



# Git & TortoiseGit - Quick Guide

Écrit par : Marc Chevaldonné, IUT Informatique, Université d'Auvergne Clermont1

12 août 2010  
Version 1.0

## Sommaire

<b>Introduction</b>	<b>3</b>
Contexte	3
Documents de références	3
Table des modifications	3
<b>Les systèmes de contrôle de version</b>	<b>4</b>
Méthodes locales	4
Systèmes de contrôle de version centralisés (e.g. Subversion et CVS)	5
Principe	5
Problème du partage de fichiers	6
Le repository	8
Systèmes de contrôle de version distribués (e.g. Git)	11
Principes généraux	11
Principes de Git	12
<b>TortoiseGit, installation et configuration</b>	<b>15</b>
Installation de Git et de TortoiseGit	15
Création d'un repository sur Hina	15
Configuration de TortoiseGit	16
<b>Utilisation de Git et TortoiseGit</b>	<b>17</b>
Cloner un repository existant en local	17
Utilisation du repository local	18
Transférer les modifications des fichiers dans le repository local : (add +) commit	18
Ajouter de nouveaux fichiers au repository local : add + commit	20
Supprimer des fichiers du repository local : rm	22
Déplacer ou renommer un fichier dans le repository local : mv + commit	24
Historique du repository local	25
Travailler avec les branches	28
Principe	28
Gestion des branches	31
Créer une branche	32
Switch/Checkout de branche ou de version	33
Fusionner des branches	33
Gestion des conflits	34
Synchronisation avec un repository distant	42

Soumettre son travail sur le repository distant	43
Récupérer le travail des collaborateurs	45
Gestion des conflits lors de l'utilisation d'un repository distant	47
<b>Autres fonctionnalités</b>	<b>49</b>
Tag	49
Rebase	49
Pull	49
<b>Méthodes d'utilisation collaboratives de Git</b>	<b>50</b>
Workflow centralisé	50
Quelques règles de bonne conduite	50

## 1. Introduction

### 1.1. Contexte

Ce document a été écrit pour les élèves de GI et SI (2ème année de l'IUT d'Informatique de Clermont-Ferrand, Université d'Auvergne). Son but est de décrire succinctement le principe des systèmes de contrôle de version, les fonctionnalités de Git et les outils de base pour une utilisation via TortoiseGit sous Windows.

### 1.2. Documents de références

Pour plus d'informations, le lecteur peut se documenter en lisant les documentations suivantes :

Ref	Title	Version	Lien
[1]	Pro Git	1	<a href="http://progit.org/book/">http://progit.org/book/</a>
[2]	Git - the fast version control system	1.7.2.1	<a href="http://git-scm.com/">http://git-scm.com/</a>
[3]	TortoiseGit	1.5.2.0	<a href="http://code.google.com/p/tortoisegit/">http://code.google.com/p/tortoisegit/</a>
[4]	Version Control with Subversion	1.5	<a href="http://svnbook.red-bean.com/">http://svnbook.red-bean.com/</a>

### 1.3. Table des modifications

Author	Modification	Version	Date
Marc Chevaldonné	Première version	1.0	12 août 2010

## 2. Les systèmes de contrôle de version

Un système de contrôle de version permet la gestion de dossiers et de fichiers ainsi que leurs modifications au cours du temps. Cela permet de récupérer des versions anciennes de documents ou d'examiner l'historique des modifications apportées. On parle aussi de «time machine».

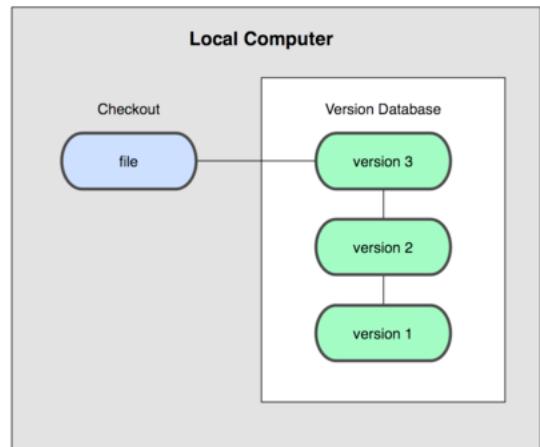
Un système de contrôle de version peut-être très généralement être utilisé en réseau, ce qui permet son utilisation par des personnes distantes. Il est donc possible à un groupe de personnes de modifier et de gérer un même ensemble de données à distance et de manière collaborative. En effet, un tel outil permet une production en parallèle et non linéaire, sans risque de confusion ou de pertes de données grâce au versionnage.

Les documents versionnés sont généralement du texte, et en particulier du code source, mais il peut s'agir de n'importe quel type de documents.

Il existe de nombreux systèmes de contrôle de version. Ils ne sont pas tous basés sur les mêmes principes. Les plus connus sont CVS et Subversion. Les parties suivantes présentent quelques exemples des outils les plus courants et les plus représentatifs.

### 2.1. Méthodes locales

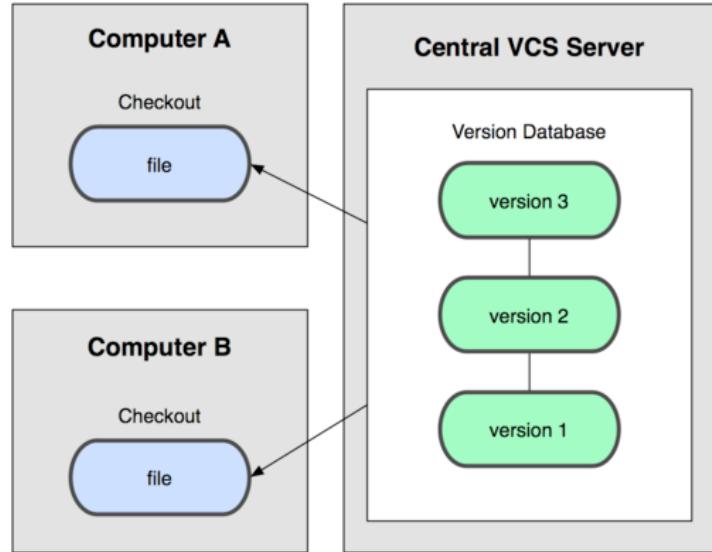
Lorsque vous travaillez sur un TP, un rapport ou une présentation, vous êtes amenés à sauvegarder votre travail. Parfois, vous souhaitez également faire des essais, sans écraser le travail déjà réalisé. Dans ce cas, vous faites des copies de votre travail, avec différentes versions. Si vous êtes un peu organisé, vous allez donner un titre «logique» à vos fichiers ou dossiers afin de pouvoir retrouver facilement une ancienne version ou les nouvelles modifications. Vous choisissez par exemple de renommez les fichiers avec un numéro de version, ou en intégrant la date et l'heure dans le nom du fichier. Beaucoup d'autres personnes sont passées par là avant vous. C'est la raison pour laquelle, certains programmeurs ont commencé à réaliser des systèmes de contrôle de version locaux, qui n'enregistraient que les différences d'une version à l'autre, dans une base de données locale, permettant de revenir en arrière à n'importe quelle version du fichier. Une des fonctionnalités de Time Machine sur MacOSX utilise ce principe.



Système de contrôle de version local (tiré de [1])

## 2.2. Systèmes de contrôle de version centralisés (e.g. Subversion et CVS)

La phase suivante dans les systèmes de contrôle de version est la collaboration. Non seulement il est nécessaire de garder une trace des modifications dans le temps, mais il est également nécessaire de travailler à plusieurs, de savoir qui a fait la dernière modification, qui a introduit le bug, soumettre son travail à ses collaborateurs (qu'on ne rencontre parfois jamais)... Les systèmes de contrôle de version centralisés ont eu pour objectifs de résoudre ces problèmes. Les plus connus sont CVS et Subversion.



Système de contrôle de version centralisé (tiré de [1])

### 2.2.1. Principe

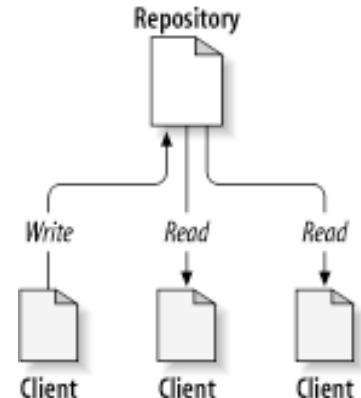
Un serveur centralise tous les fichiers constituant un projet dans ce qu'on appelle la base centrale : le «repository».

Un utilisateur a des droits d'écriture et/ou de lecture sur les fichiers stockés.

Un utilisateur rapatrie sur son poste de travail une version (généralement la dernière) des fichiers et travaille ainsi toujours sur une version locale du projet.

Avant de pouvoir modifier un fichier du projet, l'utilisateur doit l'extraire, c'est-à-dire qu'il avertit le serveur qu'il en prend possession.

Une fois qu'il a terminé la modification, l'utilisateur archive le/les fichiers. Le fichier est renvoyé vers le serveur. Ce dernier fusionne les modifications effectuées par l'utilisateur à sa version courante du fichier.

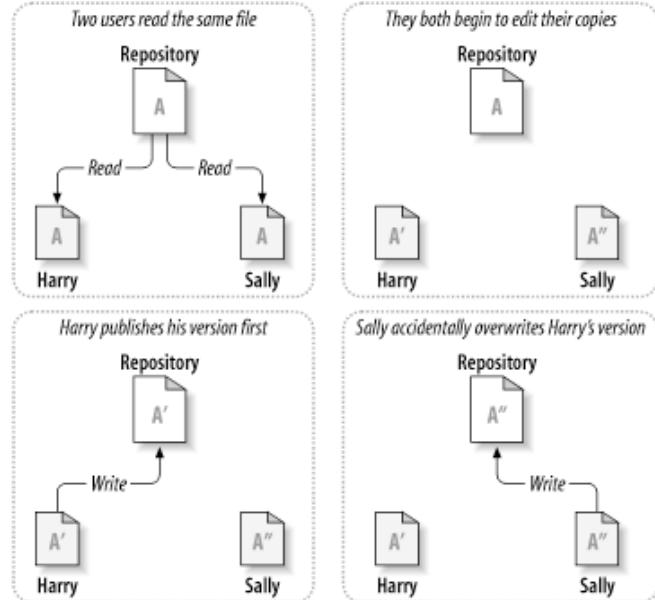


Principe du serveur centralisé  
(tiré de [4])

### 2.2.2. Problème du partage de fichiers

Un des problèmes majeurs des systèmes de gestion de versions est le partage de fichiers. Comment permettre à plusieurs utilisateurs de modifier le même fichier ?

- Harry et Sally prennent tous les deux une copie en local d'un fichier versionné sur le serveur.
- Harry et Sally le modifient de manière différente chacun de leur côté.
- Harry soumet ses modifications sur le serveur.
- Sally soumet ses modifications sur le serveur après Harry et efface toutes les modifications d'Harry accidentellement.

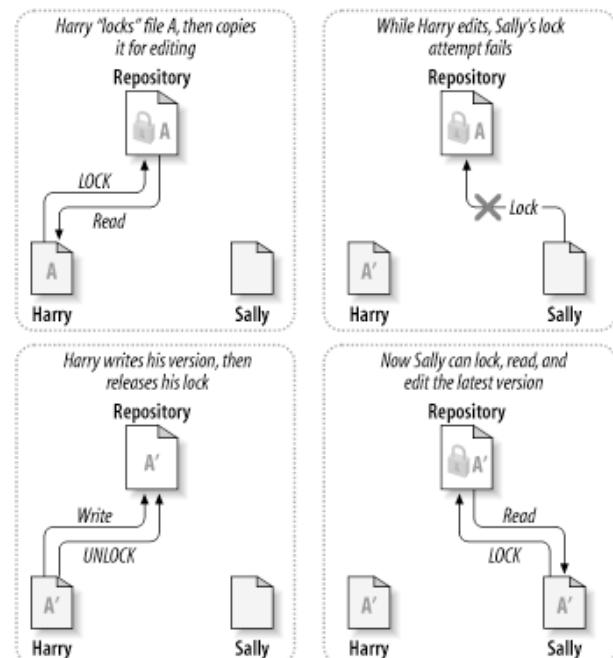


Problème du partage de fichiers  
(tiré de [4])

### Solution 1 : Lock-Modify-Unlock

Avant Subversion, la solution unique consistait à verrouiller / modifier / déverrouiller (CVS).

- Harry récupère une copie du fichier sur le serveur et le verrouille.
- Sally essaye de récupérer une version du fichier pour le modifier, mais le fichier est verrouillé.
- Harry soumet ses modifications et déverrouille le fichier.
- Sally récupère une version du fichier maintenant déverrouillé et peut le modifier. Le fichier sur le serveur est verrouillé par Sally.

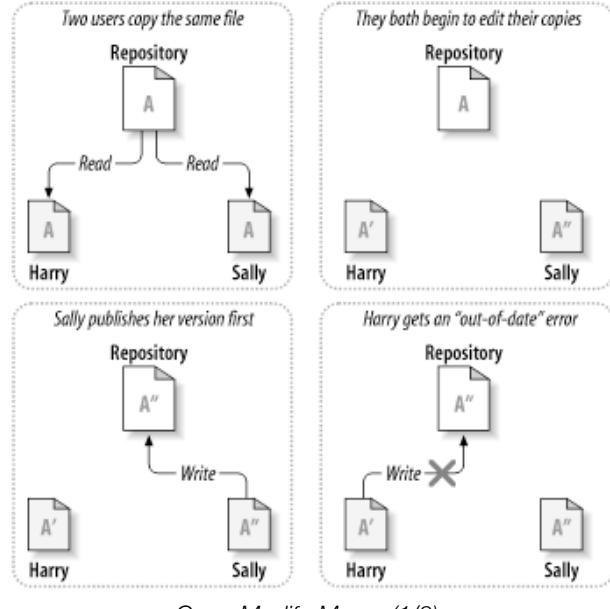


Lock-Modify-Unlock  
(tiré de [4])

### Solution 2 : Copy-Modify-Merge

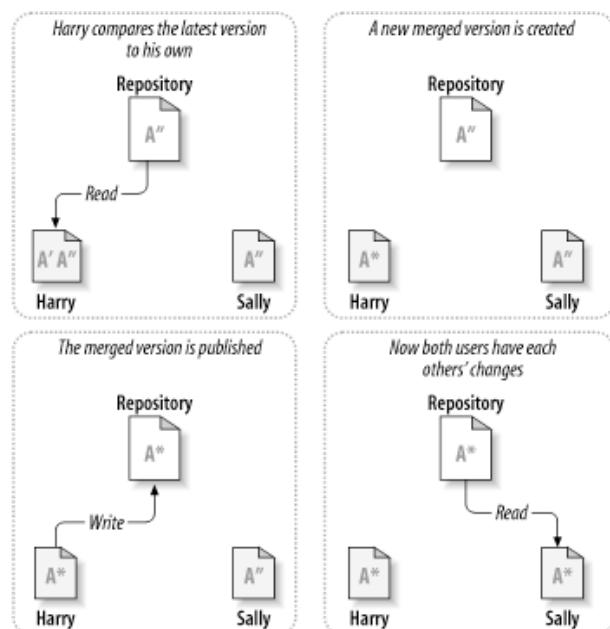
Subversion a proposé une nouvelle solution qui consiste à copier en local une version du fichier, de la modifier, et de fusionner uniquement les différences avec la version du repository.

- Harry et Sally récupèrent une copie du fichier sur le serveur.
- Harry et Sally modifient chacun de leur côté et de manière différente le fichier.
- Sally soumet ses modifications et le fichier sur le repository est donc modifié.
- Harry veut soumettre ses modifications à son tour. Deux solutions : soit il n'y a aucun conflit entre les modifications d'Harry et les modifications de Sally, c'est-à-dire qu'ils ont modifié des parties très distinctes d'un même fichier (par exemple deux classes ou deux méthodes différentes), et dans ce cas, Subversion fusionne les modifications d'Harry au fichier modifié de Sally ; soit les modifications de Sally et de Harry recoupent les mêmes parties et Subversion ne sait donc pas faire la fusion. Dans ce dernier cas, Subversion empêche alors Harry de faire son commit car sa copie n'est pas à jour.
- Harry récupère alors la dernière version sur le repository et la compare avec sa version modifiée.
- Une nouvelle version qui fusionne les modifications d'Harry et la dernière version à jour du repository est créée, avec l'aide d'Harry.



Copy-Modify-Merge (1/2)

(tiré de [4])



Copy-Modify-Merge (2/2)

(tiré de [4])

- La version fusionnée (merged) devient la nouvelle dernière version du repository.
- Sally met à jour sa copie et les deux utilisateurs ont les modifications effectuées par l'un et l'autre des développeurs.

Subversion permet d'utiliser les deux solutions, même si la solution 2 est préférable (surtout quand le nombre de développeurs devient important). De plus, les développeurs ont généralement des tâches bien distinctes et les conflits sont très rares.

Depuis, d'autres systèmes de contrôle de version ont adopté cette méthode (notamment CVS).

### 2.2.3. Le repository

Dans cette partie, nous expliquerons les principes du repository, de l'arbre des révisions et de l'organisation du repository en nous basant sur l'exemple du populaire Subversion.

Le repository est la base centrale où sont stockés les fichiers et les révisions. Un repository peut-être local ou distant, non sécurisé ou sécurisé. On accède à un repository via un URL.

Le tableau ci-dessous présente les différentes méthodes d'accès aux repositories avec Subversion.

Schema	Access method
<code>file:///</code>	accès direct sur un disque (local)
<code>http://</code>	accès via le protocole WebDAV et un serveur Apache
<code>https://</code>	comme le précédent mais avec des données cryptées (SSL)
<code>svn://</code>	accès via un protocole perso à un serveur <code>svnservve</code>
<code>svn+ssh://</code>	comme le précédent mais à travers un tunnel SSH

Dans le cas d'accès distant, l'administrateur du repository peut donner des droits d'accès en lecture / écriture à tout le repository ou seulement une partie du repository à des utilisateurs. Il peut par exemple donner des droits en lecture anonyme (pas besoin de se logger) et en écriture à des utilisateurs enregistrés avec mot de passe.

### L'arbre des révisions

À chaque fois qu'un commit est réalisé par un utilisateur, Subversion crée un nouvel état de l'arborescence des dossiers et fichiers du projet versionné. Ce nouvel état est appelé une révision.

À chaque révision, Subversion attribue un nombre entier unique, supérieur à celui de la révision précédente. La première révision est la numéro 0, et ne contient rien.

À chaque révision, Subversion n'enregistre que les différences avec la révision précédente. Pour récupérer une révision donnée, Subversion parcourt l'arbre des révisions et modifie les fichiers au fur et à mesure avec les différences enregistrées pour recréer l'état correspondant à cette révision.

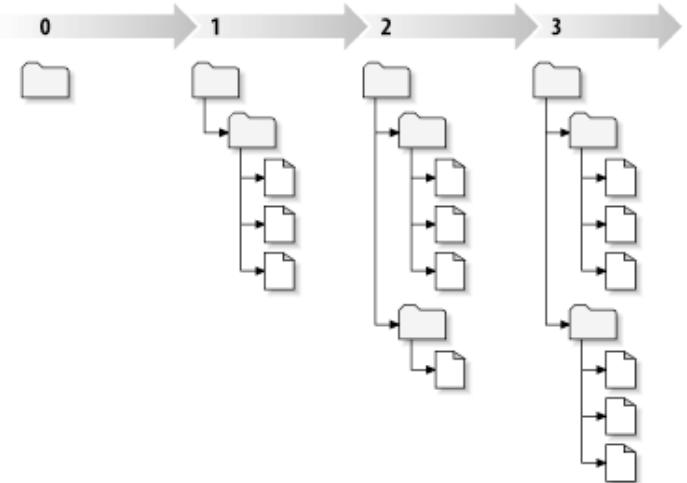
Les révisions sont donc stockées sous forme d'arbre.

