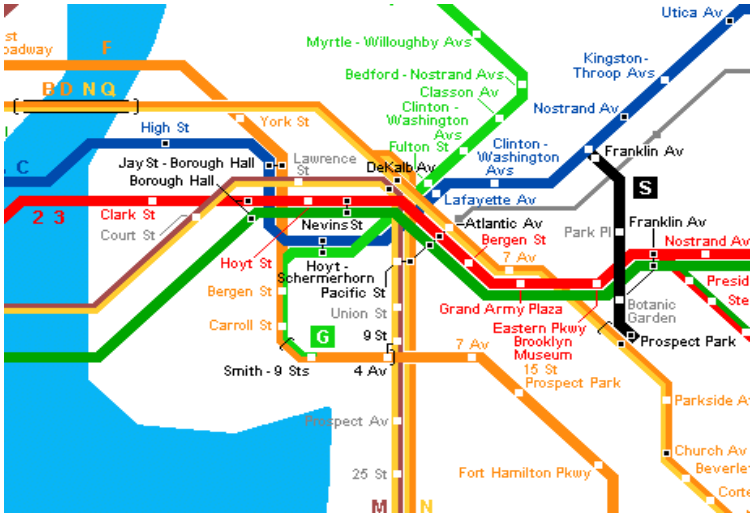
МОДУЛЬ 1.8 УЧЕБНОЙ ПРОГРАММЫ
"ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ"

ДИСЦИПЛИНА
«КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

ЛАБОРАТОРНАЯ РАБОТА №1

СИСТЕМЫ КОНТОРОЛЯ ВЕРСИЙ



ПОИСиТ

Минск, БНТУ, ФИТР, кафедра ПОИСиТ 2023 Автор: Сергей Станкевич

Лабораторная работа №1а

СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

GIT GUI

Цель работы

Ознакомиться с системами контроля версий для разработки программного обеспечения. Освоить способы организации работы в системах контроля версий. Получить навыки командной работы.

Норма времени выполнения: *3 академических часа.*
Оценка работы: *3 зачётных единицы.*

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ

Система управления версиями

Система управления версиями – это программное обеспечение для облегчения работы с изменяющейся информацией.

Также используется определение «система контроля версий», от англ. *Version Control System* (VCS) или *Revision Control System*.

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 кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM).

Контроль версий, также известный как *управление исходным кодом* (*Software Configuration Management*, SCM), это практика отслеживания изменений программного кода и управления ими. Системы контроля версий, это программные инструменты, помогающие командам разработчиков управлять

изменениями в исходном коде с течением времени. В свете усложнения сред разработки они помогают командам работать быстрее и эффективнее. Системы контроля версий наиболее полезны командам DevOps, поскольку помогают сократить время разработки и увеличить количество успешных развертываний.

Группы разработчиков программного обеспечения, *не использующие* какую-либо форму управления версиями, часто сталкиваются с такими *проблемами*, как незнание об изменениях, выполненных для пользователей, или создание в двух несвязанных частях работы изменений, которые оказываются несовместимыми и которые затем приходится скрупулезно распутывать и перерабатывать. Если вы как разработчик ранее никогда не применяли управление версиями, возможно, вы указывали версии своих файлов, добавляя суффиксы типа «финальный» или «последний», а позже появлялась новая окончательная версия. Возникает большо риск запутаться в окончательных версиях. Также на практике многие программисты комментируют блоки кода, чтобы отключить определенные возможности, при этом не удаляя их, предполагая, возможно этот код понадобиться позже. Тем самым загромождается исходный текст программы, усложняется его чтение и понимание. Решением всех подобных проблем является использование системы управление версиями.

Подведя итоги вышесказанному, можно сделать вывод. Программное обеспечение управления версиями является неотъемлемой частью повседневной профессиональной практики современной команды разработчиков ПО. Разработчики, постоянно работающие в команде с эффективной системой управления версиями, обычно признают невероятную пользу этих систем даже при работе над небольшими сольными проектами. Привыкнув к мощным преимуществам систем контроля версий, многие разработчики не представляют, как работать без них даже в проектах, не связанных с разработкой ПО.

Типы VCS

По своей архитектуре и функционалу системы контроля версий подразделяются на следующие типы:

- Локальные
- Централизованные
- Распределённые

Локальные системы хранят на локальном компьютере простые базы данных, о записи всех изменениях в файлах, осуществляя тем самым контроль ревизий. Примером является система GNU RCS (Revision Control System). GNU RCS хранит на диске наборы *патчей* (различий между файлами) в специальном

формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные *системы контроля версий* (Centralized Version Control System, CVCS) созданы для решения серьёзной проблемы – необходимость взаимодействовать с другими разработчиками. Такие системы используют *единственный сервер*, содержащий все версии файлов, и некоторое количество *клиентов*, которые получают файлы из этого *централизованного хранилища*. Примерами являются системы: CVS, Subversion и Perforce. На протяжении многих лет CVCS являлось стандартом.

Однако данный подход тоже имеет *серьёзные минусы*. Самый очевидный минус – это *единая точка отказа*, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, можно потерять всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные VCS страдают от той же самой проблемы.

Для решения этой проблемы используются ***распределённые*** *системы контроля версий* (Distributed Version Control System, DVCS). В DVCS клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) – они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, «умрёт», любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных. Более того, многие DVCS могут одновременно взаимодействовать с несколькими *удалёнными репозиториями*, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Примерами распределенных систем являются: Git, Mercurial, Bazaar или Darcs.

На Рисунок 1 представлены архитектурные типы систем контроля версий.

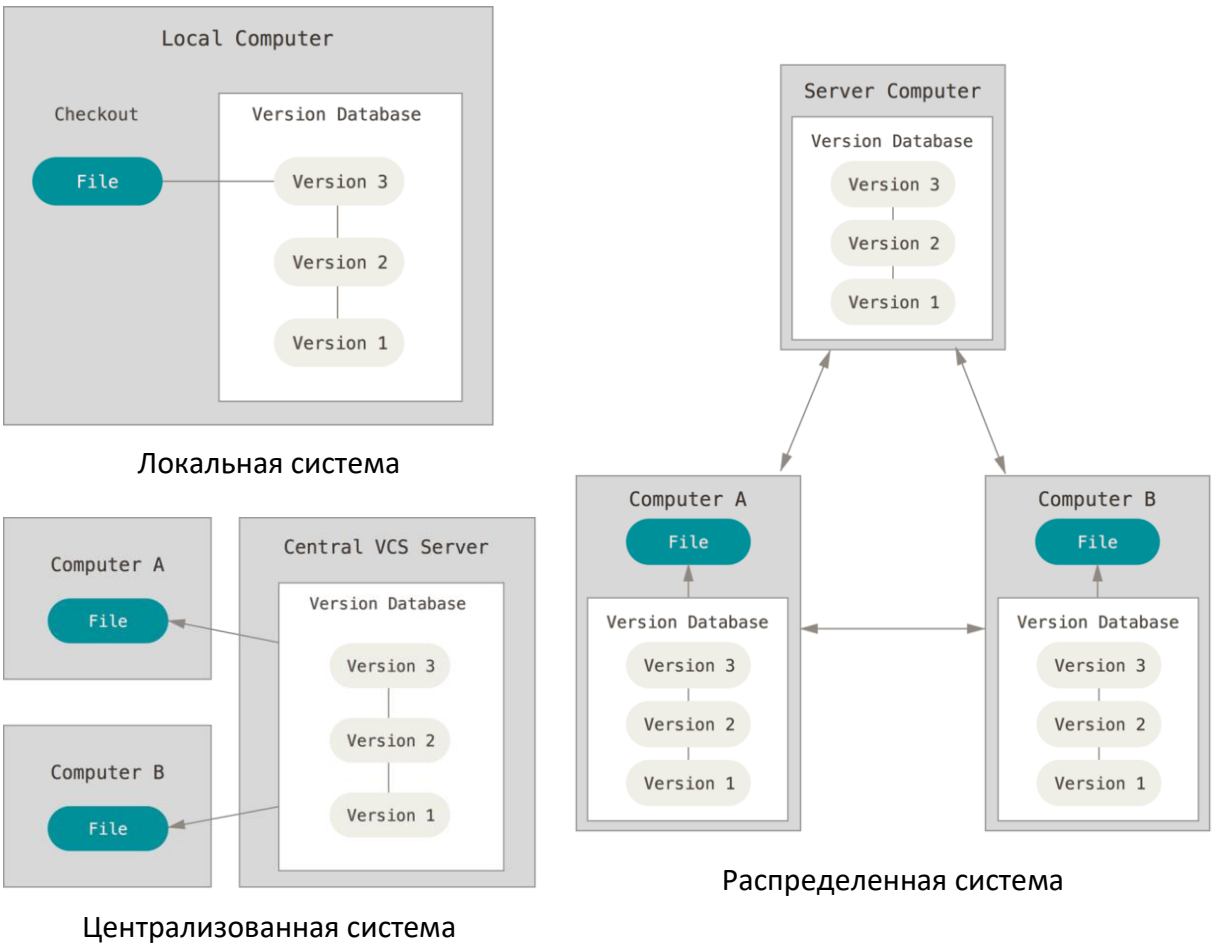


Рисунок 1 – Схемы архитектур типов систем контроля версий

Git

Git, это наиболее популярная систем управления версиями из множества других. Это развитый проект с активной поддержкой и *открытым исходным кодом*. Система Git была изначально разработана в 2005 году Линусом Торвальдсом, создателем ядра Linux для операционных систем. Поэтому для Git операционные системы на ядре Linux являются родной средой, не требующей никакой эмуляции. Git превосходно работает под управлением различных операционных систем и может применяться со множеством интегрированных сред разработки (IDE), например, Visual Studio.

Есть много программных средств использования Git. Помимо оригинального клиента, имеющего интерфейс командной строки, существует множество клиентов с графическим пользовательским интерфейсом, в той или иной степени реализующих функциональность Git.

Таким образом программные средства для работы с Git делятся на два типа:

- Интерфейс командной строки
- Графический интерфейс (GUI)

При установке Git на локальный компьютер мы получаем программы обеих типов.

Командная строка – это единственное место, где можно запустить все команды Git, так как большинство клиентов с графическим интерфейсом реализуют для простоты только некоторую часть функциональности Git. Если вы знаете, как выполнить какое-либо действие в командной строке, то вероятно, сможете выяснить, как то же самое сделать и в GUI-версии, а вот обратное не всегда верно.

При установке Git мы получаем интерфейс командной строки, но это подробно мы изучим в следующей лабораторной работе.

GUI-клиенты помогают существенно ускорить вашу работу с системой контроля версий, особенно, если вы еще не слишком хорошо с ней знакомы. Получить перечень наиболее популярных графических средств можно сделав запрос в браузере.

В базовой комплектации Git поставляется приложение Git Gui, которое мы с вами рассмотрим в этой лабораторной работе.

Базовый подход в работе с Git – Три состояния

Это ***самая важная вещь для понимания Git***, которую нужно запомнить о Git, если вы хотите, чтобы остаток процесса обучения прошёл гладко. У Git есть *три основных состояния*, в которых могут находиться ваши файлы: *изменён* (modified), *индексирован* (staged) и *зафиксирован* (committed):

1. К *изменённым* относятся файлы, которые поменялись, но ещё не были зафиксированы.
2. *Индексированный* – это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.
3. *Зафиксированный* значит, что файл уже сохранён в вашей локальной базе.

Каждому основному состоянию соответствует *секция* проекта Git: *рабочая копия* (working tree), *область индексирования* (staging area) и *каталог Git* (Git directory).

На Рисунок 2 представлена схема состояния файлов и секции проекта Git.

Рабочая копия или *директория* является снимком одной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать.

Область индексирования – это файл, обычно находящийся в каталоге Git, в нём содержится информация о том, что попадёт в следующий коммит. Её техническое название на языке Git – «индекс», но фраза «область индексирования» также работает.

Каталог Git – это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git и это та часть, которая копируется при клонировании репозитория с другого компьютера.

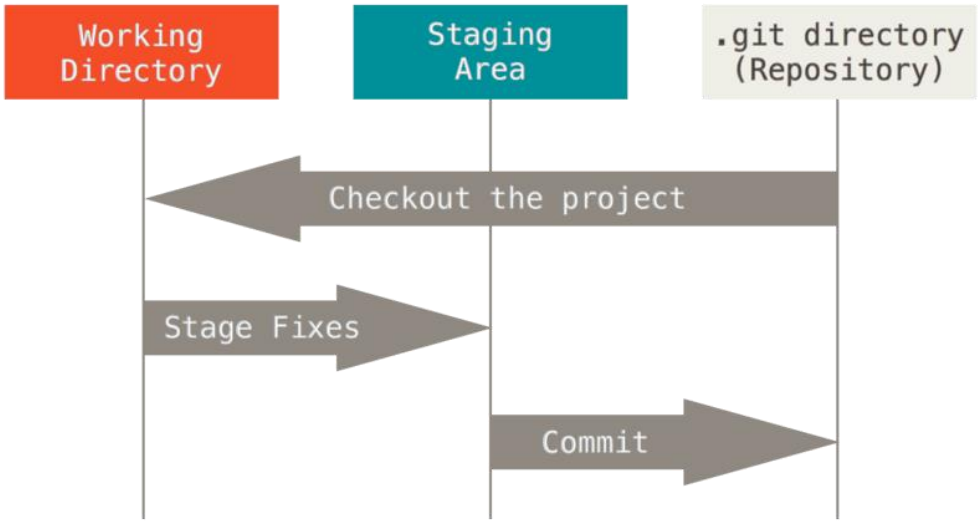


Рисунок 2 - Схема состояния файлов и секции проекта Git.

