

# Основы работы с Git

## Оглавление

Введение в систему контроля версий Git.....	1
GitHub – самый крупный хостинг IT-проектов .....	2
Установка Git .....	2
Установка TortoiseGit.....	4
Регистрация на GitHub .....	4
Основы работы с Git .....	7
Создание репозитория на GitHub .....	7
Клонирование репозитория на свой компьютер .....	8
Локальная работа с репозиторием.....	9
Работа с удаленным сервером.....	14
Процедура работы с Git: commit, pull, улаживание конфликтов, push.....	15
Работа с ветками.....	16
Создание тегов.....	17
Удаление репозитория с GitHub.....	19
Задание на практику .....	19

## Введение в систему контроля версий Git

**Система контроля версий (СКВ)** — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.

СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Если в проекте под контролем СКВ испортить или потерять некоторые файлы, их можно будет легко восстановить. Накладные расходы на выполнение восстановления и на хранение истории изменений будут минимальными.

СКВ подходит для работы не только над программным кодом, но и для работы с любыми файлами: текстами книг и документов, графическим материалом и т.д.

Существуют 3 вида СКВ: локальные, централизованные и распределенные.

Исторически первыми появились **локальные СКВ**. Они предназначены для случая, когда выполняется работа над проектом в одиночку. История изменений хранится на том же компьютере, что и сам проект, над которым ведется работа, и можно легко восстановить любую версию файла с помощью специальной БД, в которой хранятся патчи – различия между соседними версиями.

Следующим этапом в развитии СКВ стали **централизованные СКВ**. Они стали ответом на необходимость совместной работы над проектом нескольких участников. На центральном сервере хранятся версии файлов, клиенты по запросу получают копии на свои компьютеры. Самой популярной централизованной СКВ является **Subversion (svn)**, который до сих пор используется некоторыми организациями. Подход к организации хранения кода с помощью центрального сервера дает ряд преимуществ: совместная работа над проектом нескольких участников, разграничение прав доступа на чтение и изменение. Слабым местом такой архитектуры является центральный сервер. При его недоступности работа над проектом становится недоступной. А при потере файлов на нем восстановить их уже не удастся (если не выполнялось их резервное копирование).

Решением этих проблем является использование **распределенных СКВ**. Самой популярной из них на сегодняшний день является разработанная Линусом Торвальдсом система **Git**. В распределенных системах копия всех файлов проекта вместе с историей изменений хранится у каждого участника. При потере данных на сервере их легко можно восстановить, просто скопировав у любого клиента.

СКВ Git позволяет создавать и управлять репозиториями проектов с помощью команд. Это позволяет легко писать скрипты по работе с репозиториями, а так же создавать графические программы для визуализации изменений и упрощения работы с репозиторием. Одна из таких программ, с которой мы познакомимся – TortoiseGit.

Кроме того, Git очень удобен при работе с различными ветками разработки. Например, может существовать ветка «master», в которой находится протестированный рабочий код и ветка «develop» для разработки новой функциональности. После разработки и тестирования новой функции изменения из develop «заливаются» в ветку master.

Git позволяет гибко настраивать права доступа к репозиториям для разных участников проекта: чтение или изменение.

### GitHub – самый крупный хостинг IT-проектов

**GitHub** — самый крупный веб-сервис для хостинга IT-проектов и их совместной разработки. Основан на системе контроля версий Git и разработан на **Ruby on Rails** и **Erlang**.

Сервис абсолютно бесплатен для проектов с открытым исходным кодом и предоставляет им все возможности, а для частных проектов предлагаются различные платные тарифные планы.

Создатели сайта называют GitHub «социальной сетью для разработчиков». Кроме размещения кода, участники могут общаться, комментировать правки друг друга, а также следить за новостями знакомых. С помощью широких возможностей Git программисты могут объединять свои репозитории — GitHub предлагает удобный интерфейс для этого и может отображать вклад каждого участника в виде дерева.

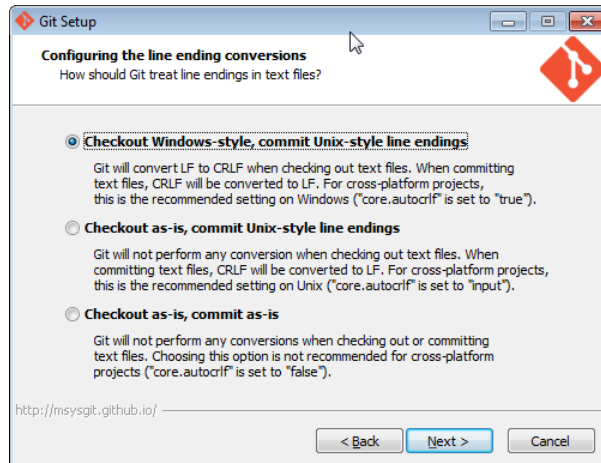
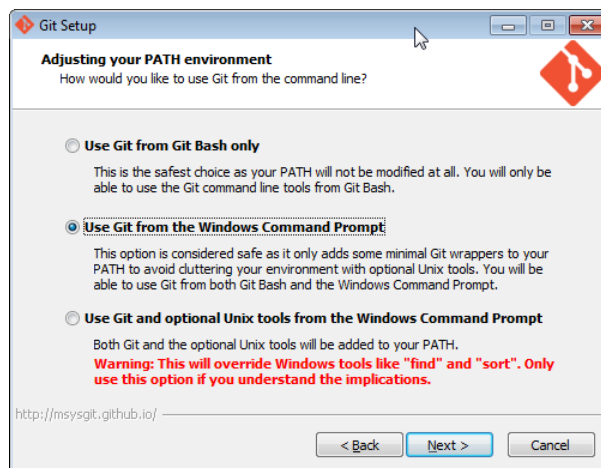
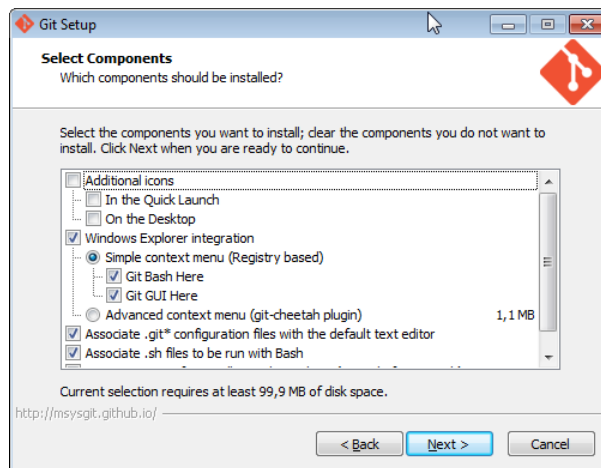
Для проектов есть личные страницы, небольшие Вики и система отслеживания ошибок. Прямо на сайте можно просмотреть файлы проектов с подсветкой синтаксиса для большинства языков программирования. На платных тарифных планах можно создавать приватные репозитории, доступные ограниченному кругу пользователей.

### Установка Git

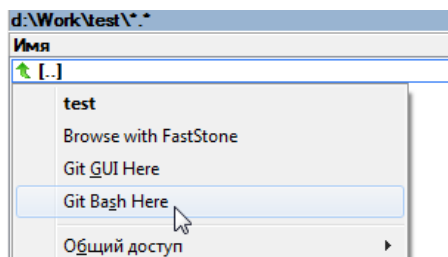
Для начала работы с Git необходимо скачать и установить Git и Windows-клиент TortoiseGit. После этого можно работать с локальными репозиториями, что часто бывает удобно для работы в одиночку.