### Organisation du repository : trunk-branches-tags

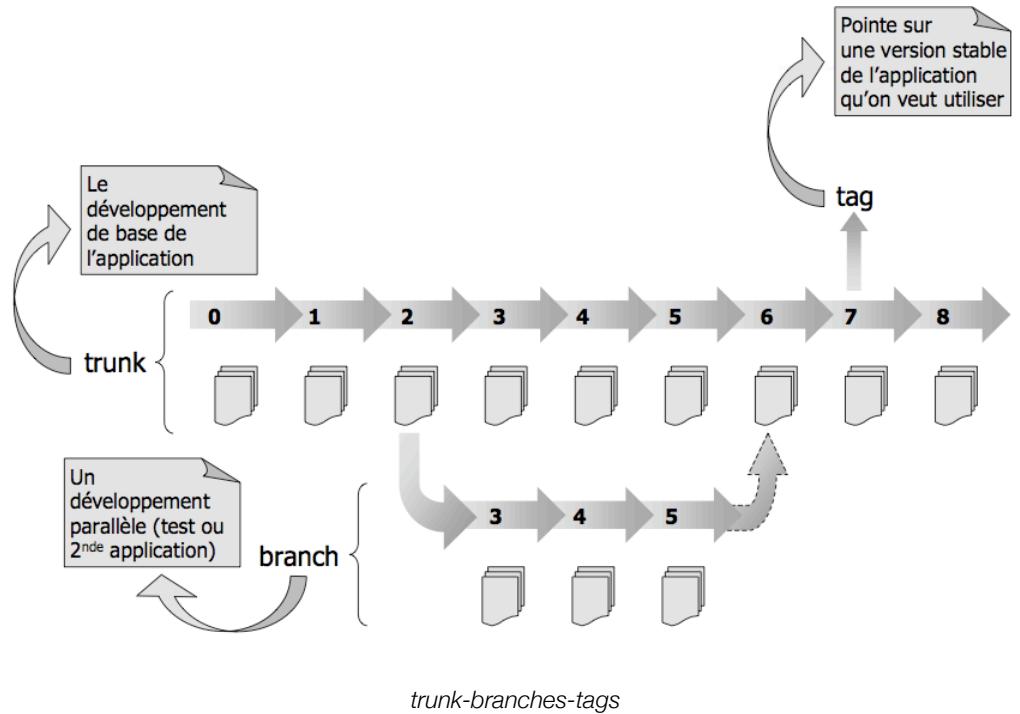
Subversion vous autorise à organiser votre repository comme vous l'entendez. Il existe de nombreuses bonnes organisations (mais encore plus de mauvaises...). Toutefois, il est fortement conseillé de suivre l'exemple du trunk - branches - tags (surtout pour les débutants).

Considérons un seul projet versionné via Subversion.

- Le dossier tronc (trunk) contient la ligne directrice de votre développement.
- Le dossier branches contient des copies du développement (une par branche). Les branches contiennent le même historique que le tronc au moment de la création de la branche. Ceci permet d'éviter d'avoir à refaire plusieurs fois la même chose pour des projets légèrement différents par exemple. Imaginons que nous souhaitons faire plusieurs versions d'un jeu : une en local et une autre pour le web. Nous développons d'abord le cœur du jeu qui sera commun à toutes les versions. Puis pour chaque version, nous créons une branche qui profitera de la partie commune déjà développée et de son historique et ajoutera ses propres particularités sans polluer le développement commun. Une branche peut également servir à faire un test de développement. Si ce test est approuvé, alors la branche peut être réintégrée dans le tronc, sinon, elle peut être coupée.



*L'arbre des révisions  
(tiré de [4])*



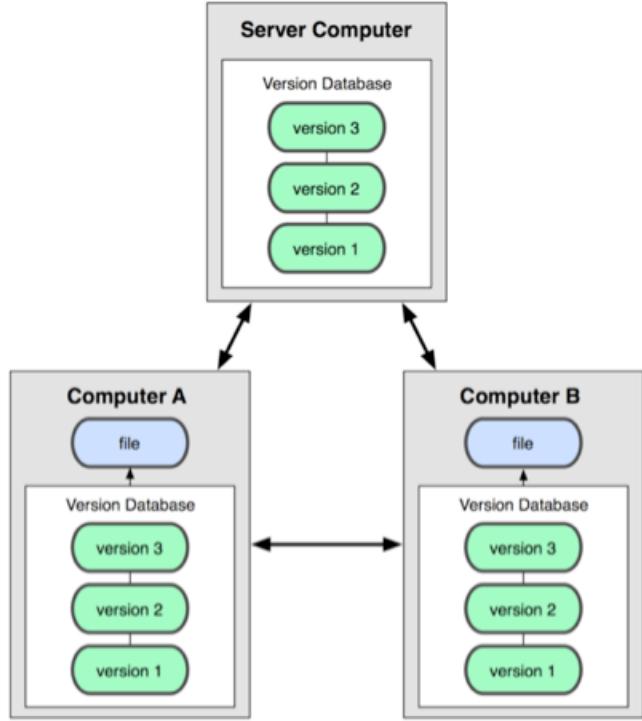
- Le dossier tags contient des «pointeurs» sur des copies à un instant donné. Un tag représente une espèce d'étiquette sur une version particulière de votre développement. Par exemple, la révision X fonctionne parfaitement, et si ce n'est pas la version finale, c'est en tout cas une très bonne version bêta. On peut alors la tagger, ce qui permettra de faire une première release ou des démos, sans empêcher l'avancement du projet. De même, on peut tagger ainsi la version 1.0, 1.1, 1.2 ... de notre logiciel. En pratique, le tag fonctionne exactement comme une branche. Subversion ne fait pas de différences pour des raisons de flexibilité. En effet, imaginez que vous taggez une version qui fonctionne parfaitement... enfin presque... après avoir taggé, vous découvrez un horrible bug ! Vous pouvez faire un commit sur un tag (c'est rare, mais ça se fait).

## 2.3. Systèmes de contrôle de version distribués (e.g. Git)

### 2.3.1. Principes généraux

Les systèmes de contrôle de version centralisés sont aujourd’hui très populaires et très utilisés (en particulier Subversion). Néanmoins, malgré leurs nombreux avantages, ils présentent quelques inconvénients :

- si le serveur est inaccessible pendant un certain temps, aucun des collaborateurs ne peut soumettre de modifications. Chaque collaborateur peut toujours travailler en local sur sa version, mais il ne peut pas créer de versions de son travail.
- si le serveur décède (et à condition que les sauvegardes du serveur ne soient pas faites ou pas à jour), tout l'historique du repository est perdu.



*Système de contrôle de version distribués  
(tiré de [1])*

Les systèmes de contrôle de version distribués (comme Git ou Mercurial) ont pour objectifs de trouver une solution à ces problèmes.

- ✓ Dans un système centralisé, chaque collaborateur récupère une version du projet à un instant donné (soit la dernière, soit une version plus ancienne). Il ne possède que quelques «photos» instantanées du repository. Dans le cas d'un système distribué, chaque collaborateur récupère la totalité du repository en local. De cette manière, si le serveur meurt, on peut retrouver le repository sur n'importe quelle machine utilisée par un collaborateur. Il possède également des versions locales du projet (une ou plusieurs, souvent la dernière version plus quelques branches).
- ✓ De plus, chaque collaborateur peut créer des versions du projet en local avant de les envoyer sur le serveur centralisé, c'est-à-dire qu'il peut continuer à versionner son travail tout en étant «offline».

Pour le reste, les systèmes de contrôle de version distribués fonctionnent grossièrement comme les systèmes de contrôle de version centralisés.

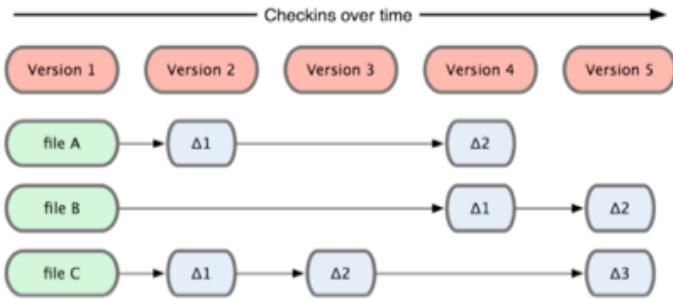
Git est un système de contrôle de version distribué créé en 2005 pour permettre la maintenance du noyau Linux. Il est depuis utilisé pour le développement et la maintenance de nombreux autres projets.

### 2.3.2. Principes de Git

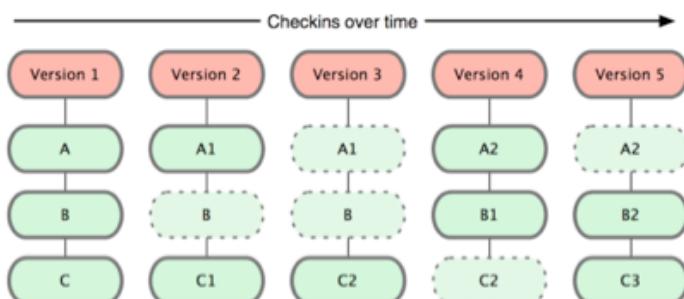
Cette partie a pour but de présenter succinctement les principes de Git. La compréhension du fonctionnement des [systèmes de contrôle de version centralisés](#) n'est pas indispensable, mais peut aider. Attention toutefois si vous êtes déjà un adepte de Subversion, Git utilise de nombreux termes similaires ayant parfois un autre sens.

La liste suivante introduit les concepts majeurs de Git.

- ➔ **Stockage de fichiers entiers vs. stockage de modifications** : les systèmes de contrôle de version centralisés stocke pour chaque fichier modifiés, les modifications dans le temps. Par exemple, les fichiers de la version 5 dans l'exemple ci-contre, seraient obtenus de la manière suivante :
- file A = file A (v1) + Δ1 (v2) + Δ2 (v4)  
 file B = file B (v1) + Δ1 (v4) + Δ2 (v5)
- Git stocke les fichiers modifiés complets, et pas seulement les modifications (seulement s'ils ont été modifiés). Ainsi pour obtenir la version 5, Git prend directement A2, B2 et C3.



*Stockage des modifications dans le temps des systèmes de contrôle de version centralisés (tiré de [1])*



*Stockage des fichiers complets dans le temps de Git  
(tiré de [1])*

- ➔ **Beaucoup d'opérations sont locales** : en effet, toutes les versions que vous effectuez sur votre travail, ainsi que des branches de tests peuvent être effectuées en local. La connection au serveur centralisé ne se fait que lorsque l'intégralité d'une phase de travail effectué est soumise au reste des collaborateurs, ou lorsqu'on souhaite récupérer le travail des autres collaborateurs. Ceci permet notamment de travailler «offline» tout en continuant à bénéficier de l'historique et du versionnage de Git.

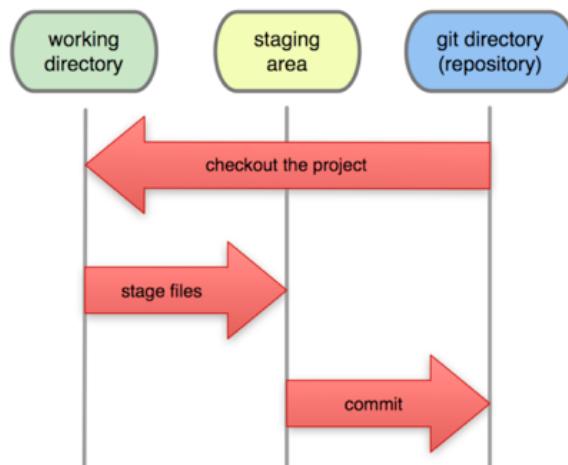
⇒ **Les 3 états des fichiers en local** : vos fichiers peuvent être dans trois états à travers Git au niveau local :

- **committed** : les données des fichiers «committed» sont enregistrées dans le repository local de Git. En d'autres termes, ce sont les fichiers sur lesquels vous avez terminé de travailler (ou simplement terminé une phase importante de votre travail) et dont les modifications viennent d'être sauvegardées dans le repository local de Git par vos soins.
- **modified** : les fichiers modifiés comportent des modifications par rapport aux fichiers correspondants dans le repository local de Git, mais ces modifications ne sont pas (encore) enregistrées dans le repository local. En d'autres termes, ce sont les fichiers sur lesquels vous êtes en train de travailler.
- **staged** : les fichiers «staged» sont des fichiers modifiés qui seront sauvegardés dans le repository local de Git lors du prochain commit. Ce sont donc des fichiers qui ont été marqués par vos soins par une étiquette «to be committed».

⇒ **Les 4 opérations avec les fichiers du repository Git en local** :

- **checkout** : permet de récupérer la dernière version (ou une version antérieure) des fichiers du repository local de Git (Git directory) dans votre dossier de travail local (working directory).
- **modification** des fichiers en local dans votre dossier de travail (working directory).
- **stage** : au moment où un fichier est marqué comme «to be committed» par vos soins, il est envoyé de votre dossier de travail local (working directory) vers la «staging area». Le fichier tel qu'il est au moment où vous l'envoyez dans cette aire sera enregistré dans le repository local de Git au prochain commit.
- **commit** : lorsque vous faites un commit, tous les fichiers de la staging area sont enregistrés dans le repository local de Git.

### Local Operations

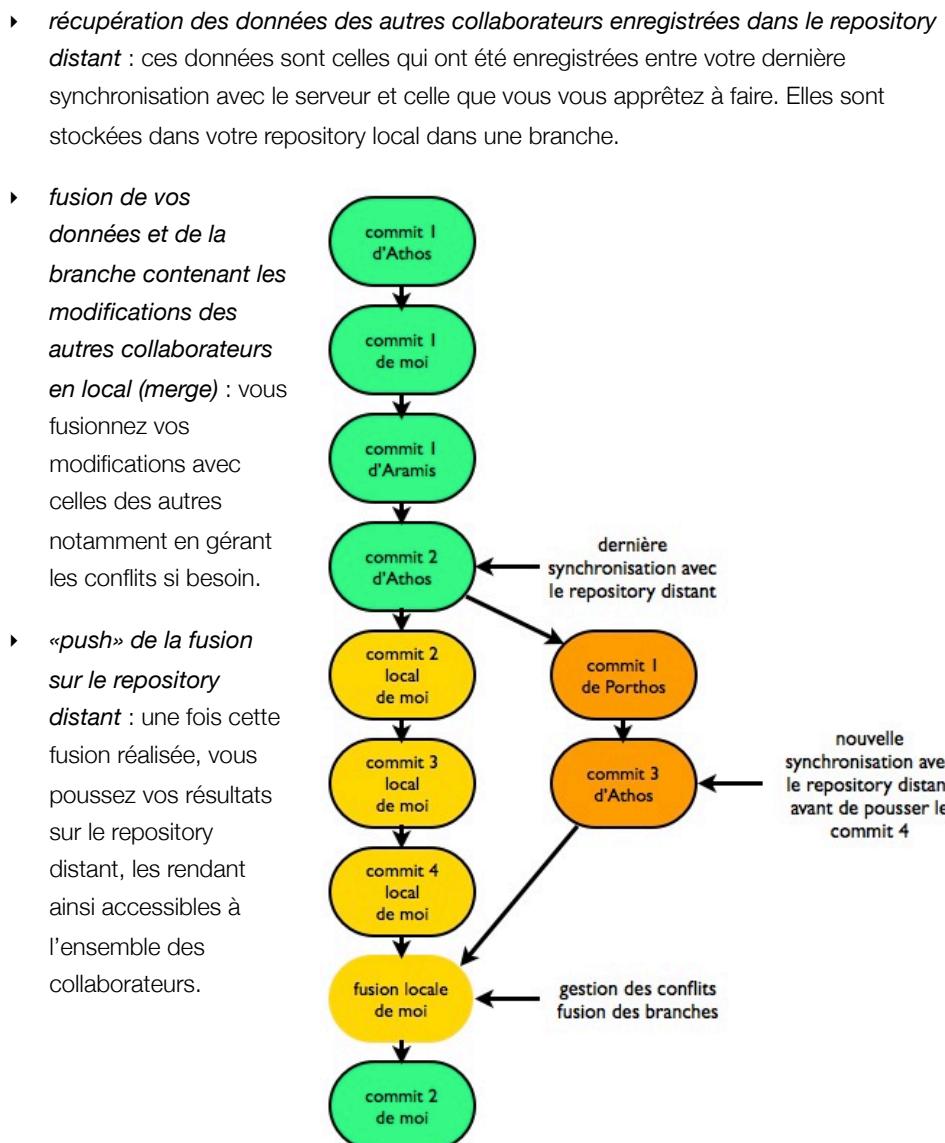


Les opérations en local avec un repository Git  
(tiré de [1])

⇒ **Les branches** : le système de branches de Git ressemble au premier regard à celui de Subversion. On peut en effet l'utiliser pour les mêmes besoins. Toutefois, si pour un débutant de Subversion, il est possible lors des premières expériences de se passer de l'utilisation de

branches, il est indispensable pour le débutant de Git de comprendre ce système. En effet, le système de branches est également utilisé pour la synchronisation et la fusion des données du repository local de Git d'un collaborateur avec le repository centralisé distant Git contenant les modifications des autres collaborateurs (depuis la dernière synchronisation...). Le système de branches est expliqué en détails dans la partie [4.3](#). L'utilisation des branches pour la synchronisation et la fusion des données avec le repository distant est expliquée dans la partie [4.4.2](#), et très succinctement ci-dessous.

- ⇒ **La synchronisation avec un repository distant** : la communication via le repository distant se fait de la manière suivante :



Les opérations de fusion avec le repository Git distant

### 3. TortoiseGit, installation et configuration

Subversion, un des systèmes de contrôle de version centralisés les plus utilisés, permet de réaliser ses tâches en ligne de commande. Afin de créer une interface graphique plus conviviale et de permettre aux développeurs sous Windows de bénéficier des atouts de Subversion, Stefan Küng et Lübbe Onken ont développé TortoiseSVN, un client Subversion implémenté comme une extension shell de Windows. TortoiseSVN s'intègre parfaitement à l'explorateur Windows et permet de retrouver la quasi-totalité des fonctionnalités de Subversion via une interface graphique très conviviale : superposition d'icônes aux répertoires et fichiers pour visualiser l'état (modifié, à jour, en conflit...), menu contextuel permettant de faire les commit, les mises à jour, graphe de l'historique, gestion des conflits sous forme graphique (via TortoiseMerge)... TortoiseSVN est très vite devenu très populaire (plus de 22 millions de téléchargements, 34 langues différentes ! Prix du meilleur outil [SourceForge.net 2007 Community Choice Award for Best Tool or Utility for Developers]).

En 2008, étant donné que Git était un système de contrôle de version distribué performant mais conscient qu'il lui manquait une interface graphique convivial, Frank Li décide de s'inspirer de TortoiseSVN et de créer une intégration shell pour Windows de Git. En étudiant le code de TortoiseSVN, il a créé TortoiseGit<sup>1</sup>. Comme TortoiseSVN, TortoiseGit est gratuit et open-source.

#### 3.1. Installation de Git et de TortoiseGit

L'installation de Git doit s'effectuer avant l'installation de TortoiseGit. Voici les liens vers les versions de Git et de TortoiseGit utilisées dans les salles de TP de l'IUT d'Informatique de Clermont-Ferrand (versions testées).

Git	<a href="http://code.google.com/p/msysgit/downloads/detail?name=Git-1.7.0.2-preview20100309.exe&amp;can=2&amp;q=">http://code.google.com/p/msysgit/downloads/detail?name=Git-1.7.0.2-preview20100309.exe&amp;can=2&amp;q=</a>
TortoiseGit	<a href="http://code.google.com/p/tortoisegit/downloads/detail?name=TortoiseGit-1.5.2.0-32bit.msi&amp;can=2&amp;q=">http://code.google.com/p/tortoisegit/downloads/detail?name=TortoiseGit-1.5.2.0-32bit.msi&amp;can=2&amp;q=</a>

#### 3.2. Création d'un repository sur Hina

- Connectez-vous sur Hina (en ssh ou à l'IUT).
- Rentrez la commande mkrepo.
- Répondez aux questions posées : rentrez la liste des noms (prénoms, noms ou uid de vous, vos collègues, et de l'enseignant responsable), rentrez le nom du projet.

(Merci à David Delon pour ce script qui vous fera gagner un temps précieux).

