Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Основы программной инженерии (ОПИ)

ОТЧЕТ

по лабораторной работе № 2

Выполнил:

Студент гр. 351051 Носович А.А.

Проверил: Фадеева Е.Е.

Минск 2023

ИНДИВИДУАЛЬНАЯ РАБОТА №2

**ТЕМА: ТЕХНОЛОГИИ КОНСТРУИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ.**

**1. Контрольные вопросы:**

1. Системы контроля версий разделяются на локальные, централизованные и распределенные.
2. - Локальные СКВ основываются на простой базе данных в которой хранятся изменения нужных фалов.

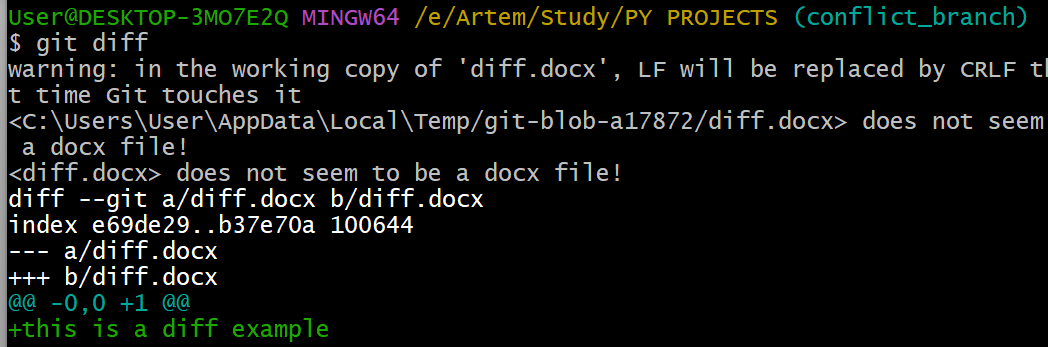
- Централизованные СКВ позволяют сотрудничать разработчиками и в них есть центральный сервер, на котором хранятся все файлы под версионным контролем, а также ряд клиентов, которые взаимодействуют с этим сервером.

- В распределенных СКВ в отличие от централизованных систем клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий.

1. - Хранилище – это то место, где хранятся файлы проектов (коды, текст, изображения и прочее). Каждый проект должен иметь собственное хранилище для сохранения изменений в проектах. При работе над проектом, хранилище является централизованным источником истины рабочей группы для управления изменениями. Примером такого хранилища является GitHub.

* В истории сохраняются все коммиты, по которым можно посмотреть автора коммита, commit message, дату коммита и его хэш. А также можно увидеть измененные файлы и изменения в каждом файле. Сделать это в командной строке Git можно с помощь команды git log.
* Рабочая копия 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. На примере проекта это был бы случай, когда основные данные хранятся на удаленном репозитории в GitHub, а мне, чтобы подтянуть эти изменения себе локально на ПК нужно выполнить команду git clone, git pull / git fetch. Это и была бы рабочая копия. Но так как основная разработка программы в моем проекте велась только мной, а не кем-то другим, клонировать данные с удаленного репозитория и создавать локально тем самым рабочую копию смысла не было.

1. Под веткой **branch** принято понимать независимую последовательность коммитов в хронологическом порядке, однако конкретно в Git реализация ветки выполнена как указатель на последний коммит в рассматриваемой ветке. После создания ветки уже новый указатель ссылается на текущий коммит. Ветки нужны для того, чтобы программисты могли вести совместную работу над проектом и не мешать друг другу при этом. Ветки всегда ответвляются от основной ветки (обычно это master или в моем случае main). Пример ветки в моем проекте это conflict\_branch.
2. Разность нужна для пониманию какие изменения были внесены на текущем коммите относительно предыдущего или на текущей ветке относительно той, в которую мы хотим вмержить свою. На скриншоте представлен пример того, что будет выведено в консоль Git после выполнения команды **git diff** (в случае если имеются изменения на ветке):



1. Команда **git merge** используется для слияния одной или нескольких веток в текущую. В практической части я использовал команду git merge master чтобы продемонстрировать пример конфликта. Эта команда совершила слияние содержимого на ветке master с содержимым на ветке conflict\_branch.
2. Команда **git commit** делает для проекта снимок текущего состояния изменений, добавленных в раздел проиндексированных файлов. Это сделано для того, чтобы в будущем в случае внесения изменений в файлы или удаления их была возможность вернуться к исходному состоянию. В практической части я использовал git commit с комментарием для создания снимка на локальном хранилище чтобы в последствии отправить эти изменения со снимка на удаленный репозиторий.
3. **Git rebase** в общих чертах похожа на git merge, однако в случае выполнения команды **git rebase** коммиты из ветки, которую мы пытаемся подлить в текущую, появляются поверх уже имеющихся коммитов на ветке, а в случае выполнения команды git merge происходит слияние изменений с двух веток в один коммит.
4. Git, Mercurial, TortoiseSVN, Bazaar, darcs и Preforce
5. Нумерация версий программного обеспечения разделяется на следующие варианты:

- Нумерация целым числом;

- Десятичная дробь;

- Последовательные числа;

- Нумерация годом;

- Дата;

- Алфавитно-цифровое название;

- Нумерация текстом.

