SmartGit/Hg Manual

syntevo GmbH, www.syntevo.com

2013

Contents

Chapter 1

Introduction

SmartGit/Hg is a graphical Git and Mercurial client which can also connect to SVN repositories. SmartGit/Hg runs on Linux, Mac OS X (10.5 or newer) and Windows (XP or newer). Git and Mercurial (Hg) are distributed version control system (DVCS).

Acknowledgments

The development of SmartGit/Hg is mainly driven by its users and we would like to thank all of them for their valuable feedback, for suggesting features and reporting bugs and thereby helping us to improve SmartGit/Hg.

Chapter 2

Git Concepts

This section helps you to get started with Git and gives you an understanding of the fundamental Git concepts.

2.1 Repository, Working Tree, Commit

First, we need to introduce some Git-specific terms which may have different meanings in other version control systems such as Subversion.

Classical centralized version control systems such as Subversion (SVN) have so-called 'working copies', each of which corresponds to exactly one repository. SVN working copies can correspond to the entire repository or just to parts of it. In Git, on the other hand, you always deal with (local) repositories. Git's *working tree* is the directory where you can edit files and it is always part of a repository. So-called *bare repositories*, used on servers as central repositories, don't have a working tree.

Example
Let's assume you have all your project-related files in a directory `D:\my-project`. Then this directory represents the repository, which consists of the working tree (containing all files to edit) and the attached repository meta data which is located in the `D:\my-project\.git` directory.

2.2 Typical Project Life Cycle

As with all version control systems, there typically exists a central repository containing the project files. To create a local repository, you need to *clone* the *remote* central repository. Then the local repository is connected to the remote repository, which, from the local repository's point of view, is referred to as *origin*. The cloning step is analogous to the initial SVN checkout for getting a local working copy.

Having created the local repository containing all project files from *origin*, you can now make changes to the files in the working tree and *commit* these changes. They will be stored in your local repository only, so you don't even need access to a remote repository when committing. Later on, after you have committed a couple of changes, you can *push them to the remote repository* (see 3.2.2). Other users who have their own clones of the origin repository can *pull from the remote repository* (see 3.2.1) to get your pushed changes.

2.3 Branches

Branches can be used to store independent series of commits in the repository, e.g., to fix bugs for a released software project while simultaneously developing new features for the next project version.

Git distinguishes between two kinds of branches: *local branches* and *remote branches*. In the local repository, you can create as many local branches as you like. Remote branches, on the other hand, are local branches of the origin repository. In other words: Cloning a remote repository clones all its local branches which are then stored in your local repository as remote branches. You can't work directly on remote branches, but have to create local branches, which are "linked" to the remote branches. The local branch is called *tracking branch*, and the corresponding remote branch *tracked branch*. Local branches can be tracking branches, but they don't have to.

The default local main branch created by Git is named *master*, which is analogous to SVN's *trunk*. When cloning a remote repository, the master tracks the remote branch *origin/master*.

2.3.1 Working with Branches

When you push changes from your local branch to the origin repository, these changes will be propagated to the tracked (remote) branch as well. Similarly, when you pull changes from the origin repository, these changes will also be stored in the tracked (remote) branch of the local repository. To get the tracked branch changes into your local branch, the remote changes have to be *merged from the tracked branch*. This can be done either directly when invoking the *Pull* command in SmartGit/Hg, or later by explicitly invoking the *Merge* command. An alternative to the Merge command is the Rebase command.

Tip	The method to be used by Pull (either *Merge* or *Rebase*) can be configured in **Project**|**Repository Settings** on the **Pull** tab.

2.3.2 Branches are just pointers to commits

Every branch is simply a named pointer to a commit. A special unique pointer for every repository is the *HEAD* which points to the commit the working tree state currently corresponds to. The HEAD cannot only point to a commit, but also to a local branch, which itself points to a commit. Committing changes will create a new commit on top of the commit or local branch the HEAD is pointing to. If the HEAD points to a local branch, the branch pointer will be moved forward to the new commit; thus the HEAD will also indirectly point to the new commit. If the HEAD points to a commit, the HEAD itself is moved forward to the new commit.

2.4 Commits

A *commit* is the Git equivalent of an SVN revision, i.e., a set of changes that is stored in the repository along with a commit message. The Commit command (see 3.3.3) is used to store working tree changes in the local repository, thereby creating a new commit.

2.4.1 Commit Graph

Since every repository starts with an initial commit, and every subsequent commit is directly based on one or more parent commits, a repository forms a "commit graph" (or technically speaking, a directed, acyclic graph of commit nodes), with every commit being a direct or indirect descendant of the initial commit. Hence, a commit is not just a set of changes, but, due to its fixed location in the commit graph, also represents a unique repository state.

Normal commits have exactly one parent commit, the initial commit has no parent commits, and the so-called *merge commits* have two or more parent commits.

```
o ... a merge commit
| \
|  o ... a normal commit
|  |
o  | ... another normal commit
| /
o  ... yet another normal commit which has been branched
|
o ... the initial commit
```

Each commit is identified by its unique *SHA*-ID, and Git allows *checking out* every commit using its SHA. However, with SmartGit/Hg you can visually select the commits to check out, instead of entering these unwieldy SHAs by hand. Checking out will set the HEAD

and working tree to the commit. After having modified the working tree, committing your changes will produce a new commit whose parent will be the commit that was checked out. Newly created commits are called *heads* because there are no other commits descending from them.

2.4.2 Putting It All Together

The following example shows how commits, branches, pushing, fetching and (basic) merging play together.

Let's assume we have commits `A`, `B` and `C`. `master` and `origin/master` both point to `C`, and `HEAD` points to `master`. In other words: The working tree has been switched to the branch *master*. This looks as follows:

```
o [> master][origin/master] C
|
o B
|
o A
```

Committing a set of changes results in commit `D`, which is a child of `C`. `master` will now point to `D`, hence it is one commit ahead of the tracked branch `origin/master`:

```
o [> master] D
|
o [origin/master] C
|
o B
|
o A
```

As a result of a Push, Git sends the commit `D` to the origin repository, moving its `master` to the new commit `D`. Because a remote branch always refers to a branch in the remote repository, `origin/master` of our repository will also be set to the commit `D`:

```
o [> master][origin/master] D
|
o C
|
o B
|
o A
```

Now let's assume someone else has further modified the remote repository and committed `E`, which is a child of `D`. This means the `master` in the origin repository now points to `E`. When fetching from the origin repository, we will receive commit `E` and our repository's `origin/master` will be moved to `E`:

```
o [origin/master] E
|
o [> master] D
|
o C
|
o B
|
o A
```

Finally, we will now merge our local `master` with its tracking branch `origin/master`. Because there are no new local commits, this will simply move `master` *fast-forward* to the commit `E` (see Section 3.4.4).

```
o [> master][origin/master] E
|
o D
|
o C
|
o B
|
o A
```

2.5 The Index

The *Index* is an intermediate cache for preparing a commit. With SmartGit/Hg, you can make heavy use of the Index, or ignore its presence completely - it's all up to you.

The **Stage** command allows you to save a file's content from your working tree in the Index. If you stage a file that was previously version-controlled, but is now missing in the working tree, it will be marked for removal. Explicitly using the **Remove** command has the same effect, as you may be accustomed to from SVN. If you select a file that has Index changes, invoking **Commit** will give you the option to commit all staged changes.

If you have staged some file changes and later modified the working tree file again, you can use the **Discard** command to either revert the working tree file content to the staged changes stored in the Index, or to the file content stored in the repository (HEAD). The **Changes** view of the SmartGit/Hg project window can show the changes between the

HEAD and the Index, between the HEAD and the working tree, or between the Index and the working tree state of the selected file. Individual change hunks can be staged and unstaged there.

When *unstaging* previously staged changes, the staged changes will be moved back to the working tree, if the latter hasn't been modified in the meantime, otherwise the staged changes will be lost. In either case, the Index will be reverted to the HEAD file content.

2.6 Working Tree States

There are some particular situations where commits cannot be performed, for instance when a merge has failed due to a conflict. In this case, there are two ways to finish the merge: Either by resolving the conflict, staging the file changes and performing the commit on the working tree root, or by reverting the whole working tree.

Chapter 3

Important Git Commands

This chapter gives you an overview of important SmartGit/Hg commands.

3.1 Project-Related

A SmartGit/Hg project consists of one or more local repositories assigned to it. For greater user convenience, a couple of (primarily) GUI-related options are stored in the SmartGit/Hg project. Depending on the selected directory, when cloning or opening a local repository, SmartGit/Hg allows creating a new project, opening an existing one in the selected directory or adding the repository to the currently open project.

To group projects, use **Project|Open or Manage Projects**. To remove a repository from a SmartGit/Hg project, use **Project|Remove Repository from Project**.

3.1.1 Opening a Repository

Use **Project|Open Repository** to either open an existing local repository (e.g. initialized or cloned with the Git command line client) or to initialize a new repository.

You need to specify which local directory you want to open. If the specified directory is not a Git or Mercurial repository yet, you have the option to initialize it.

3.1.2 Cloning a Repository

Use **Project|Clone** to create a clone of another Git, Mercurial or SVN repository.

Specify the repository to clone either as a remote URL (e.g. ssh://user@server:port/path), or, if the repository is locally available on your file system, as a file path. In the **Selection** step you can configure whether submodules should be fetched as well: usually you will have this option selected, because submodules are an integral part of the main repository

you are cloning. You should deselect this option only, if you do not wish to receive certain submodules. For details, refer to (see 3.6). Similarly, you will want to fetch the entire repository usually, including all heads and tags. If you are only interested in a specific head (branch) or tag, you may restrict the clone here. Note that a few Git commands do not work properly for such partial repositories (e.g. Pull with Rebase). In the subsequent steps you have to provide the path to the local directory where the clone should be created and the project to which the repository should be assigned.

3.2 Synchronizing with Remote Repositories

Synchronizing the states of local and remote repositories consists of pulling from and pushing to the remote repositories. SmartGit/Hg also has a Synchronize command that combines pulling and pushing.

3.2.1 Pull

The Pull command fetches commits from a remote repository, stores them in the remote branches, and optionally "integrates" (i.e. merges or rebases) them into the local branch.

Use **Remote|Pull** (or the corresponding toolbar button) to invoke the Pull command. This will open the Pull dialog, where you can specify what SmartGit/Hg will do after the commits have been fetched: Merge the local commits with the fetched commits or rebase the local commits onto the fetched commits. In the latter case, you can merge or rebase by hand, as explained in Section 3.4.4 and Section 3.4.7, respectively. These options are meaningless, if you select to **Fetch Only**.

The Pull dialog allows you to set your choice as default for the current branch. To change the default choice for new branches, go to **Project|Repository Settings**.

If a merge or rebase is performed after pulling, it may fail due to conflicting changes. In that case SmartGit/Hg will leave the repository in a *merging* or *rebasing* state so you can either resolve the conflicts and proceed, or abort the operation. See Section 3.4.4 and Section 3.4.7 for details.

Pulling tags

By default, Git (and hence SmartGit) will only pull new tags, but don't update possibly changed tags from the remote repository. To have tags updated as well, you have to configure `--tags` as `tagopt` for your remote.