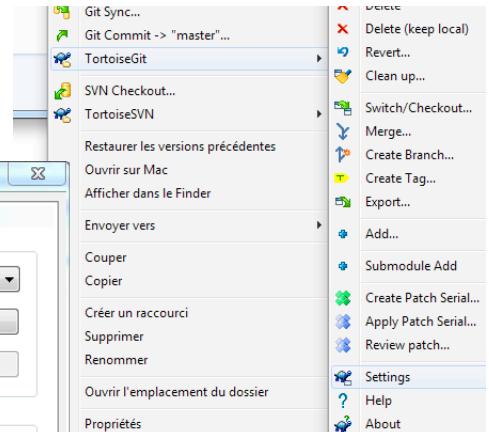
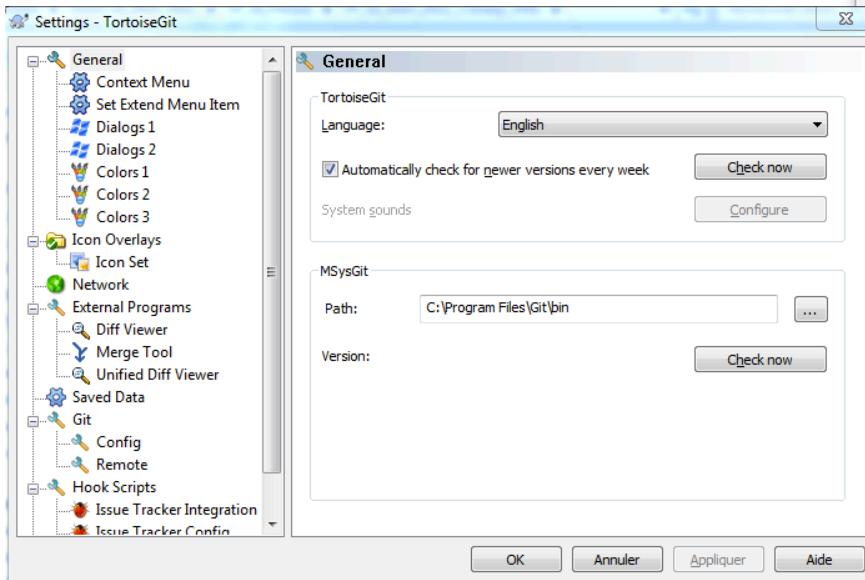
Note : pour créer un repository vous même sur un serveur sur lequel Git est installé, il suffit de créer un dossier (`$ mkdir mon_repo.git`<sup>2</sup>), et de taper la ligne de commande `$ git-init --bare` dans ce dossier.

<sup>1</sup> Il existe également d'autres inspirations de TortoiseSVN pour d'autres systèmes de contrôle de version : TortoiseCVS pour CVS, TortoiseBzr pour Bazaar, TortoiseHg pour Mercurial...

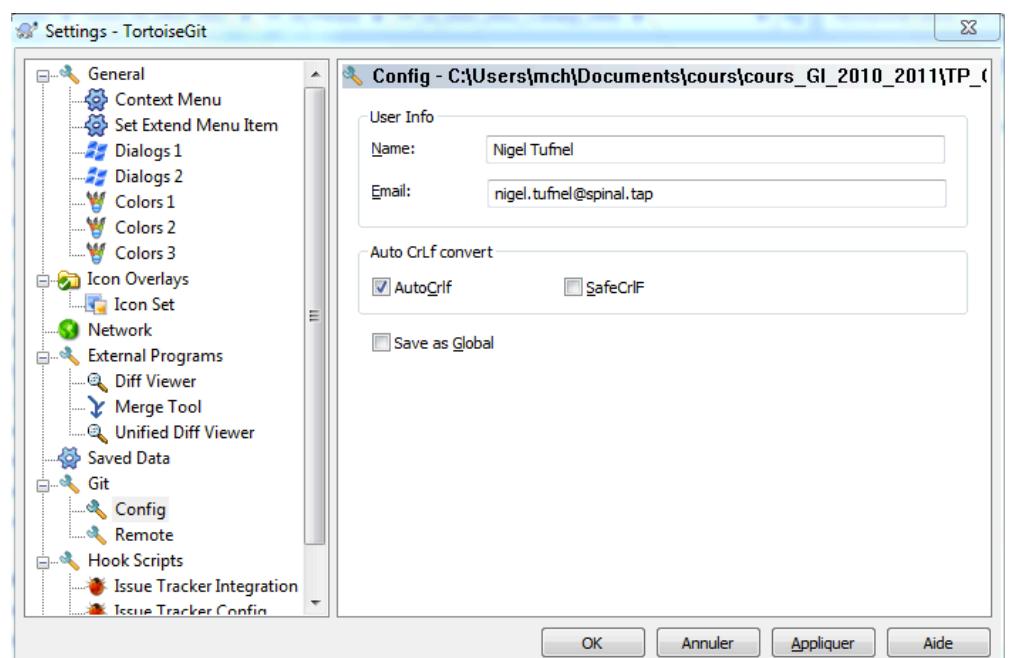
<sup>2</sup> Il n'est pas obligatoire de terminer le nom du projet par `.git`. Il s'agit d'une convention pour reconnaître rapidement les repository.

### 3.2.1. Configuration de TortoiseGit

Il y a très peu de choses à configurer avec TortoiseGit. Faites un clic droit sur un fichier ou un dossier, puis cliquez sur TortoiseGit -> Settings.



Dans le dialogue qui s'ouvre, sélectionnez dans la colonne de gauche «General». Assurez-vous que la case «Path» du panel «MSysGit» contient bien le chemin vers le dossier bin de votre installation de Git.



Dans la colonne de gauche, cliquez ensuite sur Git -> Config et rentrez votre nom et votre email. Ces informations seront utilisées lors de vos commit<sup>3</sup>.

<sup>3</sup> l'équivalent Git est :

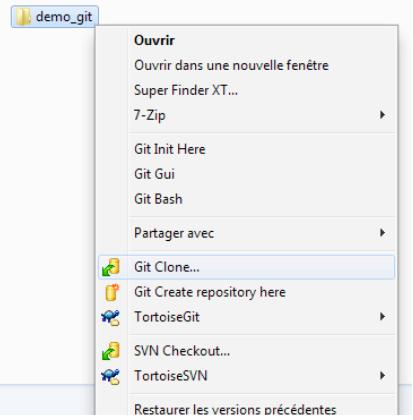
```
$git config --global user.name «nigel.tufnel»
$git config global user.email nigel.tufnel@spinal.tap
```

## 4. Utilisation de Git et TortoiseGit

Cette partie a pour objectif de présenter les commandes indispensables à l'utilisation de Git et TortoiseGit. Toutes les commandes ne seront pas présentées. Il existe notamment plusieurs solutions pour atteindre un même objectif, mais dans un soucis d'efficacité, seules certaines d'entre elles sont proposées ainsi que quelques recettes. Dans la plupart des cas, les opérations seront présentées dans leur forme Git (ligne de commande) et TortoiseGit (menu contextuel). Si nécessaire, des schémas illustreront les opérations. Des cas d'utilisation plus concrets seront présentés dans la partie suivante. Les opérations sont présentés dans l'ordre logique d'utilisation d'un collaborateur. Cette partie part du principe que le [repository est déjà créé](#).

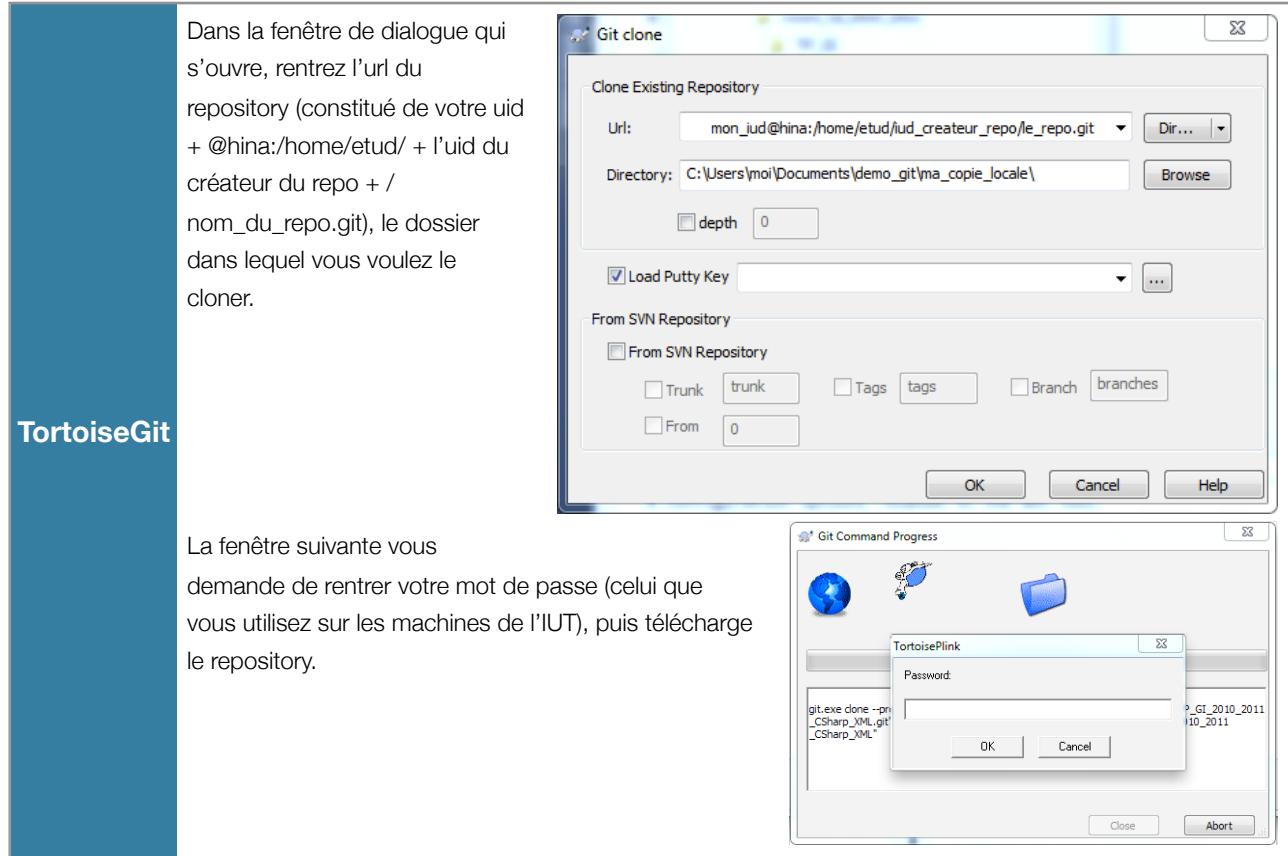
### 4.1. Cloner un repository existant en local

Lorsque vous souhaitez collaborer à un projet contrôlé avec Git, vous devez récupérer une copie du repository. La commande à utiliser est «clone»<sup>4</sup>. L'ensemble du repository (toutes les versions de tous les fichiers) est recopié sur votre disque dur. Cette commande crée automatiquement un dossier du même nom que votre repo sur votre disque dur, recopie toutes les données, et fait un «checkout»<sup>5</sup> de la dernière version.

Git	<pre>\$ git clone [url]</pre> <p>e.g. \$git clone git://giut.u-clermont1.fr:/home/mon_uid/info/mon_repo.git (le repo en local s'appelle mon_repo)</p> <p>e.g. \$git clone git://giut.u-clermont1.fr:/home/mon_uid/info/mon_repo.git mon_repo_a_moi (le repo en local s'appelle mon_repo_a_moi)</p>
TortoiseGit	<p>Créer le dossier dans lequel vous souhaitez stocker le repository en local et votre version de travail.</p> <p>Faites un clic droit sur ce dossier et cliquer sur «Git Clone...».</p> 

<sup>4</sup> Les adeptes de Subversion pourront remarquer qu'il ne s'agit pas d'un checkout !

<sup>5</sup> Le «checkout» de Git permet de [choisir la version \(ou la branche\) qui sera votre copie de travail](#). Il est possible de changer de version de travail à chaque fois qu'on le souhaite. Cette fonctionnalité ne sera pas traité plus en détails dans ce guide.



## 4.2. Utilisation du repository local

Pour cette partie, il est vivement conseillé d'avoir compris [les 3 états des fichiers en local](#) et [les 4 opérations avec les fichiers en local](#).

### 4.2.1. Transférer les modifications des fichiers dans le repository local : (add +) commit<sup>6</sup>

Le **commit** est la commande la plus effectuée lors de l'utilisation d'un projet contrôlé et versionné par Git. Les fichiers concernés et déjà versionnés doivent d'abord être placés dans la «*staging area*» dans le repository local, puis ils sont (ainsi que leurs modifications) ensuite transférés (**committed**) vers le repository local.

---

<sup>6</sup> Il semble qu'il n'y a pas de différences avec l'étape suivante dans l'appel des commandes, mais j'ai préféré faire la différence entre fichiers modifiés et nouveaux fichiers.

## Git

- ajout d'un fichier modifié et de ses modifications du répertoire de travail vers la *staging area* :

```
$ git add file
```

e.g. \$ git add OldClass.cs

- transfert des modifications des fichiers de la *staging area* vers le repository local :

```
$ git commit -m "message explaining modifications"
```

Important : tous les fichiers de la *staging area* sont transférés vers le repository local au moment du *commit*.

Note : le message accompagnant le commit `-m "message explaining modifications"` n'est pas obligatoire mais très fortement conseillé

Autre méthode plus rapide :

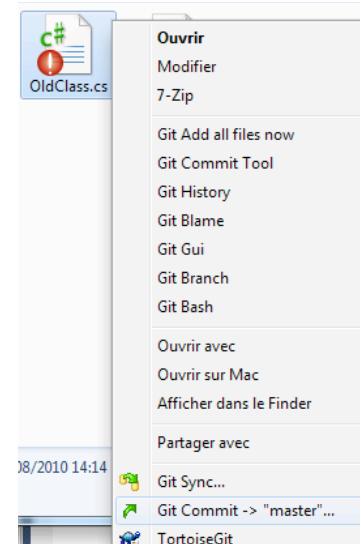
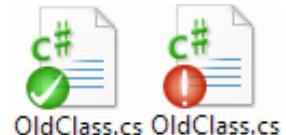
- transfert des fichiers modifiés (et de leurs modifications) du répertoire de travail directement vers le repository local sans passer par la *staging area* :

```
$ git commit -a -m "message explaining modifications"
```

Un fichier «à jour» par rapport au repository local, i.e. qui n'a pas été modifié depuis le dernier commit ou depuis la dernière synchronisation

(*ATTENTION ! à jour par rapport au repository local ne veut pas dire à jour par rapport au repository distant ! Il peut y avoir eu des soumissions d'autres utilisateurs depuis. c.f. parties suivantes*), est marqué d'une pastille verte sympathique qui rassure.

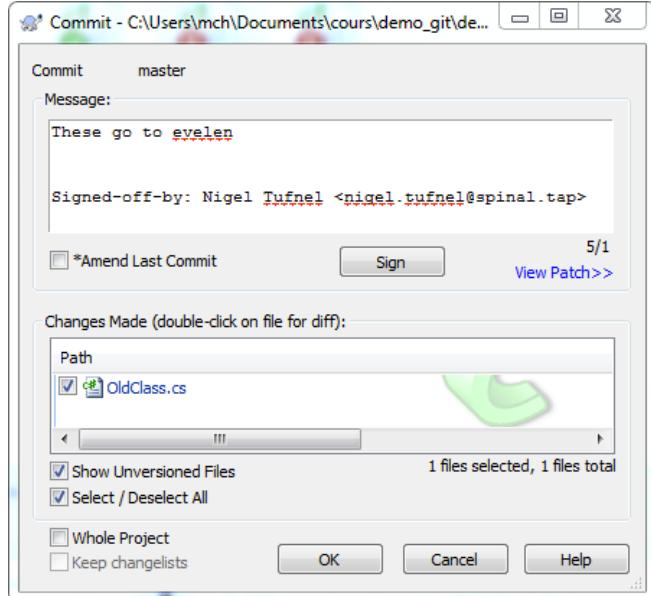
Lorsque vous avez modifié un fichier, et qu'il n'est donc plus à jour par rapport au repository, il est marqué d'une pastille rouge qui fait peur, pour indiquer qu'il est modifié.



## TortoiseGit

Un clic droit sur un fichier ou un dossier de votre répertoire de travail et un clic sur «Git Commit -> «master»...» (le master dépend de la branche sur laquelle vous êtes. [c.f. parties suivantes](#)), ouvre une nouvelle fenêtre de dialogue.

Cette fenêtre vous permet de vérifier quels sont les fichiers qui vont être transférés vers le repository local. Vous pouvez encore les (dé)sélectionner grâce à la ListBox «ChangesMade». Vous devez ensuite décrire votre **commit** à l'aide d'un message que vous pouvez signer avec le bouton «Sign» (si vous avez [configuré](#) correctement TortoiseGit).

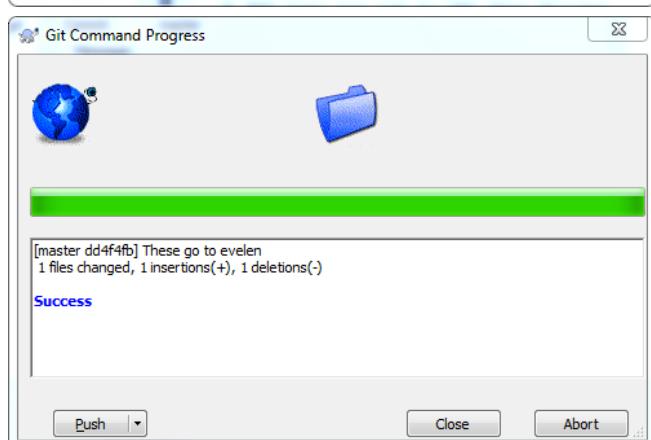


## TortoiseGit

Une nouvelle fenêtre de dialogue s'ouvre vous illustrant la progression du **commit**. Les fichiers modifiés sont transférés vers le repository local (dans la branche «master»).

Notez le bouton «Push» qui vous permet de transférer les modifications du repository local vers le repository distant (ce qui sera expliqué dans [une des parties suivantes](#)).

Le fichier est à nouveau à jour, il retrouve sa jolie pastille verte.



Note : TortoiseGit rend totalement transparent le passage par la *staging area*.

### 4.2.2. Ajouter de nouveaux fichiers au repository local : **add + commit**

L'ajout de fichiers au repository local se fait en deux étapes : la commande **add** transfert des fichiers non versionnés dans la «*staging area*» (état «*staged*») ; la commande **commit** transfert ces fichiers de la «*staging area*» sur le repository local. Cette étape ressemble beaucoup à la précédente, mais l'intention était différente, elle est traitée à part. De plus, elle se déroule légèrement différemment via TortoiseGit.

## Git

### 1. add

depuis le dossier où le fichier est contenu :

```
$ git add [file]
```

e.g. `$git add MaClasse.cs`

(*MaClasse.cs* est maintenant dans la *staging area*)

e.g. `$git add *.cs`

(Tous les fichiers \*.cs de ce dossier sont maintenant dans la *staging area*)

### 2. commit

la commande suivante ajoute *MaClasse.cs* au repository local (ainsi que tous les autres fichiers de la *staging area*).

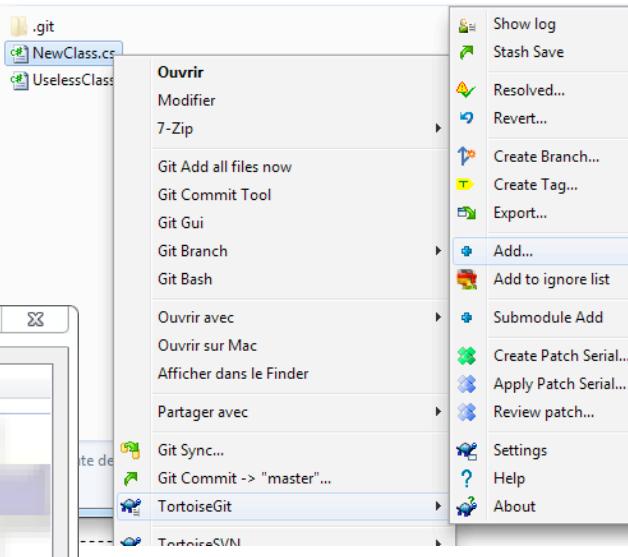
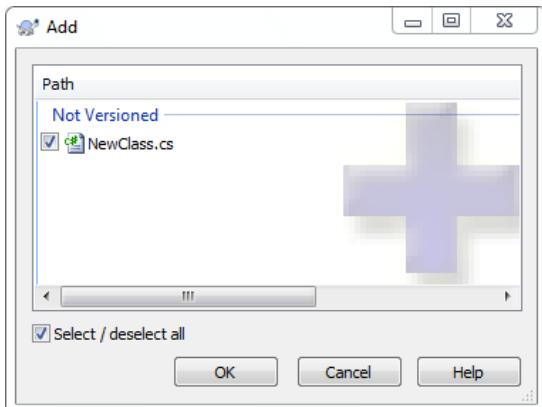
```
$ git commit -m "added MaClasse.cs"
```

## TortoiseGit

### 1. add

La commande **add** se fait via le menu contextuel : clic droit sur le(s) fichier(s) à ajouter -> Tortoise Git -> Add...

