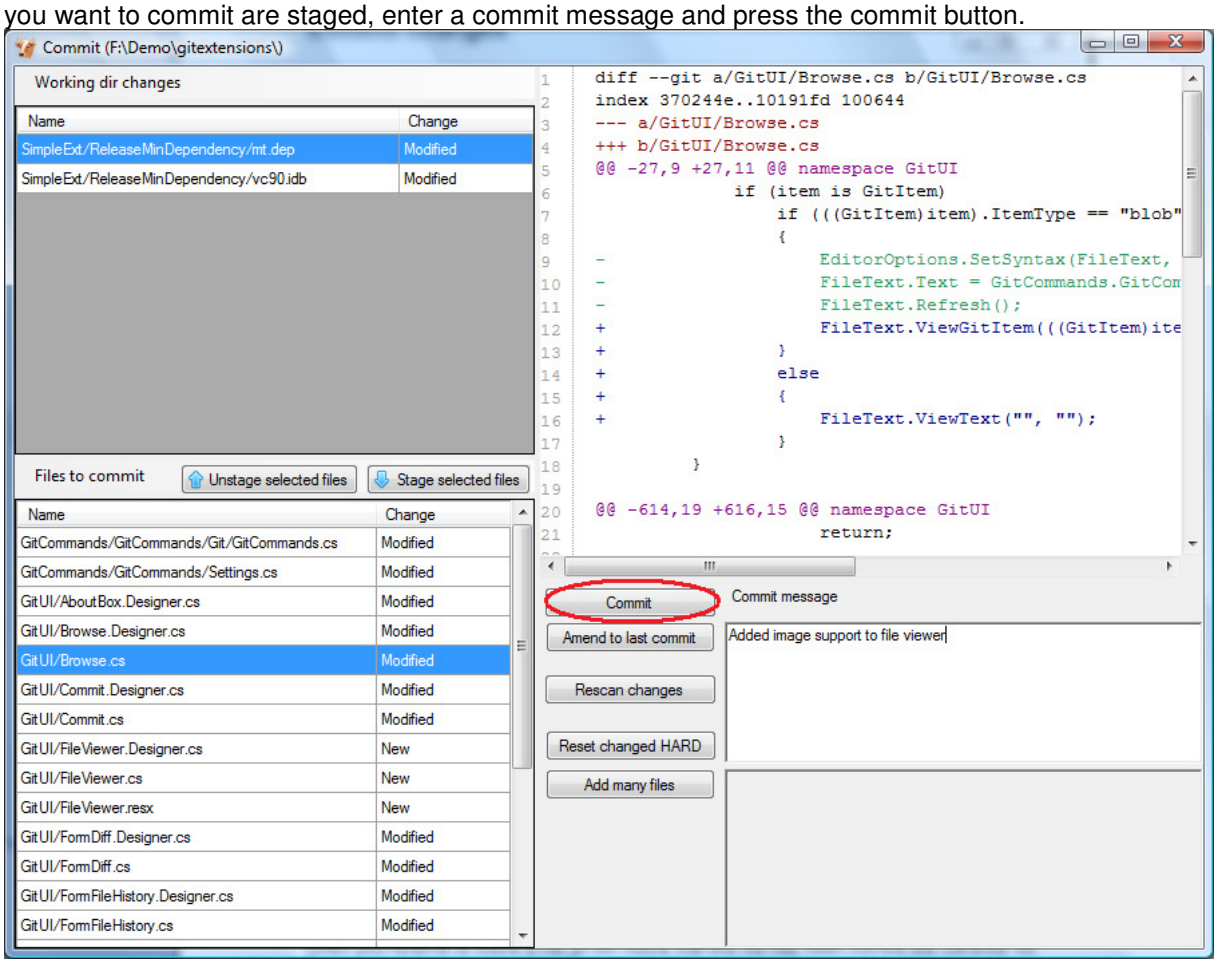
Untracked	This file is not yet tracked by Git. This is probably a new file, or a file that has not been committed to Git before.
Modified	This file is modified since the last commit.
Deleted	This file has been deleted.

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.

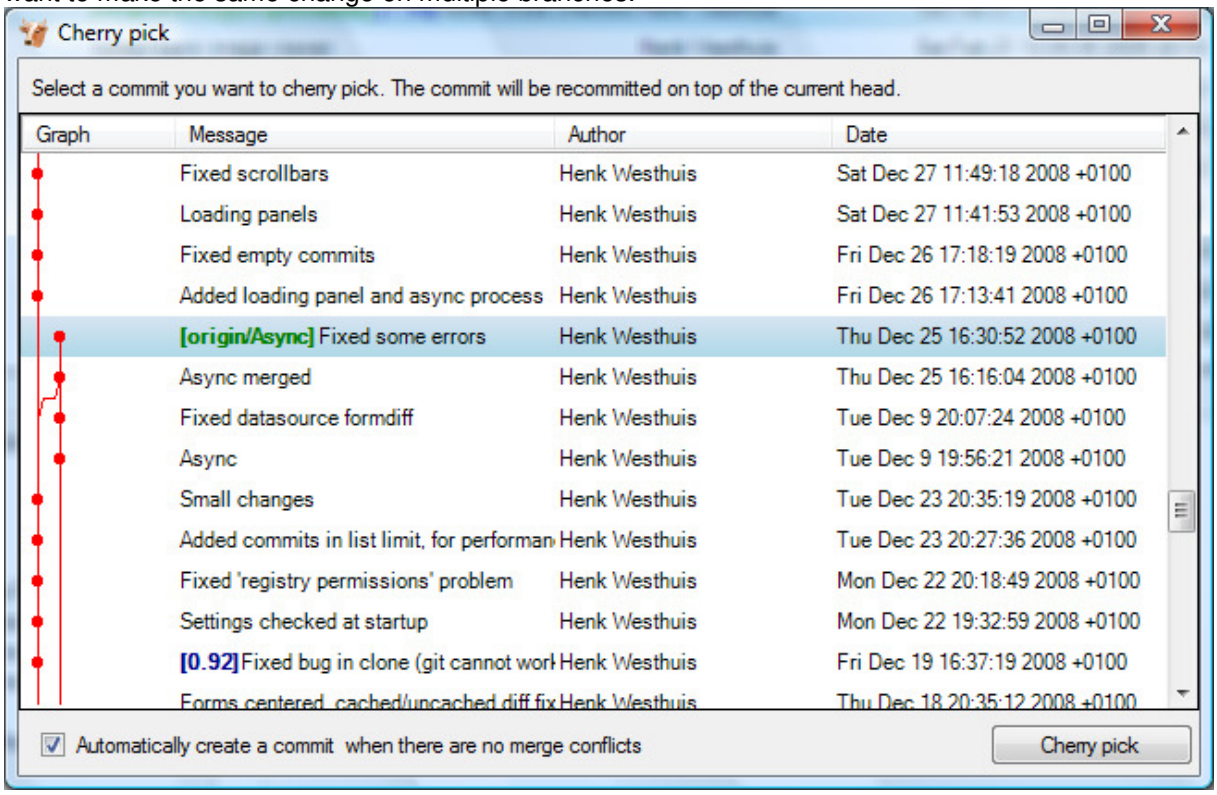


It is also possible to add files to you last commit using the 'Amend to last commit' button. This can be very useful when you forgot some changes. This function rewrites history; it deletes the last commit

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.

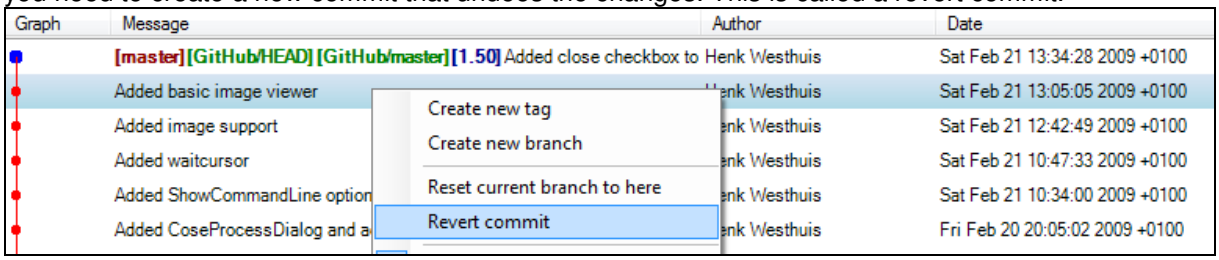
4.2 Cherry pick commit

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.



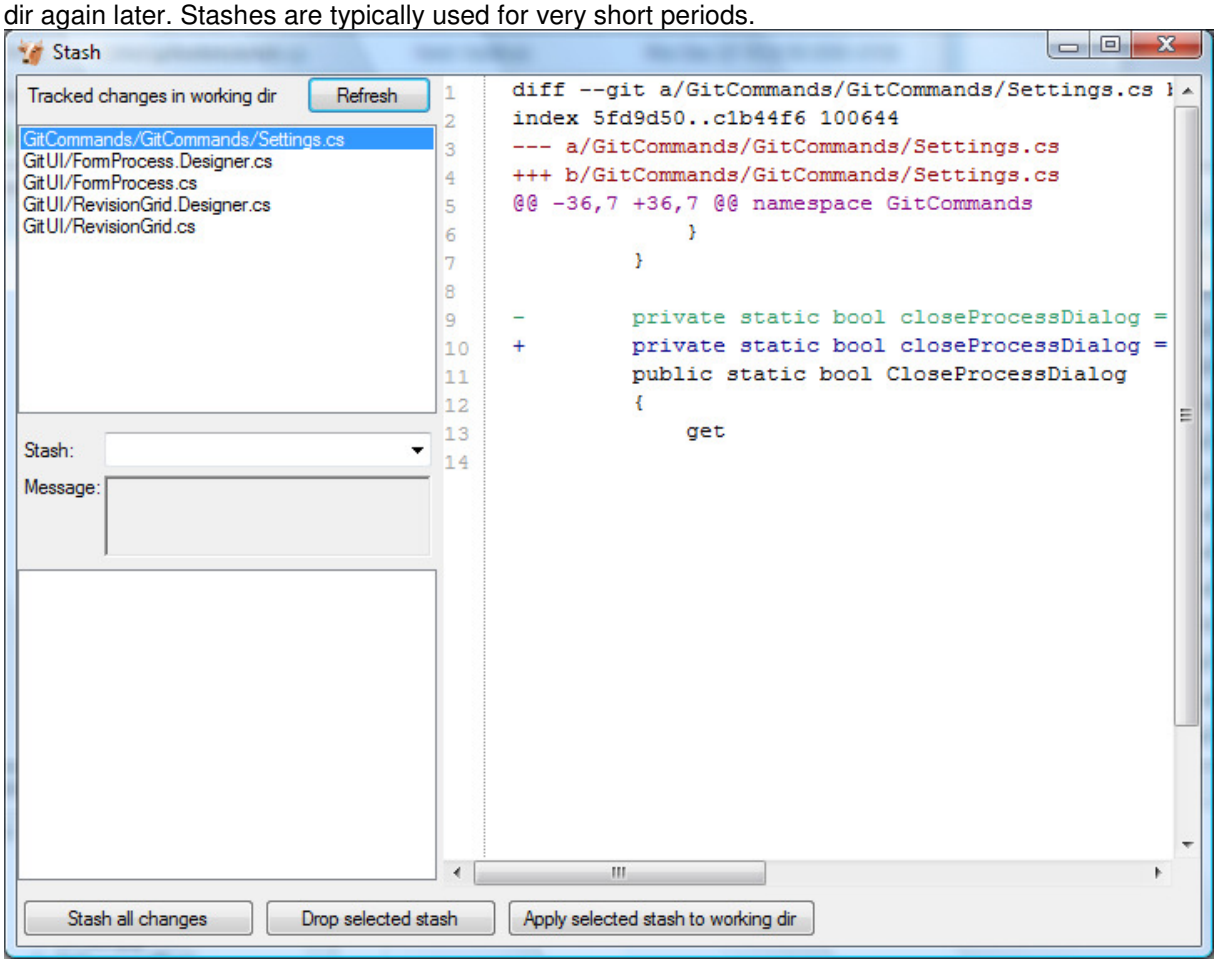
4.3 Revert commit

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.