Скачать Git можно по ссылке: <https://git-scm.com/>

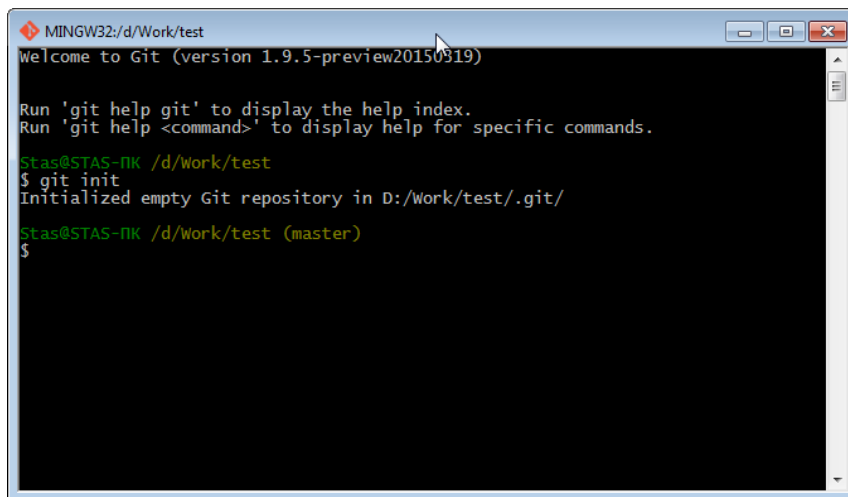
После скачивания необходимо запустить инсталлятор. На момент написания методички последняя версия Git для Windows – 1.9.5. При установке нужно оставить все параметры по умолчанию:



Проверить правильность установки можно следующим образом. Нужно создать пустой каталог на диске, перейти в него, а затем через контекстное меню запустить в этом каталоге **Git Bash Here**



В появившемся окне набрать команду **git init** и нажать Enter:



```
MINGW32:/d/Work/test
Welcome to Git (version 1.9.5-preview20150919)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

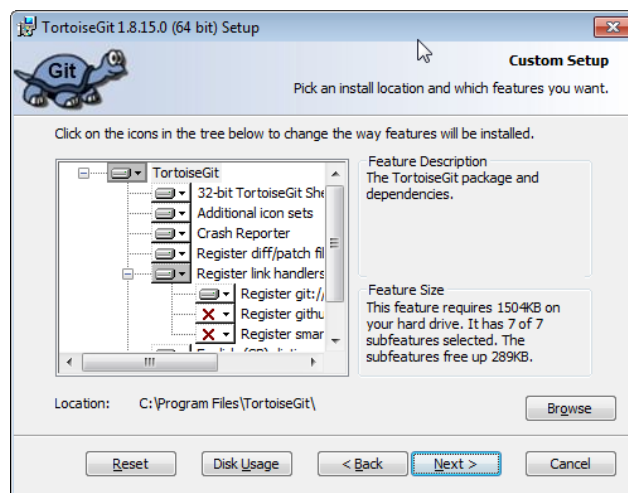
Stas@STAS-ПК /d/Work/test
$ git init
Initialized empty Git repository in D:/Work/test/.git/
Stas@STAS-ПК /d/Work/test (master)
$
```

В результате в каталоге будет инициализирован новый репозиторий. Появится подкаталог с именем **.git** – каталог, в котором Git будет хранить информацию об изменениях файлов. В принципе, этого уже достаточно, чтобы работать с локальным проектом и сохранять историю изменения файлов.

Но для большего удобства рекомендуется установить GUI-клиент для Windows – **TortoiseGit**.

### Установка TortoiseGit

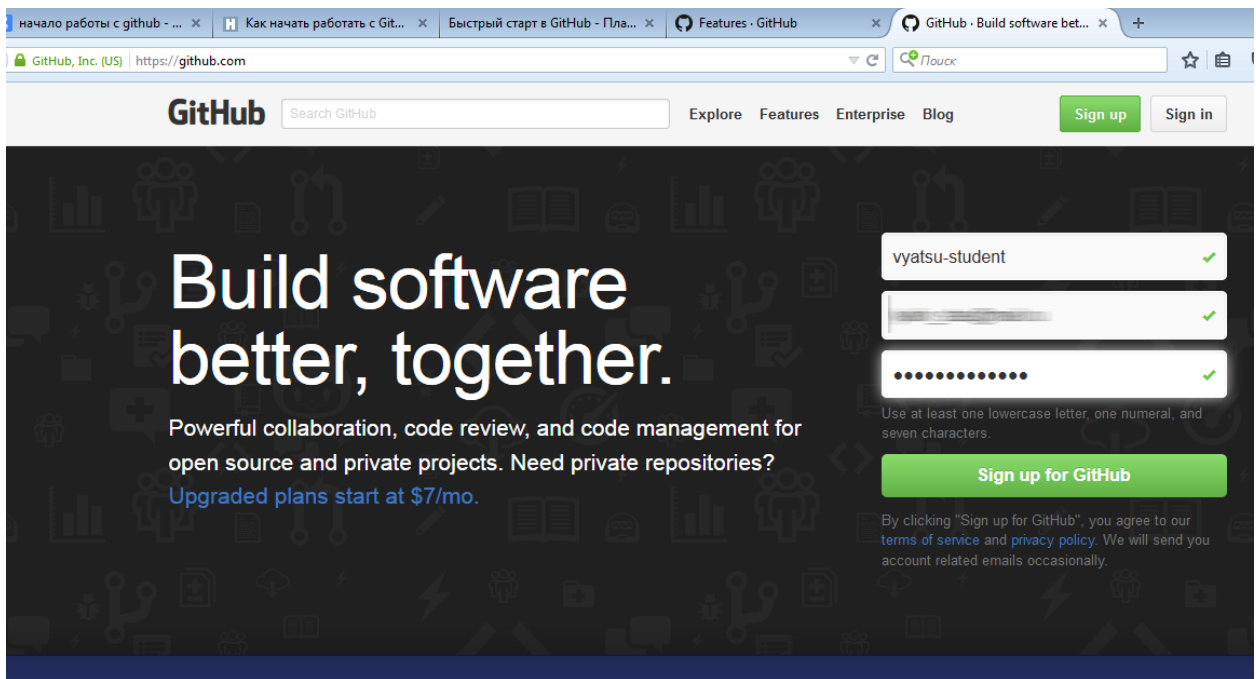
Найти последнюю версию TortoiseGit можно по ссылке: <https://code.google.com/p/tortoisegit/wiki/Download>. На момент написания методички последняя версия TortoiseGit – 1.8.15. При установке все настройки нужно оставить по умолчанию.



### Регистрация на GitHub

Для обеспечения совместной работы над проектом нескольких участников, проект нужно разместить на стороннем сервере в локальной или глобальной сети. Самым популярным git-хостингом на сегодняшний день является GitHub. Именно на нем мы и будем создавать репозиторий.

Для начала нужно зарегистрироваться в GitHub:



Необходимо указать логин, e-mail и пароль.

После этого будет предложено выбрать тарифный план. По умолчанию выбран бесплатный план, позволяющий создавать неограниченное количество публичных репозиторий. Именно он нам и нужен, поэтому ничего менять не нужно. Нажать кнопку «Finish sign up».

Сразу после входа на сайт новым пользователям GitHub позволяет пройти интерактивный тур, чтобы познакомиться с основами Git и возможностями GitHub. Вы можете пройти этот тур при желании.

Процедура регистрации требует подтверждения email. Для этого на указанный электронный ящик будет выслано письмо, в котором потребуется пройти по ссылке для завершения регистрации.

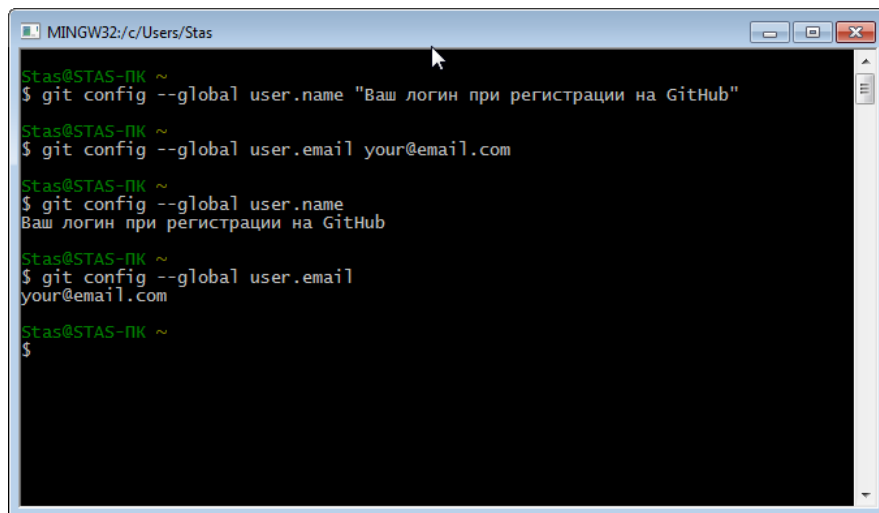
На этом регистрация на GitHub завершена, но остался последний момент для начала полноценной работы с Git – нужно идентифицировать себя. Для этого существуют разные способы. Один из них – воспользоваться утилитой **Git Bash**.

