Git в картинках

Автор: ***Игорь Ткачев***

Опубликовано: *23.05.2011*
Исправлено: *10.12.2016*
Версия текста: *1.0*

Fix that bug NOW!
© your boss

Введение

Внедрение новых технологий и инструментов в массы обычно происходит по одному сценарию. В самом начале небольшая группа энтузиастов рассказывает всему сообществу о том, что наконец-то у мира появился шанс, спаситель пришёл, и он нереально крут. При этом объяснить, почему он крут и что же именно в нём такого нереального, они ещё не готовы. Тем не менее, зачастую начавшееся оживление приводит к бурному развитию мессии, и он доводится до состояния, когда им уже можно начинать пользоваться не только апостолам и законченным фанатам, но и простым смертным. Смертные, кто добровольно, кто из-под палки, но все обязательно через ломку сознания постепенно обращаются в новую веру, и вот спаситель уже вовсю шагает по планете, одаривая своих приверженцев счастливыми мгновениями соприкосновения с добрым и вечным, вовлекая в их ряды всё новых и новых счастливцев. Затем к нему привыкают. Потом находят в нём фатальные недостатки. И вот на горизонте появляется новый спаситель и всё повторяется сначала.

Мессия, о котором пойдёт речь в данном руководстве – это Git, распределённая система контроля версий с открытым исходным кодом. На данном этапе его развития он уже доступен простым смертным.

Почему Git?

Давайте с самого начала выясним, действительно ли Git так крут, почему мы должны тратить на его изучение своё драгоценное время, и что такого принципиально нового нам может дать переход на него. Ниже приведён основной декларируемый список преимуществ Git в сравнении с Subversion (сравнение именно с SVN в данном случае не принципиально, можно было бы взять любую другую аналогичную систему контроля версий):

1. Git гораздо быстрее Subversion.
2. Git репозиторий в десятки раз компактней репозиториев Subversion (примерно в 30 раз меньше на примере проекта Mozilla).
3. Git с самого начала разрабатывался как распределённая система контроля, позволяющая каждому разработчику иметь полный локальный контроль над репозиторием.
4. Ветки (branches) в Git гораздо легче и работают значительно быстрее.

Рассмотрим всё по порядку, но для начала вспомним типичный день пользователя Subversion.

Начиная работать над новой функциональностью или над фиксом очередного бага, мы, прежде всего, синхронизируем свою локальную копию репозитория с сервером. Как правило, работа ведётся в trunk, иногда, если возникает такая острая необходимость, создаётся ветка. Далее от забора и до заката усердно пишется код, после чего его необходимо проверить на работоспособность или хотя бы на компилируемость, и отправить на сервер в центральный репозиторий. Если что-то не складывается, или мы не успеваем закончить, а новый commit нежелателен, так как ломает текущий код в репозитории, то отправка изменений откладывается на следующий день. Что же такого экстраординарного нам может предложить Git в подобной ситуации?

Быстродействие

Предположим, синхронизация с репозиторием Subversion занимает полминуты/минуту, а синхронизация с Git – 5-10 секунд. Насколько это принципиально? Да в принципе, не очень. Ну что такое минута в сутки? Приятно, конечно, но стоит ли менять систему контроля версий только из-за такой возможности? Скорее всего, вряд ли.

Компактность

Размер исходного кода большинства проектов составляет мегабайты, десятки мегабайт, иногда сотни. Современные рабочие станции имеют размер дисковой системы, измеряемый терабайтами. То есть речь идёт о разнице в 4-5 порядков. Принципиально? Не очень. Тем не менее, приятно, конечно, но стоит ли менять систему контроля версий только из-за такой возможности? Скорее всего, вряд ли.

Распределённость

Любой коммит (от англ. commit) в Git производится в локальный репозиторий, а затем эти изменения при необходимости проталкиваются на сервер. Т.е. чтобы сделать коммит на сервер, нужно нажать не одну кнопку, а две. В некоторых случаях это даже удобно. Например, можно лететь в самолёте, при этом писать код и периодически сохранять свои изменения в локальный репозиторий, не соединяясь с центральным сервером. Прикольно, конечно, но в целом удобство такой возможности весьма сомнительно. Как часто вы летаете в самолётах, а как часто при этом пишете код? Стоит ли менять систему контроля версий только из-за такой возможности? Скорее всего, вряд ли.

Ветки

Вообще создание ветки – это всегда геморрой. Если этого можно не делать, то лучше и не надо. Обычно ветки создаются при острой необходимости, когда никакое другое решение уже не помогает. Переход от ветки к ветке зачастую требует перестройки окружения на новый каталог, слияние мелких изменений двух веток лучше всего делать чуть ли не вручную методом copy-paste, а окончательное слияние с trunk, как правило, представляет собой кошмарный ужас в процессе и пышное празднество после его завершения. В общем, настоящий Subversion-индеец предпочитает не связываться с ветками вообще и правильно делает.

Итак, что у нас остаётся в сухом остатке? Получается, что ничего. А раз так, то можно закругляться. Нажмите крестик в правом верхнем углу и закройте окно браузера. Всем спасибо за внимание, до новых встреч!

Хотя постойте! А что значит – ветки гораздо легче и работают значительно быстрее? Как это может что-то принципиально изменить в работе с системой контроля версий?

Вы не поверите, но это может изменить буквально всё! Создание веток, переключение между ними и их слияние в Git похоже на забавную игру. В это трудно поверить человеку изможденному бранчингами Subversion, но это действительно так. Именно с этим нам с вами вскоре предстоит разобраться. Усаживайтесь удобнее, доставайте попкорн и приготовьтесь к ломке сознания, но прежде…

Git vs. Mercurial

Хотя Git и был первым, но на сегодняшний день является не единственной системой контроля версий с соответствующей идеологией. Ближайший его соперник – Mercurial. Добрые языки поговаривают, что Mercurial и быстрее и имеет более дружелюбный интерфейс командной строки. Но давайте разберёмся.

Предположим, что Mercurial быстрее в два раза и позволяет сократить время коммита с пяти секунд до двух с половиной. Как видим, абсолютная величина не играет принципиальной роли. Вот разница между 5 и 55 секундами принципиальна, так как пятидесяти секунд вполне достаточно для того, чтобы ненадолго прерваться, пойти сделать себе кофейку, выкурить электронную сигарету, потрепаться полчаса на кухне с сослуживцами или зависнуть на полдня в каком-нибудь флеймовом форуме RSDN.ru. Пятидесяти секунд достаточно, а двух с половиной – нет, потому это и не принципиально.

Теперь предположим, что интерфейс командной строки Mercurial тоже в два раза дружелюбней, чем командная строка Git. Но проблема в том, что для простого смертного командная строка недружелюбней любого простенького GUI в миллион раз. И если командная строка Git для нас недружелюбнее в миллион раз, то после простых арифметических манипуляций получается, что командная строка Mercurial недружелюбнее всего лишь в 500 000 раз. В общем, в любом случае неинтересно.

Таким образом, получается, что для того, чтобы обеспечить себе максимальное удобство работы необходимо выбирать исходя не столько из возможностей самих инструментов, которые, по сути, не сильно отличаются, но и исходя из наличия визуальных средств работы с ними.

Из существующих GUI для обеих систем можно выделить, пожалуй, Git Extensions для Git и TortoiseHg соответственно для Mercurial. Первый инструмент представляет собой отдельное приложение с очень лёгкой интеграцией с Visual Studio, второй сделан по образу и подобию TortoiseSVN в виде расширения оболочки Windows. Не вдаваясь в подробности, скажу лишь, что то, что русскому хорошо, немцу – смерть. На мой взгляд, работа с репозиторием как со структурой файлов вполне оправдана в случае с Subversion, но совершенно не подходит для репозиториев Git или Mercurial, по той причине, что в Subversion мы работаем с деревом каталогов, а в Git или Mercurial – с деревом коммитов. К тому же разработчики TortoiseHg находятся в состоянии затяжного творческого поиска и когда они из него выйдут совершенно не ясно.

Стоит отметить, что кроме GUI-инструмента никаких других предпочтений в пользу Git или Mercurial у меня нет. Мне нравятся обе системы, и я пользуюсь обеими для разных проектов. Но на текущий момент предпочтение отдано Git по причине наличия более развитого и удобного GUI – Git Extensions.

Устройство Git

Я не буду детально разбирать форматы файлов Git и способы хранения данных в репозитории. Если эта скукотища кому-то интересна, то информации в сети предостаточно, можно начать, например, с этой довольно обстоятельной статьи. Тем не менее, для понимания базовых механизмов работы потребуются некоторые сведения об устройстве Git.

После создания локального репозитория в вашем рабочем каталоге появится скрытый каталог с именем **.git**. Это и есть копия репозитория со всеми потрохами, включая полную историю изменений проекта. Компактность репозитория Git вовсе не преувеличена. На примере проекта BLToolkit:

- размер последней ревизии, упакованной в zip-архив, составляет около десяти мегабайт;
- распакованная ревизия – 29 мегабайт;
- рабочий каталог со всеми промежуточными, отладочными и исполняемыми файлами – 475 мегабайт;
- размер .git-репозитория – 30 мегабайт с историей за несколько лет, более чем в тысячу коммитов.

Совсем неплохо, не правда ли? Такая компактность достигается за счет того, что Git хранит не более одной копии каждой уникальной версии файла. Даже разные файлы с одинаковым содержимым будут представлены в Git одной единственной копией.

Сам по себе репозиторий самодостаточен. Вы можете клонировать его простой командой copy. Центральный репозиторий отличается от персонального лишь тем, что не содержит рабочей ревизии кода, соответственно для него нет необходимости в скрытом каталоге *.git*. Клонировать серверные репозитории также можно простым копированием. По большому счёту все репозитории в Git равноправны, и если вы хотите, чтобы какой-то репозиторий был более репозиторием, чем все остальные репозитории, то это ваше личное административное решение. Никакой технической поддержки для этого не требуется.

