SmartGit Quickstart Guide

syntevo GmbH, www.syntevo.com

2011

Contents

Chapter 1

Introduction

SmartGit is a graphical Git client which runs on all major platforms. Git is a distributed version control system (DVCS). SmartGit'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 projects, documentation projects and website projects.

Acknowledgments

We would like to thank all users who have given us feedback on SmartGit (e.g. via bug reports and feature suggestions) and thereby helped us to improve it.

Chapter 2

Git Concepts

This section helps you to get started with Git and tries to give you an understanding of some 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, everything is a repository, even the local “working copy”, which is always a complete repository, not just a partial one. Git’s *working tree* is the directory where you can edit files. Each working tree has its corresponding 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 working tree, containing all files to edit. The attached repository (or more precisely, the repository’s meta data) 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* (see 3.2.1) them to the remote repository. Other users who have their own clones of the origin repository can *pull* (see 3.2.2) from the remote repository 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*. There can be independent local branches and tracking branches.

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, 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|Working Tree 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 It's All About Commits

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 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` forward to the commit `E` (see Section 2.6.2).

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

This brief example has shown you a complete Push-Pull-Merge cycle for working with remote repositories.

2.5 The Index

The *Index* is an intermediate cache for preparing a commit. With SmartGit, 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 *Revert* 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 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 Merging and Rebasing

