# SmartGit

[SmartGit 1](#_Toc452735705)

[手册 2](#_Toc452735706)

[Introduction介绍 2](#_Toc452735707)

[Git入门 2](#_Toc452735708)

[Repository, Working Tree, Commit代码库、工作区、提交 2](#_Toc452735709)

[Typical Project Life Cycle典型项目生命周期 2](#_Toc452735710)

[Branches分支 2](#_Toc452735711)

[Working with Branches用分支工作 2](#_Toc452735712)

[Branches are just pointers to commits分支只是提交的指针 3](#_Toc452735713)

[Commits提交 3](#_Toc452735714)

[Commit Graph提交路径图 3](#_Toc452735715)

[Putting It All Together合并 3](#_Toc452735716)

[The Index 5](#_Toc452735717)

[Changes view 5](#_Toc452735718)

[Index Editor 5](#_Toc452735719)

[Working Tree States 5](#_Toc452735720)

[Important Git Commands 5](#_Toc452735721)

[Branch-related 5](#_Toc452735722)

[Checkout tasks 6](#_Toc452735723)

['Normal' Merge 6](#_Toc452735724)

[Fast-forward Merge 6](#_Toc452735725)

[Squash Merge 7](#_Toc452735726)

[Merge versus Rebase 7](#_Toc452735727)

[Resolving Conflicts 7](#_Toc452735728)

[Resolving Conflicts 10](#_Toc452735729)

[Adding, Renaming and Deleting Branches and Tags 11](#_Toc452735730)

[Project Window 11](#_Toc452735731)

[Log Window 11](#_Toc452735732)

[Repository-Related 12](#_Toc452735733)

[Opening a Repository 12](#_Toc452735734)

[Cloning a Repository 12](#_Toc452735735)

[Settings 12](#_Toc452735736)

[Local Operations on the Working Tree 12](#_Toc452735737)

[Stage, Unstage, and the Index Editor 12](#_Toc452735738)

[Ignore 13](#_Toc452735739)

[Assume Unchanged 13](#_Toc452735740)

[Skipped 13](#_Toc452735741)

[Commit 13](#_Toc452735742)

[Altering Local Commits 14](#_Toc452735743)

[Discard 14](#_Toc452735744)

[Remove 14](#_Toc452735745)

[Moving/Renaming Files 15](#_Toc452735746)

[Delete 15](#_Toc452735747)

[Stashes 15](#_Toc452735748)

[Cleanup 15](#_Toc452735749)

[Synchronizing with Remote Repositories 15](#_Toc452735750)

[Pull 15](#_Toc452735751)

[Pulling tags 16](#_Toc452735752)

[Push 16](#_Toc452735753)

[Synchronize 17](#_Toc452735754)

[Push, then Pull 17](#_Toc452735755)

[Pull, then Push 17](#_Toc452735756)

[Submodules 17](#_Toc452735757)

[Cloning Repositories with Submodules 17](#_Toc452735758)

[Adding, Removing and Synchronizing Submodules 18](#_Toc452735759)

[Updating Submodules 18](#_Toc452735760)

[Pulling on the Submodule 18](#_Toc452735761)

[Pulling on the Outer Repository 18](#_Toc452735762)

[Working within Submodules 18](#_Toc452735763)

[Managing Remotes 19](#_Toc452735764)

[Repositories, Directories and Files 19](#_Toc452735765)

[Repository Management 19](#_Toc452735766)

[File Filtering 19](#_Toc452735767)

[Primary Directory States 19](#_Toc452735768)

[Submodule States 20](#_Toc452735769)

[Additional Directory States 20](#_Toc452735770)

[File States 20](#_Toc452735771)

[Fixing 'Modified (File Mode)' on Windows 21](#_Toc452735772)

[Log 21](#_Toc452735773)

[Log Commands 22](#_Toc452735774)

[Compare Commits 22](#_Toc452735775)

[Recyclable Commits 22](#_Toc452735776)

[Bugtraq (links to issue trackers) 22](#_Toc452735777)

[Blame 23](#_Toc452735778)

[Document view 23](#_Toc452735779)

[History view 23](#_Toc452735780)

[Git-Flow 24](#_Toc452735781)

[Overview 24](#_Toc452735782)

[Develop Branch 24](#_Toc452735783)

[Master Branch 24](#_Toc452735784)

[Feature Branches 24](#_Toc452735785)

[Release Branches 25](#_Toc452735786)

[Hotfix Branches 25](#_Toc452735787)

[Support Branches 25](#_Toc452735788)

[Git-Flow Commands 26](#_Toc452735789)

[Configure 26](#_Toc452735790)

[Start Feature 26](#_Toc452735791)

[Finish Feature 26](#_Toc452735792)

[Integrate Develop 26](#_Toc452735793)

[Start Hotfix 26](#_Toc452735794)

[Finish Hotfix 26](#_Toc452735795)

[Start Release 27](#_Toc452735796)

[Finish Release 27](#_Toc452735797)

[Start Support Branch 27](#_Toc452735798)

[Migrating from the 'master-release-branch' workflow 27](#_Toc452735799)

[Migration 27](#_Toc452735800)

[Usage 27](#_Toc452735801)

[Outgoing View 28](#_Toc452735802)

[Interactive Rebase 28](#_Toc452735803)

[Preferences 29](#_Toc452735804)

[Commands 29](#_Toc452735805)

[Push 29](#_Toc452735806)

[Log 29](#_Toc452735807)

[Background Commands 29](#_Toc452735808)

[Local and Remote Changes 29](#_Toc452735809)

[Tools 30](#_Toc452735810)

[Open File 30](#_Toc452735811)

[Open in Terminal 30](#_Toc452735812)

[Format Patch 30](#_Toc452735813)

[Apply Patch 30](#_Toc452735814)

[Fast-Forward Merge 30](#_Toc452735815)

[Diff Tools 30](#_Toc452735816)

[Integrations 31](#_Toc452735817)

[GitHub integration 31](#_Toc452735818)

[Setup 31](#_Toc452735819)

[Clone 32](#_Toc452735820)

[Main Window 32](#_Toc452735821)

[Log 32](#_Toc452735822)

[Pull Requests 32](#_Toc452735823)

[Comments 32](#_Toc452735824)

[Possible Problems & Solutions 33](#_Toc452735825)

[BitBucket Server (Atlassian Stash) integration 33](#_Toc452735826)

[BitBucket integration 33](#_Toc452735827)

[JIRA 33](#_Toc452735828)

[Prerequisites 33](#_Toc452735829)

[Commit Message Selection 34](#_Toc452735830)

[Resolving on Push 34](#_Toc452735831)

[Support for 'commit.template' 34](#_Toc452735832)

[Advanced Settings 34](#_Toc452735833)

[System Properties 35](#_Toc452735834)

[Interaction with Git 35](#_Toc452735835)

[smartgit.executable.home 35](#_Toc452735836)

[smartgit.executable.language 35](#_Toc452735837)

[smartgit.core.jgit.streamFileThreshold 35](#_Toc452735838)

[smartgit.core.merge.shortCommitMessage 36](#_Toc452735839)

[smartgit.core.push.recurseSubmodulesCheck 36](#_Toc452735840)

[smartgit.core.push.onlyToFirstMatchingTarget 36](#_Toc452735841)

[smartgit.core.shortShaLength 36](#_Toc452735842)

[smartgit.projectState.enabled 36](#_Toc452735843)

[smartgit.refresh.scanIntoSubmodules 36](#_Toc452735844)

[smartgit.refresh.checkAssumeUnchangedFiles 36](#_Toc452735845)

[smartgit.refresh.inspectEol 36](#_Toc452735846)

[smartgit.storeProjectPathRelativeToApplicationRoot 36](#_Toc452735847)

[smartgit.core.jgit.similarityFileSizeLimit 36](#_Toc452735848)

[smartgit.submoduleUpdate.useGit 37](#_Toc452735849)

[smartgit.core.clone.noHardLinks 37](#_Toc452735850)

[smartgit.ssh.defaultUser 37](#_Toc452735851)

[smartgit.branch.otherRefs 37](#_Toc452735852)

[smartgit.branch.tagExcludeRegEx 37](#_Toc452735853)

[smartgit.branch.remoteBranchExcludeRegEx 37](#_Toc452735854)

[smartgit.revert.commitMessageTemplate 37](#_Toc452735855)

[smartgit.gitflow.tagLastReleaseCommitInsteadOfMaster, smartgit.gitflow.tagLastHotfixCommitInsteadOfMaster 38](#_Toc452735856)

[SVN integration 38](#_Toc452735857)

[smartgit.clone.svnAllowed 38](#_Toc452735858)

[smartgit.defaultConnectionLogging 38](#_Toc452735859)

[smartgit.svn.gitAttributesSizeThreshold 38](#_Toc452735860)

[smartgit.submoduleRecurseInUnchangedSvn 38](#_Toc452735861)

[smartgit.svn.glueFeature 38](#_Toc452735862)

[smartgit.svn.scanSubmodulesForNonSvnParents 38](#_Toc452735863)

[smartgit.svn.defaultCommitMessage 38](#_Toc452735864)

[Processes 39](#_Toc452735865)

[smartgit.filemonitor.enabled 39](#_Toc452735866)

[smartgit.filemonitor.excludeIgnoredDirectories 39](#_Toc452735867)

[smartgit.filemonitor.watchNonFixedDrives 39](#_Toc452735868)

[smartgit.filemonitor.watchUncPaths 39](#_Toc452735869)

[smartgit.backgroundTasks.idleDelaySeconds 39](#_Toc452735870)

[smartgit.process.timeout 39](#_Toc452735871)

[smartgit.disableCheckForNewVersion 39](#_Toc452735872)

[smartgit.updater.directory 39](#_Toc452735873)

[Networking 40](#_Toc452735874)

[java.net.preferIPv4Stack 40](#_Toc452735875)

[http.nonProxyHosts 40](#_Toc452735876)

[smartgit.proxy.configureForGit 40](#_Toc452735877)

[smartgit.proxy.timeout.connect and smartgit.proxy.timeout.read 40](#_Toc452735878)

[User Interface 40](#_Toc452735879)

[smartgit.accelerators.tabCtrlI 40](#_Toc452735880)

[smartgit.browserCommand 40](#_Toc452735881)

[smartgit.gui.dpiFactor 40](#_Toc452735882)

[smartgit.repository.showPath 40](#_Toc452735883)

[smartgit.repository.favoriteSuffix 40](#_Toc452735884)

[smartgit.branch.infoPrefix 41](#_Toc452735885)

[smartgit.branches.priority. <category>=<value> 41](#_Toc452735886)

[smartgit.branches.head=<value> 41](#_Toc452735887)

[smartgit.branches.head.branch=<value> 41](#_Toc452735888)

[smartgit.setupFinishedUrl 41](#_Toc452735889)

[smartgit.ui.splashscreen 41](#_Toc452735890)

[smartgit.ui.verboseDate 41](#_Toc452735891)

[smartgit.ui.verboseDate.showOnlyTimeForToday 41](#_Toc452735892)

[smartgit.ui.systemtray.linux.enabled 41](#_Toc452735893)

[smartgit.ui.toolbar.textRightBesideImage 41](#_Toc452735894)

[smartgit.pull.update-submodule-default 41](#_Toc452735895)

[smartgit.pull.initialize-submodule-default 42](#_Toc452735896)

[smartgit.pull.defaultToFetchOnly 42](#_Toc452735897)

[smartgit.pull.defaultToRebase and smartgit.pull.defaultToRememberDefault 42](#_Toc452735898)

[smartgit.annotate.maxToolTipWidth 42](#_Toc452735899)

[smartgit.annotate.maxComboboxMessageLength 42](#_Toc452735900)

[smartgit.tree.forceFullSelection 42](#_Toc452735901)

[smartgit.ui.useColorsIfSelected 42](#_Toc452735902)

[smartgit.pushableCommits.limit 42](#_Toc452735903)

[smartgit.pushableCommits.includeMirrorRemotes 42](#_Toc452735904)

[smartgit.pushableCommits.preserveAuthorDate 42](#_Toc452735905)

[smartgit.reveal.commandLine 43](#_Toc452735906)

[smartgit.repository.makeNameUnique 43](#_Toc452735907)

[smartgit.forceActive, smartgit.forceActive.dialog and smartgit.forceActive.window 43](#_Toc452735908)

[smartgit.flatLook 43](#_Toc452735909)

[Text editors / File Comparison 43](#_Toc452735910)

[smartgit.compare.skipBinaryComparison 43](#_Toc452735911)

[smartgit.compare.maximumFileSize 43](#_Toc452735912)

[smartgit.compare.text.maxLineLength 43](#_Toc452735913)

[smartgit.compare.innerLine.maximumLineLength and smartgit.compare.innerLine.maximumLongLineCount 43](#_Toc452735914)

[smartgit.textEditors.syntaxHighlighting 43](#_Toc452735915)

[Log 44](#_Toc452735916)

[smartgit.log.detectRenames 44](#_Toc452735917)

[smartgit.log.showAllRefsForOnBranches 44](#_Toc452735918)

[smartgit.log.showStashes 44](#_Toc452735919)

[smartgit.log.renameDetection.enabled 44](#_Toc452735920)

[smartgit.log.branchPriorities 44](#_Toc452735921)

[smartgit.log.commitMessage.shortLimit 44](#_Toc452735922)

[smartgit.commit.message.useSystemEncoding 44](#_Toc452735923)

[smartgit.log.onlyHead 44](#_Toc452735924)

[System property "smartgit.log.branchPriorities" to specify order of Git-Flow refs in Log 45](#_Toc452735925)

[smartgit.cherryPick.shaPrefix 45](#_Toc452735926)

[smartgit.avatar.size 45](#_Toc452735927)

[smartgit.avatar.serverUrl=http://www.gravatar.com/avatar/{0}.jpg?s={1} 45](#_Toc452735928)

[smartgit.log.graph.[type].[component] 45](#_Toc452735929)

[GitHub integration 45](#_Toc452735930)

[smartgit.github.pullRequestPageLimit 45](#_Toc452735931)

[smartgit.github.commentPageLimit 45](#_Toc452735932)

[smartgit.github.perPageLimit 45](#_Toc452735933)

[BitBucket integration 45](#_Toc452735934)

[smartgit.bitbucket.useHttps 45](#_Toc452735935)

[JIRA integration 46](#_Toc452735936)

[smartgit.jira.resolveOnPush 46](#_Toc452735937)

[smartgit.jira.closedStates 46](#_Toc452735938)

[Debug Properties 46](#_Toc452735939)

[log4j.[category] 46](#_Toc452735940)

[VM options 46](#_Toc452735941)

[Location of the Settings Directory 47](#_Toc452735942)

[Location of the Updates Directory 47](#_Toc452735943)

[Used Java Runtime Environment 47](#_Toc452735944)

[Windows 47](#_Toc452735945)

[Linux 47](#_Toc452735946)

[Memory Limit 48](#_Toc452735947)

[Extended PATH 48](#_Toc452735948)

[Distributed Reviews (add-on) 48](#_Toc452735949)

[Initialization 48](#_Toc452735950)

[Workflows 49](#_Toc452735951)

[Using a Pull Request to request a review or integration of a feature 49](#_Toc452735952)

[Reviewing and integrating a Pull Request 49](#_Toc452735953)

[Commenting changes 49](#_Toc452735954)

[Inspecting a Pull Request history 50](#_Toc452735955)

[Closed Pull Requests 50](#_Toc452735956)

[Users 50](#_Toc452735957)

[Database structure 50](#_Toc452735958)

[Conflict Resolution 51](#_Toc452735959)

[Local Review Data 51](#_Toc452735960)

[Historic Commit Refs 51](#_Toc452735961)

[Disposing your local Review Database 52](#_Toc452735962)

[Server-side component 52](#_Toc452735963)

[Installation 52](#_Toc452735964)

[Notifications 53](#_Toc452735965)

[Events 53](#_Toc452735966)

[Style files, entities and their attributes 53](#_Toc452735967)

[Roles 53](#_Toc452735968)

[User mapping 54](#_Toc452735969)

[SVN Integration 54](#_Toc452735970)

[Overview 54](#_Toc452735971)

[Compatibility and Incompatibility Modes 54](#_Toc452735972)

[Ignores (Normal Mode Only) 54](#_Toc452735973)

[EOLs (Normal Mode Only) 55](#_Toc452735974)

[Externals (Normal Mode Only) 55](#_Toc452735975)

[Symlinks and Executable Files 56](#_Toc452735976)

[Other SVN properties (Normal Mode Only) 57](#_Toc452735977)

[Adding a property 57](#_Toc452735978)

[Modifying a property 57](#_Toc452735979)

[Removing a property 58](#_Toc452735980)

[Tags 58](#_Toc452735981)

[History Processing 58](#_Toc452735982)

[Branch Replacements 58](#_Toc452735983)

[Merges 58](#_Toc452735984)

[Cherry-picks 58](#_Toc452735985)

[Branch Creation 59](#_Toc452735986)

[Anonymous Branches 59](#_Toc452735987)

[The Push Process 59](#_Toc452735988)

[SVN Support Configuration 60](#_Toc452735989)

[SVN URL and SVN Layout Specification 60](#_Toc452735990)

[Translation Options 60](#_Toc452735991)

[Tracking Configuration 61](#_Toc452735992)

[NTLM authentication 61](#_Toc452735993)

[Known Limitations 61](#_Toc452735994)

[File externals not supported 62](#_Toc452735995)

[Further Limitations 62](#_Toc452735996)

[Bug Reports 62](#_Toc452735997)

[Detailed Bug Reports 62](#_Toc452735998)

[Crash Footprints 62](#_Toc452735999)

[Command-Line Options 63](#_Toc452736000)

[Options "-?" and "--help" 63](#_Toc452736001)

[Option "--cwd" 64](#_Toc452736002)

[Option "--open" 64](#_Toc452736003)

[Option "--log" 64](#_Toc452736004)

[Option "--blame" 64](#_Toc452736005)

[Option "--anchor-commit" 64](#_Toc452736006)

[Installation and Files 65](#_Toc452736007)

[Default Location of SmartGit's Settings Directory 65](#_Toc452736008)

[Notable Files in the Settings Directory 65](#_Toc452736009)

[Resetting certain parts of the configuration to defaults 66](#_Toc452736010)

[Program Updates 66](#_Toc452736011)

[Technical Details 66](#_Toc452736012)

[Company-wide Installation 66](#_Toc452736013)

[JRE Search Order (Windows) 67](#_Toc452736014)

[Tips and Tricks 67](#_Toc452736015)

[Autoscrolling while Drag and Drop 67](#_Toc452736016)

[Completion 67](#_Toc452736017)

[Text Editors 68](#_Toc452736018)

[Compare 68](#_Toc452736019)

[Speed Search 68](#_Toc452736020)

[Table Columns 68](#_Toc452736021)

[How Tos 68](#_Toc452736022)

[Configuration 68](#_Toc452736023)

[Debugging 71](#_Toc452736024)

[Inspecting SmartGit's process using JVisualVM 73](#_Toc452736025)

[Compare with git status 75](#_Toc452736026)

[Procedure 75](#_Toc452736027)

[Performance Tuning 76](#_Toc452736028)

[Refreshing 76](#_Toc452736029)

[Log 76](#_Toc452736030)

[Miscellaneous 76](#_Toc452736031)

[Purchase-Related 77](#_Toc452736032)

[Purchase Overview 77](#_Toc452736033)

[Increase the Number of Licenses 77](#_Toc452736034)

[Decrease the Number of Licenses 77](#_Toc452736035)

[Trouble-shooting 78](#_Toc452736036)

[Background 78](#_Toc452736037)

[Resolution 78](#_Toc452736038)

[Workflows工作流 79](#_Toc452736039)

[Modifying the last commit 83](#_Toc452736040)

[Merging the last two commits 84](#_Toc452736041)

[Splitting the last commit into two 84](#_Toc452736042)

[Modifying older commits 84](#_Toc452736043)

[Merging and splitting older commits 87](#_Toc452736044)

[Merging commits 88](#_Toc452736045)

[Splitting commits 89](#_Toc452736046)

[Reordering commits 89](#_Toc452736047)

[Adding Submodules 89](#_Toc452736048)

[Initializing and Updating Submodules 90](#_Toc452736049)

[Tour入门 91](#_Toc452736050)

[Create Repositories创建代码库 91](#_Toc452736051)

[Repositories代码库 92](#_Toc452736052)

[User Interface用户界面 93](#_Toc452736053)

[Review Local Changes审查本地修改 93](#_Toc452736054)

[Commit提交 94](#_Toc452736055)

[Local Changes本地更改 94](#_Toc452736056)

[History 95](#_Toc452736057)

[Pulling and Pushing拉取和推送 95](#_Toc452736058)

# 手册

# Introduction介绍

SmartGit 是git、 hg和svn的图形化客户端。可以在linux 苹果和windows系统上运行，

Git和hg是分布式版本控制系统

## Git入门

本章节帮你了解Git并掌握git基础

## Repository, Working Tree, Commit代码库、工作区、提交

首先，我们介绍一下Git特有的术语，这些术语会与SVN一类的版本控制系统的含义略有不同。传统的中心化版本控制系统如svn签出的代码叫“工作拷贝”，它是和原始版本或原始版本的一部分完全一致的。然而，在Git中就完全不同了，你在本地电脑中处理的是代码库的本地工作区，本地工作区是代码库的一部分，而中心代码库是没有工作树目录的

**示例**

假设你的工程文件放在D盘my-project目录下，那么这个目录就是代码库，它包含了工作区和代码管理的标识数据

## Typical Project Life Cycle典型项目生命周期

跟所有的版本控制系统一样，一般都有一个包含项目文件的中心代码库。想初始化一个本地代码库时，你只要从远程的中心代码库克隆一个副本，之后本地代码库连接到远程代码库（origin）拷贝代码，跟SVN类似。从远端拷贝完项目文件后，你可以对此项目进行修改并且提交，这些更改只会存储在本地的代码库中，你不需要远端代码库的访问权限就可以提交。在你提交了更改时，你可以进行推送操作，这样其他的协同开发者就能看到你推送的更改了

### [Branches](http://www.syntevo.com/doc/display/SG/Branches?src=contextnavpagetreemode)分支

分支是用来存储代码库中各独立提交序列的，比如说为发布版的软件修复错误同时为下一版本开发功能，Git能分出两种不同分支，本地分支和远端分支，本地代码库中你可以任意建立分支，远端的分支就不一样，远端分支是属于远端代码库的本地分支。从远端代码库中克隆是会把远端代码库中存储的所有本地分支都克隆到本地。你不能直接操作远程分支，而是要创建本地分支才能连接到远端分支，本地分支也叫可追踪分支，它是与远端分支对应的。本地分支也可以不进行追踪。默认的主分支叫做master，svn的叫*trunk。克隆后的主分支就追踪远端分支的* *origin/master*.