Одним из базовых понятий в Git является коммит (commit). В отличие от некоторых систем управления версиями, в Git коммит – это не просто действие или порядковый номер в истории файла. Это полноценный объект, представляющий собой точную копию состояния рабочей ревизии на момент его создания. Любое изменение в истории представляется коммитом. Коммит может иметь ссылки на родительские и дочерние коммиты, образуя, таким образом, дерево коммитов.

Теперь можно рассмотреть, что такое ветка (branch) в Git. Ветка в Git, по своей сути, это именованный коммит, наклейка на нём и ничего более. Когда мы создаём новый коммит, то наклейка с текущего коммита переносится на новый, переходя в результате по наследству от одного коммита к другому. Но нет абсолютно никаких проблем в том, чтобы переставить любую метку в любое другое место дерева коммитов.

Таким образом, упрощённо устройство Git можно представить как дерево коммитов с наклейками, которые по недоразумению почему-то назвали ветками.

А теперь пора перейти к картинкам.

Работа с Git Extensions

Установка

Картинка 1

Скачать программу установки Git Extensions можно по адресу http://code.google.com/p/gitextensions/. Она включает в себя собственно приложение, а также последнюю версию Git for Windows и утилиту сравнения файлов KDiff3. Отметьте обе утилиты для установки, они нам ещё понадобятся.

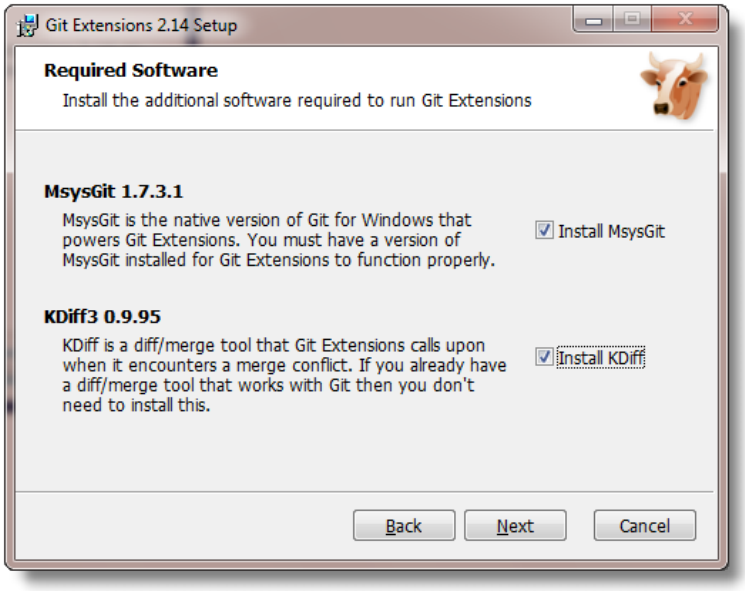

Рисунок 1.

Создание репозитория

Картинка 2

После установки запустите программу. Первым делом она откроет диалог с настройками. Проверьте, всё ли в порядке, введите своё имя и e-mail, и закройте диалог. После этого вы должны увидеть следующую картинку.

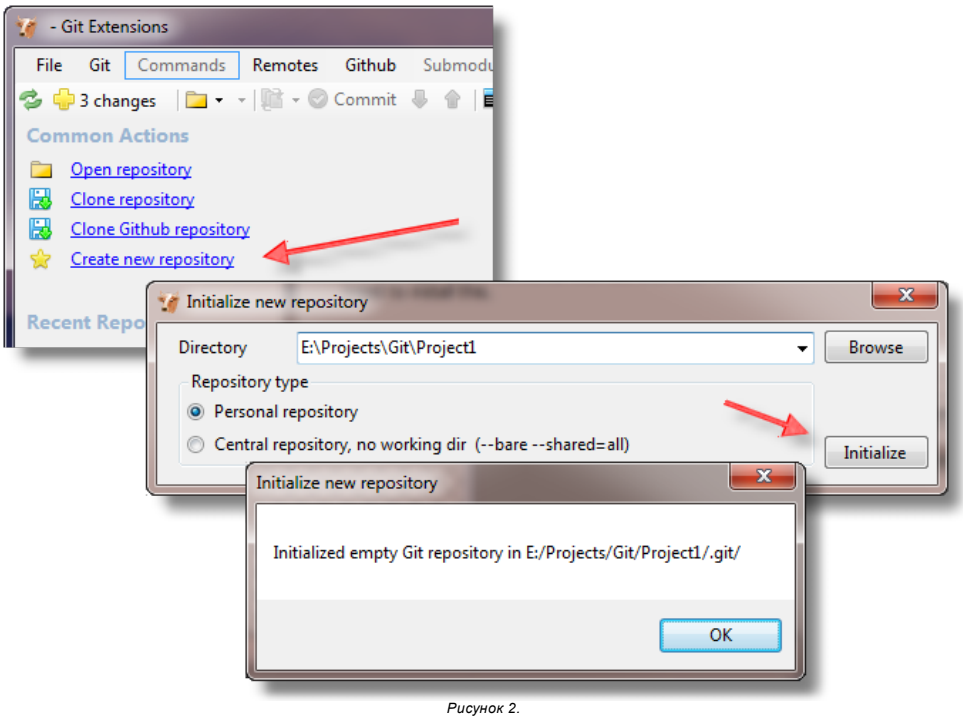

Рисунок 2.

Нажмите ссылку "Create new repository", обозначенную стрелкой на рисунке. Затем в появившемся диалоге введите имя каталога, в котором вы желаете создать новый репозиторий, и нажмите кнопку "Initialize". Поздравляю, вы только что создали новый репозиторий Git. Но это ещё не вся работа по начальной настройке репозитория, которую вам предстоит выполнить. В открывшемся окне вам будет предложено отредактировать файл **.gitignore** и сделать первый коммит. Файл *.gitignore* по своему назначению аналогичен свойству *ignore* Subversion, т.е. в нём задаётся список файлов и масок, которые Git должен игнорировать.

Коммит

Картинка 3

Нажмите кнопку "Edit .gitignore" (1). В появившемся диалоге вы можете добавить свой список файлов и масок, но Git Extensions позволяет сэкономить кучу времени и предлагает добавить исключения по умолчанию (2). Эти исключения специально подобраны для пользователей Visual Studio. Теперь нажмите "Save" (3) и затем "Commit" (4).

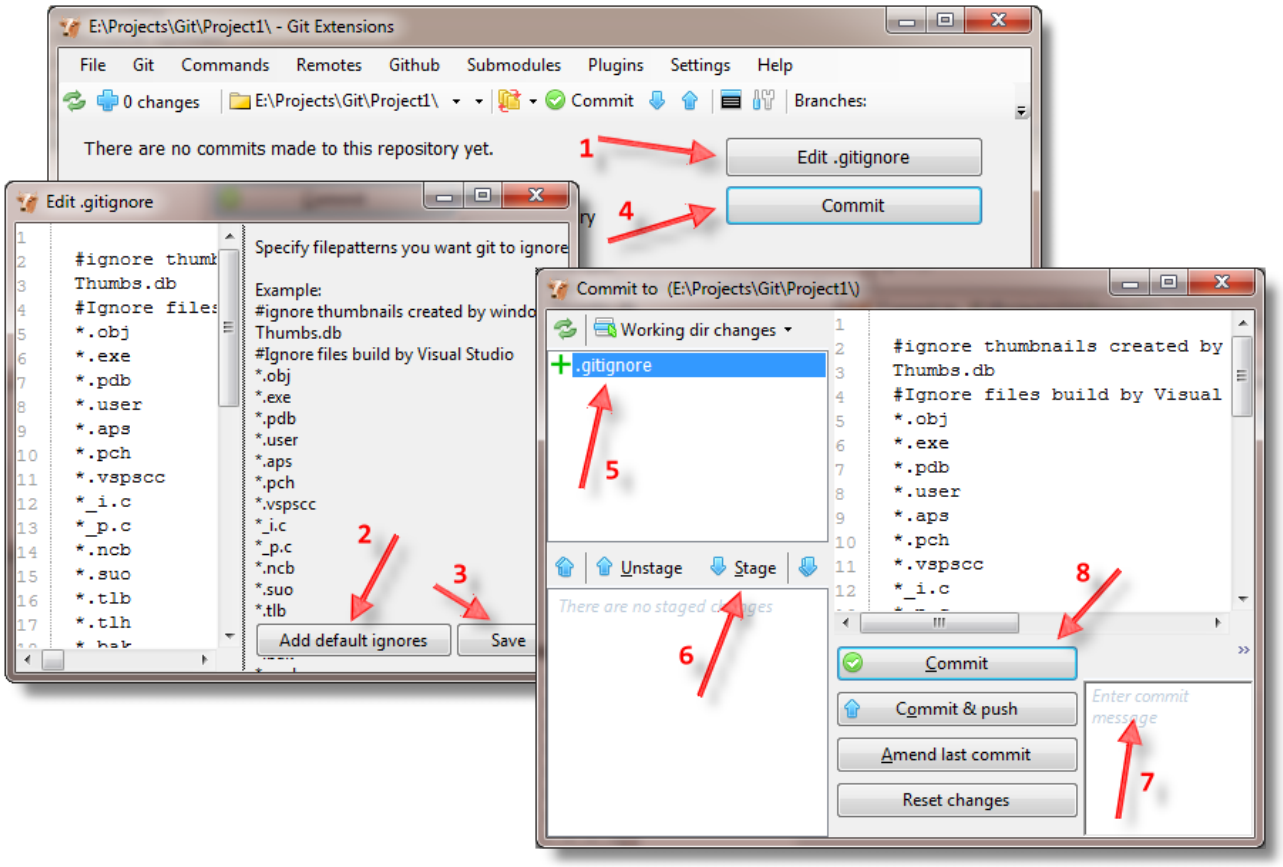

Рисунок 3.