Example
To update tags when pulling from `origin`, your `.git/config` file should look like the following ("..." represents your already currently set values):

```
[remote "origin"]
  fetch = ...
  url = ...
  tagopt = --tags
```

3.2.2 Push

The various Push commands allow you to push (i.e. send) your local commits to one or more remote repositories. SmartGit/Hg distinguishes between the following Push commands:

- **Push**: Pushes all commits in one or more local branches to their matching remote branches. More precisely, on the Push dialog you can choose between pushing the commits in the current branch to its matching remote branch, and pushing the commits in all local branches with matching remote branches to said remote branches. A local branch 'matches' a remote branch if the branch names match, e.g. 'master' and 'origin/master'. With this Push command you can push to multiple repositories in a single invocation. SmartGit/Hg will detect automatically whether a *forced push* will be necessary.
- **Push To**: Pushes all commits in the current branch either to its matching branch, or to a *ref* specified by name. With the Push To command you can only push to one remote repository at a time. If multiple repositories have been set up, the Push To dialog will allow you to select the remote repository to push to. Also, the Push To command always allows to do a *forced* push, what can be convenient. This is necessary when pushing to a *secondary* remote repository for which forcing the push may be necessary while it is not when pushing to the primary remote repository (i.e. the one which is considered by SmartGit/Hg's *forced push* detection).
- **Push Commits**: Pushes the selected range of commits from the **Outgoing** view, rather than all commits, in the current branch to its tracked remote branch.

If you try to push commits from a new local branch, you will be asked whether to set up tracking for the newly created remote branch. In most cases it is recommended to set up tracking, as it will allow you to receive changes from the remote repository and make use of Git's branch synchronization mechanism (see Section 2.3).

The Push commands listed above can be invoked from several places in SmartGit/Hg's project window:

- **Menu and toolbar**: In the menu, you can invoke the various Pull commands with **Remote|Push**, **Remote|Push To** and **Remote|Push Commits**. The first two may also be available as toolbar buttons, depending on your toolbar configuration. The third command is only enabled if the **Outgoing** view is focused.
- **Directories view**: You can invoke **Push** in the **Directories** view by selecting the project root and choosing **Push** from the root's context menu.
- **Branches view**: In the context menu of the **Branches** view, you can invoke **Push** and **Push To** on local branches. Additionally, you can invoke **Push** on tags.
- **Outgoing view**: To push a range of commits up to a certain commit, select that commit in the **Outgoing** view and invoke **Push Commits** from the context menu.

3.2.3 Synchronize

With the Synchronize command, you can push local commits to a remote repository and pull commits from that repository at the same time. This simplifies the common workflow of separately invoking Push (see 3.2.2) and Pull (see 3.2.1) to keep your repository synchronized with the remote repository.

If the Synchronize command is invoked and there are both local and remote commits, the invoked push operation fails. The pull operation on the other hand is performed even in case of failure, so that the commits from the remote repository are available in the tracked branch, ready to be merged or rebased. After the remote changes have been applied to the local branch, you may invoke the Synchronize command again.

In SmartGit/Hg's project window, the Synchronize command can be invoked as follows:

- from the menu via **Remote|Synchronize**,
- with the Synchronize toolbar button if the toolbar is configured accordingly,
- and in the **Directories** view via **Synchronize** in the project root's context menu.

Note	*Why does SmartGit/Hg first Push, then Pull?* SmartGit/Hg first tries to Push, then Pulls. If the Push failed because of remote changes, you will have the remote changes already locally available and can test the local changes before pushing them by invoking Push or Synchronize a second time. If SmartGit/Hg would first Pull and then Push, local changes either would be rebased on top of the pulled remote changes or remote changes silently merged to local changes. This would mean to push untested changes.

3.3 Local Operations on the Working Tree

3.3.1 Stage, Unstage, and the Index Editor

Git's Index (see 2.5) is basically a selection of changes from the working tree to be included in the next commit. SmartGit/Hg provides following facilities to modify this selection:

- **Stage and Unstage commands**: allow you to add and remove whole files to and from the Index, respectively.
- **Changes view**: allows you to add or remove individual 'hunks' (i.e. parts of the file) to and from the Index.
- **Index Editor**: allows you to directly edit the contents of the Index for a certain file, thereby adding or removing arbitrary content to and from the Index.

If you invoke Stage on an untracked file, e.g. via **Local**|**Stage**, that file will be scheduled for addition to the repository. On a tracked file, the effect of Stage is to schedule for the next commit any changes made to the file, including its removal.

Conversely, the Unstage command (**Local**|**Stage**) will discard the selected file's changes in the Index, meaning that the Index changes will be lost, unless they are identical to the current changes in the working tree.

Similarly, staging and unstaging hunks from the **Changes** view will schedule or unschedule parts of the file's changes for the next commit.

If you select a file and invoke the Index Editor, e.g. via **Local**|**Index Editor**, the Index Editor window will come up. It is basically a three-way diff view where the three editors represent the file's state in the repository, the Index and the working tree, respectively. You can edit the file contents in the Index and the working tree, and move changes between these two editors by clicking on the arrow and 'x' buttons in-between.

Lastly, to commit staged changes, select the working tree root in the **Directories** view and invoke the Commit (see 3.3.3) command.

3.3.2 Ignore

Invoke **Local**|**Ignore** on a selection of untracked files to mark them as to be ignored. The Ignore command is useful for preventing certain local files that should not be added to the repository from showing up as 'untracked'. This reduces visual clutter and also makes sure you won't accidentally add them to the repository. If the menu option **View**|**Show Ignored Files** is selected, ignored files will be shown.

When you mark a file in SmartGit/Hg as 'ignored', an entry will be added to the `.gitignore` file in the same directory. Git supports various options to ignore files, e.g.

patterns that apply to files in subdirectories. With the SmartGit/Hg Ignore command you can only ignore files in the same directory. To use the more advanced Git ignore options, you may edit the `.gitignore` file(s) by hand.

3.3.3 Commit

The Commit command is used for saving local changes in the local repository. You can invoke it via **Local|Commit**.

If the working tree is in *merging* or *rebasing* state (see Section 3.4.4 and Section 3.4.7), you can only commit the whole working tree. Otherwise, you can select the files to commit. Previously tracked, but now missing files will be removed from the repository, and untracked new files will be added. This behavior can be changed in the Preferences, section **Commands**.

If you have staged changes in the Index (see 3.3.1), you can commit them by selecting at least one file with Index changes or by selecting the working tree root before invoking the Commit command.

While entering the commit message, you can use <*Ctrl*>+<*Space*>-keystroke to auto-complete file names or file paths. Use **Select from Log** to pick a commit message or SHA ID from the Log.

If **Amend last commit instead of creating a new one** is selected, you can update the commit message and files of the previous commit, e.g. to add a forgotten file. By default, this option is only available for not yet pushed commits. You can enable this option for already pushed commits as well in the Preferences, section **Commands**.

If you commit while the working tree is in *merging* state, you will have the option to create either a merge commit or a normal commit. See Section 3.4.4 for details.

3.3.4 Altering Local Commits

SmartGit/Hg provides several ways to make alterations to local commits:

- **Undo Last Commit**: Invoke **Local|Undo Last Commit** from the project window's menu to undo the last commit. The contents of the last commit will be moved to the Index (see 2.5), so no changes will be lost.
- **Edit Last Commit Message**: Invoke **Local|Edit Last Commit Message** from the project window's menu to edit the commit message of the last commit.
- **Edit Commit Message**: In the **Outgoing** view on the project window, you can edit the commit message of any of the local commits by selecting the commit and invoking **Edit Commit Message** from the commit's context menu.

- **Join Commits**: To combine a range of local commits into a single commit, select the commit range in the **Outgoing** view on the project window and invoke **Join Commits** from the context menu of the commit range.
- **Reorder Commits**: In the **Outgoing** view you can drag&drop a commit to some other location in the list to effectively change its position.

Warning!	Do not undo an already pushed commit unless you know what you're doing! If you do this, you need to force-push your local changes, which might discard other users' commits in the remote repository.

3.3.5 Discard

Use **Local**|**Discard** to revert the contents of the selected files either back to their Index (see 2.5) state, or back to their repository state (HEAD). If the working tree is in a *merging* or *rebasing* state, use this command on the root of the working tree to get out of the *merging* or *rebasing* state.

3.3.6 Remove

Use **Local**|**Remove** to remove files from the local repository and optionally to delete them in the working tree.

If the local file in the working tree is already missing, staging (see 3.3.1) will have the same effect, but the Remove command also allows you to remove files from the repository while keeping them locally.

3.3.7 Delete

Use **Local**|**Delete** to delete local files (or directories) from the working tree.

3.3.8 Stashes

Stashes are a convenient way to put the current working tree changes aside and re-apply them later.

Use **Local**|**Save Stash** to stash away local modifications of your working tree. The resulting stash will show up in the **Branches** view.

Note	The option **Include untracked files** is convenient to include *untracked* files for the stash as well, however depending on the operating system it may take significantly longer to execute the operation.

Right-click the stash and select **Apply Stash** to re-apply the contained changes to your working tree again. To get rid of obsolete stashes, use **Drop Stash**, however be aware that this will irretrievably get rid of the changes which are stored in the stash.

3.4 Working with Branches and Tags

3.4.1 Switching between Branches

The simplest way to switch between branches (or more precisely, between the latest commits within branches) is to double-click on a branch in the **Branches** view and confirm the Switch Branch dialog that comes up. The **Branches** view can be found both on the project window and on the Log window (see 3.5.1).

If you switch to a remote branch, you can optionally create a new local branch (recommended) and set up branch tracking.

If you switch to a local branch that tracks a remote branch, and the latter is ahead of the local branch by a couple of commits, you can decide whether you just want to just switch to the latest commit of the local branch, or to switch and let SmartGit/Hg do a fast-forward merge to the latest commit of the remote branch. For further information on merging, see Section 3.4.4.

If you have local changes in your working tree, the Switch might fail. In this case, SmartGit/Hg offers you to stash away the local changes before executing the actual Switch command and re-apply the changes from the stash after executing the command. For further information on stashes, see Section 3.3.8.

A more general procedure than branch switching is to check out arbitrary commits, which is explained in Section 3.4.2.

3.4.2 Checking Out Commits

In SmartGit/Hg, there are two ways to switch the working tree to a certain commit:

- **Project window**: On the project window, invoke **Branch**|**Check Out** from the menu. This will open a dialog containing a Log view, where you can select the commit to switch to.
- **Log window**: On the Log window (see 3.5.1), select the commit to switch to and then select **Check Out** from its context menu.

If you select a commit where local branches point to, you will have the option to switch to these branches. If you select a commit where remote branches without corresponding local branches point to, you will have the option to create a corresponding local branch.

Similar to Switch (see 3.4.1), Check Out will optionally stash away local changes, if necessary.

3.4.3 Adding, Renaming and Deleting Branches and Tags

You can add, rename and delete branches and tags both from the project window and from the Log (see 3.5.1) window.

Project Window

The **Branches** view on the project window has various context menu entries for adding, renaming and deleting selected branches and tags. These commands can also be invoked via the entries in the **Branch** menu.

Log Window

On the Log window, you can add a branch or tag on a commit by selecting the commit in the Log graph and invoking **Add Branch** or **Add Tag** in the commit's context menu. Similarly, you can delete a branch or tag by selecting the commit to which the branch or tag pointer is attached and invoking **Delete** in the commit's context menu.

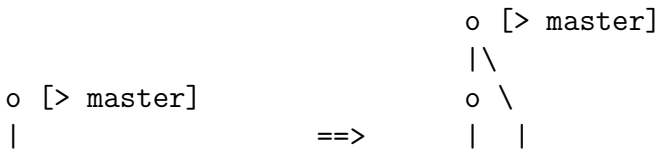
Via the context menu of the Log window's **Branches** view, you can add and delete branches and tags as well. In addition to that, the **Branches** view also allows you to rename branches.

