SmartGit/Hg Manual

syntevo GmbH, www.syntevo.com

2013

Contents

Chapter 1

Introduction

SmartGit/Hg is a graphical Git and Mercurial client which also can 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). SmartGit/Hg's target audience are users who need to manage a number of related files in a directory structure, to coordinate access to these files in a multi-user environment, and to track changes to them. Typical areas of application include software, documentation and website projects.

Acknowledgments

We would like to thank all users who have given us feedback on SmartGit/Hg by suggesting features and reporting bugs and thereby helped us to improve it.

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

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 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** preview 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.

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 next step you have to provide the path to the local directory where the clone should be created.

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. The commands Pull, Push and Synchronize will be explained in this section.

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, rebase the local commits onto the fetched commits, or do nothing. In the latter case, you can merge or rebase by hand, as explained in Section 3.4.4 and Section 3.4.5, respectively.

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 3.4.5 for details.

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.
- **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 repository at a time. If multiple repositories have been set up, the Push To dialog will allow you to select the repository to push to. Also, the Push To command is the only variant that allows you to do a *forced* push.

- **Push Commits**: Pushes a range of commits up to a certain commit, 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 **Pushable Commits** pane is focused.
- **Directories pane**: You can invoke **Push** in the **Directories** pane by selecting the project root and choosing **Push** from the root's context menu.
- **Branches pane**: In the context menu of the **Branches** pane, you can invoke **Push** and **Push To** on local branches. Additionally, you can invoke **Push** on tags.
- **Pushable Commits pane**: To push a range of commits up to a certain commit, select that commit in the **Pushable Commits** pane 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** pane via **Synchronize** in the project root's context menu.

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 three facilities to modify this selection: The first two are the Stage and Unstage commands, which allow you to add and remove whole files to and from the Index, respectively. The third one is the Index Editor, which allows you to directly edit the contents of the Index for a certain file, thereby adding or removing individual 'hunks' (i.e. parts of the file) 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.

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** pane 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 a *merging* or *rebasing* state (see Section 3.4.4 and Section 3.4.5), 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. If you have staged (see 3.3.1) changes in the Index, 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.

Note	If you commit one or more individual files which have both staged and unstaged changes, the entire working tree state will be committed.

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.

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 **Pushable Commits** pane 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 **Pushable Commits** pane on the project window and invoke **Join Commits** from the context menu of the commit range.
- **Reorder Commits**: In the **Pushable Commits** pane 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.

3.4 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 of branches) is to double-click on a branch in the **Branches** pane and confirm the various settings on the Switch Branch dialog that comes up. The Branches pane 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 switch, or to switch and let SmartGit/Hg do a fast-forward merge. For further information on merging, see Section 3.4.4.

If you select the option **Throw away local changes**, any local changes you may currently have in your working tree will be discarded during the switch operation.

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.

If you select the option **Throw away local changes** on the Check Out dialog, any local changes you may currently have in your working tree will be discarded during the checkout operation.

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** pane 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** pane, you can add and delete branches and tags as well. In addition to that, the Branches pane 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 [> master]
                                |\
o [> master]                    o \
|                     ==>       |  |
|  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
.                                 .
```

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.

```
                                        o [> master] (changes from a-branch)
                                        |
o [> master]                            o
|                         ==>           |
|   o [a-branch]                        |   o [a-branch]
.   .                                   .   .
```

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 pane**: In the **Branches** pane (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 3.4.6 for further information on how to deal with merge conflicts.

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.

A Git-specific alternative to merging is *rebasing* (see Section 3.4.5), 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 Rebase

The Rebase command allows you to apply 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:

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

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

