Системы хранилища версий

# Лабораторная работа №3

Цель лабораторной работы: научиться работать с системой управления версиями, ставить, отслеживать и выполнять поставленные задачи.

**Система управления версиями (от англ. Version Control System, VCS или Revision Control System)** — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных файлов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

## Общие сведения

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Традиционные системы управления версиями используют централизованную модель, когда имеется единое хранилище документов, управляемое специальным сервером, который и выполняет большую часть функций по управлению версиями. Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, так называемая «рабочая копия». Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Сервер может использовать т. н. дельта-компрессию — такой способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных. Поскольку обычно наиболее востребованной является последняя версия файла, система может при сохранении новой версии сохранять её целиком, заменяя в хранилище последнюю ранее сохранённую версию на разницу между этой и последней версией. Некоторые системы (например, ClearCase) поддерживают сохранение версий обоих видов: большинство версий сохраняется в виде дельт, но периодически (по специальной команде администратора) выполняется сохранение версий всех файлов в полном виде; такой подход обеспечивает максимально полное восстановление истории в случае повреждения репозитория.

Часто бывает, что над одним проектом одновременно работают несколько человек. Если два человека изменяют один и тот же файл, то один из них может случайно отменить изменения, сделанные другим. Системы управления версиями отслеживают такие конфликты и предлагают средства их решения. Большинство систем может автоматически объединить (слить) изменения, сделанные разными разработчиками. Однако такое автоматическое объединение изменений, обычно, возможно только для текстовых файлов и при условии, что изменялись разные (непересекающиеся) части этого файла. Такое ограничение связано с тем, что большинство систем управления версиями ориентированы на поддержку процесса разработки программного обеспечения, а исходные коды программ хранятся в текстовых файлах. Если автоматическое объединение выполнить не удалось, система может предложить решить проблему вручную.

Часто выполнить слияние невозможно ни в автоматическом, ни в ручном режиме, например, если формат файла неизвестен или слишком сложен. Некоторые системы управления версиями дают возможность заблокировать файл в хранилище. Блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла (например, средствами файловой системы) и обеспечивает, таким образом, исключительный доступ только тому пользователю, который работает с документом.

Многие системы управления версиями предоставляют ряд других возможностей:

* Позволяют создавать разные варианты одного документа, т. н. ветки, с общей историей изменений до точки ветвления и с разными — после неё;
* Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле;
* Ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии;
* Контролируют права доступа пользователей, разрешая или запрещая чтение или изменение данных, в зависимости от того, кто запрашивает это действие.

## Типичный порядок работы с системой

Каждая система управления версиями имеет свои специфические особенности в наборе команд, порядке работы пользователей и администрировании. Тем не менее, общий порядок работы для большинства VCS совершенно стереотипен. Здесь предполагается, что проект, каким бы он ни был, уже существует и на сервере размещён его репозиторий, к которому разработчик получает доступ.

### Начало работы с системой

Первым действием, которое должен выполнить разработчик, является извлечение рабочей копии проекта или той его части, с которой предстоит работать. Это действие выполняется с помощью стандартной команды извлечения версии (checkout или clone) либо специальной команды, фактически выполняющей то же самое действие. Разработчик задаёт версию, которая должна быть скопирована, по умолчанию обычно копируется последняя (или выбранная администратором в качестве основной) версия.

По команде извлечения устанавливается соединение с сервером, и проект (или его часть — один из каталогов с подкаталогами) в виде дерева каталогов и файлов копируется на компьютер разработчика.

Обычной практикой является дублирование рабочей копии: помимо основного каталога с проектом на локальный диск (либо в отдельный, специально выбранный каталог, либо в системные подкаталоги основного дерева проекта) дополнительно записывается ещё одна его копия. Работая с проектом, разработчик изменяет только файлы основной рабочей копии. Вторая локальная копия хранится в качестве эталона, позволяя в любой момент без обращения к серверу определить, какие изменения внесены в конкретный файл или проект в целом и от какой версии была «отпочкована» рабочая копия; как правило, любая попытка ручного изменения этой копии приводит к ошибкам в работе программного обеспечения VCS.

### Ежедневный цикл работы

При некоторых вариациях, определяемых особенностями системы и деталями принятого бизнес-процесса, обычный цикл работы разработчика в течение рабочего дня выглядит следующим образом:

* **Обновление рабочей копии.** По мере внесения изменений в основную версию проекта рабочая копия на компьютере разработчика стареет: расхождение её с основной версией проекта увеличивается. Это повышает риск возникновения конфликтных изменений (см. ниже). Поэтому удобно поддерживать рабочую копию в состоянии, максимально близком к текущей основной версии, для чего разработчик выполняет операцию обновления рабочей копии (update) насколько возможно часто (реальная частота обновлений определяется частотой внесения изменений, зависящей от активности разработки и числа разработчиков, а также временем, затрачиваемым на каждое обновление — если оно велико, разработчик вынужден ограничивать частоту обновлений, чтобы не терять время);
* **Модификация проекта.** Разработчик модифицирует проект, изменяя входящие в него файлы в рабочей копии в соответствии с проектным заданием. Эта работа производится локально и не требует обращений к серверу VCS;
* **Фиксация изменений.** Завершив очередной этап работы над заданием, разработчик фиксирует (commit) свои изменения, передавая их на сервер (либо в основную ветвь, если работа над заданием полностью завершена, либо в отдельную ветвь разработки данного задания). VCS может требовать от разработчика перед фиксацией обязательно выполнить обновление рабочей копии. При наличии в системе поддержки отложенных изменений (shelving) изменения могут быть переданы на сервер без фиксации. Если утверждённая политика работы в VCS это позволяет, то фиксация изменений может проводиться не ежедневно, а только по завершении работы над заданием; в этом случае до завершения работы все связанные с заданием изменения сохраняются только в локальной рабочей копии разработчика.

### Ветвления

Делать мелкие исправления в проекте можно путём непосредственной правки рабочей копии и последующей фиксацией изменений прямо в главной ветви (стволе) на сервере. Однако при выполнении сколько-нибудь значительных по объёму работ такой порядок становится неудобным: отсутствие фиксации промежуточных изменений на сервере не позволяет работать над чем-либо в групповом режиме, кроме того, повышается риск потери изменений при локальных авариях и теряется возможность анализа и возврата к предыдущим вариантам кода в пределах данной работы. Поэтому для таких изменений обычной практикой является создание ветвей (branch), то есть «отпочковывания» от ствола в какой-то версии нового варианта проекта или его части, разработка в котором ведётся параллельно с изменениями в основной версии. Ветвь создаётся специальной командой. Рабочая копия ветви может быть создана заново обычным образом (командой извлечения рабочей копии, с указанием адреса или идентификатора ветви), либо путём переключения имеющейся рабочей копии на заданную ветвь.

Базовый рабочий цикл при использовании ветвей остаётся точно таким же, как и в общем случае: разработчик периодически обновляет рабочую копию (если с ветвью работает более одного человека) и фиксирует в ней свою ежедневную работу. Иногда ветвь разработки так и остаётся самостоятельной (когда изменения порождают новый вариант проекта, который далее развивается отдельно от основного), но чаще всего, когда работа, для которой создана ветвь, выполнена, ветвь реинтегрируется в ствол (основную ветвь). Это может делаться командой слияния (обычно merge), либо путём создания патча (patch), содержащего внесённые в ходе разработки ветви изменения и применения этого патча к текущей основной версии проекта.

### Слияние версий

Три вида операций, выполняемых в системе управления версиями, могут приводить к необходимости объединения изменений. Это:

* Обновление рабочей копии (изменения, сделанные в основной версии, сливаются с локальными);
* Фиксация изменений (локальные изменения сливаются с изменениями, уже зафиксированными в основной версии);
* Слияние ветвей (изменения, сделанные в одной ветви разработки, сливаются с изменениями, сделанными в другой).

Во всех случаях ситуация принципиально одинакова и имеет следующие характерные черты:

1. Ранее была сделана копия дерева файлов и каталогов репозитория или его части.
2. Впоследствии и в оригинальное дерево, и в копию были независимо внесены некоторые изменения.
3. Требуется объединить изменения в оригинале и копии таким образом, чтобы не нарушить логическую связность проекта и не потерять данные.

Совершенно очевидно, что при невыполнении условия (2) (то есть если изменения были внесены только в оригинал или только в копию) объединение элементарно — достаточно скопировать изменённую часть туда, где изменений не было. В противном случае слияние изменений превращается в нетривиальную задачу, во многих случаях требующую вмешательства разработчика. В целом механизм автоматического слияния изменений работает, основываясь на следующих принципах:

* Изменения могут состоять в модификации содержимого файла, создании нового файла или каталога, удалении или переименовании ранее существовавшего файла или каталога в проекте.
* Если два изменения относятся к разным и не связанным между собой файлам и/или каталогам, они всегда могут быть объединены автоматически. Их объединение состоит в том, что изменения, сделанные в каждой версии проекта, копируются в объединяемую версию.
* Создание, удаление и переименование файлов в каталогах проекта могут быть объединены автоматически, если только они не конфликтуют между собой. В этом случае изменения, сделанные в каждой версии проекта, копируются в объединяемую версию. Конфликтующими обычно являются:
  + Удаление и изменение одного и того же файла или каталога.
  + Удаление и переименование одного и того же файла или каталога (в случае, если система поддерживает операцию переименования).
  + Создание в разных версиях файла с одним и тем же именем и разным содержимым.
* Изменения в пределах одного текстового файла, сделанные в разных версиях, могут быть объединены, если они находятся в разных местах этого файла и не пересекаются. В этом случае в объединённую версию вносятся все сделанные изменения.
* Изменения в пределах одного файла, если он не является текстовым, всегда являются конфликтующими и не могут быть объединены автоматически.

Во всех случаях базовой версией для слияния является версия, в которой было произведено разделение сливаемых версий. Если это операция фиксации изменений, то базовой версией будет версия последнего обновления перед фиксацией, если обновление — то версия предыдущего обновления, если слияние ветвей — то версия, в которой была создана соответствующая ветвь. Соответственно, сопоставляемыми наборами изменений будут наборы изменений, сделанных от базовой до текущей версии во всех объединяемых вариантах.

Абсолютное большинство современных систем управления версиями ориентировано, в первую очередь, на проекты разработки программного обеспечения, в которых основным видом содержимого файла является текст. Соответственно, механизмы автоматического слияния изменений ориентируются на обработку текстовых файлов, то есть файлов, содержащих текст, состоящий из строк буквенно-цифровых символов, пробелов и табуляций, разделённых символами перевода строки.