## Working with Branches用分支工作

当你把代码推送到远端代码库时，这些更改也会传送到远端的追踪分支上。类似的，当你从远端分支拉取更改时，变更也会存储到本地代码库中。拉取后的变更会合并到本地分支中，代码合并可以通过SmartGit的pull命令或者*Merge* 命令Rebase来实现

**Tip提示**

Pull 、*Merge*、 *Rebase*操作可以在**Pull** 选项卡的**Repository|Settings菜单中配置**

## Branches are just pointers to commits分支只是提交的指针

每个分支本质上都是一个指向提交的名字而已。每个代码库独特的指针是Head，它指向当前工作树目录对应的提交。Head不仅仅指向一个提交，同时也指向本地分支。提交更改后会在本地分支顶部新建一个提交，此时指向本地分支的Head会就把指针移到新的提交上去。

### [Commits](http://www.syntevo.com/doc/display/SG/Commits?src=contextnavpagetreemode)提交

Git的提交（commit）和svn的revision类似，是存储在代码库中的带注释的一组更改。提交命令可以用来把工作区目录中的更改提交到代码库中，然后创建一个提交。

## Commit Graph提交路径图

每个代码库都有一个初始化提交，随后的每个提交都会基于一个或多个父级提交，每次提交都是初始化提交的后代，这样代码库就会形成一个提交路径图（提交节点的无回路有向图）。因此呢，提交不只是一系列的代码更改，基于它在提交路径图上的未知它可以表示出代码库的状态。一般的提交只有一个父级节点，初始化提交没有父级节点，而合并提交会有两个及以上的父级节点。

|  |
| --- |
| o ... a merge commit合并提交  | \  |  o ... a normal commit一般提交  |  |  o  | ... another normal commit一般提交  | /  o  ... yet another normal commit which has been branched提交并分支  |  o ... the initial commit初始化提交 |

每次提交都会有一个SHAid标识（一个特别大的数字用十六进制表示），Git允许通过这个提交的标识来牵出代码，用SmartGit的话这个操作就很容易完全不必手输这个sha字符串。签出操作会设置Head和工作区到提交状态，更改过本地工作区的内容后，提交更改时会生成一个新的提交，而这个提交的父级节点就是之前签出的那个。新建的提交叫做Head，因为没有其他提交继承它

## Putting It All Together合并

下面的例子展示了提交、分支、推送、拉取和分支是如何一起使用的

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 branchorigin/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 [Fast-forward Merge](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-merge.fastForward)).

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

### [The Index](http://www.syntevo.com/doc/display/SG/The+Index?src=contextnavpagetreemode)

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 **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).

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.

## Changes view

The **Changes** view of the SmartGit project window can show the changes between the HEAD and the Index, or between the Index and the working tree state of the selected file. You can switch between both views either by clicking the left HEAD button or the right Working Tree button. The detected and expected line separators are shown in the Changes view title. Individual change hunks or inner-line changes can be staged and unstaged there (if the line separators are notmixed).

## Index Editor

The Index Editor shows a 3-pane-view of HEAD, Index and the Working Tree. The Index and the Working Tree state of a file can be edited freely, e.g. to add further modifications to the Index which are not available in the Working Tree.

### [Working Tree States](http://www.syntevo.com/doc/display/SG/Working+Tree+States?src=contextnavpagetreemode)

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.

## [Important Git Commands](http://www.syntevo.com/doc/display/SG/Important+Git+Commands?src=contextnavpagetreemode)

This chapter gives you an overview of important SmartGit commands.

### [Branch-related](http://www.syntevo.com/doc/display/SG/Branch-related?src=contextnavpagetreemode)

#### [Check Out](http://www.syntevo.com/doc/display/SG/Check+Out?src=contextnavpagetreemode)

There are various ways to check out in SmartGit:

* **Project/Log window** Double-click on a branch in the **Branches** view and confirm the **Check Out** dialog that comes up.
* **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](http://www.syntevo.com/doc/display/SG/Log#Log-log), select the commit to switch to and then select **Check Out** from its context menu.

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

If you check out a local branch that tracks a remote branch, and the latter is ahead of the local branch by a couple of commits, you can decide whether you just want to just check out the latest commit of the local branch, or to check out and let SmartGit do a fast-forward merge to the latest commit of the remote branch. For further information on merging, see [Merge](http://www.syntevo.com/doc/display/SG/Check+Out#CheckOut-merge).

## Checkout tasks

* If you have local changes in your working tree, the Check Out might fail. In this case, SmartGit offers you to stash away the local changes before executing the actual Check Out command and re-apply the changes from the stash after executing the command. For further information on stashes, see [Stashes](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-stashes).
* If a .gitattributes file is present in your repository and its content differs between the checkout source commit (old commit) and the checkout target commit (new commit), SmartGit will invoke a thorough inspection of line endings present in your working tree: due to .gitattributes changes the line endings for certain files may have to be changed.

Usually, line endings correction won't be necessary and this part of the Check Out performs quickly (or you won't even notice it). Certain changes to .gitattributes may affect many (or all files), though. In this case, line endings correction may require a significant amount of time.

#### [Merge](http://www.syntevo.com/doc/display/SG/Merge?src=contextnavpagetreemode)

### '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  .                             . |

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

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

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

If you choose **Create Merge-Commit**, SmartGit 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 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 [Resolving Conflicts](http://www.syntevo.com/doc/display/SG/Merge#Merge-resolve-conflicts) for further information on how to deal with merge conflicts.

### Squash Merge

The squash merge works like a normal merge, except that it discards the information about where the changes came from. Hence it only allows you to create normal commits. The squash merge is useful for merging changes from local (feature) branches where you don't want all of your feature branch commits to be pushed into the remote repository.

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

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

### Merge versus Rebase

A Git-specific alternative to merging is rebasing (see [Rebase](http://www.syntevo.com/doc/display/SG/Merge#Merge-rebase)), 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.

## Resolving Conflicts

When a [merge](http://www.syntevo.com/doc/display/SG/Merge#Merge-merge), a [cherry-pick](http://www.syntevo.com/doc/display/SG/Merge#Merge-cherry-pick) or a [rebase](http://www.syntevo.com/doc/display/SG/Merge#Merge-rebase) fails due to conflicting changes, SmartGit 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. 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's project window, the Resolve dialog will come up, where you can set the file's contents to either of the two conflicting versions, i.e. `Ours' or `Theirs'. Optionally, you may also choose not to stage the resetting of the file contents, meaning that the conflict marker on that file won't be removed.
* **Conflict Solver** Selecting a file containing conflicts and invoking **Query|Conflict Solver** will open the Conflict Solver, a three-way diff between the two conflicting versions (left and right editor) and a third version (center editor) that contains the conflicting hunks from both sides, along with conflict markers. You can directly edit the text in the center editor, and you can move changes from the left and right side into the center by clicking on the arrow and `x' buttons between the editors.
* **Discard command** To abort the merge, cherry-pick or rebase, select the repository in the **Repositories** view and invoke **Local|Discard**.

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

#### [Rebase](http://www.syntevo.com/doc/display/SG/Rebase?src=contextnavpagetreemode)

The Rebase command allows you to apply commits from one branch to another. Rebase can be viewed as more powerful version of [Cherry-Pick](http://www.syntevo.com/doc/display/SG/Rebase#Rebase-cherry-pick), which is optimized to apply multiple commits from one branch to another. In SmartGit, a distinction is made between **Rebase HEAD to** and **Rebase to HEAD**:

**Rebase HEAD to** rebases ("moves") the commits below the HEAD to the selected commit. The HEAD will be moved to the new fork.

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

**Rebase to HEAD** duplicates commits from a separate branch to the HEAD (similar to what [Cherry-Pick](http://www.syntevo.com/doc/display/SG/Rebase#Rebase-cherry-pick) does). The HEAD moves forward on its fork.

|  |
| --- |
| 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   E                        o   E |

To **Rebase Onto** you may use the **Log** window. Consider following example where the quickfix2 branch should not start at the quickfix1 branch, but rather on the master branch:

|  |
| --- |
| q2b (quickfix2)   |  q2a   |  q1b (quickfix1)   |  q1a   |   x (master)   |  ... |

To achieve this, just drag the q2a commit onto the x (master) commit and you will get the desired result:

|  |
| --- |
| q2b (quickfix2)   |  q2a   |   |  q1b (quickfix1)   |   |   |  q1b   | /   x (master)   |  ... |

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

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

Just like a merge, a rebase may fail due to merge conflicts. If that happens, SmartGit will leave the working tree in rebasing state, allowing you to either manually resolve the conflicts or to **Abort** the rebase. See [Resolving Conflicts](http://www.syntevo.com/doc/display/SG/Rebase#Rebase-resolve-conflicts) for further information.

## Resolving Conflicts

Core Git rebase conflicts are different to other kinds of merge conflicts, because left and right files are swapped: when rebasing branch A to B, Git will first checkout B, then applies all commits from A. If a conflict occurs, HEAD still points to B and hence the left file would be the file as it's present in B.

From a user's perspective, the left file should always be his/her own file ("ours"), i.e. the file as it's present in A. For this reason, in case of rebase conflicts, SmartGit will swap left and right files. This gives a more consistent user experience, however may result in following different behavior (compared to normal merge conflicts):

* When staging left lines (**Ours**) in the **Conflict Solver**, these lines will finally show up as staged, because your rebase branch B is actually "theirs"
* When invoking **Resolve** and selecting **Ours**, you will see staged file content, because your rebase branch B is actually "theirs"

#### [Cherry-Pick](http://www.syntevo.com/doc/display/SG/Cherry-Pick?src=contextnavpagetreemode)

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

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

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

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

#### [Revert](http://www.syntevo.com/doc/display/SG/Revert?src=contextnavpagetreemode)

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

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

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

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

#### [Manipulating branches/tags](http://www.syntevo.com/doc/pages/viewpage.action?pageId=2326563&src=contextnavpagetreemode)

## 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](http://www.syntevo.com/doc/display/SG/Log#Log-log) window.

### Project Window

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

### Log Window

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

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

### [Repository-Related](http://www.syntevo.com/doc/display/SG/Repository-Related?src=contextnavpagetreemode)

SmartGit remembers opened repositories and (primarily) GUI-related settings. To open a repository, double-click it. If it already is open in another window when double-clicking, the window with the opened repository will become focused. The repository will open in a new window if the current window is currently executing commands or **Open in New Window** from the repository's context menu has been selected. To open multiple repositories at once, use multiple selection (e.g. Ctrl/Cmd+click) and **Open** from the context menu.

Repositories can be arranged in groups. Right-click the repositories and select the target group or **New Group** from the **Move To** submenu. Alternatively, you can use drag-and-drop - either onto the target group or a repository inside the target group (see [autoscroll](http://www.syntevo.com/doc/display/SG/Tips+and+Tricks#TipsandTricks-autoscroll)).

Repositories can be marked as favorite using the **Mark as Favorite** (**Unmark as Favorite**). Favorite repositories are indicated with an asterisk after the name and are sorted before groups which are sorted before non-favorite repositories. Beside that a couple of background refresh operations are performed on favorite repositories.

## Opening a Repository

Use **Repository|Add or Create** to either open an existing local repository (e.g. initialized or cloned with the Git or Hg 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.

## Cloning a Repository

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

Specify the repository to clone either as a remote URL (e.g. ssh://user@server:port/path), or, if the repository is locally available on your file system, as a file path. In the **Selection** step you can configure whether submodules should be fetched as well: usually you will have this option selected, because submodules are an integral part of the main repository you are cloning. You should deselect this option only, if you do not wish to receive certain submodules. For details, refer to [Submodules](http://www.syntevo.com/doc/display/SG/Submodules#Submodules-submodules). Similarly, you will want to fetch the entire repository usually, including all heads and tags. If you are only interested in a specific head (branch) or tag, you may restrict the clone here. Note that a few Git commands do not work properly for such partial repositories (e.g. Pull with Rebase). In the subsequent steps you have to provide the path to the local directory where the clone should be created and the project to which the repository should be assigned.

## Settings

Use **Repository|Settings** to configure certain repository-specific settings.

On the first page you find options related to the [Pull command](http://www.syntevo.com/doc/display/SG/Synchronizing+with+Remote+Repositories#SynchronizingwithRemoteRepositories-pull).

On the second page you can configure your name and email address that will be used when [committing](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-commit) for this repository. To change the global settings (for all repositories, in your .gitconfig), select **Remember as default**.

On the third page you can configure which text encoding SmartGit should assume when showing text files in the **Changes** view or **Compare** window. A lot of UTF-8 encoded text files, with or without byte order mark (BOM), can be detected automatically.

### [Local Operations on the Working Tree](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree?src=contextnavpagetreemode)

## Stage, Unstage, and the Index Editor

Git's [Index](http://www.syntevo.com/doc/display/SG/The+Index#TheIndex-index) is basically a selection of changes from the working tree to be included in the next commit. SmartGit provides following facilities to modify this selection:

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

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

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

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

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

Lastly, to commit staged changes, select the working tree root in the **Repositories** view and invoke the [Commit](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-commit) command.

## 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** is selected, ignored files will be shown.

**Note**

**Show Ignored** will only display ignored files in versioned directories. Ignored files or sub-directories in ignored directories won't show up, as SmartGit will not even scan into these directories for performance reasons.

When you mark a file in SmartGit 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 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.

## Assume Unchanged

Invoke **Local|Toggle 'Assume Unchanged'** on a selection of modified files to 'ignore' their local modifications. Such files won't be detected as modified afterwards and hence won't be included for the next commit.

To turn a file back into Modified state, use **Toggle 'Assume Unchanged'** on an Assume-Unchanged file again. If the menu option **View|Show Assume-Unchanged Files** is selected, Assume-Unchanged files will be shown.

## Skipped

Invoke **Local|Toggle 'Skip Worktree'** on a selection of files to skip them from the Index. This is similar to [Assume Unchanged](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-assume-unchanged), but in general more persistent in case of commands like **Reset**.

To get a file back into the Index, use **Toggle 'Skip Worktree'** on a Skipped file again. If the menu option **View|Show Skipped Files** is selected, Skipped files will be shown.

## Commit

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

If the working tree is in merging or rebasing state (see [Merge](http://www.syntevo.com/doc/display/SG/Merge) and [Rebase](http://www.syntevo.com/doc/display/SG/Rebase)), you can only commit the whole working tree. Otherwise, you can select the files to commit. Previously tracked, but now missing files will be removed from the repository, and untracked new files will be added. This behavior can be changed in the Preferences, section **Commands**.

If you have [staged changes in the Index](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-stage), you can commit them by selecting at least one file with Index changes or by selecting the working tree root before invoking the Commit command.

While entering the commit message, you can use <Ctrl>+<Space>-keystroke to auto-complete file names or file paths. Use **Select from Log** to pick a commit message or SHA ID from the Log. By default, SmartGit will 'guide' you to write commit messages in some standard format, which will not exceed certain line lengths. You can disable this line length guide in the preferences.

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

If you commit while the working tree is in merging state, you will have the option to create either a merge commit or a normal commit. See [Merge](http://www.syntevo.com/doc/display/SG/Merge) for details.

**Note**

If the commit fails because Git complains "unable to auto-detect email address", you can set your name and email address in the [Repository Settings](http://www.syntevo.com/doc/display/SG/Repository-Related#Repository-Related-settings).

## Altering Local Commits

SmartGit 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](http://www.syntevo.com/doc/display/SG/The+Index#TheIndex-index), so no changes will be lost.
* **Edit Last Commit Message** Invoke **Local|Edit Last Commit Message** from the project window's menu to edit the commit message of the last commit.
* **Edit Commit Message** In the **Outgoing** view on the project window, you can edit the commit message of any of the local commits by selecting the commit and invoking **Edit Commit Message** from the commit's context menu.
* **Squash Commits** To combine a range of local commits into a single commit, select the commit range in the **Outgoing** view on the project window and invoke **Squash Commits** from the context menu of the commit range.
* **Reorder Commits** In the **Outgoing** view you can drag&drop a commit to some other location in the list to effectively change its position.

**Warning**

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

## Discard

Use **Local|Discard** to revert the contents of the selected files either back to their [Index](http://www.syntevo.com/doc/display/SG/The+Index#TheIndex-index) state, or back to their repository state (HEAD). If the working tree is in amerging or rebasing state, use this command on the root of the working tree to get out of the merging or rebasing state.

## 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](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-stage) will have the same effect, but the Remove command also allows you to remove files from the repository while keeping them locally.

## Moving/Renaming Files

In general, Git's move/rename tracking happens always on-the-fly, e.g. when logging or blaming a file. Hence, there is no need for an explicit move operation: just move your files with your favorite tools (IDE, file explorer, from command line).

Still, Git offers a git move command for convenience which performs a normal file system move and then stages the removed and the newly added file to theIndex. For a GUI client like SmartGit, providing such an operation is not necessary. Still, because users are frequently confused about the missing move operation, SmartGit provides **Local|Rename**.

Contrary to git status SmartGit's status display does not denote moved files; on-the-fly moves will only be calculated for history-related functions like**Log** or **Blame**.

## Delete

Use **Local|Delete** to delete local files (or directories) from the working tree. You either may delete the files directly or move them to the trash.

On Linux the trash-put command line tool is used to move files to the trash. You may need to install it on your own.

## Stashes

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

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

**Note**

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

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

## Cleanup

The cleanup command runs some housekeeping tasks. When rebasing commits of a side-branch to newer commits of the main-branch, the rebased commits get obsolete. Same applied for the previous commit when commit-amending. You can make those commits visible in SmartGit's Log by selecting the**Recyclable Commits** option in the Branches view.

### [Synchronizing with Remote Repositories](http://www.syntevo.com/doc/display/SG/Synchronizing+with+Remote+Repositories?src=contextnavpagetreemode)

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

## 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 will do after the commits have been fetched: Merge the local commits with the fetched commits or rebase the local commits onto the fetched commits. In the latter case, you can merge or rebase by hand, as explained in [Merge](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-merge) and [Rebase](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-rebase), respectively. These options are meaningless, if you select to **Fetch Only**.

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

If a merge or rebase is performed after pulling, it may fail due to conflicting changes. In that case SmartGit will leave the repository in a merging or rebasingstate so you can either resolve the conflicts and proceed, or abort the operation. See [Merge](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-merge) and [Rebase](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-rebase) for details.

### Pulling tags

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

**Example**

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

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

## Push

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

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

If you try to push commits from a new local branch, you will be asked whether to set up tracking for the newly created remote branch. In most cases it is recommended to set up tracking, as it will allow you to receive changes from the remote repository and make use of Git's branch synchronization mechanism (see [Branches](http://www.syntevo.com/doc/display/SG/Branches#Branches-branches)).

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

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

## 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](http://www.syntevo.com/doc/display/SG/Synchronizing+with+Remote+Repositories#SynchronizingwithRemoteRepositories-push) and [Pull](http://www.syntevo.com/doc/display/SG/Synchronizing+with+Remote+Repositories#SynchronizingwithRemoteRepositories-pull) to keep your repository synchronized with the remote repository.

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

* from the menu via **Remote|Synchronize**,
* with the Synchronize toolbar button,
* and in the **Repositories** view via **Synchronize** in the repository's context menu.

In the preferences you can configure the order of the push and pull commands.

### Push, then Pull

If 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.

### Pull, then Push

If there are both local and remote commits, the first triggered pull will fetch the remote changes, merge your local changes or rebase your local commits on top of the remote commits and if this was successful, invokes the push. This has the advantage that if there were no conflicts all your local changes are pushed. The disadvantage is that it may push untested changes.

### [Submodules](http://www.syntevo.com/doc/display/SG/Submodules?src=contextnavpagetreemode)

**Note**

This section only applies to submodules of *native* Git repositories, but not of *SVN clones*. For SVN repositories, refer to [Externals (Normal Mode Only)](http://www.syntevo.com/doc/display/SG/SVN+Integration#SVNIntegration-concepts-svn.externals).

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.

## Cloning Repositories with Submodules

If you clone an existing repository containing one or more submodules via **Repository|Clone**, make sure the option **Include Submodules** is selected, so that all submodules are automatically initialized and updated. Without this option, you may initialize the submodules later by hand via**Remote|Submodule|Initialize**. Just initialization itself will leave the submodule directory empty. For a fully functional submodule, you'll also need to do a pull on it, as described in [Updating Submodules](http://www.syntevo.com/doc/display/SG/Submodules#Submodules-submodule-update).

## Adding, Removing and Synchronizing Submodules

**Note**

Submodules are showing up in the **Repositories** as well as the **Files** view. Submodule operations (from the parent repository perspective) will be performed in the **Files** view. 'Normal' Git operations on the submodule repository itself will be performed in the **Repositories** view.

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

To remove a submodule from the working tree, select the submodule in the **Files** view, invoke **Remote|Submodule|Deinit**.

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

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

## 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 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 [Working within Submodules](http://www.syntevo.com/doc/display/SG/Submodules#Submodules-submodule-working) for more information on the latter.

### Pulling on the Submodule

Select the submodule in the **Repositories** view and invoke **Remote|Pull**. On the Pull dialog that shows up, check either the Rebase or the Merge option. Then, after the pull, the submodule will have a different appearance in the **Repositories** view if new commits have been fetched and a rebase or merge has been performed. This different appearance indicates that the submodule has changed and that you need to [commit](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-commit) the change in the outer repository.

### Pulling on the Outer Repository

Open the repository settings via **Repository|Settings**, and on the **Pull** tab, enable **Update registered submodules**, so that SmartGit automatically updates all registered submodules when pulling on the outer repository. Additionally, you may also enable **And initialize new submodules**; with this, SmartGit 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).

## Working within Submodules

You can view the history of a submodule repository by opening its [Log](http://www.syntevo.com/doc/display/SG/Log#Log-log). To do so, select the submodule in the **Repositories** view and invoke **Log** from the submodule's context menu. You can also restrict the Log to a certain branch within the submodule: Select the submodule in the **Repositories** view, then select the submodule branch in the **Branches** view, and then invoke **Log** from the context menu of the branch.

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

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

If you modify and commit files within the submodule (as part of the outer repository, not externally), the submodule will also show up as 'changed'. Then, after committing the changes, you can push them back to the remote submodule repository via **Push** from the context menu of the **Branches** view. Note that you may lose your work in the submodule if you make changes on a *detached HEAD*. To avoid this, check out a submodule branch before making the changes.

### [Managing Remotes](http://www.syntevo.com/doc/display/SG/Managing+Remotes?src=contextnavpagetreemode)

You can **Add**, **Rename** and **Delete** remotes from the **Remote** menu or directly from the **Branches** view.

SVN remotes can't be delete, because they can't be re-added easily. If you still want to get rid of an SVN remote, e.g. because this was a one-time conversion from an SVN repository to Git, you can manually delete the subtree .git/svn.

## [Repositories, Directories and Files](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files?src=contextnavpagetreemode)

The Repositories view shows all repositories known to SmartGit and, for repositories opened in this window, the directory structure (and their states). The Files view displays the status of your working tree (and Index). The primary directory states are listed in [Primary Directory States](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-directory.states), and possible states of submodules in [Submodule States](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-submodule.states). Every primary and submodule state may be combined with additional states, which are listed in [Additional Directory States](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-directory.badges). The possible file states are listed in [File States](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-file.states).

## Repository Management

To add existing or new local repositories to SmartGit, please take a look at the section [Repository-Related](http://www.syntevo.com/doc/display/SG/Repository-Related).