Запустите **Git Bash** и выполните команды:

```
git config --global user.name "ваш логин при регистрации на github"  
git config --global user.email ваш\_email@на.github
```

Проверить значения можно с помощью этих же команд, но без параметра:

```
git config --global user.name  
git config --global user.email
```

A screenshot of a MINGW32 terminal window. The title bar shows the path 'MINGW32:/c/Users/Stas'. The terminal has a black background with green text. It shows a series of Git configuration commands and their outputs. The commands are: 'git config --global user.name "Ваш логин при регистрации на GitHub"', 'git config --global user.email your@email.com', and then 'git config --global user.name' followed by 'Ваш логин при регистрации на GitHub', and 'git config --global user.email' followed by 'your@email.com'. The prompt is 'Stas@STAS-ПК ~' and the command prompt is '\$'.

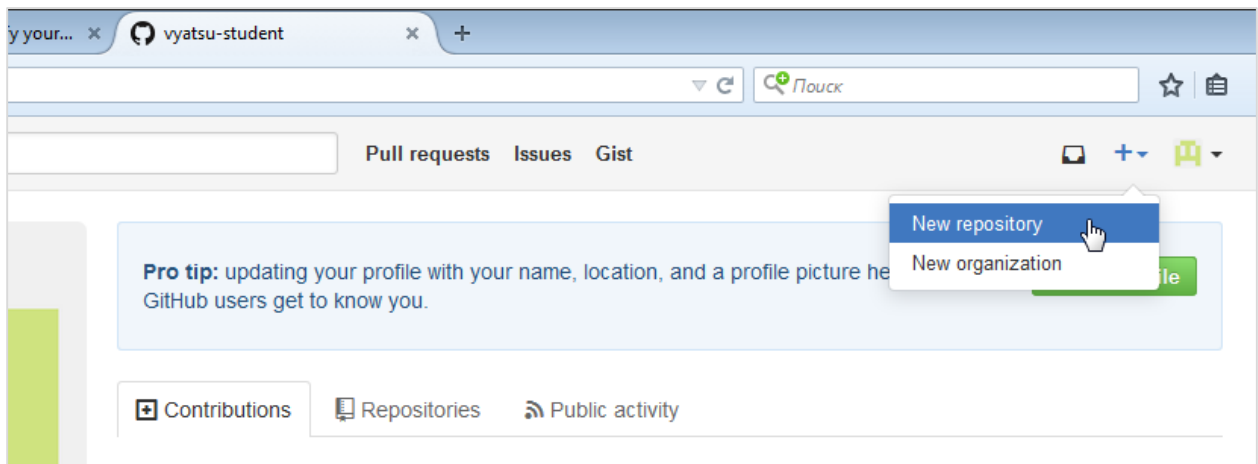
```
Stas@STAS-ПК ~  
$ git config --global user.name "Ваш логин при регистрации на GitHub"  
Stas@STAS-ПК ~  
$ git config --global user.email your@email.com  
Stas@STAS-ПК ~  
$ git config --global user.name  
Ваш логин при регистрации на GitHub  
Stas@STAS-ПК ~  
$ git config --global user.email  
your@email.com  
Stas@STAS-ПК ~  
$
```

Ключ **--global** означает, что меняются глобальные настройки Git на вашем компьютере. Они хранятся в папке пользователя Windows в файле **.gitconfig**. Их можно поменять вручную в этом файле или через интерфейс программы TortoiseGit.

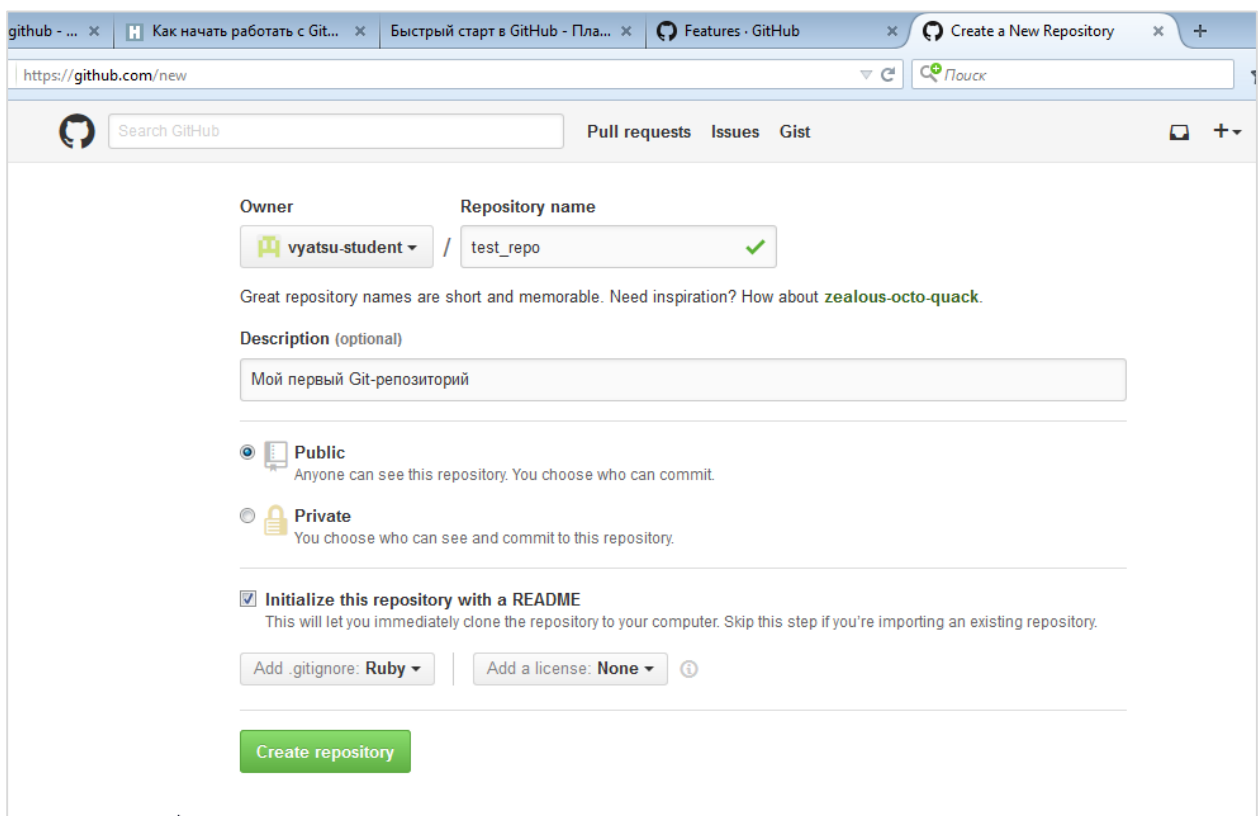
## Основы работы с Git

### Создание репозитория на GitHub

Первое, с чего нужно начать для контроля изменений в файлах проекта – это создать репозиторий на GitHub. Сделать это можно со своей страницы, нажав **New repository**:



Необходимо указать имя репозитория (проекта), добавить описание. При необходимости можно создать файл **.gitignore** под ту технологию, которую вы собираетесь использовать в проекте. **.gitignore** позволяет не засорять репозиторий теми файлами, которые генерируются автоматически. Например, файлы `obj` для программ на C++. В примере ниже создан файл **.gitignore** для проектов на языке **Ruby**.



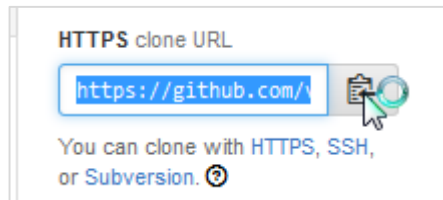
По завершении заполнения полей нужно нажать кнопку **Create repository**. Репозиторий создан.

## Клонирование репозитория на свой компьютер

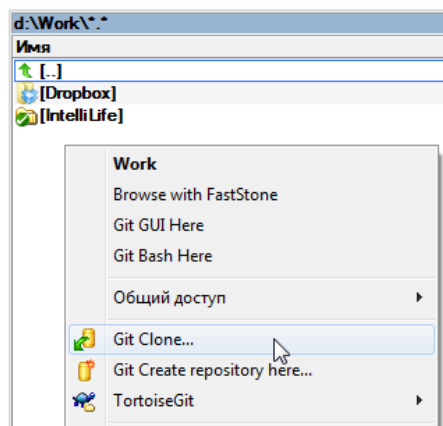
Следующим шагом нам нужно получить полную копию проекта с сервера на свой компьютер. Эта операция в терминологии Git называется клонирование, так как получается полная копия репозитория с сервера, включая информацию обо всех файлах, истории их изменений и ветках разработки.