Базовый подход в работе с Git выглядит так:

1. Изменяете файлы вашей рабочей директории.
2. Выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в индекс.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в ваш каталог Git.
4. *(взять советы из видео)*

В практическом использовании Git часть с индексом можно пропустить.

Чтобы понять принцип работы программки Git Gui, внимательно изучите рисунок X.

Установка и первоначальная настройка Git

Установка **Git** на локальный компьютер зависит от типа операционной системы, которая установлена на нем. В настоящее время любой программист должен считаться с тремя монстрами операционных систем, это *Windows*, *Mac OS* и *Linux* (но здесь правильно говорить: «ОС на Linux-е», так как Linux не является операционной системой). Также необходимо помнить, что Linux для Git это естественная среда обитания.

Чтобы установить Git перейдите на официальный сайт Git, и посмотрите инструкцию по установке для нужной вам разновидности операционной системы. Это можно сделать по ссылкам:

https://git-scm.com/download/linux

https://git-scm.com/download/mac.

https://git-scm.com/download/win

Даже если Git уже установлен, наверное, это хороший повод, чтобы обновиться до последней версии. Вы можете установить Git из *собранного пакета* или другого установщика, либо *скачать исходный код* и *скомпилировать* его самостоятельно.

Когда все необходимые зависимости установлены, вы можете пойти дальше и скачать самый свежий архив с исходниками из следующих мест: с сайта **Kernel.org** https://www.kernel.org/pub/software/scm/git, или зеркала на сайте **GitHub** https://github.com/git/git/releases. Конечно, немного проще скачать последнюю версию с сайта GitHub, но на странице kernel.org релизы имеют подписи, если вы хотите проверить, что скачиваете.

При первоначальной установке Git нужно настроить среду для работы с Git под себя. Однако несведущему человеку в этом сложно разобраться, поэтому можно довериться установщику. При обновлении версии Git ваши настройки сохраняются. Но, при необходимости, когда разберетесь глубже, вы можете поменять их в любой момент, выполнив определенные команды.

Единственной что важно при первой установке это настроить «*Имя пользователя*», указав свое *имя* или *логин* и *адрес вашей электронной почты*. Иначе коммиты не будут работать, Git и другие разработчики должны знать кто делает изменения в проекте.

Так же при настройках можно обратить внимание на следующие параметры:

- Выбор редактора (для тех, кто привык работать в Windows, рекомендуется «Notepad»).
- Настройка ветки по умолчанию (предустановлено «master»).

Остальные настройки будем менять по мере углубления в понимание системы.

Пространство имен «Имя пользователя»

Современные операционные системы являются многопользовательскими, каждый пользователь внутри системы может иметь различные проекты. Доступ к этим проектам должен быть разделен и ограничен.

Имя пользователя Git может быть зарегистрировано на трех уровнях

Уровни конфигурации проектов в Git

Каждая компьютерная программа для общения с операционной системой имеет специальные данные, как правило в виде переменных, которые называются *конфигурацией* программного обеспечения. Конфигурация, это совокупность настроек программы, задаваемая пользователем, а также процесс изменения этих настроек в соответствии с нуждами пользователя. Проекты Git также имеют конфигурацию, данные которой хранятся в специальных текстовых файла. Изменяя параметры в этих конфигурационных файлах, можно менять поведение Git.

Конфигурационные параметры Git разделяются на две категории: настройки *клиента* и настройки *сервера*. Большая часть – клиентские, для настройки ваших личных предпочтений в работе.

Параметры конфигурации хранятся на трех уровнях системы:

- системный
- глобальный
- локальный

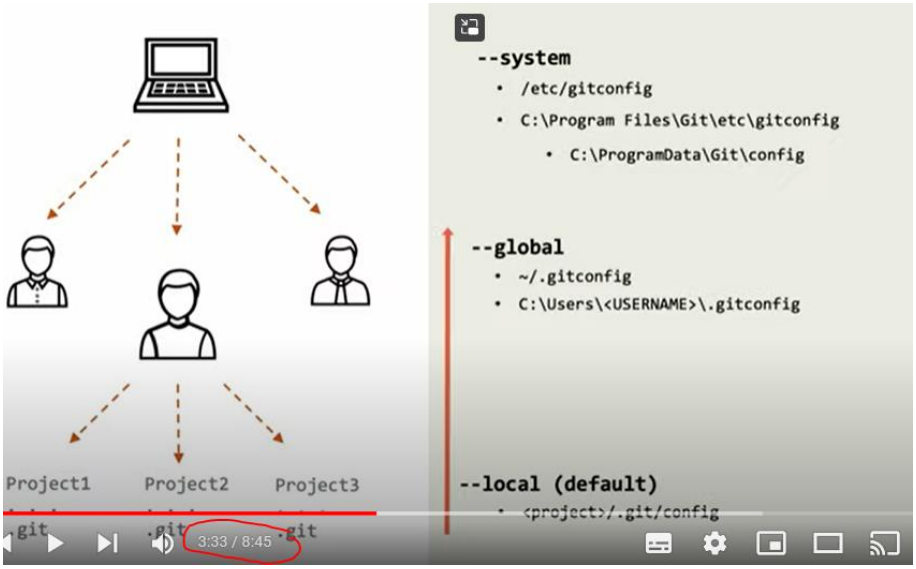
Настройки конкретного репозитория хранятся на *локальном* уровне в файле <project>/.git/config репозитория, и устанавливаются по умолчанию. Имеет опцию --local. Для установки локальных параметров пользователь должен находиться в конкретном репозитории Git

Настройки конкретного пользователя хранятся на глобальном уровне в домашней директории пользователя (**home** или ~) в файле ~/.gitconfig или ~/.config/git/config, и относятся ко всем репозитория с которыми пользователь работает в текущей системе.

В диалоге опций (Edit > Options…)

2.1 Git – Основы – Конфигурация - YouTube

https://git-scm.com/book/ru/v2/Введение-Первоначальная-настройка-Git



В состав Git входит утилита **git config**, которая позволяет просматривать и настраивать конфигурационные параметры, контролирующие все аспекты работы Git, а также его внешний вид.

Чтобы посмотреть все установленные настройки и узнать, где именно они заданы, используйте команду:

```
$ git config --list --show-origin
```

Стратегия разработки кода

Рекомендации по управлению исходным кодом

- Делайте коммиты чаще.
- Удостоверьтесь, что вы работаете с последней версией (Перед внесением обновлений обязательно выполняйте команду git pull или fetch, чтобы получить актуальный код).
- Оставляйте подробные комментарии.
- Просматривайте изменения перед выполнением коммита.

- Используйте ветки.
- Согласуйте рабочий процесс.

Заключение

Вы получили базовые знания о том, что такое Git. Получили рабочую версию Git в вашей ОС, настроенную и персонализированную. После этого самое время изучить основы Git.

Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Sergey Stankevich

УПРАЖНЕНИЯ

Упражнение №1. Установка Git, работа с Git GUI

1. Установка Git

Для установки необходимо скачать msysgit (инсталляционный пакет для Windows http://git-scm.com/download/win) и запустить его. Получить ссылку на скачивание можно и из других источников.

Все настройки в инсталляторе необходимо оставить по умолчанию, кроме представленной ниже (см. Рисунок 3).

Предупреждение! Не выбирайте третью, самую нижнюю опцию.

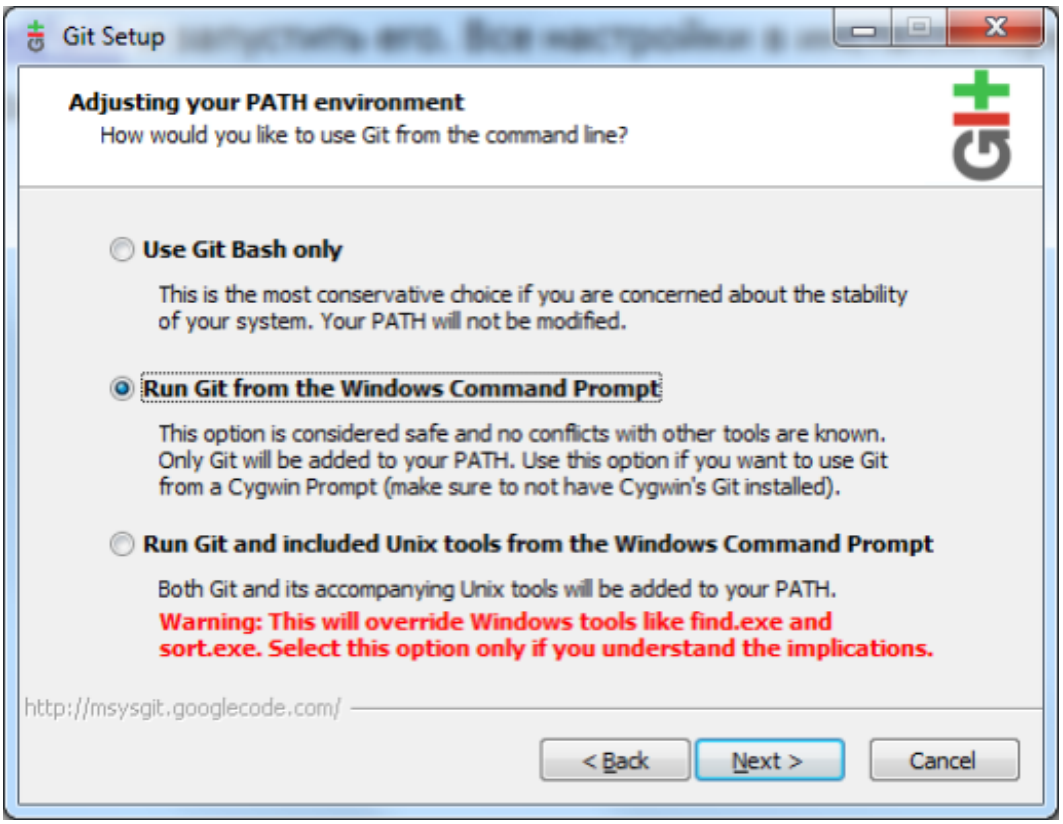


Рисунок 3 - Настройка инсталлятора Git

Проходим шаги установочной программы. При правом щелчке мыши на папке можно отметить опцию интеграции с Windows Explorer (см. *Рисунок 4*).

Для продолжения необходимо нажимать Next пока установка не закончится.

После установки Git подробно рассмотрим работу с Git GUI.

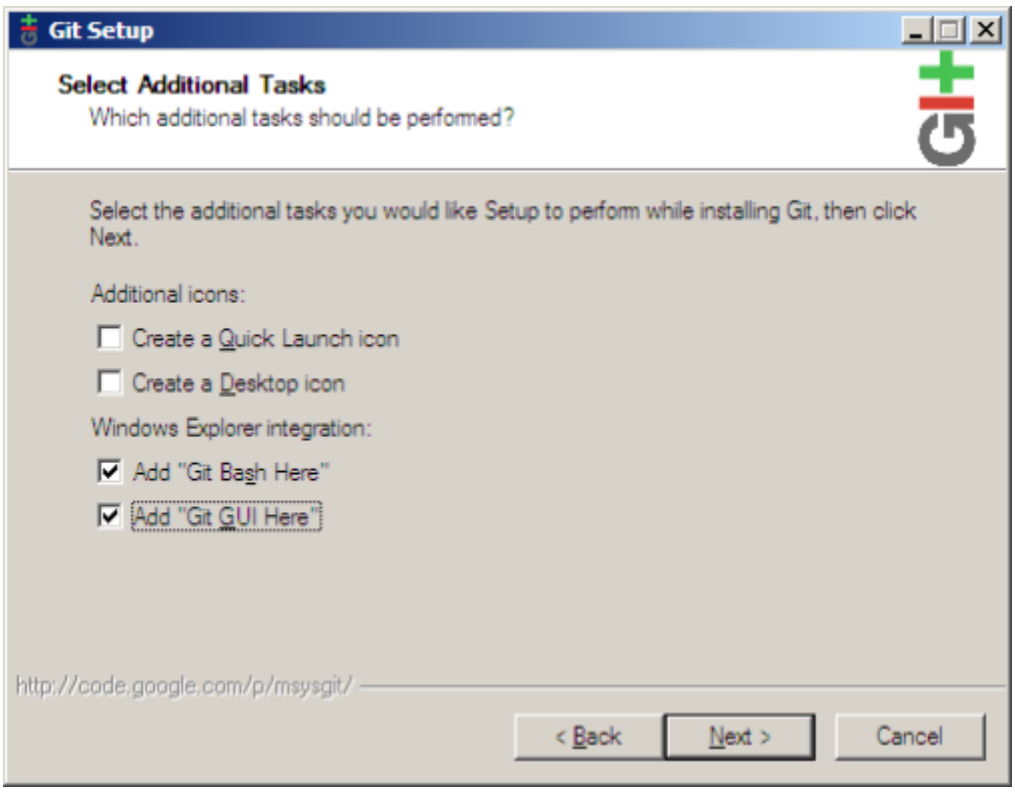


Рисунок 4 - Отметка опции интеграции с Windows Explorer

2. Создание хранилища

Чтобы создать *репозиторий* (хранилище), необходимо создать папку, в которой ваш проект будет храниться. Экспериментировать лучше всего в папке-«песочнице» («*sandbox*»). Папка-песочница пока пуста, это можно увидеть в проводнике. После создания «песочницы», нужно нажать правой кнопкой мыши по этой папке и выбирать Git GUI (см. Рисунок 5 - Выбор Git GUI*Рисунок* 5). Так как в папке пока не содержится git-хранилище, будет показан диалог создания (см. *Рисунок 6*).