## File Filtering

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

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

## Primary Directory States

|  |  |  |
| --- | --- | --- |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-default.png?version=1&modificationDate=1443735348000&api=v2 | Default | Directory is present in the repository (more precisely: there is at least one versioned file below this directory stored in the repository). |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-unversioned.png?version=1&modificationDate=1443735351000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-ignored.png?version=1&modificationDate=1443735349000&api=v2 | Ignored | Directory is not present in the repository (exists only in the working tree) and is marked as to be ignored. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-missing.png?version=1&modificationDate=1443735349000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-conflict.png?version=1&modificationDate=1443735348000&api=v2 | Conflict | Repository contains conflicting files (only displayed on the root directory). Use **Resolve** to resolve the conflict. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-merge.png?version=1&modificationDate=1443735349000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-root.png?version=1&modificationDate=1443735349000&api=v2 | Root/Submodule | Directory is either the project root or a submodule root, see [Submodule States](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-submodule.states). |

## Submodule States

|  |  |  |
| --- | --- | --- |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-root.png?version=1&modificationDate=1443735349000&api=v2 | Submodule | Unchanged submodule. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-submodule-modified-wt.png?version=1&modificationDate=1443735350000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-submodule-modified-index.png?version=1&modificationDate=1443735350000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-submodule-modified-wt-index.png?version=1&modificationDate=1443735350000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-submodule-unscanned.png?version=1&modificationDate=1443735351000&api=v2 | 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. |

## Additional Directory States

|  |  |  |
| --- | --- | --- |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-badge-local-direct.png?version=1&modificationDate=1443735347000&api=v2 | Direct Local Changes | There are local (or Index) changes within the directory itself. |
| http://www.syntevo.com/doc/download/attachments/1704326/directories-badge-local-indirect.png?version=1&modificationDate=1443735348000&api=v2 | Indirect Local Changes | There are local (or Index) changes in one of the subdirectories of this directory. |

## File States

|  |  |  |
| --- | --- | --- |
| http://www.syntevo.com/doc/download/attachments/1704326/files-unchanged.png?version=1&modificationDate=1443735354000&api=v2 | Unchanged | File is under version control and neither modified in working tree nor in Index. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-unversioned.png?version=1&modificationDate=1443735355000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-ignored.png?version=1&modificationDate=1443735352000&api=v2 | Ignored | File is not under version control (exists only in the working tree) and is marked to be ignored. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-modified.png?version=1&modificationDate=1443735353000&api=v2 | Modified | File is modified in the working tree. Use **Stage** to add the changes to the Index or **Commit** the changes immediately. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-staged.png?version=1&modificationDate=1443735354000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-modified-staged.png?version=1&modificationDate=1443735353000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-modified.png?version=1&modificationDate=1443735353000&api=v2 | Modified (File Mode) | The content of the file is not modified, but the executable bits are set different than in the repository. Refer to [Fixing 'Modified (File Mode)' on Windows](http://www.syntevo.com/doc/display/SG/Repositories%2C+Directories+and+Files#Repositories,DirectoriesandFiles-file.states.how-to-fix-modified-file-mode) on how to fix that state on Windows. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-added.png?version=1&modificationDate=1443735351000&api=v2 | Added | File has been added to Index. Use **Unstage** to remove from the Index. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-removed.png?version=1&modificationDate=1443735354000&api=v2 | Removed | File has been removed from the Index. Use **Unstage** to un-schedule the removal from the Index. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-missing.png?version=1&modificationDate=1443735353000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-modified-added.png?version=1&modificationDate=1443735353000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-intent-to-add.png?version=1&modificationDate=1443735352000&api=v2 | 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. |
| http://www.syntevo.com/doc/download/attachments/1704326/files-conflict.png?version=1&modificationDate=1443735352000&api=v2 | Conflict | A merge-like command resulted in conflicting changes. Use the **Conflict Solver** to fix the conflicts. |

## Fixing 'Modified (File Mode)' on Windows

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

Hence, if this option in not present in .git/config of your repository, invoke git config core.filemode false in the root directory of your repository to fix this problem.

## [Log](http://www.syntevo.com/doc/display/SG/Log?src=contextnavpagetreemode)

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

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

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

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

## Log Commands

In the Log Frame, many commands which are known from the Project Window are available as well:

* most of them are available from the main menu bar
* the context menu on a commit provides certain commands
* certain items in the **Graph** view, like local refs or the HEAD-arrow can be modified using drag-and-drop.

## Compare Commits

You can compare two commits in the **Graph** by selecting both commits (Ctrl-click). The difference from the newer commit compared to the older commit will be displayed in the **Files** view. By selecting a file you can see detailed change in the **Changes** view.

When comparing branches you can also invoke **Reveal Commit** from the context menu of the first branch in the **Branches** view, then invoke **Compare with Selected Commit** on the second branch.

## Recyclable Commits

In the **Branches** view, you can toggle **Recyclable Commits** to get back obsolete heads which are not reachable anymore from any ref. Technically, SmartGit will include all commits which are found in the reflogs (.git/logs-files).

### [Bugtraq (links to issue trackers)](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704589&src=contextnavpagetreemode)

If you have set up a so called Bugtraq Configuration, SmartGit will detect issue IDs in commit messages and display links to the issue tracker in this case. The Bugtraq configuration is stored either in the .gitbugtraq file in your repository root (for all users of the repository) or in your repositories' .git/config (just for you). It consists of a named bugtraq section which basically defines a regular expression to match issue IDs in your commit message and an URL template to open when clicking at such an issue link.

**Example**

An example configuration for the JIRA issue tracker at host 'host' for a project called 'SG' looks like the following.

|  |
| --- |
| [bugtraq "jira"]   url = [https://host/jira/browse/SG-%BUGID%](https://host/jira/browse/SG-%25BUGID%25)   logRegex = SG-(\\d+) |

Another example configuration (e.g. for a trouble ticketing system) where IDs like '#213' should be matched only at the beginning of a commit message. Note that the logregex needs to be put in quotes, because '#' serves as a comment character in Git configuration files.

|  |
| --- |
| [bugtraq "otrs"]   url = "[https://otrs/index.pl?Action=AgentTicketZoom;TicketID=%BUGID%"](https://otrs/index.pl?Action=AgentTicketZoom;TicketID=%25BUGID%25)   logregex = "^#[0-9]{1,5}" |

The logRegex must contain only one matching group '()' matching the issue ID. You can use additional non-matching groups '(?:)' for other parts of your regex (or '(?i)' for case insensitive matching). For more details refer to the complete specification at <https://github.com/mstrap/bugtraq>.

## [Blame](http://www.syntevo.com/doc/display/SG/Blame?src=contextnavpagetreemode)

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

## Document view

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

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

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

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

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

## History view

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

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

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

**Note**

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

## [Git-Flow](http://www.syntevo.com/doc/display/SG/Git-Flow?src=contextnavpagetreemode)

## Overview

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

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

### Develop Branch

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

### Master Branch

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

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

### Feature Branches

For each new (non-trivial) improvement which should be added to the ongoing development line, a separate 'feature' branch is created (named by default, e.g.feature/my-feature). This temporary branch will be used to work independently on this particular improvement ('feature'). If one thinks the feature is done, the commits from the 'feature' branch are integrated (either merged or rebased) into the [develop](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.develop) branch and the feature branch will usually be deleted. This way all feature branches in a repository indicate the features which are currently worked on.

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

### Release Branches

To prepare a (planned) software release, a temporary 'release' branch is created from the [develop](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.develop) branch. The 'release' branch is usually forked when all features for the upcoming release have been implemented and the develop branch is in 'feature-freeze' state. Thus, it makes the release independent of further improvements of the develop branch and hence allows to 'harden' the release by fixing bugs. When the state of this branch is ready for official release, it will be tagged and merged into both the [master](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.master) and the [develop](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.develop) branch, this way creating a new release build to be made available to your customers (e.g. 'version 4'). After successful merging, the release branch usually is deleted.

|  |
| --- |
| o ... [> develop] merged release 4\_0  | \  |  \   o ... [master] ... release 4\_0  |   |/ |  |   o  | ... <tag/release-4\_0\_0] a release-preparing commit (e.g. bug-fix)  |   |  |  o  /   | ... a develop commit for a future release  | /    |  o      | ... another develop commit  |      |  |      o ... release 3\_0\_9  |     /| |

### Hotfix Branches

If after an official release a serious bug is detected, a 'hotfix' branch will be created from the latest release state (the HEAD of the [master](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.master) branch). After fixing the bug(s) in this hotfix branch, the state will be tagged and merged into both the [master](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.master) and the [develop](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.develop) branch, this way creating a new build to be made available to your customers (bugfix release, e.g. 'version 4.0.1'). After successful merging, the hotfix branch usually is deleted.

|  |
| --- |
| o ... [> develop] merged hotfix 4\_0\_1  | \  |  \   o ... [master] ... release 4\_0\_1  |   |/ |  |   o  | ... <tag/release-4\_0\_1] a serious bug-fix  |   |  |  o    \ | ... a develop commit for a future release  |     \|  |      o ... release 4\_0  |     /| |

### Support Branches

Support branches are still in 'experimental' state, according to the Git-Flow documentation. Nevertheless, they are used if you have multiple older releases (e.g. 'version 3.0.\*') which are still supported while the head of the master represents the latest release (e.g. 'version 4.0.\*'). Changes in support branches may be unique to the support branch, because the code in the latest release is not present anymore or the bug/improvement has been implemented there already. If a commit from a support branch should still be integrated into the latest release, open a [hotfix](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.hotfix) branch, [cherry-pick](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-cherry-pick) the commit and finish the hotfix.

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

## Git-Flow Commands

### Configure

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

Here you can change the name of your **Develop Branch** and **Master Branch**, though it's strongly recommended to keep the defaults. In case you have multiple remote repositories configured, you can use **Remotes** to select which of the remote repositories should be used by Git-Flow. In the **Prefixes** section you can specify which prefix should be used for **Feature**-, **Release**-, **Hotfix**- and **Support**-branches. Having a sub-directory per category, is recommended.**Version Tags** specifies the prefix for tags which will be created when finishing a [Release](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.release) or [Hotfix](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.release). Usually, it will be fine to use no prefix, as this will give you nice and simple tag names, like 4.6.1.

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

### Start Feature

Use this command to start the work on a new [feature](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.feature). After providing a name for the feature, the corresponding feature branch will be forked off the developbranch and this new feature branch will be checked out.

**Note**

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

### Finish Feature

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

### Integrate Develop

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

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

### Start Hotfix

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

### Finish Hotfix

Use this command if you have prepared some changes for the new bugfix release version and want to make it publicly available. This will create a tag for the hotfix, merge it to the master and develop branch. The actual commit which will be tagged when finishing a hotfix depends on [system properties](http://www.syntevo.com/doc/display/SG/System+Properties)

.

### Start Release

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

### Finish Release

Use this command if you have prepared changes for the release and want to make it publicly available. This will create a tag for the release, merge it into themaster and develop branch. The actual commit which will be tagged on when finishing a release depends on [system properties](http://www.syntevo.com/doc/display/SG/System+Properties) .

### Start Support Branch

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

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

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

### Migration

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

* Fork develop from master and tell all your team-members to continue their development in develop. Directly committing to master is not allowed anymore.
* When the development of version 5 is in feature-freeze state, start a [Release branch](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.release)called release/5 , continue with work on the next version 6 indevelop, bring release/5 to production quality and finally 'finish' the release.

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

#### Old release branches become 'support' branches

The old branches release-1 ... release-4 are actually [Support branches](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.support)and should be renamed to support/release-1 ... support/release-4, hence.

### Usage

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

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

Once the first problem needs to be fixed for version 5, start a [hotfix branch](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-flow.hotfix)called hotfix/5.0.1 and apply the fix there. The hotfix/5.0.1 branch will remain open until you decide to officially release bug-fix version 5.0.1. Only then, this hotfix will be finished what results in a corresponding merge commit in master. Once a new problem needs to be fixed in version 5 series, create a new hotfix from hotfix/5.0.2 which will automatically be forked off the 5.0.1 merge commit in master. In this way, your master will proceed from version 5 release, to 5.0.1, 5.0.2, ...

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

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

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

#### Maintaining older versions

Hopefully, you won't need to apply many changes to older released versions. If you still need to, apply these changes to the corresponding support/release-branches and decide whether these changes should go into the current release as well: if not, you are all set now. If they should, [cherry-pick](http://www.syntevo.com/doc/display/SG/Branch-related#Branch-related-cherry-pick) the corresponding commits into the latest hotfix/5.0.x branch. There should be one such branch only, anyway. In this way, the changes will make it to master anddevelop later, when the hotfix is 'finished'.

## [Outgoing View](http://www.syntevo.com/doc/display/SG/Outgoing+View?src=contextnavpagetreemode)

This view shows 'outgoing' commits:

* this includes all commits of the current branch which are pushable, i.e. which are not present on any other remote branch for the corresponding remote.
* in case of a Git-Flow feature (or bugfix) branch and if the modification of pushed commits has been allowed in the Preferences, this also includes all commits of the branch, which are not present on develop (or master).

The view will include submodule commits as well.The Path column shows the relative location of the commit's repository (usually . for the currently selected repository).

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

## Interactive Rebase

The Outgoing view supports certain workflows to cleanup the commits.

* To squash adjacent commits, select them and invoke **Squash Commits** and provide the new commit message.
* To reorder commits, just use drag and drop.
* To coalesce two (not necessarily adjacent) commits with the same commit message, drag one of the commits onto the other one.
* To change the commit message, select the commit and invoke **Edit Commit Message**.
* To change the author, select the commit and invoke **Edit Author**.

**Note**

Those operations will not work if merge commits would be affected. If the working copy is not clean, SmartGit asks whether to stash the changes and apply later. The behavior of how commit times will (or will not) be adjusted can be configured by system properties ([smartgit.pushableCommits.preserveAuthorDate](http://www.syntevo.com/doc/display/SG/System+Properties#SystemProperties-properties.smartgit.pushableCommits.preserveAuthorDate)).

**Tip**

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

## [Preferences](http://www.syntevo.com/doc/display/SG/Preferences?src=contextnavpagetreemode)

## Commands

This page shows options which influence the executed Git (and Hg) commands.

### Push

Only select the **Allow modifying pushed commits (e.g. forced-push)** if you are sure you know Git well enough to understand the consequences of forced push.

### Log

When **Contact gravatar.com to show images for the users** is selected, a hash of the email address is generated and then gravatar.com is contacted to request the belonging graphic which the user first has to configure there.

## Background Commands

Options on this page define what operations SmartGit might perform automatically when it is in background.

### Local and Remote Changes

If **Detect Remote Changes** is selected, SmartGit will poll the remote repositories of the favorite Git repositories in regular intervals for changes. To avoid excessive overhead, by default only the lightweight git ls-remote command is invoked, so you only get a *notification* about changes. Note that this command will not notice if, for example, the currently checked out feature branch has been merged and removed.

If **Fetch closed 'favorite' repositories** or **Fetch open repositories when idle** is selected, SmartGit also will perform fetch-operations which actually will *fetch*the changes from the remote repositories, but can be more resource-hungry.

## Tools

On this page you can define external tools which can operate on certain selections. A couple of default tools are already preconfigured:

### Open File

This tool will invoke the system's default open command, e.g. to launch a graphic viewer for .png files.

### Open in Terminal

This tool will open the selected directory in the terminal application.

### Format Patch

This tool will create a patch between either the selected commit/ref and the working tree or the selected two commits/refs.

### Apply Patch

This tool will show a file open dialog and then apply the selected patch.

### Fast-Forward Merge

This tool allows to right-click local, tracked branches in the **Branches** view to perform a fast-forward merge for purely remote changes, especially if you want to avoid having to check out the branch.

## Diff Tools

On this page you can configure different GUI diff tools. The list is searched from top to bottom. It is also possible to configure viewer applications, e.g. for graphic files, which then will be invoked two times for the files to be compared.

## [Integrations](http://www.syntevo.com/doc/display/SG/Integrations?src=contextnavpagetreemode)

SmartGit provides an elaborate integration for following hosting providers:

* [GitHub](http://www.syntevo.com/doc/display/SG065/GitHub+integration)
* [BitBucket Server (Atlassian Stash)](http://www.syntevo.com/doc/display/SG070/BitBucket+Server+%28Atlassian+Stash%29+integration)
* [BitBucket](http://www.syntevo.com/doc/display/SG070/BitBucket+integration)

It provides a basic integration (primarily [Clone](http://www.syntevo.com/doc/display/SG/Repository-Related#Repository-Related-clone)) for Assembla, Beanstalk, Codebase, GitLab and Unfuddle.

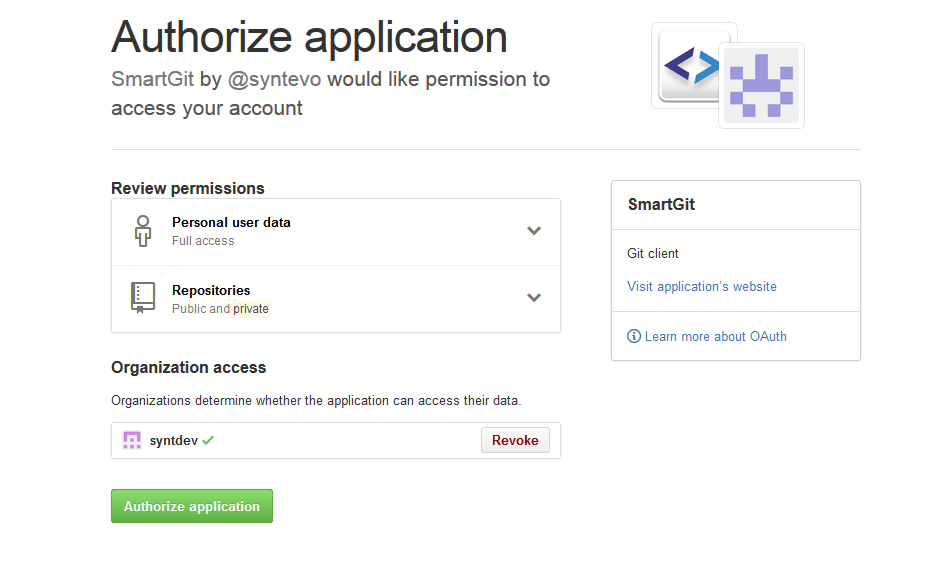
SmartGit also integrates with [JIRA](http://www.syntevo.com/doc/display/SG/JIRA).

### [GitHub integration](http://www.syntevo.com/doc/display/SG/GitHub+integration?src=contextnavpagetreemode)

SmartGit integrates GitHub workflows in various places, provided that the connection to github.com or a custom GitHub Enterprise instance has been configured in the Preferences.

### Setup

To set up the GitHub integration, go to **Preferences**, section **Hosting Providers** and use **Add** there. In the **Add Hosting Provider** dialog, have **GitHub**selected and invoke **Generate API token**. This should open up your default web browser where you will have to confirm by **Authorize Application**. Be sure to also **Grant Access** to all of your organizations, otherwise the corresponding organization repositories won't show up/can't be accessed.



Once you have confirmed this page, you will be redirected to syntevo.com, where the generated access code will be displayed. Copy&paste this code into SmartGit's **Generate API Token** dialog and invoke **Authenticate**. The code will be used to create an application access token which will be used to populate the **Token** field. Finally, confirm the **Add Hosting Provider** dialog using **Add**.

Once you have authorized SmartGit, it will show up in your GitHub **Settings**, section **Applications**. If you need to rerun through the Authorization process outlined above, **Revoke** access there and start over.

## Clone

When [cloning](http://www.syntevo.com/doc/display/SG/Repository-Related#Repository-Related-clone) a repository, you can select your repository from a list, instead of entering the URL. SmartGit will display your own (user) repositories, as well as repositories of your organization (org).

## Main Window

The main window contains a light-weight GitHub integration which gives you an overview of incoming Pull Requests in the **Branches** view.

Pull request operations are only available in the **Log** (see below).

## Log

In the Log window of your repository, you can interact with GitHub in following ways.

### Pull Requests

When initially loading the Log, SmartGit will also refresh information on related Pull Requests from the GitHub server:

* **Incoming** pull requests are those which other users are requesting to pull from their repositories. They are displayed in a separate category called **Pull Requests** in the **Branches** view.
* **Outgoing** pull requests are those which you have sent to other users/repositories, requesting them to pull your changes. They are display directly below the local (or if it does not exist), the remote branch in the **Branches** view.

Incoming pull requests, in first place, are just known on the server. To get the commits, which such a pull request includes, locally, use invoke **Fetch Pull Request** from the context menu of the pull request. This will fetch all commits from the foreign repository to a special branch in your local repository and will create an additional merge node between the base commit from which the pull request has been forked and the latest (foreign) pull request commit. When selecting this merge node in the **Graph**, you can see the entire changes which a multi-commit pull request includes and you can [comment](http://www.syntevo.com/doc/display/SG/GitHub+integration#GitHubintegration-github.comments) on these changes, if necessary. After commenting changes, it's probably a good idea to **Reject** the pull request to signal the initiator of the pull request, that modifications are required before you are willing to pull his changes. If you are fine with a pull request, you may **Merge** it. This will request the GitHub server to merge the pull request and then SmartGit will pull the corresponding branch, so you will have the merged changes locally available.

Outgoing pull requests can be **Fetch**ed as well, however this is usually not necessary, as the pull request belongs to you and it contains your own commits. If you decide that you want to take a pull request back, use **Retract**.

For a pull request which had been fetched once, there was a special ref created which will make it show up in the **Pull Requests** category, even if it is not present on the server anymore. In this case, you may use **Drop Local Data** on such a pull request to get rid of the corresponding ref, the local merge commit, all other commits of the pull request and the entry in **Pull Requests** as well. It's safe to use **Drop Local Data**, as it will only affect the local repository and you can re-fetch a pull request anytime you like using **Fetch** again.

You can invoke **Review|Sync** to manually update the displayed information. Usually you will want to do that, if you know that server-side information has changed since the Log has been opened.

To create a pull request, use **Create Pull Request** from the context menu of the **Branches** view.

### Comments

GitHub allows to comment on a commit itself or individual line changes (diffs). Comments can be applied to a commit or to a Pull Request. Comments will be refreshed together with pull requests after opening a Log or when manually invoking **Review|Sync**.

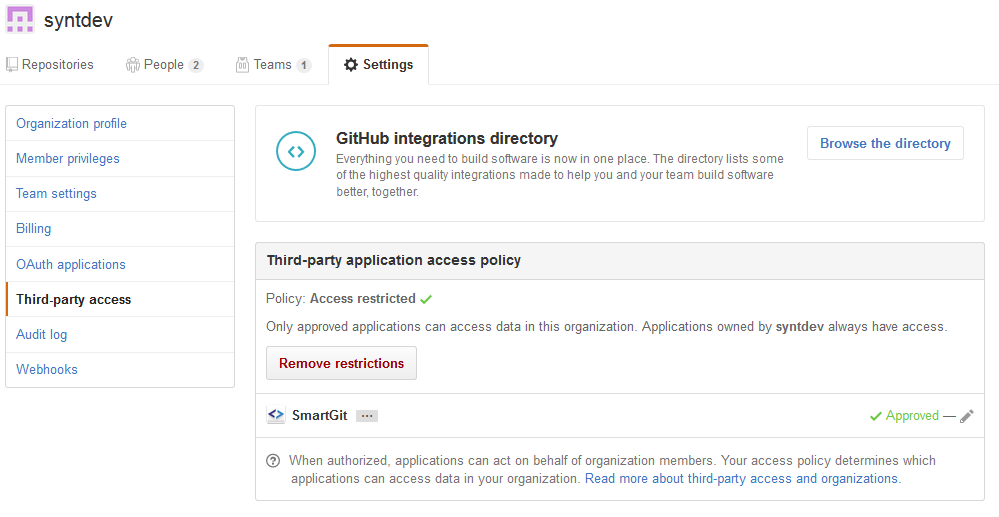
Commit comments will show up in the **Graph** view. Comments on individual lines will show up in the **Changes** view and the affected files will be highlighted in the **Files** and **Graph** view, too. This works the same way for line-comments of Pull Requests, provided that the pull request has been **Fetch**ed and the local pull request merge commit has been selected.

Comments can be created, modified and removed using the corresponding actions from the **Comments** menu or context menu actions in the **Graph** and**Changes** view. If a pull request merge commit is selected, only line-comments of the pull request can be manipulated.

Some behavior of the GitHub integration can be customized by [system properties](http://www.syntevo.com/doc/display/SG/System+Properties#SystemProperties-properties.github).

## Possible Problems & Solutions

If you are authenticating using OAuth and you can't see private repositories of your GitHub organization or pushing to your organization's repositories fails with HTTP error code 403, make sure that your organization allows **Third-party access** and SmartGit is **Approved**. Your organization settings might look like this:



### [BitBucket Server (Atlassian Stash) integration](http://www.syntevo.com/doc/display/SG/BitBucket+Server+%28Atlassian+Stash%29+integration?src=contextnavpagetreemode)

SmartGit integrates BitBucket Server (Atlassian Stash) workflows in various places, very similar to the [GitHub](http://www.syntevo.com/doc/display/SG/GitHub+integration#GitHubintegration-github) integration.

### [BitBucket integration](http://www.syntevo.com/doc/display/SG/BitBucket+integration?src=contextnavpagetreemode)

SmartGit integrates BitBucket workflows in various places, very similar to the [GitHub](http://www.syntevo.com/doc/display/SG/GitHub+integration#GitHubintegration-github) integration. Some behavior can be customized by [system properties](http://www.syntevo.com/doc/display/SG/System+Properties#SystemProperties-properties.bitbucket).