При определении допустимости слияния изменений в пределах одного и того же текстового файла работает типовой механизм построчного сравнения текстов (примером его реализации является системная утилита GNU diff), который сравнивает объединяемые версии с базовой и строит список изменений, то есть добавленных, удалённых и заменённых наборов строк. Минимальной единицей данных для этого алгоритма является строка, даже самое малое отличие делает строки различными. С учётом того, что символы-разделители, в большинстве случаев, не несут смысловой нагрузки, механизм слияния может игнорировать эти символы при сравнении строк.

Те найденные наборы изменённых строк, которые не пересекаются между собой, считаются совместимыми и их слияние делается автоматически. Если в сливаемых файлах находятся изменения, затрагивающие одну и ту же строку файла, это приводит к конфликту. Такие файлы могут быть объединены только вручную. Любые файлы, кроме текстовых, с точки зрения VCS являются бинарными и не допускают автоматического слияния.

### Конфликты и их разрешения

Ситуация, когда при слиянии нескольких версий сделанные в них изменения пересекаются между собой, называют конфликтом. При конфликте изменений система управления версиями не может автоматически создать объединённый проект и вынуждена обращаться к разработчику. Как уже говорилось выше, конфликты могут возникать на этапах фиксации изменений, обновления или слияния ветвей. Во всех случаях при обнаружении конфликта соответствующая операция прекращается до его разрешения.

Для разрешения конфликта система, в общем случае, предлагает разработчику три варианта конфликтующих файлов: базовый, локальный и серверный. Конфликтующие изменения либо показываются разработчику в специальном программном модуле объединения изменений (в этом случае там демонстрируются сливаемые варианты и динамически изменяющийся в зависимости от команд пользователя объединённый вариант файла), либо просто помечаются специальной разметкой прямо в тексте объединённого файла (тогда разработчик должен сам сформировать желаемый текст в спорных местах и сохранить его).

Конфликты в файловой системе разрешаются проще: там может конфликтовать только удаление файла с одной из прочих операций, а порядок файлов в каталоге не имеет значения, так что разработчику остаётся лишь выбрать, какую операцию нужно сохранить в сливаемой версии.

### Блокировки

Механизм блокировки позволяет одному из разработчиков захватить в монопольное использование файл или группу файлов для внесения в них изменений. На то время, пока файл заблокирован, он остаётся доступным всем остальным разработчикам только на чтение, и любая попытка внести в него изменения отвергается сервером. Технически блокировка может быть организована по-разному. Типичным для современных систем является следующий механизм.

* Файлы, для работы с которыми требуется блокировка, помечаются специальным флагом «блокируемый». Такая пометка может ставиться автоматически при добавлении файла в проект, обычно для этого предварительно создаётся список масок имён файлов, которые при добавлении должны становиться блокируемыми;
* Если файл помечен как блокируемый, то при извлечении рабочей копии с сервера он получает в локальной файловой системе атрибут «только для чтения», что препятствует его случайному редактированию;
* Разработчик, желающий изменить файл, вызывает специальную команду блокировки (lock) с указанием имени этого файла. В результате работы этой команды происходит следующее:
  + сервер проверяет, не заблокирован ли уже файл другим разработчиком; если это так, то команда блокировки завершается с ошибкой «файл заблокирован другим пользователем» и разработчик, вызывавший её, должен ожидать, пока другой пользователь не снимет свою блокировку;
  + файл на сервере помечается как «заблокированный», с сохранением идентификатора заблокировавшего его разработчика и времени блокировки;
  + если блокировка на сервере прошла удачно, на локальной файловой системе с файла рабочей копии снимается атрибут «только для чтения», что позволяет начать его редактировать;
* Разработчик работает с заблокированным файлом. Если в процессе работы выясняется, что файл изменять не нужно, он может вызвать команду снятия блокировки (unlock, release lock). Все изменения файла будут отменены, локальный файл вернётся в состояние «только для чтения», с файла на сервере будет снят атрибут «заблокирован» и другие разработчики получат возможность изменять этот файл;
* По завершении работы с блокируемым файлом разработчик фиксирует изменения. Обычно блокировка при этом снимается автоматически, хотя в некоторых системах блокировку требуется снимать вручную после фиксации, либо указывать в команде фиксации изменений соответствующий параметр. Так или иначе, при этом файл после изменений теряет флаг «заблокирован» и может быть изменён другими разработчиками.

Массовое использование блокировок, когда все или большинство файлов в проекте являются блокируемыми и для любых изменений необходимо заблокировать соответствующий набор файлов, называется ещё стратегией «блокированного извлечения». Ранние системы управления версиями поддерживали исключительно эту стратегию, предотвращая таким способом появление конфликтов на корню. В современных VCS предпочтительным является использование неблокирующих извлечений, блокировки же считаются скорее неизбежным злом, которое нужно по возможности ограничивать. Недостатки использования блокировок очевидны:

* Блокировки просто мешают продуктивной работе, поскольку вынуждают ожидать освобождения блокированных файлов, хотя в большинстве случаев даже совместные изменения одних и тех же файлов, которые делаются в ходе разных по смыслу работ, не пересекаются и объединяются при слиянии автоматически;
* Частота возникновения конфликтов и сложность их разрешения в большинстве случаев не настолько велики, чтобы создать серьёзные затруднения. Возникновение же серьёзного конфликта изменений чаще всего сигнализирует либо о существенном расхождении во мнениях разных разработчиков относительно дизайна одного и того же фрагмента, либо о неправильной организации работы (когда два или более разработчиков делают одно и то же);
* Блокировки создают административные проблемы. Типичный пример: разработчик может забыть снять блокировку с занятых им файлов, уходя в отпуск. Для разрешения подобных проблем приходится применять административные меры, в том числе включать в систему технические средства для сброса неверных блокировок, но и при их наличии на приведение системы в порядок расходуется время.

С другой стороны, в некоторых случаях использование блокировок вполне оправданно. Очевидным примером является организация работы с бинарными файлами, для которых нет инструментальных средств слияния изменений либо такое слияние принципиально невозможно (как, например, для файлов изображений). Если автоматическое слияние невозможно, то при обычном порядке работы любое параллельное изменение подобных файлов будет приводить к конфликту. В данном случае гораздо удобнее сделать такой файл блокируемым, чтобы гарантировать, что любые изменения в него будут вноситься только последовательно.

### Версии проекта, теги

Система управления версиями обеспечивает хранение всех существовавших вариантов файлов и, как следствие, всех вариантов проекта в целом, имевших место с момента начала его разработки. Но само понятие «версии» в разных системах может трактоваться двумя различными способами:

* Одни системы поддерживают версионность файлов. Это означает, что любой файл, появляющийся в проекте, получает собственный номер версии (обычно — номер 1, условной «нулевой» версией файла считается пустой файл с тем же именем). При каждой фиксации разработчиком изменений, затрагивающих файл, соответствующая часть фиксируемых изменений применяется к файлу и файл получает новый, обычно следующий по порядку, номер версии. Поскольку фиксации обычно затрагивают только часть файлов в репозитории, номера версий файлов, имеющиеся на один и тот же момент времени, со временем расходятся, и проект в целом (то есть весь набор файлов репозитория), фактически, никакого «номера версии» не имеет, поскольку состоит из множества файлов с различными номерами версий. Подобным образом работает, например, система управления версиями CVS.
* Для других систем понятие «версия» относится не к отдельному файлу, а к репозиторию целиком. Вновь созданный пустой репозиторий имеет версию 1 или 0, любая фиксация изменений приводит к увеличению этого номера (то есть даже при изменении одного файла на один байт весь репозиторий считается изменённым и получает новый номер версии). Таким способом трактует номера версий, например, система Subversion. Номера версии отдельного файла здесь, фактически, не существует, условно можно считать таковым текущий номер версии репозитория (то есть считать, что при каждом изменении, внесённом в репозиторий, все его файлы меняют номер версии, даже те, которые не менялись). Иногда, говоря о «версии файла» в таких системах, имеют в виду ту версию репозитория, в которой файл был последний раз (до интересующего нас момента) изменён.

Для практических целей обычно имеет значение не отдельный файл, а весь проект целиком. В системах, поддерживающих версионность отдельных файлов, для идентификации определённой версии проекта можно использовать дату и время — тогда версия проекта будет состоять из тех версий входящих в него файлов, которые имелись в репозитории на указанный момент времени. Если поддерживается версионность репозитория в целом, номером версии проекта может выступать номер версии репозитория. Однако оба варианта не слишком удобны, так как ни дата, ни номер версии репозитория обычно не несут информации о значимых изменениях в проекте, о том, насколько долго и интенсивно над ним работали. Для более удобной пометки версий проекта (или его частей) системы управления версиями поддерживают понятие тегов.

**Тег (tag)** — это символическая метка, которая может быть связана с определённой версией файла и/или каталога в репозитории. С помощью соответствующей команды всем или части файлов проекта, отвечающим определённым условиям (например, входящим в головную версию главной ветви проекта на определённый момент времени) может быть присвоена заданная метка. Таким образом можно идентифицировать версию проекта (версия «XX.XXX.XXX» — это набор версий файлов репозитория, имеющих тег «XX.XXX.XXX»), зафиксировав таким образом его состояние на некоторый желаемый момент. Как правило, система тегов достаточно гибкая и позволяет пометить одним тегом и не одновременные версии файлов и каталогов. Это позволяет собрать «версию проекта» любым произвольным образом. С точки зрения пользователя системы пометка тегами может выглядеть по-разному. В некоторых системах она изображается именно как пометка (тег можно создать, применить к определённым версиям файлов и каталогов, снять). В других системах (например, Subversion) тег представляет собой просто отдельный каталог на файловом дереве репозитория, куда из ствола и ветвей проекта с помощью команды копирования делаются копии нужных версий файлов. Так что визуально тег — это просто вынесенная в отдельный каталог копия определённых версий файлов репозитория. По соглашению в дерево каталогов, соответствующее тегу, запрещена фиксация изменений (то есть версия проекта, представляемая тегом, является неизменной).

## Основы использования веток для параллельной разработки

Рассмотрим управление релизами: Релиз-менеджер может оценить состояние кода и выбрать реализованный функционал для включения в релиз. Этот функционал должен быть готов и протестирован. Также релиз-менеджер может включить исправления дефектов с прошлого релиза. Неготовый, нестабильный и не протестированный функционал в релиз попасть не должен. Если от QA-специалистов поступает информация о нестабильности того или иного функционала, релиз-менеджер должен иметь возможность убрать его из релиза. Часто возникает потребность в переносе исправлений дефектов на уже работающую у конечного пользователя версию, потому что он по каким-то причинам не может перейти на новую.

Если посмотреть на процесс работы над кодом со стороны разработчика, то он должен сидеть в своей песочнице и не подвергаться влиянию дестабилизирующих коммитов со стороны коллег. В идеале, разработчики должны обмениваться только законченными и стабильными наборами изменений. Так проще понять что было сделано. Тем не менее, коммиты не должны диктовать разработчику стиль его работы, и он всегда должен иметь возможность добавить частично выполненный функционал.