2.6.1 "Normal" Merge

Usually, when merging from a different branch and both the current branch and the branch to merge contain changes, there are two different ways to merge: the normal merge and the *squash 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]
.  .                        .  .
```

2.6.2 Fast-forward Merge

A special case is that if the current branch has not diverted from the branch to be merged in and is just a few commits behind, it is sufficient to move the current branch pointer forward. No additional commits need to be created.

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

2.6.3 Squash Merge

The squash merge works like a normal merge, except that it drops the information about which branch 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 your feature branch commits be pushed into the remote repository.

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

Note Merging is a fundamental concept in Git, and SmartGit performs merges automatically in situations where you might not expect them. For example, if you are working on the `master` branch and want to switch to the `release-1` branch, SmartGit merges changes from the tracking branch `origin/release-1`. So keep in mind that a seemingly simple switch to a different branch may result in a merge conflict.

A Git-specific alternative to merging is *rebasing* (see Section 3.4.4), which can be used to keep the history linear.

For example, if a user has done local commits and performs a pull with merge, a merge commit with two parent commits - his last commit and the last commit from the tracking branch - is created. When using rebase instead of merge, Git applies the local commits on top of the commits from the tracking branch, thus avoiding a merge commit. Note that if no conflict occurs.

2.7 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.

2.8 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 directory structures into another directory structure, which 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` files 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.

Submodule repositories are not automatically cloned, but need to be initialized first. The initialization arranges necessary entries in the `.git/config` file, which can be edited later by the user, e.g., to fix SSH login names.

2.9 Git-SVN

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

Chapter 3

Important Commands

This chapter gives you an overview of important SmartGit commands.

3.1 Project-Related

A SmartGit project is a named entity with one working tree assigned to it. For greater user convenience, a couple of (primarily) GUI-related options are stored in the SmartGit project. Depending on the selected directory, when cloning or opening a working tree, SmartGit allows creating a new project, opening an existing one in the selected directory or adding the working tree to the currently open project.

To group projects, use **Project|Project Manager**. To remove a working tree from a SmartGit project, use **Project|Remove Working Tree**.

3.1.1 Open Working Tree

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

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

3.1.2 Cloning a Repository

Use **Project|Clone** to create a clone of another Git 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 a Remote Repository

The following commands can be found in the **Remote** menu:

3.2.1 Push

Use **Remote|Push** or **Remote|Push Advanced** to send local commits to a remote repository.

Remote|Push can only push the current branch or changes in all tracking branches to the origin repository, but works for multiple working trees.

Remote|Push Advanced only works for one working tree. In case multiple remote repositories are assigned to your local repository, you have to select one of them as the target repository to push the local commits to. Select the local branches or tags for which you want to push commits. 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).

3.2.2 Pull

Use **Remote|Pull** to fetch commits from a remote repository and "integrate" (i.e. merge or rebase) them into the local branch.

After the commits have been successfully fetched from the remote repository, they are stored in the remote branches. The changes have to be merged or rebased into the local branches, which can be done either automatically or manually. If the option **Merge fetched remote changes** is selected, a merge is performed automatically after fetching. Similarly, if the option **Rebase local branch onto fetched changes** is selected, a rebase is performed automatically after fetching. In case of conflicts, the working tree remains in *merging* or *rebasing* state, respectively. To change the default behavior, go to **Project|Working Tree Settings**.

Alternatively, you can use the Merge (see 3.4.3) command to manually merge the remote changes from the tracked remote branch into the local branch.

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 3.2.1) and Pull (see 3.2.2) 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.

3.3 Local Operations on the Working Tree

These commands can be found in the **Local** menu.

3.3.1 Stage

Use this command to prepare a commit, either by saving the current file content state in the Index (see 2.5), by scheduling an untracked file for addition to the repository, or by scheduling a missing file for removal from the repository.

To commit staged changes, invoke the Commit (see 3.3.4) command on the working tree root.

3.3.2 Unstage

Use this command to undo a previous Stage (see 3.3.1). The Unstage command will discard the selected changes in the Index, meaning that the index changes will be lost, unless they are identical to the current changes in the working copy.

3.3.3 Ignore

Use this command to mark untracked files as to be ignored. This is useful to distinguish purely local files which should never be stored in the repository from locally created files which show up as *untracked* and should be stored in the repository for the next commit. If the menu option **View|Ignored Files** is selected, ignored files will be shown.

Ignoring a file will write an entry 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 *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) manually.

3.3.4 Commit

Use this command to save local changes in the local repository.

If the working tree is in merging state (see Section 2.6), 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 complete file names or file paths. Use **Select from Log** to pick a commit message or SHA ID from the log.

If **Amend foregoing commit instead of creating a new one** is selected, you can update the commit message and files of the previous commit, e.g. to fix a typo or to add a forgotten file.

If a normal merge (see 2.6.1) has been performed before, you will have the option to create a merge commit or normal commit. See Section 2.6.1 and Section 2.6.3 for details.

3.3.5 Undo Last Commit

Use this command to undo the last commit. The committed file contents will be stored in the Index (see 2.5).

Warning!	Don't undo an already pushed commit unless you know exactly what you are doing! Otherwise you need to force pushing your local changes, which might discard other users' commits in the remote repository.

3.3.6 Revert

Use this command to revert the file contents either back to their Index (see 2.5) or back to their repository state (HEAD). If the working copy is in merging state, use this command on the root of the working copy to get out of the merging state.

3.3.7 Remove

Use this command to remove files from the local repository and optionally 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.

3.3.8 Delete

Use this command to delete local files (or directories) from the working tree.

> **Warning!** Note that the files will *not* be moved to the system's trash bin, so restoring the content afterwards may not be possible!

3.4 Branch Handling

3.4.1 Switch

Use this command to switch your working tree to a different branch.

If you select a remote branch, you can optionally create a new local branch (recommended).

If you switch to a local branch which has a remote tracking branch and the option **Merge changes from tracking branch** is selected, SmartGit will attempt to merge changes from the tracking branch after switching. If the aforementioned option is not selected, you can later use the Merge command (see 3.4.3) to merge changes from the tracking branch.

3.4.2 Checkout

Use this command to switch the working tree to a certain commit. 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 point to, you will have the option to create a corresponding local branch.

3.4.3 Merge

Use this command to merge changes from another branch to the current branch.

Select the option **Branch starting with the selected commit** to prepare a normal merge (see 2.6.1). This will leave your working tree in merging state to let you review or tweak the changes. When performing a commit (see 3.3.4), you will have the option to create a merge commit or a simple commit (Section 2.6.3). Also select the option **If possible, move just the branch pointer forward**, if instead of preparing a merge commit, the current branch pointer should just move forward (Section 2.6.2).

Select the option **Only the selected commits** to cherry-pick one or more commits into the working tree. This will not prepare a merge commit, so there will be no corresponding option on the commit either. If you also select the option **Undo the selected commit**, the selected commit will be *reverted* in the working tree. Use this to undo the changes introduced by the selected commit.

3.4.4 Rebase

Use this command to apply commits from one branch to another.

Select the option **Current branch onto its tracked branch** (e.g. after having fetched remote changes) to apply your recent local commits onto the tracked branch. This command is a shortcut for the option **HEAD commits to selected commit** and selecting the commit with the tracked branch. Use this option to keep the history of your repository linear. This option is only enabled if your local branch is lagging behind its tracked remote branch.

Select the option **Selected commit(s) to HEAD**, click **Next** and select a commit (or a commit range, i.e. start and end) to apply the commits to the current branch.