Выбор *Create New Repository* приводит к следующему диалогу (см. *Рисунок 7*). Здесь необходимо заполнить путь к вашей новой директории и щёлкнуть *Create*. Далее вы увидите главный интерфейс Git GUI (см. *Рисунок 8*), который в дальнейшем будет показываться, когда вы будете нажимать правой кнопкой мыши по вашей папке и выбирать Git GUI.

Хранилище уже создано, но в проводнике вы этого не увидите, оно скрыто от посторонних глаз. На панели задач проводнике выберите *Вид* > *Показать* > *Скрытые элементы*, и вы увидите свое хранилище с названием «.git». Объясните, что обозначает точка в начале имени файла.

Когда хранилище создано, необходимо сообщить Git информацию о пользователе для того, чтобы в сообщении *фиксации* (commit message) был отмечен

правильный автор. Чтобы сделать это, необходимо выбрать *Edit → Options* (*Редактировать → Настройки*).

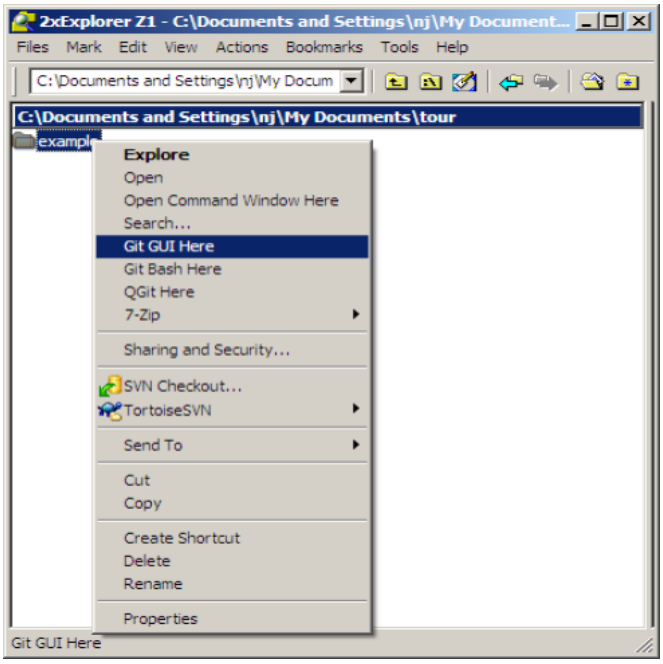


Рисунок 5 - Выбор Git GUI

Рисунок 6 - Диалог создания репозитория

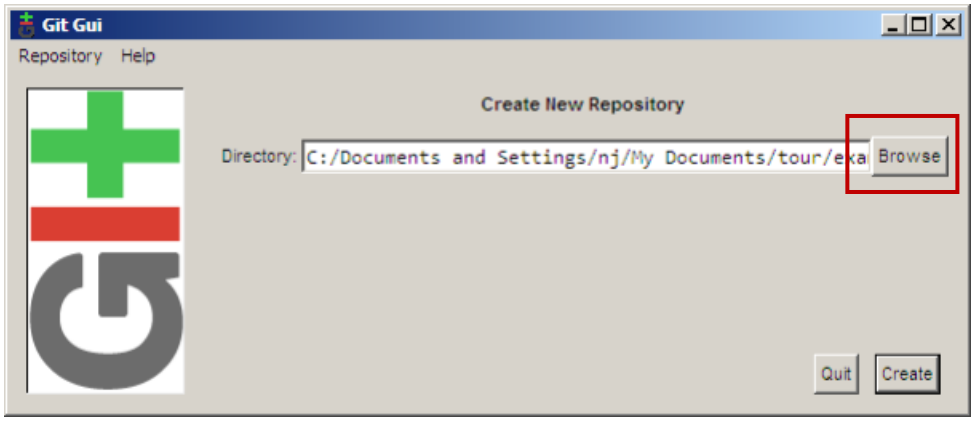


Рисунок 7 - Заполнение пути к новой директории

В диалоге опций (Edit > Options...) расположены 2 варианта на выбор (см. Рисунок 9). С левой стороны диалога опции, которые влияют только на это хранилище, с правой стороны содержатся глобальные опции, применяемые ко *всем хранилищам*. Значения по умолчанию приемлемы, поэтому пока нужно заполнить только *имя пользователя* и *email*. Это достаточно важно, так как при совместной разработке необходимо знать кто и когда вносил изменения в проект. Значение этих полей каждый раз будет отображаться в новых изменениях. Также сейчас можно выставить шрифт.

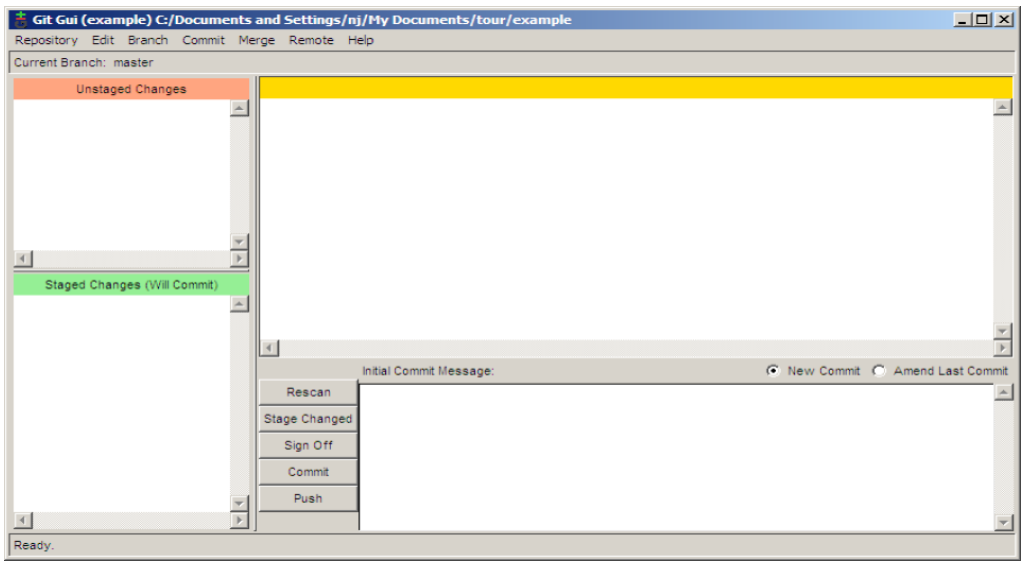


Рисунок 8 - Главный интерфейс Git Gui

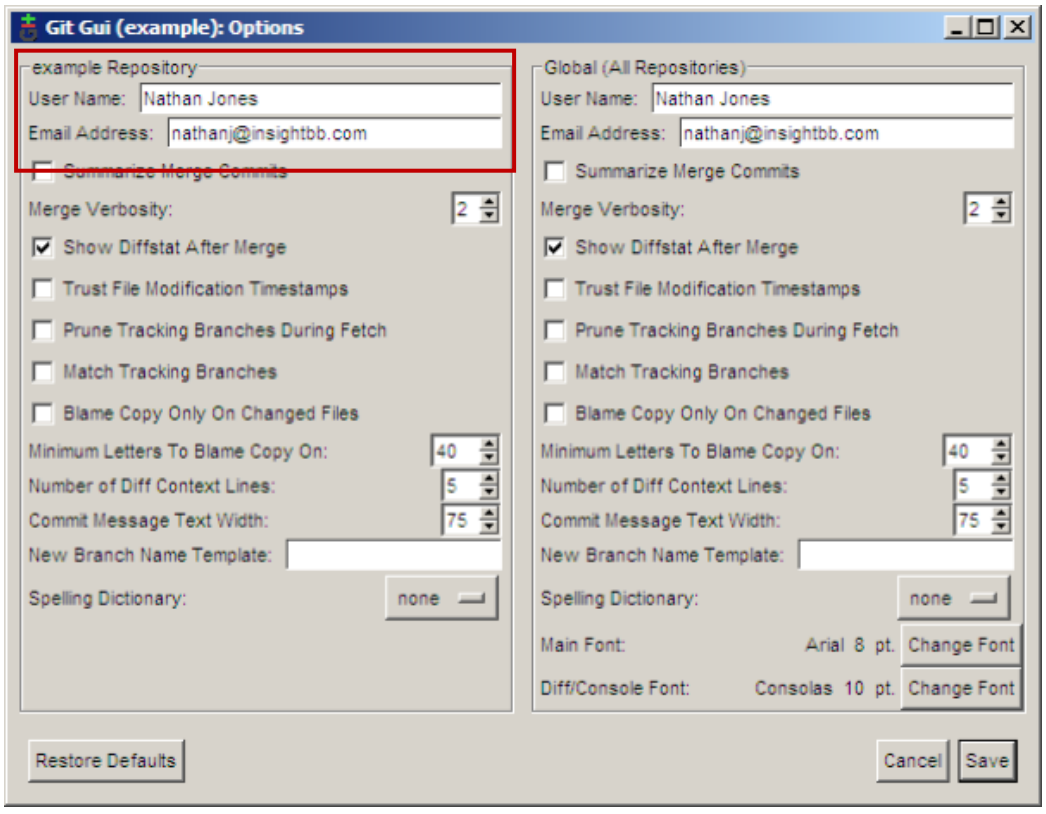


Рисунок 9 - Диалог опций

3. Закрепление изменений (commiting)

По сути Git, это система управления коммитами, то есть закреплениями изменений в проекте. Когда хранилище создано, нужно создать что-нибудь для фиксации. Для примера создадим файл *sandbox.c* со следующим содержимым.

Редактирование выполняется в *блокноте* или другом редакторе:

```
#include <stdio.h>

int main(int argc, char **argv) {
   printf("Hello Git!\n");
   return 0;
}
```

Ни в коем случае
не оскорбляя разработчика.

Щелчок на кнопку *Rescan* (*Перечитать*) в Git Gui заставит его искать новые, измененные и удалённые файлы в директории. На следующем скриншоте (см. *Рисунок 10*) показано, что Git Gui нашёл новый файл.

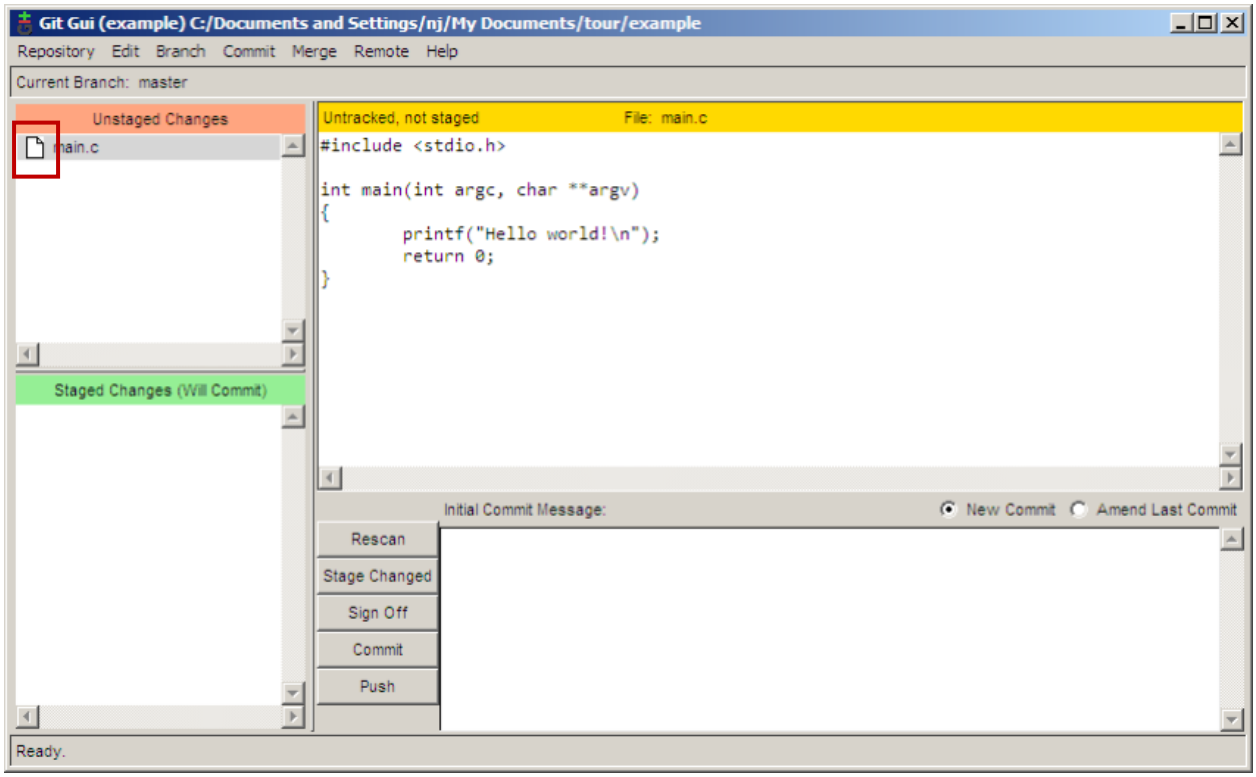


Рисунок 10 - Найден новый файл

Чтобы добавить этот файл в фиксацию, необходимо один раз щёлкнуть по иконке слева от имени файла. Файл будет перемещён с *Unstaged Changes* (*Измененено*) панели на *Staged Changes* (*Поэтапные изменения* или *Подготовлено*) панель. Теперь можно добавить *сообщение о фиксации* (*commit message*) и зафиксировать изменения *Commit* (*Сохранить*) кнопкой (см. *Рисунок 11*).