Использование ветвления позволяет: выполнять стабилизацию релизной версии (процесс более известный как исправление багов) и одновременно осуществлять расширение приложения новым функционалом для следующего релиза. Это два основных, но не единственных, способа использования ветвления версий на проекте.

### Ветвление проекта при каждом релизе

Рассмотрим случай, когда отдельная ветка создается под каждый релиз. Это делается для того, чтобы исправлять дефекты, найденные после выпуска релиза или во время его тестирования. Этот процесс обычно называется стабилизацией. При этом сами исправления (багфиксы) не остаются только в релизных ветках, а переносятся в mainline (если история релиза и mainline не слишком разошлись), делая ее стабильнее. Код в релизной ветке изолирован от дестабилизирующего влияния разработки нового функционала и при этом не блокирует ее. Сама по себе релизная ветка предоставляет легкую возможность осуществлять поддержку релизной версии. Когда прекращается поддержка релиза, его ветка замораживается. А пока идет проект, mainline продолжает свое развитие, являясь точкой, в которой накапливается новый функционал для следующих релизов.

Таким же образом можно осуществлять поддержку релизов для разных заказчиков, выделяя по ветке для каждого, если по каким-то причинам им нельзя поставить одну и ту же версию. Необходимо отметить, что поддержка разных вариаций одной и той же версии — задача трудоемкая и ее следует избегать по мере возможности.

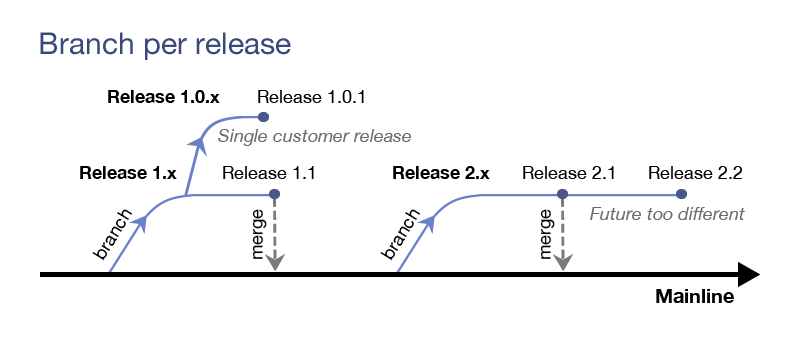


Рисунок 1 Ветвление проекта при релизах

### Ветвление при добавлении нового функционала

Также можно выделять отдельные ветки для разработки нового функционала. Как правило, это одна логически законченная функциональная область, или просто feature. Новый функционал объединяется с основной веткой только после полного завершения, что позволяет избежать негативного влияния незавершенной работы на другие линии разработки. После того как новый функционал готов и объединен с основной веткой, другие ветки разработки должны быть интегрированы с mainline, чтобы не накапливался эффект отложенной интеграции. Использование веток для релизов и разработки позволяет нам не ждать, пока окончится тестирование и стабилизация релиза, а сразу приступить к разработке функционала для следующего.

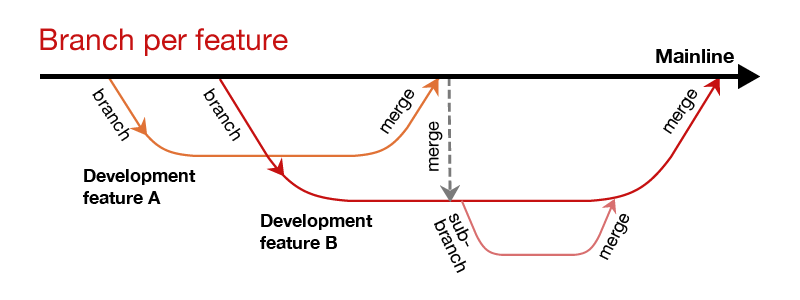


Рисунок 2 Ветвление проекта при новом функционале

Также можно создавать ветки для релизных и разработческих веток, если нужны еще уровни изоляции. Во всех случаях создания новой ветки следует понимать цену ее поддержки.

### Интеграция между ветками

Основная ветка (mainline, trunk) является главным местом интеграции при помощи кода. Все изменения, сделанные разработчиками, попадают сюда. Тем не менее, она не должна превращаться в свалку нестабильного и незаконченного кода. Поэтому, разработку новыого функционала рекомендуется проводить в отдельной ветке, интегрировать с основной, тестировать, и только потом объединять изменения. Иными словами, mainline должна содержать достаточно законченный код, который может послужить основой для стабилизационной релизной ветки. Также, исправления ошибок из релизных веток, пройдя через mainline, попадают в ветки для разработки, таким образом, работа ведется над более стабильным кодом. Хорошим правилом является то, что нельзя отдавать нестабильные изменения в другие ветки и что нужно должны принимать только стабильные изменения из других веток.

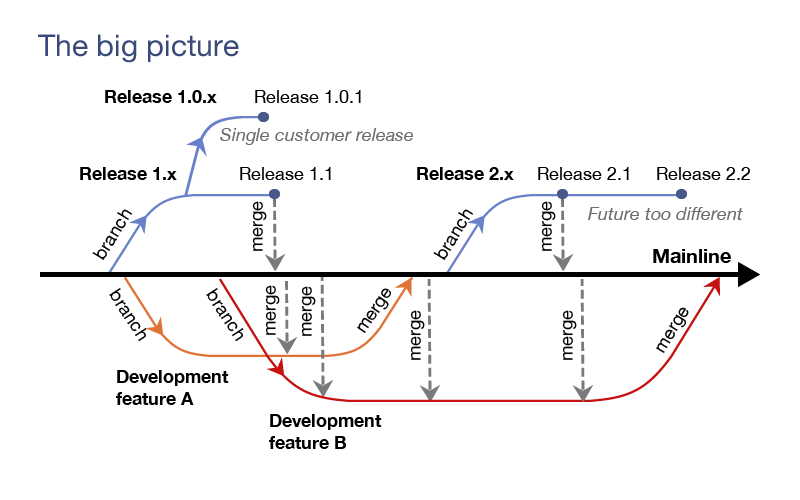


Рисунок 3 Ветвление, в общем

Рассмотрим ситуацию, отображенную на рисунке:

* В какой то момент в Mainline накопилось достаточное количество законченного функционала для выпуска Release 1.x. Для него была создана ветка, и после тестирования и стабилизации релиз ушел заказчикам;
* Параллельно с этим стартовала разработка нового функционала:feature A и feature B, – каждая на своей ветке;
* Баги, найденные закзачиками в Release 1.0, были исправлены на релизной ветке, и был выпущен Release 1.1. Багфиксы из него были объединены с Mainline, откуда попали в ветки для feature A и feature B. Таким образом, работа велась над более стабильным кодом;
* Один из заказчиков по своим причинам не смог перейти на версию 1.1 и столкнулся с рядом специфичных для себя дефектов. Это было исправлено на специально сделанной для него ветке – Release 1.0.x;
* Была закончена разработка feature A, и, после интеграции и тестирования, эти законченные изменения попали в Mainline. Ветка для feature B получает эти изменения сразу после их попадания в Mainline, чтобы работа велась над максимально актуальной версией кода;
* Принимается решение о выпуске нового Release 2.x, включающего feature A, и для него создается ветка, на которой осуществляется сервис этого релиза, – 2.1, 2.2. Причем, багфиксы для релизной версии 2.2 не объединяются с Mainline, так как истории этих линий разработки кода уже слишком разошлись.

Интеграция через Mainline не является единственным способом интегрироваться – возможна интеграция напрямую между ветками.

### Стабильность веток

У такой модели есть градация стабильности, где самыми стабильными являются релизные ветки, менее стабильной является mainline, и самыми нестабильными являются ветки для разработки. Как правило, на диаграммах самые стабильные ветки отображаются выше всех, а нестабильные – ниже всех.

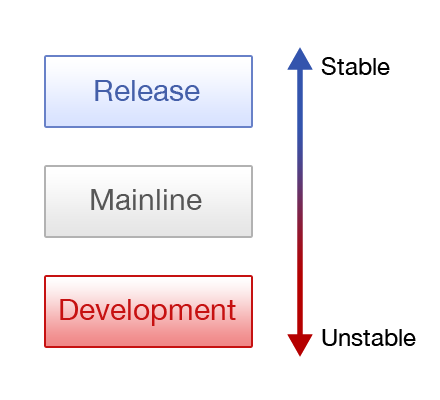


Рисунок 4 Стабильность веток

### Накладные расходы, связанные с использованием ветвления

С ветвлением связаны следующие издержки:

* **Механические** – это те действия, которые нужно совершить, чтобы создать ветку, переключиться с ветки на ветку, объединить (merge) изменения и т.п. Как правило, такие действия трудоемки для централизованных систем и относительно просты для децентрализованных;
* **Интеллектуальные** – это те усилия, которые приходится приложить, чтобы держать в голове все существующие ветки и их предназначение. Как правило, существуют инструменты, которые облегчают эту задачу. Сюда можно отнести кривую обучения для сотрудников, связанную с освоением системы управления версиями;
* **Цена за тестирование** – использование параллельной разработки способно серьезно увеличить цену ручного тестирования. Отложенное тестирование позволяет сократить расходы, но при этом имеет ряд своих недостатков. Любое автоматическое тестирование значительно уменьшает цену тестирования при использовании ветвления. В целом, этот пункт зависит от стратегии тестирования, принятой на проекте.

Типы зависимостей между ветками и способы их решения

Между ветками могут возникать следующие зависимости:

* **Архитектурные** – если мы меняем архитектуру на одной ветке, другие ветки могут зависеть от этих изменений;
* **Функциональные** – некоторая новая функциональность не может быть закончена или не имеет особой ценности, пока не будет закончен другой функционал, от которого она зависит;
* **Зависимости от исправления дефектов** – в случае исправления дефекта на одной ветке, может существовать несколько веток, которые должны получить это изменение.

Существует несколько типовых решений для работы с таким зависимостями:

* **Создание дочерних веток** – зависимая функциональность реализуется в отдельном под-ветке и потом объединяется со всеми заинтересованными ветками;
* **Остановка** – разработка на ветке замораживается, пока не будет готова нужная функциональность;
* **Архитектурная абстракция** – путем абстракции в системе создаются границы, которые изолируют разные части функциональности. В этом случае проблема решается не только на уровне системы управления версиями, но и на уровне дизайна приложения;
* **Использование заглушек** – в системе используются, так называемые, «заглушки», которые заменяются на реальный функционал по мере его готовности;
* **Релиз, патч, ре-релиз** – система выпускается в не полностью готовом виде и патчами доводится до совершенства (эту практику в некоторых отраслях принято называть платным бета-тестированием).