```
                               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
```

Select the option **HEAD commits to selected commit**, click **Next** and select the commit to which you want to apply the last commits of the current 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
|   |                 rebase  .   |               hiding       |
o   |  D                      o   |  D         lost commits  |
|  /                          .  /                             |
| /                           . /                              |
o                             o                                o
```

3.4.5 Add Branch

Use this command to create a branch at the current commit.

3.4.6 Add Tag

Use this command to create a tag at the current commit.

3.4.7 Branch Manager

Use this dialog to get an overview of all branches or to delete some of the branches. The dialog also allows you to set up tracking by selecting a local non-tracking branch and a remote branch.

Chapter 4

Directory tree and file table

Directory tree and 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

System Properties/VM Options

This section gives an overview of some low-level options that have to be specified as Java VM options rather than via the SmartGit preferences, usually because they have to be known early at startup time, or because they cannot be changed during runtime.

Options suppied to the VM are either *standard* or *non-standard* options, like `-Xmx` to set the maximum memory limit, or *system properties*, typically prefixed by `-D`. This chapter mainly describes SmartGit-specific system properties.

5.1 General Properties

The following general-purpose properties are supported by SmartGit:

smartgit.home

This propery specifies the directory into which SmartGit will put its configuration files; refer to Section 6 for details. The value of `smartgit.home` may also contain other default Java system properties, like `user.home`. Another accepted value is the special `smartgit.installation` property, which refers to the SmartGit installation directory.

Example
To store all settings in the subdirectory `.settings` of SmartGit's installation directory, you can set `smartgit.home=${smartgit.installation}\.settings`.

5.2 User Interface Properties

smartgit.splashScreen.show

This property specifies whether to show the splash screen on startup or not. It defaults to `true`.

> **Example**
> Set `smartgit.splashScreen.show=false` to disable the splash screen.

5.3 User Interface Properties

smartgit.github.server

This property specifies the server to be used by the GitHub integration. It defaults to `github.com`.

> **Example**
> Set `smartgit.github.server=localhost` to use the local host instead.

smartgit.refresh-on-frame-activation

This property triggers a full Refreshing on frame activation for operating system where file system monitoring is not supported. It is enabled by default.

> **Example**
> Set `smartgit.refresh-on-frame-activation=false` to disable Refreshing on frame activation.

5.4 Specifying VM Options and System Properties

Depending on your operating system, VM options and/or system properties may be specified in different ways.

smartgit.properties file

The `smartgit.properties` file is present on all operating systems. It is located in SmartGit's *settings directory*; see Section 5.1 for details. All *system properties* can be specified in this file.

Note	*System properties* are VM options that would be specified by a `-D` prefix if they were provided directly to the `java` process. All options listed in this chapter are *system properties* and hence can be specified in the `smartgit.properties` file.

Every option is specified on a new line, with its name followed by a "=" and the corresponding value.

Example
Add the line `smartgit.splashScreen.show=false` to disable the splash screen.

Microsoft Windows

VM options are specified in `bin/smartgit.vmoptions` within the installation directory of SmartGit. You can also specify system properties by adding a new line with the property name, prefixed by `-D`, and appending `=` and the corresponding property value.

Example Add the line

```
-Dsmartgit.splashScreen.show=false
```

to disable the splash screen.

Apple Mac OS X

System properties are specified in the `Info.plist` file. Right-click the `SmartGit.app` in the Finder and select **Show Package Contents**, then double-click the `Contents` directory, where you will find the `Info.plist` file. Open it in a text editor of your choice and specify the system properties as key-string pairs in the `dict`-tag after the `key` with the `Properties` content.

Example Use the following key-string pairs

```
<key>Properties</key>
<dict>
    ...
    <key>smartgit.splashScreen.show</key>
    <string>false</string>
</dict>
```

to disable the splash screen.

Specify a VM option by placing them in the `string`-tag to the `VMOptions` array.

Unix

The file `bin/smartgit.sh` within the installation directory of SmartGit is one place where system properties can be specified. You can set a property by adding the property name, prefixed by `-D` and appending `=` and the corresponding property value to the `_VM_PROPERTIES` environment variable. Multiple properties are separated by a whitespace. Be sure to use quotes when specifying multiple properties.

Example Add

```
_VM_PROPERTIES="$_VM_PROPERTIES -Dsmartgit.splashScreen.show=false"
```

before the `$_JAVA_EXEC` to disable the splash screen.

Chapter 6

Installation and Files

SmartGit stores its configuration files per-user. The root directory of SmartGit's configuration area contains subdirectories for every major SmartGit version, so you can use multiple versions concurrently. The location of the configuration root directory depends on the operating system.