Наша первая программа выводит 'hello Git'. Сделаем программу более персонализированной. Перепишем ее, чтобы выводилось 'hello' пользователю. Измененный код представлен ниже:

```
#include <stdio.h>
#include <string.h>
...
```

```
...
int main(int argc, char **argv)
{
   char name[255];
   printf("Enter your name: ");
   fgets(name, 255, stdin);
   printf("length = %d\n", strlen(name));    /* debug line */
   name[strlen(name)-1] = '\0';    /* remove the newline at the end */

   printf("Hello %s!\n", name);
   return 0;
}
```

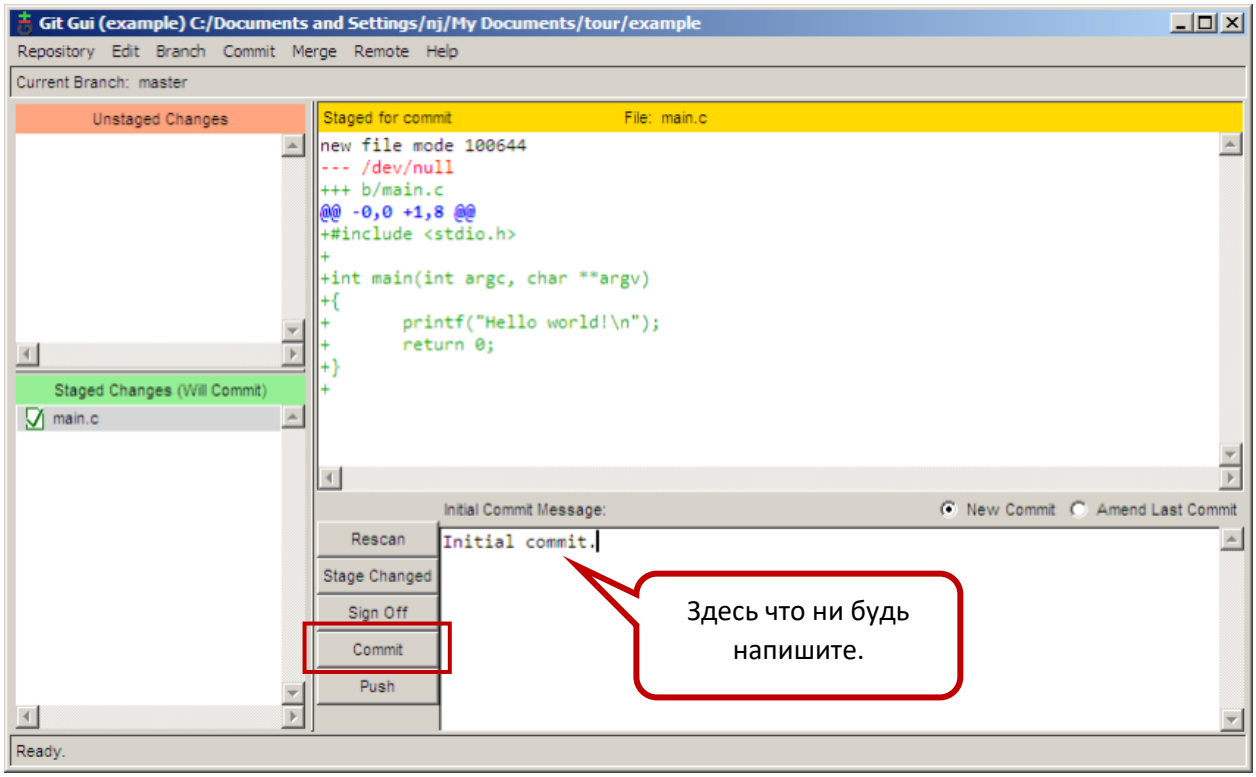


Рисунок 11 - Фиксация изменений

Допустим в программе есть ошибка. Для того чтобы найти, по какой причине возникает данная проблема, нужно добавить отладочную строку кода. Git gui позволяет зафиксировать изменение без этой отладочной линии, но при этом сохранить эту строку в рабочей копии, чтобы продолжить отладку. Сначала необходимо щёлкнуть *Rescan* (*Перечитать*) для поиска изменений. В результате изменения выделяются *красным* (удаленные линии) или *зеленым* (добавленные линии).

Снова *подготовим* (*stage*) изменения к фиксации, для этого опять щёлкнем по иконке слева от файла чтобы. Файл будет перенесен в нижнее окно. Затем

нужно правой кнопкой мыши щелкнуть по отладочной линии и выбрать *Unstage Line From Commit* (*Убрать строку из подготовленного*) (см. *Рисунок 12*).

После данного действия отладочная линия не будет подготовлена (*unstaged*) к фиксации, в то время как остальные изменения будут. Необходимо только опять заполнить сообщение фиксации и зафиксировать изменения щёлкнув по *Commit* (*Сохранить* или *Закрепить*) (см. *Рисунок 13*).

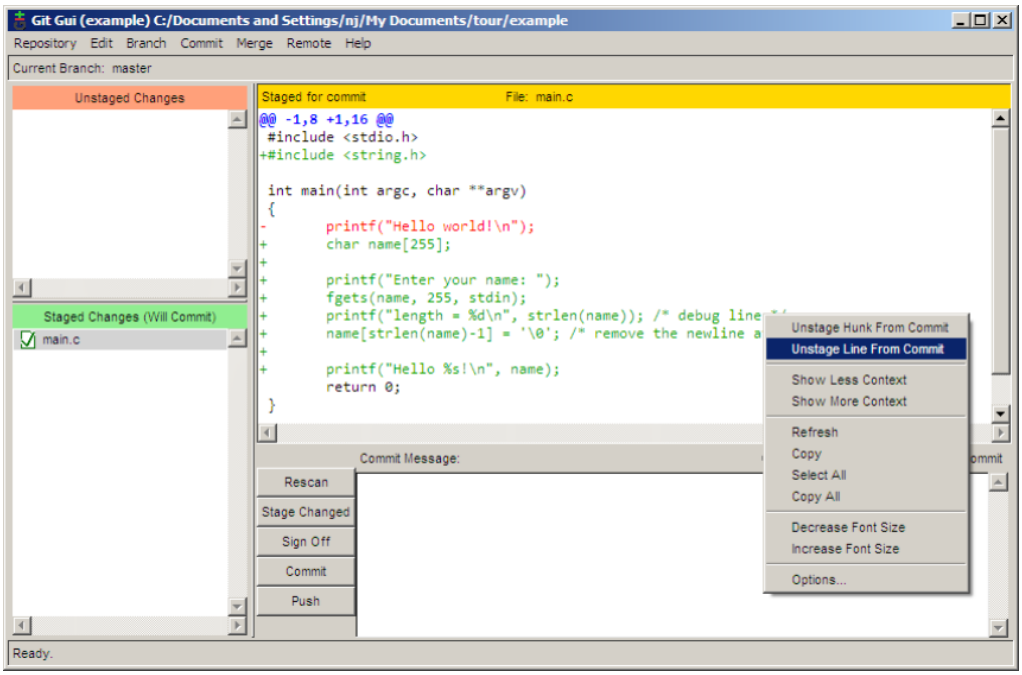


Рисунок 12 - Подготовка отладочной линии

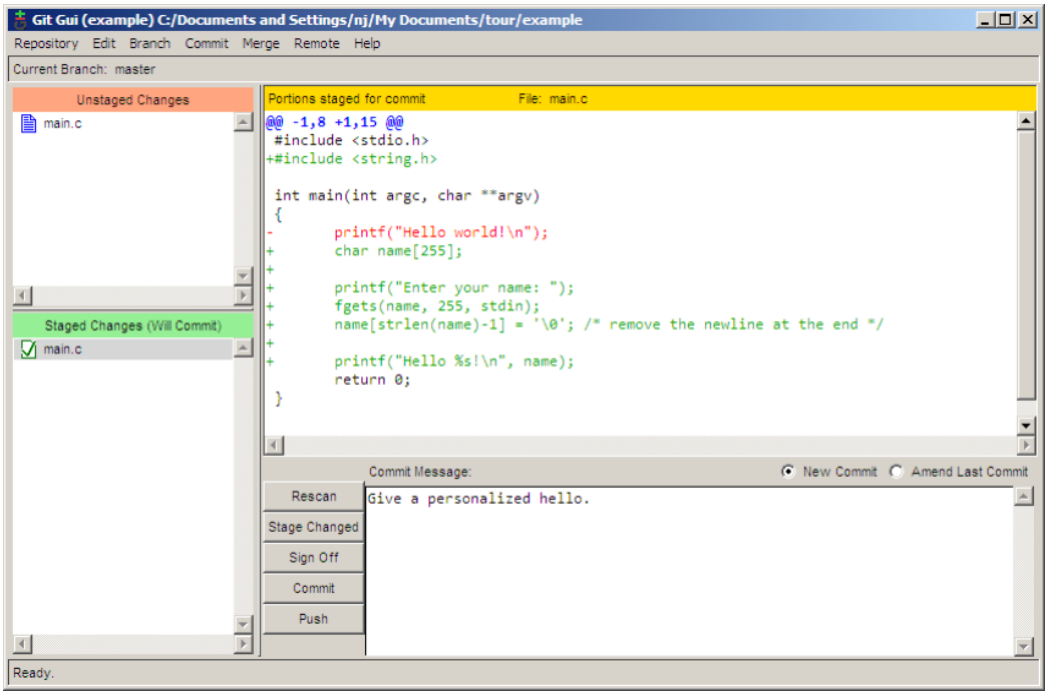


Рисунок 13 - Фиксация изменений без отладочной строки

4. Ветвление

Начнем добавлять новые возможности в нашу следующую большую версию программы. Чтобы сохранить стабильную версию, в которой можно исправлять ошибки, создадим *ветку* (*branch*) для наших новый разработок (см. *Рисунок 14*). Чтобы создать новую ветку в Git GUI, выберите *Branch* → *Create* (*Ветвь* → *Создать*). В процессе создания ветки можно задать ее название. Назовем ветку *lastname*, т.к. добавляем в код возможность спрашивать у пользователя фамилию. Опции по умолчанию подходят без изменений, так что нужно просто ввести имя и щёлкнуть *Create*.