### Примеры некорректных подходов к ветвлению

* **Merge Paranoia** – разработчики боятся объединять код, поэтому накапливается негативный эффект отложенной интеграции;
* **Merge Mania** – разработчики больше времени тратят на объединение изменений, чем на разработку;
* **Big Bang Merge** – ветки не обмениваются законченными изменениями, поэтому происходит одно гигантское объединение в конце;
* **Never-Ending Merge** – объединение никогда не останавливается, так как всегда есть что объединять;
* **Wrong Way Merge** – объединение более поздней ветки разработки с более ранней версией;
* **Branch Mania** – создание большого количества веток без нужной на то причины;
* **Cascading Branches** – создание веток без объединения их с mainline в конце разработки;
* **Mysterious Branches** – создание ветки без причины;
* **Теmporary Branches** – создание ветки с изменяющейся причиной ее существования: ветка становится временным рабочим пространством;
* **Volatile Branches** – старт ветки в нестабильном состоянии или перенос нестабильных изменений в другие ветки;
* **Development Freeze** – остановка всей разработки для создания веток, объединения или создания релизов;
* **Berlin Wall** – использование веток для разделения людей в команде, вместо разделения областей, над которыми они работают.

## Работа с хранилищем версий

В данном разделе описывается пример работы с хранилищем версий Git, с использованием хостинга проектов GitHub.

### Регистрация в GitHub

Регистрация производится на официальном сайте GitHub: <https://github.com>

После регистрации на адрес электронной почты придёт ссылка с подтверждением адреса.

На этом этап регистрации завершён.

### Главная страница GitHub

Для новых пользователей страница имеет следующий вид:

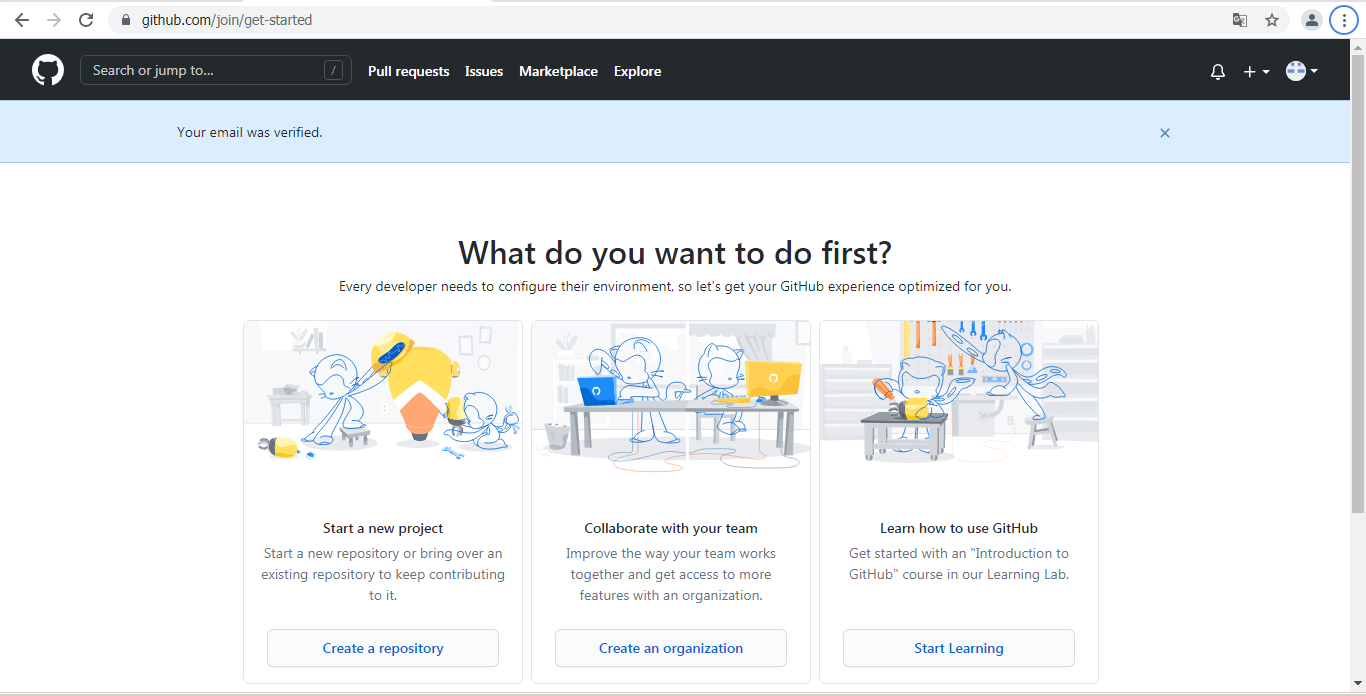


Рисунок 6 Главная страница GitHub для нового пользователя

### Создание нового проекта

Для начала работы необходимо создать новый проект. Это делается нажатием на кнопку «Create a repository».

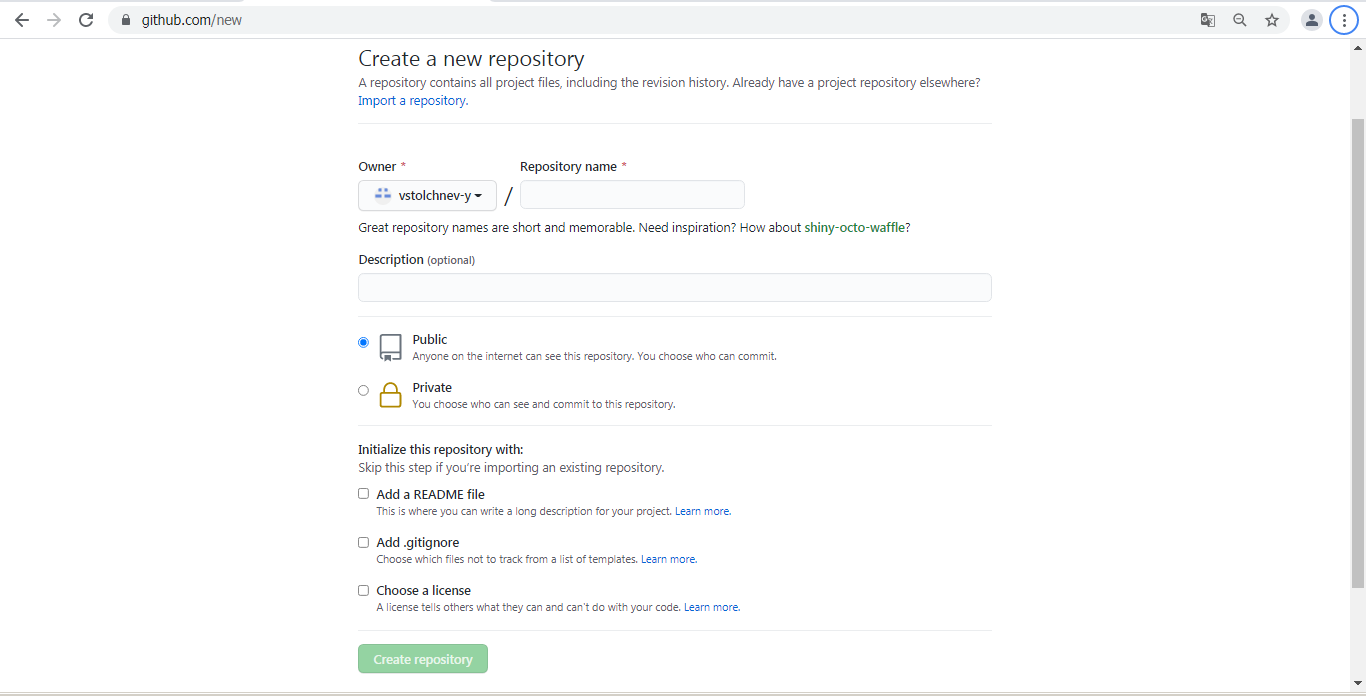


Рисунок 7 Создание нового проекта

Рассмотрим типовые поля, которые необходимо заполнить на данной форме:

* **Repository name.** В данном поле необходимо указать название репозитория латиницей. Не допускается использование пробелов;
* **Public/Private. Публичный (открытый для всех) или закрытый репозиторий. В закрытый репозиторий других пользователей может пригласить лишь сам владелец репозитория**;
* **Initialize this repository with.** Можно сразу добавить в созданный репозиторий:
  + **Добавить readme файл;**
  + **Добавить файл .gitignore –** для указания игнорируемых при работе с репозиторием файлов;
  + **Выбрать лицензию.** Данная лицензия будет говорить другим пользователям что разрешено делать с вашим кодом, а что нет.

После заполнения полей необходимо нажать кнопку «Create repository». Для создания репозитория.

После создания проекта страница автоматически обновится и покажется в следующем виде:

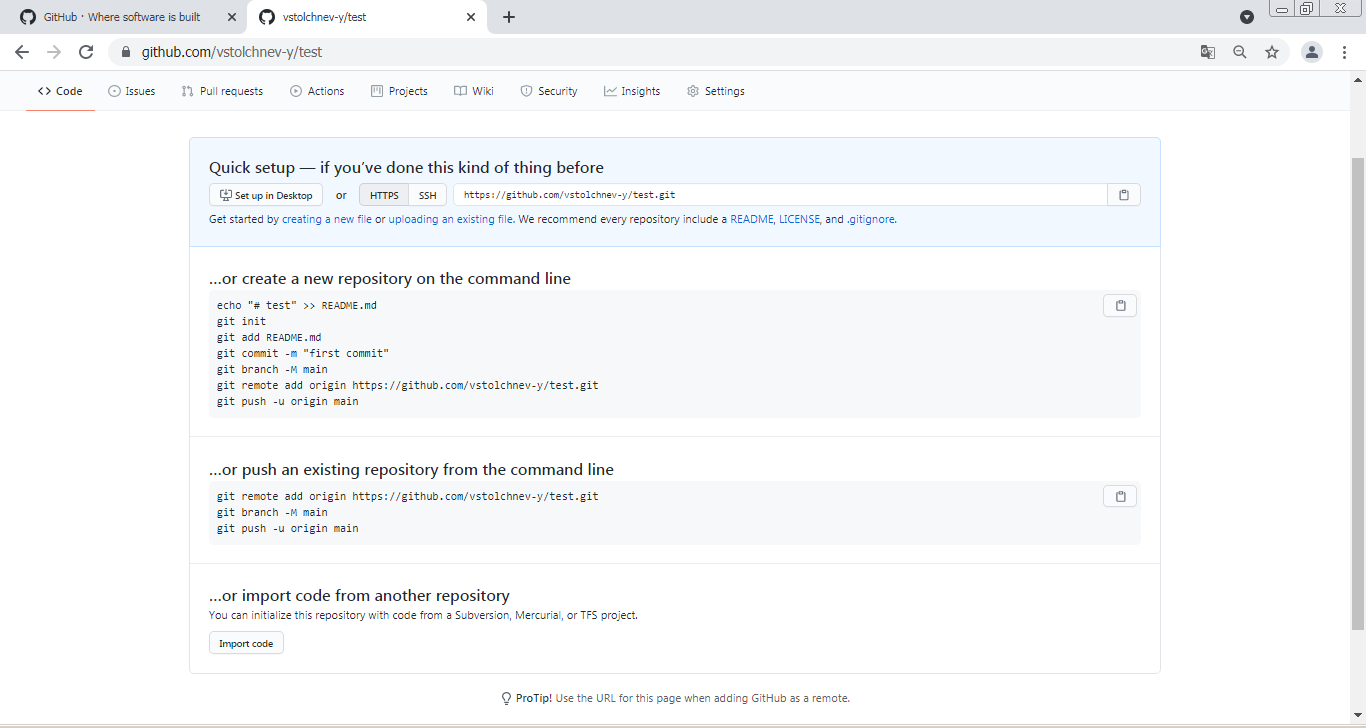


Рисунок 9 Страница нового проекта

На данной странице отображается следующая информация:

* **SSH|HTTP** – строка подключения к удалённому хранилищу через SSH или по HTTP. О подключении будет рассказано ниже;
* **…or create a new repository on the command line** – Команды git, которые необходимо сделать при первой настройке репозитория для работы с GitHub. Данная настройка делается один раз для всех проектов, с которым будет проводиться работа;
* **…or push an existing repository from the command line** – Команды git для подключения существующего локального репозитория. Перед выполнением данных команд необходимо определиться с вариантом подключения к хранилищу и произвести необходимые настройки, если нужно;
* **…or import code from another repository** – Инструкция по импорту репозитория из другого хранилища.

### Глобальные настройки

Перед началом работы с Git необходимо сделать глобальные настройки репозитория.

Сначала необходимо установить имя пользователя и адрес электронной почты. Они потребуются для доступа к глобальному репозиторию.

Выполним данные команды (команды необходимо выполнять последовательно):

git config --global user.name "Stolchnev V.K."

git config --global user.email [stolchnev-vk@fxclub.org](mailto:stolchnev-vk@fxclub.org)

Так как операционные системы могут использовать различные последовательности, обозначающие окончание строки, то для git необходимо указать какие настройки нужно использовать. Универсальным вариантом будет указать автоматическое определение настроек перевода строки:

git config --global core.autocrlf input

git config --global core.safecrlf true

git config –-global core.quotepath true

git config –-global code.longpaths true

### Выбор протокола для работы с удалённым репозиторием

Для данного набора файлом можно получить доступ через различные протоколы данных. Наиболее распространённые:

* SSH — мы имеем доступ к файлам на сервере через ssh;
* HTTP(S) — используем http в качестве приёма/передачи.

Рассмотрим подключение к репозиторию для каждого протокола отдельно:

SSH

SSH позволяет создавать удалённое соединение для обмена файлами, управление удалённым сервером и т.п.

Для работы через SSH необходимо создать ключи шифрования. В дистрибутив Git входит утилита для работы с ключами.

Необходимо запустить её (При стандартной установке: C:\Program Files\Git\Git bash.vbs) и ввести следующие команды:

* ssh-keygen -t rsa -C "адрес электронной почты, который использовался при регистрации в GitHub"
* На запрос ввода имени ключа («Enter file in which to save the key») необходимо нажать Enter;
* Если ключ с именем по умолчанию существует, то выдастся предупреждение;
* Если ключа не существует, то выдастся запрос на ввод пароля: «Enter passphrase». Для лабораторных работ проще работать без пароля, поэтому необходимо нажать «Enter»;
* На запрос повторного ввода пароля («Enter same passphrase again») также необходимо нажать Enter;

После этого покажется информация по созданным ключам. Ниже показан пример создания ключа:

Welcome to Git (version 1.8.5.2-preview20131230)

Run 'git help git' to display the help index.

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

stolchnev-vk@STOLCHNEV-VK /

$ ssh-keygen -t rsa -C "vsolchnev-vk@fxclub.org"

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/stolchnev-vk/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/stolchnev-vk/.ssh/id\_rsa.

Your public key has been saved in /c/Users/stolchnev-vk/.ssh/id\_rsa.pub.

The key fingerprint is:

f4:e3:51:e3:ae:f9:af:de:e3:38:f7:7c:e6:a5:38:5a vsolchnev-vk@fxclub.org

stolchnev-vk@STOLCHNEV-VK /

$

Для выхода из консоли, необходимо ввести команду «Exit».

Внимание. Если явно не указывать ключ, то создаётся имя файла, по умолчанию. Поэтому, небходимо внимательно читать сообщения, которые пишутся в консоли. После перезаписи существующего ключа восстановить его будет невозможно.

После создания ключа необходимо перейти в папку, где они были созданы (она написана в консоли). В данном примере это папка «C:\Users\stolchnev-vk\.ssh\». В ней необходимо открыть файл открытого ключа «id\_rsa.pub» (можно открыть в обычном текстовом редакторе). Всё содержимое данного файла необходимо скопировать с буфер обмена. Эти данные будут использоваться в GitHub для работы через SSH.

После этого необходимо перейти в GitHub и произвести регистрацию публичного ключа.

В GitHub необходимо перейти в настройки профиля (иконка вверху справа страницы) и там выбрать раздел «Settings». На открывшейся странице необходимо выбрать раздел "SSH и GPG keys"

Покажется страница со списком ключей для работы через SSH:

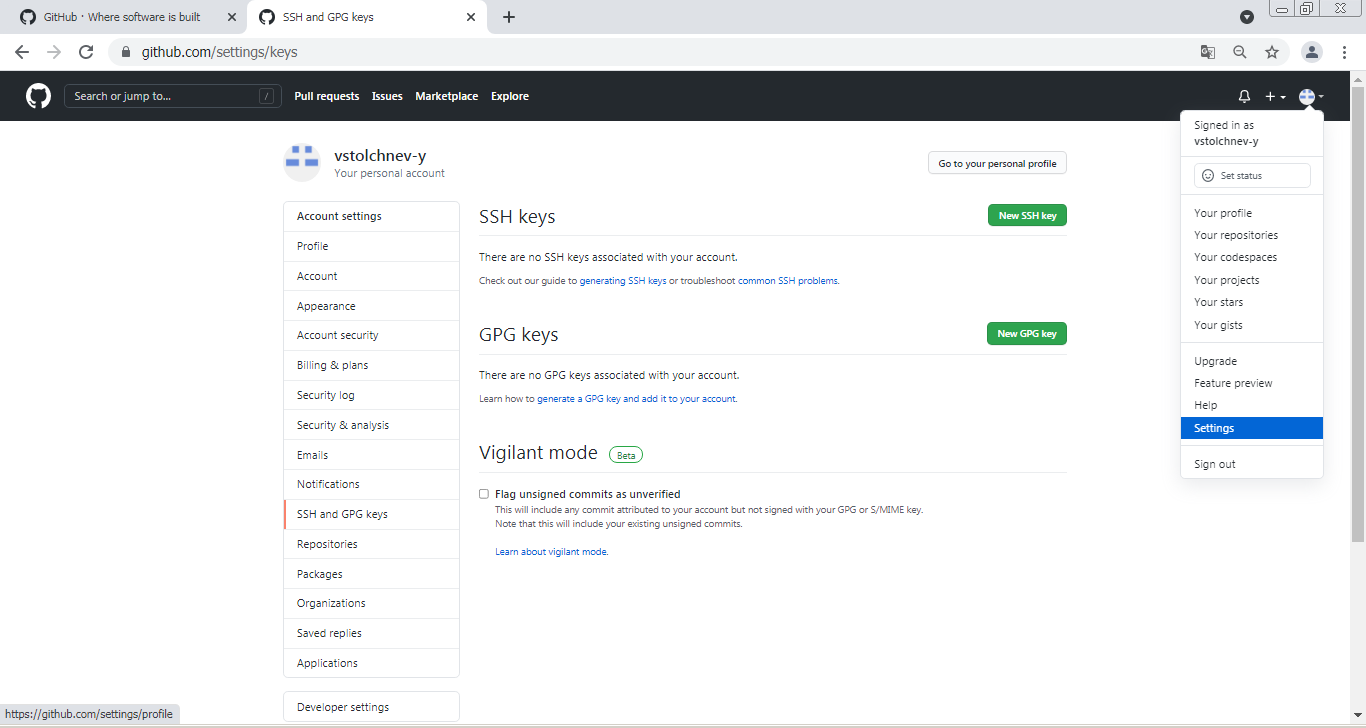


Рисунок 10 Страница со списком ключей SSH

Здесь необходимо добавить новый ключ. Для этого нужно нажать на кнопку «Add SSH Key».

После этого покажется страница добавления нового ключа:

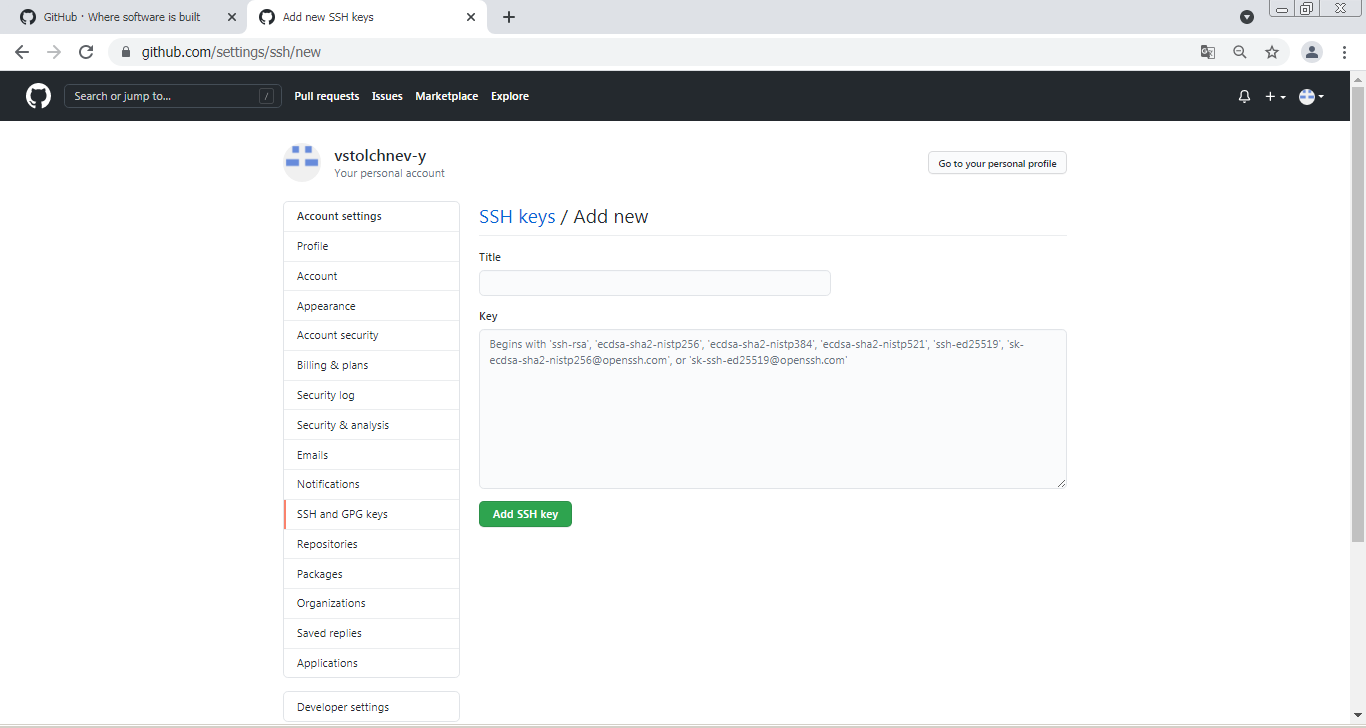


Рисунок 11 Страница создания нового ключа

На данной странице необходимо выполнить следующие действия:

* Ввести название для добавляемого ключа;
* Вставить из буфера обмена содержимое файла «id\_rsa.pub»;
* Нажать на кнопку «Add key».