Une fenêtre de dialogue apparaît alors vous permettant d'ajuster votre ajout en (dé)sélectionnant les fichiers à ajouter.



Après avoir valider votre ajout, votre fichier est marqué d'un gros + bleu, indiquant qu'il se trouve comme modifié (**staged**) et sera ajouté au prochain *commit*.



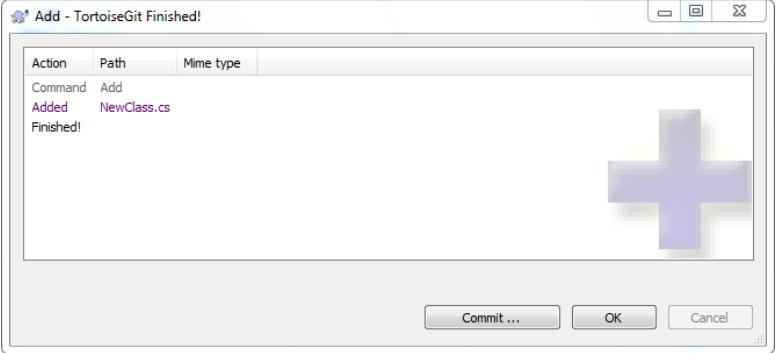
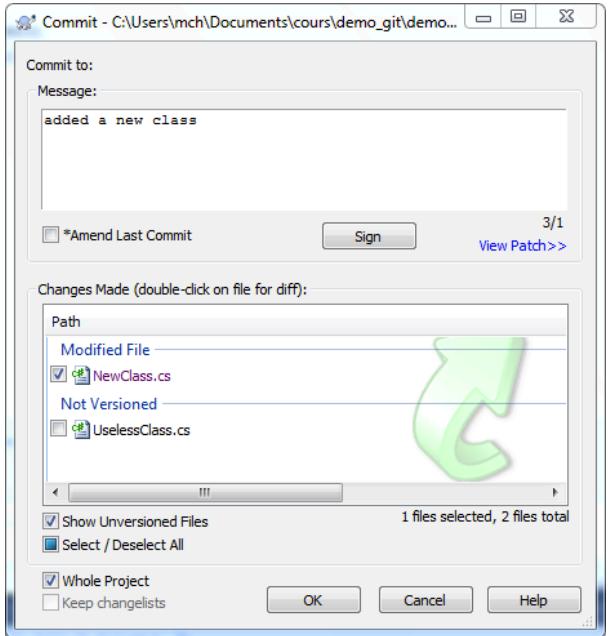
**TortoiseGit**

**2. commit**

La fenêtre de dialogue suivante vous permet d'ailleurs de faire directement le *commit*. Même s'il est conseillé de le faire une fois pour tous les fichiers ajoutés et modifiés, voici tout de même la liste des commandes à effectuer.

En cliquant sur «Commit ...», vous voyez apparaître la fenêtre de dialogue suivante. Notez qu'il est encore possible d'ajouter des fichiers non encore ajoutés (et donc *unstaged*) dans la *staging area* pour les ajouter au *commit* ! N'oubliez pas non plus d'ajouter le message pour les collaborateurs.

En cliquant sur OK, vous lancez le commit, dont la progression apparaît dans la fenêtre suivante (cf. [partie précédente](#)).

#### 4.2.3. Supprimer des fichiers du repository local : `rm`

Git permet de supprimer des fichiers du repository local. La suppression a réellement lieu au commit suivant, et il reste bien entendu possible de retrouver les fichiers avant leur suppression en revenant sur une [version précédente](#). La suppression s'opère de la manière suivante : l'appel de `rm` sur un fichier le retire de la staging area et donc des fichiers à versionner ; le `commit` suivant ne contient plus le fichier dans la nouvelle version.

## Git

### 1. rm

depuis le dossier où le fichier est contenu :

```
$ git rm [file]
```

e.g. `$git rm UselessClass.cs`

(UseClass.cs n'est maintenant plus dans la *staging area*)

e.g. `$git rm *.pdb`

(Tous les fichiers \*.pdb de ce dossier vont être supprimés du repository local)

Pour supprimer le fichier du repository mais le garder sur son disque dur, on peut aussi utiliser la commande :

```
$ git rm --cached [file]
```

### 2. commit

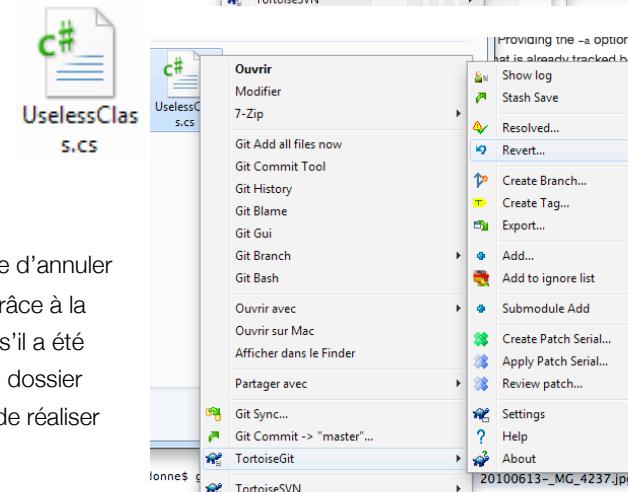
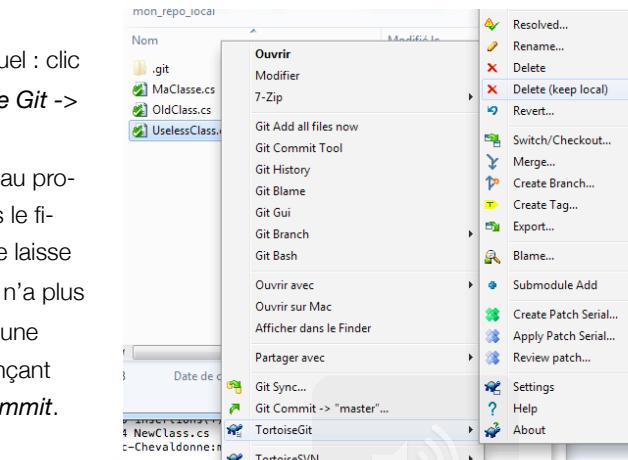
la commande suivante opère la suppression : les fichiers supprimés ne seront plus cette nouvelle version du repository.

e.g. `$ git commit -m "deleted UselessClass.cs and all *.pdb files"`

### 1. rm

La commande `rm` se fait via le menu contextuel : clic droit sur le(s) fichier(s) à supprimer -> *Tortoise Git* -> *Delete* ou *Delete (keep local)*.

Les deux préparent la suppression du fichier au prochain *commit*, mais le premier envoie en plus le fichier à la corbeille (`rm`), alors que le second le laisse en place sur le disque dur (`rm --cached`, il n'a plus aucune pastille). Le dossier contenant prend une méchante pastille en forme de X rouge annonçant qu'il y aura des suppressions au prochain *commit*.

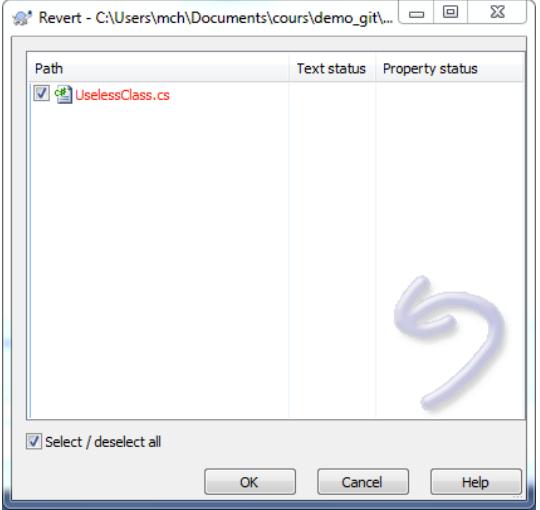


Via TortoiseGit, il est toutefois encore possible d'annuler une suppression (avant le commit bien sûr) grâce à la commande *revert*. Un clic droit sur le fichier (s'il a été supprimé avec *Delete (keep local)*) ou sur un dossier parent, puis *TortoiseGit* -> *Revert...* permet de réaliser cette opération.

## TortoiseGit

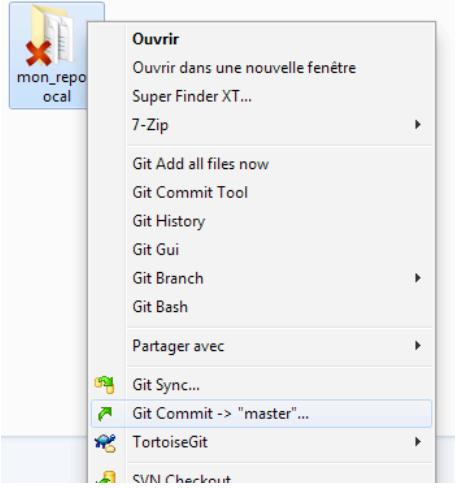
TortoiseGit

Il faut confirmer dans la boîte de dialogue qui s'ouvre ensuite, en sélectionnant les fichiers à récupérer.



2. **commit**

Pour réaliser la suppression dans la version suivante du repository local, on doit ensuite faire un *commit* (pour comme add), sur un dossier parent. La réalisation de la phase commit a ensuite été décrite dans les parties *commit* et *add* précédentes.



#### 4.2.4. Déplacer ou renommer un fichier dans le repository local : mv + commit

Git utilise la même commande pour déplacer ou renommer un fichier (les mêmes opérations sont réalisées). TortoiseGit les différencie mais fait la même chose.

Git

1. Pour renommer un fichier :

```
$ git mv old_name new_name
```

1bis. Pour déplacer un fichier :

```
$ git mv file another_folder/file
```

(note : another\_folder doit exister)

2. **commit**

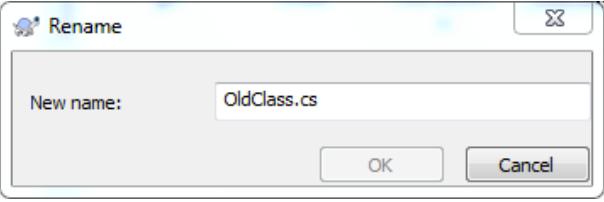
la commande suivante réalise l'opération dans la prochaine version dans le repository local.

e.g. `$ git commit -m "renamed old_named and moved file"`

**TortoiseGit**

1. Pour renommer un fichier :

La commande `mv` se fait via le menu contextuel : clic droit sur le fichier à renommer -> *Tortoise Git -> Rename...*



La boîte de dialogue ci-contre s'ouvre alors et permet de renommer le fichier. Celui-ci est maintenant marqué par un gros + bleu, car git opère en interne, une copie, un *add* et un *remove*. Le fichier renommé est donc ajouté. Il faut ensuite effectuer le *commit* comme pour un add.

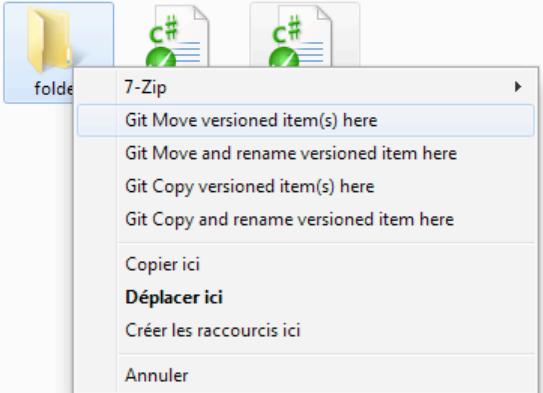
1bis. Pour déplacer un (ou plusieurs) fichiers :

On peut utiliser la même commande via TortoiseGit pour renommer ou déplacer des fichiers. C'est juste un peu pénible et très répétitifs si on veut déplacer plusieurs fichiers. Sinon, on peut aussi faire un «right-dragging» : sélectionnez les fichiers à déplacer ; faites un clic droit dessus tout en gardant le bouton droit enfoncé ; déplacer les fichiers vers le nouveau dossier parent et lâchez le bouton droit. Le menu contextuel ci-contre apparaît, avec 4 options intéressantes :

- ▶ **Git Move versioned item(s) here** : déplace les fichiers versionnés dans le nouveau dossier (ils sont ajoutés et ceux d'origine sont détruits) ;
- ▶ **Git Move and rename versioned item here** : fait la même chose mais vous autorise à renommer les fichiers avant ;
- ▶ **Git Copy versioned item(s) here** : fait la même chose que le premier mais sans détruire les fichiers dans le dossier d'origine ;
- ▶ **Git Copy and rename versioned item here** : fait la même chose que le précédente mais vous autorise à renommer les fichiers avant.

L'historique des versions des fichiers est à chaque fois conservé.

2. commit : [cf. partie 2 du add.](#)



#### 4.2.5. Historique du repository local

Au fur et à mesure de l'avancement d'un projet, on a parfois besoin de regarder en arrière, voir ce qui a été modifié par soi-même ou par ses collaborateurs. Pour cela, on regarde l'ensemble de l'historique du repository.

## Git

```
$ git log
permet de visualiser l'historique du repository de manière assez moche. On retrouve les opérations effectuées, l'auteur des modifications ainsi que le message qu'il a laissé.

exemple de résultat :
commit cf273ccca4acebdf67bdb43043409600f85cf800
Author: Marc Chevaldonné <marc.chevaldonne@u-clermont1.fr>
Date:   Sun Aug  8 15:24:45 2010 +0200

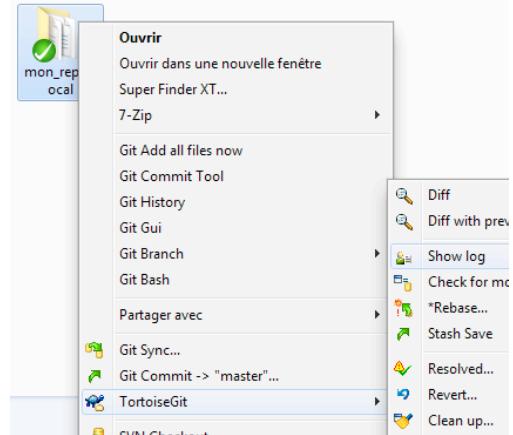
        deleting NewClass.cs

commit 5a6471bcd67289e7d04541130902411691f1f0b6
Author: Marc Chevaldonné <marc.chevaldonne@u-clermont1.fr>
Date:   Sun Aug  8 12:11:55 2010 +0200

        added MaClasse.cs
```

Pour visualiser l'historique du repository local, il vous suffit de faire un clic droit sur le dossier parent et d'aller chercher *TortoiseGit* -> *Show log*.

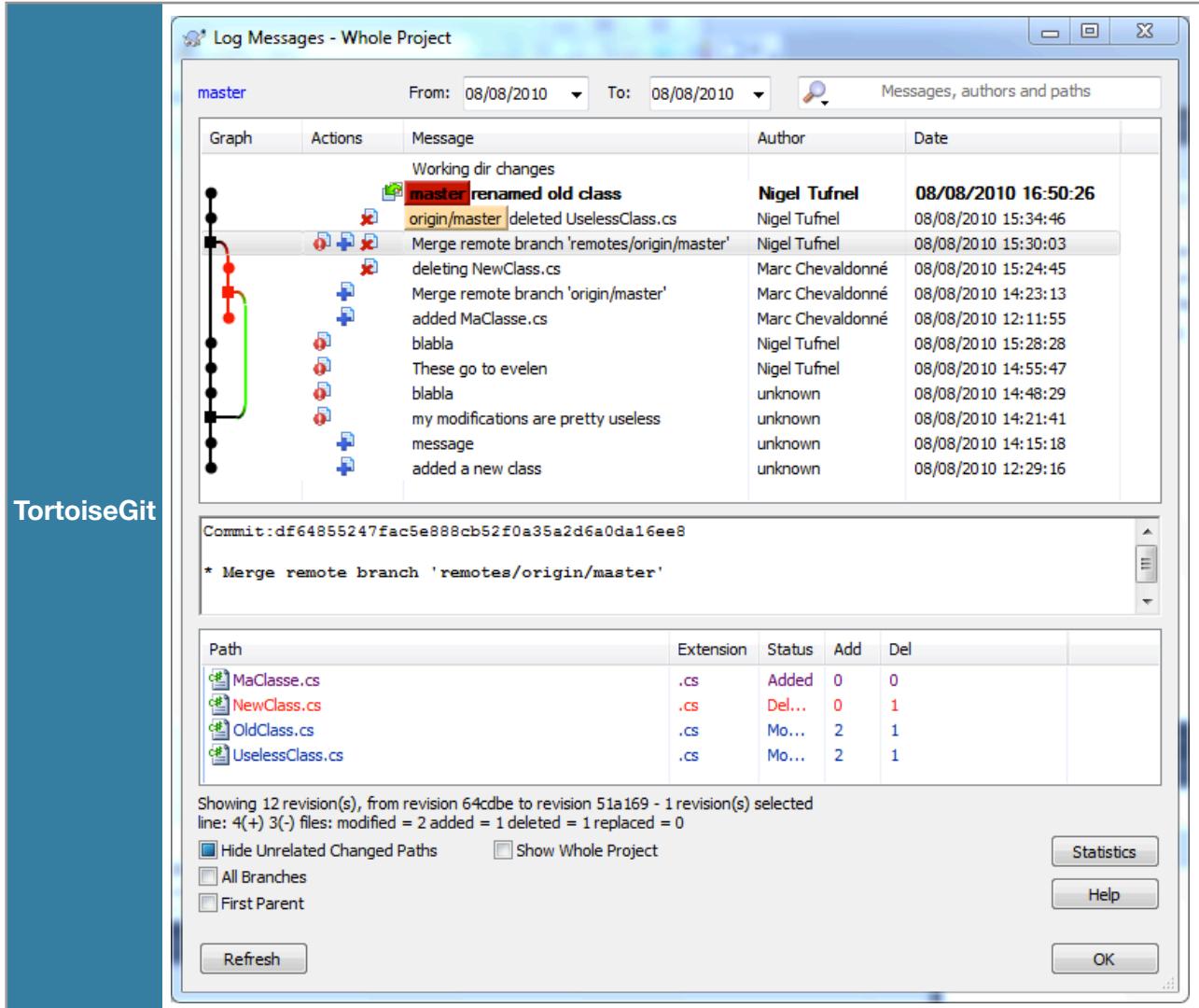
Ceci ouvre une nouvelle fenêtre de dialogue représentant l'historique des modifications du repository sous forme graphique.



## TortoiseGit

Voici un exemple de l'historique d'un repository.

- ▶ La première ListBox donne des informations sur les différents commit :
  - la colonne **Graph** représente les différentes [branches](#) et les fusions de branches,
  - la colonne **Actions** schématisé le type d'opérations qui ont été effectuées (modifications, ajouts, suppressions, [checkout](#)...),
  - la colonne **Message** diffuse les messages des commit,
  - les colonnes **Author** et **Date** donnent les informations sur l'auteur, le jour et l'heure du commit.
- ▶ La seconde case donne des informations sur le code du commit, le type d'opérations.
- ▶ La dernière ListBox donne des détails sur les fichiers modifiés et le type des modifications.