3.4.4 Merge

The Merge command allows you to merge changes from another branch into the current branch. In addition to the "normal" merge operation, there are two variants called "fast-forward merge" and "squash merge". The differences are as follows:

"Normal" Merge

In case of a normal merge, a merge commit with at least two parent commits (i.e., the last from the current branch and the last from the merged branch) is created. See the following figure, where > indicates where the HEAD is pointing to:



```
|  o [a-branch]              |  o [a-branch]
.  .                         .  .
```

Fast-forward Merge

If the current branch is completely included in the branch to be merged with (i.e. the latter is simply a couple of commits ahead), then no extra merge commits is created. Instead, the branch pointer of the current branch is moved forward to match the branch pointer of the other branch, as shown below:

```
o [origin/master]              o [> master][origin/master]
|                      ==>     |
o [> master]                   o
.                              .
```

In SmartGit/Hg, there are several places from which you can initiate a merge:

- **Menu and toolbar**: On the project window, select **Branch|Merge** to open the **Merge** dialog, where you can select the branch to be merged into the current branch. Depending on your toolbar settings, you can also open this dialog via the **Merge** button on the toolbar.
- **Branches view**: In the **Branches** view (available both on the project window and the Log window), you can right-click on a branch and select **Merge** to merge the selected branch into the current branch.
- **Log Graph**: On the Log graph of the **Log** window, you can perform a merge by right-clicking on the head commit of the branch to be merged with and selecting **Merge** from the context-menu.

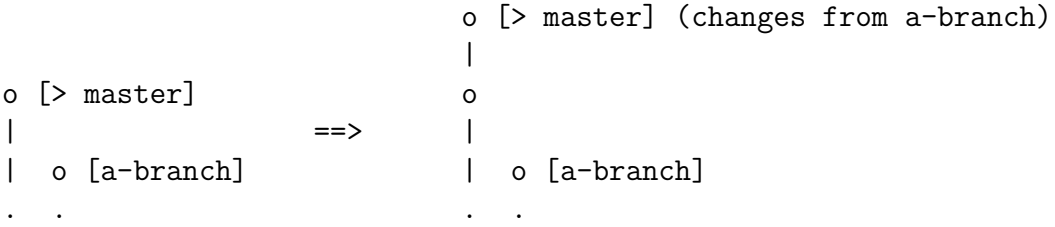
Regardless of where you invoked the Merge command, you will be given the choice between **Create Merge-Commit** and **Merge to Working Tree**, and optionally also **Fast-Forward** if a fast-forward merge is possible.

If you choose **Create Merge-Commit**, SmartGit/Hg will perform the merge and create a merge commit, assuming there are no merge conflicts. If there are merge conflicts, or if you choose **Merge to Working Tree**, SmartGit/Hg will perform the merge, but leave the working tree in a *merging* state, so that you can manually resolve merge conflicts and review the changes to be made. See Section 3.4.8 for further information on how to deal with merge conflicts.

Squash Merge

The squash merge works like a normal merge, except that it discards the information about where the changes came from. Hence it only allows you to create normal commits.

The squash merge is useful for merging changes from local (feature) branches where you don't want all of your feature branch commits to be pushed into the remote repository.



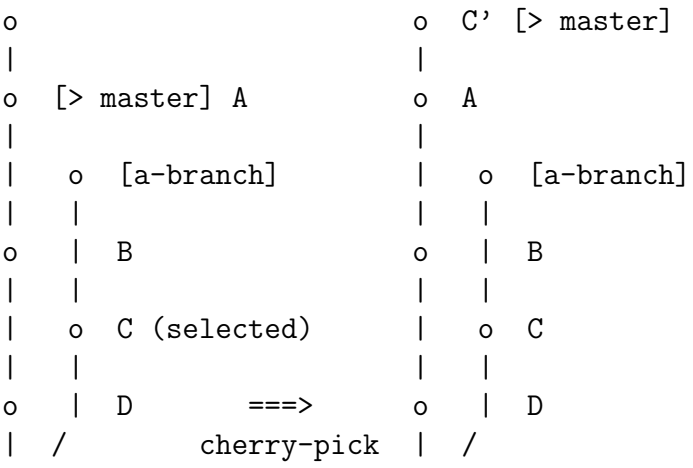
On the **Commit** dialog, you can choose between a normal merge (merge commit) and a squash merge (simple commit). Thus, to perform a squash merge you have to choose **Merge to Working Tree** when initiating the merge, since otherwise you won't see the **Commit** dialog.

Merge versus Rebase

A Git-specific alternative to merging is *rebasing* (see Section 3.4.7), which can be used to keep the history linear. For example, if a user has made local commits and performs a pull with merge, a merge commit with two parent commits - the user's last commit and the last commit from the tracked branch - is created. When using rebase instead of merge, Git applies the local commits on top of the commits from the tracked branch, thus avoiding a merge commit.

3.4.5 Cherry-Pick

The Cherry-Pick command allows you to "apply" certain commits from another branch to the current branch.



```
| /                       | /
o                         o
```

In SmartGit/Hg, there are several places from which you can initiate a cherry-pick:

- **Menu and toolbar**: On the project window, select **Branch**|**Cherry-Pick** to open the **Cherry-Pick** dialog, where you can select one or more commits to cherry-pick. Depending on your toolbar settings, you can also open this dialog via the **Cherry-Pick** button on the toolbar.
- **Log Graph**: On the Log graph of the **Log** window, you can perform a cherry-pick by right-clicking on one or more commits and selecting **Cherry-Pick** from the context-menu.

3.4.6 Revert

The Revert command allows you to "undo" certain commits (from whatever branch) in the current branch.

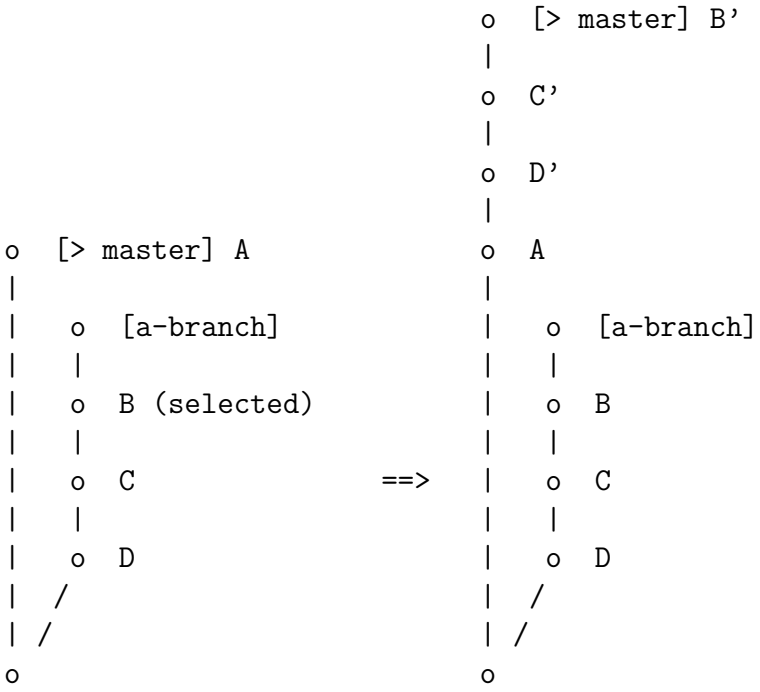
```
o                         o  reversed-C  [> master]
|                         |
o  [> master] A           o  A
|                         |
|   o  [a-branch]         |   o  [a-branch]
|   |                     |   |
o   |  B                  o   |  B
|   |                     |   |
|   o  C (selected)       |   o  C
|   |                     |   |
o   |  D          ===>    o   |  D
|  /            revert    |  /
| /                       | /
o                         o
```

In SmartGit/Hg, there are several places from which you can initiate a Revert:

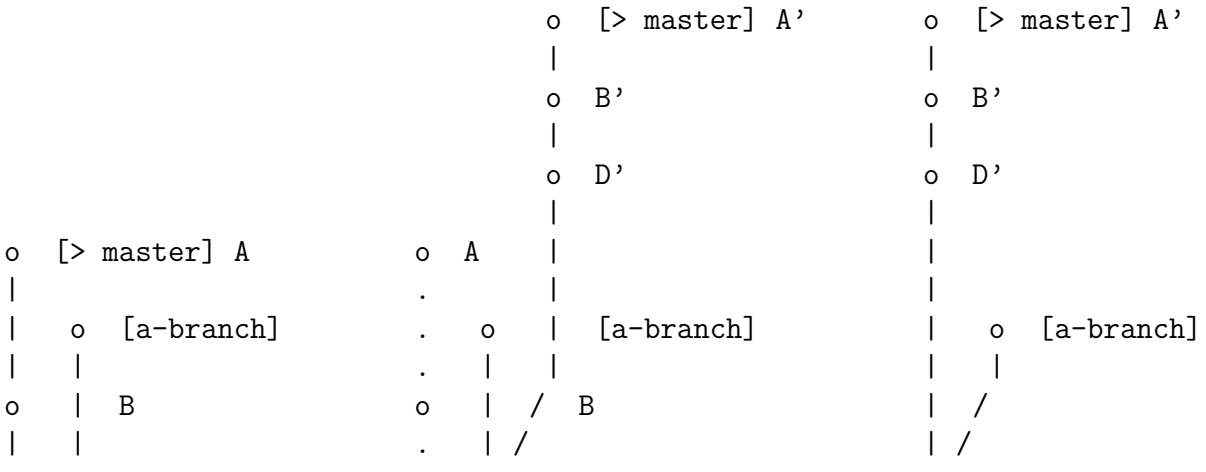
- **Menu and toolbar**: On the project window, select **Branch**|**Revert** to open the **Revert** dialog, where you can select one or more commits to revert. Depending on your toolbar settings, you can also open this dialog via the **Revert** button on the toolbar.
- **Log Graph**: On the Log graph of the **Log** window, you can perform a revert by right-clicking on one or more commits and selecting **Revert** from the context-menu.

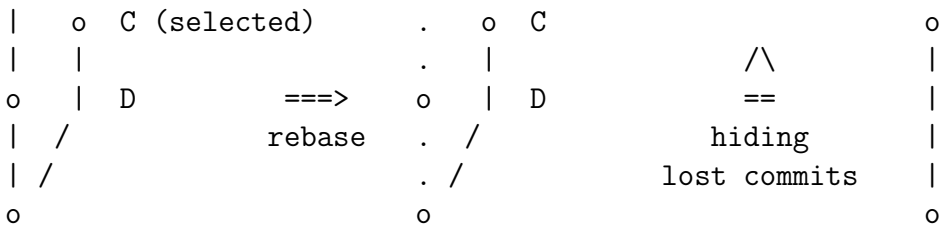
3.4.7 Rebase

The Rebase command allows you to apply commits from one branch to another. Rebase can be viewed as more powerful version of Cherry-Pick (see 3.4.5), which is optimized to apply multiple commits from one branch to another. In SmartGit/Hg, a distinction is made between **Rebase HEAD to** and **Rebase to HEAD**: The former rebases ("moves") commits from the current branch to another branch, the latter works in the opposite direction. This difference is illustrated below. First, rebasing from another branch onto the HEAD:



... and rebasing from the current branch onto another branch:





In SmartGit/Hg, there are several places from which you can initiate a rebase:

- **Menu and toolbar**: On the project window, select **Branch|Rebase HEAD to** or **Branch|Rebase to HEAD** to open the **Rebase** dialog, where you can select the branch to rebase the HEAD onto, or the branch to rebase onto the HEAD, respectively. Depending on your toolbar settings, you can also open this dialog via the buttons **Rebase HEAD to** and **Rebase to HEAD** on the toolbar.
- **Branches view**: In the **Branches** view, you can right-click on a branch and select **Rebase HEAD to** to rebase your current HEAD onto the selected branch.
- **Log Graph**: On the Log graph of the **Log** window, you can perform a rebase by right-clicking on a commit and selecting **Rebase HEAD to** or **Rebase to HEAD** from the context-menu.

Just like a merge, a rebase may fail due to merge conflicts. If that happens, SmartGit/Hg will leave the working tree in *rebasing* state, allowing you to either manually resolve the conflicts or to abort the rebase. See Section 3.4.8 for further information.