После этого покажется страница со списком ключей, где будет показан новый ключ.

HTTP(S)

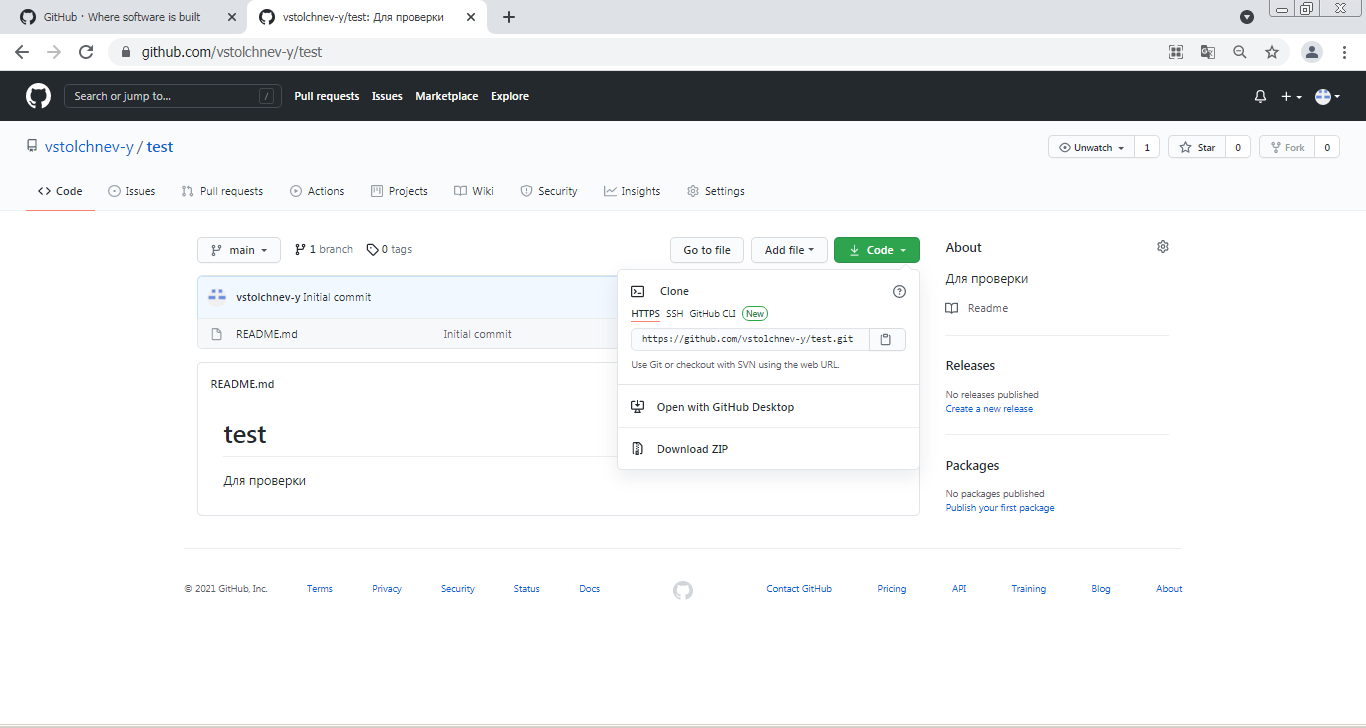
Работа через протокол HTTP (HTTPS) не требует специальных настроек в GitHub. Все необходимые данные указываются в параметрах подключения к репозиторию.

Недостатком является то, что для более быстрой настройки необходимо прописывать логин и пароль к GitHub, что является небезопасным, так как и логин и пароль будут храниться в настройках локального хранилища в открытом виде.

### Подключение к новому проекту

Для подключения к новому проекту необходимо выполнить следующие действия:

* Открыть в GitHub страницу проекта, чтобы увидеть подсказки по подключению;



* Скопировать ссылку на репозиторий (кнопка Code). В раскрывшемся окне выбрать вариант подключения (https, ssh и т.д.)
* Открыть командную строку Windows и перейти в папку с проектом;
* Ввести следующие команды из раздела Create Repository (для подключению по SSH):
  + git init
  + git remote add <строка подключения из GitHub>
  + git add --force <список папок и файлов, разделённых пробелом>
  + git commit –m “<комментарий>”
  + git push -u origin master

Если необходимо сделать подключение по HTTP, то строку подключения, которая показыается в GitHub, необходимо модифицировать следующим образом:

Для показываемого примера:

**https://github.com/vstolchnev-y/test.git**

Если всё сделано правильно, то для проекта создастся локальный репозиторий, подключится к GitHub и передаст туда файлы проекта.

Пример:

E:\Temp\JointDevelopmentExample>git init

Initialized empty Git repository in E:/Temp/JointDevelopmentExample/.git/

E:\Temp\JointDevelopmentExample>git remote add origin git@GitHub.rsreu.ru:stolchnev-vk/examplejointdevelopment.git

git add --force E:\Temp\JointDevelopmentExample\JointDevelopmentExample E:\Temp\JointDevelopmentExample\JointDevelopmentExample.sln E:\Temp\JointDevelopmentExample\JointDevelopmentExample.suo

E:\Temp\JointDevelopmentExample>git commit -m "Создан"

[master (root-commit) 2b3d2e5] Создан

15 files changed, 455 insertions(+)

create mode 100644 JointDevelopmentExample.sln

create mode 100644 JointDevelopmentExample.suo

create mode 100644 JointDevelopmentExample/FormMain.Designer.cs

create mode 100644 JointDevelopmentExample/FormMain.cs

create mode 100644 JointDevelopmentExample/JointDevelopmentExample.csproj

create mode 100644 JointDevelopmentExample/Program.cs

create mode 100644 JointDevelopmentExample/Properties/AssemblyInfo.cs

create mode 100644 JointDevelopmentExample/Properties/Resources.Designer.cs

create mode 100644 JointDevelopmentExample/Properties/Resources.resx

create mode 100644 JointDevelopmentExample/Properties/Settings.Designer.cs

create mode 100644 JointDevelopmentExample/Properties/Settings.settings

create mode 100644 JointDevelopmentExample/bin/Debug/JointDevelopmentExample.vshost.exe

create mode 100644 JointDevelopmentExample/bin/Debug/JointDevelopmentExample.vshost.exe.manifest

create mode 100644 JointDevelopmentExample/obj/x86/Debug/DesignTimeResolveAssemblyReferences.cache

create mode 100644 JointDevelopmentExample/obj/x86/Debug/DesignTimeResolveAssemblyReferencesInput.cache

Warning: Your console font probably doesn't support Unicode. If you experience strange characters in the output, consider switching to a TrueType font such as Lucida Cons

ole!

E:\Temp\JointDevelopmentExample>git push -u origin master

Counting objects: 24, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (21/21), done.

Writing objects: 100% (24/24), 17.57 KiB | 0 bytes/s, done.

Total 24 (delta 0), reused 0 (delta 0)

To git@GitHub.rsreu.ru:stolchnev-vk/examplejointdevelopment.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

Внимание! При работе с Git не нужно добавлять в него откомпилированные файлы (exe, dll), если только это не является необходимым условием для работы проекта.

***Для того, чтобы при добавлении изменений в репозиторий автоматически исключать ненужные файлы в проект добавляют файл .gitignore, где можно прописать необходимые правила по исключению определённых папок и файлов.***

После добавления проекта в удалённый репозиторий можно посмотреть результать в GitHub. Главная страница будет выглядеть следующим образом:

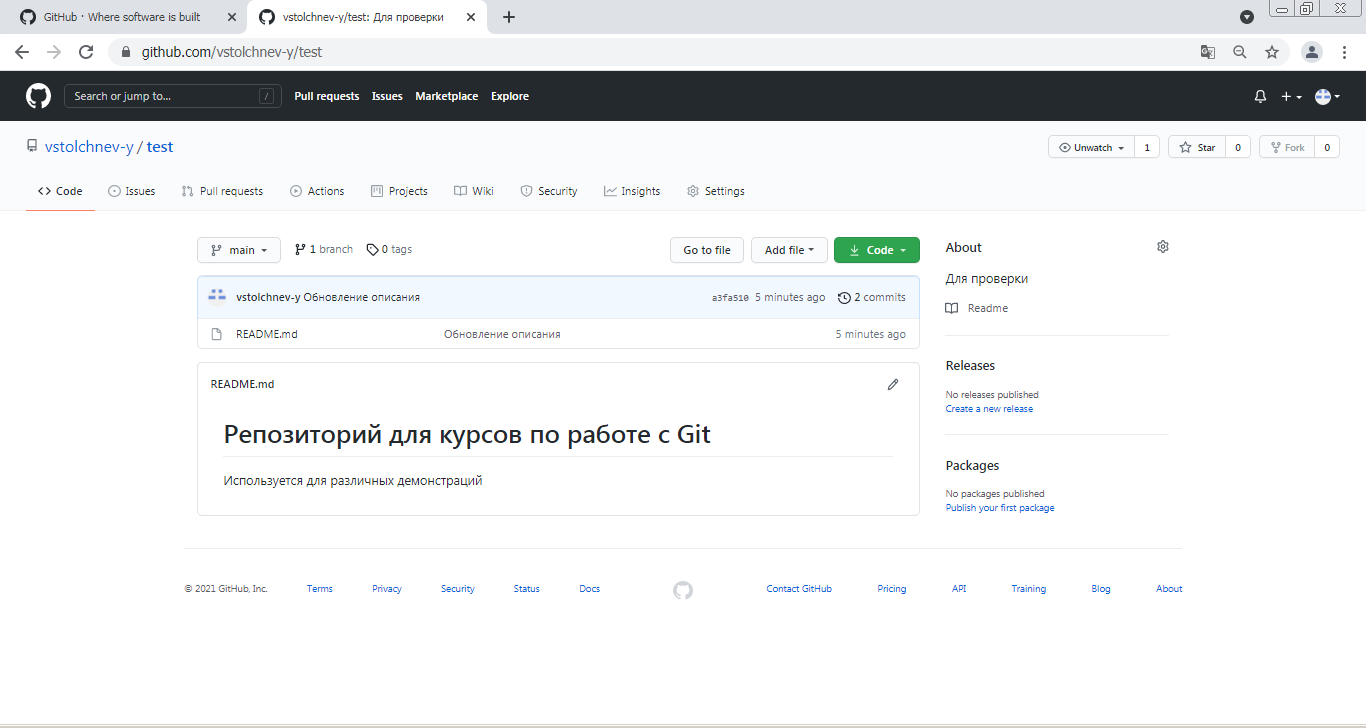


Рисунок 13 Главная страница проекта после добавления проекта

### Подключение Git к существующему проекту

Рассмотрим процесс подключение Git к существующему проекту. Подразумевается, что Git уже настроен для работы.