### [JIRA](http://www.syntevo.com/doc/display/SG/JIRA?src=contextnavpagetreemode)

The JIRA integration allows to select a commit message (including JIRA key) directly from (open) JIRA issues and to optionally mark issues as resolved on**Push**.

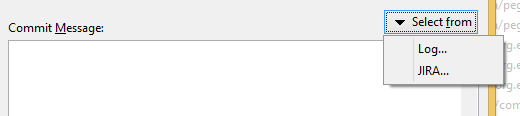
## Prerequisites

The JIRA integration will only be present, if the [Bugtraq configuration](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704589) has been set up properly for your JIRA server and a commercial license is registered in SmartGit.

When connecting to a cloud-based JIRA instance (\*.atlassian.net), you have to login with your **username**, not your email address. You can find your username in your **Profile** (top-right corner).

## Commit Message Selection

The commit message selection is available in the Commit and Edit Last Commit Message commands as well in some interactive rebase commands of the **Outgoing** view.



## Resolving on Push

For all **Push** operations (except of **Push To**), SmartGit checks the pushed commits for affected JIRA issues and offers to mark them as resolved in one or more JIRA versions. A JIRA issue is considered as affected, if

1. It's mentioned in at least one commit message of the local branch commits which are pushed; and
2. It's not mentioned in any commit message of the remote branch commits which are going to be replaced; and
3. when using Git-Flow, you are not pushing into a feature branch or a hotfix branch (SmartGit will ask you whether to resolve such commits when**Finishing** the feature or hotfix, i.e. integrating the commits into develop or master).

This functionality can be disabled using [system properties](http://www.syntevo.com/doc/display/SG/System+Properties#SystemProperties-properties.jira).

## Support for 'commit.template'

The JIRA integration will honor the Git commit.template configuration. Following keywords are substituted by the according JIRA issue attributes:

* %BUGID%
* %BUGSUMMARY%
* %BUGDESCRIPTION%

## [Advanced Settings](http://www.syntevo.com/doc/display/SG/Advanced+Settings?src=contextnavpagetreemode)

In addition to the options in the preferences dialog, SmartGit has some advanced settings that can be set through configuration files or through command-line parameters. Both are covered in the following subsections.

### [System Properties](http://www.syntevo.com/doc/display/SG/System+Properties?src=contextnavpagetreemode)

The vast majority of SmartGit's system properties can be configured by editing the file smartgit.properties in the settings directory.

**Note**

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

First, open the settings directory. Its default location is described in [Default Location of SmartGit's Settings Directory](http://www.syntevo.com/doc/display/SG/Installation+and+Files#InstallationandFiles-settings-dir.default-location). In the settings directory, you will find thesmartgit.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.

The following list shows the available system property keys:

## Interaction with Git

### smartgit.executable.home

By default, Git will search for user configuration files (primarily .gitconfig) in the directory specified by the $HOME environment variable.

Sometimes $HOME passed to SmartGit might not be identical to $HOME available from command line due to specific system configurations. To make SmartGit use the same configuration anyway, specify this property.

Sometimes $HOME might point to a network location which may result in performance problems. In this case, you should create a local HOME directory to use for SmartGit, replicate your .gitconfig into this directory and finally point SmartGit to this directory using smartgit.executable.home.

**Example**

This specifies a different HOME-directory on Linux/Mac:

|  |
| --- |
| smartgit.executable.home=/path/to/home |

This specifies a different HOME-directory on Windows (make sure to use forward slashes):

|  |
| --- |
| smartgit.executable.home=c:/Documents and Settings/user/App Data/syntevo/SmartGit/home |

### smartgit.executable.language

By default, SmartGit will invoke Git with environment variable LANG=C set. Use this system property to change the language passed to Git, for example:smartgit.executable.language=en\_EN.UTF-8. To skip the LANG environment variable, set smartgit.executable.language=.

### smartgit.core.jgit.streamFileThreshold

This option roughly specifies the maximum Git object size (in MB) to be loaded directly into memory. This helps to avoid out-of-memory-errors, but on the other hand certain operations may become extremely slow. This value may be used to tune performance, but it should be chosen with care!

### smartgit.core.merge.shortCommitMessage

When merging, Git's verbose default messages are used. If you prefer shorter, easier readable messages (as default until SmartGit 5), set this value to true.

### smartgit.core.push.recurseSubmodulesCheck

By default, SmartGit invokes "git push" using the "--recurse-submodules=check" option (Git 1.7.7 and above). To disable the check for not yet pushed submodule commits which are referenced in the parent repository, set this option to false.

### smartgit.core.push.onlyToFirstMatchingTarget

By default, SmartGit will push to all branches which have been declared by (multiple) remote.<name>.push declarations (like command line Git does). To make SmartGit stop after pushing to the first branch, set this option to true.

### smartgit.core.shortShaLength

This option specifies the short SHA length, to be used by SmartGit (mainly for displaying). Valid values are in the range between 4 to 40.

### smartgit.projectState.enabled

By default, SmartGit will scan for outgoing commits. This may slow down the program for large repositories. To disable the scan, set the value to false.

### smartgit.refresh.scanIntoSubmodules

Set to false to make the Refresh skip scanning submodule folders.

### smartgit.refresh.checkAssumeUnchangedFiles

By default, SmartGit will check whether 'assume-unchanged' files are really unchanged or not. To disable this check, set the option to false.

### smartgit.refresh.inspectEol

Set to true to have SmartGit distinguish between content and EOL-only changes, in addition to the plain modified state which Git reports (in either case, files which Git considers as modified will be reported). If you do not want Git to care about EOLs at all (that's usually the case if you are on Windows), you might want to use [core.autocrlf](https://www.kernel.org/pub/software/scm/git/docs/git-config.html) or [text and eol attributes](https://www.kernel.org/pub/software/scm/git/docs/gitattributes.html).

### smartgit.storeProjectPathRelativeToApplicationRoot

If set to true, repository paths are stored relative to the SmartGit install location (useful for portable bundles).

### smartgit.core.jgit.similarityFileSizeLimit

The detection of renamed files is by default disabled for very large files. Use this option to adjust when a file is considered 'large'. The value should be given in bytes, like 1000000.

### smartgit.submoduleUpdate.useGit

By default, SmartGit is rather conservative when updating submodules: for instance modified submodules won't be touched, even if the registered commit in the parent repository has changed. Sometimes it may be convenient to still 'update' these submodules, e.g. by merging local commits with the changed, registered commit. This is what 'git submodule update' will do, if key 'submodule.$name.update' is properly configured in .git/config of the parent repository. To invoke 'git submodule update' for modified submodules, set this value to true.

### smartgit.core.clone.noHardLinks

If you are encountering problems when cloning a local repository to/from a mapped network drive, it may be necessary to add '--no-hardlinks' for 'git clone'. To achieve that, set this option to true. Do not forget to reset the option, once you have cloned successfully.

### smartgit.ssh.defaultUser

If an SSH-URL does not contain 'user@', SmartGit will by default use the system user name to authenticate at the SSH server. Use this property to change to a custom default user name.

### smartgit.branch.otherRefs

Use this option to configure additional other refs to be displayed in the Branches view; the default value is notes. To include e.g. archive-refs, set tonotes;archive.

### smartgit.branch.tagExcludeRegEx

To exclude certain tags from being processed by SmartGit, specify a regular expression matching the corresponding tag names (not including refs/tags/). This can be useful if you have a large amount of auto-generated tags which may slow down various SmartGit operations, especially related to the Log.

**Example**

To exclude all tags starting with the prefix autotag-, specify:

|  |
| --- |
| smartgit.branch.tagExcludeRegEx=autotag-.\* |

### smartgit.branch.remoteBranchExcludeRegEx

To exclude certain remote branches from being processed by SmartGit, specify a regular expression matching the corresponding remote branch names (not including refs/remotes/<remote>). For details, see smartgit.branch.tagExcludeRegEx.

### smartgit.revert.commitMessageTemplate

Use this option to customize the commit message used by the **Revert** command: ${message} will be substituted by the commit's message, ${sha} will be substituted by the commit's SHA and \\n will be substituted by a newline character.

**Example**

To get the same commit message as set by command line Git, use:

|  |
| --- |
| smartgit.revert.commitMessageTemplate=revert "${message}"\n\nThis reverts commit ${sha}. |

### smartgit.gitflow.tagLastReleaseCommitInsteadOfMaster, smartgit.gitflow.tagLastHotfixCommitInsteadOfMaster

Set the appropriate option to true to make SmartGit tag the last release/hotfix commit instead of the subsequent merge commit on master when [finishing a Git-Flow release or hotfix](http://www.syntevo.com/doc/display/SG/Git-Flow).

## SVN integration

### smartgit.clone.svnAllowed

Set to false to disable the possibility to clone SVN repositories.

### smartgit.defaultConnectionLogging

Set to true to have SVN connection logging enabled. This will create a connection.log in SmartGit's settings directory which will be helpful for error diagnosis.

### smartgit.svn.gitAttributesSizeThreshold

For certain SVN repositories, the .gitattributes file may become very large, which would slow down various (Smart)Git operations. For that reason, the mapping of svn:eol-style and svn:mime-type will be disabled if the size of the .gitattributes file exceeds the threshold specified by the specified value (in bytes). For more information, see http://www.syntevo.com/smartgit/documentation.html?page=concepts-svn To have the .gitattributes mapping always enabled, you may set the threshold to a large value.

### smartgit.submoduleRecurseInUnchangedSvn

Set to true to enable recursion into unchanged SVN submodules when a submodule update is performed. This may be useful if you always want to fetch the latest revisions from an SVN repository even if the svn:external which is mapped to .gitsvnextmodules does require fetching these revisions.

### smartgit.svn.glueFeature

Set to false to prevent glueing of multiple revisions together when performing an SVN clone. Glueing revisions together improves performance and usually has no negative side-effects. Sometimes, for large repositories, it may result in out-of-memory errors, though.

### smartgit.svn.scanSubmodulesForNonSvnParents

Set to true to scan (refresh) SVN sub-modules for non-SVN parent repositories.

### smartgit.svn.defaultCommitMessage

Use this option to change the default SVN "commit message" when e.g. pushing a tag. There are a couple of variables, which will be replaced by proper strings: {Action}, {Actiond}, {action}, {actiond}, {target}, {source}, {sourceRevision}.

**Example**

Some example definitions:

|  |
| --- |
| {Actiond} {target}{ from {source}{:{sourceRevision}}}  {target} {action} { from {source}{:{sourceRevision}}} |

## Processes

### smartgit.filemonitor.enabled

Set to false to disable the file monitor (which watches for file system changes).

### smartgit.filemonitor.excludeIgnoredDirectories

On Linux, SmartGit will by default not monitor files which are located in ignored directories. Usually this does not cause any problems but improves Refreshing performance and reduces required inotify handles. Sometimes ignored directories are containing tracked files in which case changes to these files might not show up automatically. If this is the case for you, you should disable this optimization by setting this system property to false.

### smartgit.filemonitor.watchNonFixedDrives

On Windows, set this option to false to disable file monitoring for folders that lie on removable drives.

### smartgit.filemonitor.watchUncPaths

On Windows, set this option to true to enable file monitoring for UNC paths. Depending on the network drive type, this may slow down file monitoring and/or may not work reliably.

### smartgit.backgroundTasks.idleDelaySeconds

When SmartGit is idle for a certain amount of time, it will start executing various background tasks. With this option you may set the number of seconds to wait until the 'idle' state is reached.

### smartgit.process.timeout

This setting specifies the number of seconds to wait for a response from a hanging external process before suggesting to kill it. Among other things, this helps to avoid hanging Git processes when trying to access a repository.

### smartgit.disableCheckForNewVersion

Set to true to disable the automatic checking for new program versions. You should only turn this check off for network installations where SmartGit users may not be able to perform the update themselves.

Note that this will also disable notifications of new bugfix releases which you can upgrade to for free and which improve the reliability of SmartGit.

### smartgit.updater.directory

Use this property to customize the [program updater](http://www.syntevo.com/doc/display/SG/Installation+and+Files)'s temporary directory, which is by default located in your home directory/profile. This should only be necessary if updating is not possible due to (file system) restrictions in this default directory, e.g. if execution of files is prevented by the system. On Windows, paths have to be specified using forward-slashes, like c:/temp.

## Networking

### java.net.preferIPv4Stack

By default, SmartGit prefers to connect via IPv4. To connect via IPv6 instead, set this option to false.

### http.nonProxyHosts

Use these properties to specify servers to connect directly to, bypassing the configured proxy, for example: \*.foo.com|localhost. Note, that only internal code of SmartGit is honoring http.nonProxyHosts. This does not include Git itself.

### smartgit.proxy.configureForGit

Set this property to false if you have configured your http(s) proxy in SmartGit for accessing URLs outside your network, but your https-authenticated repositories are located inside your network and should not be tunneled through your proxy.

### smartgit.proxy.timeout.connect and smartgit.proxy.timeout.read

These two settings specify the timeout in milliseconds for connecting to and for reading from a proxy, respectively. Proxies may be used e.g. by "SmartGit Updates".

## User Interface

### smartgit.accelerators.tabCtrlI

By default Ctrl+I enters a tab character. SmartGit contains code to prevent this. To use the default behavior (disable SmartGit Ctrl+I code), select this option totrue.

### smartgit.browserCommand

By default the system browser is used on Windows and Mac. On Linux common browsers are tried to find. If you want to use a certain browser, you may set its path using this option.

### smartgit.gui.dpiFactor

By default SmartGit tries to use the right scaling factor depending on the DPI setting of the operating system. On some systems (e.g. Linux), though, this does not work or needs tweaking. With this option you can set the dpiFactor to either 1 (= 100%) or 2 (200%).

### smartgit.repository.showPath

Set to true to show the path behind closed repositories in the Repositories view.

### smartgit.repository.favoriteSuffix

The Repositories view indicates favorite repositories with a trailing \* behind the repository name. With this option you can define the used indicator, e.g. \u2605 for a filled 5-star, \u2606 for an outlined 5-star or \u2665 for a heart sign.

### smartgit.branch.infoPrefix

The Branches view indicates pull requests with a leading (!) before the branch name. With this option you can define the used indicator, e.g. \u21b3 for a |\_> arrow, \u2709 for an envelope or \u275e for bold double-quotes.

### smartgit.branches.priority. <category>=<value>

The Branches view sorts groups in a certain order. By using these category constants you can set different values to change the order: plugin, flowFeatures, flowHotfixes, flowReleases, flowSupports, localBranches, remotes, tags, stashes, metaRefs, otherRefs, insignificantMerges, lostHeads.

### smartgit.branches.head=<value>

The Branches view uses a unicode character to indicate the checked out branch. If you like to change that, set this property. To add a trailing line space, e.g., use the value "> ".

### smartgit.branches.head.branch=<value>

This is similar as for smartgit.branches.head, but for the current checked out Hg branch while a bookmark is checked out.

### smartgit.setupFinishedUrl

This setting specifies the URL to open after SmartGit has been started for the first time and the setup wizard was completed.

### smartgit.ui.splashscreen

To hide the splash screen, set this option to false.

### smartgit.ui.verboseDate

By default, times from today and yesterday are shown in a short form. To always show the date in the full form, set this option to false.

### smartgit.ui.verboseDate.showOnlyTimeForToday

To show only the time (i.e. without "Today") for times from today, set this option to false.

### smartgit.ui.systemtray.linux.enabled

On Linux, the system tray is disabled by default due to problems with Unity desktops. To enable the system tray, set this option to true.

### smartgit.ui.toolbar.textRightBesideImage

Defines where the text in toolbar buttons is shown relative to the image.

### smartgit.pull.update-submodule-default

By default, for new Git repositories the "Update registered submodules" option is enabled. To disable by default, set this option to false.

### smartgit.pull.initialize-submodule-default

By default, for new Git repositories the "And initialize new submodules" option is enabled. To disable by default, set this option to false.

### smartgit.pull.defaultToFetchOnly

Set this otpion to true to default the Pull dialog to the Fetch option.

### smartgit.pull.defaultToRebase and smartgit.pull.defaultToRememberDefault

Set both options to true to have the Pull dialog always default to "Rebase" and remember this as default. This will be helpful to fix Merge vs. Rebase settings after upgrading to SmartGit 4.

### smartgit.annotate.maxToolTipWidth

In the Annotate window, if the mouse hovers over the yellow line detail column, a tooltip with additional information is shown. With this property you can specify a maximum width in pixels for this tooltip. The specified value must lie in the range [10; 1000]. If the value is not specified or out of range, the default value of 400 pixels is used.

### smartgit.annotate.maxComboboxMessageLength

In the Annotate window, commit messages are displayed in the combobox in the upper part of the window. This property limits the length of the displayed commit messages, i.e. all commit messages that are longer than this property's value will be truncated.

### smartgit.tree.forceFullSelection

Defines whether the tree views throughout the program should use "full selection" style. "Full selection" means that when an item on a tree is selected, the selection box will fill the entire width of the tree, rather than only the item's width.

### smartgit.ui.useColorsIfSelected

Defines whether tree or table views are allowed to use colors other than the system's default background and foreground color to display selected items. Depending on the current platform, the usage of additional colors is disabled by default to improve the readability of selected items.

### smartgit.pushableCommits.limit

The number of entries in the Outgoing view is limited for performance reasons. Use this option to change the limit.

### smartgit.pushableCommits.includeMirrorRemotes

Usually, secondary remotes which have "mirror" set to "true" in the .git/config are not interesting from the perspective of which commits are pushable and hence are ignored. Set this option to true to also have these remotes honored as well.

### smartgit.pushableCommits.preserveAuthorDate

Set this option to false to let SmartGit adjust the author dates when rearranging commits.

### smartgit.reveal.commandLine

"Reveal in File Manager" by default uses hard-coded, platform-specific file managers (Windows: Explorer, OS X: Finder, Linux: Nautilus). Depending on your file manager, you may use the placeholder ${filePath} or ${fileUri}.

### smartgit.repository.makeNameUnique

When creating a new repository, e.g. by dropping a directory onto the Repositories view, SmartGit will use the directory name as name for the repository. This could lead to equally named repositories. When setting this option to true, SmartGit will append number suffixes to ensure that the new repository names don't collide with already existing ones.

### smartgit.forceActive, smartgit.forceActive.dialog and smartgit.forceActive.window

Set one of these options to true to forcibly make new windows/dialogs active.

### smartgit.flatLook

Set this option to true to use a lighter color for the border of view controls.

## Text editors / File Comparison

### smartgit.compare.skipBinaryComparison

To disable the comparison of binary files set this option to true. This may be useful if you usually have large binary files in your repository. When read, these files may slow down certain views, e.g. the "Changes" view.

### smartgit.compare.maximumFileSize

By default, the file comparison is disabled for very large files. Use this setting to adjust at what size (in bytes) a file is considered 'too large'.

### smartgit.compare.text.maxLineLength

By default, the file comparison treats files with very long lines as binary files for performance reasons. Use this option to configure the maximum characters per line (default: 10000) which should be treated as text files.

### smartgit.compare.innerLine.maximumLineLength and smartgit.compare.innerLine.maximumLongLineCount

The file comparison algorithm consists of two phases. In the first phase, changed lines are detected, and in the second phase, the algorithm will search for changes within corresponding lines. The second phase becomes slower if there are too many long lines, so it is skipped when that happens. These settings allow you to adjust the detection of long lines: The second phase is skipped if the two files contain more than 'maximumLongLineCount' lines that are longer than 'maximumLineLength' characters in length.

### smartgit.textEditors.syntaxHighlighting

Set this option to false to switch syntax highlighting off permanently, regardless of Preferences settings.

## Log

### smartgit.log.detectRenames

Set to false to disable the detection of renamed files in the Log. In rare cases (large commits with many additions and removals and large files), rename detection may slow down the Log significantly.

### smartgit.log.showAllRefsForOnBranches

Set to true line if you want to include all refs for the "On Branches" display in the Log Details view.

### smartgit.log.showStashes

By default, the Log shows stashes. To change this, set to false.

### smartgit.log.renameDetection.enabled

By default, detection of renamed files is enabled for Log/Blame. To switch rename detection off, set to false.

### smartgit.log.branchPriorities

Use this option to change the priorities of Git-Flow branch types in the Log. The priorities are affecting the order in which these branches will be displayed in the Log graph. The order is specified using a comma-separated list of following branch categories where every category must occur exactly once in the list: develop, feature, release, master, hotfix, support.

**Example**

To layout in the inverse default order, specify:

|  |
| --- |
| smartgit.log.branchPriorities=support,hotfix,master,release,feature,develop |

### smartgit.log.commitMessage.shortLimit

SmartGit uses a shortened commit message in various places, e.g. in the commit message column of the Log table. Shortening means the full commit message is cut off after the number of characters specified by this setting.

### smartgit.commit.message.useSystemEncoding

By default, commit messages are UTF-8 encoded. To use the default encoding for your platform instead, set to true.

### smartgit.log.onlyHead

Set this system property to true if you are having performance- or memory-related problems when showing a File or Sub-Directory Log for a huge repository: by default, SmartGit will run the log for all branches and tags. This is necessary to e.g. properly assign branches/tags to the File or Sub-Directory commits. Also, by default, you will be able to toggle Log roots in the **Branches** view. When setting this option, SmartGit will only log the current HEAD commit and the**Branches** view will remain empty.

# System property "smartgit.log.branchPriorities" to specify order of Git-Flow refs in Log

### smartgit.cherryPick.shaPrefix

Git's cherry-pick command has an -x option that tells Git to automatically append a "cherry picked from commit ..." message to the created commits, in order to indicate where the latter came from. SmartGit does not do this by default, but you can enable it by setting this propery to cherry picked from commit.

### smartgit.avatar.size

To change the size of the avatar image, change this option.

### smartgit.avatar.serverUrl=http://www.gravatar.com/avatar/{0}.jpg?s={1}

If you want to use a different server than gravatar.com to provide the avatars shown in the log, you may configure it in the next line.

* {0} will be replaced by the MD5-hash of the lowercase email
* {1} will be replaced by the avatar size (see above)

### smartgit.log.graph.[type].[component]

The colors of the graphical [Log](http://www.syntevo.com/doc/display/SG/Log#Log-log)-elements can be configured using this property. Valid types are: tag, branchLocal, branchRemote, branchOther and headArrow. Valid components are red, green and blue. Valid values are 0..255. Example: smartgit.log.graph.branch.red=128

## GitHub integration

### smartgit.github.pullRequestPageLimit

The number of pull requests to fetch from GitHub is limited by "pages" (one page contains a couple of pull requests). The default page limit is 3.

### smartgit.github.commentPageLimit

The number of comments to fetch from GitHub is limited by "pages" (one page contains a couple of comments). The default page limit is 3.

### smartgit.github.perPageLimit

The number of objects per page retrieved from the server (default: 100).

## BitBucket integration

### smartgit.bitbucket.useHttps

Set to true to force SmartGit to connect to repositories using HTTPS, instead of SSH.

## JIRA integration

### smartgit.jira.resolveOnPush

Set to false to disable optional resolving of [JIRA](http://www.syntevo.com/doc/display/SG/JIRA) issues on [Push](http://www.syntevo.com/doc/display/SG/Synchronizing+with+Remote+Repositories).

### smartgit.jira.closedStates

Specifies the name of the JIRA states which are considered as closed: SmartGit will only offer to resolve an issue, if the issue is not yet in a closed state and there is at least one transition available for the issue which puts into a closed state.

**Example**

Default closed states are resolved,closed,done.

## Debug Properties

### log4j.[category]

Use this property to enable debug logging for certain SmartGit modules; [category] has to be replaced by the appropriate module identifier.