3.4.8 Resolving Conflicts

When a merge (see 3.4.4), a cherry-pick (see 3.4.5) or a rebase (see 3.4.7) fails due to conflicting changes, SmartGit/Hg stops the operation and leaves the working tree in a conflicted state, so that you can either abort the operation, or resolve the conflicts and continue with the operation. This section explains how you can do that with SmartGit/Hg. Generally, the following options are available:

- **Resolve dialog**: If you select a file containing conflicts and then invoke **Local|Resolve** in the menu of SmartGit/Hg's project window, the Resolve dialog will come up, where you can set the file's contents to either of the two conflicting versions, i.e. 'Ours' or 'Theirs'. Optionally, you may also choose not to stage the resetting of the file contents, meaning that the conflict marker on that file won't be removed.
- **Conflict Solver**: Selecting a file containing conflicts and invoking **Query|Conflict Solver** will open the Conflict Solver, a three-way diff between the two conflicting versions (left and right editor) and a third version (center editor) that contains the conflicting hunks from both sides, along with conflict markers. You can directly edit

the text in the center editor, and you can move changes from the left and right side into the center by clicking on the arrow and 'x' buttons between the editors.

- **Discard command**: To abort the merge, cherry-pick or rebase, select the project root in the **Directories** view and invoke **Local**|**Discard**.

Lastly, if all conflicts have been resolved, you can continue with the merge, cherry-pick or rebase by selecting the project root in the **Directories** view and invoking **Local**|**Commit**.

3.5 Viewing the Project History

3.5.1 Log

SmartGit/Hg's Log displays the repository's history as a list of commits, sorted by increasing age, and with a graph on the left side to show the parent-child relationships between the commits. What is shown on the Log depends on what was selected when the Log command was invoked:

- To view the history of the entire repository (*root* Log), select the project root in the **Directories** view before invoking the Log command.
- To view the history of a directory within the repository, select the directory in the **Directories** view before invoking the Log command.
- To view the history of a single file within the repository, select the file in the **Files** view before invoking the Log command. If the file is not visible in the **Files** view, either adjust the file table's filter settings (on its top right), or enter the name of the file in the search field above the file table.

A *root* Log can be invoked from other places in SmartGit/Hg as well:

- **Branches view**: In the **Branches** view (just project window), you can right-click on a branch and select **Log** to open the Log for the selected branch.
- **Outgoing view**: In the **Outgoing** view, you can right-click on a specific commit and invoke **Log** to open the Log for the current branch, with the selected commit pre-selected in the Log.

3.5.2 Blame

SmartGit/Hg's Blame window displays the history information for a single file in a way that helps you to track down the commit in which a certain portion of code was introduced into the repository. You can open the Blame window by selecting a file in the **Files** view in SmartGit/Hg's project window and invoking **Query**|**Blame** from the program menu. The Blame window consists of a **Document** view and a **History of current line** view.

Document view

The **Document** view is divided into three parts: some controls on top, a read-only text view on the right, and an *info area* on the left. With the controls on top, you can do two things: specify the **View Commit** at which the text view will display the file contents, and set how to **Highlight** the lines in the text view:

- **Change Since**: The color chosen for a particular line reflects whether the line is "older" or "newer" than the specified **Commit**. More precisely, the color reflects whether the date when the line was last modified lies before or after the date of the commit.
- **Age**: The color chosen for a particular line lies somewhere "between" the two colors used for the oldest and the newest commit in which the file was modified, and thus reflects the line's relative "age". You can choose between two age criteria for determining the line color: either **Commit Index** which refers to the relative position in the relevant commit range (first commit, second commit, etc.) or **Time** which refers to the commit date, i.e. the point in time at which the commit was created.
- **Author**: The color chosen for a particular line depends on the author who made the most recent modification to that line. Each author is mapped to a different color.

The *info area* shows the commit meta data in a compact format for each line and consists of following columns:

- **SHA**: short SHA of the commit
- **Status**: will display an "~"-mark if the line has been modified (and not added) and/or an "M" if the line has been introduced in a merge commit
- **Author**: the initials or a short name of the author
- **Date**: a compact display of the commit date
- **Line number**: the number of the current line

More detailed information for a specific commit will be displayed in the tooltip, when hovering over the *info area*. The hyperlinks can be used to navigate to the specific commit.

History view

The **History of current line** view displays the "history" of the currently selected line from the **Document** view: the "history" consists of all detected *past* and *future* versions of the line, as it is present in the select **View Commit**. The position of the currently selected line from the **Document** view is denoted by pale borders surrounding the corresponding line in the **History** view.

The detection of a link between a *past* and a *future* version of a line depends on the changes which have happened in a commit:

- If a certain number of lines has been replaced by exactly the same number of lines within a commit, this change will be detected as *modification* of the corresponding lines and hence they will share the same history.
- If a certain number of lines has been replaced by larger or smaller number of lines, the detection of *matching* lines depends on the outcome of the internal *compare algorithm*. For the case where a line has been detected as *removed* in a commit (instead of *replaced* by another line, what might be more appropriate), the history contains a leading *commit after line has been removed* entry to which you can navigate. For the case where a line has been detected as *added* in a commit (instead of *replacing* another line, what might be more appropriate), the history contains a trailing *commit before line has been added* entry to which you can navigate.

Note	For lines having a "~"-mark in the **Document** view, the **History** view will always show *past* commits.

3.6 Submodules

Note	This section only applies to submodules of *native* Git repositories, but not of *SVN clones*. For SVN repositories, refer to Section 9.5.

Often, software projects are not completely self-contained, but share common parts with other software projects. Git offers a feature called *submodules*, which allows you to embed one Git repository into another. This is similar to SVN's "externals" feature.

A submodule is a nested repository that is embedded in a dedicated subdirectory of the working tree (which belongs to the parent repository). The submodule is always pointing at a particular commit of the embedded repository. The definition of the submodule is stored as a separate entry in the parent repository's git object database.

The link between working tree entry and foreign repository is stored in the `.gitmodules` file of the parent repository. The `.gitmodules` file is usually versioned, so it can be maintained by all users and/or changes are propagated to all users.

Setting submodule repositories involves an initialization process, in which the required entries are added to the `.git/config` file. The user may later adjust it, for example to fix SSH login names.

3.6.1 Cloning Repositories with Submodules

If you clone an existing project containing one or more submodules via **Project**|**Clone**, make sure the option **Include Submodules** is selected, so that all submodules are automatically initialized and updated. Without this option, you may initialize the submodules later by hand via **Remote**|**Submodule**|**Initialize**. Just initialization itself will leave the submodule directory empty. For a fully functional submodule, you'll also need to do a pull on it, as described in Section 3.6.3.

3.6.2 Adding, Removing and Synchronizing Submodules

To add a new submodule to a repository, invoke **Remote**|**Submodule**|**Add** on the project root in the **Directories** view and follow the dialog instructions.

To remove a submodule, select the submodule in the **Directories** view, invoke **Remote**|**Submodule**|**Unregister**, and then commit your changes. After the submodule is unregistered, you may delete the submodule directory.

If the URL of a submodule's remote repository has changed, you need to modify the URL in the `.gitmodules` file and then *synchronize* the submodule, via **Remote**|**Submodule**|**Synchronize**, so that the new URL is written into Git's configuration.

3.6.3 Updating Submodules

After a submodule has been set up, the usual workflow is that some files in the submodule repository are modified externally, and you perform an *update* on the submodule, i.e. you pull the new changes into your local submodule repository.

You can perform an update either by doing a pull on the submodule itself, or, if the outer repository is connected to a remote repository, by configuring SmartGit/Hg to automatically update all submodules when you do a pull on the outer repository. These two cases will be described in the following subsections.

Note that in either case, pulling will fetch new commits without changing the submodule if it has a *detached HEAD*. See Section 3.6.4 for more information on the latter.

Pulling on the Submodule

Select the submodule in the **Directories** view and invoke **Remote**|**Pull**. On the Pull dialog that shows up, check either the Rebase or the Merge option.

Then, after the pull, the submodule will have a different appearance in the **Directories** view if new commits have been fetched and a rebase or merge has been performed. This different appearance indicates that the submodule has changed and that you need to commit (see 3.3.3) the change in the outer repository.

Pulling on the Outer Repository

Open the repository settings via **Project|Repository Settings**, and on the **Pull** tab, enable **Update registered submodules**, so that SmartGit/Hg automatically updates all registered submodules when pulling on the outer repository.

Additionally, you may also enable **And initialize new submodules**; with this, SmartGit/Hg will update not only registered submodules when pulling, but also uninitialized submodules, after having initialized them.

The aforementioned Update option will only fetch commits as needed, i.e. when a commit is referenced by the outer repository as the current state of the submodule. If you want to fetch all new commits instead, enable the option **Always fetch new commits, tags and branches from submodule**.

Note that when you do a pull on the outer repository, you need to pull with subsequent rebase or merge, otherwise new submodule commits will only be fetched, without changing the submodule state (i.e. the commit the submodule is currently pointing at).

3.6.4 Working within Submodules

You can view the history of a submodule repository by opening its Log (see 3.5.1). To do so, select the submodule in the **Directories** view and invoke **Log** from the submodule's context menu. You can also restrict the Log to a certain branch within the submodule: Select the submodule in the **Directories** view, then select the submodule branch in the **Branches** view, and then invoke **Log** from the context menu of the branch.

In the submodule Log, you can switch the submodule to another commit by selecting the commit in the Log graph and invoking **Check Out** from the commit's context menu. If you want to switch to the tip of a certain branch, you can also just double-click on the branch in the **Branches** view.

After switching the submodule to another commit, the submodule will be shown as "changed" in the **Directories** view. That means you can either commit the change in the outer repository or roll back the change. For the latter, select the submodule in the **Directories** view and invoke **Reset** from its context menu.

If you modify and commit files within the submodule (as part of the outer repository, not externally), the submodule will also show up as "changed". Then, after committing the changes, you can push them back to the remote submodule repository via **Push** from the context menu of the **Branches** view. Note that you may lose your work in the submodule

if you make changes on a *detached HEAD*. To avoid this, check out a submodule branch before making the changes.

Chapter 4

Directory Tree and File Table

The directory tree and the file table display the status of your working tree (and Index). The primary directory states are listed in Table 4.1, and possible states of submodules in Table 4.2. Every primary and submodule state may be combined with additional states, which are listed in Table 4.3. The possible file states are listed in Table 4.4.

4.1 File Filtering

The **Files** table of the project window can be filtered by file state and name. The state filters can be set using the small toolbar buttons above the table as well as the menu items in the **View** menu. By default, if committable files (e.g. *index-only changed* or *untracked*) are hidden, the background turns to light-red as a reminder to not forget about them. If this annoys you because you permanently have untracked files in your working, you should consider to mark them as ignored. If this is no option for you, you can deactivate this coloring feature in the Preferences (page **User Interface**, option **Use background color for the file table to indicate certain states**).

To filter by name, use the input field (or *<Ctrl/Cmd>+<F>*-keystroke) above the file table. The context menu allows to enable regular expressions and save patterns for later usage or delete them. Even if unchanged files are hidden, they can be found by filtering by name - the files matching by name but not by state are shown in gray, while a light-yellow background indicates the name-filtering state.

4.2 Fixing 'Modified (File Mode)' on Windows

On Windows, the *Modified (File Mode)* state is usually caused due to a misconfiguration of your local repository, when not having `core.filemode` configuration option explicitly set to `false` (the default value is `true`).