Строку подключения можно увидеть на странице проекта, к которому необходимо подключить локальный репозиторий:

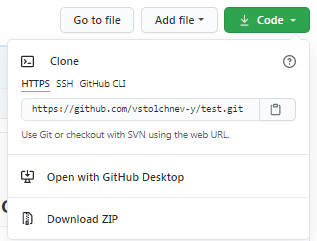


Рисунок 18 Строка подключения существующего проекта

Для подключения к новому проекту необходимо выполнить следующие действия:

* Открыть в GitHub страницу проекта, чтобы увидеть подсказки по подключению;
* Открыть командную строку Windows и перейти/создать папку, где будет производиться работа с подключаемым проектом;
* Ввести следующие команды из раздела Create Repository (для подключению по SSH):
  + git init
  + git remote add <строка подключения из GitHub>
  + git pull origin master

Для примера, который рассмотрен выше, будут выполнены следующие команды:

E:\Temp\GraphicsExample>git init

Initialized empty Git repository in E:/Temp/GraphicsExample/.git/

E:\Temp\GraphicsExample>git remote add origin git@GitHub.rsreu.ru:vstolchnev/graphicsexample.git

E:\Temp\GraphicsExample>git pull origin master

remote: Counting objects: 238, done.

remote: Compressing objects: 100% (87/87), done.

Receiving objects: 100% (238/238), 183.90 KiB | 0 bytes/s, done.

emote: Total 238 (delta 144), reused 238 (delta 144)

Resolving deltas: 100% (144/144), done.

From GitHub.rsreu.ru:vstolchnev/graphicsexample

\* branch master -> FETCH\_HEAD

\* [new branch] master -> origin/master

E:\Temp\GraphicsExample>

После этого, все файлы проекта будут загружены с сервера GitHub в указанную папку.

### Проверка состояния репозитория

Для проверки текущего состояния репозитория служит команда git status

git status

On branch master

nothing to commit, working directory clean

Данный ответ означает, что в репозитории хранится текущее состояние рабочего каталога и нет никаких изменений, ожидающих записи.

git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README

no changes added to commit (use "git add" and/or "git commit -a")

Данный ответ говорит о том, что в есть файлы, которые были изменены/добавлены с момента последнего помещения в репозиторий. Показывается список файлов, которые были изменены или добавлены, а также подсказываются команды, которые необходимо использовать для добавления/прописывания изменений, а также для отмены изменений в локальном репозитории.

### Добавление новых файлов в локальный репозиторий

Добавление новых файлов в локальный репозиторий производится командой git add

Общий формат команды следующий:

**git add [--force] <список файлов и папок, разделённый пробелом>**

Необязательный ключ --force указывает, что должны добавиться все файлы из всех дочерних папок.

После выполнения данной команды файлы пометятся (но ещё не добавятся) в локальный репозиторий.

Если добавить в папку с репозиторием файл readme.txt и посмотреть статус, то будет следующий результат:

git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

readme.txt

nothing added to commit but untracked files present (use "git add" to track)

Здесь указывается, что в папке появился новый файл, который можно добавить в репозиторий командой git add…

Добавим данный файл в репозиторий и посмотрим статус:

git add readme.txt

git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: readme.txt

В результате выполнения команды git status показано, что файл помечен к добавлению в репозиторий. Также подсказывается команда для отмены пометки файла к добавлению: git reset…

Пометка файла к добавлению в репозиторий ещё не помещает туда весь файл. Чтобы изменения в файле добавились в репозиторий необходимо использовать команду git commit, которая описывается в следующем разделе.

### Прописывание изменений в локальный репозиторий

После добавления новых файлов, а также после того, как были сделаны все изменения в существующих файлах, необходимо прописать их в репозиторий.

Это делается командой git commit.

Общий формат команды следующий:

**git commit <необязательный список файлов и папок> -m ”<обязательный комментарий>”**

В результате выполнения данной команды все изменения попадут в локальный репозиторий.

Для прописывания в репозитории файла readme.txt, из предыдущего примера, необходимо воспользоваться командой:

git commit readme.txt -m "Создан"

[master 69b93bf] Создан

1 file changed, 1 insertion(+)

create mode 100644 readme.txt

### Работа с изменениями

Большинство систем версионного контроля работают с файлами. Файл добавляется в репозиторий, а система отслеживает изменения файла с этого момента.

Git фокусируется на изменениях в файле, а не самом файле. Когда выполняется команда git add file, git не добавляет файл в репозиторий. Правильней сказать, что git отмечает текущее состояние файла, коммит которого будет произведен позже.

Добавим в папку с репозиторием файл info.txt со следующим текстом:

Информация:

Пометим файл к добавлению в репозиторий:

git add info.txt

После этого добавим текст в файл:

Информация:

Здесь описывается дополнительная информация

Проверим статус:

git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: info.txt

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: info.txt

Файл info.txt указывается в состоянии дважды.

1. Когда вызывалась команда git add после создания файла. Эти изменения помечены для добавления в репозиторий;
2. Когда в файл добавился текст. Эти изменения ещё не помечены для добавления в репозиторий.

Для проверки, сделаем commit файла в репозиторий и проверим статус:

git commit info.txt -m "Первое добавление файла"

[master 88ef2c1] Первое добавление файла

1 file changed, 3 insertions(+)

create mode 100644 info.txt

git status

On branch master

nothing to commit, working directory clean

Все изменения сохранены в репозиторий.

### Прописывание изменений в удалённый репозиторий

После того, как в локальном репозитории появилась рабочая версия проекта, его можно добавлять в удалённый репозиторий (добавлять нерабочую версию проекта не рекомендуется)

Это делается командой:

**git push –u origin master**

Здесь:

* origin – имя удалённого репозитория, которое даётся по умолчанию (может быть другим, если изменялось при подключении);
* master – имя ветки по умолчанию (ветки создаются в процессе разработки и могут иметь другие названия).

### Получение обновленных файлов из удалённого репозитория

Чтобы получить обновлённую версию файлов из удалённого репозитория необходимо использовать команду «get pull».

Общий формат команды следующий:

**get pull origin master**

Здесь:

* origin – имя удалённого репозитория, которое даётся по умолчанию (может быть другим, если изменялось при подключении);
* master – имя ветки по умолчанию (ветки создаются в процессе разработки и могут иметь другие названия).

### Разрешение конфликтов

При прописывании изменений в удалённый репозиторий возможно возникновение различных конфликтов. Рассмотрим наиболее часто возникающий конфликт, когда изменяются одинаковые части исходного кода.

В этом случае, при попытке прописать изменения в удалённый репозиторий выдаётся ошибка о том, что объединение невозможно, потому что кто-то уже закоммитил изменения и текущая версия локального репозитория не соответствует удалённой.

E:\Temp\GraphicsExample>git push -u origin master

To git@GitHub.rsreu.ru:vstolchnev/graphicsexample.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'git@GitHub.rsreu.ru:vstolchnev/graphicsexample.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

После попытки объединить изменения удалённого репозитория в локальным может возникнуть конфликт. Например, конфликт возник в одном из изменённых файлов:

E:\Temp\GraphicsExample>git pull origin master

remote: Counting objects: 9, done.

remote: Compressing objects: 100% (5/5), done.

remote: Total 5 (delta 3), reused 0 (delta 0)

Unpacking objects: 100% (5/5), done.

From GitHub.rsreu.ru:vstolchnev/graphicsexample

\* branch master -> FETCH\_HEAD

db4f919..a1b7785 master -> origin/master

Auto-merging GraphicsDevelopingExamples/GraphicsDevelopingExample/MainForm.cs

CONFLICT (content): Merge conflict in GraphicsDevelopingExamples/GraphicsDevelopingExample/MainForm.cs

Automatic merge failed; fix conflicts and then commit the result.

В данном примере, присутствуют изменения одной и той же части кода в файле «MainForm.cs».

Для разрешения данного конфликта необходимо воспользоваться утилитой «git mergetool». Рекомендуется настроить её для работы с утилитой сравнения файлов «kdiff3» (как настроить написано в приложении).

Для запуска утилиты сравнения необходимо выполнить команду: «git mergetool --tool=kdiff3».