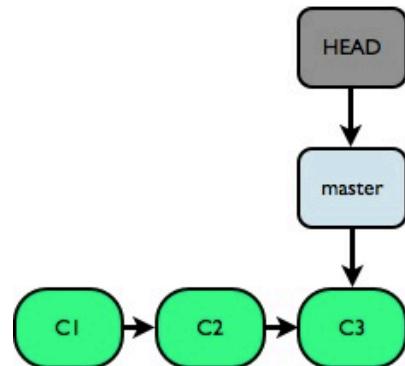


## 4.3. Travailler avec les branches

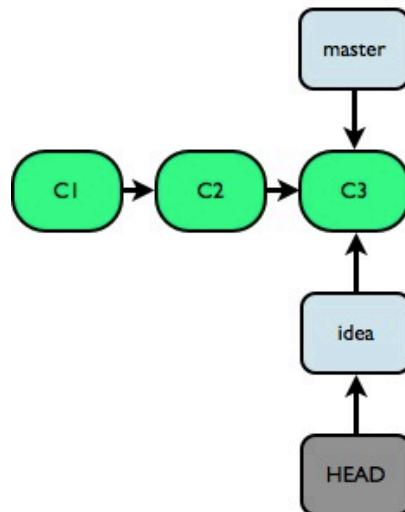
Si vous utilisez Git pour un projet sur lequel vous travaillez seul, vous pouvez vivre sans branches. Dans ce cas, vous n'utilisez d'ailleurs pas vraiment un système de contrôle de version distribué, mais plutôt un système de contrôle de version local, avec sauvegardes sur un serveur distant. Toutefois, l'utilisation de branches pourrait déjà vous être grandement utile. En revanche, dès que vous travaillez dans une équipe, même de 2 personnes, en utilisant Git, le système de branches devient incontournable, en particulier pour la [synchronisation avec le repository distant](#). Cette partie présente le principe des branches et leur utilisation dans un repository Git local. La partie suivante traite de l'utilisation des branches lors de la synchronisation avec le repository distant.

### 4.3.1. Principe

Le concept de branches de Git est indissociable de celui des copies de travail. Pour bien comprendre ces principes, il faut se représenter deux types d'entités : les commits et un système de pointeurs sur commit. Par commit, nous entendons ici une version du repository, i.e. un état (une photographie) du projet à un instant donné. Ces commits seront représentés dans les schémas suivants par une case avec Ci où i est un indice qui croît avec le temps. La branche principale s'appelle «master»<sup>7</sup>. «HEAD» est un pointeur sur la branche courante (celle que vous êtes en train d'utiliser), soit «master» tant que vous ne faites pas de branches.

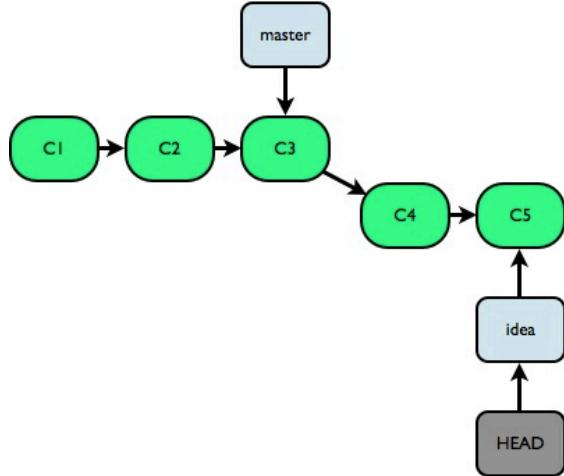


Considérez les images ci-contre. Votre repository local contient quelques commits. Vous avez une idée pour la suite, mais vous n'êtes pas encore certain de son intérêt ou de sa faisabilité. Vous ne voulez donc pas faire de commits sur votre branche «master» avec ce test. La branche est là pour vous aider : vous [créez une branche](#) qui vous permet de bénéficier de l'historique des commits de master, mais dont les commits futurs divergeront à partir de la création de la branche.

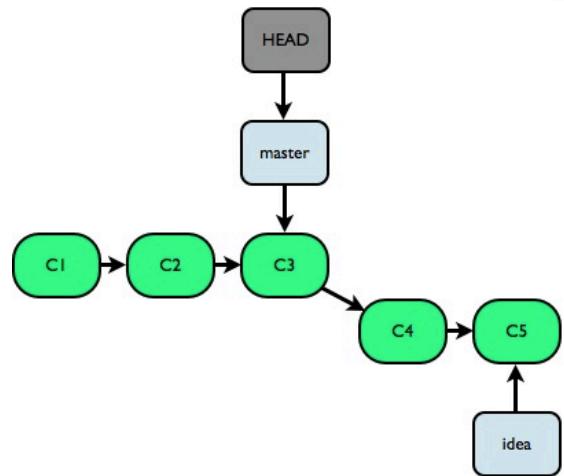


<sup>7</sup> c'est «un peu» l'équivalent du trunk de Subversion

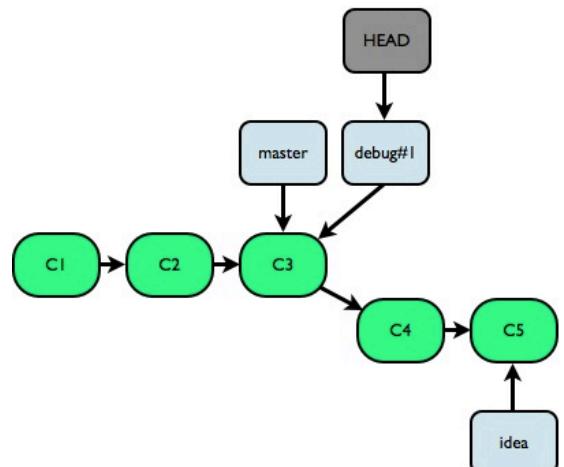
Vous réalisez dès lors quelques commits sur cette branche. Notez qu'au démarrage, les pointeurs «master» et «idea» pointent sur la même version du projet. «idea» étant la branche active, le pointeur se déplace avec les nouveaux commits, alors que «master» reste inchangé. De plus, «idea» étant la branche active, votre répertoire de travail local contient la version correspondant à C5.



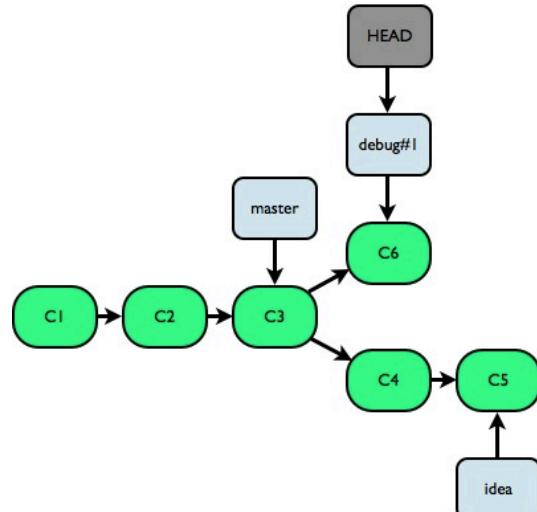
Pendant ce développement, vous découvrez (ou votre binôme, votre maman...) un horrible bug qu'il faut vite corriger car la version pointée par «master» était déjà utilisée par le client. Vous vous replacez donc sur «master» à l'aide d'un [checkout](#), et la copie de travail locale redevient la version correspondant à C3.



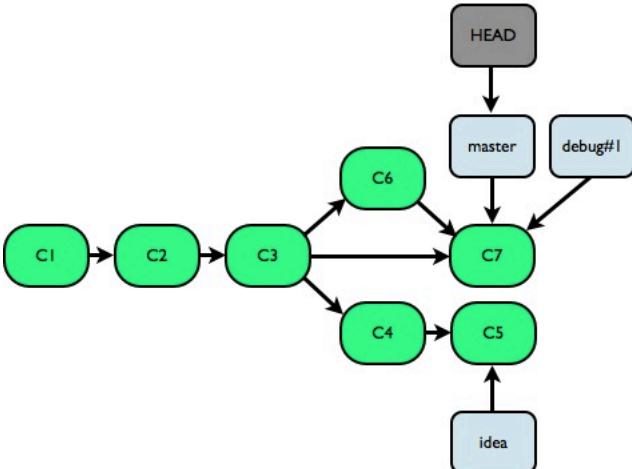
Vous [créez une nouvelle branche](#) pour résoudre le bug 1 («debug#1»), sans modifier la branche «master», tant que le bug n'est pas corrigé. La branche «debug#1» devient la branche active, et la copie de travail locale correspond toujours à C3 pour le moment.



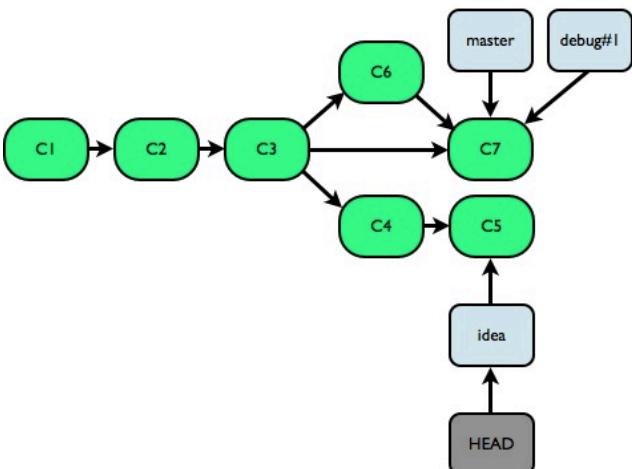
Un commit suffit pour corriger ce bug. Cette correction doit maintenant être réintégrée dans la branche «master». La copie de travail locale est la version correspondant à C6.



Pour cela, on opère une [fusion des branches](#) «master» et «debug#1» (merge). Cette fusion s'opère assez bien puisque «master» n'a pas été modifiée depuis la création de «debug#1», créée à partir de «master». «master» redevient la branche courante, et la copie de travail locale devient la version fusionnée correspondant à C7<sup>8</sup>.



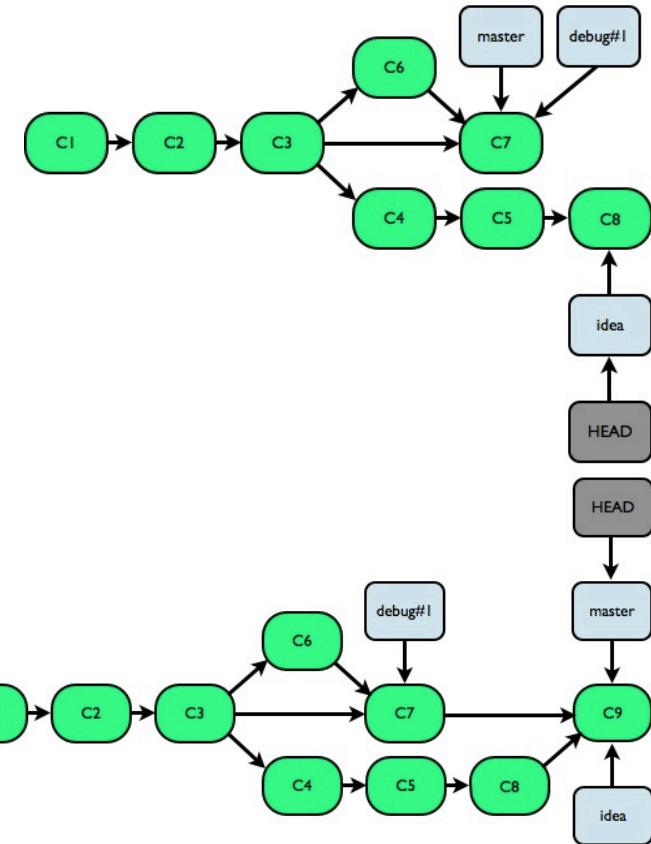
Le bug corrigé, vous pouvez revenir sur le développement de votre idée, grâce à un [checkout](#) sur la branche «idea». La copie de travail locale redevient la version correspondant à C5.



<sup>8</sup> On pourrait également supprimer la branche ou faire un [rebase](#). Mais ceci ne fait pas l'objet de ce document.

Votre développement nécessite encore quelques commits. Vous découvrez que cette idée s'avère géniale, et vous décidez donc de la réintégrer dans la branche «master». Cette fois-ci, la [fusion](#) s'avère plus compliquée : l'ancêtre commun à «idea» et «master» est C3. Les deux branches ont depuis évolué : «master» en C7 et «idea» en C8.

La fusion de «master» et «idea» s'opère en C9. La branche active redevient «master» et la copie de travail locale correspondant à C9. La différence avec la fusion précédente est que cette nouvelle fusion ne pourra certainement pas être totalement automatique. Elle nécessitera une intervention manuelle, une [gestion des conflits](#), si des fichiers ont été modifiés dans «master» et dans «idea» depuis l'ancêtre commun des deux branches.



#### 4.3.2. Gestion des branches

Quand faut-il faire une branche ? Quand faut-il en fusionner ? Il n'y a pas de réponses à ces questions, il n'y a que des suggestions. En voici quelques-unes...

- ▶ Branches de debug : ces branches sont généralement créées dans un but précis qui est celui de corriger un bug en particulier et ont donc normalement une durée de vie limitée. Elles sont supposées être réintégrées assez rapidement dans la branche qui les a vues naître.
- ▶ Branches de test : elles ont pour objectif de tester de nouvelles idées, pas forcément prévues au démarrage du projet. Leur durée de vie est moyenne et leur réintroduction dans la branche qui les a vues naître dépend de l'idée elle-même.
- ▶ Branches de module : l'objectif de telles branches est de préparer une amélioration, un module, complètement à part, et de ne l'intégrer qu'une fois terminer. Ces branches doivent donc être réintégrées dans la branche «master». Leur durée de vie dépend de la taille du module à ajouter. Notons toutefois que plus celle-ci est longue, plus le risque de problèmes lors de la fusion sera importante.
- ▶ Branches de développement : elles représentent tout un pan de votre projet, une version de l'application. Ces branches vivent tout au long du projet et ne sont jamais réintégrées dans la branche «master». On peut par exemple imaginer dans le cadre de notre jeu, la branche «Web» et la branche «XBox».

#### 4.3.3. Créer une branche

##### Git

1. Créer la branche :

```
$ git branch branch_name
```

e.g. \$ git branch debug#1

2. rendre la branche active :

```
$ git checkout branch_name
```

e.g. \$ git checkout debug#1

Autre méthode plus rapide (crée la branche et la rend active) :

```
$ git checkout -b branch_name
```

e.g. \$ git checkout -b debug#1

Note : \$ git branch

renvoie la liste des branches du repository. La branche avec un \* est la branche active.

e.g.

debug#1

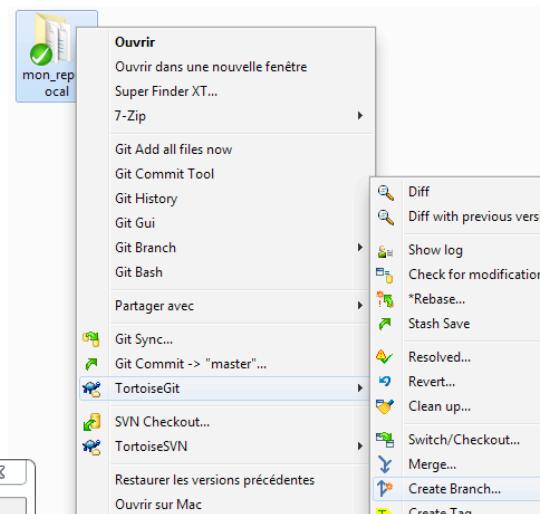
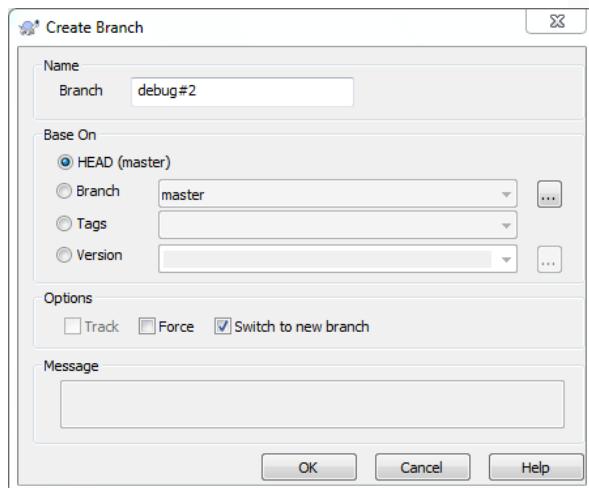
idea

\* master

Faites un clic droit sur le dossier parent, puis choisissez *TortoiseGit -> Create branch...*

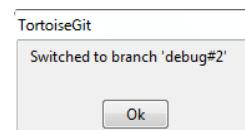
Une boîte de dialogue s'ouvre pour vous aider à créer la branche. Vous pouvez alors choisir :

- ▶ le nom de la branche,
- ▶ à partir de quelle branche celle-ci est créée (par défaut, la branche active),
- ▶ d'ajouter un message,
- ▶ d'indiquer si la branche créée devient la branche active ou non.



Si vous n'avez pas choisi de la rendre active automatiquement, vous pouvez le faire à l'aide d'un *checkout* par la suite.

Si vous avez choisi de la rendre active, ceci est confirmé par le message suivant :

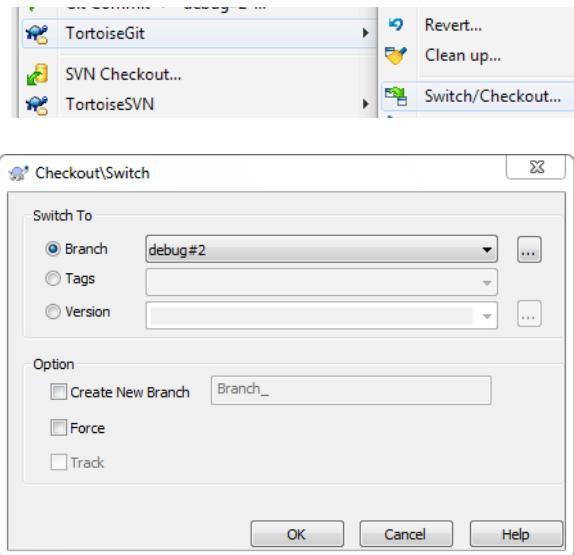


Enfin, lorsque vous voulez faire un *commit*, vous remarquerez que le click droit n'offre plus l'option «*Git Commit -> master...*» mais «*Git Commit -> debug#2...*» indiquant par là que la branche active sur laquelle vous opérez est «*debug#2*». Git Commit -> "debug#2"...

##### TortoiseGit

#### 4.3.4. Switch/Checkout de branche ou de version

Git	<pre>\$ git checkout branch_name</pre> <p>e.g. pour retourner sur la branche debug#1 \$ git checkout debug#1</p> <p>e.g. pour retourner sur la branche master \$ git checkout master</p>
TortoiseGit	<p>Faites un clic droit sur le dossier parent, puis choisissez <i>TortoiseGit -&gt; Switch/Checkout...</i></p> <p>Une boîte de dialogue s'ouvre pour vous aider à changer de branche ou de version. Vous pouvez alors choisir :</p> <ul style="list-style-type: none"> <li>▶ la branche dans le menu déroulant ou à travers une représentation en arbre (en cliquant sur le bouton «...»),</li> <li>▶ le <a href="#">tag</a>,</li> <li>▶ la version (en la choisissant dans le menu déroulant ou dans l'historique en cliquant sur le bouton «...»),</li> <li>▶ une nouvelle branche (en indiquant son nom).</li> </ul> <p>Enfin, vous pouvez vérifier sur quelle branche vous vous trouvez lorsque vous voulez faire un <i>commit</i> : clic droit offre l'option «<i>Git Commit -&gt; branch_name...</i>» où branch_name est le nom de la branche active.</p> <p> <a href="#">Git Commit -&gt; "debug#2..."</a></p>



#### 4.3.5. Fusionner des branches

La fusion de deux branches peut se passer bien ... ou mal. Entendons par là, peut se faire automatiquement ou «à la main». Si les fichiers modifiés dans chacune des branches sont différents, alors la fusion se passera «bien», automatiquement. Si vous avez modifié le même fichier dans les deux branches qui fusionnent, il y aura des [conflits à gérer](#), «à la main». Cette partie présente les commandes à effectuer pour fusionner deux branches sans considérer les conflits, comme [«master» et «debug#1»](#) dans le paragraphe d'introduction. La partie suivante introduit les outils de gestion des conflits.