Для того, чтобы клонировать себе только что созданный пустой репозиторий, нужно выполнить следующие шаги:

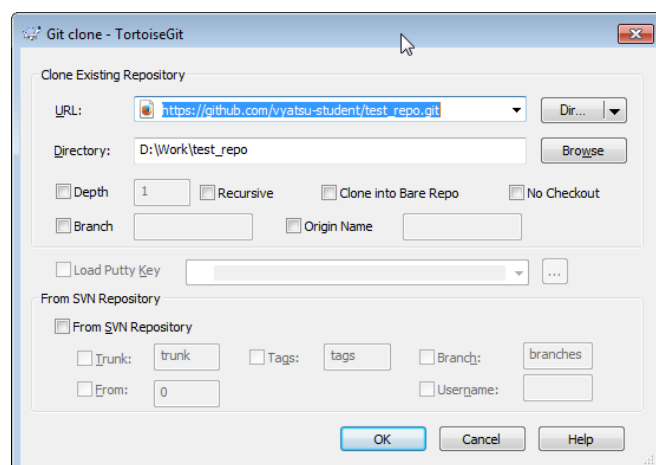
на странице проекта скопировать ссылку для клонирования репозитория:



На своем компьютере в рабочем каталоге вызвать контекстное меню, выбрать пункт **Git Clone...**

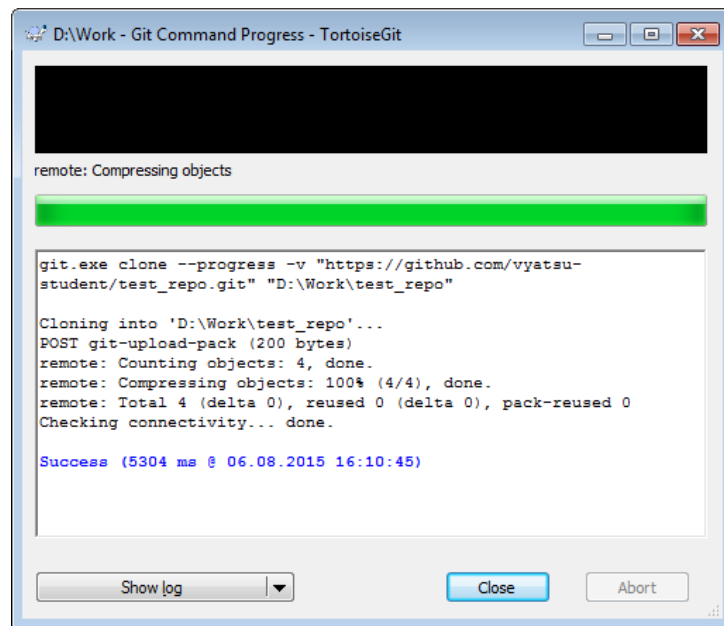


В появившемся окне указать путь до репозитория на сервере GitHub (тот, что только что скопировали в буфер обмена – на самом деле TortoiseGit автоматически заполнит это поле, взяв значение из буфера обмена) и подкаталог проекта, куда будет клонирован репозиторий:

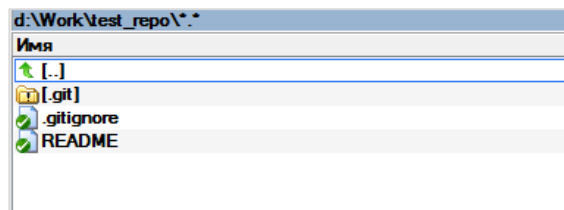


В случае успешного клонирования вы увидите следующее окно:





Если теперь зайти в только что созданный каталог проекта, то он будет иметь такую структуру:



Скрытый каталог **.git** хранит служебную информацию Git. Если удалить его, будет потеряна вся история изменений, а каталог из репозитория превратится в простой каталог Windows.

Файл **.gitignore** содержит набор масок для игнорируемых файлов. Изменения в таких файлах отслеживаться не будут.

**README.MD** – файл, сгенерированный GitHub. Сюда можно вносить любые пояснения. Содержимое этого файла увидят пользователи на странице репозитория.

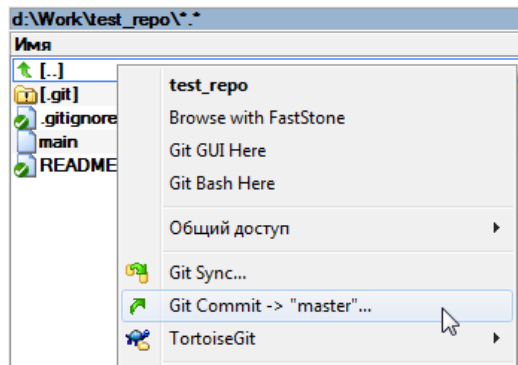
### Локальная работа с репозиторием

#### *Добавление в репозиторий файлов и фиксация изменений*

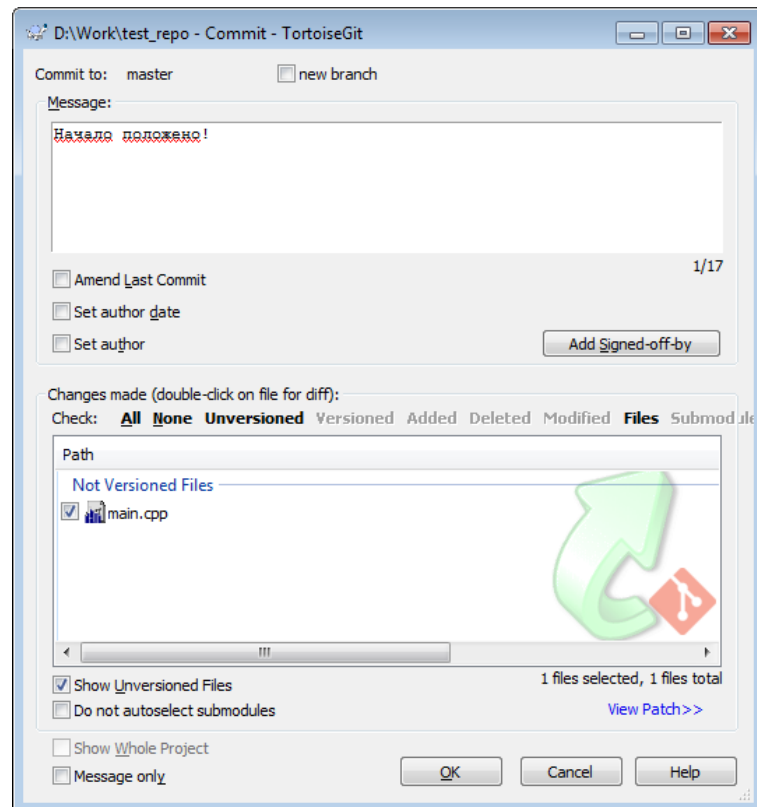
Теперь давайте наполним пустой репозиторий. Создадим в нем текстовый файл. Например, **main.cpp** такого содержания:

```
int main()  
{  
    //  
}
```

Запустите контекстное меню в каталоге с проектом и выберите **Git Commit -> "master"**... Это означает, что вы хотите зафиксировать изменения в ветке **master**. Изменения в данном случае – это добавление в репозиторий нового файла. Ветка **master** создается по умолчанию. В git легко создавать новые ветки, переключаться на них, соединять изменения из разных веток.



В появившемся окне отметьте файлы, изменения в которых нужно зафиксировать. В нашем случае это всего один файл **main.cpp**. Заполните поле **Message** и нажмите **Ok**. К сообщениям в коммитах нужно относиться серьезно. Они должны быть четкими, лаконичными и информативными, чтобы разным людям, работающим над проектом, было проще понять, какие изменения были сделаны. Но и дублировать словами очевидные вещи, вроде «добавлен новый файл» тоже не нужно – это и так легко увидеть по журналу изменений.