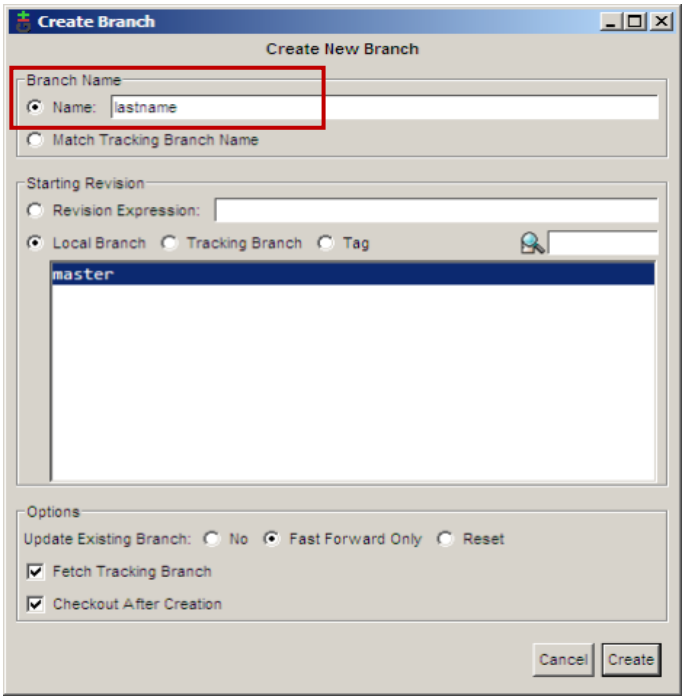


Рисунок 14 - Создание новой ветки

В ветке *lastname* можно делать новые модификации исходного файла-песочницы:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
   char first[255], last[255];
   printf("Enter your first name: ");
   fgets(first, 255, stdin);
   first[strlen(first)-1] = '\0'; /* remove the newline at the end */

   printf("Now enter your last name: ");
   gets(last); /* buffer overflow? what's that? */
```

```
    printf("Hello %s %s!\n", first, last);
    return 0;
}
```

Необходимо зафиксировать изменения (см. *Рисунок 15*). В примере на рисунке изменения фиксируются с использованием другого имени. Как это делается, рассмотрим позже. Обычно мы всегда будем использовать одно и тоже имя для фиксаций.

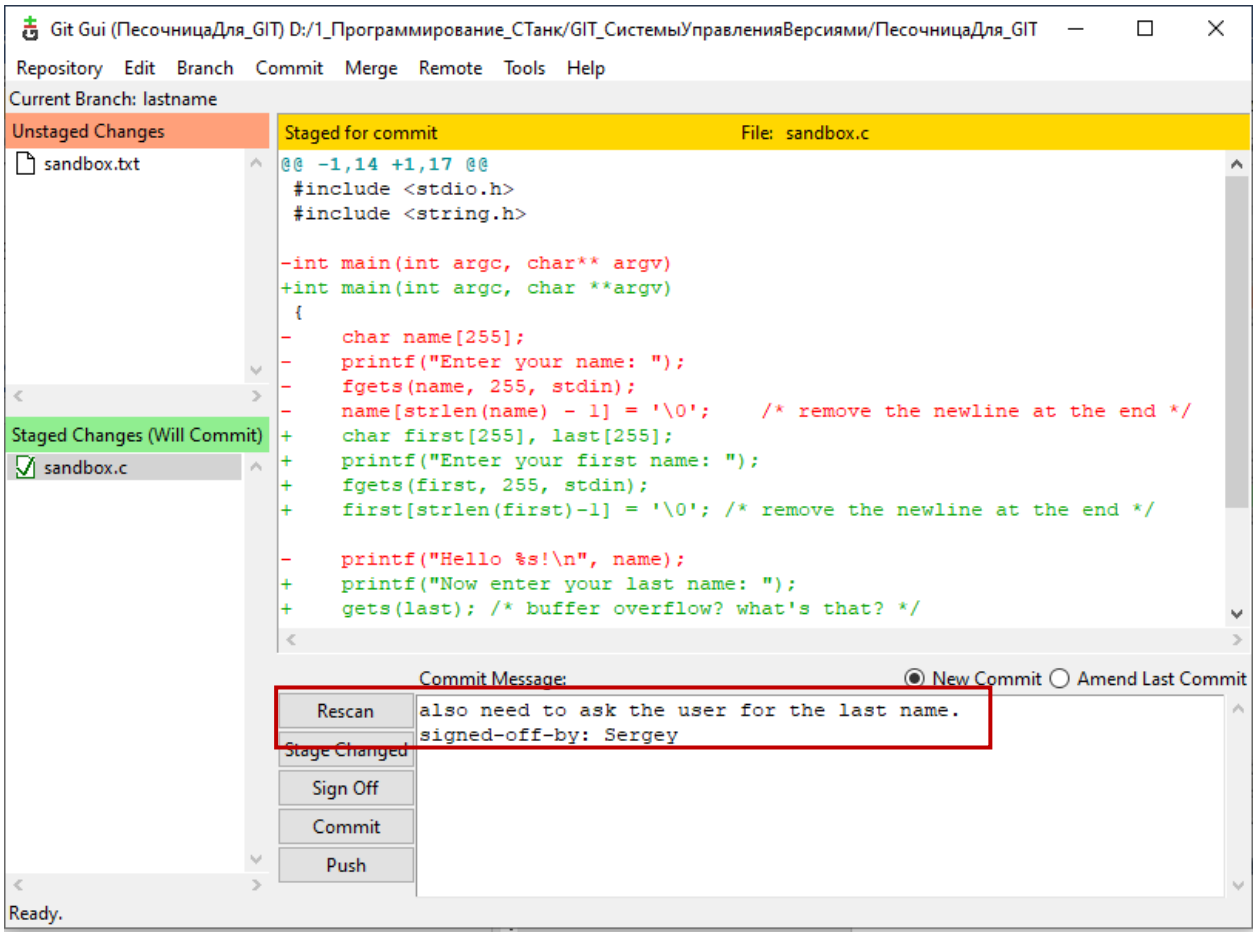


Рисунок 15 - Фиксация изменений в новой ветке

Другой разработчик проинформировал нас, что отсутствие запятой после прямого обращения к кому-то, это серьёзная ошибка. Чтобы исправить её в нашей стабильной ветке, вы сначала должны переключиться назад на неё. Для этого необходимо использовать *Branch → Checkout* (*Ветвь → Перейти*) (см. *Рисунок 16*). Теперь можно исправить ошибку (см. *Рисунок 17*).

Для просмотра истории изменений необходимо выбрать *Repository → Visualize All Branch History* (*Репозиторий → Показать историю всех ветвей*). История отобразится в виде графа и таблицы (см. *Рисунок 18*).

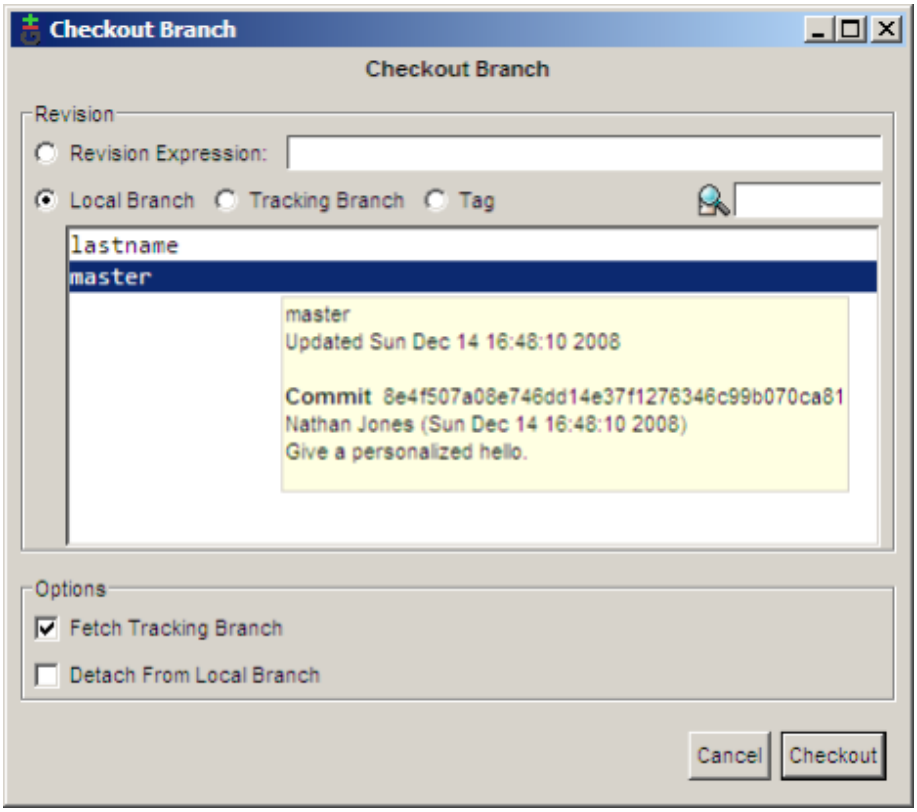


Рисунок 16 - Переключение в стабильную ветку