Git	<ol style="list-style-type: none"> <li>1. se placer dans la branche qui «restera» après la fusion</li> </ol> <pre>\$ git checkout main_branch</pre> <p>e.g. «master» : \$ git checkout master</p> <ol style="list-style-type: none"> <li>2. fusionner l'autre branche sur la première</li> </ol> <pre>\$ git merge merged_branch</pre> <p>e.g. «debug#1» : \$ git merge debug#1</p> <p>S'il n'y a aucun message d'avertissement, la fusion est réussie.</p> <ol style="list-style-type: none"> <li>3. <u>optionnel</u>, détruire la branche fusionnée :</li> </ol> <pre>\$ git branch -d merged_branch</pre> <p>e.g. «debug#1» : \$ git branch -d debug#1</p>
-----	---

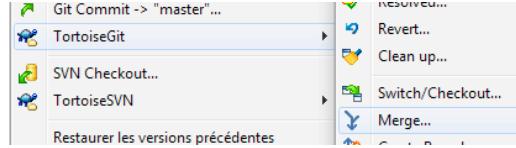
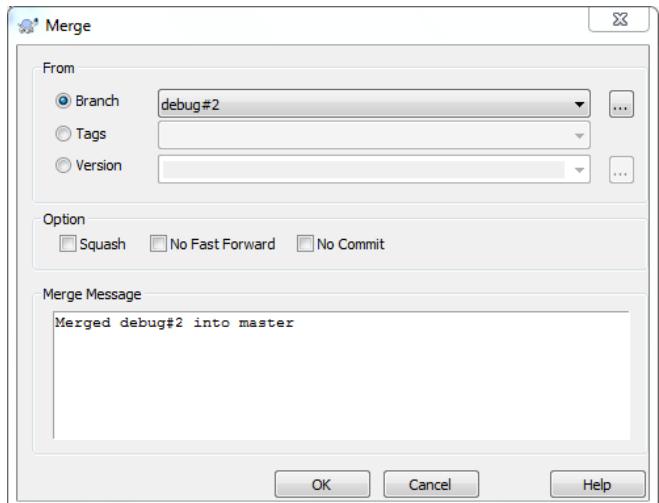
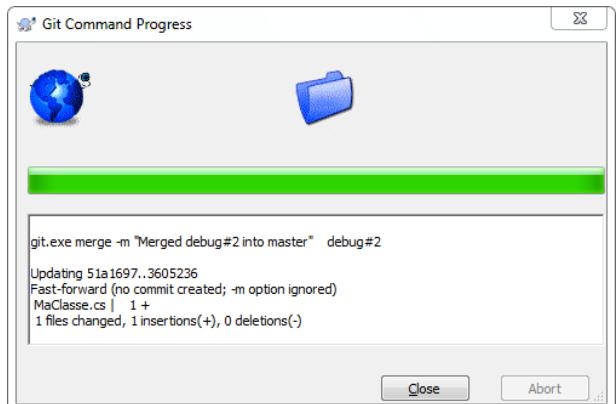
**TortoiseGit**

1. Faites un clic droit sur le dossier parent, puis choisissez *TortoiseGit -> Switch/Checkout...*. Placez-vous dans la branche qui va recevoir la branche fusionnée (e.g. «master»).

2. Faites un clic droit sur le dossier parent, puis choisissez *TortoiseGit -> Merge...*.

3. Une boîte de dialogue s'ouvre. Dans «From», choisissez la branche qui va fusionner dans votre branche courante (e.g. «debug#2»). N'oubliez pas le petit message explicatif.

4. La dernière fenêtre montre la progression de la fusion et indique si elle s'est bien déroulée.

#### 4.3.6. Gestion des conflits

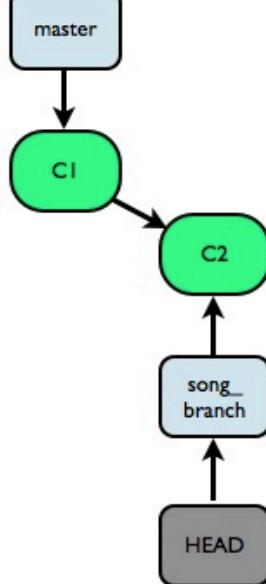
Oui... parfois ça se passe mal. Juste quelques conflits en fait, mais qu'il faut éditer à la main. Malheureusement, c'est même la plupart du temps comme ça. Imaginez un projet VisualStudio. Deux collaborateurs du projet Git travaillent sur deux fichiers différents. A priori, il n'y a pas de soucis car les fichiers sont différents, mais s'ils appartiennent au même projet, le projet lui-même risque fort d'être en conflit.

Pour expliquer les opérations à réaliser, imaginons la situation suivante.

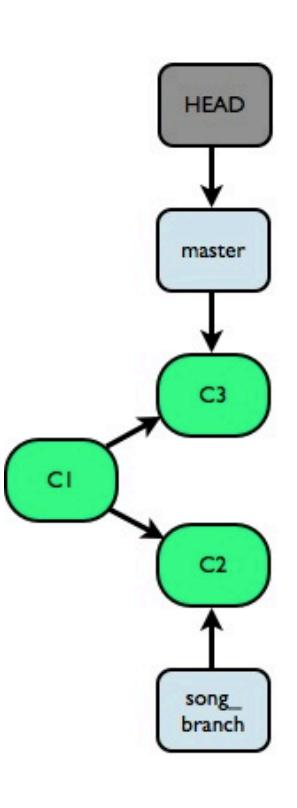
- i. dans la branche «master», le fichier Tap\_tappe.txt est créé et au commit C1, contient le texte suivant :

C1	<pre>&lt;Onstage&gt; New York M C:         You want it right, direct from hell, Spinal Tap! --- Spinal Tap performs Tonight I'm Gonna Rock You Tonight --- David:      We are Spinal Tap from the UK you must be the USA!</pre>	 <pre> graph TD     HEAD[HEAD] --&gt; master[master]     master --&gt; C1[C1] </pre>
----	---	---

- ii. la branche «song\_branch» est créée ; le fichier Tap\_tappe.txt est modifié et au commit C2, contient le texte suivant (en rouge le texte modifié) :

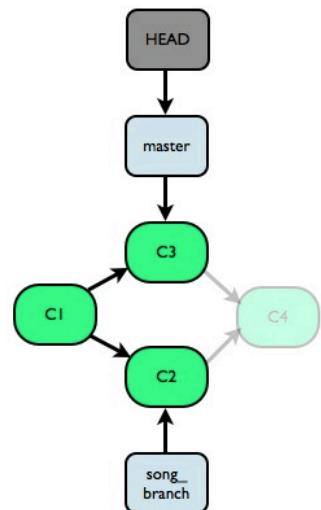
C2	<pre>&lt;Onstage&gt; New York M C:         You want it right, direct from hell, Spinal Tap! --- Spinal Tap performs Tonight I'm Gonna Rock You Tonight --- David:      We are Spinal Tap from the UK you must be the USA! --- Tonight I'm Gonna Rock You Tonight (lyrics) --- Little girl, it's a great big world but there's only one of me You can't touch 'cause I cost too much but Tonight I'm gonna rock you (Tonight I'm gunna rock you) Yeah tonight I'm gonna rock you (Tonight I'm gunna rock you) Tonight!  You're sweet but you're just four feet And you still got your baby teeth You're too young and I'm too well hung Tonight I'm gonna rock you (Tonight I'm gunna rock you) Yeah tonight I'm gonna rock you (Tonight I'm gunna rock you) Tonight!  Whoa yeah  You're hot, you take all we got, not a dry seat in the house Next day, we'll be on our way Tonight we're gonna rock you (Tonight we're gunna rock you) Yeah tonight we're gunna rock you (Tonight we're gunna rock you) Tonight!  Chorus: Little girl, it's a great big world, but there's only one of - meeeeeeeeee --- end of the song ---</pre>	 <pre> graph TD     HEAD[HEAD] --&gt; song_branch[song_branch]     song_branch --&gt; C1[C1]     C1 --&gt; C2[C2] </pre>
----	---	---

iii. on retourne sur la branche «master» ; le fichier Tap\_tappe.txt est modifié à nouveau et au commit C3, contient donc le texte suivant (en rouge le texte modifié)<sup>9</sup> :

<pre> &lt;Onstage&gt; New York M C:     You want it right, direct from hell, Spinal Tap! --- Spinal Tap performs Tonight I'm Gonna Rock You Tonight --- David:      We are Spinal Tap from the UK you must be the USA!  &lt;Garden Interview I&gt; Marty:      Let's...uh talk a little bit about the history of the             group. I understand Nigel you and David originally started             the band wuh...back in...when was it... 1964? David:      Well before that we were in different groups, I was in a             group called The Creatures and w-which was a skiffle group. Nigel:      I was in Lovely Lads. David:      Yeah. Nigel:      And then we looked at each other and says well we might as             well join up you know and uh.... David:      So we became The Originals. Nigel:      Right. David:      And we had to change our name actually.... Nigel:      Well there was, there was another group in the East End             called The Originals and we had to rename ourselves.             The New Originals. Nigel:      The New Originals and then, uh, they became.... David:      The Regulars, they changed their name back to The Regulars             and we thought well, we could go back to The Originals but             what's the point? Nigel:      We became The Thamesmen at that point. </pre>	 <pre> graph TD     HEAD[HEAD] --&gt; master["master"]     master --&gt; C3((C3))     C3 --&gt; CI((CI))     C3 --&gt; C2((C2))     CI --&gt; songbranch["song_branch"]     C2 --&gt; songbranch </pre>
--	---

On veut maintenant fusionner la branche «song\_branch» dans «master» et faire le commit C4 ci-contre.

Si on tente une fusion comme dans la [partie précédente](#), on obtient un conflit. Voici comment le résoudre avec Git et avec TortoiseGit.



<sup>9</sup> notez que le texte modifié en C2 n'est pas contenu ici puisqu'il est modifié dans une autre branche

## Git

1. on tente un *merge*

```
$ git checkout main_branch e.g. «master» : $ git checkout master
$ git merge merged_branch e.g. «song_branch» : $ git merge song_branch
```

on obtient une réponse semblable en cas de conflit :

```
Auto-merging Tap_tappe.txt
CONFLICT (content): Merge conflict in Tap_tappe.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git ne sait pas dans quel ordre il doit fusionner les modifications des deux branches (doit-il mettre d'abord les paroles de Tonight ou la Garden Interview ?)

Si on ouvre le fichier, il contient des insertions lors du merge qu'il faut éditer. Par exemple, le fichier Tap\_tappe.txt ressemble désormais à :

```
<Onstage>
New York M C:
    You want it right, direct from hell, Spinal Tap!

--- Spinal Tap performs Tonight I'm Gonna Rock You Tonight ---
```

David: We are Spinal Tap from the UK you must be the USA!

```
<<<<< HEAD
<Garden Interview I>
Marty: Let's...uh talk a little bit about the history of the
group. I understand Nigel you and David originally started
the band wuh...back in...when was it... 1964?
David: Well before that we were in different groups, I was in a
group called The Creatures and w-which was a skiffle group.
Nigel: I was in Lovely Lads.
David: Yeah.
Nigel: And then we looked at each other and says well we might as
well join up you know and uh....
David: So we became The Originals.
Nigel: Right.
David: And we had to change our name actually....
Nigel: Well there was, there was another group in the East End
called The Originals and we had to rename ourselves.
David: The New Originals.
Nigel: The New Originals and then, uh, they became....
David: The Regulars, they changed their name back to The Regulars
and we thought well, we could go back to The Originals but
what's the point?
Nigel: We became The Thamesmen at that point.
=====
--- Tonight I'm Gonna Rock You Tonight (lyrics) ---
```

## Git

```

Little girl, it's a great big world but there's only one of me
You can't touch 'cause I cost too much but
Tonight I'm gonna rock you (Tonight I'm gunna rock you)
Yeah tonight I'm gonna rock you (Tonight I'm gunna rock you)
Tonight!

You're sweet but you're just four feet
And you still got your baby teeth
You're too young and I'm too well hung
Tonight I'm gonna rock you (Tonight I'm gunna rock you)
Yeah onight I'm gonna rock you (Tonight I'm gunna rock you)
Tonight!

Whoa yeah

You're hot, you take all we got, not a dry
seat in the house
Next day, we'll be on our way
Tonight we're gonna rock you (Tonight we're gunna rock you)
Yeah tonight we're gunna rock you (Tonight we're gunna rock you)
Tonight!

Chorus:
Little girl, it's a great big world, but there's
only one of - meeeee

--- end of the song ---
>>>>> song_branch

```

2. On voit que Git a inséré dans le fichier, le texte suivant :

- ▶ la partie commune,
- ▶ la ligne : <<<<< HEAD
- ▶ les modifications en conflit de la branche active,
- ▶ la ligne : ======
- ▶ les modifications en conflit de la branche à fusionner,
- ▶ la ligne : >>>>> song\_branch

À cause du conflit, l'auto-commit n'a pas été fait. Il nous faut éditer le conflit et faire le commit à la main. Nous voulons par exemple mettre les paroles de la chanson avant la Garden Interview (avec des coupes pour des raisons de place) et bien sûr enlever les lignes insérées par Git :

## Git

```
<Onstage>
New York MC:
    You want it right, direct from hell, Spinal Tap!

--- Spinal Tap performs Tonight I'm Gonna Rock You Tonight ---

David:      We are Spinal Tap from the UK you must be the USA!

--- Tonight I'm Gonna Rock You Tonight (lyrics) ---

Little girl, it's a great big world but there's only one of me
You can't touch 'cause I cost too much but
Tonight I'm gonna rock you (Tonight I'm gunna rock you)
Yeah tonight I'm gonna rock you (Tonight I'm gunna rock you)
Tonight!

[ ... ]

Chorus:
Little girl, it's a great big world, but there's
only one of - meeeee

--- end of the song ---


<Garden Interview I>
Marty:      Let's...uh talk a little bit about the history of the
            group. I understand Nigel you and David originally started
            the band wuh...back in...when was it... 1964?
David:      Well before that we were in different groups, I was in a
            group called The Creatures and w-which was a skiffle group.

[ ... ]

David:      The Regulars, they changed their name back to The Regulars
            and we thought well, we could go back to The Originals but
            what's the point?
Nigel:      We became The Thamesmen at that point.

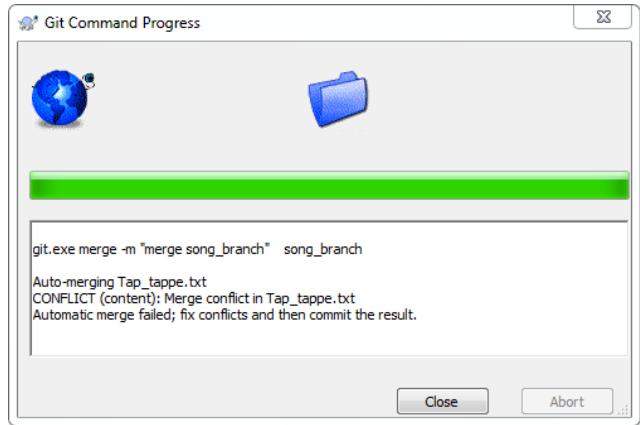
3. On peut maintenant faire le commit :
$ git commit -a -m «merged song_branch»