**Example**

To enable debug logging for the Refreshing modules, set following properties:

|  |
| --- |
| log4j.smartgit.refresh=DEBUG  log4j.sc.vcs.model.refresh=DEBUG |

### [VM options](http://www.syntevo.com/doc/display/SG/VM+options?src=contextnavpagetreemode)

Certain configuration of SmartGit has to be done by VM options, in files called smartgit.vmoptions. Usually you will want to specify VM options just for your account (current user):

* **Windows:** %APPDATA%\syntevo\SmartGit\smartgit.vmoptions (%APPDATA% is the path defined in the environment variable APPDATA)
* **Linux:** ~/.smartgit/smartgit.vmoptions
* **Mac:** ~/Library/Preferences/SmartGit/smartgit.vmoptions

Alternatively (but **not recommended**), VM options can also be specified system-wide in following files:

* **Windows:** the global file is bin\smartgit.vmoptions in SmartGit's installation directory
* **Linux:** the global file is bin/smartgit.vmoptions in SmartGit's installation directory
* **Mac:** the global file is Contents/MacOS/smartgit.vmoptions in SmartGit's installation directory SmartGit.app

## Location of the Settings Directory

The settings contains SmartGit's settings. See [Installation and Files](http://www.syntevo.com/doc/display/SG/Installation+and+Files#InstallationandFiles-installation) for information about the default location and contents of the settings directory. On Windows and Linux, you can change its location by modifying the VM option -Dsmartgit.settings.

**Note**

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 specialsmartgit.installation property, which refers to the SmartGit installation directory.

**Example**

To tell SmartGit to store its settings in the subdirectory .settings of the SmartGit installation directory, add follow line to smartgit.vmoptions:

-Dsmartgit.settings=${smartgit.installation}\.settings (Windows)

-Dsmartgit.settings=${smartgit.installation}/.settings (Linux)

## Location of the Updates Directory

The Updates directory contains downloaded program updates. See [Installation and Files](http://www.syntevo.com/doc/display/SG/Installation+and+Files#InstallationandFiles-installation) for information about the default location and contents of the Updatesdirectory. On Windows and Linux, you can change its location by modifying the VM option -Dsmartboot.sourceDirectory.

**Example**

To tell SmartGit to store its program updates in the subdirectory .updates of the SmartGit installation directory, add follow line to smartgit.vmoptions:

-Dsmartgit.settings=.updates

## Used Java Runtime Environment

### Windows

Use the Windows environment variable SMARTGIT\_JAVA\_HOME to tell SmartGit which JRE to use. In case of using a 64-Bit JRE, you will have to run SmartGit using bin/smartgit64.exe.

**Example**

To tell SmartGit to use the 64-Bit JRE located at in C:\Program Files\Java\jre8, set the environment variable SMARTGIT\_JAVA\_HOME=C:\Program Files\Java\jre8.

### Linux

On Linux, you can configure the JRE to be used by adding jre=/path/to/jre to smartgit.vmoptions.

**Example**

To tell SmartGit to use the JRE located in /opt/jre/jre32/jre1.7.0, add following line to smartgit.vmoptions:

jre=/opt/jre/jre32/jre1.7.0

## Memory Limit

The memory limit (also known as maximum heap size) specifies how much RAM the SmartGit process is allowed to use. The memory limit can be configured by the VM option -Xmx.

**Example**

To change the maximum memory limit to 1GB, add following line to smartgit.vmoptions:

-Xmx1024m

If the set value is too low, SmartGit may run out of memory during memory-intensive operations.

**Note**

32-Bit Java VMs only allow to configure a maximum memory limit of roughly 1200M. For almost all setups and repository sizes, this should be sufficient. If you nevertheless need to configure a higher limit, you will have to use a 64-bit VM and on Windows invoke smartgit64.exe.

## Extended PATH

On Linux and Mac OS X, you can extend the PATH used by SmartGit (and all processes invoked by SmartGit, especially Git itself) by adding path=/additional/path to smartgit.vmoptions. This path= lines can be used multiple times and will be **appended** to the PATH in the order of occurrence.

**Example**

To add the directory /opt/git-lfs to the PATH, add following line to smartgit.vmoptions:

path=/opt/git-lfs

## [Distributed Reviews (add-on)](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704359&src=contextnavpagetreemode)

The Distributed Reviews add-on allows your team to create a Pull Request from one branch to another and to comment on pull requests and individual commits. Contrary to centralized code reviewing systems like GitHub, no dedicated server is required: all review metadata is stored in the Git repository itself and distributed the same way as your primary content/commits. This gives you all the benefits of Git: you can work offline and you can prepare your pull requests and comments locally and only push them when ready, or discard them if you change your mind, without affecting anyone else.

## Initialization

To initialize the Review system for your local repository invoke **Review|Configure**.

SmartGit will query all currently configured remotes to see whether any of them already contains a Review database. If this is the case, SmartGit will clone this database and connect the local Review system to it. If none of the remotes contains a Review database, a completely new database will be created and the[users database](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704359#DistributedReviews(add-on)-review.users)will be populated from the repository log.

For a shared repository (including all its clones), the initialization of a completely new database should only be done once, e.g. when starting to introduce the Review system for your project. This will usually happen in one of the clones. After the initialization of the local repository, invoke **Review|Configure** and select**Initialize a Remote** to push your Review database back to the selected remote, thus initializing this repository as well. From this point on, **Review|Configure**can be invoked in all other clones of the shared repository to clone and connect to the same Review database.

## Workflows

### Using a Pull Request to request a review or integration of a feature

Pull Requests are used to signal that you consider a feature branch as 'ready' and request someone else to review and integrate this branch into the main development line (usually master or develop in case of [Git-Flow](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-git-flow)).

To create a Pull Request, select your feature branch in the **Branches** view and invoke **Create Pull Request** from the context menu. In the upcoming dialog, select the **target** branch and a **Message** describing the purpose of the Pull Request. It's recommended to specify at least one **Assignee**, who is the right person to review and integrate this Pull Request. The Pull Request will be highlighted to all assignees in the **Branches** view (in the Project as well as the Log window).

Once the Pull Request has been created, it is in pending state and will show up in the **Pull Requests** category of the **Branches** view as local only. It will be published, i.e. sent to the remote repository, for the next time you invoke **Push**. Alternatively, you can manually force publishing the Pull Request using**Review|Sync**.

### Reviewing and integrating a Pull Request

Once a Pull Request is published (i.e. present in the remote repository), it will be fetched by all other users for their next invocation of **Pull** or by doing**Review|Sync** manually. If you are amongst the **Assignees** of a Pull Request, it will be highlighted to you in the **Branches** view.

To review or integrate a Pull Request, open the Log and select and reveal the Pull Request from the **Pull Requests** category. The Pull Request is represented by a merge commit which connects its source (the feature branch) with the merge base between the source and the Pull Request's target (probably master ordevelop). When selecting this Pull Request commit in the **Commits** graph, the **Files** view will show all affected files of the Pull Request and you can drill down to content changes in the **Changes** view.

* If the Pull Request is not yet ready for merge and/or there is something to object to, you may leave a comment (see section [Commenting changes](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704359#DistributedReviews(add-on)-review.commenting)) and**Reject** the Pull Request. At the same time, you will probably want to assign the Pull Request back to the author or someone else who you think can best deal with your comments. Comments may be applied to the Pull Request commit itself (either to the commit of the affected files' changes) or to individual commits of the Pull Request, whatever suits your needs better.
* If the Pull Request is fine, you may **Integrate** the Pull Request from the **Branches** view context menu. The **Integrate** dialog is similar to [Git-Flow's Finish Feature dialog](http://www.syntevo.com/doc/display/SG/Git-Flow#Git-Flow-git-flow): you can select how to integrate the commits and some optional cleanup tasks. Alternately, you may just **Approve** the Pull Request and assign it to someone who should finally **Integrate** the Pull Request.
* If you don't feel to be the right assignee for the Pull Request at all, you may simply **Assign** it to someone else who you think can deal with it better.

In either case it's recommended to add an optional **Comment** which explains the reason for the reassignment. Such Pull Request Comments will show up in the **Branches** view directly below the Pull Request and they will be displayed in the **Details** view as well, when selecting the corresponding Pull Request commit.

When you are done with the integration (or review), be sure to **Sync** your activity with the remote repository.

### Commenting changes

A main feature of the Review system is attaching comments to changes. Usually, you will apply comments while doing a review of a Pull Request, however you can comment on arbitrary changes (commits) as well. Contrary to other reviewing systems, like GitHub, there is no difference between comments on Pull Request commits and comments on any other commits and thus the procedure is identical in both cases:

* To comment on a commit in general, use **Add Review Comment** from the graph context menu.
* To comment on an individual change in the **Changes** view, use **Add Review Comment** from the context menu on the corresponding line.

Comments will show up in the **Commits** graph, the **Files** table and **Changes** view. The **Comments** table will show all comments currently present in the repository and allows to **Jump To** a comment, or **Edit** or **Delete** a comment.

### Inspecting a Pull Request history

Pull Requests include the information on all head commits they had ever been 'on'. Especially, the current head of a Pull Request will change when Rebasing a Pull Request, e.g. to keep the corresponding feature branch up-to-date with master. By having access to all 'historical' heads, you will also have access to all (historical) comments for the Pull Request, thus being able to understand the history and development of a Pull Request.

By default, only the most recent head will show up in the Log **Graph** when toggling the pull request in the **Branches** view. To view details for the pull request, select the pull request commit in the **Graph** and switch to the **Details** view. Here you will also be able to display older heads by clicking the **Version** links.

### Closed Pull Requests

After integrating or closing a pull request, the pull request will still be present in the Review database, but by default be hidden from the **Branches** view. To display closed pull requests, select **Review|Show Closed Pull Requests**.

As long as a pull request is still present in the database, SmartGit maintains refs pointing to all of its heads, thus preventing Git from garbage collecting the corresponding commits, see [Historic Commit Refs](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704359#DistributedReviews(add-on)-review.historic-commit-refs). Only by invoking **Delete** on a closed pull request, it will be removed from the database and correspondingsrefs will be deleted.

## Users

## Database structure

The entire review metadata is stored in a couple of special refs of your Git repository: the latest version of your local metadata is referenced byrefs/meta/smartgit/reviews, the latest (fetched) metadata for each configured remote is referenced by refs/meta/smartgit/remotes/<remote-id>/reviews.<remote-id>s are unique identifiers of remotes and are defined in .git/config. Using IDs instead of the remotes' names avoids problems when renaming a remote.

The metadata refs are pointing to an orphan tree structure which stores the metadata as a set of many small text files. The tree and file structure has been defined in a way to avoid possible content conflicts which would require a manual resolution by the user. (This could happen e.g. if two users have locally modified the same comment). The root of the metadata tree consists of following categories:

* a README file which gives information about the purpose of this directory tree;
* a commit category which contains comments attached to commits. The naming scheme for comments affecting the commit in general (not a particular file) is comments/<commit>/h/<comment-id>/<file-sha>. The scheme for comments affecting a particular file is comments/<commit>/f/<file-path>/<comment-id>/<file-sha>;
* a ref category which contains pull requests. The naming scheme for a pull request is ref/<ref>/h/<pull-request-id>/<file-sha>;
* a note category which contains comments on pull requests. The naming scheme for a pull request comments is note/<pull-request-id>/h/<comment-id>/<file-sha>.
* a global category which contains information on [users](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704359#DistributedReviews(add-on)-review.users).

Thus, every comment or pull request is represented in the file system by its ID. From the user perspective a comment as well as pull request is exactly oneentity. From a more low-level perspective all these entities are treated the same way and one entity actually comprises a set of immutable files which represent different versions of the entity and which have their own hash as filename.

If an entity is modified, this will result in the deletion of the file <old-file-sha> and the creation of a new file <new-file-sha> in the entities' directory. <old-file-sha> as well as <new-file-sha> contain the entire old/new state of the entity and in addition <new-file-sha> will have a replaces: <old-file-sha>-information attached, so SmartGit will be able to interpret the new file as being a replacement of the old file.

### Conflict Resolution

The immutable files design may look overly complex, however it has the crucial advantage of getting rid of conflicts on the core file-system (or syntactical) level, and instead moving these conflicts up to the semantical level.

Syntactical conflicts would occur e.g. when pulling or pushing review metadata and encountering the same file with modified content in both versions. Thus, the user would be forced to solve a conflict in such a core metadata file. The move up to the semantical level happens by simply having two versions of the entity alive, which will be stored in different <file-sha>-files.

Even better, changes which would have resulted in conflicts on the syntactical level, may be mergeable on the semantical level: e.g. if one users assigns a pull request to A and another user assigns the pull request to B, it will be 'merged' the safe way to have the pull request be assigned to A as well as B. This is similar for the content of a comment: if changed concurrently to X as well as Y, it will finally show up as X and Y. SmartGit knows about these 'diverged' versions of one entity and it will highlight this fact to the user, where necessary.

The resolution of these 'conflicts' is done by the user when editing the entity: in our example, it was unclear whether A or B should be the assignee, so either of these users might have a look at the pull request and if he does not feel a suitable assignee, he may remove himself from the list. In case of the diverged comment X and Y, keeping both comment versions might be fine. If not, one of them can simply be removed and the other one fixed to include some additional information from the removed one.

### Local Review Data

From the difference between refs/meta/smartgit/reviews and refs/meta/smartgit/remotes/<remote-id>/reviews, SmartGit will derive which pull requests and comments ('entities') are local.

After pushing regular commits to a certain remote or when invoking **Review|Sync**, SmartGit will include only those entities for which their referred commits (or refs) are already present in the corresponding remote. In this way, you can create local commits together with local review data and push them only all at once. Before pushing local review data, SmartGit will squash all your local review tree commits into a single commit: this especially means that additions followed by deletions of certain entities or some back and forth while editing entities will be skipped. Thus, similar as for 'normal' Git commits, you have freedom in how to assemble your review data; only the final result will matter.

### Historic Commit Refs

SmartGit maintains a list of special refs refs/meta/smartgit/commits/<sha> for all commits which are representing pull request heads (current heads as well as historic heads) and for all commits to which a comment is assigned. This happens to (1) make such commits pushable and pullable and (2) to preserve them from being garbage-collected by Git: while on the client-side, the reflog-files are usually preventing a garbage collection, this is by default not the case on the server-side.

By default, historic commit refs will be removed, once the corresponding entity will be removed from the database. To disabling pruning of such commits, you have to unset the prune-commit-refs option in the review database:

* Create a new clone of your repository, initialize the review database for this clone and use this clone for subsequent operations
* Checkout refs/meta/smartgit/reviews:

|  |
| --- |
| git checkout refs/meta/smartgit/reviews |

Now the working tree will contain the entire content of the review database; you may see top-level directories like ref, commit or global.

* Create top-level directory options (if it does not exist yet)
* From the repository root, invoke following command (on Windows, you have to use git bash):

|  |
| --- |
| git rm options/prune-commit-refs  git commit -m "configuration changed to prune obsolete commit refs"  git update-ref refs/meta/smartgit/reviews `git rev-parse HEAD` |

* Finally, sync this change using SmartGit's **Review|Sync**.

Once this change has been pulled to a different clone for which Distributed Reviews are enabled, obsolete refs won't be removed anymore from now on. To confirm that the option actually works, you may invoke following command before adding and after removing a comment locally:

|  |
| --- |
| git show-ref | grep -c refs/meta/smartgit/commits |

If the displayed number remains increased after having removed the comment, you are done.

## Disposing your local Review Database

Sometimes it may be desirable to reinitialize your local review database from the server's database. For that, it's necessary to first dispose your local database using **Review|Configure** and select **Dispose Database** there.

To dispose the review database from command line (not recommended), invoke:

git update-ref -d refs/meta/smartgit/reviews

### [Server-side component](http://www.syntevo.com/doc/display/SG/Server-side+component?src=contextnavpagetreemode)

The Distributes Reviews server-side component is **optional**. Once installed and configured, it will track changes to the review meta data and send appropriate emails to affected users.

# Installation

Unzip the contained files from the bundle to a new directory on the same server which is also hosting your Git repositories, let's assume this will be/opt/reviewserver. Make sure that post-receive.sh is executable – if not, make it executable using chmod +x post-receive.sh.

If the selected directory is not writable by the system Git account, i.e. the account which will be used to write to the Git repositories, the log file location in log4j.properties must be adjusted.

Adjust the configuration file /opt/reviewserver/config.email for your environment. All **mandatory** parameters have to be supplied.

Finally, following post-receive hook has to be set up for all repositories which should be tracked:

|  |
| --- |
| #!/bin/bash  /opt/reviewserver/post-receive.sh "$(cd "$(dirname "${BASH\_SOURCE[0]}")" && pwd)" |

# Notifications

## Events

Emails will be sent for following events:

* Creation of a pull request (prCreated)
* Assignment of a pull request (prAssigned)
* Editing the message of a pull request (prEdited)
* Updating the base commit of a pull request, e.g. by rebasing onto a more recent commit (prUpdated)
* Closing of a pull request (prClosed)
* Deletion of a pull request (prDeleted)
* Integration of a pull request (prIntegrated)
* Reviewing a pull requests (prReviewed)
* Other (unexpected) modifications of a pull request (prModified)

## Style files, entities and their attributes

For every event, template/email contains a corresponding style file which can be adjusted to your needs. Valid entities and their attributes are which can be used in styles files are:

* The affected repository name repo.name
* Details of the current (new, updated): pr.{name, source, target, integrationCommit, assignees, createdAt, createdBy, updatedAt, updatedBy, text, sourceHead, targetHead, rawAttributes}  
  Note that not all attributes will be available for every type of event. For instance, pr.updatedAt is not available for prCreated, the creation of a pull request.
* Details of the old pull request, similar as above: prOld.{name ... rawAttributes}  
  Note that all of these attributes will only be available for pull requests which have been existing before, what is not the case for the prCreated event.
* For prReviewed, details about the review: review.{reviewer, kind, text, commentCount}
* For prReviewed, detailed information about all affected comments: comments.{path, text}.
* The SHA of the meta-ref commit meta.commit; this will usually just be interesting for debugging purposes.

## Roles

The role mapping config.roles defines which events should be delivered to which kinds of users (roles):

* Creator of the pull request (creator)
* Assignee(s) of the pull request (assignee)
* Last updater of pull request (updater)
* Curator of the affected repository (curator); curators are defined in the user mapping config.users.

## User mapping

The user mapping config.users specifies one or more curators for a specific repository using lines like:

|  |
| --- |
| curator.<repo>: curator1@domain1 [,curator2@domain2 ...] |

where <repo> is the path name of the repository (as present in the file system).

In addition to repository-specific curators, global curators can be defined by a trailing line like:

|  |
| --- |
| curator: curator1@domain1 [,curator2@domain2 ...] |

## [SVN Integration](http://www.syntevo.com/doc/display/SG/SVN+Integration?src=contextnavpagetreemode)

## Overview

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

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

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

## 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 is freshly cloned with SmartGit (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 can work with repositories that were created using the git-svn command. In this mode advanced features like EOLs-, ignores- and externals-translation are turned off. The SVN history is processed in a similar way as by git-svn.

## Ignores (Normal Mode Only)

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

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

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

**Example**

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

|  |
| --- |
| A {    B {}    C {}  } |

And we add .gitignore with only one line:

|  |
| --- |
| somefile |

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

|  |
| --- |
| A/somefile  A/B/somefile  A/C/somefile |

Git doesn't support patterns that contain spaces. Hence, SmartGit 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.

## 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-styleset to CRLF, its content is stored with CRLF line endings. Moreover, git-svn puts the file contents directly into Git blobs without modification. Now, if you have thecore.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 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 it doesn't matter what thecore.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.

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-typeproperty 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.

## 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 directories are supported, not externals pointing to individual files ('file externals').

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

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

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

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

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

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

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

## 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.

## Other SVN properties (Normal Mode Only)

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

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

* A property which belong to the svn:-namespace and does not yet have a special mapping (as explained in the previous sections) will be mapped to a Git attribute starting with the svn\_-prefix. This means, that property svn:keywords will be mapped to attribute svn\_keywords and svn:needs-lock will be mapped to svn\_needs-lock.
* For all other properties (custom properties), an SVN property with name foo will be mapped to a Git attribute with name svnc\_foo.

### Adding a property

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

**Example**

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

|  |
| --- |
| /file.txt svnc\_foo=bar |

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

**Example**

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

|  |
| --- |
| /file.txt svnc\_foo |

and add file .gitattributes/file.txt/svnc\_foo with the desired binary property content.

### Modifying a property

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

**Example**

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

|  |
| --- |
| /file.txt svnc\_foo |

and add file .gitattributes/file.txt/svnc\_foo with the desired property content.

### Removing a property

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

## 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 be sent back to the SVN repository (as SVN tags) by right-clicking the tag in the **Branches** view and invoking **Push**.

**Note**

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

## History Processing

### Branch Replacements

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

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

**Tip**

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

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

### 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 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.

### Cherry-picks

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

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

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

### 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.

### 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.

## The Push Process

Pushing a commit consists of 3 phases:

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

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

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

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

**Example**

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

## SVN Support Configuration

### 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 uses the file.git/svn/.svngit/svngitkit.config for this purpose.

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

The section generally looks like this:

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

* **url** specifies the physical SVN URL with which to connect to the SVN repository.
* **rewriteRoot** specifies the URL to be used in the Git commit messages of fetched commits. If this option is omitted, it is assumed to be the same as the value of the url option. The rewriteRoot option is useful for continuing working with the repository after the original SVN URL has been changed (in this case rewriteRoot should be changed to the old SVN URL value).
* **fetch, branches, additional-branches, tags** specify pairs consisting of an SVN path and a Git reference for various interesting paths in the SVN repository. All paths beyond these won't be considered by SmartGit. 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 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 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.

### Translation Options

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

The section looks like this:

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

These boolean options specify whether SmartGit should enable special handling of svn:externals, svn:eol-style/svn:mime-type, svn:ignore, SVN tags and all other SVN properties, as explained above. The gitSvnAttributesThreshold specifies the maximum length of a Git attribute value, which may be stored in.gitattributes. If the length of the attribute value is larger, it will be stored in the .gitsvnattributes directory structure, as explained in [Other SVN properties (Normal Mode Only)](http://www.syntevo.com/doc/display/SG/SVN+Integration#SVNIntegration-concepts-svn.other-properties).

**Warning**

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

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

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

### Tracking Configuration

SmartGit'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 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 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

### NTLM authentication

If your server supports NTLM authentication and SmartGit fails to connect to your server with authentication-related errors like 401 Authorization Required, you should try to force SmartGit to use Basic authentication first by setting following [system property](http://www.syntevo.com/doc/display/SG/System+Properties):

|  |
| --- |
| svnkit.http.methods=Basic,NTLM |

## Known Limitations

There are following notable limitations of SmartGit's SVN integration,

### File externals not supported

File externals are currently not supported. There are following possible workarounds:

* create an unversioned symlink locally without adding it to the repository
* replace the file with a symlink in the repository (won't work on Windows)
* use directory externals (they are supported by SmartGit)

### Further Limitations

* 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

## [Bug Reports](http://www.syntevo.com/doc/display/SG/Bug+Reports?src=contextnavpagetreemode)

## Detailed Bug Reports

In case of a crash, SmartGit offers to send a detailed bug report directly to our webserver. If you prefer to have a look at the file first, you may locate the given ZIP file in the file system, inspect its contents, remove possibly sensitive data and finally send us the file by email.