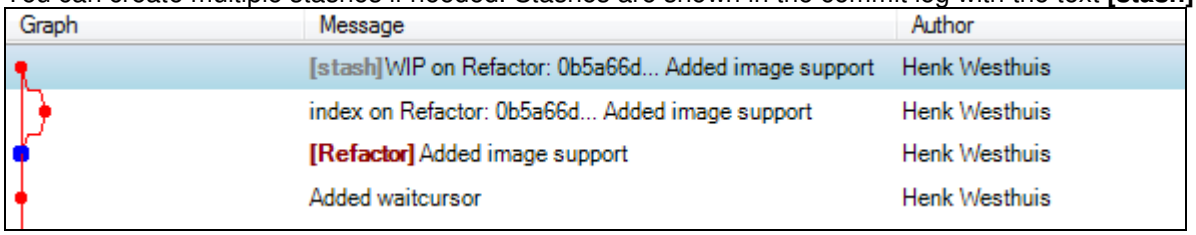


4.4 Stash changes

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.



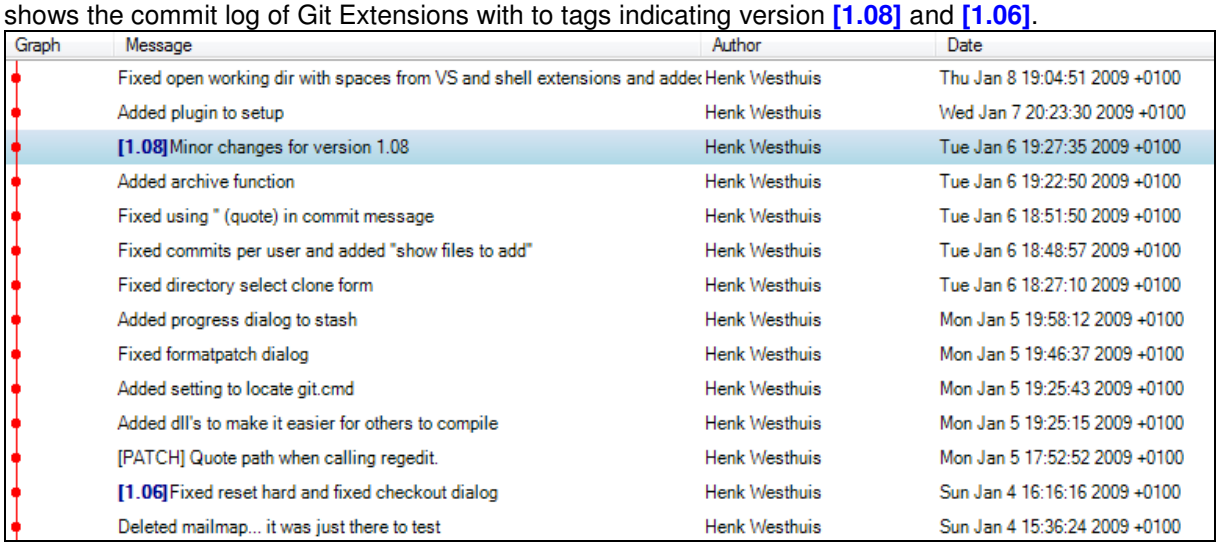
You can create multiple stashes if needed. Stashes are shown in the commit log with the text **[stash]**.



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

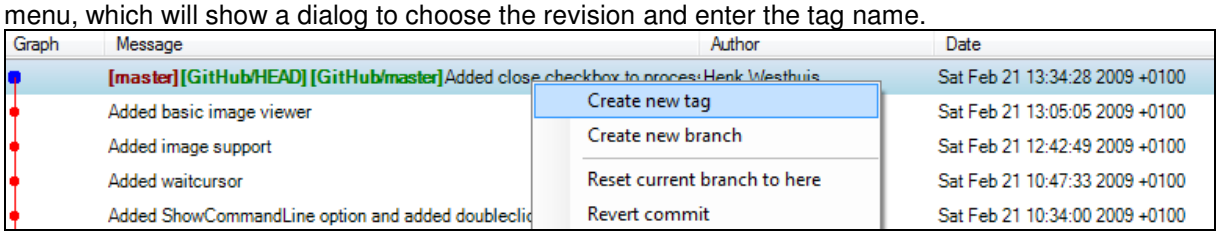
Tags are used to mark a specific version. Usually a tag will not be moved anymore. The image below shows the commit log of Git Extensions with to tags indicating version **[1.08]** and **[1.06]**.



Graph	Message	Author	Date
	Fixed open working dir with spaces from VS and shell extensions and adde	Henk Westhuis	Thu Jan 8 19:04:51 2009 +0100
	Added plugin to setup	Henk Westhuis	Wed Jan 7 20:23:30 2009 +0100
	[1.08] Minor changes for version 1.08	Henk Westhuis	Tue Jan 6 19:27:35 2009 +0100
	Added archive function	Henk Westhuis	Tue Jan 6 19:22:50 2009 +0100
	Fixed using " (quote) in commit message	Henk Westhuis	Tue Jan 6 18:51:50 2009 +0100
	Fixed commits per user and added "show files to add"	Henk Westhuis	Tue Jan 6 18:48:57 2009 +0100
	Fixed directory select clone form	Henk Westhuis	Tue Jan 6 18:27:10 2009 +0100
	Added progress dialog to stash	Henk Westhuis	Mon Jan 5 19:58:12 2009 +0100
	Fixed formatpatch dialog	Henk Westhuis	Mon Jan 5 19:46:37 2009 +0100
	Added setting to locate git.cmd	Henk Westhuis	Mon Jan 5 19:25:43 2009 +0100
	Added dll's to make it easier for others to compile	Henk Westhuis	Mon Jan 5 19:25:15 2009 +0100
	[PATCH] Quote path when calling regedit.	Henk Westhuis	Mon Jan 5 17:52:52 2009 +0100
	[1.06] Fixed reset hard and fixed checkout dialog	Henk Westhuis	Sun Jan 4 16:16:16 2009 +0100
	Deleted mailmap... it was just there to test	Henk Westhuis	Sun Jan 4 15:36:24 2009 +0100

5.1 Create tag

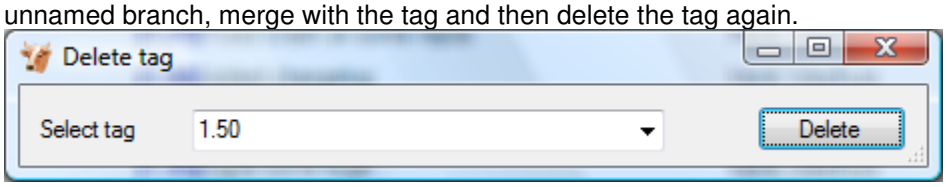
In Git Extensions you can tag a revision by choosing 'Create new tag' in the commit log context menu. A dialog will prompt for the name of the tag. You can also choose 'Create tag' from the 'Commands' menu, which will show a dialog to choose the revision and enter the tag name.



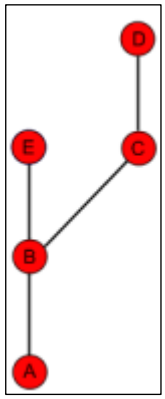
Once a tag is created, it cannot be moved again. You need to delete the tag and create it again to move it.

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.



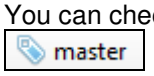
6 Branches



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.

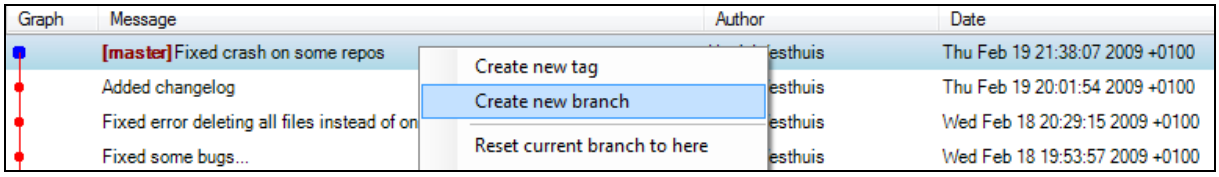
In Git branches are created very often. Creating a branch is very easy to do and it is recommended to create a branch very often. In fact, when you make a commit to a cloned repository you start a new branch. I will explain this in the pull chapter.

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

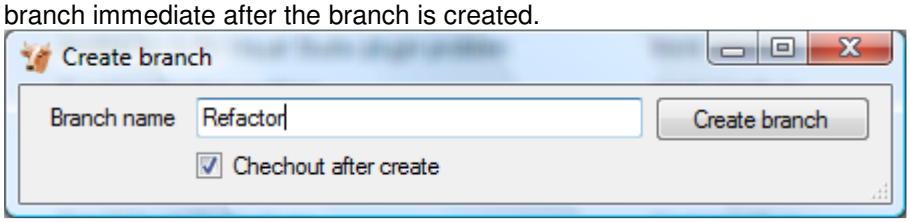


6.1 Create branch

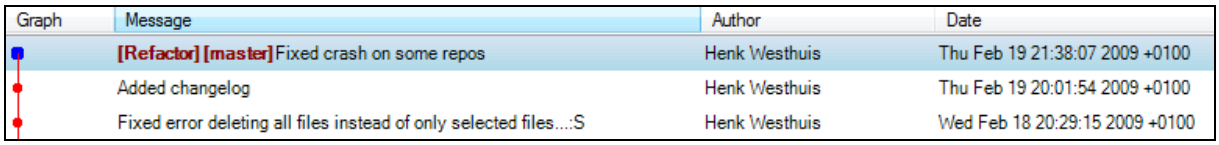
In Git Extensions there are multiple ways to create a new branch. In the image below I create a new branch from the context menu in the commit log. This will create a new branch on the revision that is selected.



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.



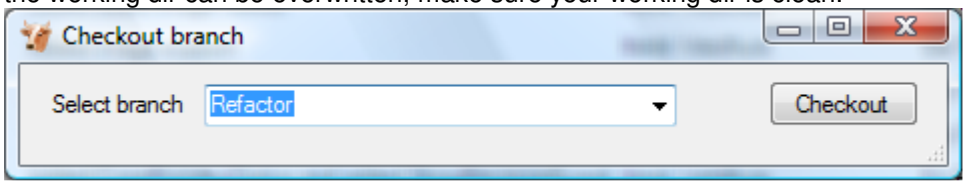
When the branch is created you will see the new branch **[Refactor]** in the commit log. If you chose to checkout this branch the next commit will be committed to the new branch.



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 workflow of Git is optimized for branching and merging.