Le conflit est résolu et la fusion réalisée et dans le nouveau commit C4.
```

1. On tente une [fusion](#).  
on obtient une réponse semblable en cas de conflit :

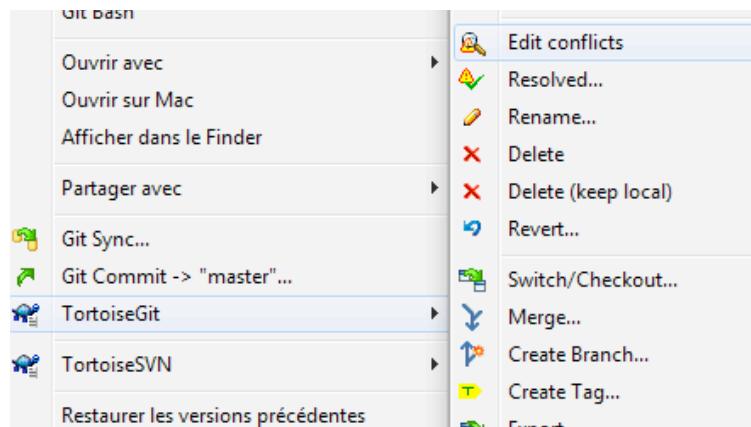
```
Auto-merging Tap_tappe.txt
CONFLICT (content): Merge
conflict in Tap_tappe.txt
Automatic merge failed; fix
conflicts and then commit the
result.
```

Git ne sait pas dans quel ordre il doit fusionner les modifications des deux branches (doit-il mettre d'abord les paroles de Tonight ou la Garden Interview ?).

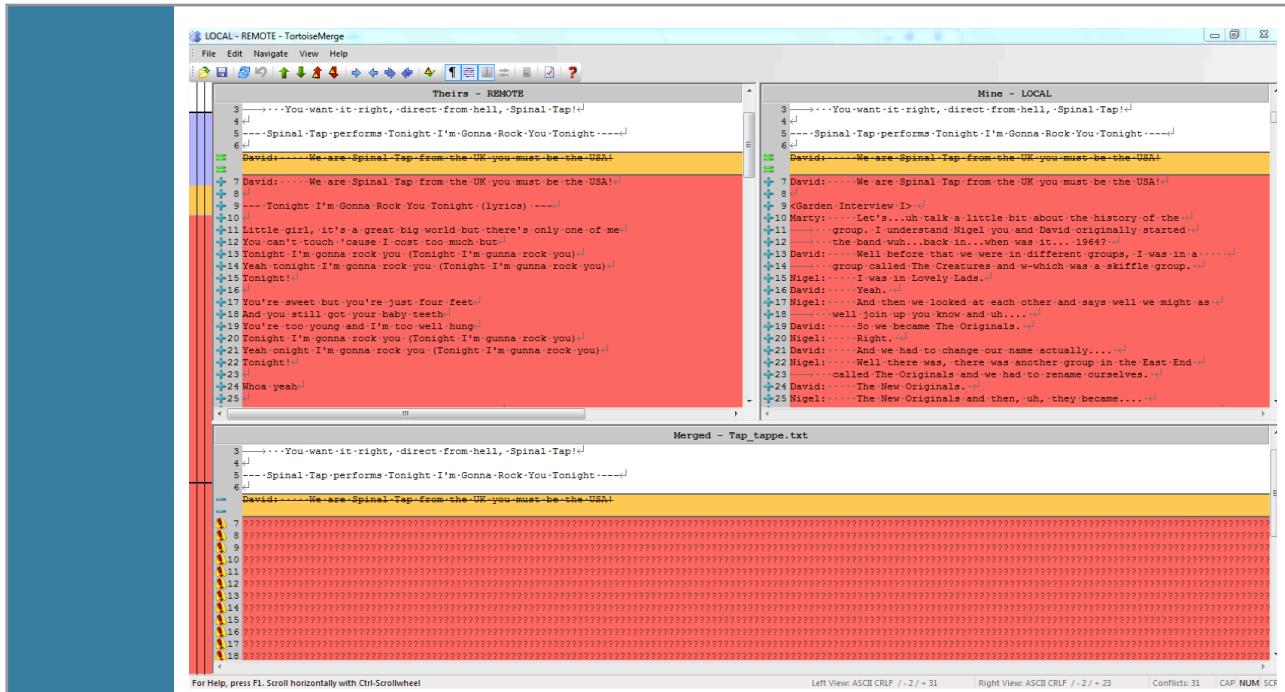


## TortoiseGit

2. Pour éditer les conflits, il faut éviter de faire comme pour Git, sinon TortoiseGit fait un peu n'importe quoi. Il vaut mieux passer par l'éditeur de conflits. Pour cela, faites un clic droit sur le fichier à éditer, puis cliquer sur *TortoiseGit* -> *Edit conflicts...*



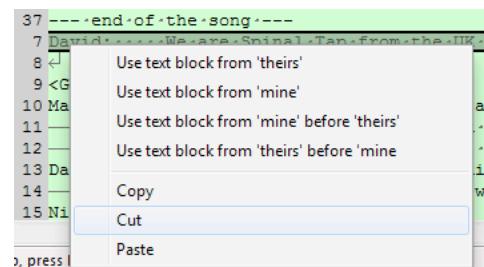
L'éditeur de conflits par défaut s'ouvre. Il présente trois zones de texte : à gauche, la version sur la branche à fusionner, à droite, la version sur la branche active, en bas, le résultat de votre édition.



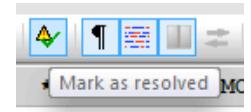
## TortoiseGit

Les zones en rouge (également schématisées sur l'ensemble du fichier dans la colonne de gauche), sont les zones en conflits. On peut par exemple ensuite, choisir de récupérer tout le bloc sur la branche à fusionner d'abord, grâce à clic droit «*Use text block from 'theirs' before 'mine'*», '*theirs*' représente la branche à fusionner et '*mine*' la branche active.

On peut affiner l'édition. En effet, dans notre exemple, la première ligne de chaque bloc était la même et a donc été copiée deux fois. On peut éditer la zone de texte du bas qui montre le résultat, à l'aide des commandes classiques d'insertion, de copie, de collage, etc...

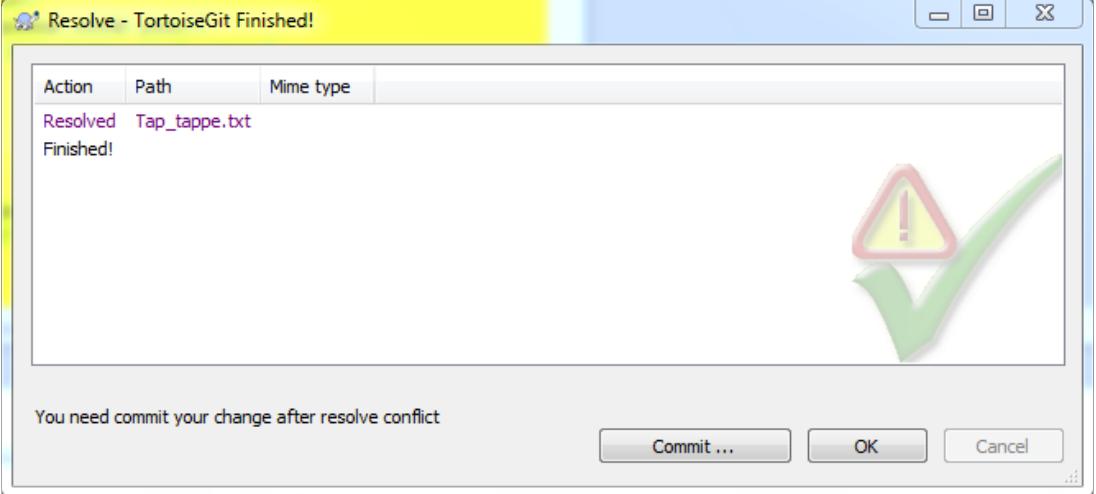


Il ne faut surtout pas oublier de préciser ensuite que l'édition des conflits est terminée en cliquant sur «*Mark as resolved*» dans la barre de tâches.



Le fichier est alors considéré comme résolu et modifié. Il a perdu sa méchante pastille pour la pastille rouge avec un point d'exclamation. Il peut être maintenant «**commité**».

**TortoiseGit**



The screenshot shows a TortoiseGit dialog titled "Resolve - TortoiseGit Finished!". It contains a table with columns "Action", "Path", and "Mime type". One row shows "Resolved" for "Tap\_tappe.txt". Below the table, the word "Finished!" is displayed. To the right of the table is a graphic featuring a yellow warning sign with an exclamation mark and a green checkmark. A message at the bottom of the dialog reads "You need commit your change after resolve conflict". At the bottom right are buttons for "Commit ...", "OK", and "Cancel".

3. [commit](#) comme précédemment.  
 note : il peut se faire depuis la fenêtre précédente, mais s'il y a plusieurs fichiers en conflit, il faudra d'abord résoudre tous les conflits).

#### 4.4. Synchronisation avec un repository distant

Pourquoi cette partie, qui semble si importante (il s'agit quand même du partage du travail avec les autres collaborateurs !) arrive aussi tard dans ce document ? Pour deux raisons principales :

1. avec Git, la quasi-totalité des opérations se passent en local (y compris la création de branches),
2. la récupération du travail des collaborateurs se fait via les branches distantes de Git.

Il était donc nécessaire d'avoir compris la gestion de Git en local et la gestion des branches avant de parler de la synchronisation avec le repository distant.

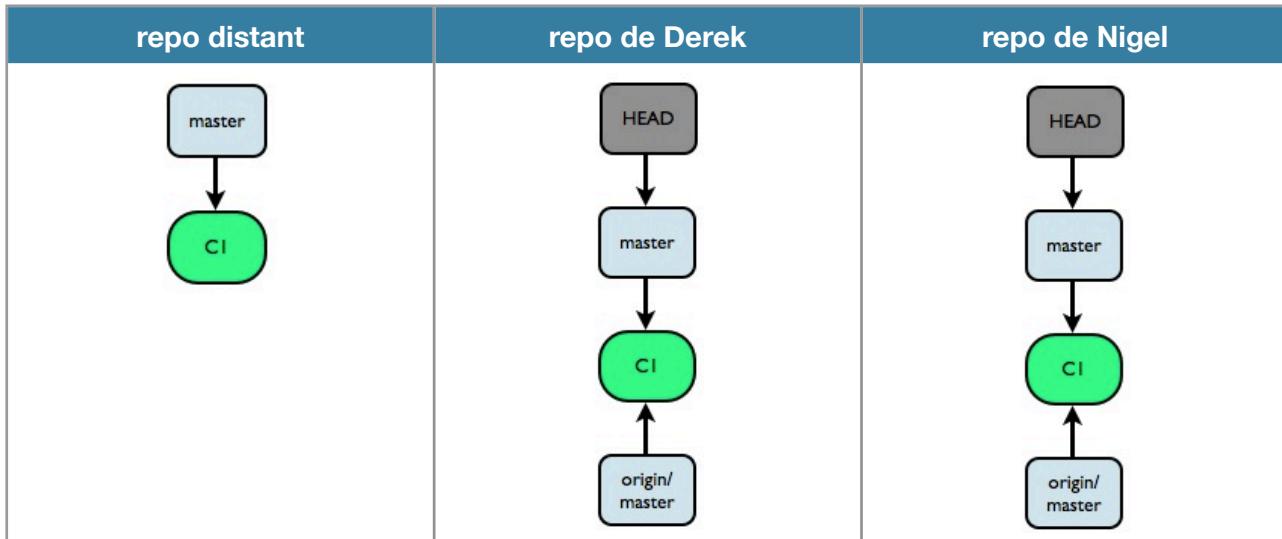
Cette partie présente comment soumettre son travail dans le cas où personne n'a rien soumis depuis votre dernière mise à jour ; comment récupérer le travail des collaborateurs dans le cas où vous n'avez rien soumis depuis votre dernière mise à jour ; comment soumettre et récupérer le travail des autres collaborateurs en cas de conflits (vous avez fait des mises à jour et/ou quelqu'un a fait des mises à jour) grâce aux branches distantes et à la fusion de branches<sup>10</sup>.

Pour l'ensemble de ces parties, nous utiliserons le cas suivant. Il existe un repository distant déjà créé contenant un fichier sur la branche **master**. Ce fichier est modifié par deux utilisateurs : Derek Smalls et Nigel Tufnel.

---

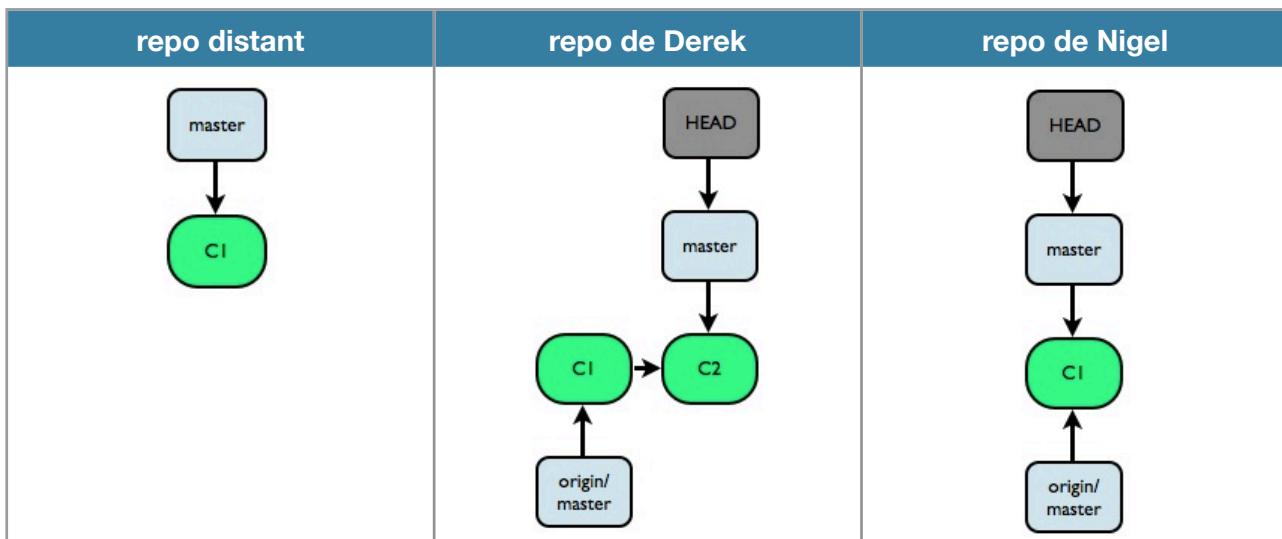
<sup>10</sup> Nous ne travaillerons ici qu'avec un seul repository distant, mais sachez qu'il est possible d'avoir plusieurs repositories distants pour un même projet.

La synchronisation avec le repository distant se fait à l'aide de branches (*remote branches*). Par exemple, la branche *master* du repository distant (*origin*) est représentée en local par *origin/master*. Attention, ce pointeur *origin/master*, correspond à l'état de la branche du repository distant au moment de la dernière mise à jour. Dans la suite de cet exemple, nous représenterons donc l'évolution des repository de Derek, de Nigel et distant. Au démarrage, puisque les 2 utilisateurs sont à jour, les repositories ressemblent à ça.

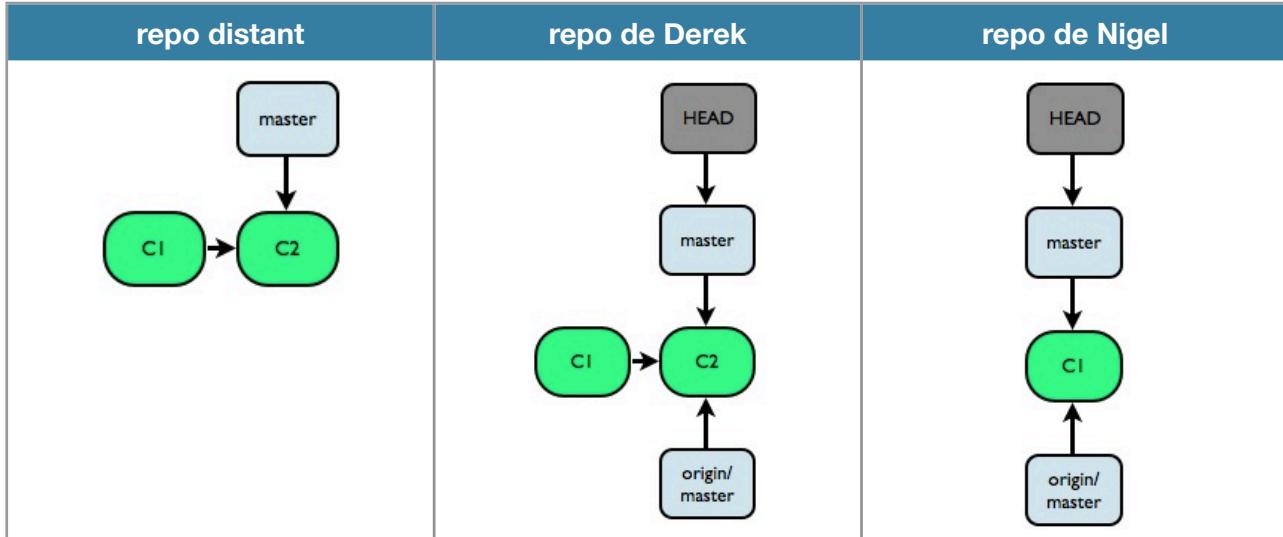


#### 4.4.1. Soumettre son travail sur le repository distant

Derek Smalls modifie ce fichier sur son repository local. Il fait un *commit* sur sa branche «*master*» locale (C2). Les repositories ressemblent donc à ça.



Il souhaite partager son travail sur le repository distant. Pour cela, il fait un *push*. Le repository est mis à jour, ainsi que la branche distante *origin/master* de Derek, mais pas celle de Nigel qui n'a rien fait.



**Git**

Pour faire un *push*, il faut utiliser la commande suivante :

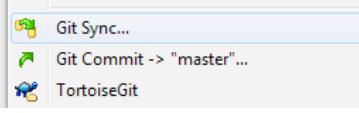
```
$ git push [remote-name] [branch-name]
```

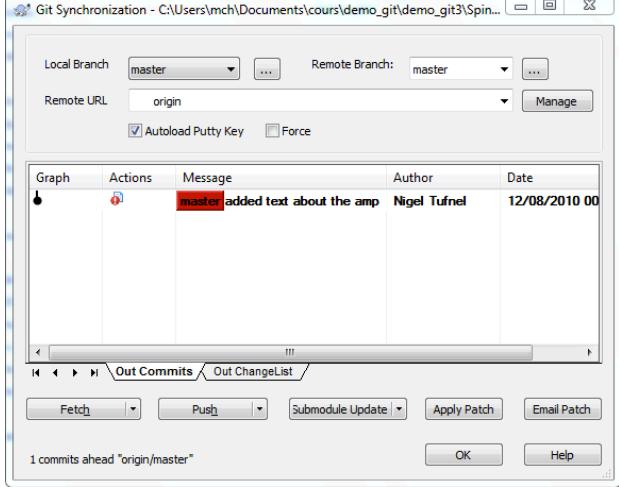
où *remote-name* correspond au serveur distant (e.g. *origin* est attribué automatiquement à votre repository distant au moment du clone, mais comme il peut y avoir plusieurs repositories, il faut pouvoir préciser) et *branch-name* à votre branche locale que vous voulez «pousser» sur le repository distant. e.g. `$ git push origin master` pour pousser la branche «*master*» sur le repository distant

Cette opération se passe bien s'il n'y a pas eu d'autres apports des autres collaborateurs (ici Nigel) sur le repository. Dans le cas contraire, il faut d'abord mettre à jour votre repository, résoudre les éventuels conflits, puis pousser à nouveau.

**TortoiseGit**

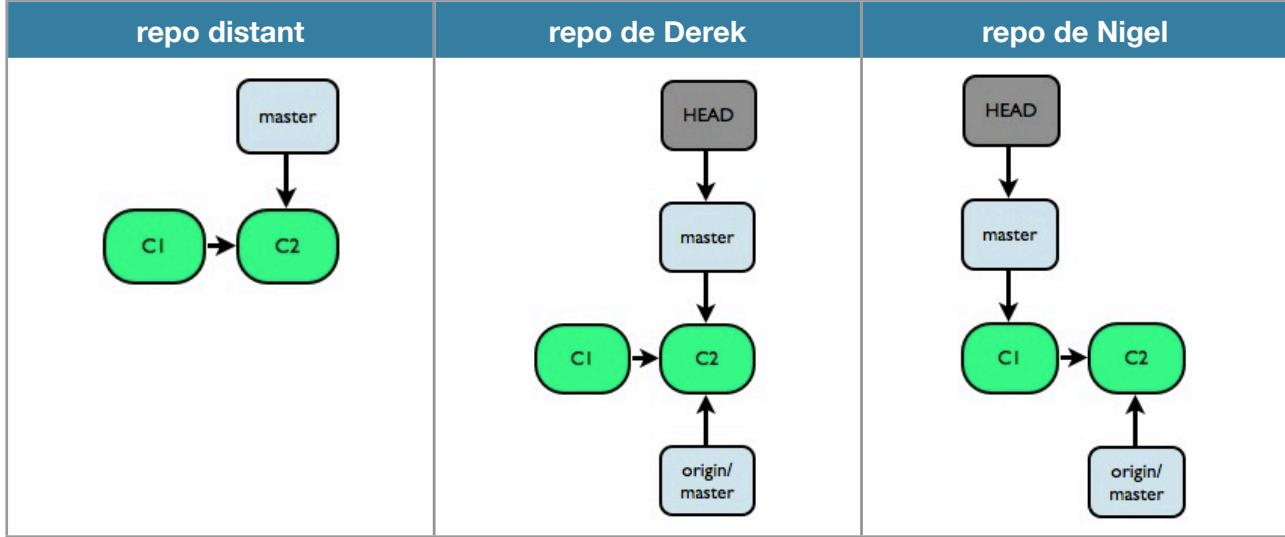
Dans la boîte de dialogue qui s'ouvre, il suffit de cliquer sur *Push*, après avoir vérifié que la branche locale et la branche distante sélectionnées sont bien celle depuis laquelle on veut faire le *Push* (les données poussées doivent avoir été *commitées* avant !) et celle qui doit recevoir les données.



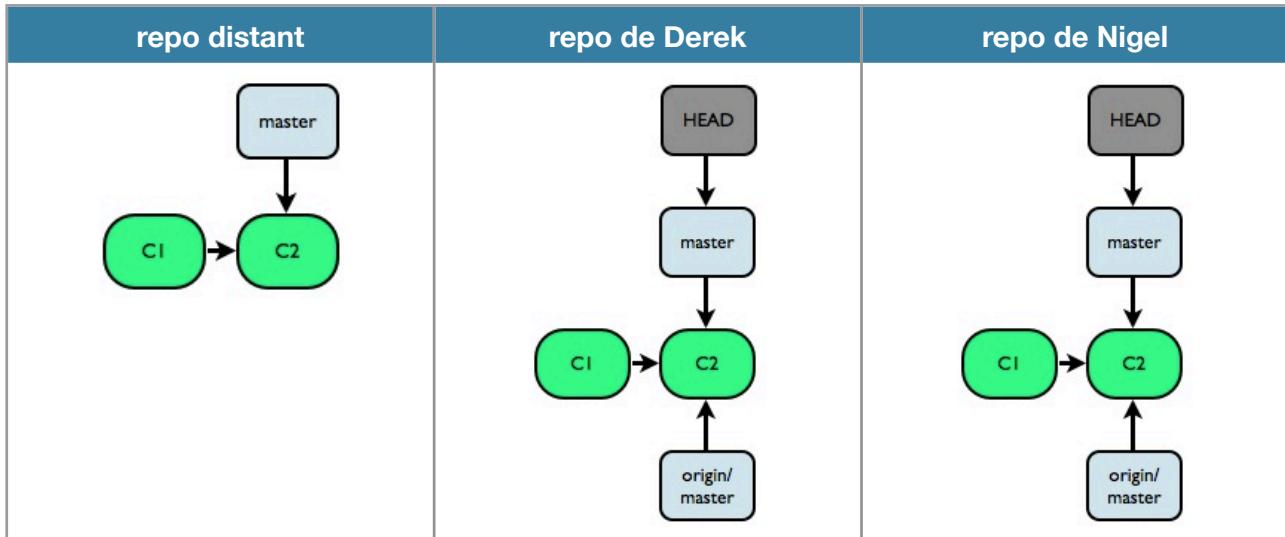


#### 4.4.2. Récupérer le travail des collaborateurs

Nigel veut maintenant récupérer les modifications de Derek. Pour cela, il fait un *fetch origin* qui met à jour la branche distante *origin/master* en local. Sa branche locale *master*, elle, ne bouge pas.



Pour que sa branche *master* soit mise à jour, il doit fusionner les deux branches *master* et *origin/master*. Cette fusion ne pose aucun problème puisque *master* pointe toujours sur un ancêtre de *origin/master*. En d'autres termes, ça ne pose aucun problème parce que Nigel n'a pas fait de *commits* entre sa dernière mise à jour et le *fetch*. Après la fusion, les repositories ressemblent donc à ça.



## 2 étapes : *fetch* et *merge*

1. Pour faire un *fetch*, il faut utiliser la commande suivante :