```
                                o  [> master] A'       o  [> master] A'
                                |                      |
                                o  B'                  o  B'
                                |                      |
                                o  D'                  o  D'
                                |                      |
o  [> master] A         o  A    |                      |
|                       .       |                      |
|   o  [a-branch]       .   o   |  [a-branch]          |   o  [a-branch]
|   |                   .   |   |                      |   |
o   |  B                o   |  /  B                    |  /
|   |                   .   | /                        | /
|   o  C (selected      .   o  C                       o
|   |                   .   |               /\         |
o   |  D                o   |  D              ==         |
|  /          ===>      .  /                hiding       |
| /          rebase     . /              lost commits    |
o                       o                                o
```

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.
- **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 a *rebasing* state, allowing you to either manually resolve the conflicts or to abort the rebase. See Section 3.4.6 for further information.

3.4.6 Resolving Conflicts

When a merge (see 3.4.4) or a rebase (see 3.4.5) 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 (middle editor) that contains the conflicting hunks from both sides, along with conflict markers. You can directly edit the text in the middle editor, and you can move changes from the left and right side into the middle by clicking on the arrow and 'x' buttons between the editors.
- **Discard command**: To abort the merge or rebase, select the project root in the **Directories** pane and invoke **Local|Discard**.

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

3.5 Viewing the Project History

3.5.1 Log

SmartGit/Hg's Log window 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. The Log window can be opened by invoking **Query|Log** from the menu of SmartGit/Hg's project window.

What is shown on the Log depends what was selected when the Log command was invoked:

- To view the history of the entire repository, select the project root in the **Directories** pane before invoking the Log command.
- To view the history of the entire repository and put the initial focus on a certain branch, select the branch in the **Branches** pane before invoking the Log command.
- To view the history of a directory within the repository, select the directory in the **Directories** pane before invoking the Log command.
- To view the history of a single file within the repository, select the file in the **Files** pane before invoking the Log command. If the file is not visible in the **Files** pane, 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.

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** pane in SmartGit/Hg's project window and invoking **Query|Blame** from the program menu.

The Blame window is divided into three parts: Some controls on top, a read-only text view on the right, and an info pane on the left. The latter displays various meta data for each line, e.g. author and commit date. With the controls on top, you can do two things: Specify the commit at which the text view will display the file contents, and set the color scheme used to color the lines in the text view. The following color schemes are available:

- **Commit**: The color chosen for a particular line reflects whether the line is "older" or "newer" than a certain 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 critera for determining the line color: Either the relative position in the relevant commit range (first commit, second commit, etc.) or 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.

3.6 Submodules

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**. Initialization in itself will leave the submodule folder 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** pane and follow the dialog instructions.

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

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** pane 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 pane 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** pane 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** pane, then select the submodule branch in the **Branches** pane, 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** pane.

After switching the submodule to another commit, the submodule will be shown as 'changed' in the **Directories** pane. 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** pane 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** pane. 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.

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

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.
📄	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

Advanced Settings

In addition to the options on 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 `C:\Users\[UserName]\.gitconfig` for global configuration.

5.1 System Properties

SmartGit/Hg can be configured by editing the file `smartgit.properties` in the settings folder. 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 6.1. 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`

5.2 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**: `SmartGit 4.app\Contents\MacOS\SmartGit`
- **Linux**: `bin/smartgit.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.

5.2.1 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.

5.2.2 Option ”–cwd”

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

- If the `open` or `log` 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` or `log` options are specified with relative paths, these relative paths will be resolved against the path given with the `cwd` option.
- If the `open` or `log` 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.

5.2.3 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
```

5.2.4 Option ”–log”

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

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

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

5.2.5 Option ”–blame”

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

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

5.3 Location of the Settings Directory

The settings directory is where SmartGit/Hg will store its settings. See Section 6 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`. Note that 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:

5.3.1 Windows

All Users

In the file `bin/smartgit.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/smartgit.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:

- 2000/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`.

5.3.2 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.

5.4 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/smartgit.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/smartgit.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 `SMARTGIT_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 SMARTGIT_MAX_HEAP_SIZE 512m`.
- **Linux**: Set the environment variable `SMARTGIT_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 SMARTGIT_MAX_HEAP_SIZE=512m`.

Chapter 6

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.

6.1 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 5.3).

6.2 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`.
- `passwords` is an encrypted file and stores the *passwords* used throughout SmartGit/Hg.
- `accelerators.xml` stores the *accelerators* configuration.
- `credentials.xml` stores authentication information (not including the corresponding passwords).