Новое открывшееся окно – это окно создания нового коммита. Здесь мы будем проводить много времени. В верхней левой части окна (5) находятся файлы, которые были распознаны Git как отличные от текущего рабочего коммита. В этом окне можно отменять все или часть изменений в файле, можно добавлять файлы в *.gitignore* или перемещать их в область Stage (6). В коммит попадают только файлы из этой области. В TortoiseSVN что-то подобное делается с помощью пометки галочками файлов, участвующих в коммите. Здесь эту функцию выполняет staging area. В верхнем правом углу находится окно, в котором мы можем просмотреть фактические изменения, которые будут сохранены в репозитории. В нижнем правом окне (7) нужно ввести комментарий к новому коммиту. Когда все приготовления завершены, нажмите кнопку "Commit" (8) и изменения будут сохранены в репозитории.

Теперь давайте добавим в проект какой-нибудь файл и попробуем сделать ещё один коммит. Зайдите в каталог, в котором только что создали репозиторий (в моём примере это E:\Projects\Git\Project1), создайте в нём файл с именем test.txt, вбейте в него строчку "1234" и сохраните.

Картинка 4

Возвратившись в Git Extensions, вы увидите, что картинка изменилась кардинальным образом.

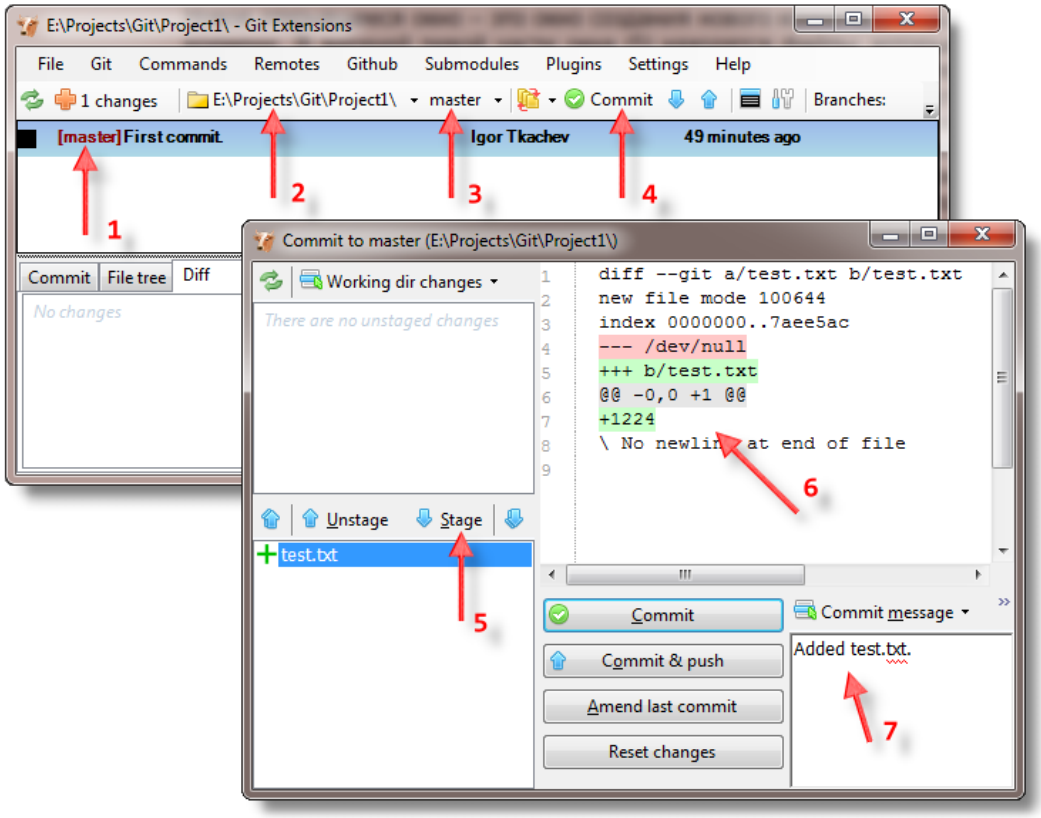

Рисунок 4.

Прежде всего, появилась запись о предыдущем коммите. Тёмно-красная надпись *[master]* (1) в квадратных скобках является именем локальной ветки. Название *[master]* обычно используется для основной ветки проекта. Придерживаться ли этого соглашения, зависит только от вас. Красным цветом отображаются рабочие метки. Позже мы увидим и другие цвета. Стрелочка (2) указывает на текущий репозиторий, точнее, на список, из которого этот репозиторий можно выбрать. Стрелочка (3) указывает на текущую ветку. (4) представляет собой кнопку "Commit". При нажатии на нее появится уже знакомое окно коммита. Добавьте файл *test.txt* в staging area (5) (обратите внимание на окно показа изменений (6)), задайте комментарий (7) и нажмите "Commit".

Картинка 5

После создания нового коммита ветка *[master]* передвинулась на новый коммит (1). В окне Diff (2) можно видеть файлы, которые затронул выбранный коммит, а правее (3) – и сами изменения в этих файлах.

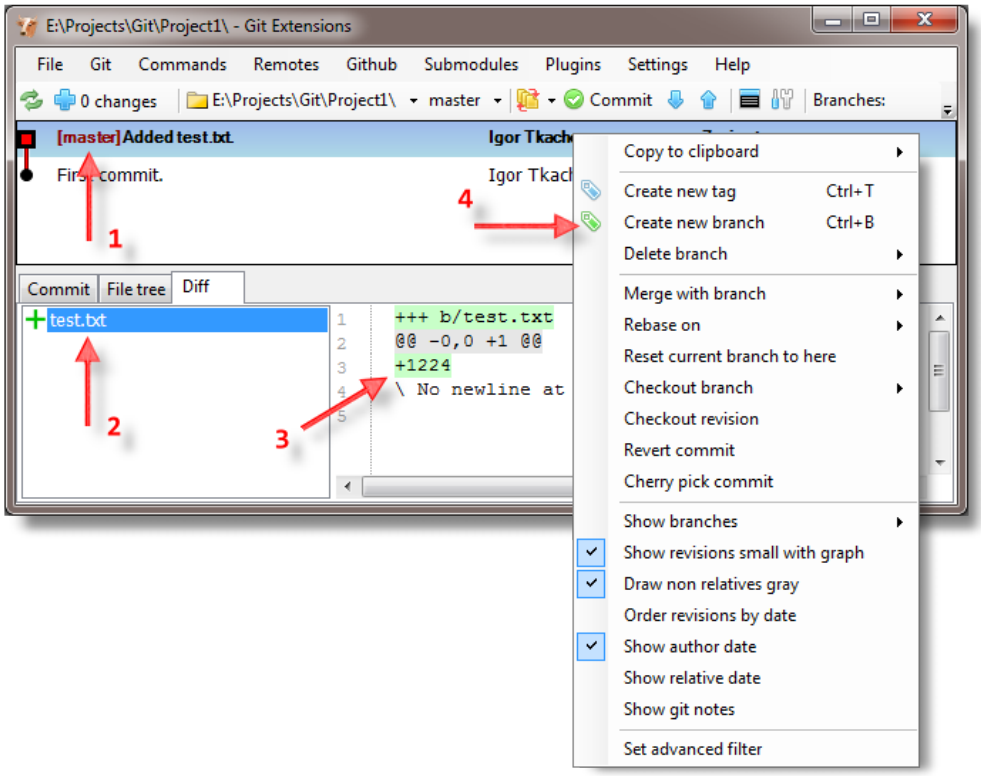

Рисунок 5.

Ветки

Теперь пришло время для ветвлений. Я обещал, что это будет легко и даже в каком-то смысле забавно, давайте в этом убедимся. Выберите интересующий вас коммит (это не должен быть обязательно текущий коммит), вызовите контекстное меню и выберите пункт "*Create new branch*" (4). В открывшемся диалоге введите *[branch1]* и жмите "Create branch". Загляните в рабочий каталог. Файл *test.txt* всё ещё на месте. Что же произошло? Да фактически ничего не произошло. Просто к текущему коммиту приклеилась ещё одна метка.

Картинка 6

Теперь выберите самый первый коммит, и создайте ещё одну ветку. Назовите её *[branch2]*. Смотрим содержимое текущего каталога – файл *test.txt* исчез.

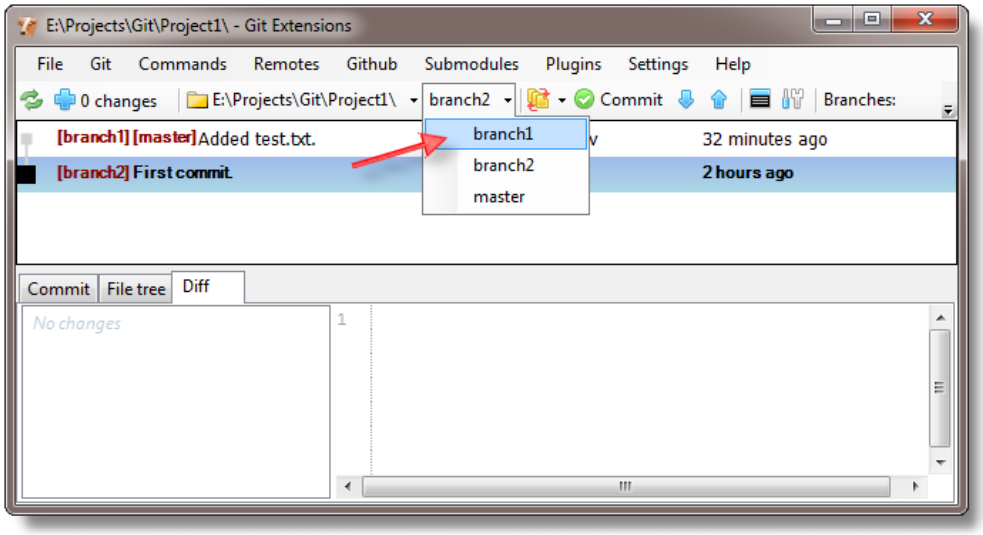

Рисунок 6.