## Crash Footprints

SmartGit by default automatically transfers the footprint of a crash to a *crash server* for the purpose of quality assurance. The sent information contains details about the user machine (e.g. version of operating system), SmartGit version/build, the JVM state and where the error occurred. *It contains no potentially sensitive information like user names, email addresses, file contents, file paths or server names.*

A crash footprint file looks like this:

|  |
| --- |
| This crash footprint has been created automatically and will be sent to  <http://www.syntevo.com/smartgit/bugstat> for the purposes of quality assurance  for SmartGit. This footprint does not contain any sensitive information  nor any information from your repositories.    HEADER  time=20141111183916  run=46bb4878207d0b97  debugging=false  version=6.5  build=4113  buildid=  machine=64f9152d  os=Windows 7  osv=6.1  osarch=x86  maxmem=1024  java=1.7.0\_45  uptime=0.01  thread=smartgit.F  exceptionCount=1    STACKTRACE  smartgit.G    at smartgit.F.a(SourceFile:33)    at smartgit.Yd.run(SourceFile:28)  Caused by: java.lang.RuntimeException    at smartgit.F.a(SourceFile:33)    at smartgit.Yd.run(SourceFile:28)  Caused by: java.lang.InternalError    at smartgit.F.a(SourceFile:33)    at smartgit.Yd.run(SourceFile:28) |

You can switch the automatic crash reporting off in the Preferences, section **Bug Reports**.

## [Command-Line Options](http://www.syntevo.com/doc/display/SG/Command-Line+Options?src=contextnavpagetreemode)

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

* **Windows** bin\smartgit.exe or bin\smartgitc.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 <version-number>.app/Contents/MacOS/SmartGit
* **Linux** bin/smartgit.sh

In the following, we'll use smartgitc.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.

## 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 you're using:

**Example**

smartgitc.exe -?

smartgitc.exe --help

**Note**

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

## Option "--cwd"

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

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

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

## Option "--open"

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

**Example**

smartgitc.exe --open C:\path\to\repository

**Example**

smartgitc.exe --cwd C:\path --open to\repository

## Option "--log"

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

**Example**

smartgitc.exe --log C:\path\to\repository\path\to\file

**Example**

smartgitc.exe --cwd C:\path --log to\repository

## Option "--blame"

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

**Example**

smartgitc.exe --blame C:\path\to\repository\path\to\file

When adding a colon with the line number at the file end, it will scroll to the specified line.

**Example**

smartgitc.exe --blame C:\path\to\repository\path\to\file:400

## Option "--anchor-commit"

This option can be optionally specified in addition to "--log" and defines the anchor commit of the Log. The anchor commit will be made visible and preselected in the **Commits** view.

**Example**

smartgitc.exe --log C:\path\to\repository\path\to\file --anchor-commit=10de7ee0313e79c406d729f4c3e11f286df54f05

## [Installation and Files](http://www.syntevo.com/doc/display/SG/Installation+and+Files?src=contextnavpagetreemode)

SmartGit stores its settings files per-user. Each major SmartGit 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.

## Default Location of SmartGit's Settings Directory

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

**Tip**

You can change the directory where the settings files are stored by changing the property [smartgit.settings](http://www.syntevo.com/doc/display/SG/VM+options#VMoptions-settings-dir.change-location). This is used by the portable bundle for Windows.

## Notable Files in the Settings Directory

* license stores your SmartGit license key.
* log.txt contains debug log information. It can be configured via log4j.properties. You may remove this file: afterwards, SmartGit will return to its default logging settings.
* passwords is an encrypted file and stores the passwords used throughout SmartGit. You may remove this file: afterwards, all passwords are lost.
* accelerators.xml stores the accelerators configuration. You may remove this file: afterwards, all accelerators will be reset to their defaults.
* credentials.xml stores authentication information (not including the corresponding passwords). You probably do not want to remove this file: afterwards, all credentials (user names, private keys, certificates) will be lost.
* hostingProviders.xml stores information about configured hosting provider accounts (not including the corresponding passwords). You probably do not want to remove this file: afterwards, all connect details for all hosting provides will be lost.
* notifications.xml stores information about the state of notifications which show up in the status bar in various cases. You may remove this file: afterwards, various notifications may show up again.
* projects.xml stored all former SmartGit projects (up to SmartGit 5) including their settings. Beside that it contains all repository root specific settings.
* repositories.xml stores the information about known repositories, their names and repository groups.
* repository-cache.xml stores all cached information about repository states, e.g. what local branch is checked out, whether there are incoming or outgoing changes.
* settings.xml stores the application-wide settings (e.g. the preferences) of SmartGit. You should not remove this file, unless you want to completely reset SmartGit.
* tools.xml stores external tools which have been configured in the Preferences. You probably do not want to remove this file: afterwards, all you external tools configurations will be lost.
* ui-config.xml stores UI related, more stable settings, e.g. the toolbar configurations. You may remove this file: afterwards, various aspects of the UI will be reset to defaults.
* ui-settings.xml stores UI related, volatile settings, e.g. window sizes or column widths. You may remove this file: afterwards, various aspects of the UI will be reset to defaults.

### Resetting certain parts of the configuration to defaults

To reset certain parts of SmartGit's configuration ("settings") to the defaults:

1. locate the appropriate configuration file (\*.xml)
2. Exit SmartGit, using **Repository|Exit**
3. Get rid of the file(s)
4. Start SmartGit again

## Program Updates

SmartGit stores program updates which have been downloaded automatically through SmartGit itself by default in the subdirectory updates of the Settings root directory (see [Default Location of SmartGit's Settings Directory](http://www.syntevo.com/doc/display/SG/Installation+and+Files#InstallationandFiles-settings-dir.default-location)). This allows light-weight, patch-like updates which do not require write access to the actual SmartGit installation directory. As a consequence, your SmartGit installation directory is usually not up-to-date, but it will launch the downloaded updates from the updates directory. Only under specific conditions, SmartGit will detect that an upgrade of the installation directory itself is necessary.

**Tip**

You can manually trigger the update of the installation directory from the **About** dialog, section **Information**, **...**-button right beside **Version**.

If you prefer to keep your SmartGit installation always up-to-date, you can select **Update SmartGit application in place** in the Preferences, section**SmartGit Updates**. Note, that updating with this option selected may require administrator privileges.

### Technical Details

The root directory of the Updates directory contains sub-directories for every major version. Such a major version directory contains a control file for the latest downloaded build and a current-file which points to the currently used build. Usually, this will be the highest build which shows up in this directory. Thecontrol-file only configures which binaries are part of the build by linking to the actual binaries which are stored in the repo-subdirectory and which are shared among all builds.

Each new build has a corresponding, digitally-signed control file which contains information about all required application files with their download location and the expected file content hash. To reduce band-width, application files only will be downloaded if they are not yet locally available. After download, the content will be verified with the hash from the control file.

When starting SmartGit, the bootloader.jar from the installation directory is launched. This uses the control file from the Updates directory to determine which updated SmartGit files to launch that contain the actual application code.

**Warning**

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

## Company-wide Installation

For company-wide installations, the administrator may install SmartGit 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 inSmartGit.app/Contents/Resources/ (parallel to the Java directory), for other operating systems within SmartGit's installation directory (parallel to the lib andbin directories).

When a user starts SmartGit for the first time, the following files will be copied from the default directory (on the network share) to the user's personal SmartGit settings directory (refer to [Default Location of SmartGit's Settings Directory](http://www.syntevo.com/doc/display/SG/Installation+and+Files#InstallationandFiles-settings-dir.default-location)):

* accelerators.xml
* credentials.xml
* hostingProviders.xml
* repository.xml
* settings.xml
* tools.xml
* ui-config.xml
* ui-settings.xml

The license file (only for 10+ user Commercial 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**

Be sure to name the license file license in the default directory without any extension.

## JRE Search Order (Windows)

On Windows, the smartgit.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's installation directory
* Environment variable JAVA\_HOME
* Environment variable JDK\_HOME
* Registry key HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment

## [Tips and Tricks](http://www.syntevo.com/doc/display/SG/Tips+and+Tricks?src=contextnavpagetreemode)

## Autoscrolling while Drag and Drop

To make a tree or table control scroll automatically while dragging items, be sure to wait a short time over the first or last row of the control.

Feel free to cast your vote to [this Eclipse issue](https://bugs.eclipse.org/bugs/show_bug.cgi?id=457108) to get normal auto-scrolling behavior.

## Completion

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

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

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

## Text Editors

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

## Compare

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

## Speed Search

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

## Table Columns

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

## [How Tos](http://www.syntevo.com/doc/display/SG/How+Tos?src=contextnavpagetreemode)

### [Configuration](http://www.syntevo.com/doc/display/SG/Configuration?src=contextnavpagetreemode)

#### [Changing the Folder Watch Limit (Linux)](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704436&src=contextnavpagetreemode)

SmartGit's file monitor watches the folders of your Git repositories for file changes, and updates all affected views automatically so you don't have to refresh them by hand. On Linux, the file monitor requires Java 7 or later, and the number of folders that can be watched simultaneously is limited by a system-wide variable.

This means if you open a Git repository in SmartGit, and this repository contains more subfolders than is allowed by the system-wide limit, the file monitor may stop working. If that happens, you'll need to raise the watch limit. This can be done as follows: For example, to raise the watch limit to 100K watches, log in as root (described below) and insert the following line into the file /etc/sysctl.conf:

|  |
| --- |
| fs.inotify.max\_user\_watches = 102400 |

For this change to take effect, either reboot the system or execute the command /sbin/sysctl -p.

**Tip**

To understand implications of increasing the limit, have a look at <http://askubuntu.com/questions/154255>.

In order to obtain root privileges, enter su, followed by the root password. On some systems, you may have to "unlock" the root account the first time. To do so, enter sudo su instead.

#### [Enabling the "cherry picked from commit ..." message](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704435&src=contextnavpagetreemode)

Git's cherry-pick command has an -x option that tells Git to automatically append a "cherry picked from commit ..." message to the created commits, in order to indicate where the latter came from. SmartGit does not do this by default, but you can enable it by setting the following system property:

|  |
| --- |
| -Dsmartgit.cherryPick.shaPrefix="cherry picked from commit" |

For more information on how to set system properties, see :EXTREF:the manual:EXTREF:.

#### [Setting up an SSH tunnel (Windows)](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704438&src=contextnavpagetreemode)

This page explains how to set up an SSH tunnel on Windows using Putty, for the purpose of routing traffic between SmartGit and your Git repository through the SSH tunnel.

When you start Putty, a configuration window shows up where you can configure the connection you're about to open. On the **Session** page, enter the host name or IP address and the port number of the connection. For this tutorial let's say we'll connect to my.private.server, port number 22. On the **Session** page, you may save your configuration as a session.

Next up, enter the required authentication information on the pages **Connection|Data** and **Connection|SSH|Auth**. For this example, we'll log in as user john, using the private key file F:\certificates\my-git-certificate.ppk.

Now comes the part where we set up the SSH tunnel: On the page **Connection|SSH|Tunnels**, enter the source and destination of the tunnel. We'll use 2022 as source port and my.private.server:22 as destination. Leave the other options as they are, i.e. **Local** and **Auto**. Click the **Add** button to add the tunnel to the connection configuration.

Finally, click the **Open** button at the bottom of the Putty configuration window to open the connection. Now, while the connection is open, you can access your Git repository from SmartGit through the SSH tunnel. In our example, we can now clone the Git repository via a URL like this:ssh://john@localhost:2022/path/to/gitrepo

#### [Setting up Multiple URLs per Remote](http://www.syntevo.com/doc/display/SG/Setting+up+Multiple+URLs+per+Remote?src=contextnavpagetreemode)

In Git you can associate multiple URLs with a remote. This allows you to push your commits to multiple remote repositories in a single step. To set this up, open the config file in the .git folder beneath your Git repository and add your URLs under the appropriate remote entries.

Example: Suppose your Git repository is named myproject. Open the file myproject/.git/config with a text editor. Assuming you have remote named *origin*, your modified config file might look like this:

|  |
| --- |
| ...  [remote "origin"]      fetch = +refs/heads/\*:refs/remotes/origin/\*      url = C:/work/repo.git      url = C:/backup/repo.git  ... |

Now when you do a push either from the terminal (e.g. git push origin) or from within SmartGit, your commits will be send to all repositories you added in theconfig file.

#### [SVN: cloning with a custom branch layout](http://www.syntevo.com/doc/display/SG/SVN%3A+cloning+with+a+custom+branch+layout?src=contextnavpagetreemode)

There are scenarios where the default branch-tag-layout may not be sufficient. Following procedure allows you to customize the layout:

* Edit the smartgit.properties file that you can find in :EXTREF:SmartGit's settings directory:EXTREF:.
* Add following line to the beginning of the file:

|  |
| --- |
| smartgit.debugMenu=true |

* Restart SmartGit.
* Select **Project|Clone**, enter the URL, press **Next**, then check **Just initialize clone (expert mode)** and complete the wizard.

SmartGit will initialize the repository without starting cloning. Don't press Pull once finished, but exit SmartGit (**Project|Exit**).

* Edit path/to/repository/.git/svn/.svngit/svngitkit.config to configure your custom layout by defining the additional-branches key.
* Start SmartGit, reopen your project and invoke **Pull**.

#### Example: trunk has been copied from different project

To fetch an *old* trunk from a different project, a configuration could look like as follows (there must be no line break for the whole additional-branches key).

|  |
| --- |
| [svn-git-remote "svn"]  url = <http://servername/svn/MyRepository>  fetch = ProjectB/trunk:refs/remotes/svn/trunk  branches = ProjectB/branches/\*:refs/remotes/svn/branches/\*  tags = ProjectB/tags/\*:refs/remotes/svn/tags/\*  additional-branches =  "ProjectA/trunk:refs/remotes/svn-old/trunk;ProjectA/branches/\*:refs/remotes/svn-old/branches/\*;ProjectA/tags/\*:refs/remotes/svn-old/tags/\*" |

#### [What do I do if I want to rename/move files or directories?](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704437&src=contextnavpagetreemode)

If you've been using SVN before trying out Git, you might wonder what you should do to properly rename or move files and directories in your Git repository. The answer is that you can simply rename or move them. This is due to the fact that Git handles renaming and moving quite differently from SVN:

* **SVN** In SVN, you have to explicitly mark renames and moves as such, otherwise SVN will think you've created a new file/directory.
* **Git** In Git, you just rename or move your files/directories, and later Git will automatically detect renames. For details, refer to [Local Operations on the Working Tree](http://www.syntevo.com/doc/display/SG/Local+Operations+on+the+Working+Tree#LocalOperationsontheWorkingTree-LocalOperationsontheWorkingTree-moving-files)

### [Debugging](http://www.syntevo.com/doc/display/SG/Debugging?src=contextnavpagetreemode)

#### [Creating Thread Dumps](http://www.syntevo.com/doc/display/SG/Creating+Thread+Dumps?src=contextnavpagetreemode)

By creating thread dumps, you can help us fix program lock-ups (UI freezes, never-ending operations, etc.) and performance problems, among other things. How a thread dump is created depends on your operating system. In the following, we'll assume SmartGit has been installed in the directory install-dir.

#### Windows

* Shut down SmartGit if it's still running. Also check that it isn't hidden in the system tray, and make sure no more SmartGit processes are running, by terminating any smartgit.exe (and smartgitc.exe) processes in the Windows Task Manager.
* Open a command prompt and navigate to the directory <install-dir>\bin.
* Execute bin\smartgitc.exe > dump.txt. This will start SmartGit and keep the command prompt attached to the program.
* Perform the necessary actions to reach the program state for which you want to create a thread dump.
* In the command prompt, press <Ctrl>+<Break>-keystroke to create a thread dump. Then wait a few seconds to create another dump and repeat this step at least five times in order to get a reasonable number of dumps. Check that the thread dumps were actually written to dump.txt.
* Follow the instructions at the bottom of this section to send the thread dumps to our customer support.

#### Mac OS X

* If SmartGit is already running, quit it.
* Open a terminal window and launch SmartGit redirecting the output to a file: <install-dir>/SmartGit.app/Contents/MacOS/SmartGit > ~/dump.txt
* Perform the necessary actions to reach the program state for which you want to create a thread dump.
* Open a second terminal window and execute ps -A | grep SmartGit to find the process ID (PID) of the running SmartGit instance.
* Execute kill -3 PID where PID is the process ID you obtained in the previous step. This will append a thread dump to ~/dump.txt. Then wait a few seconds and repeat this step at least 5 times in order to get a reasonable number of dumps.
* Check that the thread dumps were indeed written to the dump.txt files.
* Follow the instructions at the bottom of this section to send the thread dumps to our customer support.

#### Linux and other Unix-like operating systems

* Open a terminal window.
* Shut down SmartGit if it's still running. Also make sure it's not hidden in the system tray and that no more SmartGit processes are running. The latter can be checked as follows: SmartGit runs as a java process, and you can enter ps -A | grep java in the terminal to find all running java processes. If you have other Java applications running, you may execute ps -Af | grep java to find out the process IDs (PID) of the SmartGit processes. Terminate all of these processes, if there are any.
* Execute <install-dir>/bin/smartgit.sh > dump.txt.
* Perform the necessary actions to reach the program state for which you want to create a thread dump.
* Open another terminal window.
* Execute ps -a | grep java to find out the process ID of SmartGit's java process. If there's more than one, you'll have to try them all.
* Send the KILL signal to SmartGit by executing kill -3 PID in the second terminal window, where PID is the process ID you obtained in the previous step. Wait a few seconds and create a another dump. Repeat this step at least 5 times. After you're done, make sure the file dump.txt has actually been filled.
* Send the thread dumps to our customer support, as explained below.

How to send the dump.txt file: Compress the dump.txt file as a zip or tar/gzip archive, and also include the file log.txt from :EXTREF:SmartGit's settings directory:EXTREF:. Send this archive, as well as a short description of how to reproduce the problem, to [smartgit@syntevo.com](mailto:smartgit@syntevo.com).

#### [Periodical Thread Dumps](http://www.syntevo.com/doc/display/SG/Periodical+Thread+Dumps?src=contextnavpagetreemode)

Logging periodical thread dumps is a built in feature of SmartGit to track well reproducible, pervasive performance problems. To investigate a problem using periodical thread dumps, do the following:

* While SmartGit is running, locate the settings directory: invoke **Help|About SmartGit** and check the **Information** page for the settings directory (**Settings Path**).
* Shutdown SmartGit (use **Repository|Exit** and afterwards make sure that no more SmartGit process is running).
* In the settings directory, open smartgit.properties (create this file if necessary) and add following line to enable the **Debug** menu:

|  |
| --- |
| smartgit.debug=true |

* Remove all log.txt\* files from the settings directory.
* Startup SmartGit.
* Trigger the slow operation.
* Invoke **Debug|Create Periodical Thread Dumps.**
* After 10 seconds, shutdown SmartGit again.
* Compress all log.txt.\* files from the settings directory as a zip or tar/gzip archive and send this archive, as well as a short description of how to reproduce the problem, to [smartgit@syntevo.com](mailto:support@smartsvn.com).

#### [Debugging SSL-related problems for SVN repositories](http://www.syntevo.com/doc/display/SG/Debugging+SSL-related+problems+for+SVN+repositories?src=contextnavpagetreemode)

When experiencing problems to connect to SVN repositories over the *https*-protocol, it can be helpful to debug Java's networking layer. To do that, first add the system property javax.net.debug=ssl to the :EXTREF:smartgit.properties:EXTREF: file. After that, restart SmartGit from the command line and pipe the output to a file:

* On Windows: execute <install-dir>\bin\smartgitc.exe > dump.txt.
* On Mac OS: execute <install-dir>/Contents/MacOS/SmartGit > dump.txt.
* On Linux: execute <install-dir>/bin/smartgit.sh > dump.txt.

Finally, verify that dump.txt contains SSL-related output.

*How to send the dump.txt file*: Compress the dump.txt file as a *zip* or *tar/gzip* archive, and also include the file log.txt from :EXTREF:SmartGit's settings directory:EXTREF:. Send this archive, as well as a short description of how to reproduce the problem, to [smartgit@syntevo.com](mailto:smartgit@syntevo.com).

#### [Inspecting SmartGit's process using JVisualVM](http://www.syntevo.com/doc/display/SG/Inspecting+SmartGit%27s+process+using+JVisualVM?src=contextnavpagetreemode)

# [Inspecting SmartGit's process using JVisualVM](http://www.syntevo.com/doc/display/SG/Inspecting+SmartGit%27s+process+using+JVisualVM)

*JVisualVM* is a Java Profiler which is part of the JDK and can be helpful to diagnose various problems.

* Install the latest *JDK* from <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (unless you already have a JDK installed).
* Run JVisualVM by invoking bin/jvisualvm.exe (on Windows). On Linux and Mac, locate jvisualvm which should be in the same directory as javac.
* Once JVisualVM has started up, locate the *SmartGit*-related VMs: there will be one *main* VM, if SmartGit is running (on Windows showing up asSmartGit, on Mac/Linux showing up as QBootLoader). If a SmartGit upgrade is in progress, there will be an additional VM (on Windows showing up asSmartGitUpdater). If SSH connections are currently in progress, there may be more VMs.

**Note**

If you can't connect to *SmartGitUpdater* VM, try again running JVisualVM as administrator, because SmartGitUpdater is usually invoked with administrative privileges, too.

#### Taking thread dumps

Thread dumps are useful to investigate hangs or slow performance.

* Select the VM you want to inspect.
* Select tab **Threads** and invoke **Thread Dump**.
* When investigating a problem, it's usually a good idea to create a couple of thread dumps.

#### Taking a heap dump

Heap dumps are useful to investigate memory-related problems.

* Select the VM you want to inspect.
* Select tab **Monitor** and invoke **Heap Dump**.
* The **Classes** section will give you a good overview of the memory usage: sorting by **Instances** and **Size** can be helpful to detect (unusually) large occurrences of specific classes. By double-clicking a certain class, you will switch to the **Instances** view which gives you detailed information for every object of this class.

**Note**

For heap dumps, it's expected to see a lot of char[], byte[] and String objects, as version control is more or less all about bytes and texts (paths, file contents, commit messages).

#### [Obfuscating a repository to report a problem](http://www.syntevo.com/doc/display/SG/Obfuscating+a+repository+to+report+a+problem?src=contextnavpagetreemode)

Obfuscating a repository means building another repository which has the same commit and file system structure as the original repository, however file contents and file names are replaced by 'dummy' texts. In this way, possibly confidential information is removed from the repository.

Obfuscated repositories are helpful to debug *certain* kind of problems. Following steps are necessary:

#### Preparation

* Make sure the problem is *reproducible* with your *original* repository.
* Open a terminal and cd to your repository root.

#### Obfuscate the repository

* On Windows: execute <install-dir>\bin\smartgitc.exe --obfuscate <temp-dir>/repo.obfuscated > mapping.txt.
* On Mac OS: execute <install-dir>/Contents/MacOS/SmartGit --obfuscate <temp-dir>/repo.obfuscated > mapping.txt.
* On Linux: execute <install-dir>/bin/smartgit.sh --obfuscate <temp-dir>/repo.obfuscated > mapping.txt.

This will give you an *obfuscated* repository in <temp-dir>/repo.obfuscated and a mapping.txt in your current repository root, which contains the output of the obfuscation process as well as a mapping from *original* to *obfuscated* file names.

#### Report Problem

* Open the *obfuscated* repository in SmartGit and make sure the problem is still reproducible: use mapping.txt to map file names from the *original*repository to the obfuscated repository.
* Create an *ZIP* or *tar.gz* archive of the obfuscated repository and send it to *smartgit@syntevo.com*, together with a description and *obfuscated* file names on how to reproduce the problem. Be sure to *NOT* include mapping.txt in the email, as this file is just for you.

#### [Tracing the modified state of a file](http://www.syntevo.com/doc/display/SG/Tracing+the+modified+state+of+a+file?src=contextnavpagetreemode)

When the reason for the **modified** state of a specific file is unclear, following procedure allows to debug details for this file.

### Compare with git status

Give git status a try and see whether the same modification is present there. If so, you may still run through this procedure, but likely the answer to this question is found elsewhere (reason may be a wrong .git/config or EOL-related issues).

If invoking git status does not show the same modification or even resets the modification in SmartGit, the resulting debug information should be sufficient to understand the difference between SmartGit's status detection and Git's status.

### Procedure

* Shutdown SmartGit.
* Add smartgit.statusscanner.trace=<relative-path> to [smartgit.properties](http://www.syntevo.com/doc/display/SG/System+Properties) (in the settings directory, see About dialog).  
  <relative-path> is the repository-relative path of the file for which the modification should be traced. On Windows, also replace backslashes by forward slashes.