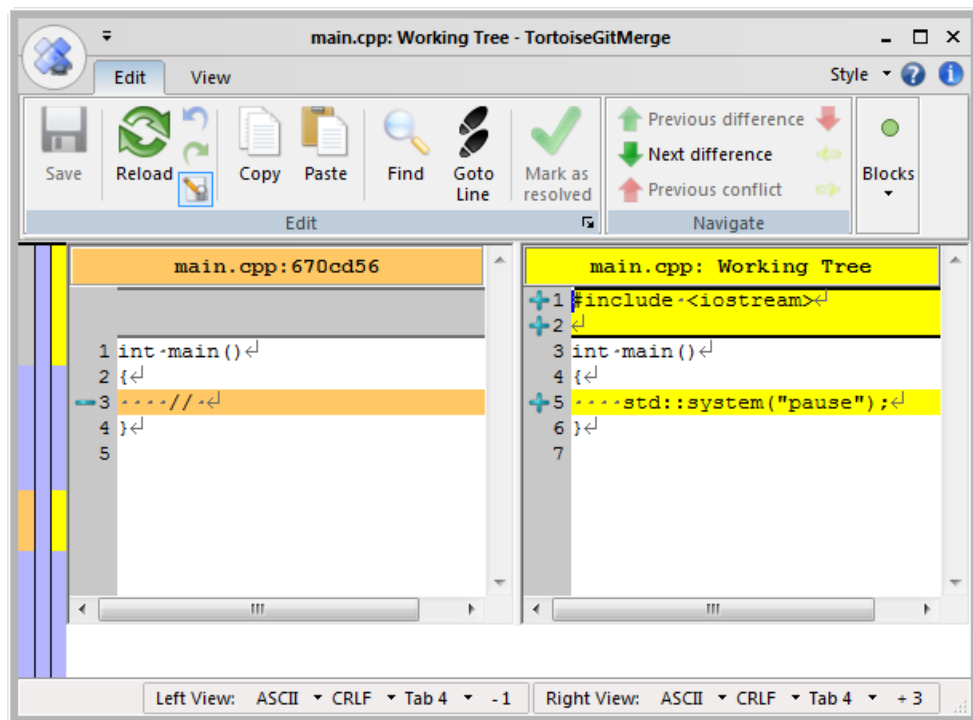
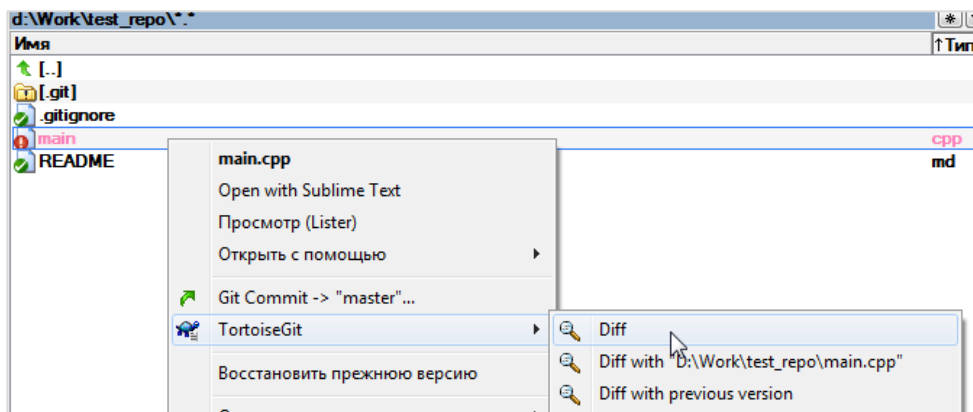
### *Внесение изменений в файл*

Продолжим работу над проектом. Допустим, мы решили внести изменения в наш файл. Изменим содержимое **main.cpp** на следующий код:

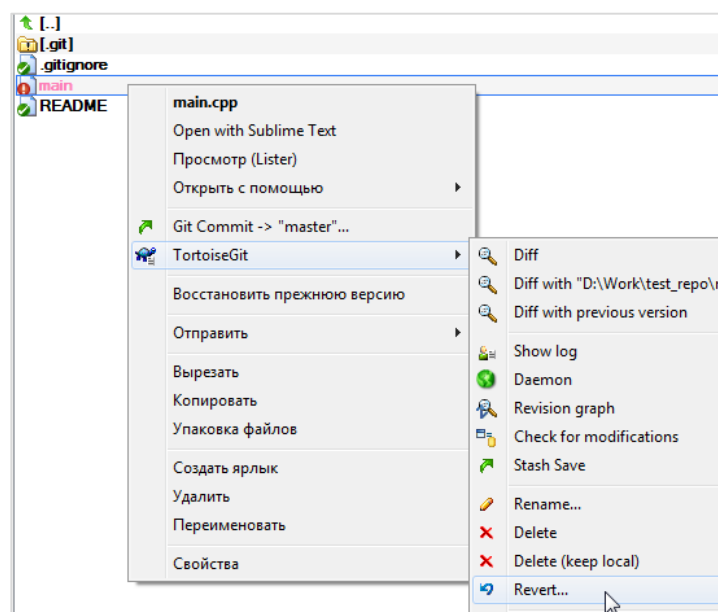
```
#include <iostream>

int main()
{
    std::system("pause");
}
```

Если теперь выполнить команду **diff**, можно увидеть изменения, которые были выполнены в файле:



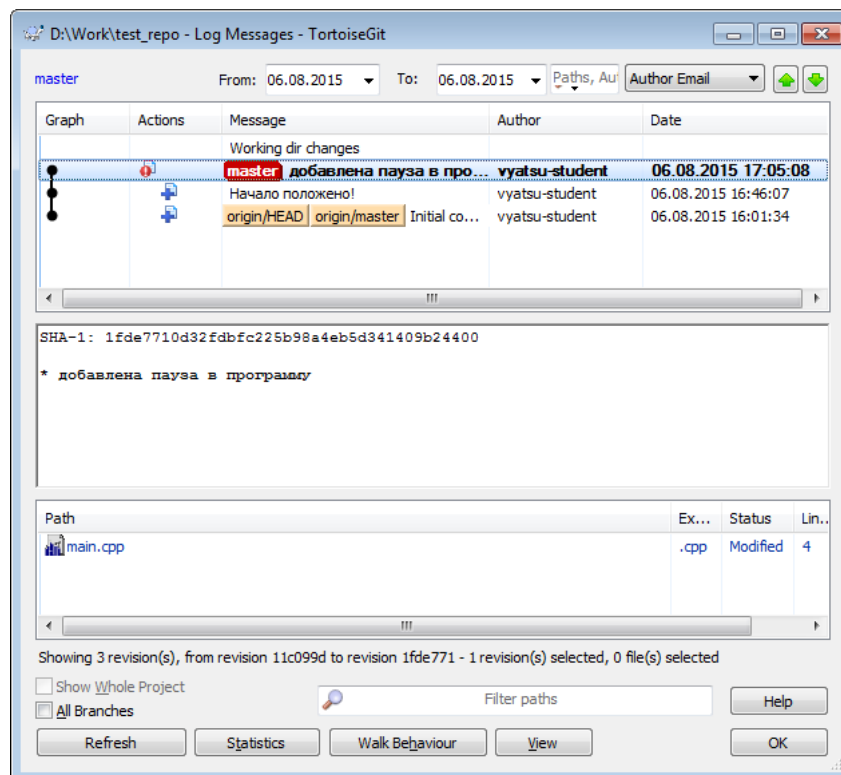
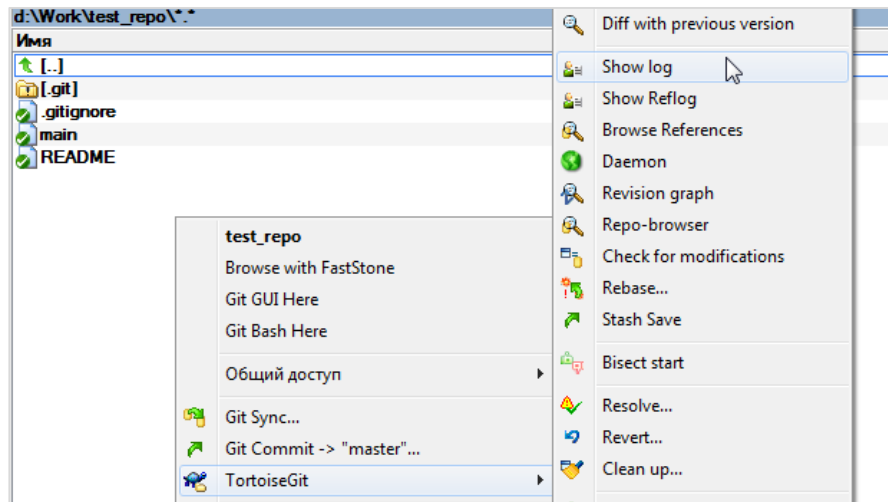
Если вы хотите откатить сделанные изменения, воспользуйтесь командой **revert**. Но будьте осторожны! Если вы отмените изменения с помощью **revert**, их уже невозможно будет вернуть.



Просмотрите после этого содержимое файла. Он восстановил свое состояние до изменений.

Внесите снова изменения в файл и зафиксируйте их с помощью команды **commit**.

Посмотрите журнал изменений с помощью команды **Show log**. Изменения можно посмотреть как для отдельного файла, подкаталога, так и для всего репозитория целиком:



Здесь можно увидеть журнал коммитов (фиксаций изменений), авторов и дату коммитов, комментарии к коммитам, а так же измененные файлы и статус (добавлен, удален, изменен). Как видим, для нашего репозитория было сделано 3 коммита.

#### *Возврат к предыдущей версии файла*

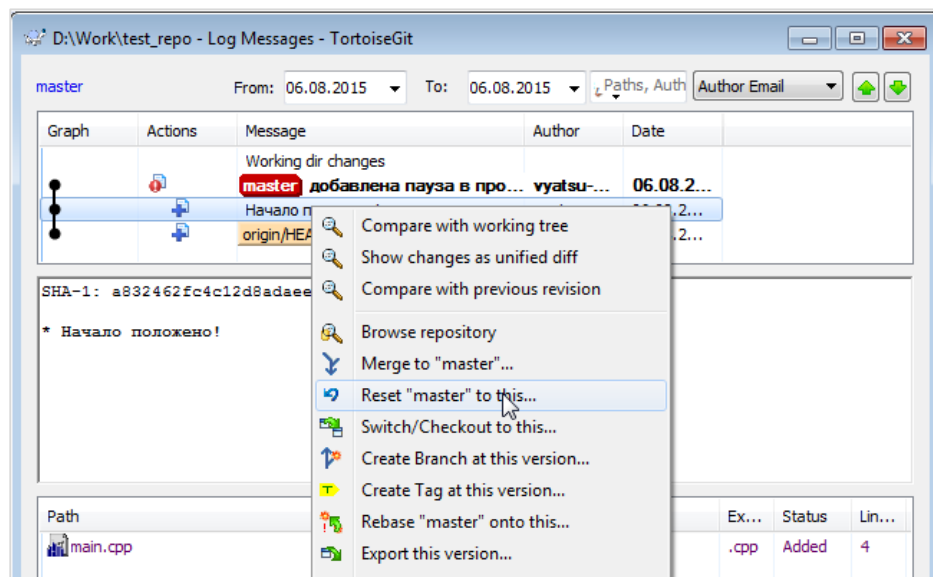
Как уже говорилось выше, назначение системы контроля версий (СКВ) – отслеживание изменений в файлах и возврат любого состояния из прошлого при необходимости. Бывает, что в процессе работы были допущены ошибки, и программа перестала работать. Требуется вернуть ее на последнее рабочее состояние. На практике, конечно, такое бывает не часто. Обычно, новую функциональность разрабатывают в отдельной ветке, не трогая основную, а когда функция дописана и протестирована, ветки сливают в одну.

Либо для нового разрабатываемого функционала коммит не делают до тех пор, пока функция не будет дописана и протестирована. И если что-то пошло не так, то можно отменить изменения с помощью **revert**.

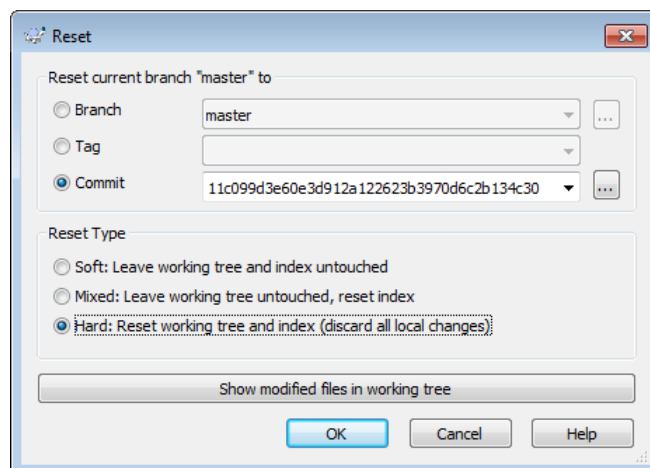
Но предположим, что мы сделали коммит программы с ошибкой, и хотим откатиться на один из предыдущих коммитов. Сделать это можно через журнал, вызвав контекстное меню на том коммите, к которому хотим вернуться.

Существуют разные способы откатиться к одному из предыдущих коммитов. Например, с помощью команды **checkout** или с помощью команды **reset**. Рассмотрим откат к предыдущему коммиту с помощью **reset**.

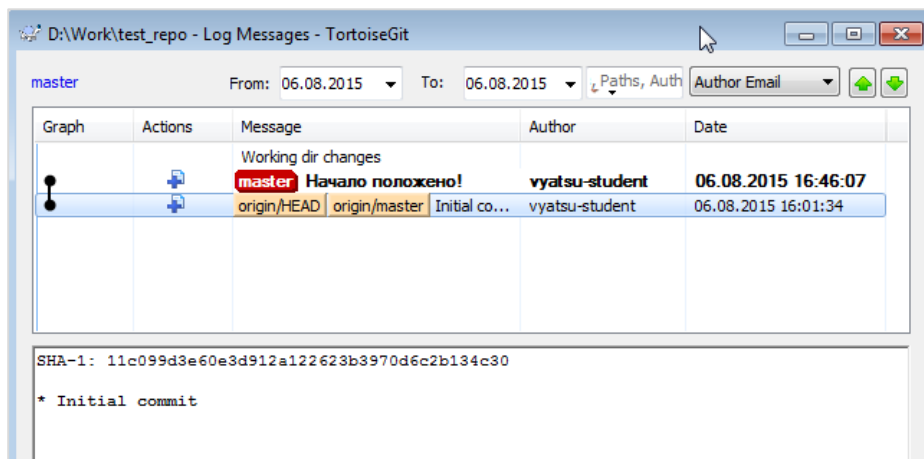
Выберите в контекстном меню журнала команду **Reset "master" to this...**



В появившемся окне выберите режим **Hard** – все изменения в репозитории, сделанные после этого коммита, будут утеряны.



Проверить текущее состояние можно по журналу. Видим, что последний коммит исчез, а мы находимся на предыдущем состоянии. Можете проверить также содержимое файла **main.cpp**. Он соответствует предпоследнему коммиту.

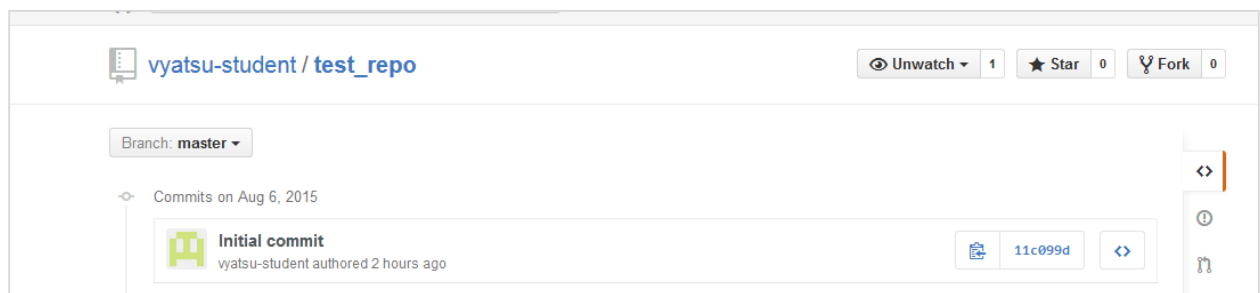


Описанных приемов работы с Git вполне достаточно для организации самостоятельной работы над проектом и отслеживания изменений в нем. Следующая часть методички посвящена синхронизации изменений между участниками проекта.

## Работа с удаленным сервером

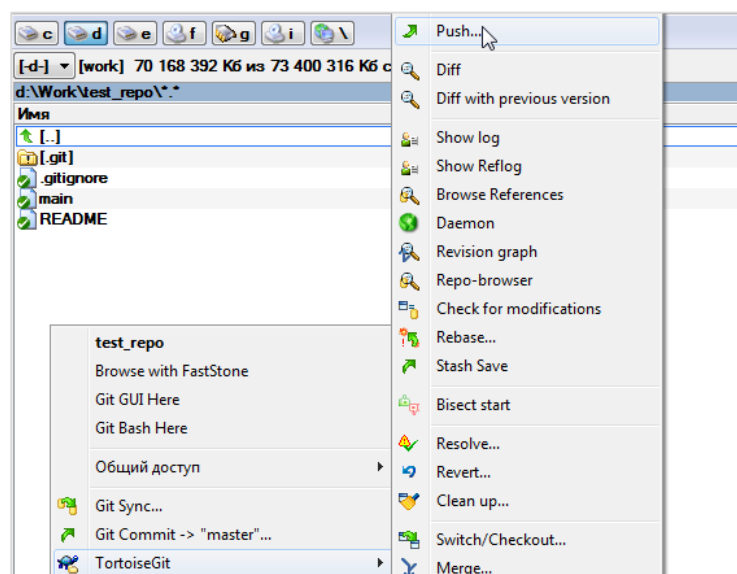
### Отправка изменений на сервер

Мы уже сделали несколько коммитов в репозитории, означающих фиксацию изменений в файлах проекта. Но если зайти сейчас на GitHub и посмотреть журнал изменений, там будет пусто:

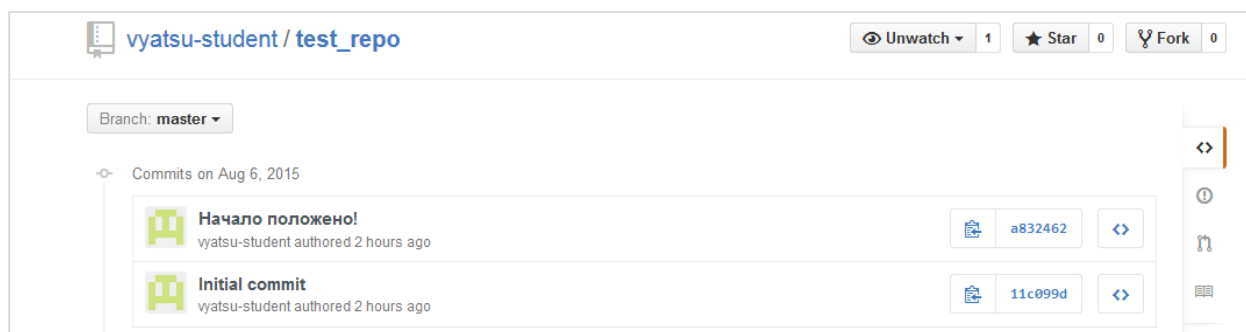


Сохранилась только информация о создании репозитория.

Дело в том, что мы работали только с локальной копией репозитория. Чтобы изменения попали на сервер и стали доступны для других участников, необходимо выполнить команду **Push**:



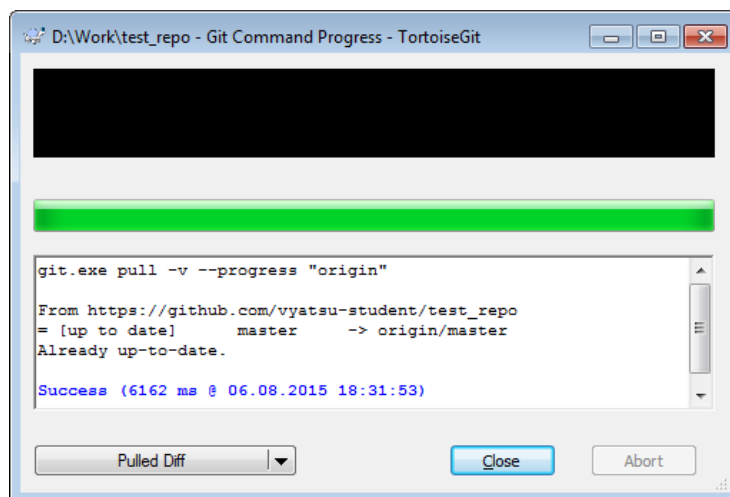
Оставьте все параметры по умолчанию. Система запросит имя пользователя и пароль. После их ввода изменения будут записаны на сервер. Проверим это:



Как видим, изменения появились на сервере и стали доступны участникам проекта.

#### *Получение последней версии проекта с сервера*

Чтобы получить актуальную версию с сервера, нужно выполнить команду **Pull**. Оставьте все параметры команды по умолчанию. По завершении операции вы увидите такое окно:



Это значит, что у вас уже имеется самая актуальная версия, что не удивительно, ведь никто кроме нас репозиторий не трогал.

#### *Процедура работы с Git: commit, pull, улаживание конфликтов, push*

В процессе работы над одним проектом разные участники могут вносить изменения в один и тот же файл. Чтобы избежать сложностей с объединением изменений, сделанных разными людьми, следует придерживаться следующей последовательности разработки:

1. Вы работаете над проектом, вносите изменения, делаете **коммит** (фиксацию) изменений
2. Делаете **Pull** актуальной версии с сервера. Ваши изменения и изменения с сервера автоматически объединятся. Если же встретится ситуация, что вы изменили какой-то файл одновременно с другим участником, система сообщит о конфликте и предложит его разрешить. В составе Git имеется удобная программа для решения конфликтов.
3. После того, как все конфликты решены (если они были), необходимо выполнить команду **Push**, чтобы отправить свои изменения на сервер.

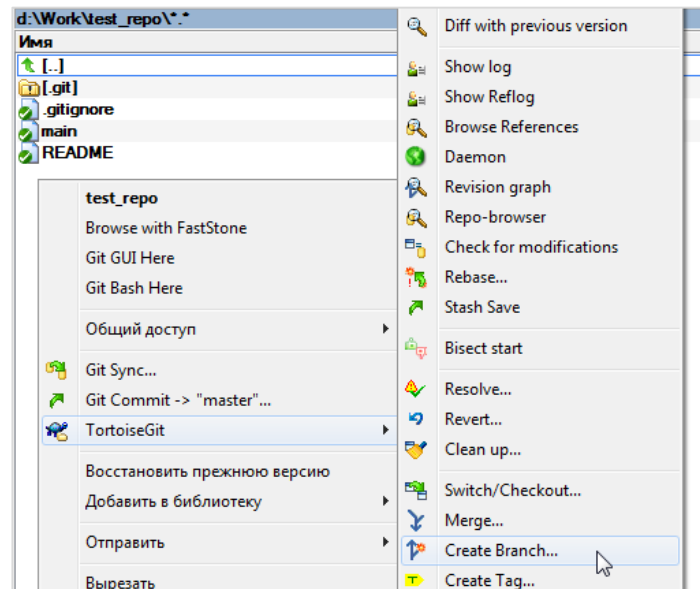
Такой подход хорошо зарекомендовал себя на практике. Разрешение конфликтов – редкая ситуация, так как, как правило, каждый участник работает над своей частью проекта, и других не трогает.

## Работа с ветками

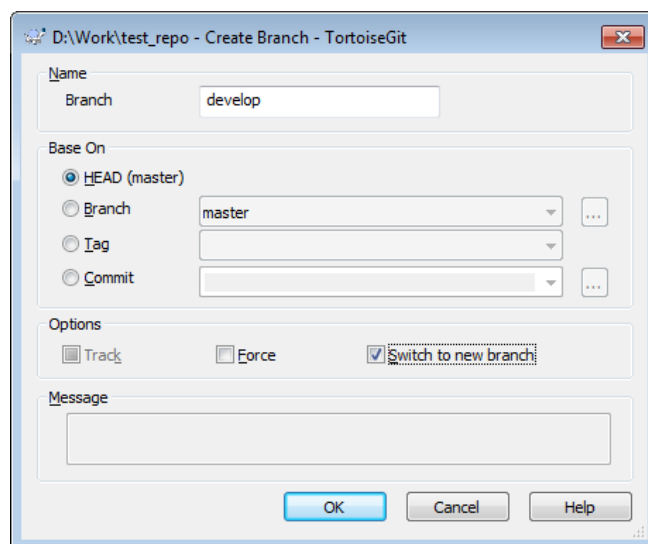
Рассмотрим еще одну важную сторону Git, которая делает его столь привлекательным, особенно для реализации крупных промышленных систем – работа с ветками. При разработке новой функции бывает полезно выполнять ее в отдельной ветке, чтобы в главной ветке находился только протестированный готовый к релизу код. Разработку новых функций можно вести в ветке **develop**. Либо под каждую новую функцию создавать отдельную ветку. В Git делать это очень просто.

### Создание ветки *develop* и переключение на нее

Создайте ветку и переключитесь на нее. Для этого используйте команду **Create Branch...**:



В появившемся окне задайте имя ветки – **develop** и отметьте галочку **Switch to new branch**.



Теперь, если вы будете вносить изменения в этой ветке, они не затронут основную ветку **master**.

Измените содержимое файла **main.cpp** на такое:

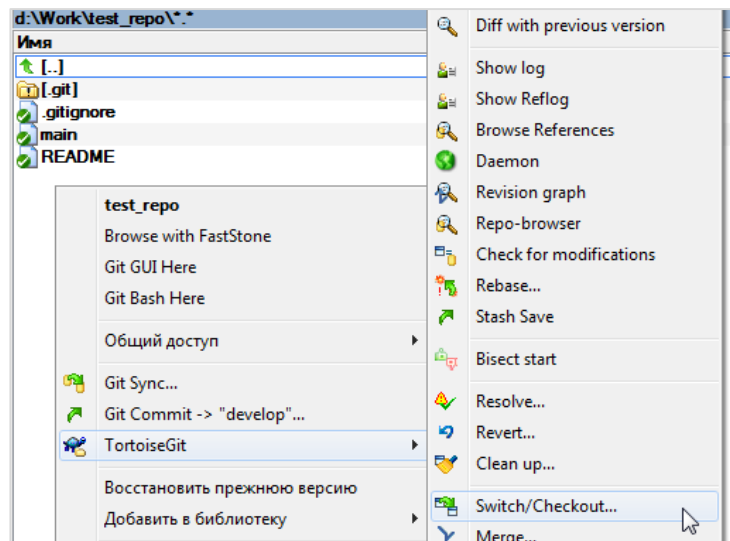
```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    std::system("pause");
}
```



Выполните коммит.