Но, право же, не стоит так отчаиваться! На самом деле наш бесценный файл никуда не исчез. Просто мы создали новую ветку от коммита, в котором этого файла ещё не было, и тут же на неё переключились. Git синхронизировал содержимое рабочего каталога с текущим коммитом, которым теперь стал наш самый первый коммит. Попробуйте переключиться на *[branch1]*, видите, файл появился. Теперь переключитесь на *[branch2]* – исчез, *[branch1]* – появился, *[branch2]* – исчез. Самое интересное в этом то, что переключение веток для реальных проектов происходит примерно с такой же скоростью.

Но где же собственно ветки? Вы их видите? Я – нет. У нас в наличии лишь два последовательных коммита с тремя приклеенными к ним метками. Давайте же уже, наконец, ветвиться!

Переключитесь на *[branch1]*, измените файл *test.txt*, сохраните изменения в репозитории. Переключитесь на *[branch2]*, создайте файл *test2.txt*, сделайте коммит. Переключитесь на *[master]*, измените *test.txt*, сделайте ещё один коммит. Закончили? Смотрим, что у нас получилось.

Картинка 7

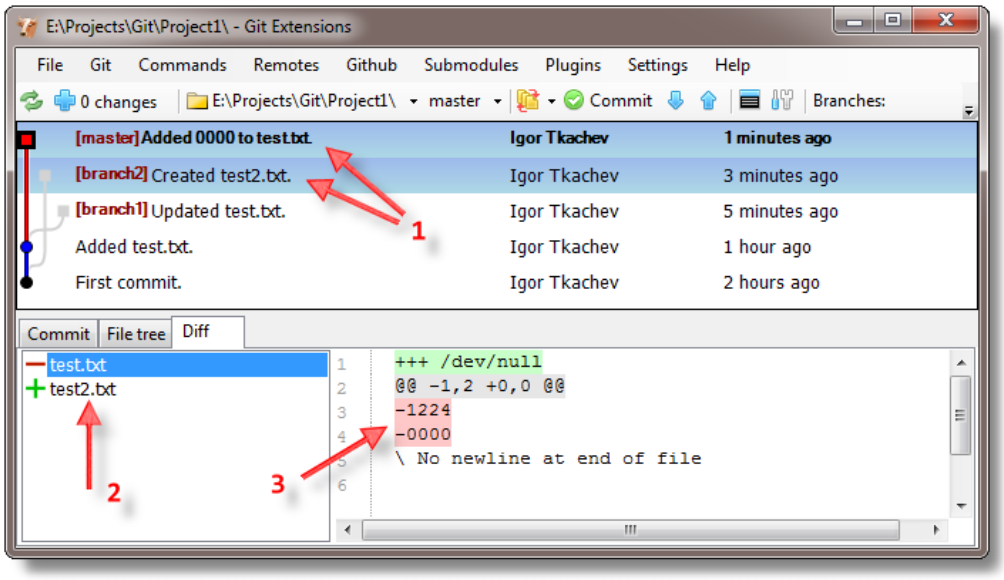

Рисунок 7.

Это уже похоже на дерево. Если выбрать одновременно несколько коммитов в дереве (1), то в окне Diff (2) и соседнем поле (3) можно увидеть разницу между этими коммитами. Выбирать можно более двух коммитов одновременно, от последовательности выбора результат тоже зависит.

Слияние

Слияние (merge) представляет собой процедуру согласования изменений, произведённых в различных версиях одного и того же проекта. В большинстве случаев слияние может быть произведено автоматически, если изменения в проекте не конфликтуют между собой.

Картинка 8

Слияние в Git выполняется не сильно сложнее создания новых веток. Единственная вещь, которую нужно понимать – это то, что любые манипуляции делаются над текущей веткой. То есть, если вы ходить слить две ветки *[master]* и *[branch1]*, то у вас есть выбор: объединить изменения в ветке *[master]* или объединить их в ветке *[branch1]*. Текущая ветка в нашем примере – *[master]* (1). Выделите в списке ветку, изменения из которой вы хотите внести в текущую ветку. В нашем примере это *[branch2]* (2). Вызовите контекстное меню и выберите пункт "Merge with branch" -> "branch2" (3). В появившемся диалоге оставьте всё как есть и жмите "Merge" (4).

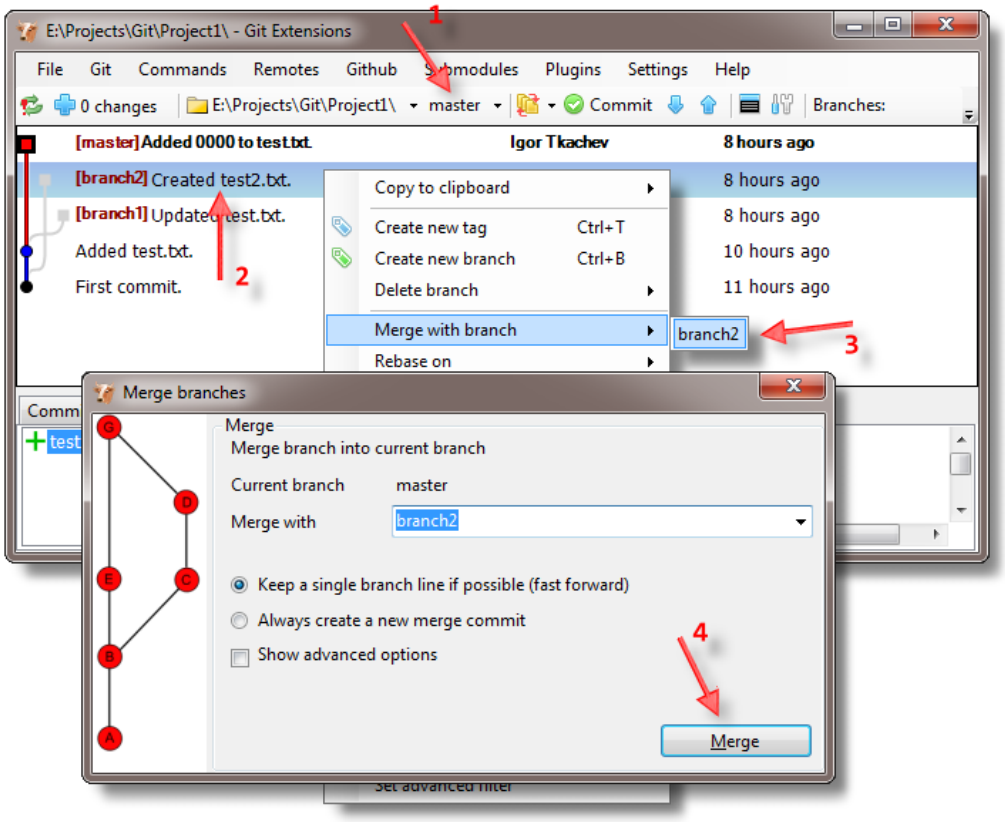

Рисунок 8.

Всё должно пройти гладко, и Git создаст новый коммит "Merge branch'branch2'", а в рабочем каталоге должен появиться новый файл *test2.txt* из ветки *[branch2]*.

Картинка 9

Теперь проделаем то же самое с веткой *[branch1]*. Помните, в обеих ветках и *[master]* и *[branch1]* мы вносили изменения в файл *test.txt*. На этот раз при попытке слияния возникает конфликт. Git Extensions выдаст окно с предупреждением. Нажимайте "OK" (1). Вам будет предложено разрешить конфликт прямо сейчас. Соглашайтесь (2).

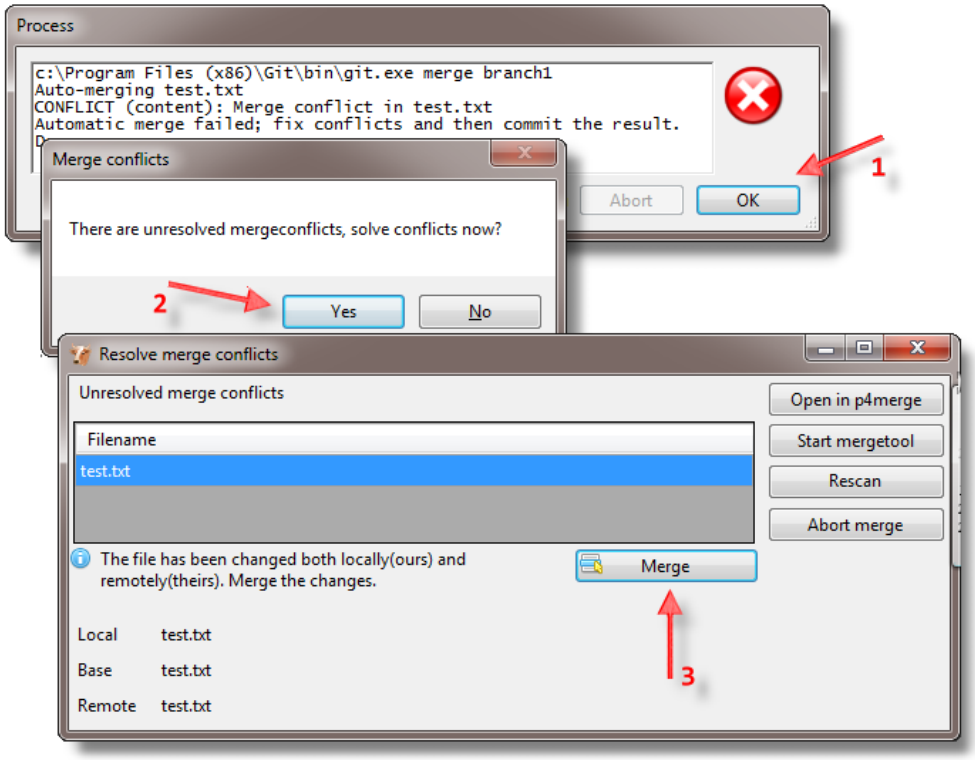

Рисунок 9.