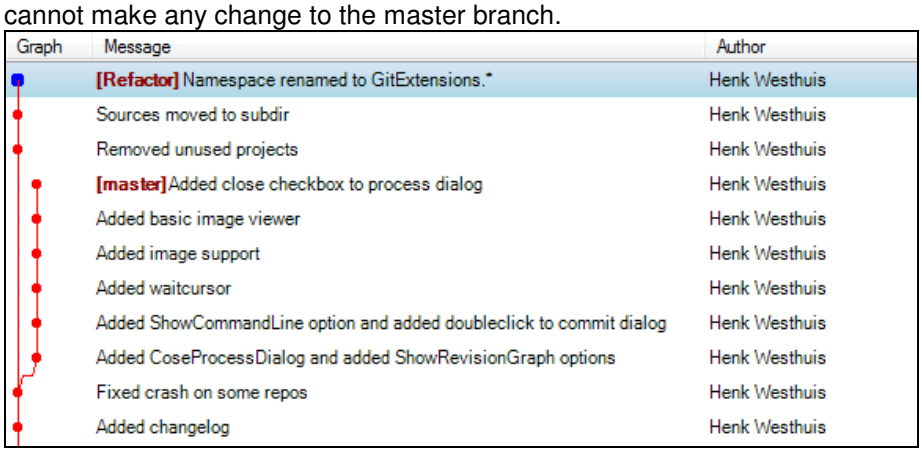
6.2 Checkout branch

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 dir. Uncommitted changes in the working dir can be overwritten, make sure your working dir is clean.

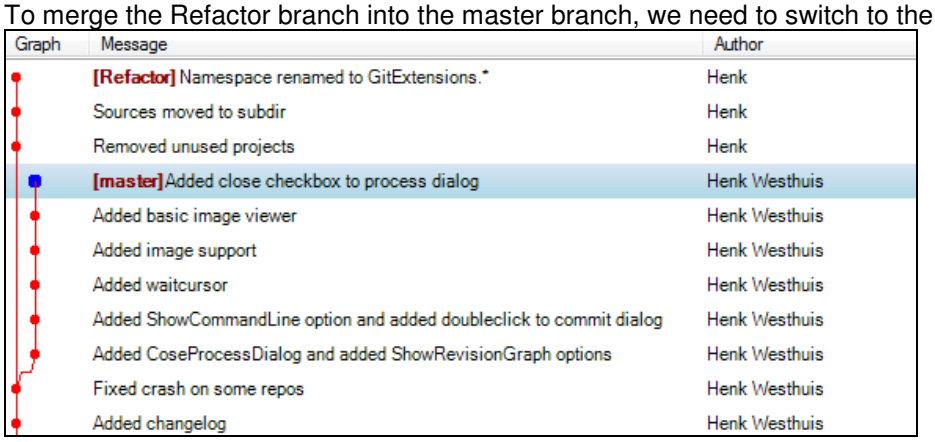


6.3 Merge branches

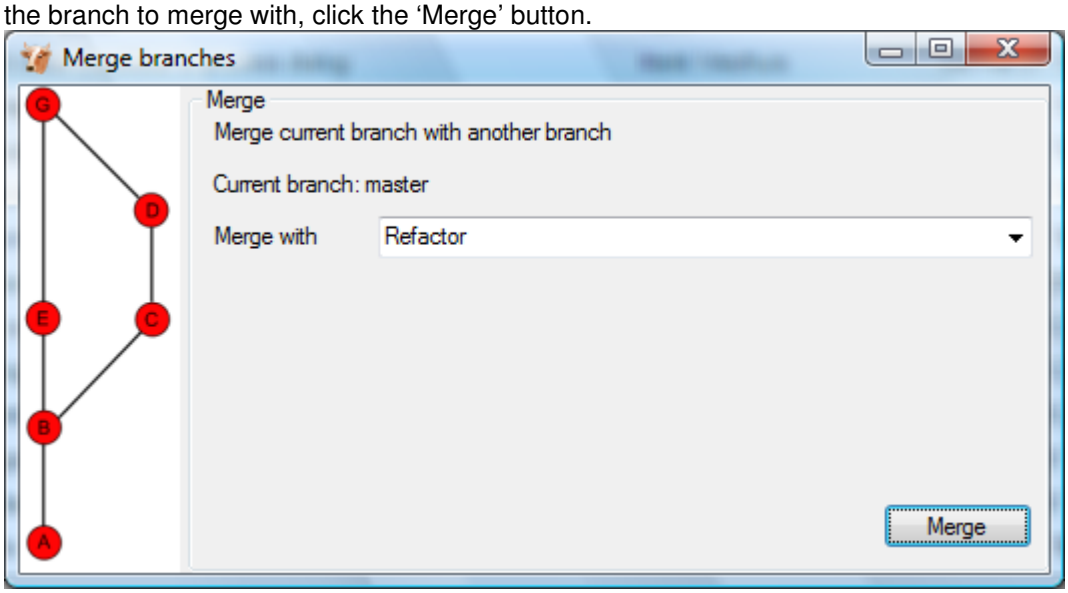
In the image below there are two branches, **[Refactor]** and **[master]**. We can merge the commits from the master branch into the Refactor. If we do this, the Refactor branch will be up to date with the master branch, but not the other way around. As long as we are working on the Refactor branch we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



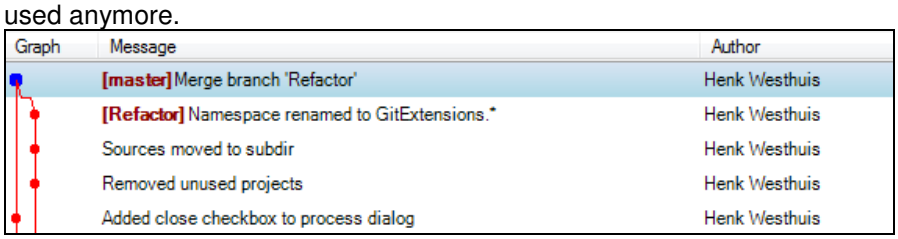
To merge the Refactor branch into the master branch, we need to switch to the master branch first.



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.



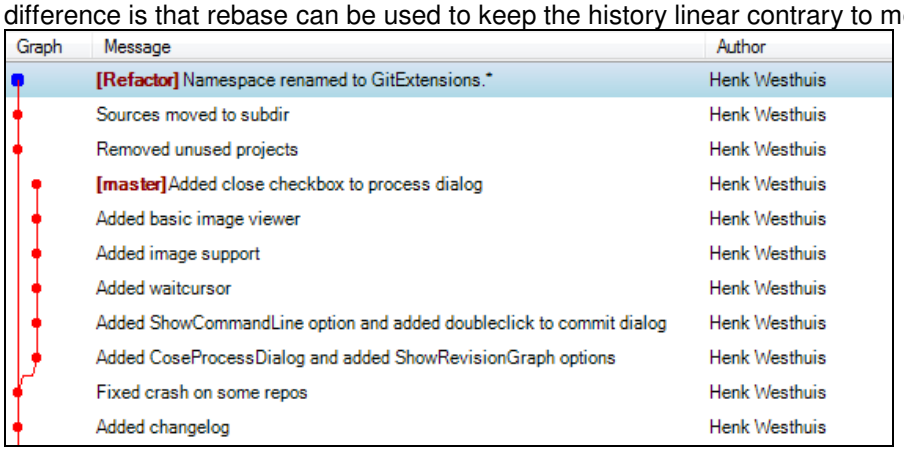
After the merge the commit log will show the new commit containing the merge. Notice that the Refactor branch is not changed by this merge. If you want to continue working on the Refactor branch you can merge the Refactor branch with master. You could also delete the Refactor branch if it is not used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

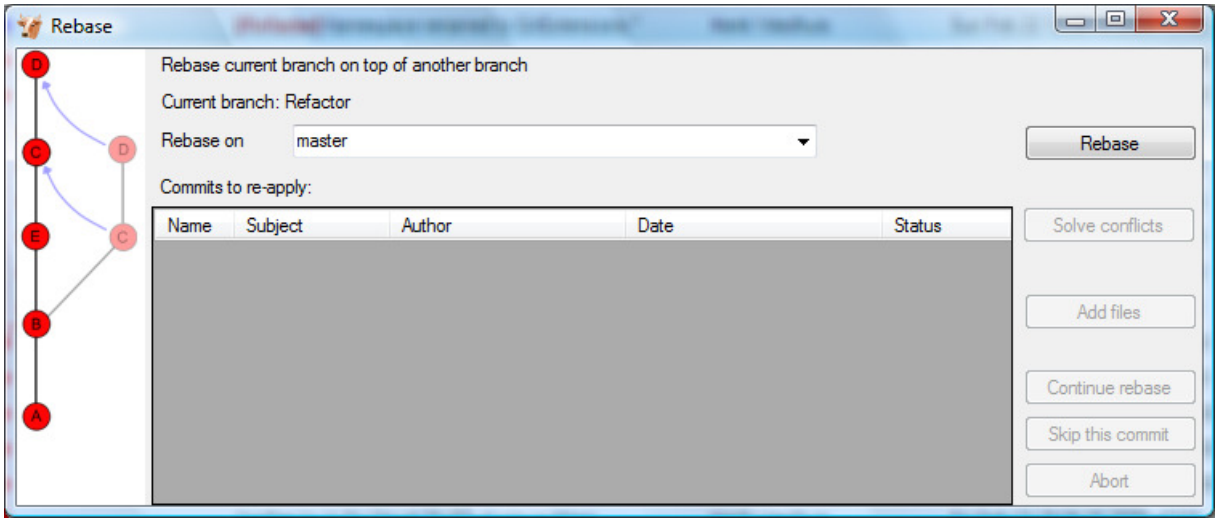
The rebase command is the most complex command in Git. The rebase command is very similar to the merge command. Both rebase and merge are used to get a branch up-to-date. The main difference is that rebase can be used to keep the history linear contrary to merges.



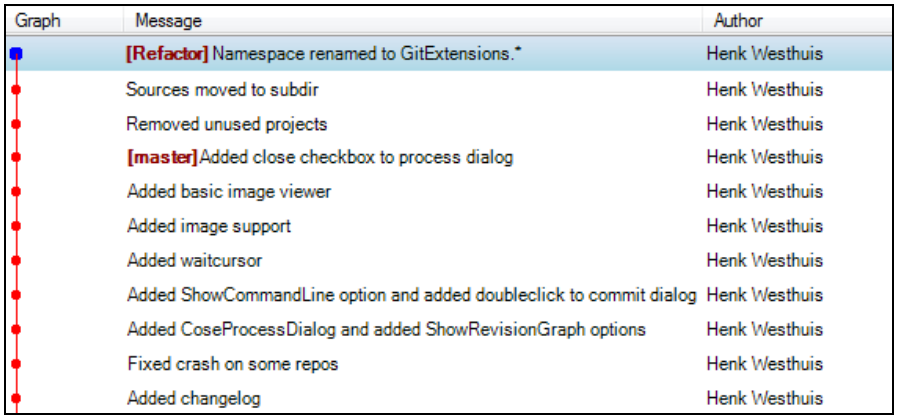
A rebase of Refactor on top of master will perform the following actions:

- 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.