Hence, if this option in not present in `.git/config` of your repository, invoke `git config`

Icon	State	Details
	Default	Directory is present in the repository (more precisely: there is at least one versioned file below this directory stored in the repository).
	Unversioned	Directory (and contained files) are present in the working tree, but have not been added to the repository yet. Use **Stage** to add the files to the repository.
	Ignored	Directory is not present in the repository (exists only in the working tree) and is marked as to be ignored.
	Missing	Directory is present in the repository, but does not exist in the working tree. Use **Stage** to remove the files from the repository or **Discard** to restore them in the working tree.
	Conflict	Repository contains conflicting files (only displayed on the root directory). Use **Resolve** to resolve the conflict.
	Merge	Repository is in 'merging' or 'rebasing' state (only displayed on the root directory). Either **Commit** the merge/rebase or use **Discard** to cancel the merge/rebase.
	Root/Submodule	Directory is either the project root or a submodule root, see Table 4.2.

Figure 4.1: Primary Directory States

`core.filemode false` in the root directory of your repository to fix this problem.

Icon	State	Details
	Submodule	Unchanged submodule.
	Modified in working tree	Submodule in working tree points to a different commit than the one registered in the repository. Use **Stage** to register the new commit in the Index, or **Reset** to reset the submodule to the commit registered in the repository.
	Modified in Index	Submodule in working tree points to a different commit than the one registered in the repository, and this changed commit has been staged to the Index. **Commit** this change or use **Discard** to revert the Index.
	Modified in WT and Index	Submodule in working tree points to a different commit than the one in the Index, and the staged commit in the Index is different from the one in the repository. Use either **Stage** to register the changed commit in the Index (overwriting the Index change), **Discard** to revert the Index, or **Reset** to reset the submodule to the commit registered in the Index.
	Foreign repository	Nested repository is not registered in the parent repository as submodule. Use **Stage** to register (and add) the submodule to the parent repository.

Figure 4.2: Submodule States

Icon	State	Details
	Direct Local Changes	There are local (or Index) changes within the directory itself.
	Indirect Local Changes	There are local (or Index) changes in one of the subdirectories of this directory.

Figure 4.3: Additional Directory States

Icon	State	Details
	Unchanged	File is under version control and neither modified in working tree nor in Index.
	Unversioned	File is not under version control, but only exists in the working tree. Use **Stage** to add the file or **Ignore** to ignore the file.
	Ignored	File is not under version control (exists only in the working tree) and is marked to be ignored.
	Modified	File is modified in the working tree. Use **Stage** to add the changes to the Index or **Commit** the changes immediately.
	Modified (Index)	File is modified and the changes have been staged to the Index. Either **Commit** the changes or **Unstage** changes to the working tree.
	Modified (WT and Index)	File is modified in the working tree and in the Index in different ways. You may **Commit** either Index changes or working tree changes.
	Modified (File Mode)	The content of the file is not modified, but the executable bits are set different than in the repository. Refer to Section 4.2 on how to fix that state on Windows.
	Added	File has been added to Index. Use **Unstage** to remove from the Index.
	Removed	File has been removed from the Index. Use **Unstage** to un-schedule the removal from the Index.
	Missing	File is under version control, but does not exist in the working tree. Use **Stage** or **Remove** to remove from the Index or **Discard** to restore in the wirking tree.
	Modified (Added)	File has been added to the Index and there is an additional change in the working tree. Use **Commit** to either commit just the addition or commit addition and change.
	Intent-to-Add	File is planned to be added to the Index. Use **Add** or **Stage** to add actually or **Discard** to revert to unversioned.
	Conflict	A merge-like command resulted in conflicting changes. Use the **Conflict Solver** to fix the conflicts.

Figure 4.4: File States

Chapter 5

Outgoing View

This view shows "outgoing" commits. This usually means unpushed commits, but in the case of a Git-Flow feature or bugfix branch, it also means already pushed commits of this specific branch. In case of submodules commits, they will also be shown. The "Path" column shows the relative location of the commit's repository (usually . for the currently selected repository).

To push only a subset of your local commits, select the latest commit to be pushed and invoke **Push Commits**.

5.1 Interactive Rebase

The Outgoing view supports certain workflows to cleanup the commits. To join adjacent commits, select them and invoke **Join Commits** and provide the new commit message. To reorder commits, just use drag and drop. To change the commit message, select the commit and invoke **Edit Commit Message**.

Note	All those operations need a clean working tree and will not work if merge commits would be affected.

Tip	To just change the commit message of the last commit (even for a merge commit), a clean working tree is not required if you invoke **Local**|**Edit Last Commit Message**.

Chapter 6

Git-Flow

6.1 Overview

Git-Flow is a high-level command set wrapping low-level Git commands to support the "successful branching model" (see http://nvie.com/posts/a-successful-git-branching-model/). It reduces the workflow steps necessary for the user.

To achieve this, Git-Flow assigns a special meaning to its branches. For Git-Flow, there are two main branches which live forever, the "develop" and "master" branch.

6.1.1 Develop Branch

The single "develop" branch (named by default `develop`) contains the ongoing development line. It contains all finished improvements and fixes.

6.1.2 Master Branch

The single "master" branch (named by default `master`) contains the stable release line. Its HEAD represents the latest stable release.

Other branches usually exist only for a certain period of time.

6.1.3 Feature Branches

For each new (non-trivial) improvement which should be added to the ongoing development line, a separate "feature" branch is created (named by default, e.g. `feature/my-feature`). This temporary branch will be used to work independently on this particular improvement ("feature"). If one thinks the feature is done, the commits from the "feature" branch are integrated (either merged or rebased) into the develop (see 6.1.1)

branch and the feature branch will usually be deleted. This way all feature branches in a repository indicate the features which are currently worked on.

```
o ... [> develop] merged feature A
| \
|  o ... a feature commit
|  |
o  | ... a develop commit
| /
o  ... another develop commit
```

6.1.4 Release Branches

To prepare a (planned) software release, a temporary "release" branch is created from the develop (see 6.1.1) branch. The "release" branch is usually forked when all features for the upcoming release have been implemented and the develop branch is in "feature-freeze" state. Thus, it makes the release independent of further improvements of the develop branch and hence allows to "harden" the release by fixing bugs. When the state of this branch is ready for official release, it will be tagged and merged into both the master (see 6.1.2) and the develop (see 6.1.1) branch, this way creating a new release build to be made available to your customers (e.g. "version 4"). After successful merging, the release branch usually is deleted.

```
o ... [> develop] merged release 4_0
| \
|  \   o ... [master] ... release 4_0
|   |/ |
|   o  | ... <tag/release-4_0_0] a release-preparing commit (e.g. bug-fix)
|   |  |
o  /   | ... a develop commit for a future release
| /    |
o      | ... another develop commit
|      |
|      o ... release 3_0_9
|     /|
```

6.1.5 Hotfix Branches

If after an official release a serious bug is detected, a "hotfix" branch will be created from the latest release state (the HEAD of the master (see 6.1.2) branch). After fixing the bug(s) in this hotfix branch, the state will be tagged and merged into both the master (see 6.1.2) and the develop (see 6.1.1) branch, this way creating a new build to be made

available to your customers (bugfix release, e.g. "version 4.0.1"). After successful merging, the hotfix branch usually is deleted.

```
o ... [> develop] merged hotfix 4_0_1
| \
|  \   o ... [master] ... release 4_0_1
|   |/ |
|   o  | ... <tag/release-4_0_1] a serious bug-fix
|   |  |
o    \ | ... a develop commit for a future release
|     \|
|      o ... release 4_0
|     /|
```

6.1.6 Support Branches

Support branches are still in "experimental" state, according to the Git-Flow documentation. Nevertheless, they are used if you have multiple older releases (e.g. "version 3.0.*") which are still supported while the head of the master represents the latest release (e.g. "version 4.0.*"). Changes in support branches may be unique to the support branch, because the code in the latest release is not present anymore or the bug/improvement has been implemented there already. If a commit from a support branch should still be integrated into the latest release, open a hotfix (see 6.1.5) branch, cherry-pick (see 3.4.5) the commit and finish the hotfix.

Usually, feature branches are created by developers, whereas release, hotfix and support branches are created by the release manager.

6.2 Git-Flow Commands

6.2.1 Configure

Use this command before starting to use Git-Flow. You can use the default branch naming or change it according to your needs. This will write the Git-Flow configuration to `.git/config` of your repository.

If there is a `.gitflow` file in the root of your working tree, the default values will be read from this file. When cloning a repository which already contains the `.gitflow` file, Git-Flow will be initialized automatically. This allows a quick Git-Flow configuration for each of your team-members even if you use a non-default Git-Flow branch naming scheme.

6.2.2 Start Feature

Use this command to start the work on a new feature (see 6.1.3). After providing a name for the feature, the corresponding feature branch will be forked off the `develop` branch and this new feature branch will be checked out.

Note	If the `develop` branch is currently check out, the **Flow** toolbar button defaults to this command.

6.2.3 Finish Feature

Use this command if you have committed your changes necessary for the feature and want to integrate them into main development line. There are 3 ways of doing this: by creating a merge commit (your feature commits will be preserved), by creating a simple commit (all your feature commits will be squashed into one commit) or by using rebase (your feature commits will be re-created on top of the `develop` branch). When merging or squashing, you need to enter the commit message for the new commit. Usually, you need to push the `develop` branch later.

6.2.4 Integrate Develop

If new commits were created in the `develop` branch after you've created a feature branch, you may use this command to get the changes from the `develop` branch into your feature branch. You have the choice between using merge (which will create a merge commit in your feature branch) or rebase (your feature branch commits will be re-created on top of the latest develop commit).

The default operation is determined from your repository settings (**Project**|**Repository Settings**). Rebasing might not be available in case of merge commits, though.

6.2.5 Start Hotfix

Use this command to prepare a new bugfix release version from the latest release version (HEAD of the `master` branch) without using any new changes from the `develop` branch. This will create a hotfix branch from the `master` branch using the given hotfix name.

6.2.6 Finish Hotfix

Use this command if you have prepared some changes for the new bugfix release version and want to make it publicly available. This will tag the hotfix branch, merge it to the `master` and `develop` branch.

6.2.7 Start Release

Use this command to prepare a release, independent of further changes in the `develop` branch. This will create a release branch from the `develop` branch using the given release name.

6.2.8 Finish Release

Use this command if you have prepared changes for the release and want to make it publicly available. This will tag the release branch, merge it into the `master` and `develop` branch.

6.2.9 Start Support Branch

Use this command to create a support branch from the `master` branch. There is no corresponding **Finish Support** command available, as support branches live forever.

6.3 Migrating from the 'master-release-branch' workflow

A common workflow and repository structure is to have a `master` in which all development takes place and once it comes to a release of the software, a `release`-branch is forked off from the master. This `release`-branch represents the stable (production-ready) state of the software at its current version, lives forever and all bug-fixing for this specific software version happens in that `release`-branch only. From time to time the `release` branch is merged into the `master`.

6.3.1 Migration

Let's assume a project which has an active `master` and release branches `release-1` ... `release-4` for the already released versions *1* ... *4* of the software. A good occasion to switch to Git-Flow will be immediately before the release of upcoming version *5*:

- Fork `develop` from `master` and tell all your team-members to continue their development in `develop`. Directly committing to `master` is not allowed anymore.
- When the development of version *5* is in feature-freeze state, start a Release branch (see 6.1.4) called `release/5` , continue with work on the next version *6* in `develop`, bring `release/5` to production quality and finally "finish" the release.

The mapping from the *old* `master` to the Git-Flow `develop`-branch is straight-forward. The interesting point now is how to proceed with bug-fixes for already released versions:

Old release branches become 'support' branches

The old branches `release-1` ... `release-4` are actually Support branches (see 6.1.6) and should be renamed to `support/release-1` ... `support/release-4`, hence.

6.3.2 Usage

Once the branches have been migrated, you now can adopt the Git-Flow branching model, which does not know about long-living *release*-branches anymore.