Открывшийся диалог позволяет просмотреть неразрешённые изменения в вашей любимой утилите слияния, которую можно задать в настройках, и разрешить конфликт. Я использую для этих целей *p4merge*, в основном в силу привычки.

Картинка 10

После разрешения конфликтов Git Extensions спросит (1), не желаете ли вы сохранить коммит. Соглашаемся и опять попадаем в диалог коммита. Здесь мы уже как у себя дома. Выполняем необходимые действия и сохраняем коммит.

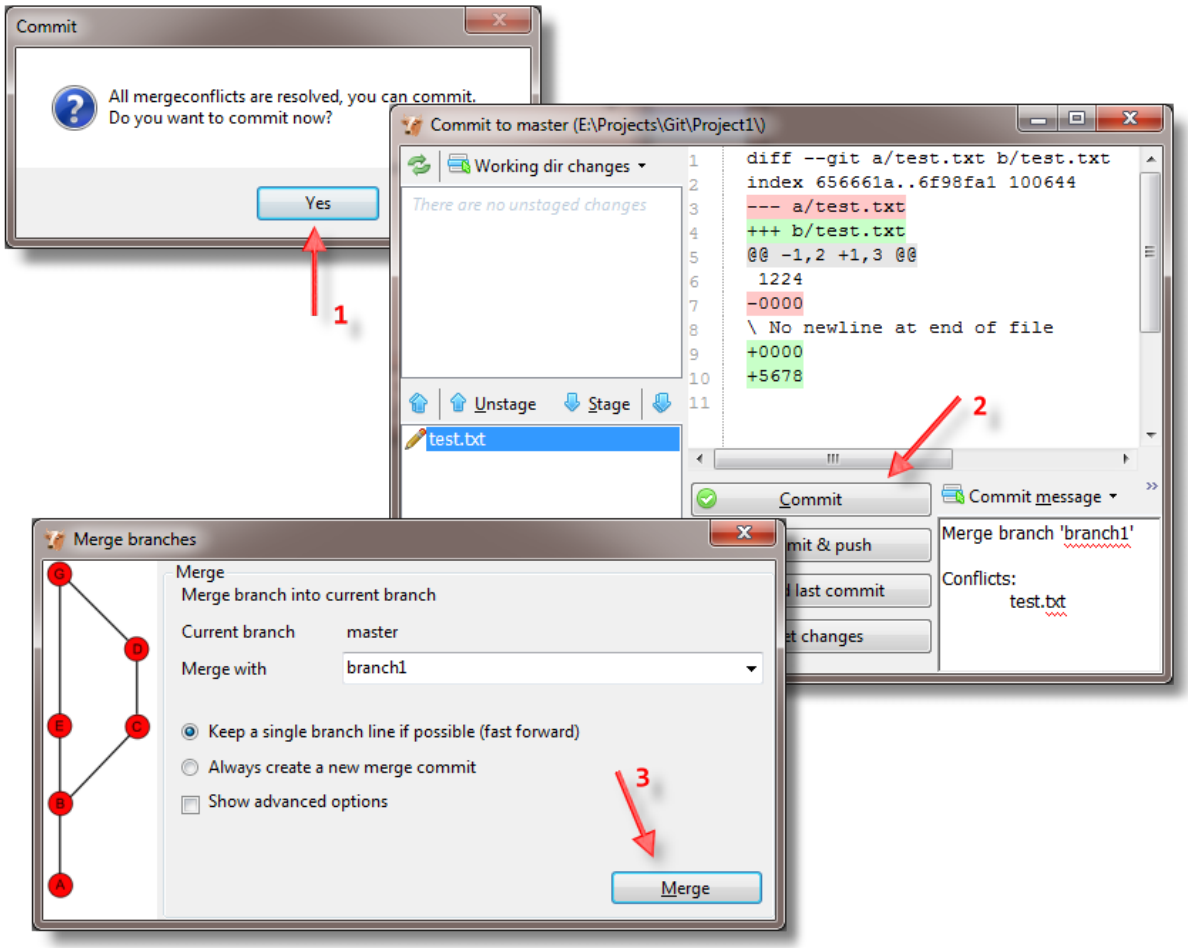

Рисунок 10.

После этого вернитесь в окно "Merge branches". Нажмите "Merge" ещё раз (3). Работа по слиянию веток завершена. Вот что должно получиться в результате:

Картинка 11

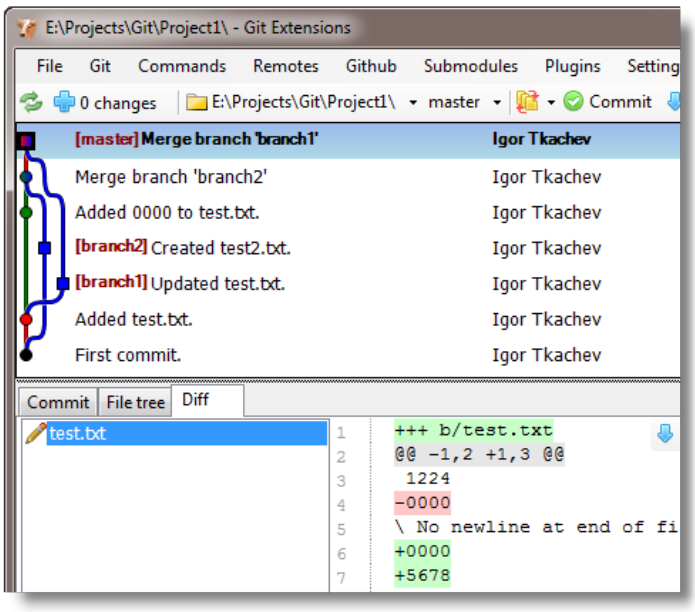

Рисунок 11.

После всех манипуляций было создано два новых коммита, и ветка *[master]* переехала в самый верх. *[branch1]* и *[branch2]* остались на своих прежних местах. В рабочем каталоге имеются два файла с изменениями из всех трёх веток. В принципе, если ветки *[branch1]* и *[branch2]* больше не нужны, их можно удалить. Коммиты, к которым они привязаны, никуда теперь не денутся, так как они являются частью истории ветки *[master]*.

Но мы поступим следующим образом. Ветку *[branch1]* мы удалим за ненадобностью, а *[branch2]* передвинем в самый верх к *[master]*.

В каждый конкретный момент времени мы работаем с одной единственной веткой, которая для нас в этот самый момент является текущей (checkout). Для удаления *[branch1]* он не должен быть текущим . Выберите его в дереве коммитов и вызовите контекстное меню. Выберите пункт "Delete branch" -> "branch1". Git Extensions немного запаникует и предупредит вас о том, что "мы его теряем", если на коммит больше не останется ссылок. Наш коммит – часть истории ветки *[mater]* и с ним всё будет в порядке. Удаляйте.

Картинка 12

Теперь займёмся синхронизацией веток *[branch2]* и *[master]*. Для начала переключимся на *[branch2]* (1). Обратите внимание, дерево изменило свой цвет, серым показаны те участки дерева, которые никак не причастны к текущей ветке.

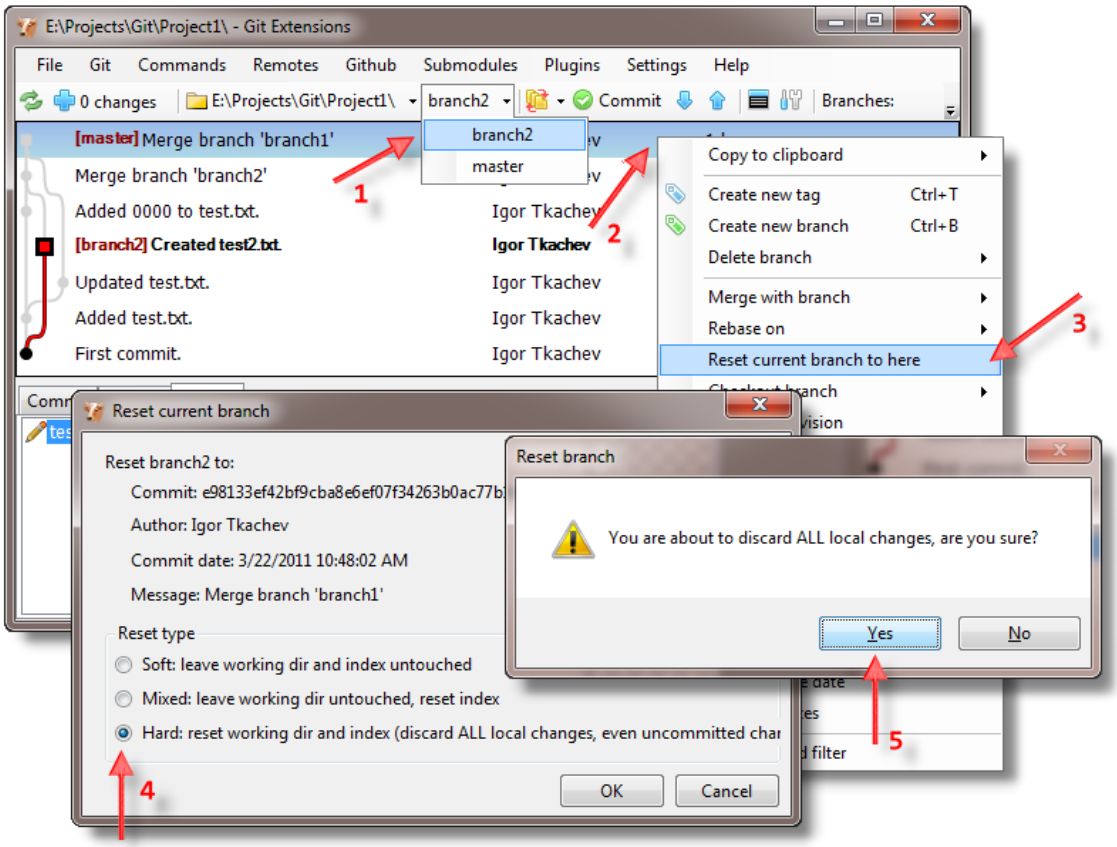

Рисунок 12.