6.1 Location of SmartGit's Settings Directory

- **Windows:**: The configuration files are located below `%APPDATA%\syntevo\SmartGit`.
- **Mac OS:**: The configuration files are located below `~/Library/Preferences/SmartGit`.
- **Unix/Other:**: The configuration files are located below `~/.smartgit`.

Tip	You can change the directory where the configuration files are stored by changing the system property smartgit.home (see 5.1).

6.2 Notable Configuration Files

- `accelerators.xml` stores the *accelerators* configuration.
- `credentials.xml` stores authentication information, except the corresponding passwords.
- `license` stores your SmartGit's *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.
- `projects.xml` stores all configured *projects* including their settings.

- `settings.xml` stores the application-wide Preferences of SmartGit.
- `uiSettings.xml` stores the *context menu* configuration.

6.3 Company-wide Installation

For company-wide installations, the administrator can install SmartGit on a read-only location or network share. To make deployment and initial configuration for the users easier, certain configuration 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's installation directory (parallel to the `lib` and `bin` directories).

When a user starts SmartGit for the first time, the following files will be copied from the `default` directory to the user's private configuration area:

- `accelerators.xml`
- `credentials.xml`
- `projects.xml`
- `settings.xml`
- `uiSettings.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 will prefill the **License** field in the **Set Up** wizard when a user starts Smartgit for the first time. When upgrading SmartGit, 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 `smartgit.exe` launcher will search for an appropriate JRE in the following order (from top to bottom):

- Environment variable SMARTGIT_JAVA_HOME

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

Chapter 7

Internals of the SVN integration

7.1 Compatibility and incompatibility modes

SmartGit'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 was freshly cloned with SmartGit (not *git-svn*). All the features are supported in this mode. Created repositories are not compatible with *git-svn*.
- **git-svn compatibility mode**: (short just "compatibility mode") SmartGit can work with repositories that are created by the *git-svn* command. In this mode advanced features like *EOLs-*, *ignores-* or *externals-translation* are turned off. The SVN history is processed similarly as *git-svn* does.

7.2 Ignores (normal mode only)

SmartGit tries to map `svn:ignore` properties onto `.gitignore` file(s). Unlike `git svn create-ignore` command SmartGit puts `.gitignore` file under version control. If the user modifies `.gitignore` and pushes, the corresponding `svn:ignore` property is changed.

`.gitignore` syntax is significantly richer compared to `svn:ignore`, so `.gitignore` can contain a pattern that can't be mapped to `svn:ignore`. In this case it is not translated.

Adding/removing a recursive pattern in `.gitignore` corresponds to setting/unsetting that pattern on every existing directory in the SVN repository. When such an SVN revision is fetched (back) to the Git repository it doesn't result into a recursive pattern anymore. Instead it is translated to a set of non-recursive patterns: one pattern per 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 `svn:ignore`-property to `somefile` for all directories: *A*, *B*, *C*. After fetching such a revision we have the following `.gitignore` contents (lines order is not important):

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

Git doesn't support patterns that contain space. SmartGit replaces all spaces in the `svn:ignore` value with `[!!-~]` while .gitignore construction. The opposite it true: all newly added patterns with `[!!-~]` are converted to `svn:ignore` with spaces at these places.

7.3 EOLs (normal mode only)

Different EOLs on different systems can cause some troubles when using *git-svn* on Windows. For instance, if the SVN repository contains a file with `svn:eol-style` set to `CRLF`, its content is stored with CRLF line endings as well. *git-svn* directly puts the file contents into Git blobs without any modifications. If the user has `core.autocrlf` Git option set to `false` this can make it 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 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 *core.autocrlf*-option, so with SmartGit it doesn't matter what the *core.autocrlf* value is.

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

By default, a newly added text file (that is detected by Git as 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 `svneol` git attribute. The syntax is `svneol=<svn:eol-style value>#<svn:mime-type value>` where for example:

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

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

7.4 Externals (normal mode only)

SmartGit 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 `.gitsvnextmodules`. The file 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 from 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 the option or using *HEAD* means the latest available revision.
- **fetch, branches, tags**: 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: a path from the URL of *url* option. It must not contradict the SVN repository layout. The *empty* branch (if `fetch=:refs/remotes/git-` 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`). For the most of the practical cases submodule points to a directory. Some svn:externals can point to a file, then the option has 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 case the external points to a third party library which 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, parent repository and submodule repository, at the same time.

SmartGit shows the repository status in the **Directories**-window. 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.