Hotfix branches are used instead of a 'current-release' branch

Once the first problem needs to be fixed for version *5*, start a hotfix branch (see 6.1.5) called `hotfix/5.0.1` and apply the fix there. The `hotfix/5.0.1` branch will remain open until you decide to officially release bug-fix version *5.0.1*. Only then, this hotfix will be *finished* what results in a corresponding merge commit in `master`. Once a new problem needs to be fixed in version *5* series, create a new hotfix from `hotfix/5.0.2` which will automatically be forked off the `5.0.1` merge commit in `master`. In this way, your master will proceed from version *5* release, to *5.0.1*, *5.0.2*, ...

If there is a serious problem in e.g. version *5.0.2*, which needs to be fixed immediately and `hotfix/5.0.3` is already in progress, do the following:

- start another hotfix branch `hotfix/5.0.2a`, which is forked off from `5.0.2` commit of `master`, as `hotfix/5.0.3` is,
- apply the fix and
- finish the `hotfix/5.0.2a` immediately (and make the 5.0.2a version of the software public)

Now, `master` will contain a top-most `5.0.2a` commit, derived from `5.0.2` commit. When finishing `hotfix/5.0.3`, the resulting `5.0.3` commit in `master` will be derived from the `5.0.2a` and have the `hotfix/5.0.3` merged in, i.e. it will represent the changes from both versions, *5.0.2a* and *5.0.3*. That's exactly what you would like to release now as *5.0.3* version.

Maintaining older versions

Hopefully, you won't need to apply many changes to older released versions. If you still need to, apply these changes to the corresponding `support/release-` branches and decide

whether these changes should go into the current release as well: if not, you are all set now. If they should, cherry-pick (see 3.4.5) the corresponding commits into the latest `hotfix/5.0.x` branch. There should be one such branch only, anyway. In this way, the changes will make it to `master` and `develop` later, when the hotfix is "finished".

Chapter 7

Advanced Settings

In addition to the options in the preferences dialog, SmartGit/Hg has some advanced settings that can be set through a configuration file named `smartgit.properties` or through command-line parameters. Both are covered in the following subsections.

Moreover, there are two special settings, the location of the settings directory and the program's memory limit. Both of these special settings will be described in their own subsections.

Note	The file `smartgit.properties` contain only settings for SmartGit/Hg itself. If you want to configure your Git repositories, have a look at the various Git configuration files instead, such as `.git/config` for the configuration of individual Git repositories, and `~\.gitconfig` (in your HOME directory) for global configuration options.

7.1 System Properties

SmartGit/Hg can be configured by editing the file `smartgit.properties` in the settings directory. The `smartgit.properties` file contains further documentation about the available settings, so the latter will not be listed here. In this section, we will only show an example in order to give a general idea of how to alter settings in the `smartgit.properties` file.

First, open the settings directory. Its default location is described in Section 8.1. In the settings directory, you will find the `smartgit.properties` file. Open it with a text editor, such as Windows Notepad.

Each of the settings in `smartgit.properties` is specified on a separate line, according to the following syntax: `key=value`

If a line starts with `#`, the entire line is treated as a comment and ignored by the program. By default, the available settings are prefixed with a `#`, so that their default values will

be used. To alter a setting, uncomment it by removing the `#` character and modify the setting's value as needed.

Example
In the `smartgit.properties` file, uncomment the following line in order to disable the splash screen:
`#smartgit.ui.splashscreen=false`

7.2 Command-Line Options

This section gives an overview of the various options SmartGit/Hg can be started with. These options should be given as parameters to the SmartGit/Hg launcher. The launcher to be used depends on your platform:

- **Windows**: `bin\smartgithg.exe` or `bin\smartgithgc.exe`. The first one is meant for regular usage, while the second one will print additional information on the console while the program runs.
- **Mac OS X**: `SmartGitHg 4.5.app/Contents/MacOS/SmartGit`
- **Linux**: `bin/smartgithg.sh`

In the following, we'll use `smartgithgc.exe` as an example to explain the available options. Substitute it with the respective launcher for your platform if you're not using Windows.

There may be additional options available that mainly serve debugging purposes and are therefore not documented here.

7.2.1 Options "-?" and "–help"

With either of the two following commands you can print all command-line options on the console that are specifically supported by the version of SmartGit/Hg you're using:

Example
```
smartgithgc.exe -?
smartgithgc.exe --help
```

Note	On Windows, make sure to call `smartgithgc.exe` (with 'c' on the end), otherwise when calling `smartgithg.exe` this parameter has no effect, since the SmartGit/Hg process won't be attached to any console to print the help output to.

7.2.2 Option "–cwd"

This option sets the current working directory, which affects the path given in the `open`, `log` and `blame` option (see below) as follows:

- If the `open`, `log` or `blame` options are specified without their own path arguments, the path given with the `cwd` option will be used as argument for `open` or `log`.
- If the `open`, `log` or `blame` options are specified with relative paths, these relative paths will be resolved against the path given with the `cwd` option.
- If the `open`, `log` or `blame` options are specified with absolute paths, the path given with the `cwd` option is ignored.

The path given with the `cwd` option must be an absolute path. If the path is relative, it will be ignored.

7.2.3 Option "–open"

This option launches SmartGit/Hg and opens the repository in the specified location.

Example
```
smartgithgc.exe --open C:\path\to\repository
```

Example
```
smartgithgc.exe --cwd C:\path --open to\repository
```

7.2.4 Option "–log"

This option opens SmartGit/Hg's Log window for the repository or file in the specified location. The project window is not opened.

Example
```
smartgithgc.exe --log C:\path\to\repository\path\to\file
```

Example
```
smartgithgc.exe --cwd C:\path --log to\repository
```

7.2.5 Option "–blame"

This option opens SmartGit/Hg's Blame window for the specified file.

Example
```
smartgithgc.exe --blame C:\path\to\repository\path\to\file
```

7.3 Location of the Settings Directory

The settings directory is where SmartGit/Hg will store its settings. See Section 8 for information about the default location and contents of the settings directory. On Windows and Linux, you can change its location by modifying the system property `smartgit.settings`. Changing the settings directory's location is *not* supported on Mac OS X.

Within the value of `smartgit.settings`, certain Java system properties are allowed, such as `user.home`. Another accepted value is the special `smartgit.installation` property, which refers to the SmartGit/Hg installation directory.

Example
To tell SmartGit/Hg to store its settings in the subdirectory `.settings` of the SmartGit/Hg installation directory, you can set `smartgit.settings` to the following value:
`smartgit.settings=${smartgit.installation}\.settings`

How the `smartgit.settings` property is set depends on your platform:

7.3.1 Windows

All Users

In the file `bin/smartgithg.vmoptions` inside the SmartGit/Hg installation directory there is a line that looks like this:

`-Dsmartgit.settings=${smartgit.installation}\.settings`

Replace the path given after the `=` character with a path of your choice.

Current User

The settings directory specified in `bin/smartgithg.vmoptions` can be overridden on a per-user basis. To do so, create a file named `vmoptions` in the directory `syntevo\SmartGit` inside the application data directory. The location of the latter depends on the Windows version:

- XP: `C:\Documents and Settings\[Username]\Application Data`
- Vista/7: `C:\Users\[Username]\AppData`

In the newly created `vmoptions` file, insert a line that sets the settings directory, e.g. `-Dsmartgit.settings=C:\SmartGitHg\.settings`.

7.3.2 Linux

Near the end of the file `bin/smartgit.sh`, there should be a line that looks like this:

```
#_VM_PROPERTIES="$_VM_PROPERTIES -Dsmartgit.settings=...
```

Uncomment this line by removing the `#` character at the beginning, then insert a path of your choice after the `-Dsmartgit.settings=` part.

7.4 Location of the Updates Directory

The *Updates* directory is where SmartGit/Hg will store downloaded program updates. See Section 8 for information about the default location and contents of the *Updates* directory. On Windows and Linux, you can change its location by modifying the system property `smartboot.sourceDirectory`.

The configuration of the *Updates* directory depends on the platform and works identical as for the Settings directory (see 7.3.1). Relative paths are supported and resolved against SmartGit/Hg's installation root directory.

Example
On Windows, to tell SmartGit/Hg to store the updates in the directory `d:/smartgit-updates`, you have to add following line to the file `bin/smartgithg.vmoptions` inside the SmartGit/Hg installation directory:
`-Dsmartboot.sourceDirectory=d:/smartgit-updates`

7.5 Memory Limit

The memory limit (also known as maximum heap size) specifies how much RAM the SmartGit/Hg process is allowed to use. If the set value is too low, SmartGit/Hg may run out of memory during memory-intensive operations. How the memory limit is set depends on your operating system:

- **Windows (all users)**: In the file `bin/smartgithg.vmoptions` inside the SmartGit/Hg installation directory, there is a line that looks like this: `-Xmx256m`. This sets a memory limit of 256 MB. To set a memory limit of 512 MB, change this to `-Xmx512m`.
- **Windows (current user)**: The memory limit specified in `bin/smartgithg.vmoptions` can be overridden on a per-user basis. To do so, create a file named `vmoptions` in the directory `syntevo\SmartGit` inside the application data directory. The location of the latter is usually either `C:\Documents and Settings\[Username]\Application Data` (for Windows 2000/XP) or `C:\Users\[Username]\AppData`

(for Windows Vista/7). In the newly created `vmoptions` file, insert a line that specifies the memory limit, e.g. `-Xmx512m` for a memory limit of 512 MB.

- **Mac OS X**: Set the environment variable `SMARTGITHG_MAX_HEAP_SIZE` to the desired value, e.g. `512m` for a memory limit of 512 MB. One way to set this variable for all users is to open the file `/etc/launchd.conf` with root priviledges (creating it if it doesn't exist) and to add the following line: `setenv SMARTGITHG_MAX_HEAP_SIZE 512m`. Alternatively, you may edit `SmartGitHg 4.5.app/Contents/MacOS/SmartGit` by changing the value of the `SMARTGITHG_MAX_HEAP_SIZE=256m` line.

- **Linux**: Set the environment variable `SMARTGITHG_MAX_HEAP_SIZE` to the desired value, e.g. `512m` for a memory limit of 512 MB. One way to set this variable for all users is opening the file `/etc/profile` with root priviledges and adding the following line at the end (after `unmask xxx`): `export SMARTGITHG_MAX_HEAP_SIZE=512m`.

Chapter 8

Installation and Files

SmartGit/Hg stores its settings files per-user. Each major SmartGit/Hg version has its own default settings directory, so you can use multiple major versions independent of each other. The location of the settings directory depends on the operating system.

8.1 Default Location of SmartGit/Hg's Settings Directory

- **Windows**: `%APPDATA%\syntevo\SmartGit\<major-smartgit-hg-version>` (`%APPDATA%` is the path defined in the environment variable `APPDATA`)
- **Mac OS**: `~/Library/Preferences/SmartGit/<major-smartgit-hg-version>`
- **Linux/Unix**: `~/.smartgit/<major-smartgit-hg-version>`

Tip	You can change the directory where the settings files are stored by changing the property smartgit.settings (see 7.3).

8.2 Notable Files in the Settings Directory

- `license` stores your SmartGit/Hg *license key*.
- `log.txt` contains debug log information. It can be configured via `log4j.properties`. You may remove this file: afterwards, SmartGit/Hg will return to its default logging settings.
- `passwords` is an encrypted file and stores the *passwords* used throughout SmartGit/Hg. You may remove this file: afterwards, all passwords are lost.