Выделите в списке коммит, на котором у нас находится *[master]* и вызовите контекстное меню (2). Выберите пункт "Reset current branch to here" (3). В появившемся окне "Reset current branch" установите "Reset type", как показано на картинке (4), и нажмите "OK". Git Extensions вынесет последнее предупреждение (5). Жмём "Yes".

Картинка 13

Вот что должно получиться на текущий момент:

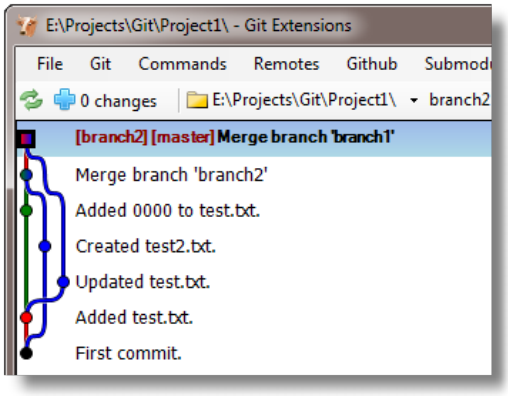

Рисунок 13.

Что же произошло? Фактически, последней операцией мы жёстко переставили метку в другое место дерева коммитов. В данном случае того же самого можно было бы добиться, если бы мы просто попросили Git слить *[branch2]* с *[master]*. Так всё получилось бы даже проще. Но указанным выше способом можно не только сливать ветки, но и переставлять их в любое место дерева коммитов, где никаких других веток нет.

Работа с репозиториями

Всё, что мы делали до сих пор, касалось исключительно локального репозитория. В жизни нам всё же приходится иногда обменивать плоды своего труда на плоды труда других людей. Для синхронизации с другими репозиториями в Git используются команды Push (толкать) и Pull (тянуть). Чтобы тянуть и толкать, нам понадобится ещё один репозиторий, пока не серверный, а обычный персональный.

Вернитесь в меню, где мы создавали наш первый репозиторий (нажмите Esc в текущем окне), и выполните процедуру по созданию нового репозитория с шага, описанного на картинке 2. Новый репозиторий назовите *Project2*. Создайте пару коммитов, пару веток и пару тестовых файлов. Одной из новых веток дайте имя *[branch3]*.

Картинка 14

Вот что получилось у меня:

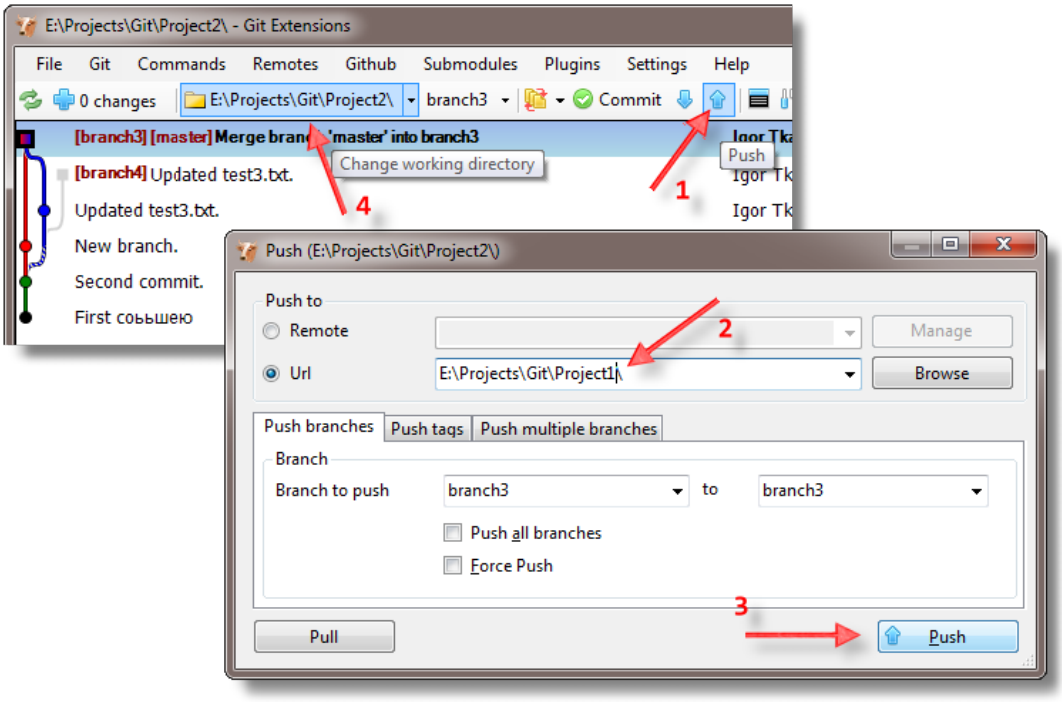

Рисунок 14.

Теперь нажмите на кнопку Push на панели инструментов (1), в появившемся диалоге выберите в качестве "Push to" поле "Url" и в нём введите каталог нашего первого репозитория (2). Нажмите "Push" (3). Теперь перейдите к нашему первому репозиторию (4).

Картинка 15

Вот так он теперь должен выглядеть:

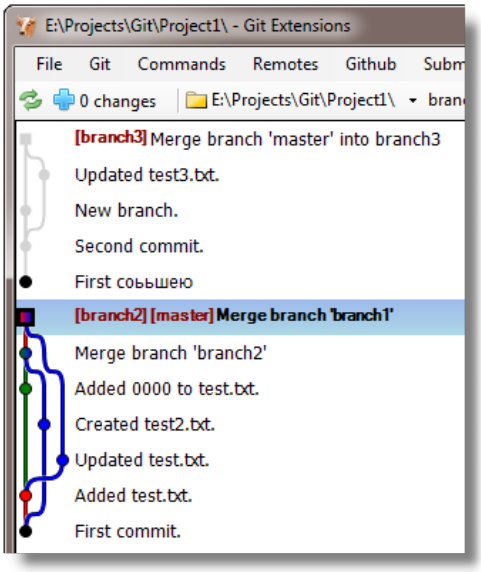

Рисунок 15.

Принято считать, что в Git-репозитории может быть только один корневой коммит. Не верьте этому. Только что мы с успехом опровергли этот миф, получив в результате два независимых дерева в одном репозитории.

В реальной жизни такая ситуация вряд ли имеет какой-то смысл. Но нам для понимания работы инструмента вполне подойдёт.

Картинка 16

Давайте теперь соберём это всё в одно дерево.

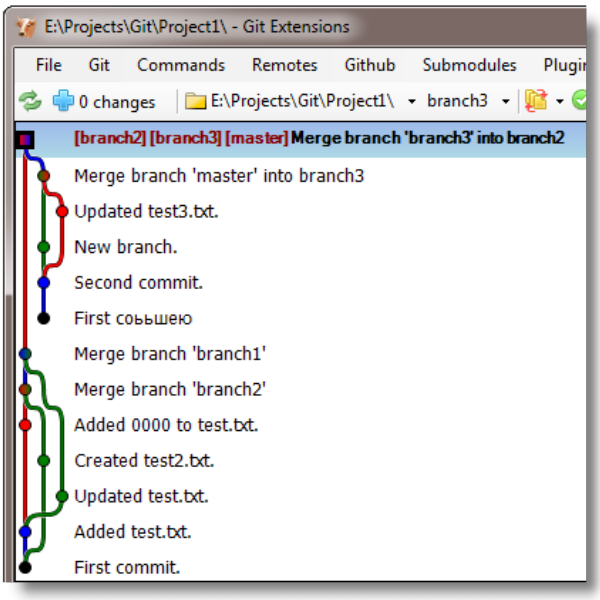

Рисунок 16.

Центральный репозиторий

Картинка 17

Наконец-то пришло время для создания центрального репозитория. Вернёмся опять к картинке 2, но в этот раз при выборе типа репозитория установим его тип в Central repository (1). Имя задайте по вкусу (2).

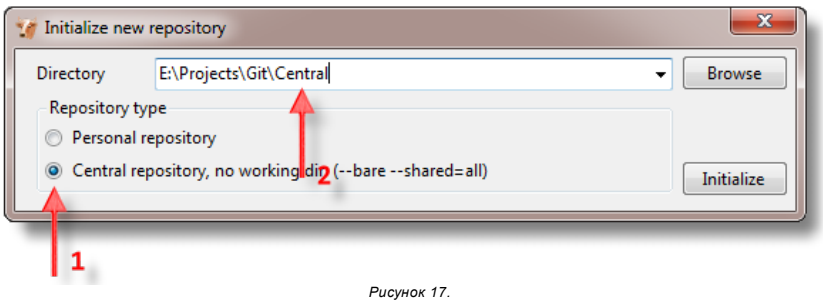

Рисунок 17.

Обычным соглашением именования центральных репозиториев является использование имени проекта с приставкой .git. В нашем случае это могло бы быть Project1.git. Но для примеров лучше использовать что-то отличное, чтобы было меньше путаницы. Каталог центрального репозитория может располагаться на сервере в сети, и вы можете использовать в том числе и UNC-имена. Создавать и инициализировать файл .gitignore для этого репозитория нет необходимости.