**Example**

On Windows, for example for file C:\vss\Project\FOO\bar\file with repository root C:\vss\Project, you should add line:

smartgit.statusscanner.trace=FOO/bar/file

* Get rid of all existing log.txt\* files in the settings directory.
* Restart SmartGit.
* Open the offending repository and wait until the Refresh has finished and the modified file is showing up.
* Make sure the log.txt\* files contain debug line for the smartgit.status.index category.  
  You may use grep "smartgit.status.index" log.txt\* to find out.
* ZIP the log.txt\* files and send to smartgit@syntevo.com, together with an explanation of the problem.

### [Performance Tuning](http://www.syntevo.com/doc/display/SG/Performance+Tuning?src=contextnavpagetreemode)

# Refreshing

There are a couple of possible reasons which may result in slower refreshing:

* **Insufficient heap size:** try to increase the [maximum memory limit](http://www.syntevo.com/doc/display/SG/VM+options).
* **Network Shares:** make sure, your repository and Git's HOME directory is located on a local drive.
* **Symlinks:** try to open the repository from its real location. When opening a repository over a symlink (Windows, too!), refreshing may be slower.
* **Background processes (since version 6):** in the **Preferences**, section **Background Commands**, try to disable all options related to **Refreshing** and**Local and Remote Changes**.
* **Big Repositories:** if SmartGit is close to the maximum memory limit, refreshing may become very slow. Usually, the pre-configured memory limit is sufficient, however for large repositories it may be too low. Try to increase the limit according to [these instructions](http://www.syntevo.com/doc/display/SG/VM+options).
* **.git/index timestamps wrong:** sometimes .git/index timestamps may be incorrectly set/rounded, resulting in a content comparison for all your working tree files. A possible solution may be to delete .git/index and then invoke git reset --hard from the command line.

**Warning**

Be sure that you have no local nor staged changes, before doing so!

If you encounter wrong .git/index timestamps frequently, please contact [smartgit@syntevo.com](mailto:smartgit@syntevo.com)

# Log

* If opening the Log takes a long time, make sure that you deselect as many **Branches** as possible, in the ideal case only leaving **HEAD** selected. Once reopening the Log, it should be faster.
* Large amounts of tags (or branches) may slow down Logs, but also the Working Tree window. For tags, you may try to use system propertysmartgit.branch.tagExcludeRegEx (supported since SmartGit 6.5.4) and for remote branches smartgit.branch.remoteBranchExcludeRegEx (supported since SmartGit 7.2) to exclude the specified set of refs from processing. It may be necessary to invoke a git pack-refs --all after doing so, to have the changes take effect in SmartGit.

If nothing of the above helps, shutdown SmartGit, get rid of all log.txt\*-files from the Settings directory (see About dialog), restart SmartGit, capture the slowness and send resulting log.txt\*-files to [smartgit@syntevo.com](mailto:smartgit@syntevo.com).

# Miscellaneous

* We have received user reports that SmartGit's user interface may be slow if **SuperPuTTY** version 1.0.4.6 (or lower) is **running**. Especially, closing SmartGit may take several seconds.

### [Purchase-Related](http://www.syntevo.com/doc/display/SG/Purchase-Related?src=contextnavpagetreemode)

# Purchase Overview

To get an overview of all your Cleverbridge orders, please provide your e-mail address at <https://www.cleverbridge.com/?scope=cusecolp>.

#### [Purchase Upgrades](http://www.syntevo.com/doc/display/SG/Purchase+Upgrades?src=contextnavpagetreemode)

There are 2 kinds of purchasing licenses for SmartGit, the perpetual license and the subscription license. When purchasing perpetual licenses, you will get at 1, 2 or 3 years of support and updates - depending on the option you choose. This means, you can use any version of SmartGit that has been release up to 1, 2 or 3 years after the purchase date, including all belonging bugfix releases.

If this update period is expired, you still continue to use SmartGit, but it will be unsupported and you won't be able to use newer versions of SmartGit. To renew the support and update period, you need to purchase upgrade licenses. Basically, upgrade licenses are the same as new perpetual SmartGit licenses, but for a discounted price.

The easiest way is to use **Help|Register** in SmartGit and click the **Purchase Update** button. Alternatively, you may open<http://www.syntevo.com/smartgit/purchase-update> in your browser and upload your existing license file.

#### [Subscription Licenses](http://www.syntevo.com/doc/display/SG/Subscription+Licenses?src=contextnavpagetreemode)

Subscription licenses are valid only for a certain period. They renew automatically which requires a payment type that can be renewed automatically, e.g. credit card or Paypal. After each successful subscription payment you will get an email with the latest SmartGit license file. You will need the latest SmartGit license file to register a new copy of SmartGit - an older, already expired license will not work.

SmartGit 6.5 or newer will check for updated subscription licenses periodically and more frequently towards the end of the subscription period, so usually the user will not notice subscription management/update at all. This will require SmartGit to have access to the internet and, if necessary, a correctly configured proxy configuration in SmartGit.

Instead of ordering one subscription license for multiple users, you may order single-user licenses for explicitly named users. That way you easily can cancel subscriptions of users who leave your company.

## Increase the Number of Licenses

If you want to increase the number of subscription licenses, you have following choices.

* Just order the additional licenses. This way you will have two (or more) licenses running.
* Cancel the running subscription and if the licenses expire after one month, order the new overall subscription license count.

## Decrease the Number of Licenses

If you want to increase the number of subscription licenses, you may cancel the running subscription and if the licenses expire after one month, order the new overall subscription license count.

### [Trouble-shooting](http://www.syntevo.com/doc/display/SG/Trouble-shooting?src=contextnavpagetreemode)

#### [git: '...' is not a git command](http://www.syntevo.com/doc/display/SG/git%3A+%27...%27+is+not+a+git+command?src=contextnavpagetreemode)

If running certain commands from SmartGit fails, which work when running from command line, this is usually caused by a different PATH environment of SmartGit. This happens e.g. when extending the PATH in files like ~/.bash\_profile, ~/.bash\_login, and ~/.profile. Because SmartGit is not starting an interactive shell, these files won't be processed. The solution is to specify the PATH either in a file which will be processed by the system when starting SmartGit or by adding PATH= to smartgit.vmoptions, see [here](http://www.syntevo.com/doc/display/SG/VM+options).

Typical failures:

* git: 'credential-osxkeychain' is not a git command.

#### [How to fix an Index which contains the same file name multiple times, but with different encodings](http://www.syntevo.com/doc/display/SG/How+to+fix+an+Index+which+contains+the+same+file+name+multiple+times%2C+but+with+different+encodings?src=contextnavpagetreemode)

## Background

The *Index* file (.git/index) contains meta-information related to the working tree and is constrained by Git to have exactly one entry per file (and stage level). While Git handles file names as raw *byte-sequences*, SmartGit interprets them as *(Unicode-)Strings* to display them more nicely. Certain parts of a byte sequence may not represent a valid encoding, hence the relevant parts of the byte sequence will be replaced by the 'Unicode replacement character ' (�, U+FFFD). If the *Index* contains two byte sequences which differ only in 'unconvertible' parts, the resulting *Strings* may be identical. Such *Index* entries are confusing (not only SmartGit) and need to be fixed.

**Note**

This kind of problem might occur after updating to *msysgit 1.7.10* and not properly converting non-unicode file names.

## Resolution

If an encoding problem in the *Index* is encountered, SmartGit will show an error message which contains information about the problematic file. The resolution is performed using Git from the command line:

* cd to your repository.
* Invoke git status to be sure you have a clean working tree and Index. If not clean, either commit or stash away your local modifications.
* For msysgit, on Windows: invoke bash.exe to enter Git's shell.
* Invoke git ls-files | grep <readable-part-of-the-file-name>.
* Inspect the output of the command and identify the offending entries: usually they will be almost identical, but only differ at certain positions where the *octal* encoding (\xyz) slightly varies. In the next step we will remove these offending entries from the repository (and re-add them again, if necessary). Hence, for every offending entry*<path-to-file>*:
* Make a backup of *<path-to-file>* and
* Invoke git rm --cached "`printf "<path-to-file>"`" (Note: all the quotes are necessary!).

**Note**

On Windows, you may also work without bash.exe: just open a *command line (DOS box)* and invoke git rm --cached <path-to-file-pattern> with <path-to-file-pattern> being an unambiguous pattern of the file's path where the non-ASCII characters are replaced by ?. E.g.: x?y.txt.

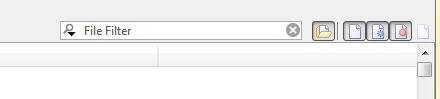
* After all entries have been removed, invoke git commit -m "clean up bad encoding of file names".
* If necessary, re-add the entries (this time with correct encoding) with git add . and finally commit with git commit --amend -m "clean up bad encoding of file names".
* Now SmartGit should be able to open the repository. Don't forget to push your changes!

#### [My files are displayed with invalid characters, what can I do?](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704447&src=contextnavpagetreemode)文件乱码怎么办

If your files are displayed with invalid characters, you'll need to change the encoding scheme. To do so, open the Repository Settings dialog with **Project|Repository Settings** and select an appropriate encoding on the *Text Encoding* tab.

#### [Why are my files/directories not shown?](http://www.syntevo.com/doc/pages/viewpage.action?pageId=1704445&src=contextnavpagetreemode)为什么目录、文件不显示

If some of your files or directories are not showing up on the file table in the project window, check your settings in the filter component on the top right of the file table:



The buttons on the right of the filter component control the visibility of various types of files and directories. For instance, you can toggle the visibility of ignored files and directories (i.e. files and directories that match the rules specified in your .gitignore) by changing the selection state of the right-most button of the filter component.

### [Workflows](http://www.syntevo.com/doc/display/SG/Workflows?src=contextnavpagetreemode)工作流

#### [How to compare two branches/commits?](http://www.syntevo.com/doc/pages/viewpage.action?pageId=2719994&src=contextnavpagetreemode)比较分支、提交

* directly in the **Commits** graph, select the two commits you want to compare (Ctrl-click the second commit to extend the selection);
* in the **Branches** view, use **Reveal Commit** on the first branch you want to compare, then use **Compare with Selected Commit** on the second branch you want to compare (this will again result in the selection of two commits in the **Commits** graph).

Once exactly two commits are selected, the **Files** view will show the changes from the older to the younger commit. To see content changes of an individual file, select the file and check the **Changes** view.

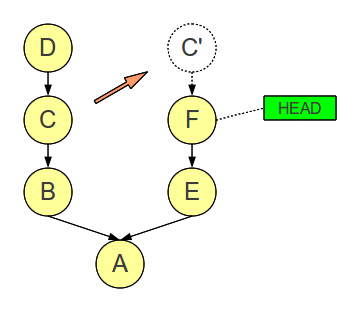
#### [How to examine the content of a stash](http://www.syntevo.com/doc/display/SG/How+to+examine+the+content+of+a+stash?src=contextnavpagetreemode)

Stashes are saved modifications of the working tree. Saving a stash allows to quickly clean-up a working tree without loosing the changes. Later one can re-apply the saved stash to continue with the partly done work.

To examine the saved modifications (the content of a stash), open the log, ensure that the checkbox in front of the stash in question in the *Branches* view is selected and scroll to the stash. You will see up to 3 commits referring to the index and working tree changes as well as to untracked files.

#### [How to perform a cherry-pick](http://www.syntevo.com/doc/display/SG/How+to+perform+a+cherry-pick?src=contextnavpagetreemode)

A cherry-pick allows you to take the changes from a single commit of another branch and apply these changes on top of the current branch. This operation is illustrated below:



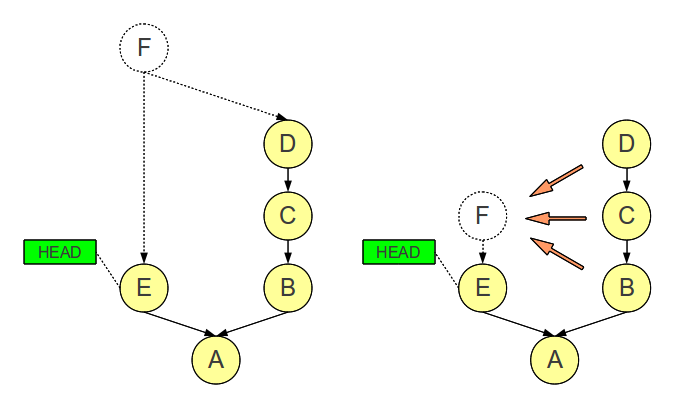
Here, we have two branches and our HEAD is at commit F. Now we want to take the changes made in commit C and apply them on top of F.

In SmartGit, there are two ways to perform a cherry-pick, either from the main window or from the log window. In both cases, you have to first check out the branch containing commit F.

* Main window: From the main menu, select **Branch|Merge** . On the Merge dialog that shows up, select the commit to cherry-pick from, then select the option **Only the selected commits (cherry-pick)**. Finally, click on the **Merge** button to perform the cherry-pick operation.
* Log window: On the log graph pane, right-click on the commit to cherry-pick from. From the context menu, select **Cherry-Pick**. On the following dialog, choose whether you just want to apply the changes from the selected commit to the working tree, or whether you want to apply the changes to the working tree and commit these changes as well.

#### [How to perform normal merges and squash merges](http://www.syntevo.com/doc/display/SG/How+to+perform+normal+merges+and+squash+merges?src=contextnavpagetreemode)

A normal merge allows you to join two branches. A squash merge is similar to a normal merge, except that the information about where the merged-in changes came from is not recorded. This is illustrated below:



A normal merge is shown on the left, and a squash merge on the right. In the normal merge, a new commit F is created by merging the changes from another branch into the current branch. As a result, F has the two parents D and E. In the squash merge on the other hand, the new commit F is created by combining the commits B, C and D, and applying them on top of E. In the history, it will then look as if F had a single parent, namely E. You may of think of squash merging as cherry-picking multiple commits instead of just one.

In SmartGit, normal and squash merges can be performed either from the main window or the log window. In either case, make sure you've checked out the branch onto which you want to apply the merge commit.

* Main window: From the main menu, select **Branch|Merge**. On the Merge dialog that shows up, first select the topmost commit (i.e. the newest) from the ones you want to want to merge or squash merge into the current branch. If you want to perform a normal merge, choose either **Merge to Working Tree** or **Create Merge-Commit**. The second option will automatically create a commit. In the case of a squash merge, you have to select the **Merge to Working Tree** option.
* Log window: On the log graph pane, right-click on the topmost commit from the ones to merge or squash merge into the current branch. In the context menu, click on**Merge**. Now a confirmation dialog shows up. If you want to perform a normal merge, choose either **Merge to Working Tree** or **Create Merge-Commit**. The second option will automatically create a commit. In the case of a squash merge, you have to select the **Merge to Working Tree** option.

Regardless of whether you merged from the main window or the log window, if you had selected **Create Merge-Commit**, you are most likely done. If there were merge conflicts or you had selected **Merge to Working Tree**, conflicts have to be resolved now and you'll finally need to perform a commit (e.g. by selecting **Local|Commit** in the menu) to conclude the merge operation. On the commit dialog, you can choose now whether you want to perform a normal merge (**Merge Commit**) or a squash merge (**Simple commit**). Enter a commit message, select the appropriate merge type, then click on **Commit**.

#### [How to perform squash merges](http://www.syntevo.com/doc/display/SG/How+to+perform+squash+merges?src=contextnavpagetreemode)

In SmartGit, squash merges can be performed either from the main window or from the log window. In both cases, you first have to make sure you've checked out the branch onto which you want to apply the merge commit.

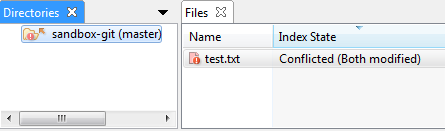
* **From the main window:** In the main menu, select **Branch|Merge**. On the Merge dialog that shows up, first select the topmost commit (i.e. the newest) from the ones you want to want to squash merge into the current branch. Second, make sure the option **Branch consisting of selected commit and its ancestors** is selected. Then click on the **Merge** button.
* **From the log window:** On the log graph pane, right-click on the topmost commit from the ones to squash merge into the current branch. In the context menu, click on**Merge**. On the confirmation dialog that shows up, select the **Merge to Working Tree** option.

Regardless of whether you merged from the main window or the log window, your working tree is now in a merging state. If there are any conflicts, you'll have to resolve them before you can proceed. Resolving conflicts is covered in another How-To: [How to resolve conflicts](http://www.syntevo.com/doc/display/SG/How+to+resolve+conflicts#Howtoresolveconflicts-workflows.resolve-conflicts).

If there weren't any conflicts, or after all conflicts have been resolved, you can perform a commit (e.g. by selecting **Local|Commit** in the menu) to finish the squash merge operation. On the commit dialog, you'll get to choose whether you want to perform a normal merge or a squash merge. Select the latter option and enter a commit message, then click on the **Commit** button.

#### [How to resolve conflicts](http://www.syntevo.com/doc/display/SG/How+to+resolve+conflicts?src=contextnavpagetreemode)解决冲突

A merge, rebase or cherry-pick may fail due to conflicting changes between two or more commits. If that happens, SmartGit will halt the operation and remain in a conflict state. The following screenshot shows what the *Directories* pane and *Files* pane would look like in that state:



There are two ways to leave the conflict state: Resolving the conflicts and continuing the operation, or canceling the operation.

* **Resolving the conflicts:** To resolve the conflicts, go through all conflicted files one by one and modify them as needed. SmartGit provides two features to facilitate this: First, there is a three-panel editor called the *Conflict Solver*. It can be opened for the currently selected file either through the menu entry **Query|Conflict Solver** in the main menu or through the file's context menu. Second, you can instruct SmartGit to resolve the conflicts in a particular file by choosing a certain version of it. To do this, open the*Resolve* dialog, either through the main menu **Local|Resolve**, or through the file's context menu, and choose the version of the file to keep.
* **Canceling the operation:** You can cancel the operation that led to the conflict state by selecting the project root in the *Directories* pane and invoking the *Discard* command. The latter can be done either by clicking on the *Discard* button on the main toolbar, or through the main menu: **Local|Discard**.

#### [How to set up a Git server](http://www.syntevo.com/doc/display/SG/How+to+set+up+a+Git+server?src=contextnavpagetreemode)设置Git服务器

Though Git is a distributed version control system, a common usage setup is similar to, e.g. SVN, having one central Git server repository and multiple user-specific Git repositories.

Of course, you could use cloud based Git hosting providers like GitHub, Assembla or BitBucket, but here we want to cover the topic of using an own server. Some Git hosting providers also offer a version that you could install on your own server. A free and good one is GitLab ([www.gitlab.com](http://www.syntevo.com/doc/display/SG/www.gitlab.com)).

Usually, Linux servers are used. They make authentication with SSH simple and, if there already runs an Apache, https-authentication also should be quite easy to install. We don't recomment network shares as central Git repository location, if it is not for reliability then for the possibility to delete or manipulate it directly without intent. If you want to install on a Windows server, setting up SSH authentication can be tricky, so one either has to install Apache to handle https authentication or directly use an own hosting provider like GitLab.

On the Git server, you will need to have installed the Git executables. Download from <http://git-scm.com/downloads> and install correctly.

Basically, it is possible to pull from and push to any repository, but Git has problems to push branch changes to a repository with working copy if it involves changing the checked out branch. On servers usually you don't need a Git working tree (you usually don't edit and commit directly there). Hence, socalled *bare* repositories without a working tree are used. The directories of bare Git repositories usually end with .git. To create one, invoke

|  |
| --- |
| git init --bare /path/to/repository.git |

No matter whether you will use SSH or https, you should add all allowed users to the same group, e.g. *gitusers* and set the guid permissions recursively on the created repository directory structure.

|  |
| --- |
| chgrp -R gitusers /path/to/repository.git          chmod -R u=rwX,g=rwXs,o= /path/to/repository.git |

Now just set up the SSH as usually, either using private keys (recommended) or password. Then you can clone the repository using the URI<ssh://server:22/path/to/repository.git>. If you already have a local repository that you want to import into the central repository, just add the remote *origin* with this URI (**Remote|Add** and then push.

#### [Modifying the History](http://www.syntevo.com/doc/display/SG/Modifying+the+History?src=contextnavpagetreemode)修改历史

This page gives an overview of the various ways in which you can use SmartGit to modify your repository's history.

**Warning**

It is generally a bad idea to modify the *remote* history (i.e. to modify commits you have already pushed to remote repositories), since this may cause a lot of trouble for other people working in them.

## Modifying the last commit

The simplest scenario is modifying the contents or the commit message of the last commit. SmartGit offers the following operations to do this:

* **Edit Last Commit Message**: You can edit the commit message of the last commit via **Local|Edit Last Commit Message**.
* **Amend**: When performing a commit (e.g. via **Local|Commit** or by pressing the **Commit** button on the main toolbar), you may toggle the option **Amend last commit instead of creating new one** on the commit dialog. With this option, the new changes will be added to the last commit, instead of being committed as a new commit.
* **Undo Last Commit**: With the menu entry **Local|Undo Last Commit** you can undo the last commit. By doing so, you won't lose any changes: The commit's changes will be restored in the Index and working tree so you can edit and possibly re-commit them.

If all you want is to edit the commit message of the last commit, **Edit Last Commit Message** is the simplest way to do it. If you want to alter the contents of the last commit, you can choose between **Amend** and **Undo Last Commit**. The former is typically used when you don't need to know what the exact contents of the last commit are and you just want to add some smaller changes to it. An example for this is when you want to fix a typo. On the other hand, if you want to alter the last commit in a substantial way (or discard it altogether), the **Undo Last Commit** option is the way to go.

## Merging the last two commits

Merging the last two commits into a single commit is straight-forward: Undo the last commit with **Local|Undo Last Commit**, then re-commit the changes, but with the option**Amend last commit instead of creating new one** checked on the Commit dialog.

## Splitting the last commit into two

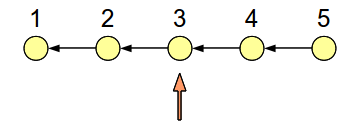
Doing the opposite of merging the last two commits, i.e. splitting the last commit into two, requires using the Index. SmartGit provides a specialized editor for it, called the Index Editor. How it is used is explained in another How-To: [Staging, unstaging and the Index Editor](http://www.syntevo.com/doc/display/SG/Staging%2C+unstaging+and+the+Index+Editor#Staging,unstagingandtheIndexEditor-workflows.index)

In essence, the Index allows you to commit only a subset of the changes in your working tree, instead of all the changes in it. So, in order to split the last commit A into two commits A1 and A2, do the following:

* Undo commit A with **Local|Undo Last Commit** from the main menu.
* Use the Index Editor to pick out (i.e. "stage") the changes to be included in commit A1. Commit these changes.
* Commit the remaining changes as commit A2.

## Modifying older commits

If you want to modify commits older than the last one, things get more complicated. It's not all that difficult though once you understand the general strategy. Let's take a look at an example: Suppose you have five commits, numbered 1 through 5, and the goal is to modify just the third commit, as shown below:

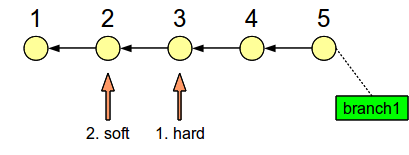


To accomplish this, we need a Git command called *reset*. This command comes in several flavors that all do slightly different things. In this case we need *reset hard* and *reset soft*, which have the following effects:

* **reset hard**: Moves the current branch pointer to the specified commit *and* resets the contents of the Index and of the working tree to that commit's contents.
* **reset soft**: Moves the current branch pointer to the specified commit, *but* leaves the contents of the Index and of the working tree as they are.