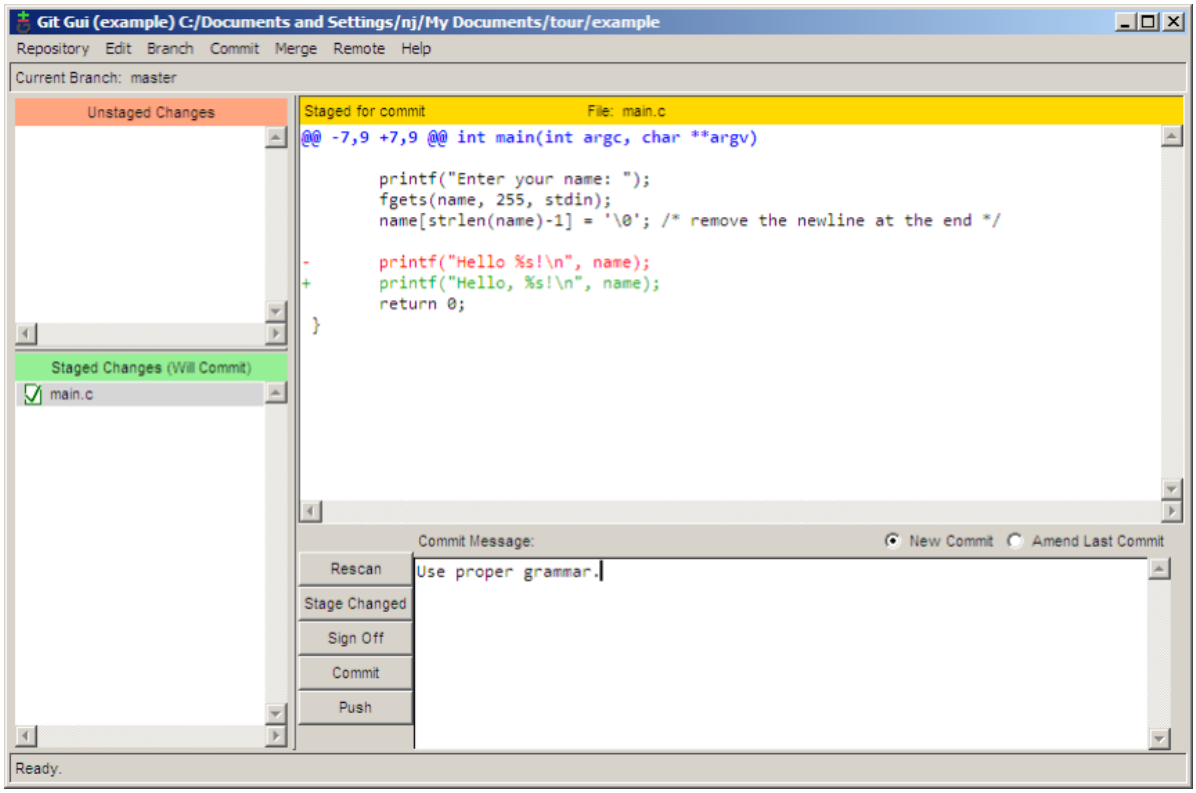


Рисунок 17 - Исправление ошибки

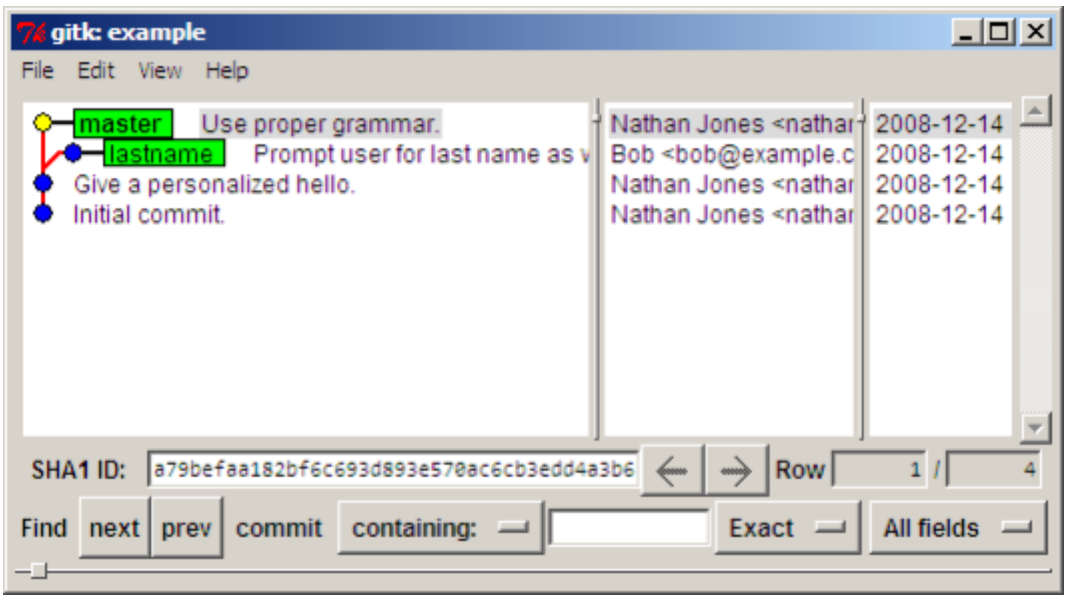


Рисунок 18 - Демонстрация истории

5. Слияние

Если *lastname* ветка достаточно стабильна, ее можно влить в *master* ветку. Чтобы выполнить слияние, необходимо использовать *Merge → Local Merge* (*Слияние → Локальное слияние*) (см. *Рисунок 19*).

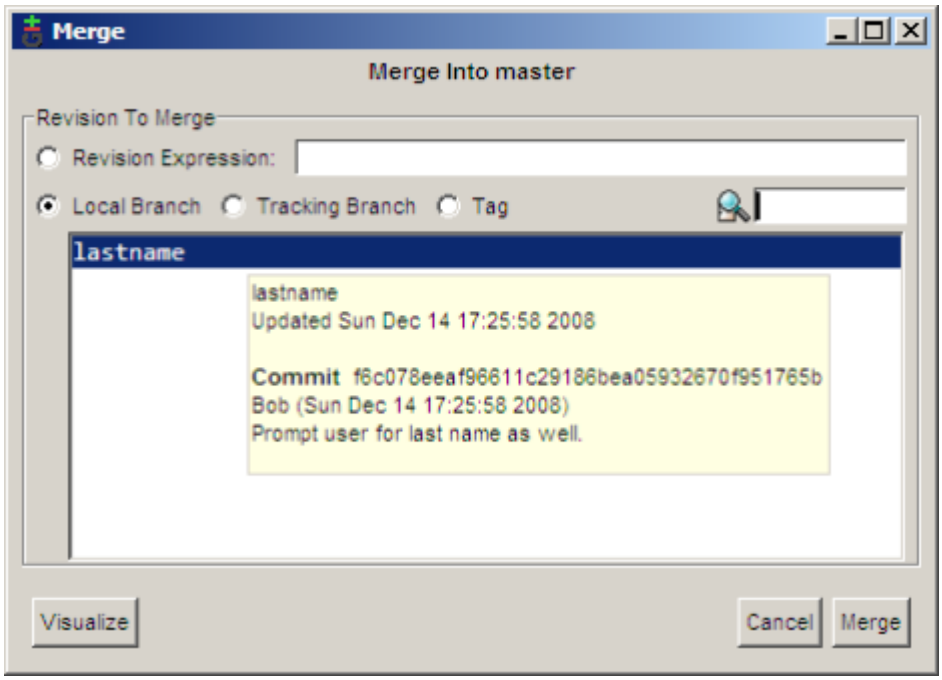


Рисунок 19 - Слияние веток

Так как две разные фиксации делали два разных изменения на одной и той же строке (см. *Рисунок 21*), происходит *конфликт* (*conflict*) (см. *Рисунок 20*).

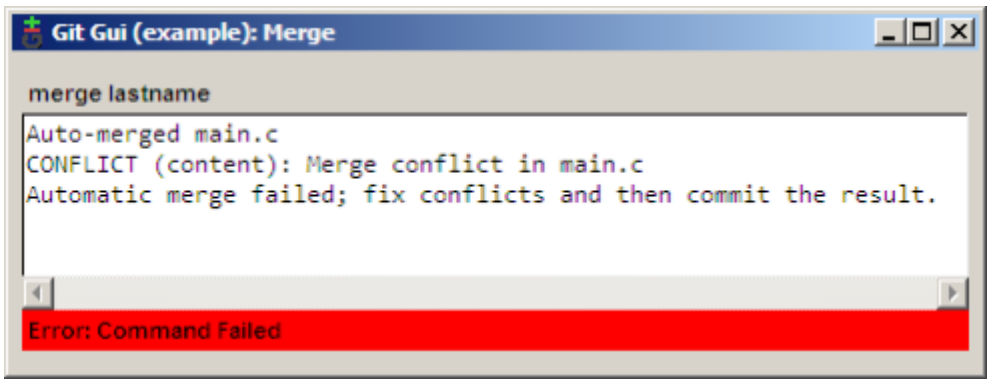


Рисунок 20 - Информация о конфликте при слиянии

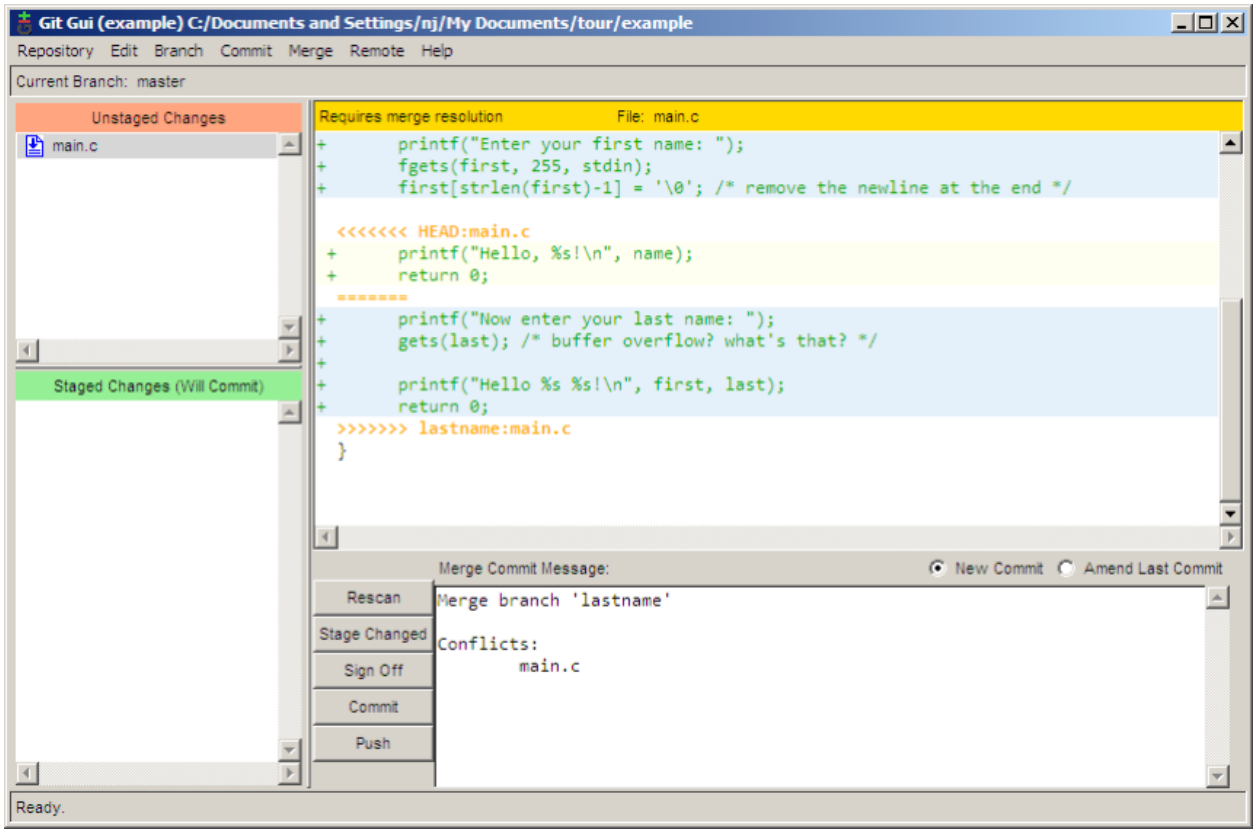


Рисунок 21 - Демонстрация различий документа в разных ветках

Конфликт может быть разрешён с использованием любого текстового редактора (нужно оставить в тексте только правильный вариант).

После разрешения конфликта необходимо подготовить изменения, щёлкнув на иконке файла, и зафиксировать слияние, щёлкнув по *Commit*-кнопке, в редакторе внести изменения в файл и сохранить его: *Перечитать – Подготовить – Сохранить* (см. *Рисунок 22*).

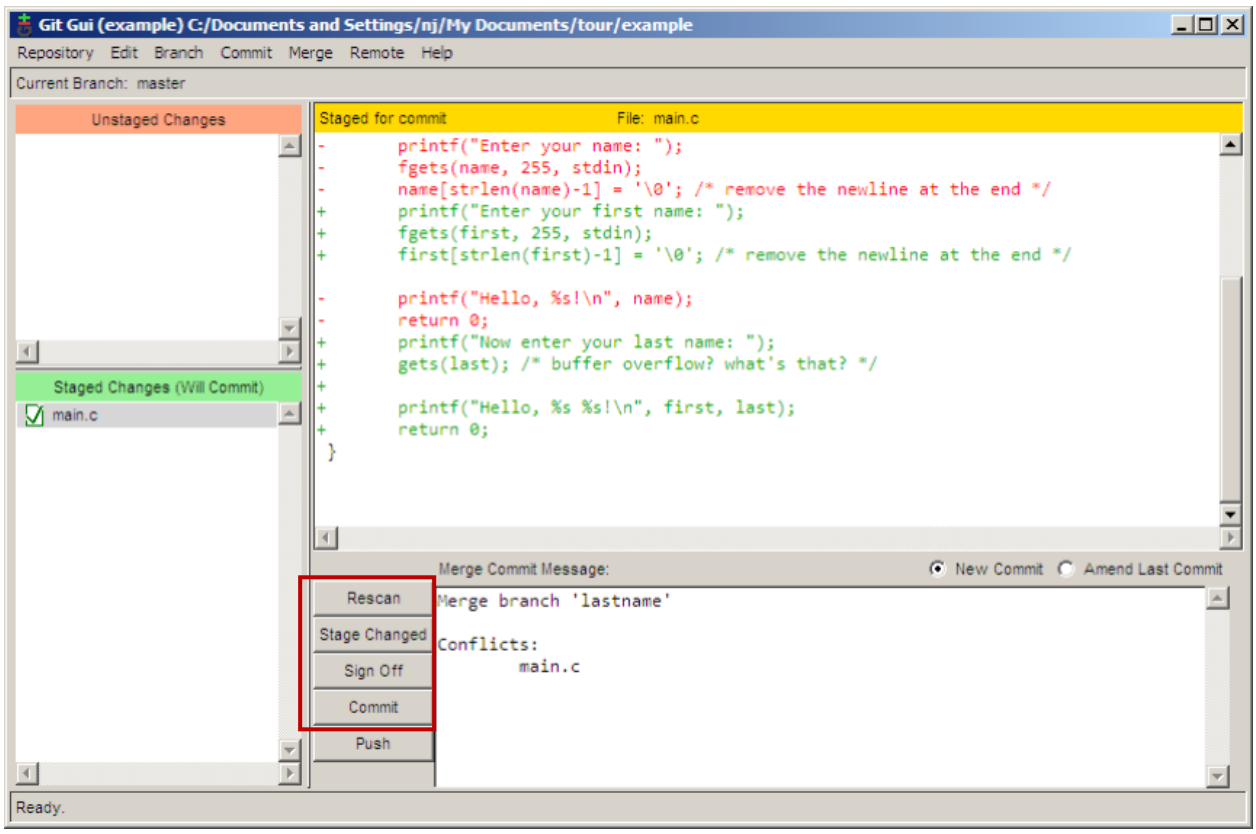


Рисунок 22 - Разрешение конфликта и фиксация изменений

6. Просмотр истории

Чтобы уменьшить объем файла *main.c*, вынесем код, спрашивающий имя пользователя, в отдельную функцию. Также можно вынести эту функцию в отдельный файл. Хранилище теперь содержит файлы *main.c*, *askname.c*, и *askname.h*.

```
/* main.c */
#include <stdio.h>

#include "askname.h"

int main(int argc, char **argv)
{
   char first[255], last[255];
   askname(first, last);
   printf("Hello, %s %s!\n", first, last);
   return 0;
}
```

```
/* askname.h */
void askname(char *first, char *last);
```

```
/* askname.c */
#include <stdio.h>
#include <string.h>

void askname(char *first, char *last)
{
   printf("Enter your first name: ");
   fgets(first, 255, stdin);
   first[strlen(first)-1] = '\0'; /* remove the newline at the end */

   printf("Now enter your last name: ");
   gets(last); /* buffer overflow? what's that? */
}
```

Файлы создаются в редакторе. Затем необходимо *перечитать* репозиторий, *подготовить* все и *сохранить*.

Для просмотра и изучения истории хранилища необходимо выбрать *Repository → Visualize All Branch History*. На следующем скриншоте пытаемся найти в какой фиксации была добавлена *last* переменная, ища все фиксации, в которых было добавлено или убрано слово *last* (см. Рисунок 23).

Фиксации, которые подходят под условия поиска отмечены жирным шрифтом, чтобы быстро и легко обнаружить нужную фиксацию. Можно посмотреть старую и новую версии. Цветом выделены изменения.

В Git GUI есть возможность посмотреть историю даже через несколько дней после изменений. Например, другой пользователь, просматривая код, заметил, что gets функция может вызвать переполнение буфера. Этот человек может запустить git blame для того, чтобы увидеть, кто последний раз редактировал эту линию кода. Git может обнаружить изменения, даже если их вносили разные пользователи. Например, Боб зафиксировал линию в хранилище, а мы последние, кто трогал её, когда переместили строку в другой файл.

Чтобы запустить *blame*, нужно выбрать *Repository → Browse master's Files* (Репозиторий → Показать файлы ветви *master*). Из дерева, которое появится, дважды щёлкните на файле с интересующей строкой, который в данном случае *askname.c*. Наведённая на интересующую линию мышка показывает нам подсказку, которая говорит всё, что нам надо знать (см. Рисунок 24).

Здесь мы можем видеть, что эта линия была зафиксирована Бобом в фиксации *f6c0*, а затем мы её переместили в её новое месторасположение в фиксации *b312*.

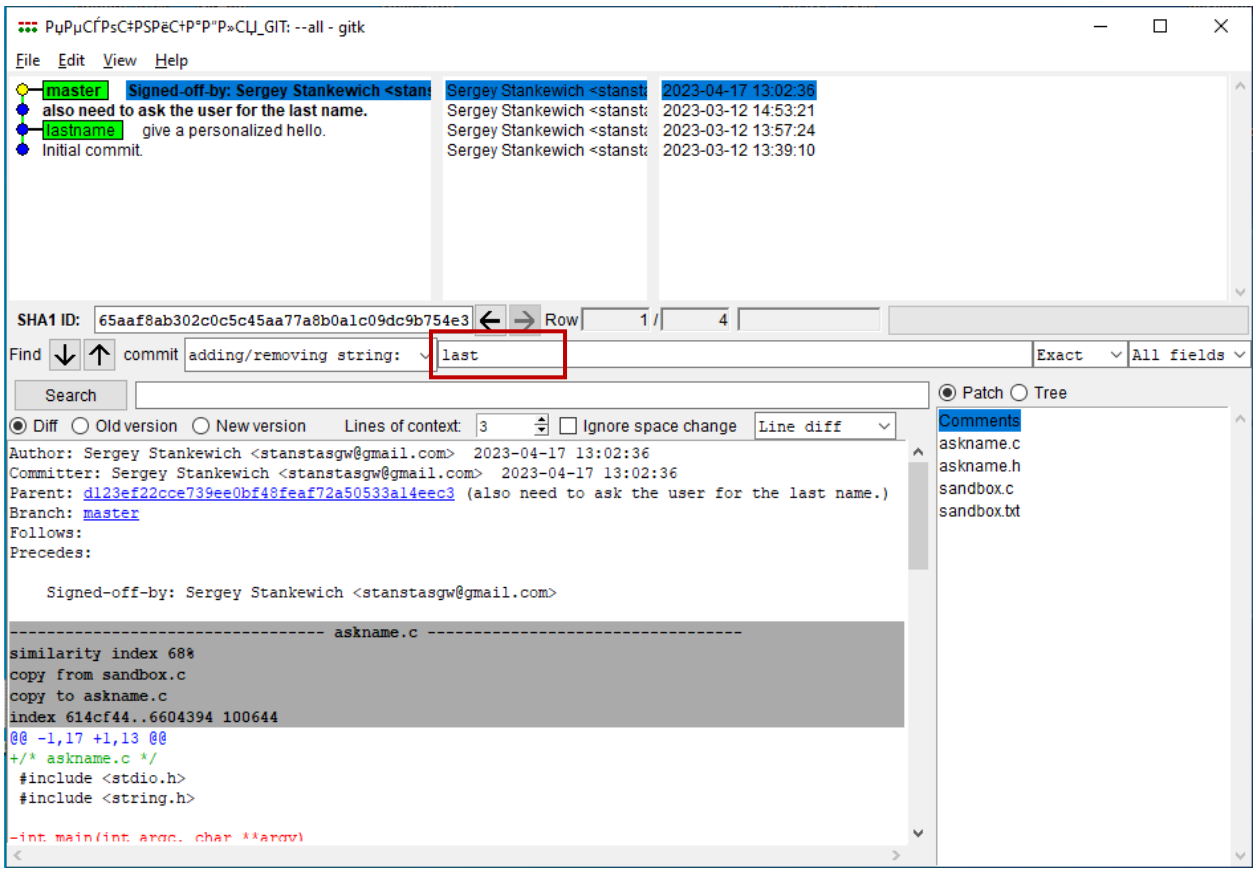


Рисунок 23 - Поиск в истории добавления last переменной.

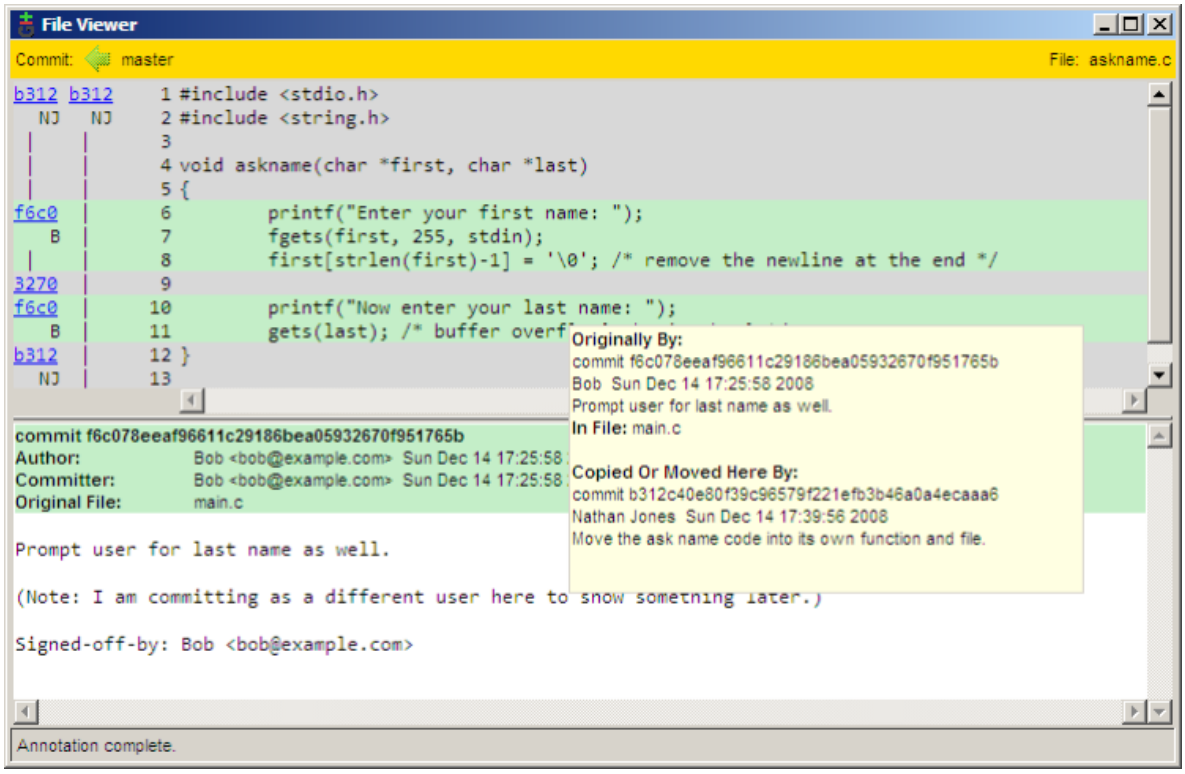


Рисунок 24 - Просмотр истории изменений разных пользователей.

7. Отмена изменений (revert или reset)

Для отмены внесенных изменений до состояния последней фиксации:

Меню: Состояния – Отменить изменения.
Для отката к конкретной фиксации (reset):

Меню *Репозиторий – История ветки* (выбрать нужное состояние и в контекстном меню выбрать: «Установить для ветви это состояние»). Возможны варианты (мягкий – изменение только индекса или жесткий – изменения в индексе и на диске).

Для отката через новую фиксацию создается новая фиксация, содержащая изменения, обратные зафиксированным (revert):

Меню *Репозиторий – История ветки* – (выбрать нужное состояние и в контекстном меню выбрать *revert this commit*).

8. Публикация изменений на удалённом сервере

Зарегистрируйтесь на сайте https://github.com/ (укажите логин, почту и пароль, дальше выберите бесплатный доступ). GitHub – не единственный доступный удаленный репозиторий Git. Это, однако, самое популярное решение, и мы будем использовать его в качестве примера.

Создайте новый репозиторий. Для этого на сайте выберите *Create new repository*, укажите его название, тип (публичный) и нажмите *Create repository*. После создания репозитория будет отображено его имя - адрес (в формате HTTPS) и команды для работы с ним в режиме командной строки (для желающих).

Далее из каталога репозитория на локальном ПК вызовите Git Gui, выберите меню: *Внешние репозитории → Добавить*, в новом окне укажите псевдоним удаленного репозитория (любой, на Рисунок 25 это *Test1*) и его адрес (https://github.com/ivanov/test2.git , где *ivanov* – логин пользователя сайта, *test2* – название репозитория на сайте). Рекомендуется адрес скопировать с сайта.

Затем выбрать в меню *Внешние репозитории → Отправить*, указать псевдоним удаленного репозитория и отправляемую ветку, нажать «Отправить» (см. Рисунок 26). При запросе ввести логин и пароль, с которыми регистрировались на сайте.

Если все успешно (см. Рисунок 27), то на сайте перейти в репозиторий и просмотреть его содержимое. Можно просматривать историю изменений репозитория, содержимое фиксаций, изменения.

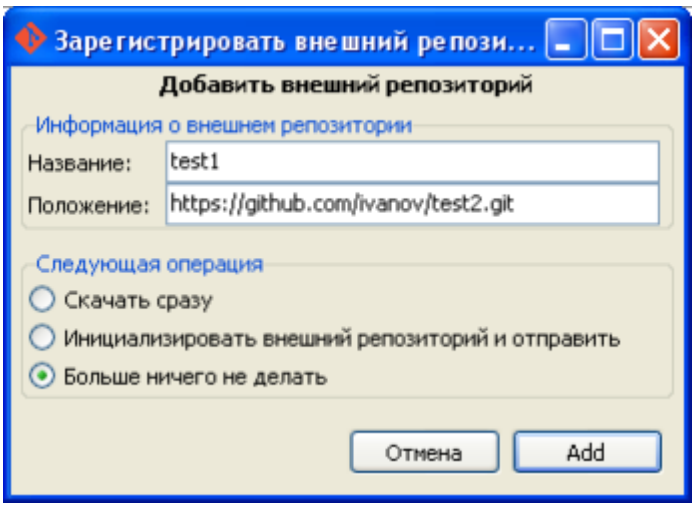


Рисунок 25 – Добавление внешнего репозитория.

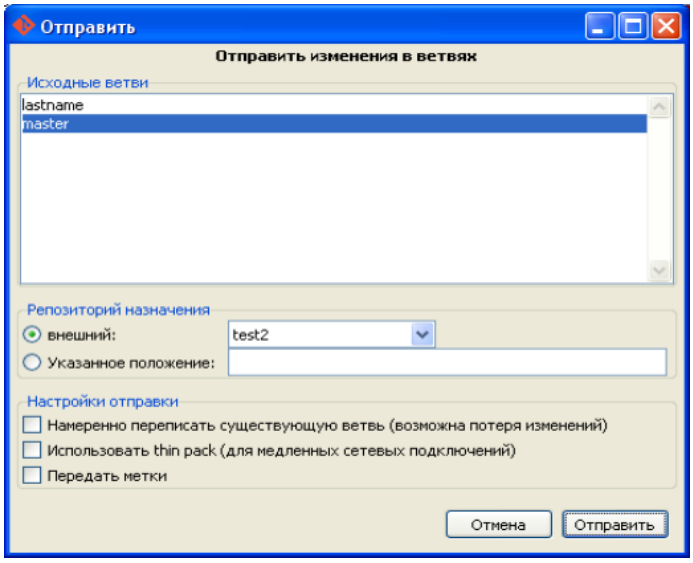


Рисунок 26 – Отправка изменений в ветвях.

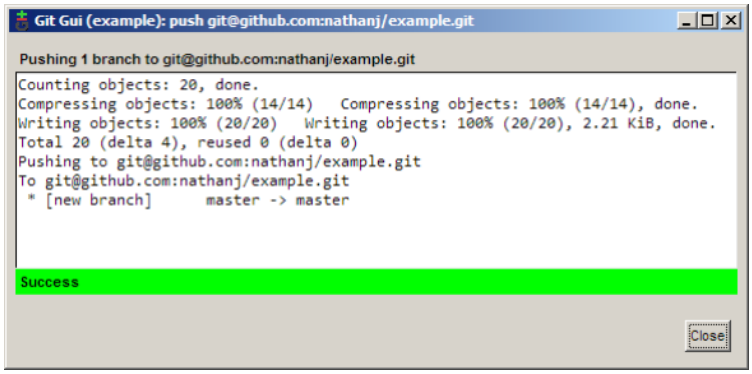


Рисунок 27 – Подтверждение отправки.

Подробнее как выполнить задание можно прочитать здесь:

https://coderlessons.com/articles/veb-razrabotka-articles/git-na-windows-dlia-novichkov

9. Получение изменений с удалённого сервера

Для получения копии удаленного репозитория нужно открыть проводник, щелкнуть правой кнопкой мыши и из контекстного меню выбрать Git GUI. Вам будет показан диалог создания (см. Рисунок 27).

Выбрать Clone (Клонировать существующий репозиторий). Будет открыт диалог клонирования (см. Рисунок 28). В качестве источника укажите удаленный репозиторий (его адрес), в качестве приемника – новый каталог.

Рисунок 28 – д

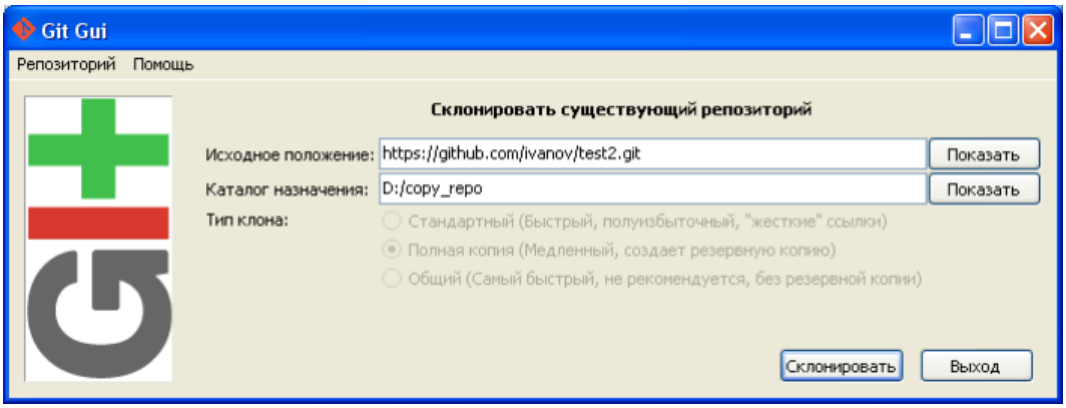


Рисунок 29 –

Нажать «Клонировать». Если все успешно, то откроется среда Git Gui, в которой возможно посмотреть содержимое файлов и историю изменений.

Дальше можно работать над проектами независимо. После внесения изменений и их фиксации отправим изменения обратно на сервер (меню Внешние репозитории → Отправить, указать псевдоним удаленного репозитория и отправляемую ветку, нажать «Отправить»).

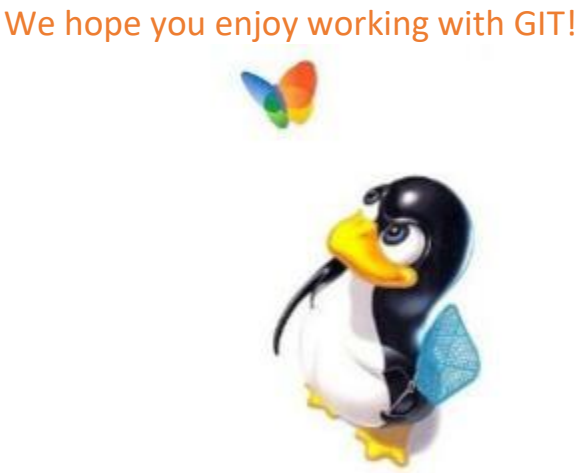
Предварительно необходимо разрешение владельца репозитория на внесение изменений. Для этого владелец проекта на сайте выбирает (сверху вверху) New Collaborator, откроется страница участников проектов. На ней нужно указать имя добавляемого участника (он должен быть найден по имени или его

части) и нажать Add Collaborator. Новый участник будет добавлен в список участников.

Все участники проекта могут вносить свои изменения, используя адрес репозитория, свои логин и пароль.

Код, залитый на github, могут смотреть и скачивать другие люди и использовать программу. Один человек, Фред, скопировал к себе репозиторий (сделал вилку – Fork) и добавил в него собственные изменения. Теперь, когда он добавил свой код, мы можем «перетянуть» его изменения в своё хранилище.

Для получения изменений Фреда, необходимо использовать Remote → Fetch from → fred (Внешние репозитории → Получить) (см. Рисунок 29).



Упражнение №2. Стратегия разработки или последовательность работы с репозиторием

Пример последовательности работы с Git в команде разработки приведен на Рисунок 34

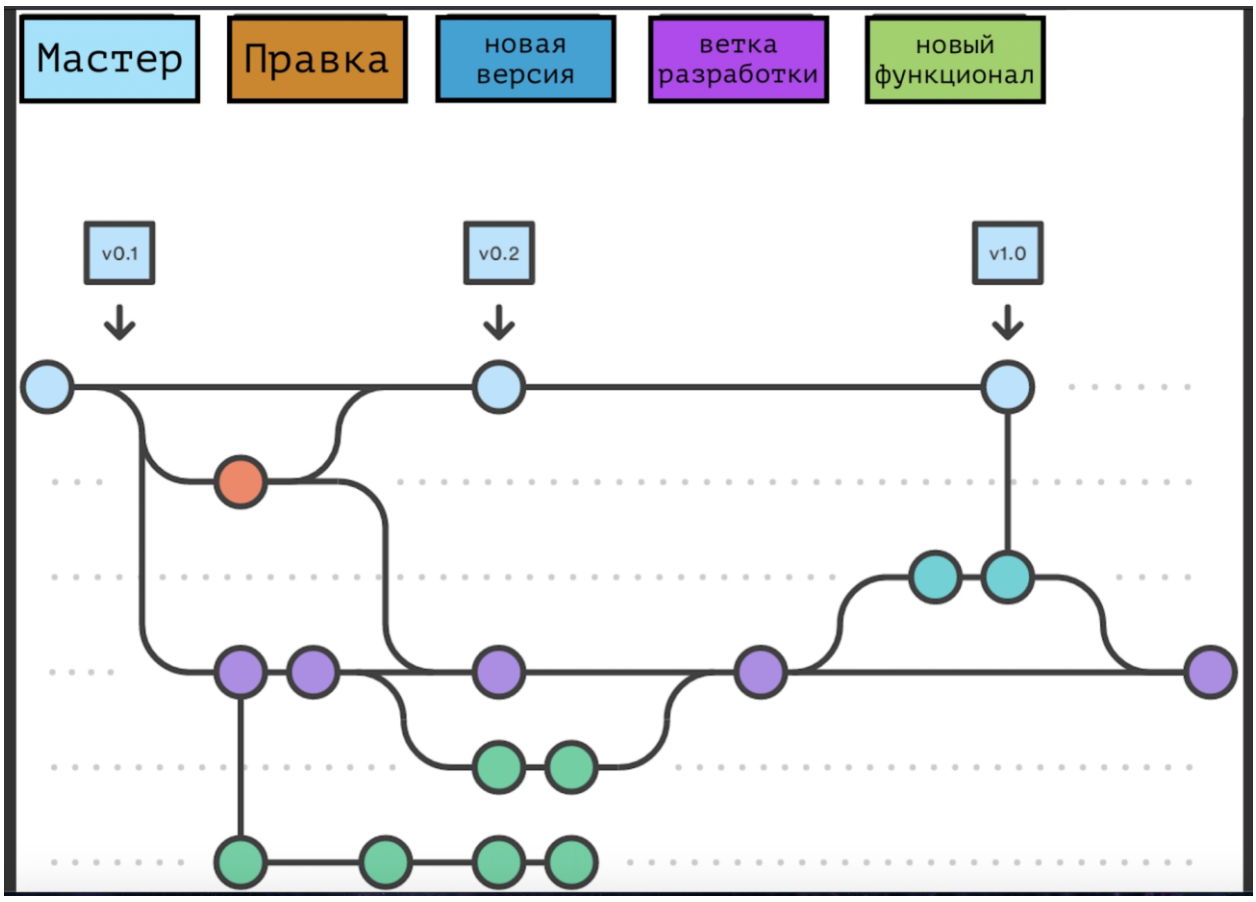


Рисунок 34. Последовательность работы с Git в команде разработки.

```
git log --since=«1 day»
```

Методические указания к лабораторной работе №2
по курсу «Технологии разработки программного обеспечения»

Преимущества отказа от линейной модели:

- у программиста появляется дополнительная ги

X: 26,0мм
Y: 266,1мм
38 / 42

Преимущества отказа от линейной модели:

- у программиста появляется дополнительная гибкость: он может переключаться между задачами (ветками);
- под рукой всегда остается «чистовик», ветка master;
- коммиты становятся мельче и точнее.

We hope you enjoy working with GIT!

ЗАДИНИЯ

Задание 1. Действия при работе с локальным репозиторием

Выполните с помощью Git GUI следующую последовательность действий с локальным репозиторием, реализую определенную стратегию разработки:

1. Создайте рабочую директорию проекта, например, gitgui-demo.
2. Перейдите в рабочую директорию.
3. Создайте репозиторий в директории.
4. Индексируйте все существующие файлы проекта (добавляем в репозиторий).
5. Создайте инициализирующий коммит.
6. Создайте новую ветку.
7. Переключитесь в новую ветку.
8. После непосредственной работы с кодом индексируйте внесенные изменения.
9. Сделайте коммит.
10. Переключитесь в основную ветку.
11. Проверьте отличия между последним коммитом активной ветки и последним коммитом экспериментальной.
12. Проведите слияние.
13. Если не было никаких конфликтов, удалите ненужную больше ветку.
14. Оцените проведенную за последний день работу.

Если что-то не получилось сделать, отметьте это в отчете, и попробуйте объяснить почему так вышло.

Задание 2. Действия при работе с удаленным репозиторием

Стр. 39

Общие требования

Выполните вышеуказанные упражнения, покажите результаты и ход выполнения укажите в отчете подтверждая скриншотами.

Выберите приемлемую вам систему контроля версий, зарегистрируйтесь в ней. Создайте команду разработчиков, добавив членов команды. Создайте проект (предприятие), и начните работу.

Название проекта должно содержать атрибуты нашей дисциплины, номера лабораторной работы, имена разработчиков и т.п. Ваш проект должен легко идентифицироваться среди тысяч других.

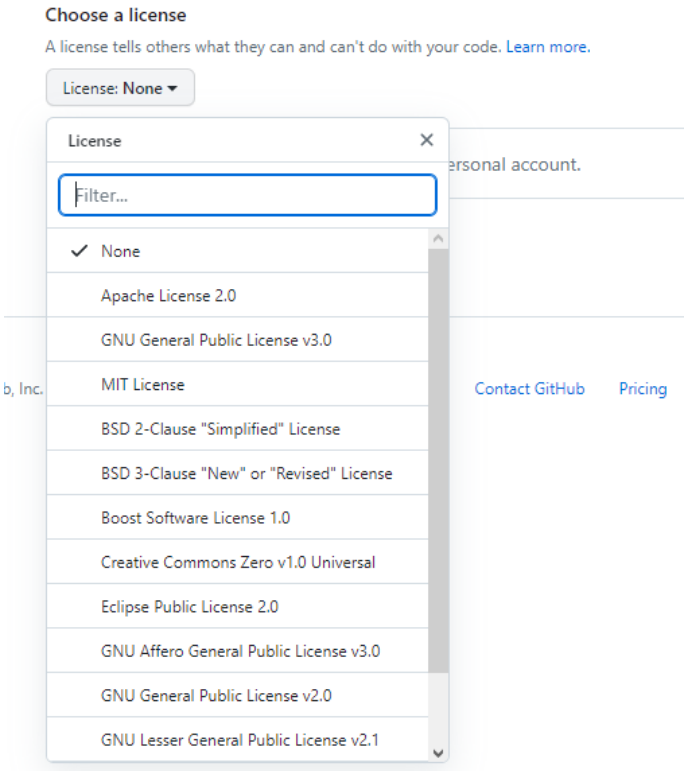
Желательно ограничить доступ к проекту, но у преподавателя должна быть возможность просматривать и работать с этим проектом.

Напишите отчет. Ключевые моменты вашей работы должны быть отражены с помощью скриншотов. Отчет должен храниться в проекте в системе контроля версий.

Индивидуальное задание

Изучите типы лицензий и сделайте реферат по типу лицензии в соответствии с вариантами заданий, представленными ниже:

№ варианта	Тип лицензии
1.	
2.	
3.	
4.	
5.	



«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть возможными»

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое системы контроля версий?
2. В каких сферах деятельности могут использоваться системы контроля версий?
3. Назовите наиболее популярные СКВ для разработки ПО.
4. Какую СКВ вы выбрали для работы и почему?

5. Какие виды и типы контроля версий существуют?
6. Что такое GIT?
7. Знаете ли вы историю создания GIT?
8. Кто является создателем GIT?
9. Как слово «git» переводится с английского языка?
10. Укажите основные команды, компоненты и процедуры работы в СКВ.
11. Что такое система управления версиями?
12. Как создать репозиторий?
13. Что такое commit, и почему Git называют системой управления коммитами?
14. Как создать ветку?
15. Как провести слияние? Как разрешить конфликт и что это такое?
16. Как зафиксировать изменения?
17. Как провести откат? Различия в reset и revert, мягкий и жесткий reset.
18. Какова последовательность действий при работе с локальным репозиторием?
19. Какова последовательность действий при работе с удаленным репозиторием?
20. Каковы возможности при работе с удаленным репозиторием? Как его клонировать, получать и отправлять данные?
21. Что будет если в процессе работы изменить название папки с рабочим проектом, у нас это «песочница».
22. Почему перед в названии “.git” в папке-песочнице стоит точка, к какому типу файла относится эта папка?

ИСТОЧНИКИ

https://ru.wikipedia.org/wiki/Система_управления_версиями

Установка GitHub Desktop

https://docs.github.com/ru/desktop/installing-and-configuring-github-desktop/installing-and-authenticating-to-github-desktop/installing-github-desktop

Проверка подлинности в GitHub

https://docs.github.com/ru/desktop/installing-and-configuring-github-desktop/installing-and-authenticating-to-github-desktop/authenticating-to-github

https://coderlessons.com/articles/veb-razrabotka-articles/git-na-windows-dlia-novichkov

Небольшое руководство по Git и развертыванию

Прежде чем я отпущу вас, вот список фантастических ресурсов для продолжения изучения Git.

- Git Essentials (курс Tuts + Premium)
- Github – неограниченные бесплатные публичные репозитории
- Bitbucket – неограниченное количество бесплатных публичных и частных репозиториев
- Beanstalk – Частный Git с отличными развертываниями FTP
- DeployHq – Развертывание любого репозитория Git через FTP
- Простое управление версиями с помощью Git
- Terminal, Git и GitHub для остальных