Вернёмся в наш предыдущий репозиторий и сделаем наш первый коммит в центральный репозиторий, воспользовавшись всё той же кнопкой Push на панели инструментов.

Картинка 18

На этот раз мы сконфигурируем наш удалённый репозиторий таким образом, чтобы связать его с нашим и упростить себе работу в дальнейшем. В открывшемся окне нажимайте "Manage" (1).

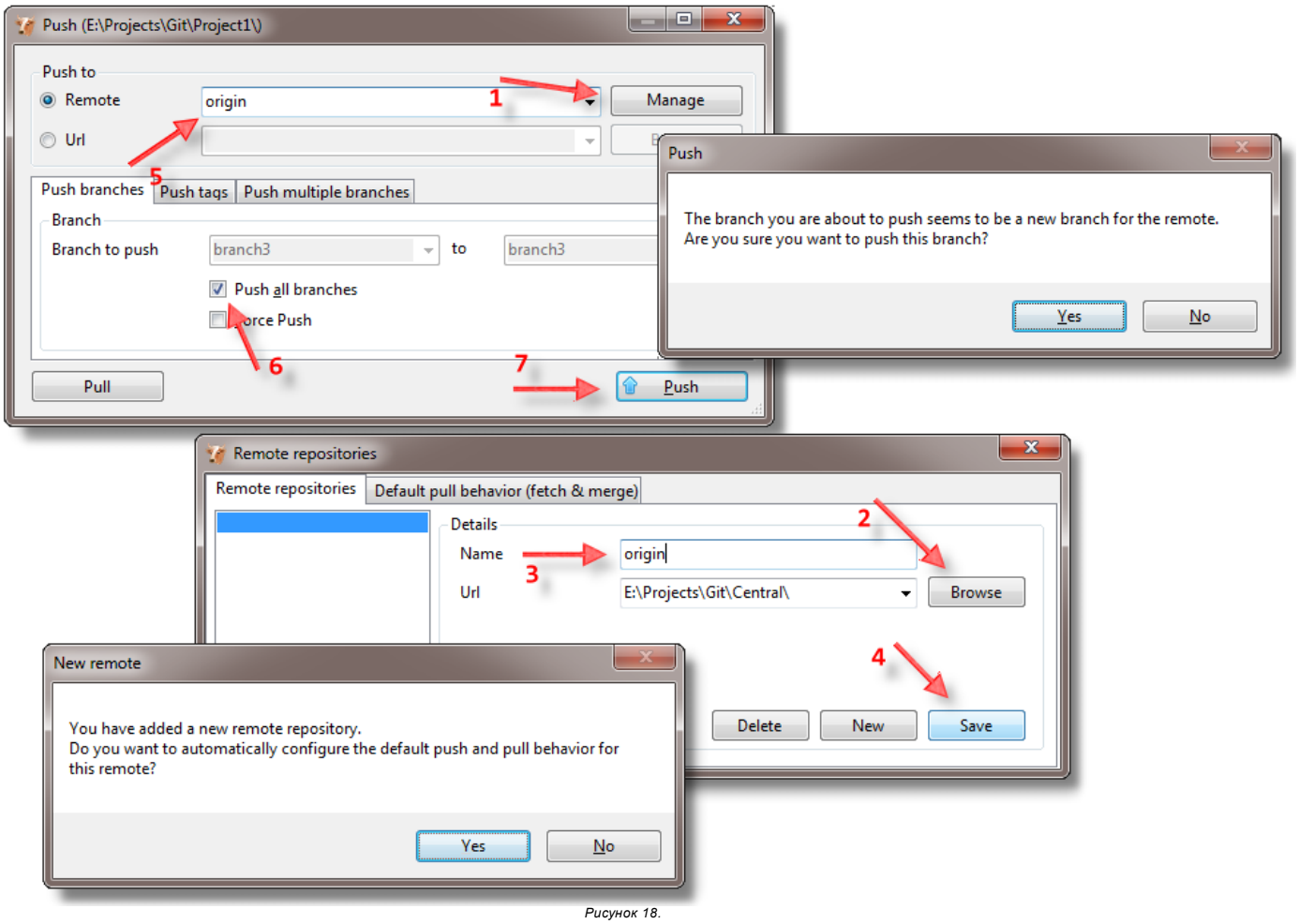

Рисунок 18.

В диалоге "Remote repositories" выберите URL центрального репозитория (2) и задайте ему имя "origin". Origin – это ещё одно соглашение. Так именуются центральные репозитории проектов, но опять же, это совсем не обязательно. Более того, вы можете иметь несколько удалённых репозиториев и вообще можете создать такую конфигурацию, в которой будет целая иерархия удалённых репозиториев. В общем, всё зависит только от вашего вкуса и ваших потребностей. После того, как вы дали имя удалённому репозиторию, нажмите "Save" (4). Git Extensions предложит вам автоматически сконфигурировать Push и Pull. Соглашайтесь. Закройте диалог "Remote repositories". В диалоге "Push" сделайте две вещи: введите имя удалённого репозитория (5) и взведите флажок "Push all branches" (6). Жмите "Push" (7).

После того как Git завершит синхронизацию, вы обнаружите, что рядом с красными метками веток появились ещё и зелёные: [origin/branch2] [origin/branch2] [origin/master]. Это положение веток удалённого репозитория. Переключитесь на репозиторий Central; в нём эти метки красные, что логично – для него они являются локальными. Теперь пойдём во второй рабочий репозиторий и вытянем в него изменения из центрального репозитория.

Картинка 19

Кнопка "Pull" находится на панели инструментов рядом с кнопкой "Push". Как и для первого репозитория, первым делом необходимо будет сконфигурировать удалённый сервер (1).

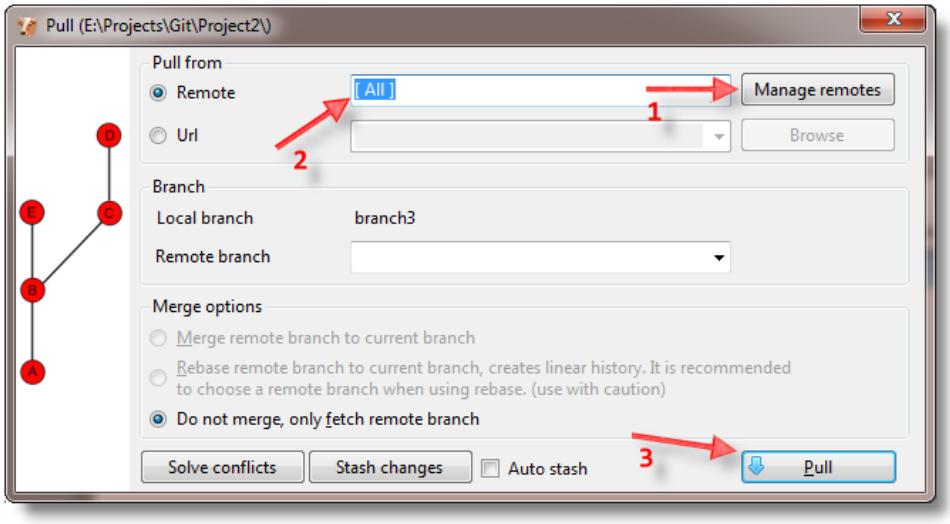

Рисунок 19.

В качестве Remote выберите [All] (2) и нажмите кнопку "Pull" (3). [All] означает все удалённые репозитории, если у вас их несколько.

Картинка 20

Теперь и второй репозиторий синхронизирован с центральным. Локальные ветки остались на своих местах и, чтобы их совместить с ветками из центрального репозитория, необходимо будет выполнить жёсткую перестановку или слияние, что в данном случае не должно вызвать никаких трудностей.

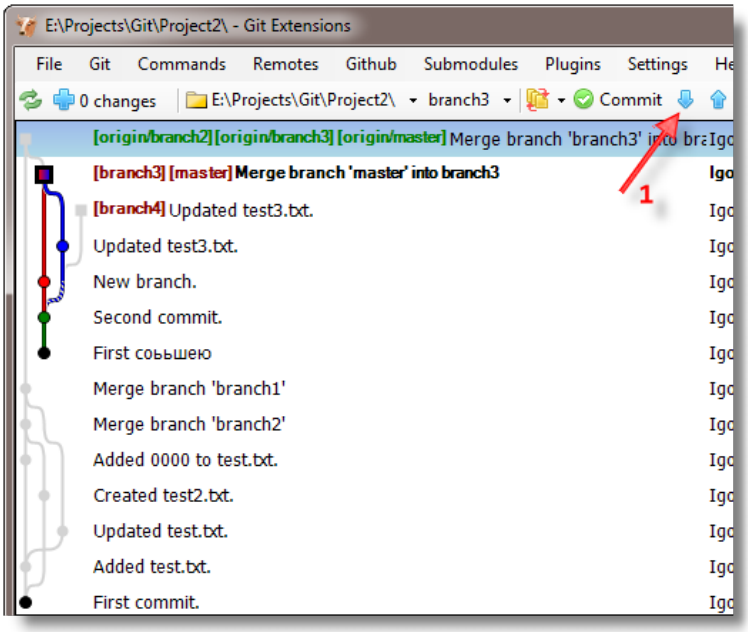

Рисунок 20.

В принципе, это всё, что нужно знать для того, чтобы начать плодотворно работать с Git, хотя возможности самого Git и Git Extensions на этом далеко не заканчиваются. Далее я рассмотрю ещё некоторые приёмы работы с Git, но пока вернусь к тому, с чего мы начали.

Почему Git?