```
$ git fetch [remote-name]
```

où *remote-name* correspond au serveur distant (e.g. *origin* est attribué automatiquement à votre repository distant au moment du *clone*, mais comme il peut y avoir plusieurs repositories).

e.g. `$ git fetch origin` pour mettre à jour la branche «*origin/master*» en local

## Git

2. Il faut ensuite fusionner les deux branches : pour cela, on se place d'abord dans la branche qui doit recevoir (e.g. *master*) et on fusionne (puis *commit*) :

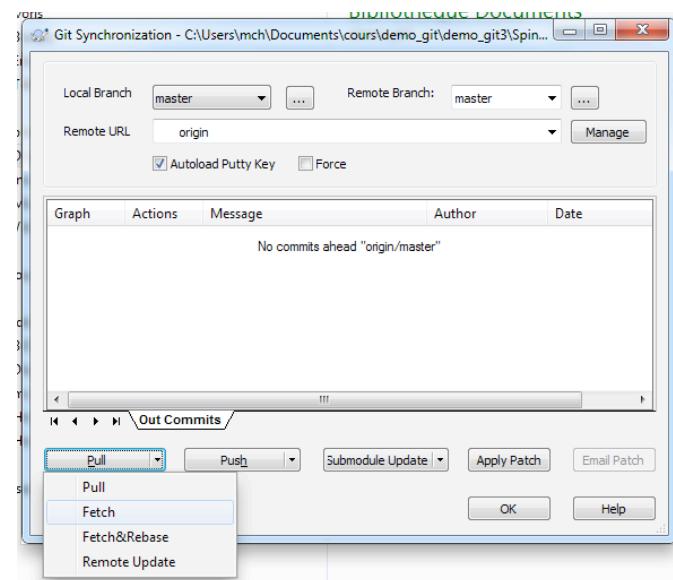
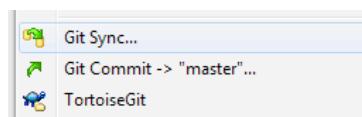
```
$ git merge [remote-branch]
```

e.g. `$ git merge origin/master`

Cette opération se passe bien s'il n'y a pas de conflits, c'est-à-dire si le collaborateur qui fait le fetch n'a pas fait de modifications générant de conflits.

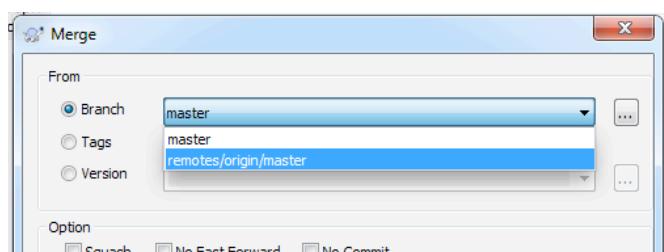
## 2 étapes : *fetch* et *merge*

1. Pour faire un *fetch*, il faut faire un clic droit sur le dossier versionné et cliquer sur *Git Sync...*



Dans la boîte de dialogue qui s'ouvre, il suffit de cliquer sur *Fetch*, après avoir vérifié que la branche locale et la branche distante sélectionnées sont bien celle depuis laquelle on veut transférer les données et celle qui doit les recevoir.

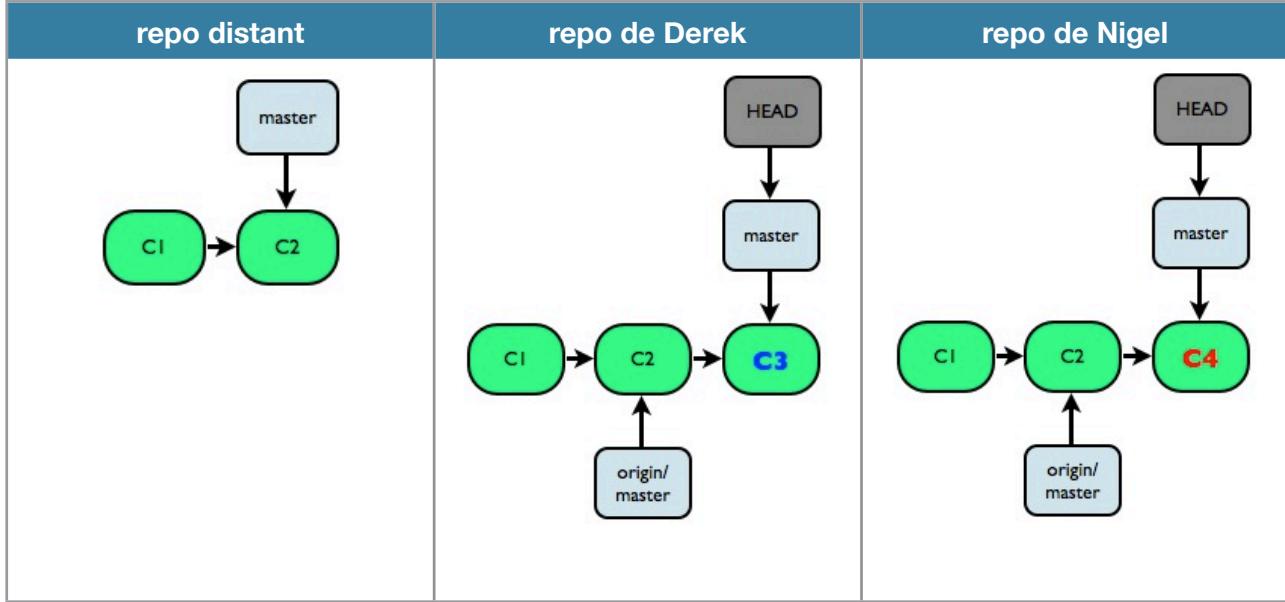
2. fusion : celle-ci se passe comme présenté [précédemment](#). Il faut juste penser à sélectionner la branche distante (une branche dont le nom est précédé de *remotes/* e.g. *remotes/origin/master*). Le commit est fait automatiquement.



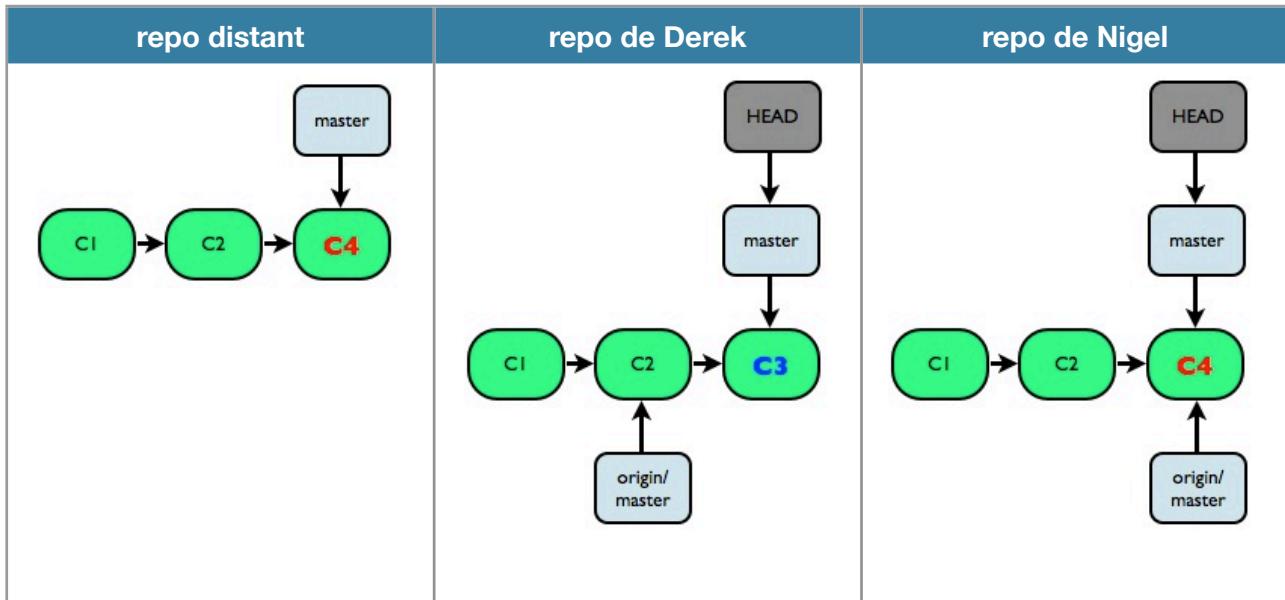
## TortoiseGit

#### 4.4.3. Gestion des conflits lors de l'utilisation d'un repository distant

Maintenant, Derek et Nigel modifient chacun de leur côté le fichier. Les repositories évoluent différemment. En l'absence de push, le repository distant n'évolue pas.

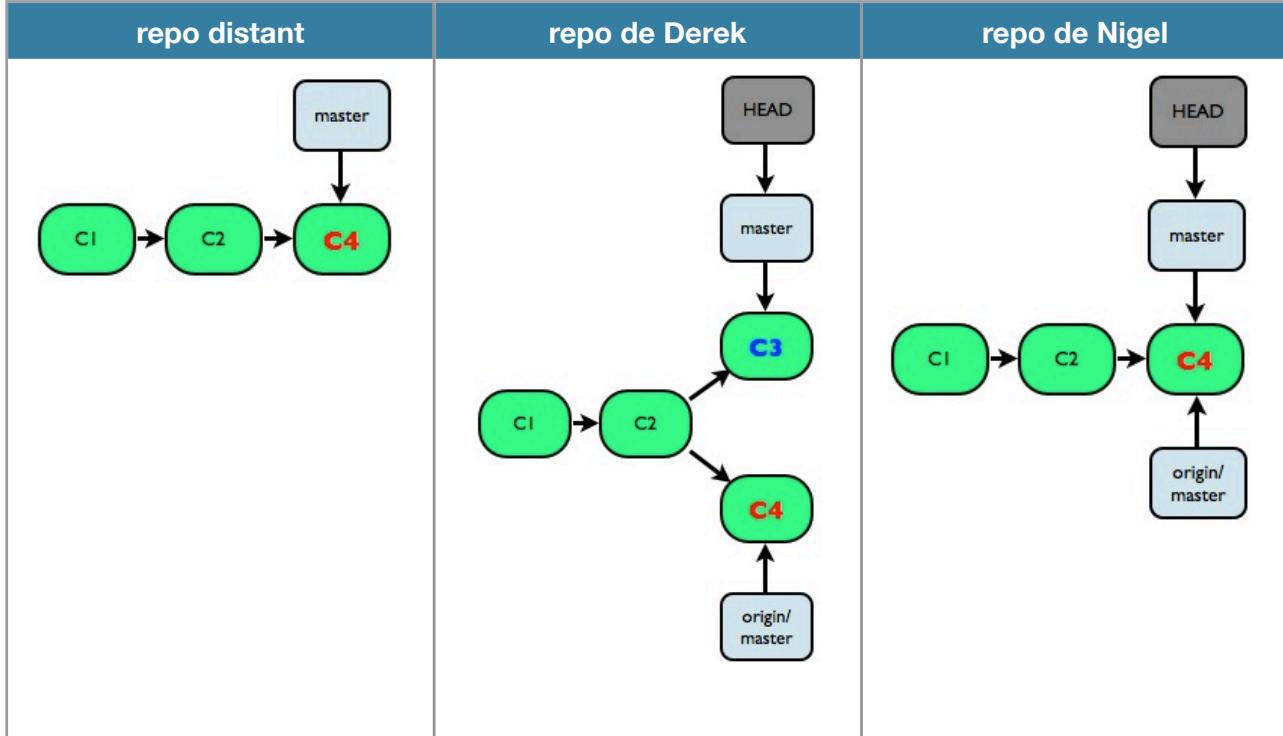


Nigel «pousse» son travail sur le serveur via un *push*. Le repository distant est modifié. Celui de Nigel aussi et est à jour. En revanche, la branche *origin/master* de Derek n'est plus à jour mais il ne le sait pas encore.

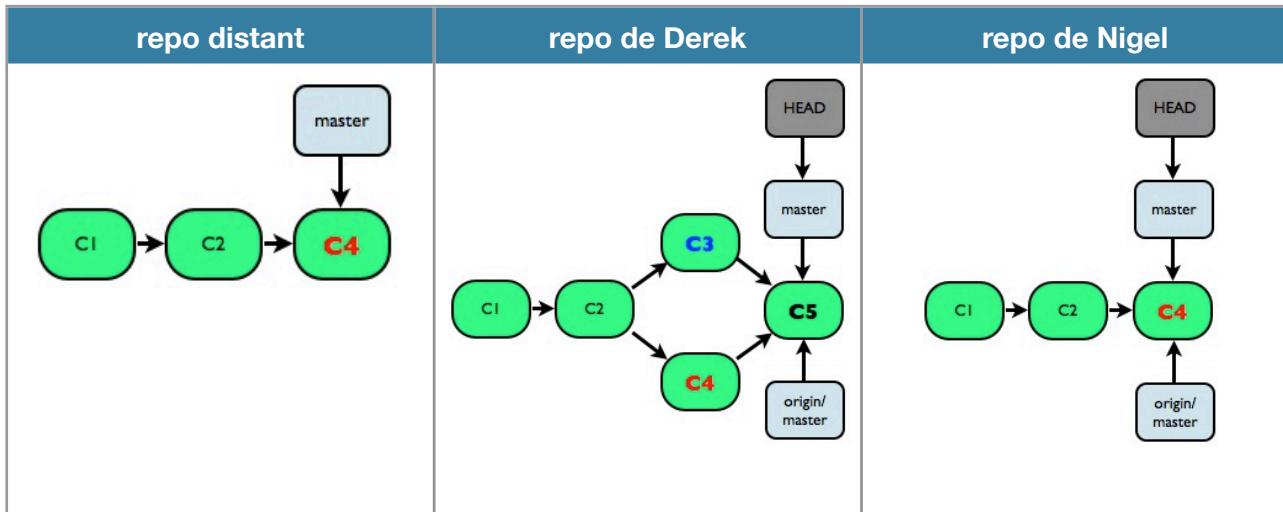


Derek a terminé son travail et souhaite le pousser à son tour. Il fait un *push* mais se voit refuser par le serveur, qui lui demande de faire d'abord un *fetch*, de fusionner ses données, puis de faire un *commit* et de repousser à nouveau.

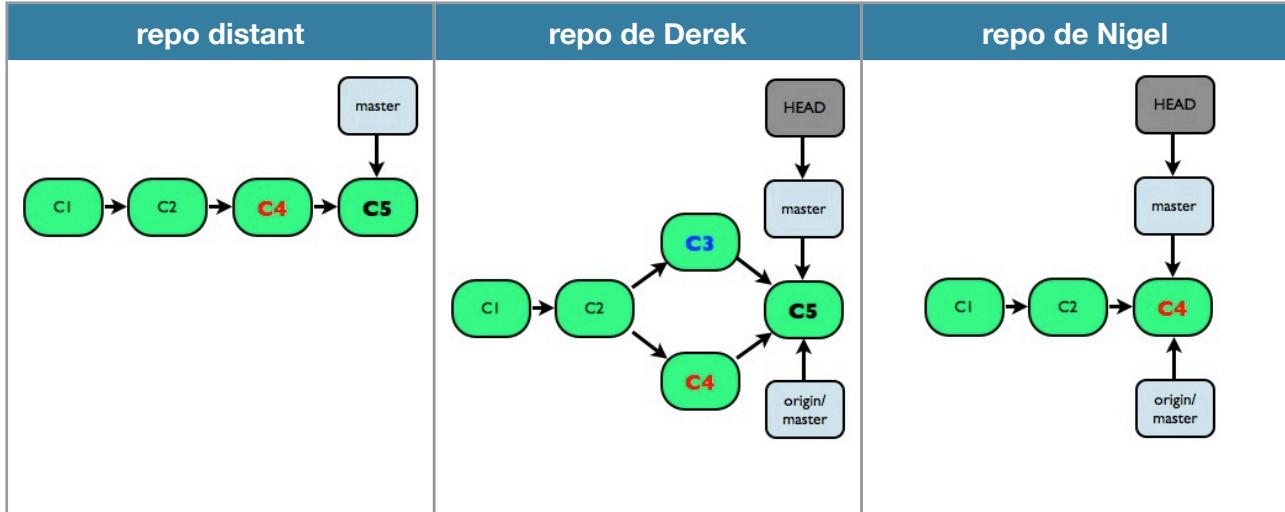
Derek fait alors un *fetch*, et obtient le repository suivant.



Il obtient des conflits sur le fichier modifié par Nigel au commit C4 et ses propres modifications du commit C3. Après avoir géré les conflits, fusionné les branches et fait le commit C5, son repository ressemble à :



Il peut cette fois-ci faire son *push*, puisqu'il est à jour avec le repository distant.



## 4.5. Autres fonctionnalités

Cette partie a pour but de présenter quelques fonctionnalités supplémentaires mais secondaires de Git. Elle sera remplie dans des futures versions en fonction des besoins.

### 4.5.1. Tag

À suivre...

### 4.5.2. Rebase

À suivre...

### 4.5.3. Pull

À suivre...

## 5. Méthodes d'utilisation collaboratives de Git

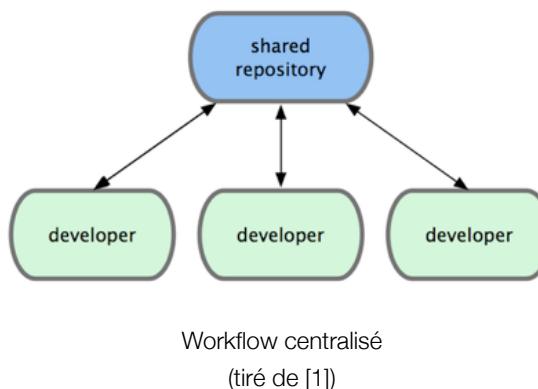
L'utilisation de systèmes de contrôle de version permet le travail collaboratif. Celui-ci doit tout de même s'organiser. Il existe de nombreuses solutions d'organisation. Scott Chacon en cite notamment trois intéressantes<sup>11</sup> dans son livre Pro Git [1] :

- Centralized workflow,
- Integration-Manager workflow,
- Dictator and Lieutenants workflow.

Dans le cadre des cours à l'IUT d'Informatique, nous utiliserons uniquement un workflow centralisé.

### 5.1. Workflow centralisé

Il est constitué d'un noyau central sur le serveur, le repository distant. Il n'y en a qu'un, et chaque collaborateur a un rôle équivalent à celui des autres. Il n'y a pas de collaborateur avec un pouvoir plus important qu'un autre. On retrouve donc le même cas de figure que l'exemple de gestion de conflits en repository distants. Ce workflow est parfaitement adapté aux petites équipes.



Dans le cadre du cours de C# et d'XML en GI, les équipes seront constituées de 4 éléments, ce qui justifie l'utilisation de ce workflow.

### 5.2. Quelques règles de bonne conduite

Cette partie contient un ensemble de règles et de conventions à respecter pour une utilisation en groupe de d'un système de contrôle de version.

- Le projet stocké dans le repository distant (et encore plus, celui de la branche *master*), doit **toujours** être compilable.
  - Avant de faire un *push*, vérifiez toujours que votre projet compile sur votre copie.
  - Si vous n'êtes pas sûr de vous, faites un *clone* dans un dossier temporaire et vérifiez que votre *push* compile.
  - Si vous devez faire exceptionnellement un *push* qui ne compile pas, prévenez tous les autres utilisateurs.
- Faites des mises à jour (*fetch* + *merge*) très régulièrement.

<sup>11</sup> <http://progit.org/book/ch5-1.html>

- ➔ Faites des *push* réguliers.
  - ▶ Si vous gardez trop longtemps des fichiers extraits modifiés, vous augmentez le risque de conflits qui seront de plus en plus compliqués à résoudre.
  - ▶ Sans *push* réguliers, vous faites perdre à tous le bénéfice de l'historique.
- ➔ Utilisez le header suivant au début de chaque fichier

```
// =====
// Copyright (C) 2008-2009 IUT CLERMONT1 - UNIVERSITE D'AUVERGNE
// www.iut.u-clermont1.fr
//
// Module      : Board - source file
// Author       : Marc Chevaldonné
// Creation date: 2008-11-25
//
// =====
```