- `accelerators.xml` stores the *accelerators* configuration. You may remove this file: afterwards, all accelerators will be reset to their defaults.
- `credentials.xml` stores authentication information (not including the corresponding passwords). You probably do not want to remove this file: afterwards, all credentials (user names, private keys, certificates) will be lost.
- `hostingProviders.xml` stores information about configured hosting provider accounts (not including the corresponding passwords). You probably do not want to remove this file: afterwards, all connect details for all hosting provides will be lost.
- `notifications.xml` stores information about the state of notifications which show up in the status bar in various cases. You may remove this file: afterwards, various notifications may show up again.
- `projects.xml` stores all configured *projects* including their settings. You should not remove this file, unless you want to completely reset SmartGit/Hg.
- `settings.xml` stores the application-wide settings (e.g. the preferences) of SmartGit/Hg. You should not remove this file, unless you want to completely reset SmartGit/Hg.
- `tools.xml` stores *external* tools which have been configured in the Preferences. You probably do not want to remove this file: afterwards, all you external tools configurations will be lost.
- `ui-config.xml` stores UI related, more stable settings, e.g. the toolbar configurations. You may remove this file: afterwards, various aspects of the UI will be reset to defaults.
- `ui-settings.xml` stores UI related, volatile settings, e.g. window sizes or column widths. You may remove this file: afterwards, various aspects of the UI will be reset to defaults.

8.3 Program Updates

SmartGit/Hg stores program updates which have been downloaded automatically through SmartGit/Hg itself by default in the subdirectory `updates` of the *Settings* root directory (see Section 8.1).

The root directory of the *Updates* directory contains sub-directories for every major version. Such a *major version* directory contains sub-directories for the latest downloaded builds and some control files, of which `current` is the most interesting one: it points to the currently used build number (and hence sub-directory to take the binaries from). Usually, this will be the highest build number which shows up in this directory.

> **Warning!** By modifying the `control` file or any other contents within the *Updates* directory, you may easily screw up your SmartGit/Hg installation. Hence, do not touch these files unless you have good reasons to do so.

8.4 Company-wide Installation

For company-wide installations, the administrator may install SmartGit/Hg on a read-only location or network share. To ease deployment and initial configuration for the users, certain settings files can be prepared and put into a directory named `default`. For Mac OS X this `default` directory must be located in `SmartGit.app/Contents/Resources/` (parallel to the `Java` directory), for other operating systems within SmartGit/Hg's installation directory (parallel to the `lib` and `bin` directories).

When a user starts SmartGit/Hg for the first time, the following files will be copied from the `default` directory (on the network share) to the user's personal SmartGit/Hg settings directory (refer to Section 8.1):

- `accelerators.xml`
- `credentials.xml`
- `hostingProviders.xml`
- `projects.xml`
- `settings.xml`
- `tools.xml`
- `ui-config.xml`
- `ui-settings.xml`

The `license` file (only for *Enterprise* licenses and 10+ users *Professional* licenses) can also be placed into the `default` directory. In the latter case, SmartGit/Hg will prefill the **License** field in the **Set Up** wizard when a user starts SmartGit/Hg for the first time. When upgrading SmartGit/Hg, this `license` file will also be used, so users won't be prompted with a "license expired" message, but can continue working seamlessly.

> **Note** Typically, you will receive license files from us wrapped into a *ZIP* archive. In this case you have to unzip the contained `license` file into the `default` directory.

8.5 JRE Search Order (Windows)

On Windows, the `smartgithg.exe` launcher will search for a suitable JRE in the following order (from top to bottom):

- Environment variable SMARTGITHG_JAVA_HOME
- Subdirectory `jre` within SmartGit/Hg's installation directory
- Environment variable JAVA_HOME
- Environment variable JDK_HOME
- Registry key HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment

Chapter 9

SVN Integration

9.1 Overview

Git allows you to interact not only with other Git repositories, but also with SVN repositories. This means that you can use SmartGit/Hg like an SVN client:

- Cloning from an SVN repository is similar to checking out an SVN working copy.
- Pulling from an SVN repository is similar to updating an SVN working copy.
- Pushing to an SVN repository is similar to committing from an SVN working copy to the SVN server.

In addition to the SVN functionality, you can use all (local) Git features like local commits and branching. SmartGit/Hg performs all SVN operations transparently, so you will notice only a few points in the program where you need to understand which server VCS you are using.

9.2 Compatibility and Incompatibility Modes

SmartGit/Hg's SVN integration is available in two modes:

- **Normal Mode**: This is the recommended mode of operation. It is used by default when a repository is freshly cloned with SmartGit/Hg (not with *git-svn*). All features are supported in this mode. The created repositories are not compatible with *git-svn*.
- **git-svn Compatibility Mode**: In the git-svn compatibility mode (or just "compatibility mode") SmartGit/Hg can work with repositories that were created using the *git-svn* command. In this mode advanced features like *EOLs*-, *ignores*- and

externals-translation are turned off. The SVN history is processed in a similar way as by *git-svn*.

9.3 Ignores (Normal Mode Only)

SmartGit/Hg maps `svn:ignore` properties to `.gitignore` files. Unlike the `git svn create-ignore` command SmartGit/Hg puts `.gitignore` files under version control. If you modify a `.gitignore` file and pushes the change, the corresponding `svn:ignore` property is changed.

The `.gitignore` syntax is significantly more powerful than the `svn:ignore` syntax. Hence, `svn:ignore` can be mapped losslessly to `.gitignore`, however a `.gitignore` file may contain a pattern that can't be mapped backed to `svn:ignore`. In that case the pattern is not translated.

Adding or removing a recursive pattern in `.gitignore` corresponds to setting or unsetting that pattern on every existing directory in the SVN repository. Conversely, when an SVN revision is fetched (back) into the Git repository, a recursive pattern will be translated to a set of non-recursive patterns, one pattern for each directory.

Example
Let's assume we have the following directories in the SVN repository:

```
A {
  B {}
  C {}
}
```

And we add `.gitignore` with only one line:

```
somefile
```

and push. This will set the `svn:ignore`-property to `somefile` for all directories: *A*, *B*, *C*. After fetching such a revision we have the following `.gitignore` contents (ordering of lines is unimportant):

```
A/somefile
A/B/somefile
A/C/somefile
```

Git doesn't support patterns that contain spaces. Hence, SmartGit/Hg replaces all spaces in the `svn:ignore` value with `[!!-~]` during the creation of `.gitignore`. Conversely, all newly added patterns containing `[!!-~]` are converted to `svn:ignore` with spaces at the corresponding places.

9.4 EOLs (Normal Mode Only)

When using *git-svn* on Windows, different EOLs on different systems may cause trouble. For instance, if the SVN repository contains a file with `svn:eol-style` set to `CRLF`, its content is stored with CRLF line endings. Moreover, *git-svn* puts the file contents directly into Git blobs without modification. Now, if you have the `core.autocrlf` Git option set to `true`, it may be impossible to get a *clean* working tree, and hence `git svn dcommit` won't work. This happens because while checking whether the working tree is *clean*, Git converts working tree file EOLs to *LF* and compares with the blob contents (which has *CRLF*). On the other hand, setting `core.autocrlf` to `false` causes problems with files that contain *LF* EOLs.

Instead of setting a global option, SmartGit/Hg carefully sets the EOL for every file in the SVN repository using its `svn:eol-style` and `svn:mime-type` values. It uses the versioned `.gitattributes` file for this purpose. Its settings have higher priority than the *core.autocrlf*-option, so with SmartGit/Hg it doesn't matter what the *core.autocrlf* value is.

> **Warning!** The `.git/info/attributes` file has higher priority than the versioned `.gitattributes` files, so it is strongly recommended to delete the former or leave it empty. Otherwise, this may confuse Git or SmartGit/Hg.

By default, a newly added text file (or more precisely, a file that Git thinks is a text file) which is pushed has `svn:eol-style` set to `native` and no `svn:mime-type` property set. A newly added binary file has no properties at all.

One can control individual file properties using the `svneol` Git attribute. The syntax is `svneol=<svn:eol-style value>#<svn:mime-type value>`, so for example

```
*.c svneol=LF#unset
```

means all `*.c` files will have `svn:eol-style=LF` and no `svn:mime-type` set after pushing. Recursive attributes are translated like recursive ignores: Their changes result in changes of properties of all files in the SVN repository.

9.5 Externals (Normal Mode Only)

SmartGit/Hg maps `svn:externals` properties to its own kind of submodules, that have the same interface as Git submodules.

> **Note** Only externals pointing to directories are supported, not externals pointing to individual files ("file externals").

SVN submodules are defined in the file `.gitsvnextmodules`, which has the following format:

```
[submodule "path/to/submodule"]
  path = path/to/submodule
  owner = /
  url = https://server/path
  revision = 1234
  branch = trunk
  fetch = trunk:refs/remotes/svn/trunk
  branches = branches/*:refs/remotes/svn/*
  tags = tags/*:refs/remote-tags/svn/*
  remote = svn
  type = dir
```

- **path**: specifies the submodule location relative to the working tree root.
- **owner**: specifies the SVN directory that has a corresponding svn:externals property. The owner directory should be a parent of the submodule location. If the owner is the root of the parent repository itself, the option should be set to "/".
- **url**: specifies the SVN URL to be cloned there (svn:externals syntax can be used here) without a certain branch.
- **revision**: specifies the revision to be cloned. Absence of this option or using *HEAD* means the latest available revision.
- **fetch, branches, tags**: they all specify the SVN repository layout and have the same meaning as the corresponding *git-svn* options of `.git/config`.
- **branch**: specifies the branch to checkout. It should be a path relative to the URL of the *url* option, and it must be consistent with the SVN repository layout. The *empty* branch (if `fetch=:refs/remotes/git-svn`) should be specified using slash `/`.
- **remote**: specifies the name of the *svn-git-remote* section of the submodule.
- **type**: specifies the type of the submodule (default: `dir`). In practice, this is usually a directory. If svn:externals points to a file, this option should have the value `file`.

Changes in `.gitsvnextmodules` are translated to the SVN repository as changes in `svn:externals` and vice versa.

There are two types of SVN submodules between which you can choose during *submodule initialization*:

- **snapshot submodules**: contain exactly one revision of the SVN repository. They are useful in those cases where the external points to a third party library that is not changed as part of the project (parent repository).
- **normal submodules**: are completely cloned repositories of the corresponding externals. It's recommended to use them when working in both the parent repository and the submodule repository.

SmartGit/Hg shows the repository status in the **Directories** pane. If the submodule's current state does not exactly correspond to the state defined by `.gitsvnextmodules` (same URL, revisions, ...), it will show ups as *modified*. In this case, can use **Local**|**Stage** to update the `.gitsvnextmodules` configuration to the current SVN submodule state or you can use **Remote**|**Submodule**|**Reset** to put the submodule back into the state as it is registered in `.gitsvnextmodules`.

9.6 Symlinks and Executable Files

Symlink processing and *executable*-bit processing work in the same way as in *git-svn*. SVN uses the `svn:special` property to mark a file as being a *symlink*. Then its content should look like this:

```
link path/to/target
```

Such files are converted to *Git symlinks*. In a similar way, files with `svn:executable` are converted to *Git executable* files and vice versa.

9.7 Other SVN properties (Normal Mode Only)

SmartGit/Hg maps all other properties which do not have a special meaning in the Git world to *Git attributes*.

Depending on the size of the property value, SmartGit/Hg may store the entire property definition (name and value) just as an attribute in `.gitattributes` or it may decide to store the property value in a separate file. In the latter case, `.gitattributes` will contain an attribute `attr` which is just *set*, i.e. has no value, denoting the presence of the property for the corresponding file. The value will be stored in `.gitsvnattributes/path/to/entry/attr` instead, where `path/to/entry` is the same path as in `.gitattributes`, i.e. the working tree path of the file or directory to which the property belongs. The mapping between SVN property name and Git attribute name depends on the property name:

- A property which belong to the *svn:*-namespace and does not yet have a special mapping (as explained in the previous sections) will be mapped to a Git attribute

starting with the `svn_`-prefix. This means, that property `svn:keywords` will be mapped to attribute `svn_keywords` and `svn:needs-lock` will be mapped to `svn_needs-lock`.

- For all other properties (custom properties), an SVN property with name `foo` will be mapped to a Git attribute with name `svnc_foo`.

9.7.1 Adding a property

If the property value is a small, one-line text property, you may add it directly to `.gitattributes`.