- `hostingProviders.xml` stores information about configured hosting provider accounts (not including the corresponding passwords).
- `projects.xml` stores all configured *projects* including their settings.
- `settings.xml` stores the application-wide settings (e.g. the preferences) of SmartGit/Hg.
- `ui-config.xml` stores UI related, more stable settings, e.g. the toolbar configurations.
- `ui-settings.xml` stores UI related, volatile settings, e.g. window sizes or column widths.

6.3 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 to the user's settings directory:

- `accelerators.xml`
- `credentials.xml`
- `hostingProviders.xml`
- `projects.xml`
- `settings.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 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.

6.4 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 SMARTGIT_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 7

SVN Integration

7.1 Overview

Git allows you to interact not only with other Git repositories, but also with SVN repositories. This means you can use SmartGit/Hg as a simple 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 common SVN client features, you can use all (local) Git features like local commits and branching. SmartGit/Hg performs all SVN operations transparently, so you almost never have to think about which server VCS is hosting your main repository.

7.2 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 manner similar to how *git-svn* does it.

7.3 Ignores (Normal Mode Only)

SmartGit/Hg tries to map `svn:ignore` properties to `.gitignore` files. Unlike the `git svn create-ignore` command SmartGit/Hg puts `.gitignore` files under version control. If the user modifies 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, so a `.gitignore` file may contain a pattern that can't be mapped 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
Suppose 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. 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.

7.4 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 in Git blobs without modification. Now, if the user has 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.

7.5 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 the directories are supported, not externals pointing to individual files.

SVN submodules are defined in the file `.gitsvnextmodules`. It 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 to 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 the user 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 directory has *default* color (yellow), the submodule's current state exactly corresponds to the state defined by `.gitsvnextmodules`, and there are no local commits. Otherwise it has *modified* color (pink).

One can use **Local|Stage** to update the `.gitsvnextmodules` configuration to the current SVN submodule state.

7.6 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.

7.7 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 also be converted to SVN tags via **Remote|Push Advanced**.

Note	Git tags that are actual *objects* (not simple *refs*) are not supported.

7.8 History Processing

7.8.1 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	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.

7.8.2 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.

Warning!	When using SmartGit/Hg to merge revisions which have not been completely translated to Git, the corresponding merges are *lossy*. This means that pushing back such a merge commit to the SVN repository would cause the loss of some information, which is most likely not intended by the merge. This happens for instance if one of the merged revisions contains an `svn:keyword` property change (as `svn:keyword` is not mapped to the Git repository). In this case SmartGit/Hg will issue a warning when attempting to merge and when pushing.

7.8.3 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.

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.

7.8.4 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.

7.8.5 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.

7.9 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 get 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 the user 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*, but in the meantime someone else has 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*.

7.10 Non-ASCII Symbols Support

An SVN repository allows using any UTF-8 symbol within file and directory names. Git treats path names as an array of bytes and cannot display characters that are not supported by the system encoding.

To solve this problem, SmartGit/Hg uses its own system-dependent %-coding method such that ASCII characters and characters that are supported by the system encoding are displayed as-is and unsupported characters are encoded using the following format:

```
%<hex digit><hex digit><hex digit><hex digit>[<hex digit><hex digit>]
```

where *hex digits* is a byte array representation of the path in UTF-8 encoding. The symbol % can be encoded if there is no ambiguity (i.e. if there aren't 4 or 6 hex digits after %).

Example

- SVN path “100%” corresponds to Git path “100%”
- SVN path “100%123” corresponds to Git path “100%123”
- SVN path “100%0025” corresponds to Git path “100%” (as one can see, the mapping is not one-to-one and some exotic Git paths like “%0025” have no SVN representation)
- SVN path “100%1234” corresponds to Git path “100” + symbol whose UTF-8 representation consists of 2 bytes (assuming such a symbol exists): 12 and 34

7.11 SVN Support Configuration

7.11.1 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.con` 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.

7.11.2 Translation Options

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

The section looks like this:

```
[core]
processExternals = true
processIgnores = true
processEols = true
processTags = true
```

These boolean options specify whether SmartGit/Hg should treat `svn:externals`, `svn:eol-style/svn` `svn:ignore` and SVN tags in a special way. The 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* is set to `false`).

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

7.11.3 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

7.12 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