7.5 Symlinks and executable files

Symlinks-processing and *executable*-bit processing it identical as with *git-svn*. SVN uses `svn:special`-property to mark a file as *symlink*. Then its content should be like the following:

```
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.

7.6 Tags

Unlike *git-svn*, SmartGit creates Git tags for SVN tags. If an SVN tag was created by a simple directory copying, SmartGit creates a tag that points to the copy-source; otherwise SmartGit creates a tag that points to the corresponding commit of `refs/remote-tags/svn/<tagname>`. Git tags can also be converted to SVN tags by **Remote|Push Advanced**.

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

7.7 History processing

7.7.1 Branch replacements

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

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

Tip It is easy to create a branch replacement commit from SmartGit:

- Use **Local|Reset** and reset to some other commit
- Invoke **Remote|Push**: SmartGit will propose to replace the current branch.

7.7.2 Merges

Translating merges from SVN to Git

Completely merged SVN branches correspond to merged Git branches. So, for SVN revisions that change `svn:mergeinfo` in a way that some branch becomes completely merged, SmartGit 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 to merge revisions which have not been completely translated to Git, corresponding merges are *lossy*. This means that pushing back such a merge commit to the SVN repository would lose certain kind of information what 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 will issue a warning when attempting to merge and when pushing.

7.7.3 Cherry-picks

SmartGit supports translation of two kinds of cherry-pick merges between SVN and Git which are:

- either done using SmartGit;
- or done using another Git client, without `--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.

7.7.4 Branch creation

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

Note	SmartGit 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.

7.7.5 Anonymous branches

Anonymous branches can be met very often in a Git repository, usually when 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, 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.

7.8 The Pushing process

Pushing of 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 are dependent on it. For example, if one has a local commit and a Git tag set on it, after pushing the Git tag will be moved to the commit which 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. So it is *NOT RECOMMENDED* to make any changes in the working tree during the pushing process. Otherwise these changes can be lost.

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 and these commits contain conflicting changes with the remaining local commits. In this case, SmartGit leaves the working tree clean and asks the user to solve the problem. The easiest way to solve the problem is to press Pull with **Rebase** option turned on, then the rebase process will be started.

> **Example**
> The last repository revision is *r10*. There are 2 local commits *A* and *B* which are going to be pushed. First, *A* is sent, resulting in revision *r12*, as in the meanwhile 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 fetched back commits *r11* and *r12* won't work, because changes of *B* are conflicting now and can't be applied onto *r12*.

7.9 Non-ASCII symbols support

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

To solve this problem SmartGit 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 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're not 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 which UTF-8 representation consists of 2 bytes (if such symbol exists): 12 and 34.

7.10 SVN support configuration

7.10.1 SVN URL and SVN layout specification

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

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

The section has contents like the following:

```
[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 to connect to the SVN repository;
- **rewriteRoot**: specifies 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 *url* value. The option is useful for continuing working with the repository if the original SVN URL has been changed (in this case *rewriteRoot* should be changed to old SVN URL value);
- **fetch, branches, additional-branches, tags**: specify pairs (SVN path, Git reference) of for all interesting paths of SVN repository. All path beyond these won't be considered by SmartGit. 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 and allows arbitrary number of `;`-separated pairs. Option *fetch* for *compatibility mode* defines the branch to be checked out and configured as tracked after fetch. SmartGit doesn't support `git-svn` patterns in the config and allows only usage of asterisk (`*`). The number of asterisks in the SVN path and Git reference pattern should be equal. No patterns except maybe *fetch* pattern should intersect.

7.10.2 Translation options

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

The section looks like the following:

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

These boolean options define if SmartGit should process `svn:externals`, `svn:eol-style/svn:mime-ty`
`svn:ignore` and SVN tags in a special way. The options are set once before the first fetch and shouldn't be changed.

In *nomal mode* all these options are set to `true` by default except the case if SmartGit detects that `.gitattributes` file becomes too large (in this case *processEols* is set to `false`).

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

7.10.3 Tracking configuration

SmartGit's SVN support has tracking configuration that is similar to Git tracking configuration. If the some local branch (say, `refs/heads/branch`) tracks some remote branch (`refs/remotes/svn/branch`) this means that:

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

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

The section looks like the following:

```
[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.

7.11 Known limitations

There are following notable limitations when working with SmartGit:

- Custom user properties are not translated to Git and hence can't be manipulated
- svn:keywords property is not supported

- 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
- Merging of files/directories which contain custom properties or svn:keywords will not retain these properties on merge (SmartGit will tell you when pushing the commits back to the server)
- svk:merge is not supported