Если всё настроено правильно, то в результате выполнения команды покажется следующий текст:

E:\Temp\GraphicsExample>git mergetool --tool=kdiff3

Merging:

GraphicsDevelopingExamples/GraphicsDevelopingExample/MainForm.cs

Normal merge conflict for 'GraphicsDevelopingExamples/GraphicsDevelopingExample/MainForm.cs':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (kdiff3):

После нажатия на кнопку «Enter» вызовется утилита сравнения для разрешения конфликта:

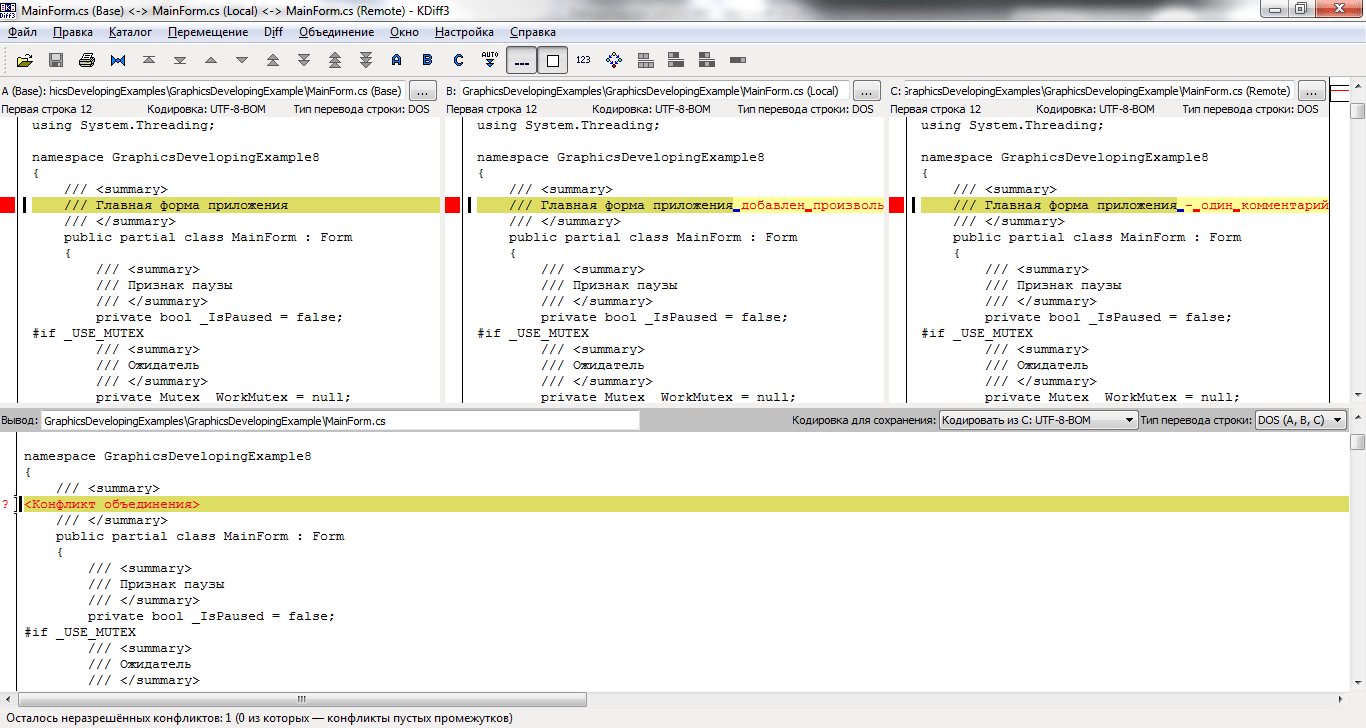


Рисунок 19 Утилита разрешения конфликтов

Сверху показан исходный файл, изменения, которые произвёл другой пользователь и изменения, которые сохранились в локальном репозитории.

Внизу показывается файл, в котором необходимо разрешить конфликт. Для конфликта вызывается контекстное меню, где можно выбрать какой из вариантов необходимо оставить.

После внесения всех изменений файл сохраняется.

Когда разрешены все конфликты, необходимо заново вызвать команду «git commit» и прописать изменения в удалённый репозиторий.

## Задание

Необходимо сделать следующее:

1. Зарегистрироваться в GitHub (если нет регистрации)
2. Подключиться к тестовому проекту (клонировать репозиторий)
   1. https: <https://github.com/vstolchnev/test.git>
   2. ssh: [git@github.com:vstolchnev/test.git](mailto:git@github.com:vstolchnev/test.git)
3. Переключиться на ветку Materials с дополнительными материалами. После чего переключиться обратно на основную ветку;
4. Получить тему на проектирование класса у преподавателя
5. Создать ветку из master в локальном репозитории
6. Спроектировать класс на заданную тему (использовать уникальное пространство имён – проговорить с преподавателем)
   1. В классе должно быть 2-3 поля
   2. В классе должен быть метод для получения информации об объекте класса в виде строки (описание класса и значения полей)
7. Залить ветку в Git
8. Сделать pull request на добавление своей ветки в Master
9. Далее необходимо доработать собственный класс следующим образом:
   1. Переключиться обратно в свою ветку;
   2. Подключить к своему класс класс, спроектированный другим студентов (например, следующим по списку). Для этого необходимо залить в свою ветку ветку, где делал свой класс другой студент;
   3. Написать метод, который создаёт массив объектов класса, сделанного другим студентом, и возвращает его;
   4. Залить свою ветку в удалённый репозиторий;
   5. Сделать пул-реквест по добавлению своих изменений в основную ветку.

# Приложения

## Общие термины и определения

Общепринятой терминологии не существует, в разных системах могут использоваться различные названия для одних и тех же действий. Ниже приводятся некоторые из наиболее часто используемых вариантов. Приведены английские термины, в литературе на русском языке используется тот или иной перевод или транслитерация.

### **branch**

* **Ветвь** — направление разработки, независимое от других. Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные — после неё.

### **changeset, changelist, activity**

* **Набор изменений**. Представляет собой поименованный набор правок, сделанных в локальной копии для какой-то общей цели. В системах, поддерживающих наборы правок, разработчик может объединять локальные правки в группы и выполнять фиксацию логически связанных изменений одной командой, указывая требуемый набор правок в качестве параметра. При этом прочие правки останутся незафиксированными. Типичный пример: ведётся работа над добавлением новой функциональности, а в этот момент обнаруживается критическая ошибка, которую необходимо немедленно исправить. Разработчик создаёт набор изменений для уже сделанной работы и новый — для исправлений. По завершении исправления ошибки отдаётся команда фиксации только второго набора правок.

### check-in, commit, submit

* **Создание новой версии, фиксация изменений**. Распространение изменений, сделанных в рабочей копии, на хранилище документов. При этом в хранилище создаётся новая версия изменённых документов.

### check-out, clone

* **Извлечение документа из хранилища и создание рабочей копии.**

### conflict

* **Конфликт** — ситуация, когда несколько пользователей сделали изменения одного и того же участка документа. Конфликт обнаруживается, когда один пользователь зафиксировал свои изменения, а второй пытается зафиксировать и система сама не может корректно слить конфликтующие изменения. Поскольку программа может быть недостаточно разумна для того, чтобы определить, какое изменение является «корректным», второму пользователю нужно самому разрешить конфликт (resolve).

### head

* **Основная версия** — самая свежая версия для ветви/ствола, находящаяся в хранилище. Сколько ветвей, столько основных версий.

### merge, integration

* **Слияние** — объединение независимых изменений в единую версию документа. Осуществляется, когда два человека изменили один и тот же файл или при переносе изменений из одной ветки в другую.

### rebase

* **Перенос точки ветвления (версии, от которой начинается ветвь) на более позднюю версию основной ветви**. Например, после выпуска версии 1.0 проекта в стволе продолжается доработка (исправление ошибок, доработка имеющегося функционала), одновременно начинается работа над новой функциональностью в новой ветви. Через какое-то время в основной ветви происходит выпуск версии 1.1 (с исправлениями); теперь желательно, чтобы ветвь разработки новой функциональности включала изменения, произошедшие в стволе. Вообще, это можно сделать базовыми средствами, с помощью слияния (merge), выделив набор изменений между версиями 1.0 и 1.1 и слив его в ветвь. Но при наличии в системе поддержки перебазирования ветви эта операция делается проще, одной командой: по команде rebase (с параметрами: ветвью и новой базовой версией) система самостоятельно определяет нужные наборы изменений и производит их слияние, после чего для ветви базовой версией становится версия 1.1; при последующем слиянии ветви со стволом система не рассматривает повторно изменения, внесённые между версиями 1.0 и 1.1, так как ветвь логически считается выделенной после версии 1.1.

### repository, depot

* **Хранилище документов** — место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.

### revision

* **Версия документа**. Системы управления версиями различают версии по номерам, которые назначаются автоматически.

### shelving

* **Откладывание изменений**. Предоставляемая некоторыми системами возможность создать набор изменений (changeset) и сохранить его на сервере без фиксации (commit’а). Отложенный набор изменений доступен на чтение другим участникам проекта, но до специальной команды не входит в основную ветвь. Поддержка откладывания изменений даёт возможность пользователям сохранять незавершённые работы на сервере, не создавая для этого отдельных ветвей.

### tag, label

* **Метка, которую можно присвоить определённой версии документа.** Метка представляет собой символическое имя для группы документов, причем метка описывает не только набор имен файлов, но и версию каждого файла. Версии включенных в метку документов могут принадлежать разным моментам времени.

### trunk, mainline, master

* **Ствол — основная ветвь разработки проекта.** Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол; если требуется серьёзное изменение, способное привести к нестабильности, создаётся ветвь, которая сливается со стволом, когда нововведение будет в достаточной мере испытано; перед выпуском очередной версии создаётся «релизная» ветвь, в которую вносятся только исправления.

### update, sync

* **Синхронизация рабочей копии до некоторого заданного состояния хранилища.** Чаще всего это действие означает обновление рабочей копии до самого свежего состояния хранилища. Однако при необходимости можно синхронизировать рабочую копию и к более старому состоянию, чем текущее.

### working copy

* **Рабочая (локальная) копия документов**.

## Термины и определения, связанные с Git

* **index** — область зафиксированных изменений, т.е. всё то, что вы подготовили к сохранению в репозиторий;
* **commit** — изменения, отправленные в репозиторий;
* **HEAD** — указатель на commit, в котором мы находимся;
* **master** — имя ветки по-умолчанию, это тоже указатель на определённый коммит;
* **origin** — имя удалённого репозитория по умолчанию (можно дать другое);
* **checkout** — взять из репозитория какое-либо его состояние.

## Полезные команды Git

Если вы сделали что-то не так, запутались, не знаете, что происходит — эти две команды вам помогут:

* **git status** — показывает состояние вашего репозитория (рабочей копии) и где вы находитесь.
* **gitk** — графическая утилита, которая показывает наш граф. В качестве ключей передаём имена веток или --all, чтобы показать все.

## Настройка Git для работы с KDiff3

Утилита kdiff3 удобна для работы с разрешением конфликтов в файлах.

Скачать её можно по следующей ссылке: <http://sourceforge.net/projects/kdiff3/files/>

Чтобы настрить Git для работы с данной утилитой, необходимо произвести следующие действия:

* Открыть настроечный файл Git (По умолчанию находится в папке "C:\Program Files\Git\etc\gitconfig");
* Добавить в него следующие строки:

[merge]

tool = kdiff3

[mergetool "kdiff3"]

path = C:/Program Files/KDiff3/kdiff3.exe

keepBackup = false

trustExitCode = false

Внимание! Путь к утилите kdiff3.exe указывается через прямой слеш «/». Если вставить путь, который предлагает командная строка Windows, то при запуске Git будет выдаваться сообщение об ошибке.

Если всё сделано правильно, то в результате выполнения команды «git mergetool --tool=kdiff3» будет вызываться нужный редактор.