Теперь переключитесь на ветку **master** (команда **Switch/Checkout**) и посмотрите содержимое файла **main.cpp**.

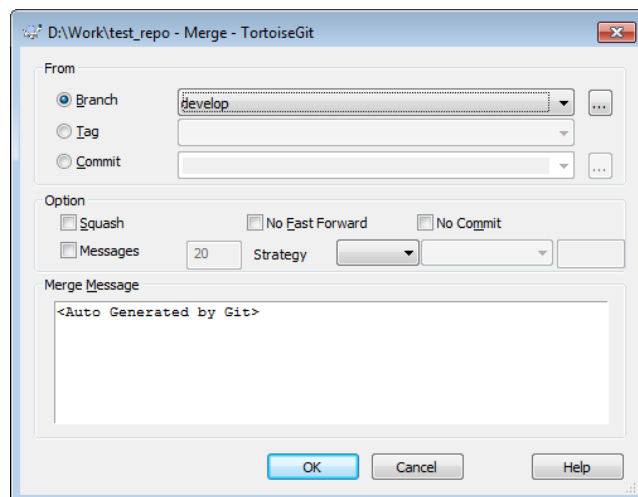


Как видите, его содержимое осталось прежним, а изменения произошли только в ветке **develop**.

#### Слияние веток

После того, как новая функция разработана и протестирована, необходимо применить изменения ветки **develop** к ветке **master**. Для этого переключитесь на ветку **master** и выполните команду **Merge...**

В появившемся окне укажите ветку, из которой нам нужно взять изменения. В нашем случае это **develop**:

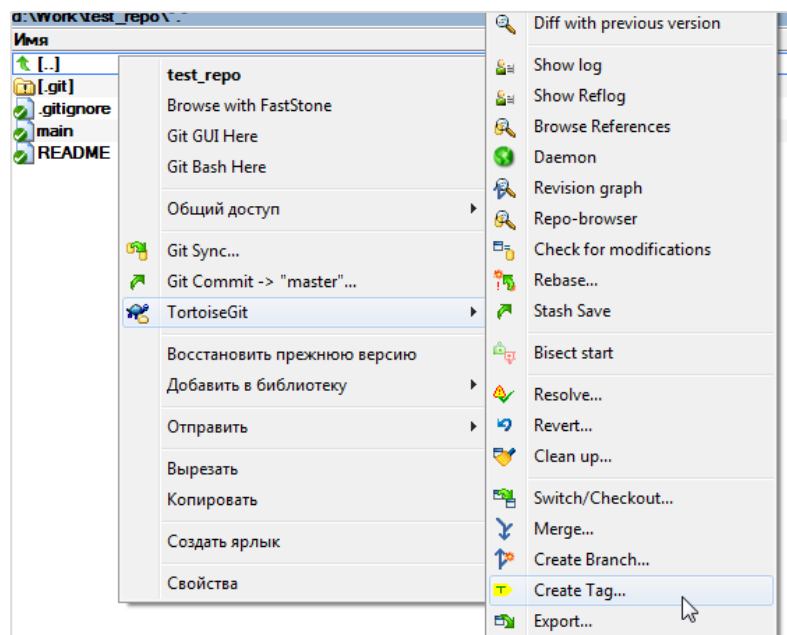


Нажмите кнопку Ok. Теперь содержимое ветки **develop** перенесено в ветку **master**, и файлы в обеих ветках полностью идентичны.

#### Создание тегов

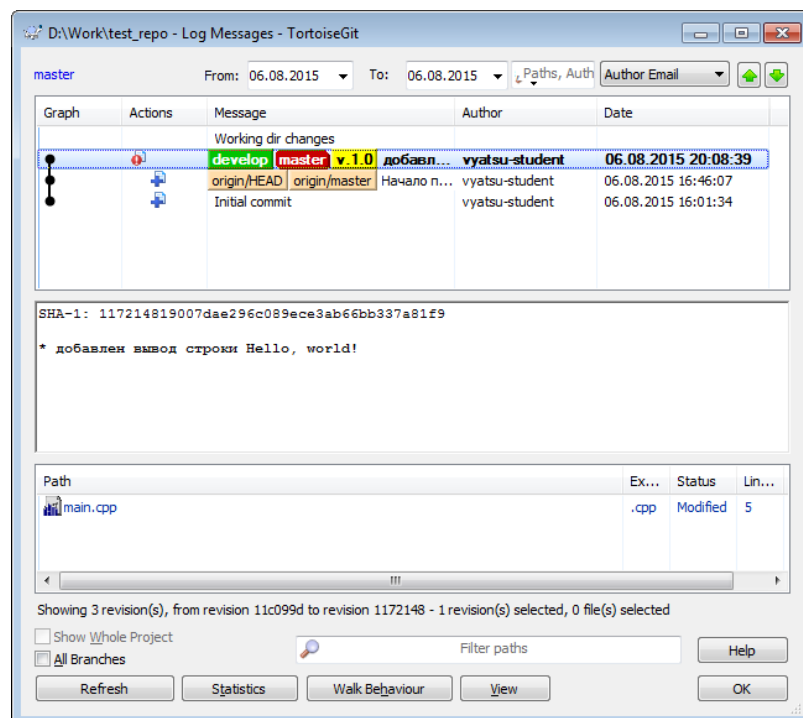
Еще одной интересной возможностью Git является создание тегов. Теги – это именованные метки, которые позволяют выделять некоторые коммиты. Например, коммит может соответствовать выходу бета-версии программы, и этот факт можно отметить тегом.

Чтобы создать тег, откройте журнал, откройте контекстное меню на интересующем коммите и выполните команду **Create Tag...**



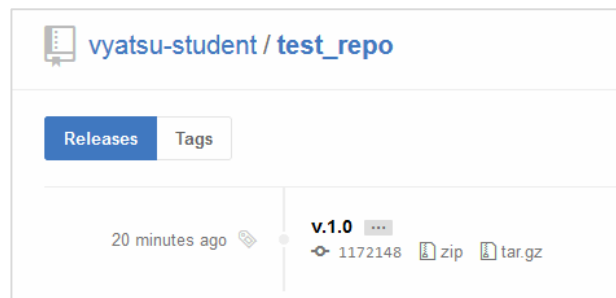
В появившемся окне укажите имя тега и нажмите **Ок**.

Откройте журнал. Коммит будет помечен тегом.



С помощью команды **Push** отправьте изменения на сервер. При выполнении команды **Push**, отметьте галочку **Include Tags**, чтобы созданный тег был отправлен на сервер.

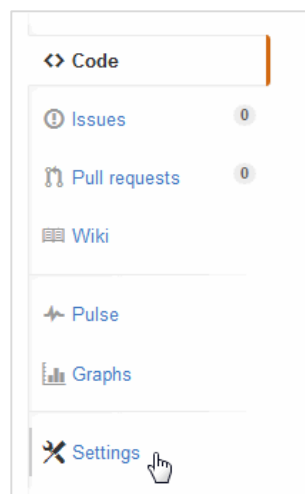
Теперь в GitHub зайдите на страницу репозитория. Созданный тег появился на сервере.



### Удаление репозитория с GitHub

Мы освоились с основными командами по работе с Git и GitHub. Репозиторий больше не нужен. Чтобы не засорять GitHub, удалите репозиторий.

Для этого на странице репозитория откройте настройки (Settings).



В самом низу найдите кнопку **Delete this repository**, нажмите ее, выслушайте предупреждения и удалите репозиторий.

Git предоставляет еще много полезных возможностей, например, слияние изменений, сделанных разными участниками в одном файле, или использование подмодулей (подпроектов в одном большом проекте). При необходимости, с этими возможностями можно ознакомиться самостоятельно. При выполнении курсовой работы они, скорее всего, не понадобятся.

### Задание на практику

1. Создать на GitHub репозиторий для курсовой работы
2. Создать в репозитории файл **task.txt** с формулировкой задания и именами участников
3. Пометить последний коммит тегом **v.0.1**