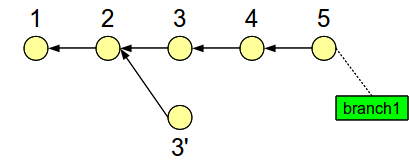
In Git, if you move a branch pointer elsewhere, you risk losing the commits the branch pointer was pointing at, since the commits might become unreachable. To prevent this, we'll have to create an additional branch pointer, in this case named *branch1*, which points at commit 5.

Now here's the trick: After creating *branch1* (without switching to it), we'll do a hard reset on commit 3, then a soft reset on commit 2. This is shown below:

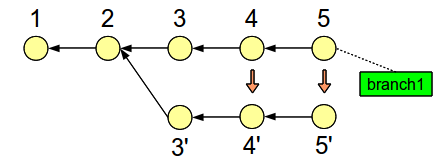


The result is that our branch pointer is now at commit 2, but our Index and working tree contain the changes of commit 3.

In this state we can now alter the contents of the Index and/or working tree in order to prepare a modified version of the original commit 3. Then, when we commit these changes, the history will branch off from commit 2 (since that's where our branch pointer was) to a new, modified commit 3':



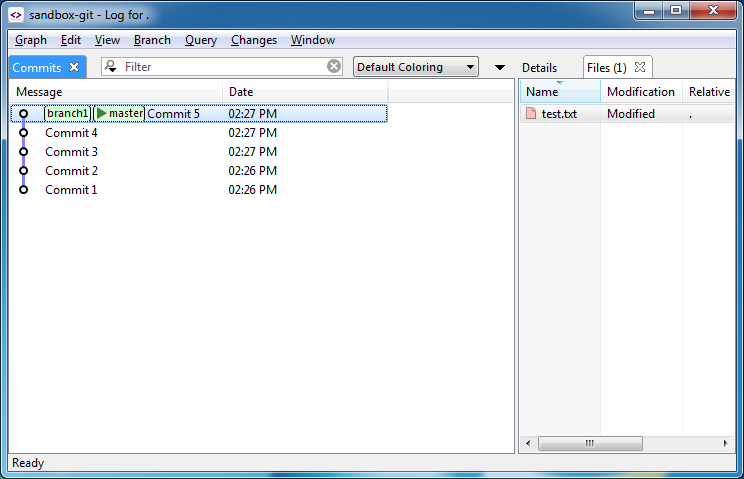
We're almost there. Now we need to somehow move the commits 4 and 5 over to the newly created branch:



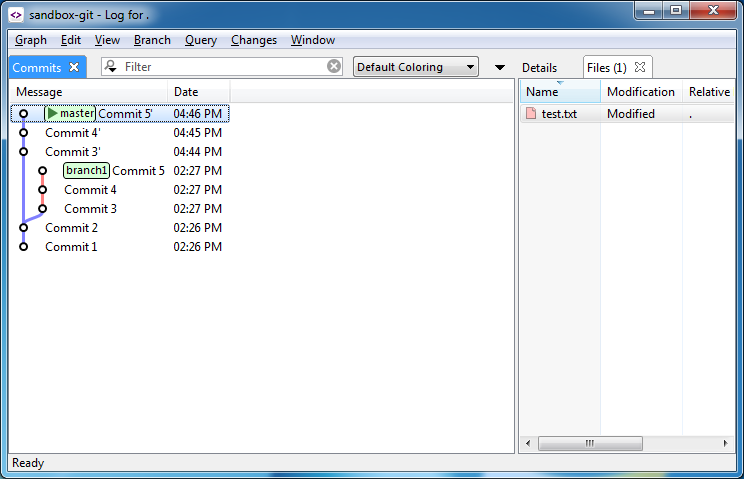
There are two ways to do this, either with cherry-picking or with rebasing. In both cases, you may need to resolve conflicts between your new changes made in commit 3' and the changes from commits 4 and 5. After that, the final step will be to remove *branch1* in order to discard the original history.

Now that you've seen the general idea, let's see how we can do this with SmartGit.

* First, open the repository with SmartGit. Select the project root in the Directories pane, then open the Log window, either by clicking on the **Log** button on the main toolbar, or by selecting **Query|Log** from the main menu.
* The first thing to do is to create a new branch on the last commit (without switching to the branch) in order to make sure we won't lose any commits while performing resets. To create the branch, select the last commit, then select **Add Branch** in the commit's context menu. On the dialog, enter the branch name and confirm by clicking on the **Add Branch** button. This is what the Log might look like afterwards:



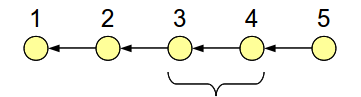
* Now, we need to do a hard reset on commit 3, and a soft reset on commit 2, in that order. After that, the *master* branch pointer will be at commit 2, and the working tree will contain the changes from commit 3.
* You may (but don't have to) alter the contents of the working tree. Then stage the changes and commit them in order to create an alternate commit 3'.
* Now we need to put commit 4 and commit 5 on top of commit 3'. There are two options here, rebase and cherry-pick.
* With rebase: Select **Branch|Rebase** from the main menu. Assuming you're still on the branch where commit 3' is, select the option **Selected commit to HEAD** and click**Next**. On the following dialog, select commit 4 and 5, then confirm the rebase by clicking on the **Rebase** button.
* With cherry-pick: Select **Branch|Merge** from the main menu. On the dialog, select commit 4 and 5. If they aren't shown, click on the arrow button on the top right to open a drop-down menu. In this menu, select **All branches**. Now select the option **Only the selected commits** on the dialog and confirm the operation by clicking on the**Merge** button.
* With both rebase and cherry-pick, you may encounter conflicts: There may be conflicts between the changes from commit 3' and the changes made in commit 4 and 5. If that happens, SmartGit will stop the rebase or cherry-pick and remain in a conflict state. For further information on how to proceed here, see: [How to resolve conflicts](http://www.syntevo.com/doc/display/SG/How+to+resolve+conflicts#Howtoresolveconflicts-workflows.resolve-conflicts)
* If all went well, the Log might now look like this:



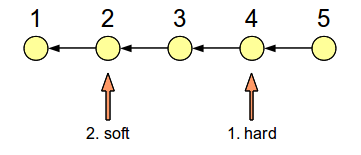
* Finally, remove *branch1* to discard the old history. There are two ways to do this: (1) Select **Branch|Branch Manager** from the main menu, then select *branch1* on the dialog, then click on the **Delete** button. (2) Open the Log window (**Query|Log**), select the commit to which *branch1* is attached, select **Delete** from the commit's context menu, then follow the dialog instructions.

## Merging and splitting older commits

So far, we've seen how to modify an older commit. A more general problem is modifying an entire *range* of older commits. For example, how can we merge the changes from commit 3 and 4 into a single commit? This is shown below:



For this problem, we can use a more general version of the hard-soft trick explained above: First, we'll do a hard reset on commit 4, then a soft reset on commit 2, as shown below:



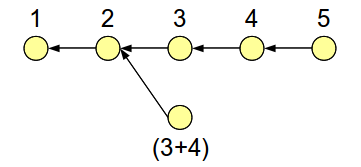
The result of this operation is twofold:

* The branch pointer is at commit 2.
* Our Index and working tree contain the changes that were made in commit 3 and 4.

At this point, you can alter the original history in any way you want. We'll discuss two possibilities, merging and splitting commits.

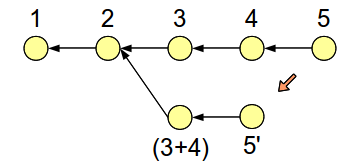
### Merging commits

If you perform a commit at this point, you'll end up with a new commit that branches off from commit 2 and contains the combined changes from commit 3 and 4. This is shown below:



It is also possible to alter the contents of the working tree before committing. In that case, you'd get a modified version of the combined changes from commit 3 and 4.

Finally, we need to move commit 5 over to our new, alternate history:



This can be done with cherry-picking or rebasing, as was explained above.

### Splitting commits

Recall that after the hard and soft reset, the working tree contains the combined changes from the original commits 3 and 4. That means we can use the Index Editor to split these combined changes into as many commits as we want. For example, we could split them up into three commits, where the last one contains changes from both commit 3 and 4.

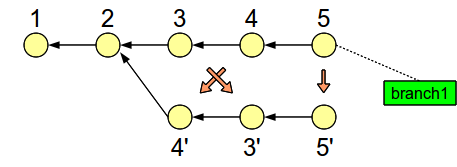
In any case, you would commit these "changed changes" one by one, and then append the remaining commits from the old history, in this case commit 5, to the altered history with cherry-picking or rebasing.

## Reordering commits

A special case of altering the history is to change the order of commits. This is a bit simpler than the history modifications described above. As an example, suppose we have five commits, numbered 1 through 5, and we want to swap commit 3 and commit 4. This can be done as follows:

* Put a branch pointer on commit 5 so we won't lose any commits while altering the history.
* Hard reset to commit 2. Effectively, this creates an alternate history that branches off from the original history at commit 2.
* Cherry-pick commit 4.
* Cherry-pick commit 3.
* Cherry-pick commit 5.
* Delete the branch pointer on commit 5 that you created in the first step. By doing so, you will discard the old history.

This procedure is illustrated below:



#### [Setting up submodules](http://www.syntevo.com/doc/display/SG/Setting+up+submodules?src=contextnavpagetreemode)

Git's submodules allow you to embed one Git repository within another. This page explains how to set up submodules using SmartGit.

## Adding Submodules

New submodules can be added to an existing Git repository as follows: In the Directories pane of the project window, select the folder into which the new submodule should be inserted. Then, from the program menu select **Remote|Submodule|Add**. In the wizard that shows up, specify a folder name for the new submodule and a URL pointing to the submodule's repository. On pressing the Finish button on the wizard's last page, SmartGit will start checking out the submodule's contents from its repository.

## Initializing and Updating Submodules

If you perform a Pull operation, e.g. by clicking on the Pull button in the main toolbar, a Pull dialog will be opened. On this dialog, you can find two submodule-related options:

* Update registered submodules
* And initialize new submodules

As indicated by their names, if these two options are enabled, SmartGit will automatically update and initialize submodules, respectively, when a Pull operation is performed. See the Git documentation on the commands git submodule update and git submodule init for more information about these two operations.

#### [Staging, unstaging and the Index Editor](http://www.syntevo.com/doc/display/SG/Staging%2C+unstaging+and+the+Index+Editor?src=contextnavpagetreemode)

Git has a special place called the *Index*, which allows you to select only a subset of the current changes in your working tree for committing, rather than all of your changes. A typical scenario where this is useful is when you're working on a larger feature involving the files A, B and C, and while working on these files, you discover several unrelated typos, which you want to fix and commit separately.

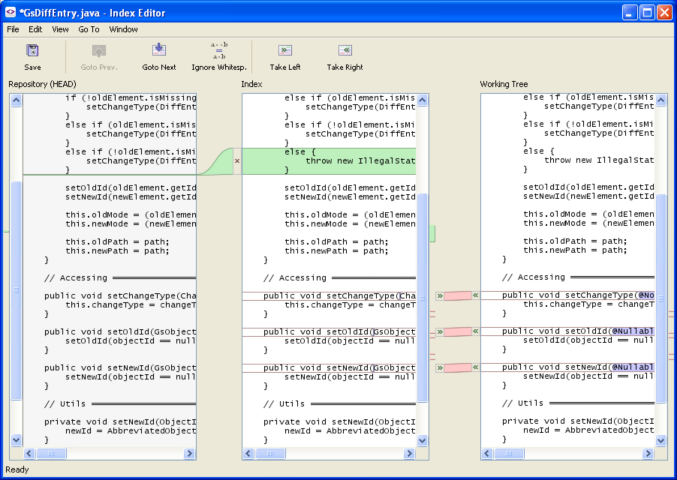
If the typos are in files other than A, B and C, the Index could be used, but is not really needed. For example, if the typos were in the files X and Y, you could simply select these two files in the Files pane and click on the **Commit** button in the main toolbar. This is due to the fact that in SmartGit what will be committed is determined by the current selection. If you had selected the project root in the Directories pane instead before committing, all changed files in the working tree would have been chosen for the commit.

Now, what if some of those typos were in files related to the feature you've been working on? In this case, Git's Index is needed, because it allows you to not only pick out certain files for the next commit, but also certain chunks *within* files. In the following, let's assume your typos were in the files C and D. Therefore, we need to put the entire file D, and all the relevant chunks in file C in the Index.

Putting something in the Index is called *staging*, and the opposite, i.e. moving something out of the Index, is called *unstaging*. Here's how you stage and unstage an entire file (in this case file D) in SmartGit:

* **Staging:** On the Files pane, select one or more files to stage. Then either click on the **Stage** button on the main toolbar, or select **Local|Stage** in the main menu.
* **Unstaging:** On the Files pane, select one or more files to unstage. Then either click on the **Unstage** button on the main toolbar, or select **Local|Unstage** in the main menu.

For the file C, from which we want to stage only parts rather than everything, we need SmartGit's so-called *Index Editor*. To open it, select the file in the Files pane, then select either **Index Editor** in the file's context menu, or select **Local|Index Editor** from the main menu. Here's what the Index Editor looks like:



As you can see, the Index Editor consists of three separate text editors, which show the state of a particular file in the HEAD of the repository, in the Index, and in the working tree, respectively. Only the contents of the middle and the right editor can be modified. You can alter the contents of the Index either by editing the contents of the middle editor, or by moving chunks between the three editors. To do so, either click on the arrow and 'x' buttons between the editors, or click on the **Take Left** and **Take Right** buttons on the toolbar. After you're done, save your changes and close the Index Editor.

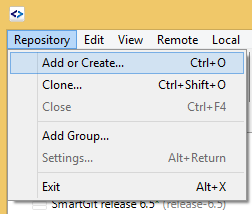
Note that when you do a commit, you may have to choose at the top of the **Commit** dialog between committing the staged changes and committing the local changes. The former includes only the contents of the Index, while the latter includes all of your changes in the working tree, including the contents of the Index.

## [Tour](http://www.syntevo.com/doc/display/SG/Tour?src=contextnavpagetreemode)入门

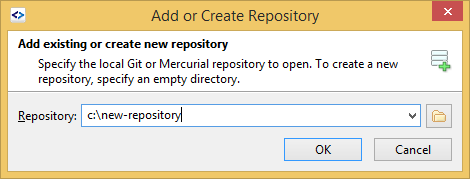
This page gives an overview of the basic usage of SmartGit. In addition to explaining the basics, we will focus on those features and peculiarities of the program that may be difficult to discover for first-time users. For more comprehensive documentation, please see the [manual](http://134.119.46.64:8090/doc/Documentation.html).

## Create Repositories创建代码库

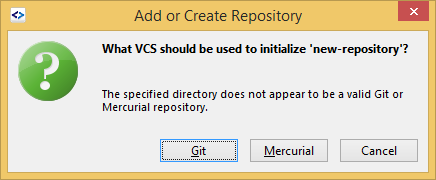
If you have no Git or Mercurial repository available, but want to create a new local one (just for you), click Repository|Add or Create.



Now enter the directory of the new repository. This either can be an already existing one, but you also can enter a new one. In the latter case it will be create automatically.



Clicking OK will check whether the entered directory is already a repository (contains either a .git or .hg subdirectory). If it is not already an repository, SmartGit will ask you whether to initialize it. If both, Git and Mercurial executables, are configured in the preferences, you will see following choice (otherwise a simpler confirmation dialog):



Now you've created a new Git or Mercurial repository. You can add files (outside SmartGit) to the entered directory and later commit those in SmartGit.

You also can use these steps to add already existing repositories to SmartGit. Just enter their path as for the new repository. An alternative to open an existing repository is to drag it from your file manager into SmartGit's Repositories view.

## Repositories代码库

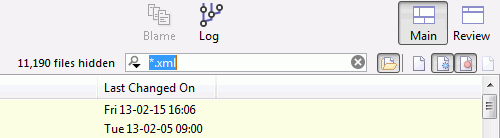
To open a repository, either double-click it, select one or more, right-click and select Open. To open a repository in a new window, right-click it and select Open in New Window.

Each repository can have a name for better description. To change that, right-click the repository and select Rename.

If you have a large number of repositories, you may organize belonging repositories into groups. Just create a group using Repository|Add Group and drag and drop the repositories into this group.

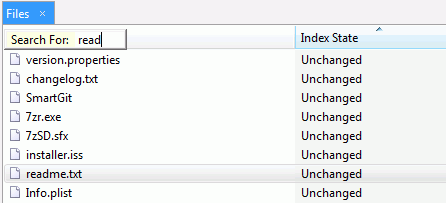
## User Interface用户界面

After opening a repository, SmartGit reads its directory structure. Expanding the repository node will show this directory structure. The selection in the Repositories view determines what is shown in the Files view. Use the small toolbar buttons above the Files view (or the View menu) to **filter the shown files** by their state, e.g. untracked or unmodified files. To filter by name, use the File Filter input field.



By default, files are listed recursively, so not only files from the selected directory, but also from subdirectories are shown. Directories marked as 'ignored' will not be scanned and hence the files in them will not be shown. New ("untracked"), modified or deleted ("missing") files will show up with special icons on the Files view. In the Directories view**red arrows indicate where committable files are located** inside the directory structure. If committable (e.g. untracked) files are hidden because of filtering, the file table will show a light pink background to remind you about the filtering, so you won't forget to commit them. If untracked files you don't want to commit filling up your repository, consider to ignore them (Local|Ignore).

To **quickly select a file**, simply start typing its name. As for the filtering, you may use [glob pattern](http://en.wikipedia.org/wiki/Glob_(programming)). Use the up and down cursor keys to select the next or preview matching file. This feature is also available in other parts of SmartGit.



You can use drag and drop to rearrange the layout of the views. There are two layouts, Main and Review, independent from each other configurable. To hide a view, click the small cross button in its title. To **restore a closed or activate a view**, use the corresponding menu item from the Window menu.

## Review Local Changes审查本地修改

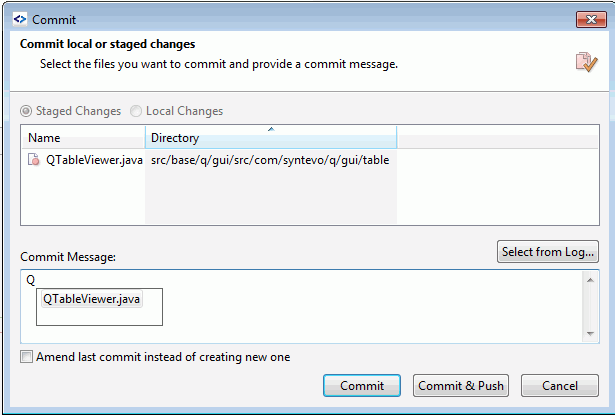
SmartGit offers different ways to review local changes:

* The Changes view shows the changes of the selected file. Use the small arrow buttons in its toolbar to navigate from change to change over all shown committable (text) files.
* To open an **editable Compare**window click the Changes toolbar button or double-click a file in the Commit dialog.
* To commit only parts of modified files, use the Index Editor for that file. This allows to **freely edit the file's Git Index content** as well as the working tree content. The Git Index content can be committed with the next commit independent of the working tree content.

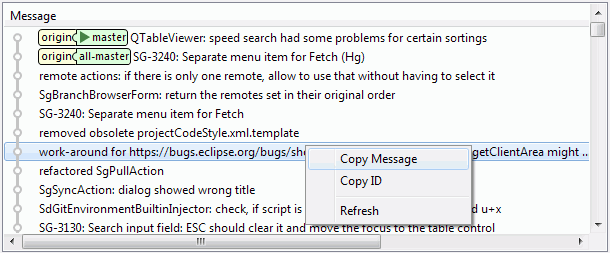
## Commit提交

Window|Commit Message will show a view to **prepare the commit message** while reviewing local changes. When invoking the Commit command, this prepared message will be used to prefill the occurring commit dialog. The initial file selection on the commit dialog depends on your current selection on the project window. For example, if you select the root directory of your repository before opening the commit dialog, all changed files in the repository will be selected for committing. If you select a particular changed file on the Files view, only this file will be selected for the commit. To commit staged changes, either select a file with index changes or select the root directory before invoking theCommit button.

To **ease entering of file names** in the Commit dialog, use Ctrl+Space shortcut to get a list of all files selected for the commit.



To reuse the message of an existing commit, click the Select from Log button. As for all Logs (e.g. Merge, Cherry-Pick, too), **right-click to copy the commit ID** (or message).



## Local Changes本地更改

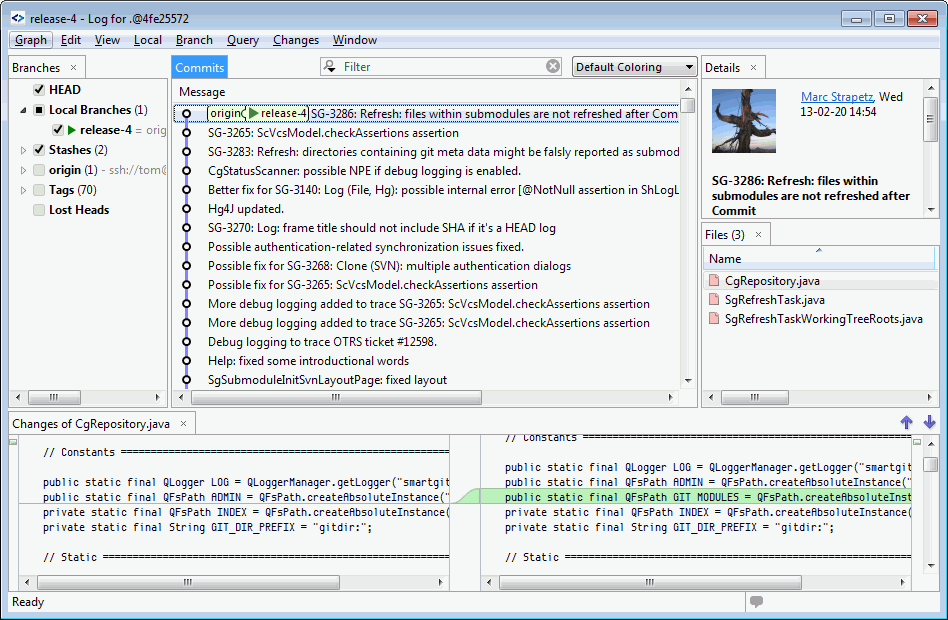
For those cases where an error does make it through the review phase and ends up being committed, SmartGit provides several options in addition to the standard Git procedures to fix commits:

* You can **undo the last commit** by selecting Local|Undo Last Commit. This will restore the committed changes in the Index.
* You can "add" further changes to the previous commit by marking the Amend option on the commit dialog.
* To **change the previous commit's message** (even if there are local changes), use Local|Edit Last Commit Message.

See [this manual section](http://134.119.46.64:8090/doc/Documentation.html?page=commands-wt) for further information.

## History

To **show the commit history**, select the repository root (for the entire history), file or directory and click the Log button from the main toolbar.



In the Branches view, select the checkboxes before the branches, tags or stashes you are interested in. Their commit history structure then will be displayed in the Commitsview. When selecting a checkbox before a category, e.g. Local Branches, all items will be shown, otherwise only the selected ones - even if you create new items. **Double-click a local branch to switch**. Double-click a remote branch to create a corresponding local branch. The Lost Heads will show all commit heads which are not accessible from a tag or branch.

Tracking and tracked branches will be shown as **jigsaw pieces**. If both are pointing at the same commit, the jigsaw pieces are shown connected.

The Files view shows all changed files of the selected commit. If **two commits are selected**, the **difference between both commits** will be shown. Select a file entry to see its detailed changes in the Changes view. By right-clicking a file you can **save the file state to disk** or compare it with a file on your disk defaulting to the matching one in the working tree.

## Pulling and Pushing拉取和推送

SmartGit有很多拉取和推送到远端代码库的方法，最明显的就是工具条上的拉取和推送按钮。由于远端代码库的变动，直接执行可能会出错，所有执行前要先同步一下

如果不想推送所有的更改，使用下拉菜单中的Pushable Commits

推送指定tag或分支使用下拉菜单中的Branches view.