Давайте ещё раз выясним, действительно ли Git так крут, почему мы должны тратить на его изучение своё драгоценное время, и что такого принципиально нового нам может дать переход на него. Ниже приведён основной декларируемый список преимуществ Git в сравнении с Subversion (сравнение именно с SVN в данном случае непринципиально, можно было бы взять любую другую аналогичную систему контроля версий):

1. Git гораздо быстрее Subversion.
2. Git репозиторий в десятки раз компактней репозиториев Subversion (примерно в 30 раз меньше на примере проекта Mozilla).
3. Git с самого начала разрабатывался как распределённая система контроля, позволяющая каждому разработчику иметь полный локальный контроль над репозиторием.
4. Ветки (branches) в Git гораздо легче и работают значительно быстрее.

Рассмотрим всё по порядку, но для начала опишем типичный день пользователя Git.

Начиная работать над новой функциональностью или над фиксом очередного бага, мы, прежде всего, создаём новую ветку или используем свободную существующую. Работа в ветке *[master]*, как правило, никогда не ведётся. Далее от забора и до заката усердно пишется код, и в процессе по завершении очередной мысли делается коммит в репозиторий. После завершения работы над функцией код проверяется на работоспособность и изменения сливаются с веткой *[master]*. Синхронизация с центральным репозиторием делается по необходимости. Если что-то не складывается, или мы не успеваем закончить – ничего страшного, наша ветка никому не мешает и не ломает текущий код в главной ветке. Более того, если вдруг, посередине работы, возникла срочная необходимость внести какие-либо изменения в основной код, то мы просто сохраняем в репозитории текущие изменения, создаём новую ветку или переключаемся на существующую и начинаем работу над новой функциональностью. Ниже рассмотрены причины, по которым всё это становится возможным.

Быстродействие

Как мы выяснили, Git хранит всю историю локально. В результате для переключения на другие ветки не требуется соединения с сервером, и само переключение происходит практически мгновенно. Если бы для этого требовались минуты, то, скорее всего, вся идея вряд ли работала бы.

Компактность

Компактность, во-первых, обеспечивает быстрый обмен с сервером, во-вторых, быстрое клонирование репозитория. Например, клонирование репозитория BLToolkit, размеры которого мы обсуждали выше, занимает пару секунд. Само по себе клонирование удобно для одновременной работы над разными версиями продукта, например, для сравнения работы старой и новой функциональности.

Распределённость

Распределённость позволяет не толпиться всем у одного единственного центрального репозитория и снимает зависимость от работоспособности и доступности этого репозитория. Сломался репозиторий? У каждого из разработчиков в наличии имеется полная его копия. Не работает сеть? Склонировали репозиторий на флешку, отдали соседу. Работа не останавливается ни на секунду.

Ветки

Признаюсь честно. Я уже более десятка лет использую самые разные системы контроля версий, но за всё это время мною было создано веток меньше, чем при написании этой статьи. Ветки всегда были наказанием, чем-то, что являлось расплатой либо за изначальный кривой дизайн приложения, либо за неверно поставленные процессы в команде. Теперь это стало в буквальном смысле игрой, не только спасающей во многих ситуациях и упрощающей жизнь, но и открывающей новые принципы работы с кодом.

Раньше, забирать свежие изменения из центрального репозитория приходилось с замиранием сердца. Ведь если код и не сломан, то не исключено, что придётся его сливать со своей локальной копией, чтобы работать дальше, а времени на это нет. Сейчас можно вытащить последние изменения вообще без слияния (pull + merge – это вообще как бы всего лишь опция), ведь локальные и серверные ветки-метки вполне могут указывать на разные коммиты.

Беспокоиться о том, что по неосторожности сломается код в центральном репозитории, тоже не стоит. Не уверен – не сливай. Залей пока только свою ветку. Даже если что-то и сломалось – не велика беда, откатить изменения в Git проще простого. При этом откат изменений, как вы уже догадались – это тоже новый коммит, т.е. ломающий коммит останется в истории и к нему можно будет всегда вернуться.

Раньше коммит обычно выполнялся после реализации какой-либо функциональности или в конце рабочего дня. В Git коммит делается после реализации очередной мысли. Ветки тоже могут создаваться буквально под апробацию очередной идеи и, при желании, к ней можно будет потом вернуться и ещё раз её подумать. У вас никогда не бывало такого – написали код, подумали, переписали по-другому, подумали ещё раз и захотели вернуться к предыдущей идее? Теперь об этом можно не беспокоиться, просто сделайте вовремя очередной коммит или даже целую ветку.

Git vs. Mercurial

Повторюсь ещё раз. Git и Mercurial – это новый подход в работе с кодом. Мы теперь работаем с деревом коммитов, а не с деревом каталогов. Судя по последним версиям TortoiseHg, его разработчики тоже начинают это понимать, и очень хотелось бы надеяться на то, что скоро у нас появится ещё один замечательный инструмент.

Другие возможности Git

Tags

Tag – это неперемещаемая метка. Она выглядит как обычная метка в дереве коммитов, только Git Extensions отображает её синим цветом. Позволяет именовать и быстро находить конкретные коммиты и не замусоривать список веток.

Stash

Stash (нычка) – место, куда можно временно припрятать текущие изменения. Git не даст вам переключиться на другую ветку, если вы не сохранили текущие изменения. Если же вы не хотите по какой-либо причине делать коммит, то можете отложить их в stash, а потом забрать. Работает как стек. Полезно, если вы ошиблись текущей веткой и начали делать в ней изменения. Эти изменения можно переложить в другую ветку через stash.

Bisect

Последовательное переключение коммитов для выявления ломающего коммита. Если вы обнаружили какую-либо проблему по прошествии нескольких недель, месяцев, лет, то найти ломающее изменение становится нетривиальной задачей. Режим bisect позволяет определить два коммита, один плохой – в котором проблема проявляется, и один хороший, в котором всё в порядке. Далее Git будет загружать коммиты, используя алгоритм двоичного деления, и предлагать вам решить, хороший коммит загрузился или плохой. Вскоре вы выйдете на ломающий коммит. Если вы часто делаете коммиты, то вполне возможно, что только лишь просмотрев изменения, вы сможете найти проблемный код.

Интересно, что эта возможность пригодилась мне буквально на следующий день после того, как я о ней узнал. Баг был внесён в код несколько месяцев назад. Тем не менее, на его поиск понадобилось лишь несколько минут.

Cherry pick

Команда, позволяющая забрать изменения из другого коммита. В рабочий код вносятся только изменения из другого коммита. Полезна при переносе мелких изменений между ветками без их слияния. Например, вы или кто-то другой исправил баг в главной ветке, и вы хотите перенести в свою рабочую ветку только это изменение.

Revert commit

Допустим, вы хотите отменить изменения, внесённые каким-то коммитом. Выполните для него команду "revert commit". С самим коммитом ничего не произойдёт, но в рабочую копию кода будут внесены изменения, отменяющие изменения этого коммита. Далее по обычной схеме вы можете их сохранить в репозиторий. Если вы хотите отменить только часть изменений, то поместите в staging area только эти изменения, а для остальных файлов выполните "Reset file changes".

Recover lost objects

Если вы удалите метку с коммита, на который больше нет никаких ссылок, то этот коммит исчезнет, но не бесследно. В меню "Settings" можно найти команду "Recover lost objects", которая позволяет восстановить потерянные объекты. Восстановить коммит можно будет до тех пор, пока для репозитория не будет собран мусор. После этого коммит окончательно исчезнет.

Полезные советы

Делайте коммиты чаще. Это позволит проще сливать изменения с другими ветками.

Git позволяет отменять не только изменения целых файлов, но и их частей.

В Git нет команды переименования/перемещения файлов. Тем не менее, Git пытается выполнять эту работу автоматически. Чтобы гарантированно сохранить историю переименований, сделайте коммит, потом выполните переименования/перемещения файлов и сделайте ещё один коммит. Точное совпадение содержимого файлов даст гарантию сохранения истории. Если же вы одновременно переименуете и измените файл, то такой гарантии не будет.

Git Extensions содержит далеко не все команды, которые поддерживает Git. Однако это не является большой проблемой. Если нужно выполнить подобную команду, воспользуйтесь для этого окном "Git bash". Git Extensions открывает его для текущего репозитория и текущей ветки.

Помните, что все изменения делаются над текущей веткой. Например, слияние делается **из** желаемого коммита **в** рабочую копию.

Git для боссов

Лучшее – враг хорошего. Обычно так рассуждает среднестатистический босс. Чтобы внедрить что-то новое и перспективное, разработчик должен объяснить, чем новый инструмент так уникален, и какие бенефиты принесёт его применение. Обычно с этим могут возникнуть определённые проблемы, но только не в случае с Git.

В этой статье описывается почему. Обычной практикой в Subversion является не использование веток (ввиду их геморройности). Разработка новой функциональности, как правило, ведётся поверх текущего кода. Когда же в старом коде обнаруживается баг, то исправление этого бага становится нетривиальной задачей. Быстрое и лёгкое создание веток в Git обеспечивает лёгкий доступ к разным версиям одного и того же кода в одно и тоже время, что в результате позволяет исправлять баги немедленно по ходу дела. Это действительно так, и это тот самый аргумент, который подействовал и на моего босса.

Заключение

Прошло не так много времени с тех пор, как Git Extensions прочно обосновался на моём рабочем столе. И вот однажды в один из пасмурных зимних дней мне пришлось работать с проектом, для которого в качестве репозитория использовался Subversion. Надо признаться – мне довелось испытать необычайный дискомфорт. До тех пор подобное приходилось испытывать лишь дважды: при переходе обратно с C++ на C и с Nemerle – на C#. Собственно, именно это и побудило меня к написанию этой статьи.

Полезные ссылки

- GitHub.com – Git-хостинг, в том числе бесплатный.
- Hosting a Git server under IIS7 on Windows – Git-хостинг в IIS7.
- Git Web Access – ASP.NET Git-хостинг.

P.S. Особые благодарности "апостолу" Игорю Н. за обращение в веру :-)