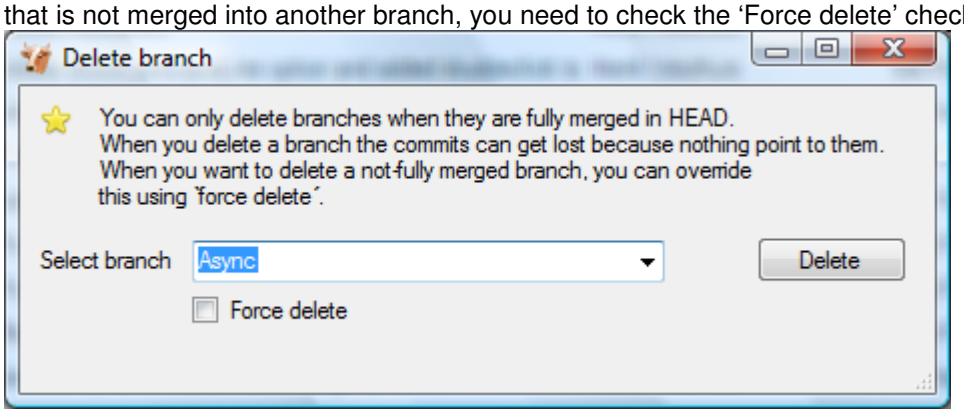


Because this function rewrites history you should only use this on branches that are not published to other repositories yet. When you rebase a branch that is already pushed it will be harder to pull or push to that remote. If you want to get a branch up-to-date that is already published you should merge.

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.



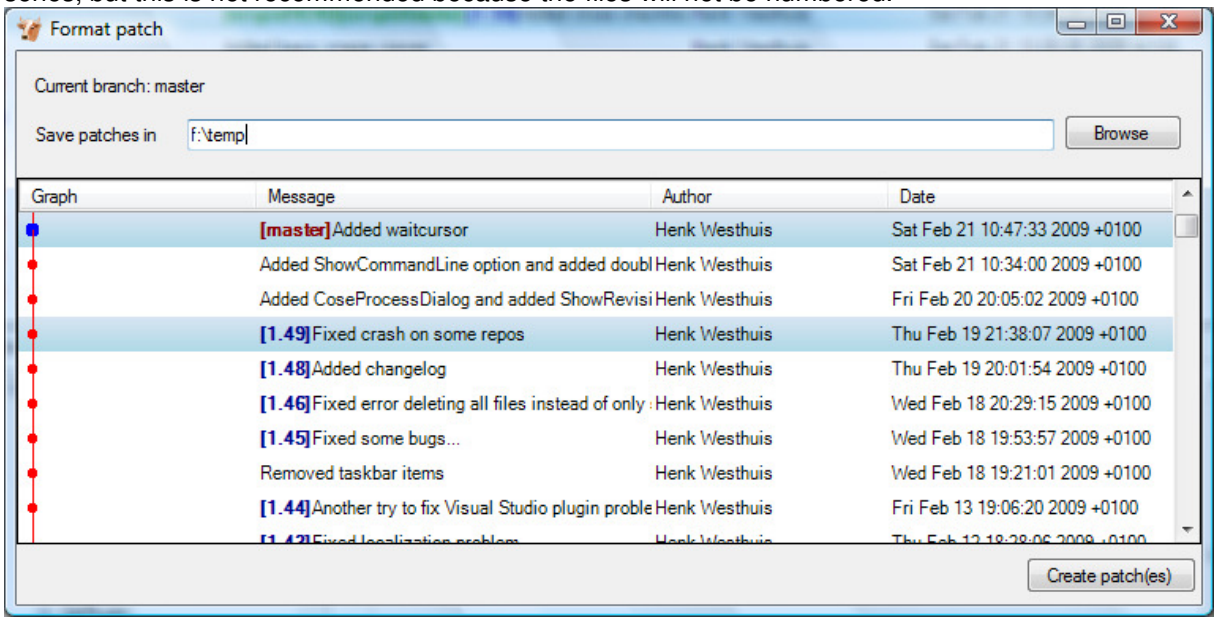
7 Patches

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.

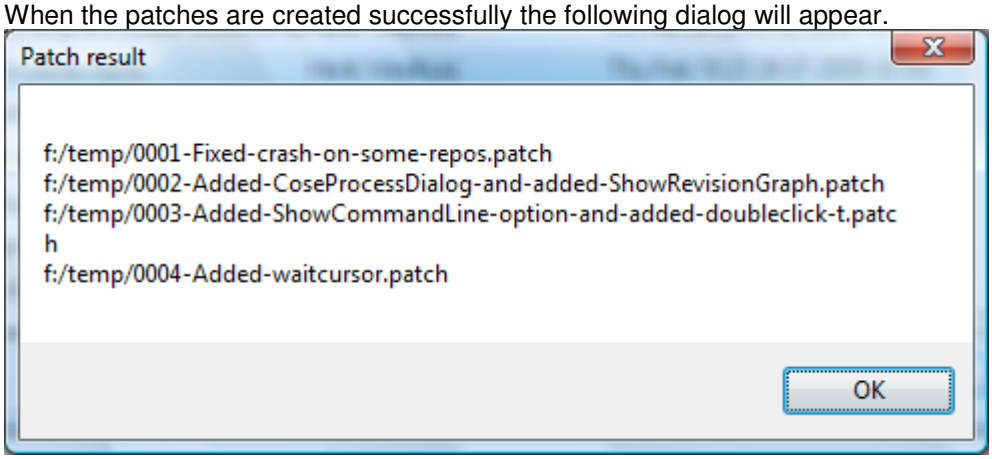
```
From 58c02ec4701c94c671a41e1e5d50c582e859851f Mon Sep 17 00:00:00 2001
From: Russell King <rmk@dyn-67.arm.linux.org.uk>
Date: Sun, 17 Apr 2005 15:40:46 +0100
Subject: [PATCH 000213/123824] [PATCH] ARM: h3600_irda_set_speed arguments

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.

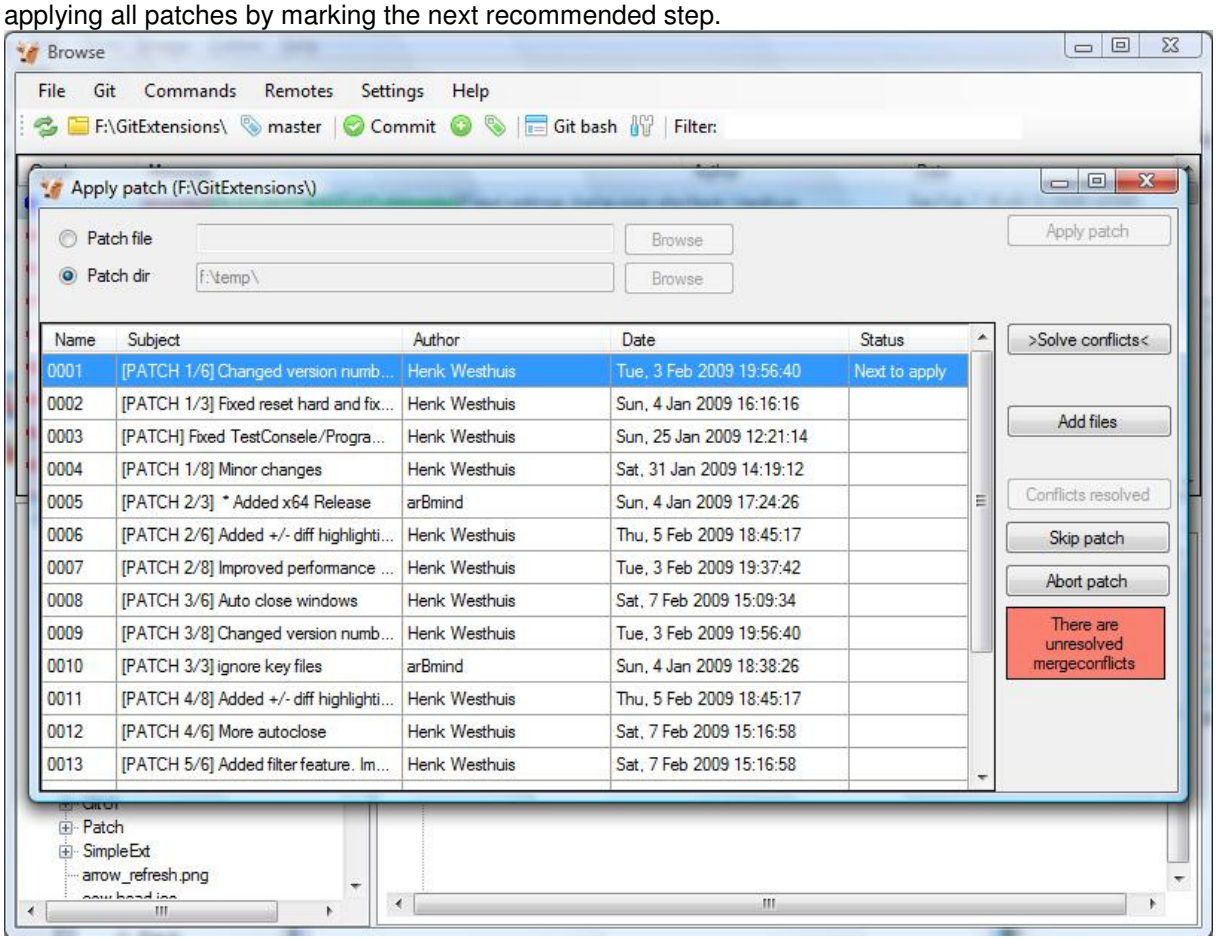


When the patches are created successfully the following dialog will appear.



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.

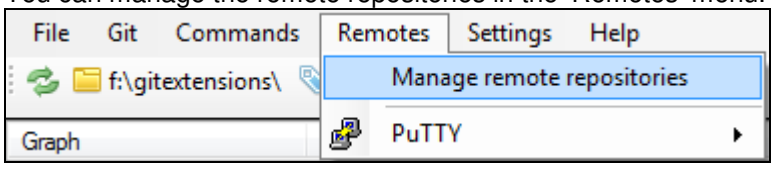


8 Remote features

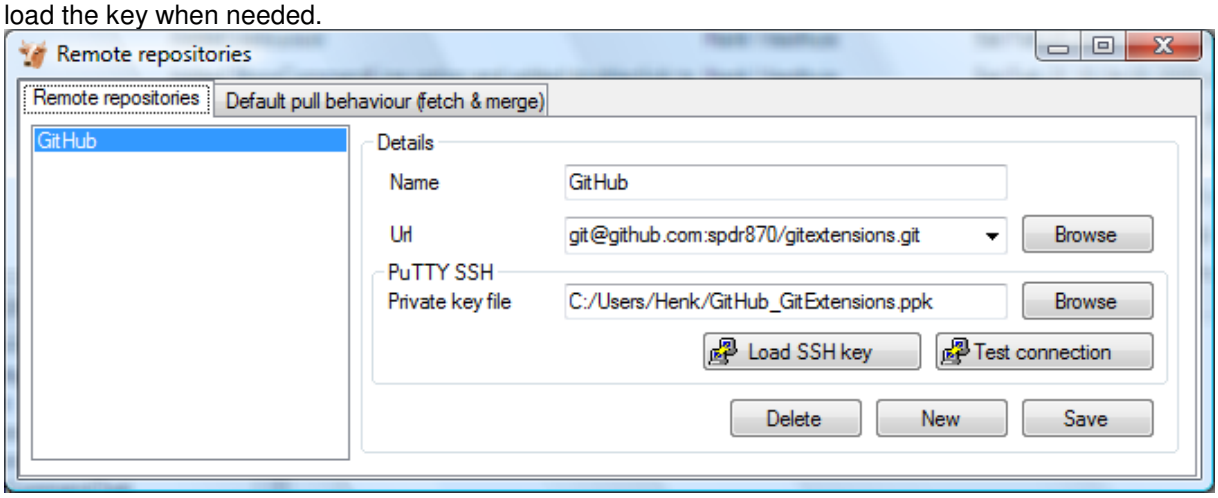
Git is a distributed source control management system. This means that all changes you make are local. When you commit changes, you only commit them to your local repository. To publish your local changes you need to push. In order to get changes committed by others, you need to pull.

8.1 Manage remote repositories

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

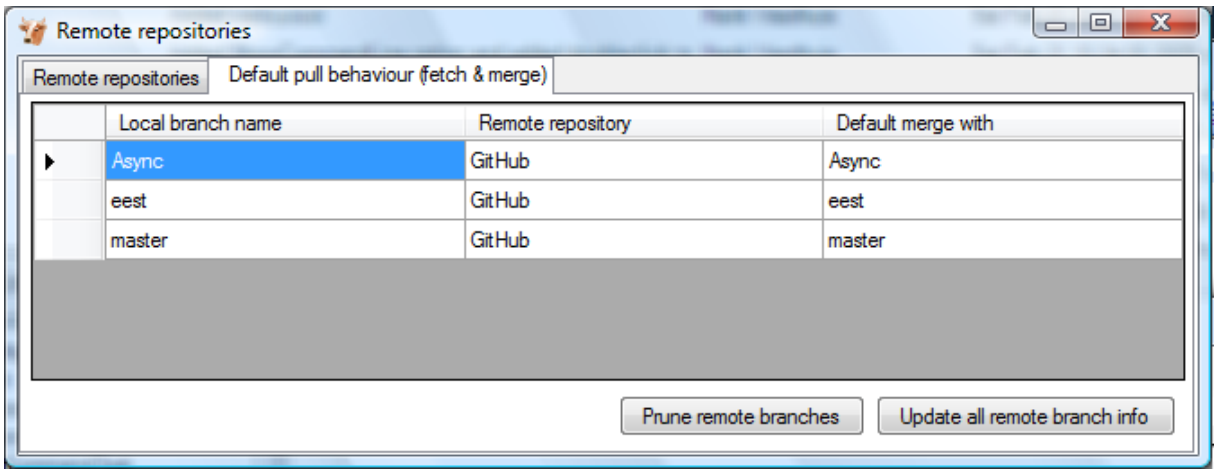


When you cloned your repository from a public repository, this remote is already configured. You can rename each remote for easy recognition. The default name after cloning a remote is 'origin'. If you use PuTTY as SSH client you can also enter the private key file for each remote. Git Extensions will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



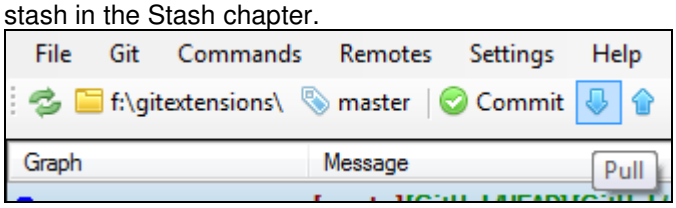
In the 'Default pull behaviour' tab you can configure the branches that need to be pulled and merged by default. If you configure this correctly you will not need to choose a branch when you pull or push. There are two buttons on this dialog:

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



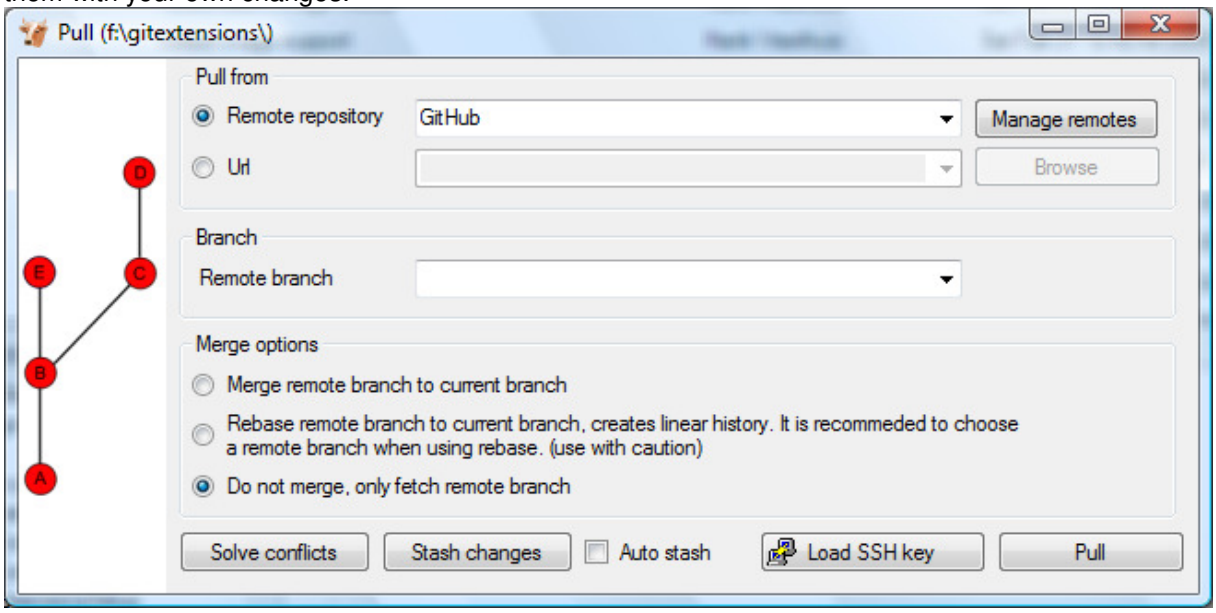
8.2 Pull changes

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.

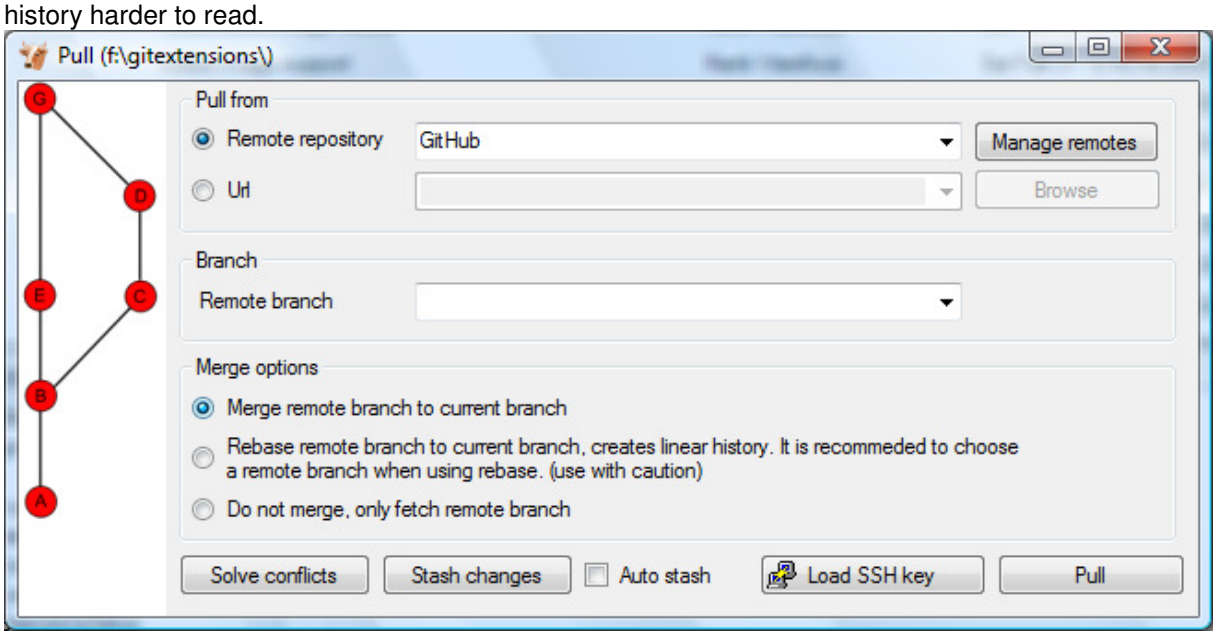


In order to get your personal repository up-to-date, you need to fetch changes from a remote repository. You can do this using the 'Pull' dialog. When the dialog starts the default remote for the current branch is set. You can choose another remote or enter a custom url if you like. When the remote branches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

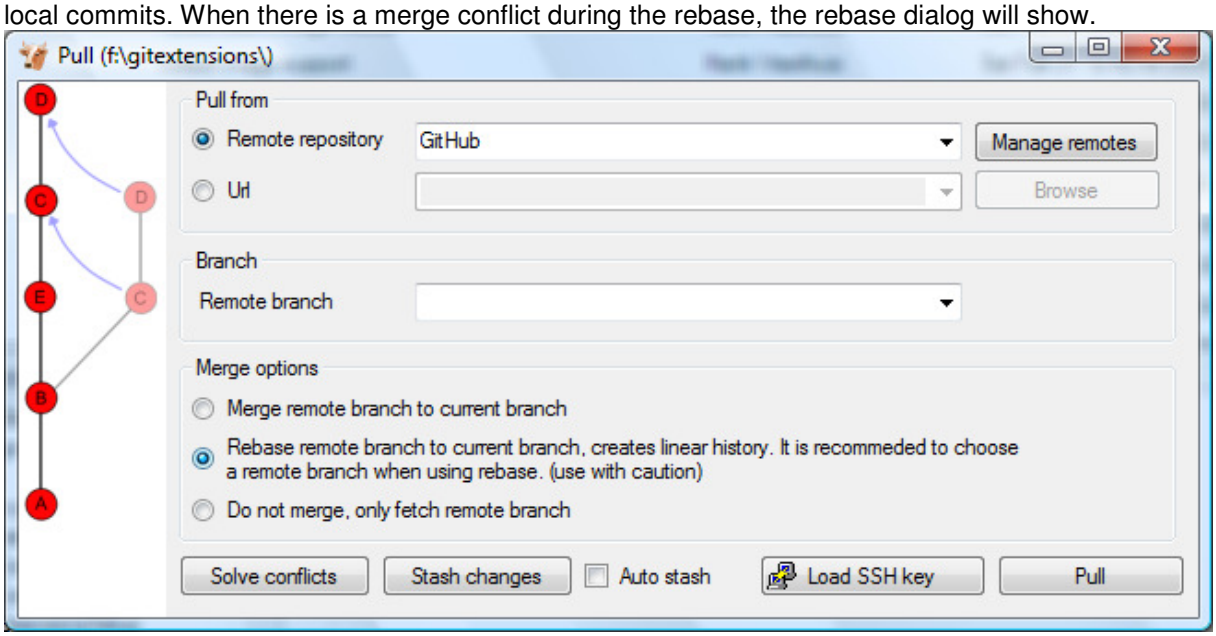
If you just fetch the commits from the remote repository and you already committed some changes to your local repository, the commits will be in a different branch. In the pull dialog this is illustrated in the image on the left. This can be useful when you want to review the changes before you want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

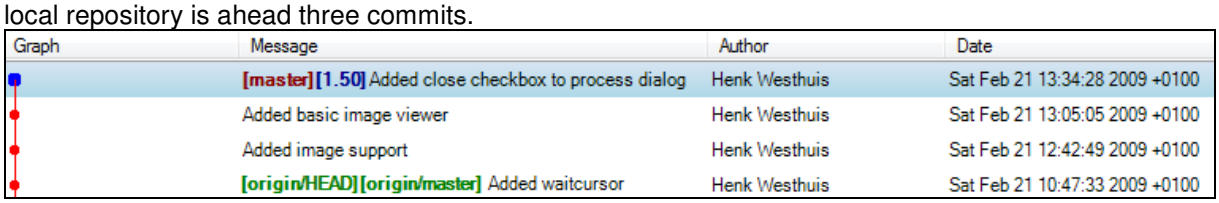


Next to the pull button there are some buttons that can be useful:

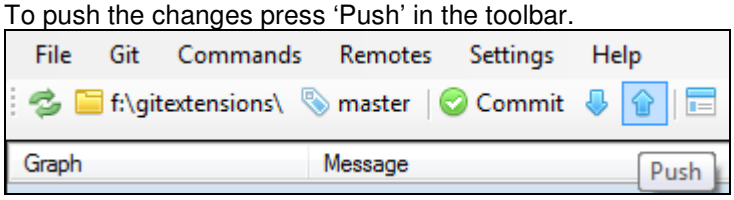
Solve conflicts	When there are merge conflicts, you can solve them by pressing this button.
Stash changes	When the working dir contains uncommitted changes, you need to stash them before pulling.
Auto stash	Check this checkbox if you want to stash before pulling. The stash will be reapplied after pulling.
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.

8.3 Push changes

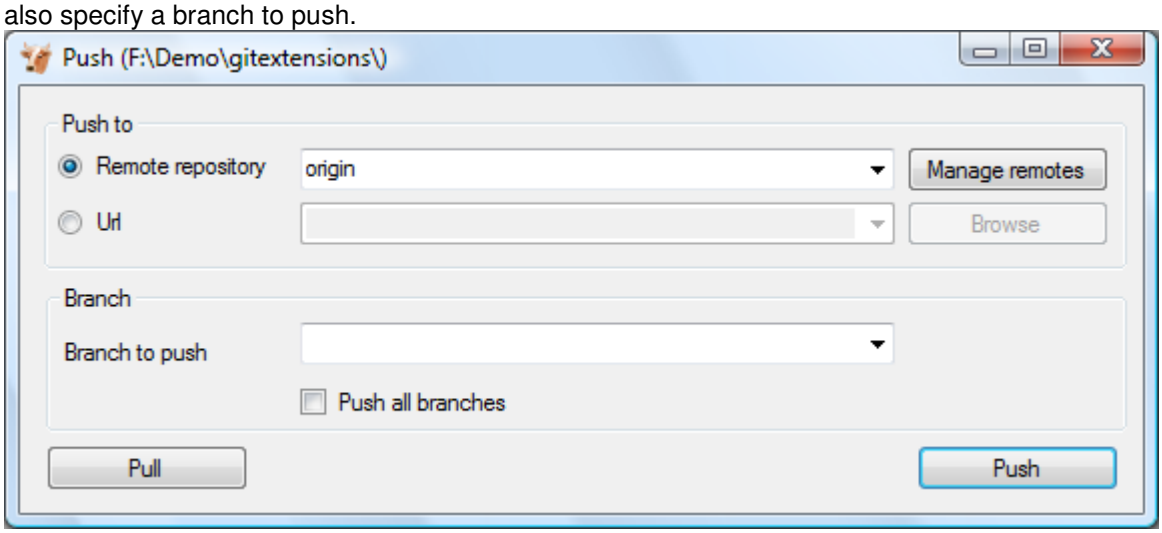