Система версионирования выбирается по нескольким критериям: поддержка той или иной системы со стороны ПО для разработки (компилятора, системы контроля версий и т. п.), частота выхода новых версий и уровень их готовности.

**2. Практическая работа с Git на индивидуальном проекте из ИПР №1.**

Для отработки навыков использования хранилища на локальной машине я выбрал систему контроля версий **Git**, а в качестве веб-сервиса для хостинга проекта я выбрал **GitHub**.

Ссылка на публичный репозиторий в **GitHub**: <https://github.com/artyomnosovich/OPI>

Основная ветка в репозитории - **main**.

Для определения текущей директории в качестве директории репозитория я воспользовался командой **git init**

Чтобы добавить все файлы из текущей директории в новый локальный репозиторий я использовал команду: **git add .**

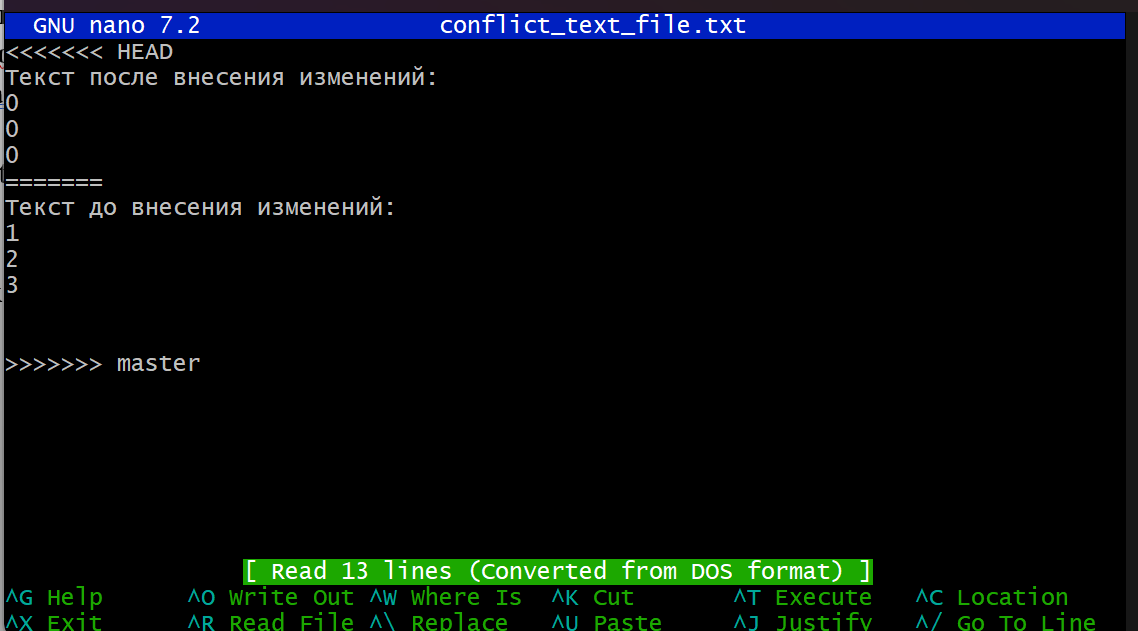
Первый коммит файлов мной выполнялся с помощью команды: **git commit -m "first task opi"**, где "first task opi" – комментарий к коммиту.

Для определения удаленного репозитория я использовал команду **git remote add origin** [**https://github.com/artyomnosovich/OPI**](https://github.com/artyomnosovich/OPI) , после чего можно воспользоваться командой **git remote -v** чтобы убедиться, что желаемая ссылка на удаленный репозиторий установлена.

Отправку изменений локального репозитория в удаленный репозиторий на GitHub я произвел с помощью команды: **git push origin main**