Example
To add custom property *foo* with value *bar* to file `file.txt`, insert following line into `.gitattributes` (or add the attribute to an already existing line for `/file.txt`):

```
/file.txt svnc_foo=bar
```

If the property value is large, consists of multiple lines or is even of binary content, you have to add the *control* entry to `.gitattributes` and create the file `.gitsvnattributes/path/to/entry/attr` containing the property's value.

Example
To add a binary property *foo* with some binary value to file `file.txt`, insert following line into `.gitattributes` (or add the attribute to an already existing line for `/file.txt`):

```
/file.txt svnc_foo
```

and add file `.gitattributes/file.txt/svnc_foo` with the desired binary property content.

9.7.2 Modifying a property

Depending on the size of the property value, the value will either be stored directly in `.gitattributes`, or in `.gitsvnattributes`, as explained before. To modify the value, locate either of the two places where it is stored and modify the value there. After having committed this change and pushed to SVN, the modified property value will show up in the SVN repository.

Example
To change a small, one-line property value *foo* for `file.txt` which is stored in `.gitattributes` to a larger, multi-line or binary property value, replace the Git attribute value by the Git attribute name itself (i.e. mark it as *set*):

```
/file.txt svnc_foo
```

and add file `.gitattributes/file.txt/svnc_foo` with the desired property content.

9.7.3 Removing a property

Depending on the size of the property value, the value will either be stored directly in `.gitattributes`, or in `.gitsvnattributes`, as explained before. To remove the property, remove the corresponding attribute from `.gitattributes` and remove the corresponding file from `.gitsvnattributes`, if it's present there. After having committed this change and pushed to SVN, the property deletion will show up in the SVN repository.

9.8 Tags

Unlike *git-svn*, SmartGit/Hg creates Git tags for SVN tags. If an SVN tag was created by a simple directory copying, SmartGit/Hg creates a tag that points to the copy-source; otherwise SmartGit/Hg creates a tag that points to the corresponding commit of `refs/remote-tags/svn/<tagname>`. Git tags can be sent back to the SVN repository (as SVN tags) by right-clicking the tag in the **Branches** view and invoking **Push**.

Note	Git tags that are actually *objects* on their own (not just simple *refs*) are not supported.

9.9 History Processing

9.9.1 Branch Replacements

In *compatibility mode*, SmartGit/Hg processes the SVN history like *git-svn* does, with the difference that SmartGit/Hg doesn't support the `svk:merge` property. In the case where one SVN branch was replaced, SmartGit/Hg and *git-svn* create a *merge-commit*.

In *normal mode*, SmartGit/Hg uses its own way of history processing: In case of branch replacements no *merge commit* is created; instead a Git reference `refs/svn-attic/svn/<branch name>/<the latest revision where the branch existed>` is created.

Tip Though that functionality should be used with care, it is easy to create a branch replacement commit from SmartGit/Hg:

- Use **Local|Reset** to reset to some other commit
- Invoke **Remote|Push**. SmartGit/Hg will ask whether the current branch should be replaced.

9.9.2 Merges

Translating Merges from SVN to Git

Completely merged SVN branches correspond to merged Git branches. In particular, for SVN revisions that change `svn:mergeinfo` in such a way that some branch becomes completely merged, SmartGit/Hg creates a Git *merge-commit*. For branches which have not been completely merged, no *merge-commit* is created.

Translating Merges from Git to SVN

Pushing Git *merge-commits* results in a corresponding `svn:mergeinfo` modification, denoting that the branch has been completely merged.

9.9.3 Cherry-picks

SmartGit/Hg supports translation of two kinds of cherry-pick merges between SVN and Git:

- either done using SmartGit/Hg,
- or done using another Git client, without the `--no-commit` option, to make sure that the commit message meta info (`git-svn: ...`) is preserved.

Only cherry-picks of Git commits that correspond to (already pushed) SVN revisions (but not local commits) are supported. Pushing of a cherry-pick commit results in a corresponding `svn:mergeinfo` change.

9.9.4 Branch Creation

SmartGit/Hg allows to create SVN branches simply by pushing locally created Git branches. In this case, SmartGit/Hg will ask you to configure the branch for pushing.

Note	SmartGit/Hg always creates a separate SVN revision when creating a branch, which contains purely the branch creation. This helps to avoid troubles when merging from that branch later.

9.9.5 Anonymous Branches

Anonymous branches show up very often in Git repositories where the default Pull behavior is *merge* instead of *rebase*. Such branches are not mapped back to SVN, as *anonymous SVN branches* are not supported. For instance, the following history:

```
  E-F
 /   \
A-B-C-D-G-H (branch)
```

will be pushed as a linear list of commits: *A,B,C,D,G,H*. *E* and *F* won't be pushed at all.

9.10 The Push Process

Pushing a commit consists of 3 phases:

- sending the commit to SVN
- fetching it back
- replacing the existing local commit with the commit being fetched back

Note that not only the local commit is replaced but also all commits and tags that depend on it. For example, if there is a local commit with a Git tag attached to it, after pushing the Git tag will be moved to the commit that has been fetched back from the SVN repository.

The pushing process requires the working tree to be clean to start, and it uses the working tree very actively during the whole process. Hence, it is *STRONGLY RECOMMENDED* not to make any changes in the working tree during the pushing process, otherwise these changes may become discarded.

Sometimes it is impossible to replace the existing local commit with the commit being fetched back, because other commits (from other users) might have been fetched back as well, containing changes that conflict with the remaining local commits. In this case, SmartGit/Hg leaves the working tree clean and asks you whether to resolve the problem. The easiest way to do so is to press Pull with the **Rebase** option turned on, which will start the rebase process.

Example
The last repository revision is *r10*. There are 2 local commits *A* and *B* that will be pushed. First, *A* is sent, resulting in revision *r12*, because in the meantime someone else had committed *r11*. Now, *r11* and *r12* (corresponding to local commit *A*) are fetched back. Let's assume that *r11* and local commit *B* contain changes for the same file in the same line. Hence, replacing *A* by the fetched-back commits *r11* and *r12* won't work, because the changes of *B* are conflicting now and can't be applied on top of *r12*.

9.11 SVN Support Configuration

9.11.1 SVN URL and SVN Layout Specification

In *compatibility mode*, `.git/config` is used for the specification of the SVN URL and the SVN repository layout. In *normal mode*, SmartGit/Hg uses the file `.git/svn/.svngit/svngitkit.config` for this purpose.

In *compatibility mode*, the SVN URL and SVN layout are specified in the `svn-remote` section. In *normal mode*, the corresponding section is called `svn-git-remote`.

The section generally looks like this:

```
[svn-git-remote "svn"]
url = https://server/path
rewriteRoot = https://anotherserver/path
fetch = trunk:refs/remotes/svn/trunk
branches = branches/*:refs/remotes/svn/*
additional-branches = path/*:refs/remotes/*;another/path:refs/remotes/another/branch
tags = tags/*:refs/remote-tags/svn/*
```

- **url**: specifies the physical SVN URL with which to connect to the SVN repository.
- **rewriteRoot**: specifies the URL to be used in the Git commit messages of fetched commits. If this option is omitted, it is assumed to be the same as the value of the *url* option. The *rewriteRoot* option is useful for continuing working with the repository after the original SVN URL has been changed (in this case *rewriteRoot* should be changed to the old SVN URL value).
- **fetch, branches, additional-branches, tags**: specify pairs consisting of an SVN path and a Git reference for various interesting paths in the SVN repository. All paths beyond these won't be considered by SmartGit/Hg. There's practically no difference between these options. Options *fetch*, *branches*, *tags* are supported by `git-svn` and allow only 1 pair. Option *additional-branches* is only supported by SmartGit/Hg and allows an arbitrary number of `;`-separated pairs. Option *fetch* for *compatibility mode* defines the branch to be checked out and configured as tracked

after fetch. SmartGit/Hg doesn't support `git-svn` patterns in the config and only allows the usage of asterisks (`*`). The number of asterisks in the SVN path and Git reference pattern must be equal. No patterns except the *fetch* pattern must intersect.

9.11.2 Translation Options

SmartGit/Hg keeps all translation options in the `core` section of the file `.git/svn/.svngit/svngitkit.config`.

The section looks like this:

```
[core]
processExternals = true
processIgnores = true
processEols = true
processTags = true
processOtherProperties = true
gitSvnAttributesThreshold = 32
```

These boolean options specify whether SmartGit/Hg should enable special handling of `svn:externals`, `svn:eol-style/svn:mime-type`, `svn:ignore`, SVN tags and all other SVN properties, as explained above. The `gitSvnAttributesThreshold` specifies the maximum length of a Git attribute value, which may be stored in `.gitattributes`. If the length of the attribute value is larger, it will be stored in the `.gitsvnattributes` directory structure, as explained in Section 9.7.

> **Warning!** These options are set once before the first fetch and shouldn't be changed afterwards.

In *nomal mode*, all these options are set to `true` by default except when SmartGit/Hg detects that the `.gitattributes` file has become too large (in that case *processEols* and *processOtherProperties* is set to `false`).

In *compatibility mode*, all these options are set to `false`.

9.11.3 Tracking Configuration

SmartGit/Hg's SVN support has a tracking configuration similar to Git's. If some local branch (say, `refs/heads/branch`) tracks some remote branch (`refs/remotes/svn/branch`), then:

- it is possible to push the local branch, and doing so will result in the corresponding SVN branch modification according to the repository layout. If the local branch is not configured as tracking branch of some remote branch, it won't be pushed;
- while fetching, SmartGit/Hg proposes to rebase the local tracking branch onto the tracked branch after a Pull if the corresponding option is selected.

> **Warning!** If some branch contains a merge-commit that has a merge-parent that doesn't belong to any tracking branch, `svn:mergeinfo` won't be modified when pushing such a branch.

SmartGit/Hg uses the `branch` sections of the `.git/svn/.svngit/svngitkit.config` file for tracking configuration.

The section looks like this:

```
[branch "master"]
tracks = refs/remotes/svn/trunk
remote = svn
```

The name of the section is the local branch name.

- **tracks**: specifies the remote tracked branch
- **remote**: specifies the remote section name with the SVN URL and SVN repository layout

9.12 Known Limitations

The following is a list of notable limitations of SmartGit/Hg's SVN integration:

- Empty directories can't be managed by Git and won't be available
- File locks are not supported
- Sparse check-outs are not supported
- Explicit copy and move operations are not possible; Git recognizes them automatically
- The svk:merge property is not supported

Chapter 10

Tips and Tricks

10.1 Completion

In input fields for files and directories when typing the file path, you will get completion hints after a (back) slash. On Unix-like systems you even can use `~/` to complete content in the home directory.

For the input field to add or edit the path of a Git remote (**Remote**|**Add** and **Remote**|-**Edit URL**) you can use `../` to complete relative paths in case you've cloned a repository parallel to another.

To complete file names in the **Commit Message** input field of the **Commit** dialog, use *<Ctrl>*+*<Space>*-keystroke.

10.2 Text Editors

To select words or larger parts of text, you can press *<Ctrl>*+*<W>*-keystroke multiple times until you have expanded the selection as desired. To copy or cut the whole line to the clipboard, set the caret inside the line without selecting anything and press *<Ctrl/Cmd>*+*<C>*-keystroke or *<Ctrl/Cmd>*+*<X>*-keystroke.

10.3 Compare

To quickly scroll to next and previous changes, position the mouse cursor over the area between both sides ("connector") and use the scroll wheel.

10.4 Speed Search

In tables and tree controls you can quickly find entries by starting to type their name. With <*Up*>-keystroke or <*Down*>-keystroke you can select the previous or next matching entry, <*Home*>-keystroke and <*End*>-keystroke go to the first or last matching entry.

10.5 Table Columns

For all important tables, you can configure the displayed columns by right-clicking the table header and selecting the desired columns from the context menu.