In the browse window you can check if there are local commits that are not pushed to a remote repository yet. In the image below the green labels mark the position of the master branch on the remote repository. The red label marks the position of the master branch on the local repository. The local repository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



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



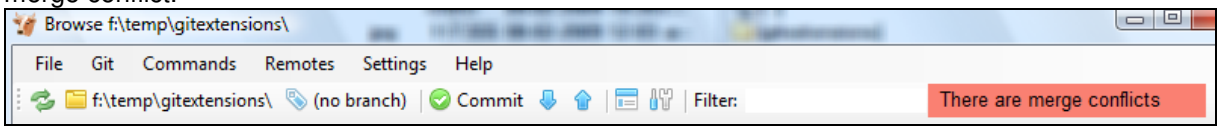
The push dialog allows you to choose the remote repository to push to. The remote repository is set to the remote of the current branch. You can choose another remote or choose a url to push to. You can also specify a branch to push.



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.

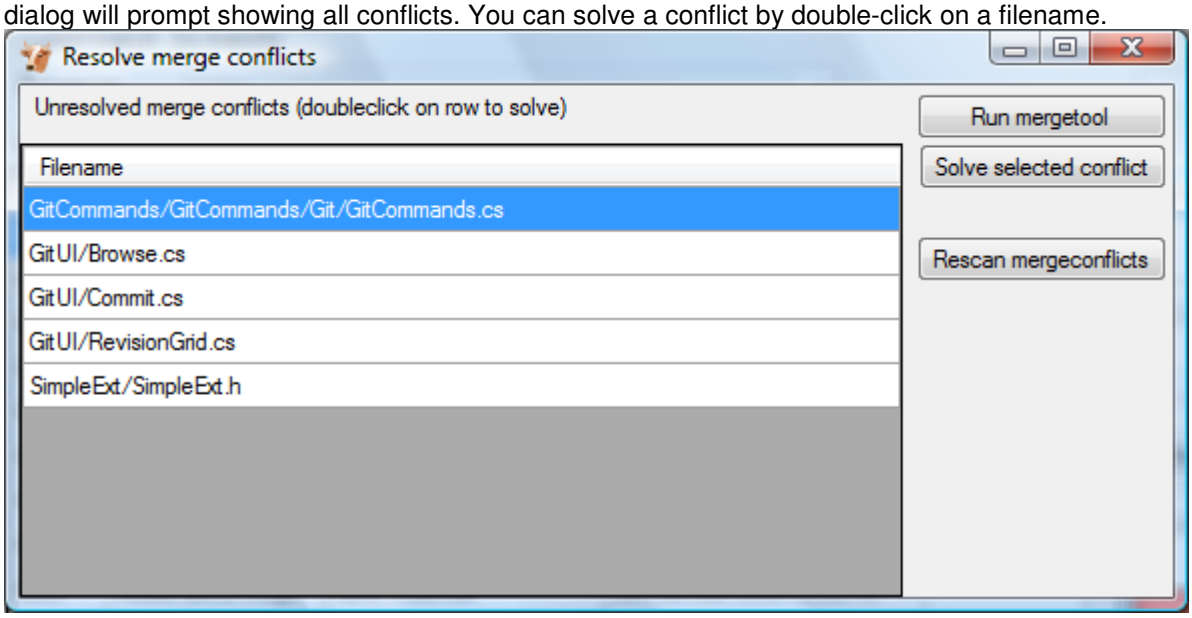
9 Merge conflicts

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:

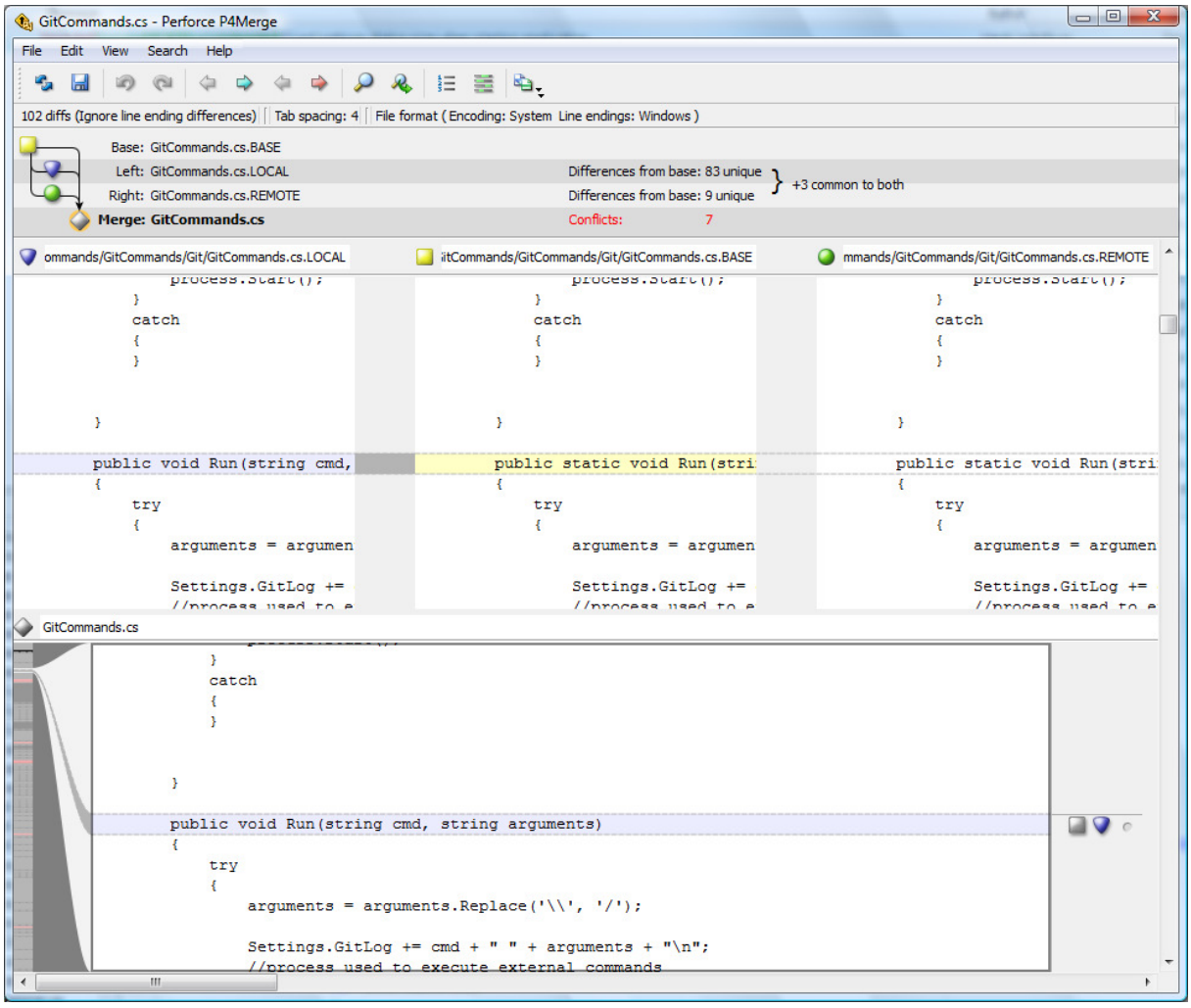
File deleted and changed	Use modified or deleted file?
File deleted and created	Use created or deleted file?
File changed both locally and remotely	Start merge tool.

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.

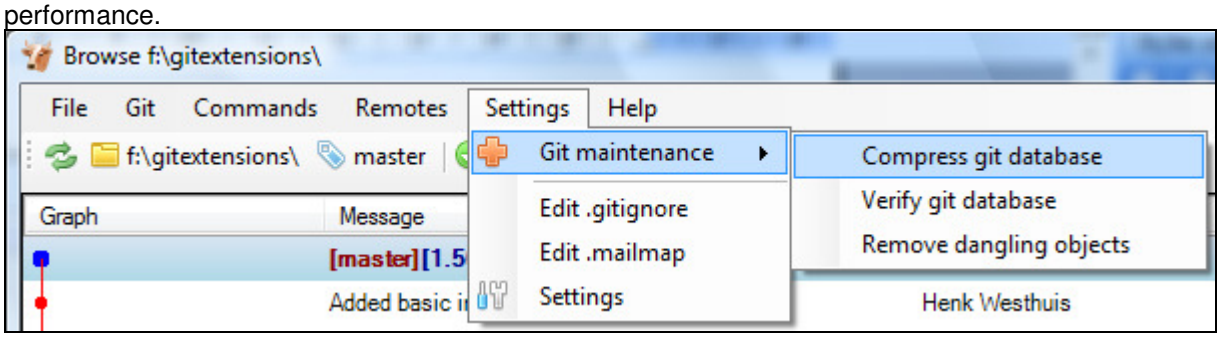

GitCommands.cs - Perforce P4Merge
File Edit View Search Help
102 diffs (Ignore line ending differences) Tab spacing: 4 File format (Encoding: System Line endings: Windows)
Base: GitCommands.cs.BASE
Left: GitCommands.cs.LOCAL
Right: GitCommands.cs.REMOTE
Merge: GitCommands.cs
Differences from base: 83 unique
Differences from base: 9 unique
+3 common to both
Conflicts: 7
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 Maintenance

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

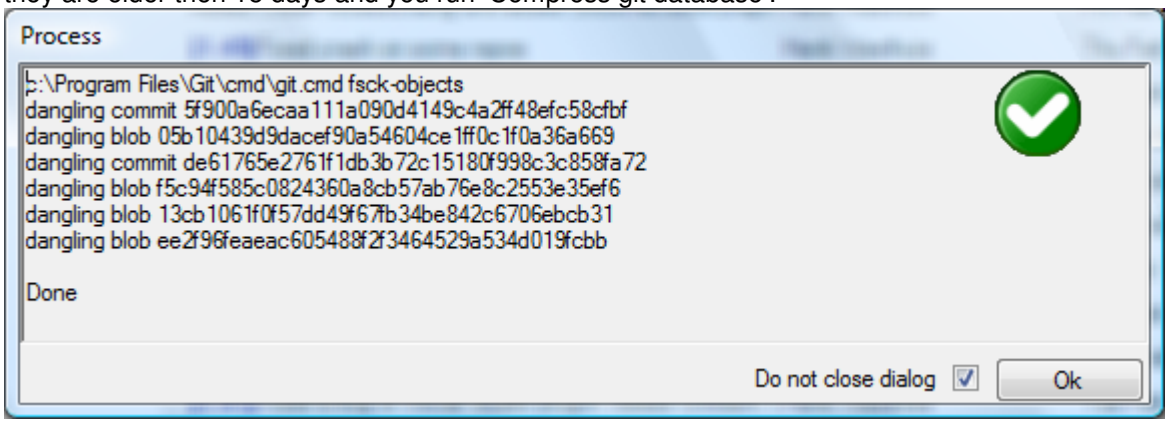
10.1 Compress Git database

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.



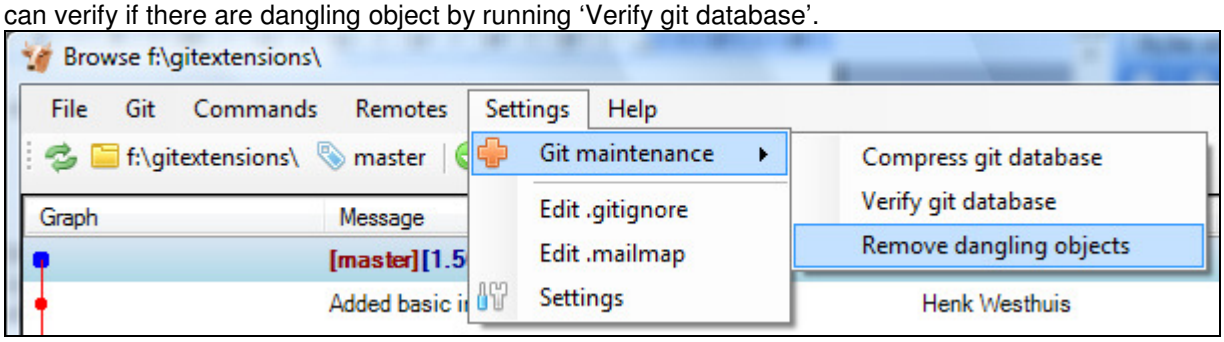
10.2 Verify Git database

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'.



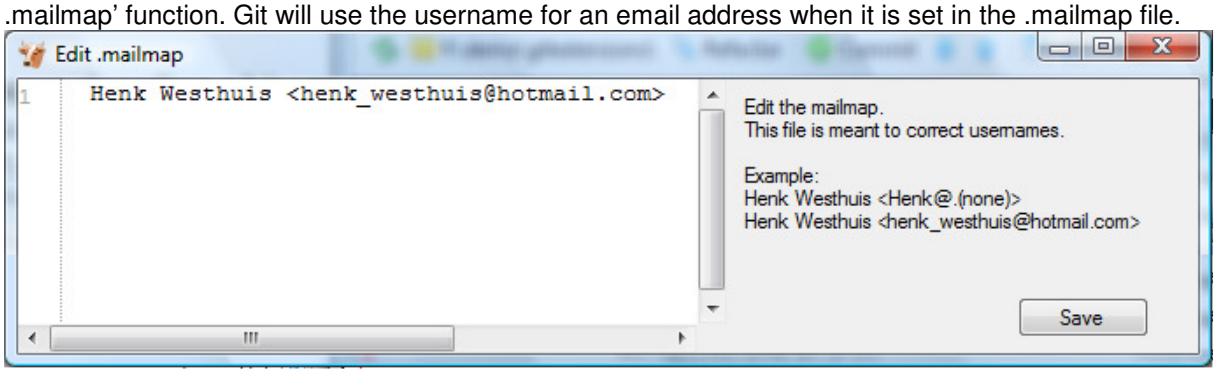
10.3 Remove dangling objects

You can force Git to delete removed objects by running the 'Remove dangling objects' function. You can verify if there are dangling object by running 'Verify git database'.



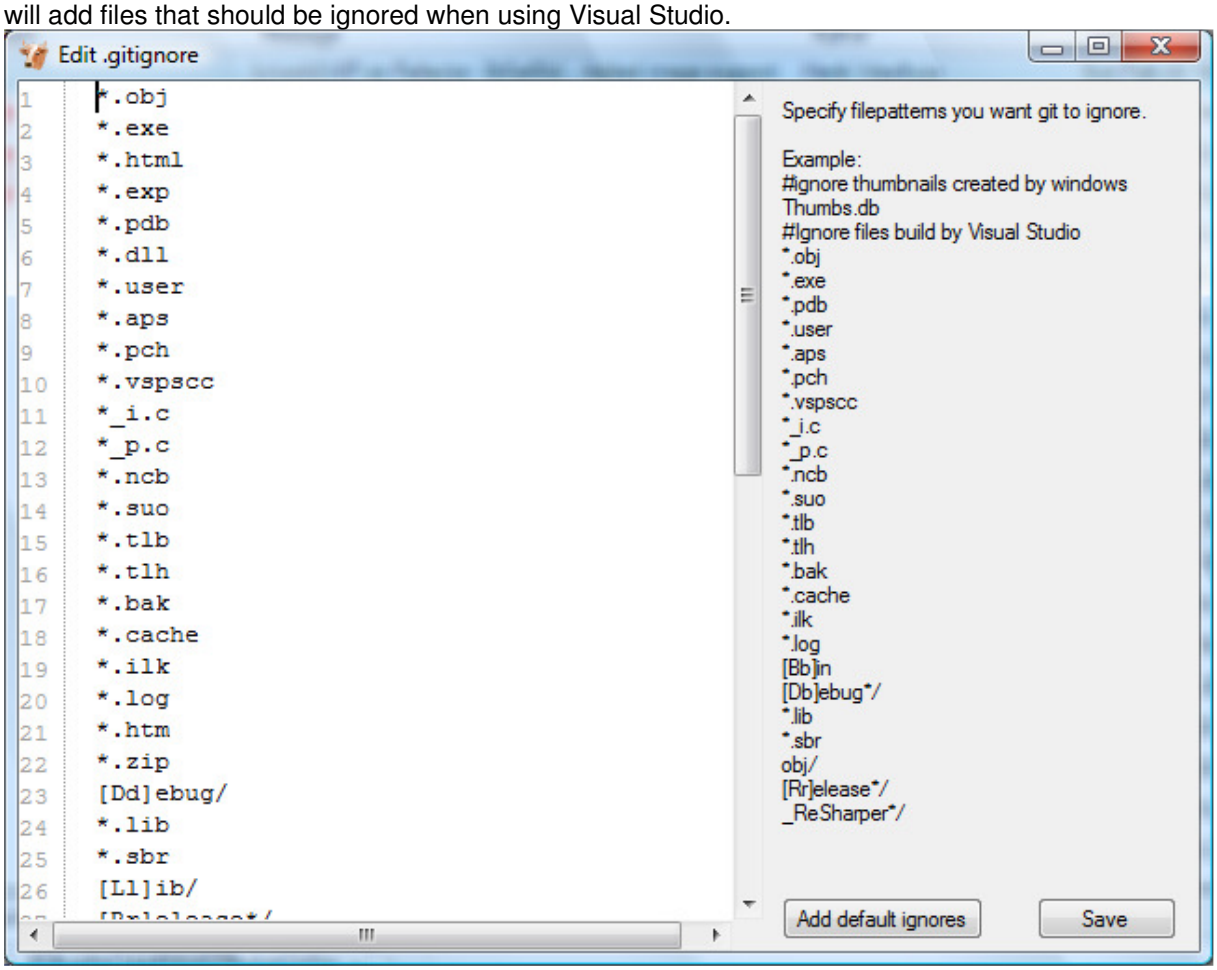
10.4 Fix user names

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.



10.5 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.



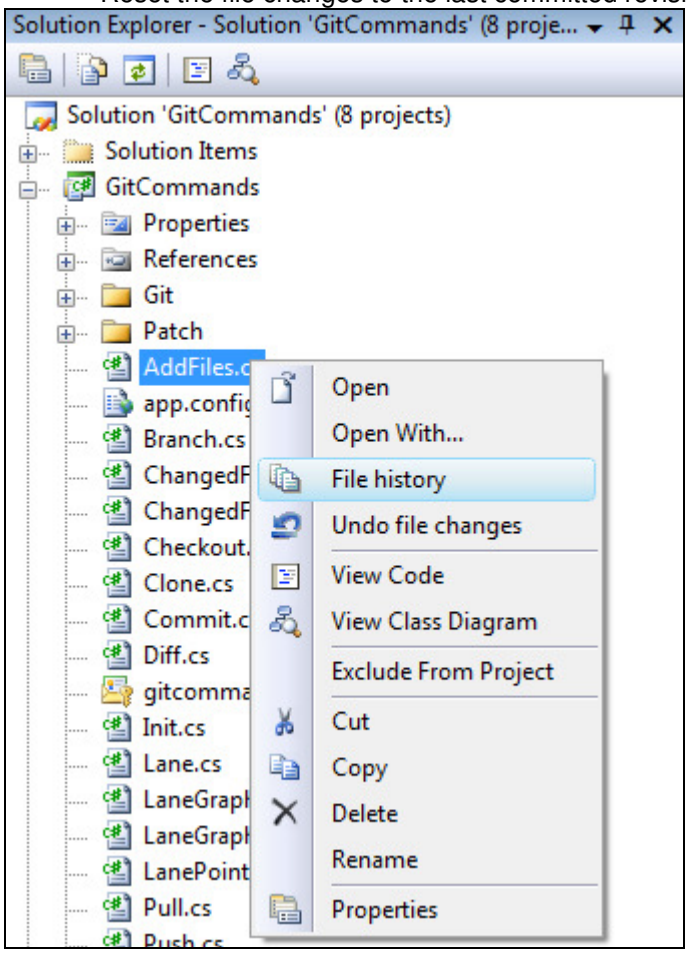
11 Integration

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

11.1 Visual Studio

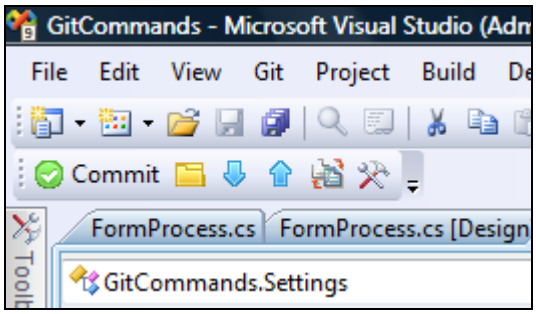
There are two options in the context menu on files.

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

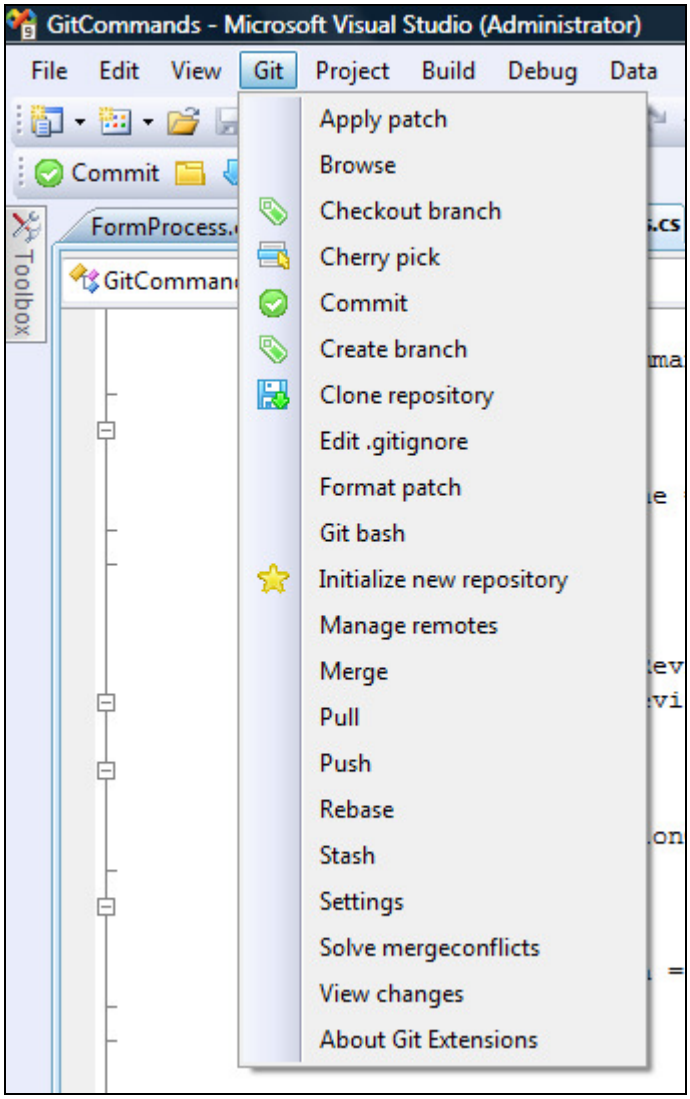


A Git Extensions toolbar allows you to perform the most common actions.

	Commit
	Browse
	Pull
	Push
	Stash changes
	Settings

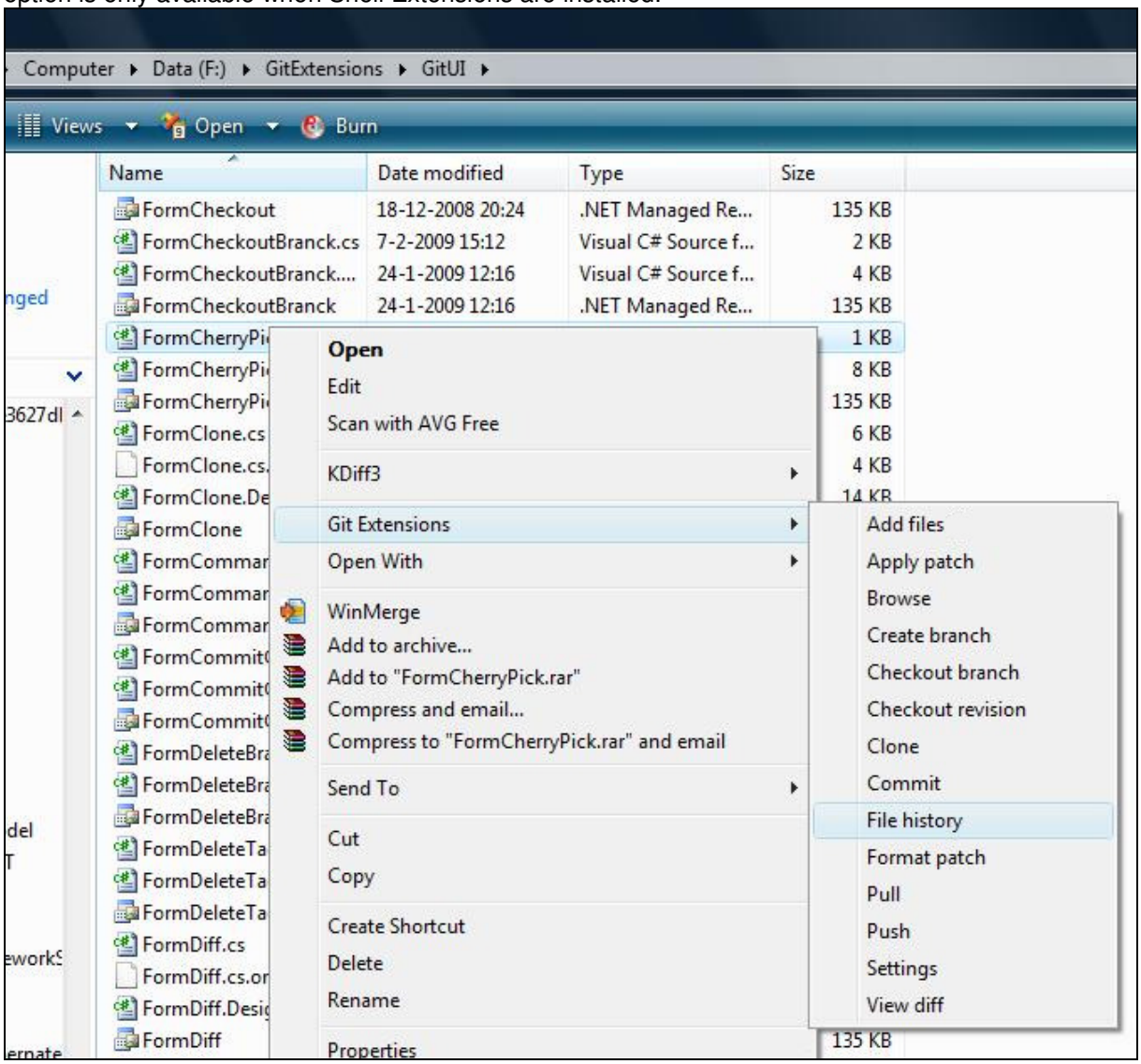


Almost all function can be started from the 'Git' menu in Visual Studio.



11.2 Windows Explorer

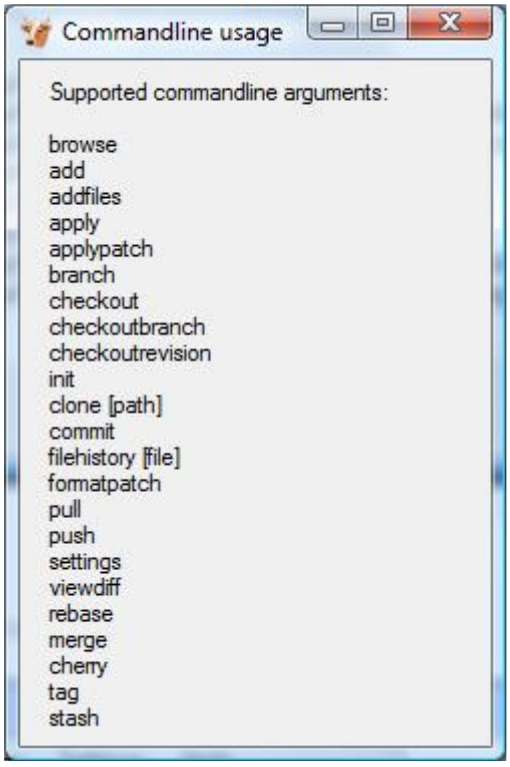
The common commands can be started from Windows Explorer using the shell extensions. This option is only available when Shell Extensions are installed.

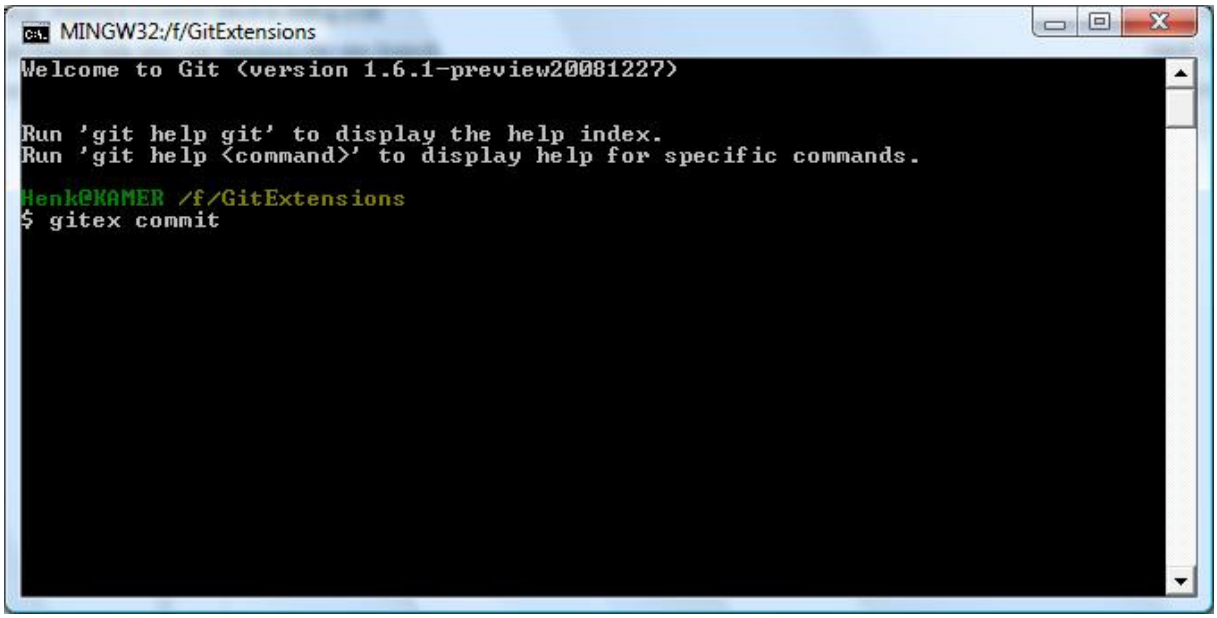


12 Command line

12.1 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