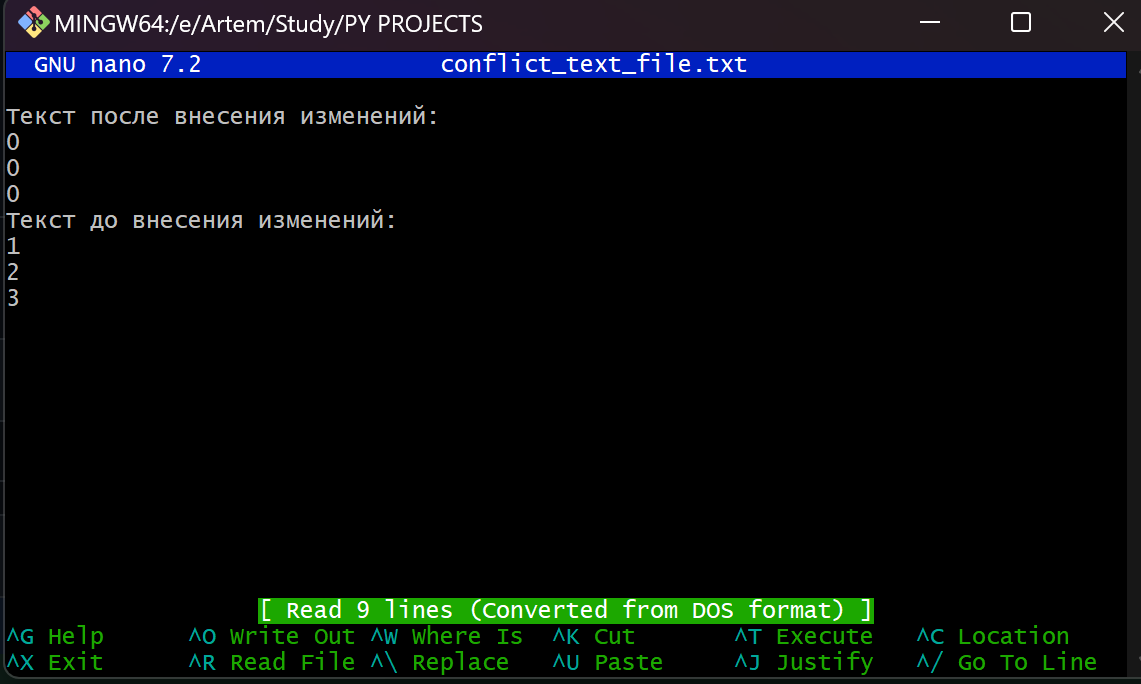
Для создания ситуации, когда на моем репозитории есть конфликты я создал ветку с помощью команды **git checkout -b master** и переключился на неё этой же командой, добавил туда текстовый файл под именем **conflict\_text\_file.txt** с некоторым содержимым и добавил в коммит с помощью команды **git add .**, после чего закоммитил файлы с помощью команды **git commit -m "commit for add text file".** После этого создал ветку **conflict\_branch**, ответвленную от ветки master и внёс изменения в файл **conflict\_text\_file.txt** и при попытке подмержить ветку master в ветку conflict\_branch в командной строке появилось сообщение о конфликте и ветка приняла статус **MERGING**.

Для того, чтобы увидеть в каких строках конфликт в текстовом файле я использовал команду **nano conflict\_text\_file.txt** и увидел следующее содержимое:



Для решения конфликта нужно расположить строки с нашей ветки **conflict\_branch**   
и с ветки **master** так, чтобы они оказались в разных строках, а не на одних и тех же, как это выглядит сейчас, из-за чего и происходит конфликт.

Решил конфликт я следующим образом:



Удалив лишние строки с обозначениями стрелками в каком месте изменения с ветки master, а в каком c локальной ветки и расположив содержимое так, как показано на скриншоте выше я выше из редактора с сохранением изменений и добавил файл в список измененных, после чего закоммитил изменения в этом файле. После указанных действий статус **MERGING** в командной строке git возле названия ветки пропал.

Для создания pull-request в ветку main я выполнил команду **git push origin conflict\_branch** и на веб-сервисе **GitHub** нажал кнопку **Create Pull-Request**, после чего **Merge Pull-Request**. После указанных выше действий в ветке main появились изменения в текстовом файле с ветки **conflict\_branch**, на которой я решил конфликт, возникший в ходе слияния ветки master с веткой **conflict\_branch**.

Такое решение конфликтов возможно лишь с текстовыми файлами, однако не обязательно для этого использовать команду **nano conflict\_text\_file.txt.** Тот же результат получится и в случае открытия текстового файла подобного или любого другого формата с помощью поддерживаемого данные форматы текстового редактора.

В случае возникновения merge конфликта с любыми нетекстовыми файлами пользователю будет предоставлен выбор между двумя версиями этих файлов (первая – с ветки **conflict\_branch**, вторая версия с ветки **master**, если проецировать ситуацию на мой проект).

Для усваивания того, что из себя представляет команда **git rebase** я создал ветку first и переключился на неё с помощью команды **git checkout -b first** и на ней выполнил ещё **git checkout -b second.** Далее переключился снова не ветку first командой **git checkout first** и добавил на этой ветке текстовый файл first.txt, после чего выполнил команду **git add .** и **git commit -m “commit 2”.** После этого переключился на ветку second и выполнил команду **git rebase first**, благодаря чему коммит с ветки first и добавленный в нем файл first.txt были добавлены в ветку second.

Чтобы посмотреть коммиты я выполнил команду **git log** и вот что увидел в командной строке git:

