**Лабораторная работа №1.**

Тема: **Настройка работы системы контроля версий (типов импортируемых файлов, путей, фильтров и др. параметров импорта в репозиторий)**

Цель работы: освоить навыки работы с системой контроля GIT

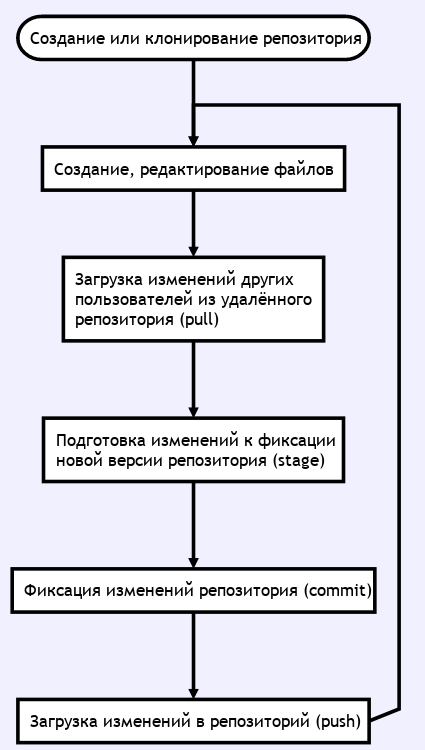
1. **Необходимые теоретические сведения**

**Введение Git**

Git — распределённая система контроля версий: система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

**Схема работы с git**



**Основные команды git**

Создание пустого git репозитория в папке <dir>. Если <dir> не задана — пустой репозиторий инициализируется в текущей папке (<dir> = .). Все базы данных для работы git создаются в «скрытой» папке <dir>/.git. Такой репозиторий хранит локальную историю версий рабочей директории.

git init <dir>

Создание простого пустого git репозитория в папке <dir>. Если <dir> не задана — пустой репозиторий инициализируется в текущей папке (<dir> = .). При таком варианте создания репозитория, папка <dir> содержит исключительно историю версий. Такой вариант создания репозитория необходимо использовать для совместного использования созданного репозитория. **Репозиторий не содержит «рабочей» версии, позволяющей работать с проектом непосредственного в рабочей папке.**

git init --bare <dir>

Сам по себе git работает полностью локально, без выхода в Интернет. При этом вокруг git появилась целая экосистема сервисов: облачные хранилища репозиториев (с поддержкой git-команд прямо в браузере), системы автоматической сборки (проекта, документации к нему и т.д.), автоматического поиска уязвимостей в коде (например, [Snik.io](https://snyk.io/)) и многие другие. Поскольку git работает полностью локально, работа с ним обычно строится следующим образом: сначала код выгружается из какого-либо облачного хранилища, затем локально (на компьютере) делаются изменения, делается коммит этих изменений (о нем ниже, пока "коммит" стоит понимать как "сохранение") и в отправляется обратно в облачное хранилище. Выгрузить репозиторий с облака можно командой git clone. Она создаёт копию git репозитория <rep> в папке <dir> и автоматически настраивать git так, чтобы он был в курсе, откуда был скачан репозиторий. Если имя папки <dir> не задано — имя папки выбирается автоматически на основании <rep>.

git clone <rep> <dir>

**Примеры использования:**

Клонирование локального репозитория:

git clone /path/to/repository/directory

Клонирование удалённого интернет репозитория:

git clone https://github.com/path/to/repository

Клонирование удалённого репозитория по протоколу ssh (не будет просить пароль и соединение будет более безопасным):

git clone ssh://<username>@<address>/path/to/repository/directory

Почти во всех облачных хранилищах (далее будем их иногда называть "удаленное хранилище", remote) есть разделение прав доступа, и у вас не всегда получится отправить свои изменения в чужой репозиторий. Проблема решается так: вы создаете полную копию кода в своем аккаунте (такой репозиторий называется fork), делаете в нем изменения и потом предлагаете свои изменения оригинальному владельцу.

Есть возможность получить только содержимое папки .git (обычно используется на серверных машинах):

git clone --bare <rep>

**Настройка git**

После создания (клонирования) репозитория, необходимо настроить информацию, о том, кто будет работать с ним работать. Для этого необходимо сообщить информацию об имени и почте пользователя. Данная информацию может быть сохранена локально — сохраняется только для данного репозитория и глобальной — информация сохраняется для текущего пользователя ОС (Windows, Linux, OS) и её не надо будет заново вводить для последующих репозиториев.

git config --global user.name="Тут имя" git config --global user.email="Почта"

или

git config --local user.name="Тут имя" git config --local user.email="Почта"

**Работа с репозиторием**

Отметить изменения в файле <filename> — изменения, сделанные в файле, подготавливаются для их фиксации как отдельной версии в репозитории.

git add <filename>

Отметить изменения во всех доступных файлах репозитория:

git add \*

Отменить действие команды add для <filename>

git reset -- <filename>

Зафиксировать все подготовленные, с использованием git add, изменения

git commit -m "commit message"

Посмотреть текущее состояние репозитория

git status

Откатить все изменения, сделанные после фиксации (commit)

git checkout -- .

Коммит - это "снимок" вашего кода в данный момент времени. Когда вы сделали коммит, вы сможете вернуться к нему в любой момент - и получить свой проект в том состоянии, в котором вы сделали этот коммит.

**Взаимодействие с внешним репозиторием**

Загрузить из внешнего репозитория все изменения. **Возможно потребуется разрешение конфликтов, если файл был изменён различными пользователями**

git pull

Выгрузить, зафиксированные на вашем компьютере, версии (коммиты) в удалённом репозиторий. При этом git должен знать, откуда вы скачиваете изменения. Если Вы не выполнили операцию pull, системы выдаст соответствующую ошибку.

git push

**Ветки GIT**

Под капотом git выстраивает связи между коммитами: когда вы делаете новый коммит, он по факту становится после старого. Это образует граф. Например, если были коммиты A и B в порядке: A -> B, то после очередного коммита состояние станет таким: A -> B -> C. При этом можно в любой момент вернуться на коммит B, "отпочковаться" (т.е. решить от нее пойти в другом направлении, а не в C) и делать новые изменения. Этот процесс называется ветвлением.

Git позволяет создавать ветки и проводить разработку проекта в них. При этом получится писать код независимо от других пользователей. Обычно при этом код из ветки перетекает в master (сейчас много где переименован в main), но считается хорошим тоном это делать только после тщательной проверки и доработки всех изменений внутри отдельно созданной ветки.

Чтобы создать ветку <branch\_name> необходимо выполнить команду.

git branch <branch\_name>

В случае, если <branch\_name> не задан, будет выведен список существующих веток. **Создав новую ветку, вы остаётесь в старой.**

Переключение на ветку <branch\_name> осуществляется по команде

git checkout <branch\_name>

Создать и сразу переключиться на новую ветку <branch\_name> можно по команде

git checkout -b <branch\_name>

Для того, чтобы объединить ветки (например, присоединить <branch\_name> в главную ветку master). Необходимо переключиться на основную ветку (master), и присоединить к ней (merge) другую ветку (<branch\_name>). В примере ниже, создаётся отдельная ветка hotfix\_123. После решения проблемы фиксируются все изменения в данной ветке, после чего происходит присоединение ветки с hotfix-ом в основную ветку (master)

git checkout -b hotfix\_123 ... git add \* git commimt -m "HOTFIX for #ISSUE-123" git checkout master git merge hotfix

**.gitignore**

Для того, чтобы быстро зафиксировать все сделанные изменения, удобно использовать команду add \*, но такая команда может поместить в репозиторий те файлы, изменение которых Вы не хотите отслеживать (например, служебные файлы питона .pyc). Информацию о таковых удобно поместить в файл .gitignore (**имя файла начинается с точки**), тогда add \* не будет «подхватывать» новые файлы из заданных папок (с заданным именем, расширением, пр.).

**GITHUB**

Крупнейшим веб-сервисом для хостинга IT-проектов и их совместной разработки на основании технологии git, является сайт [Github](https://github.com/). Он целиком построен на git и поддерживает все перечисленные команды, дополняя их различными возможностями. К дополнительным возможностям относятся: Issues — назначение заданий пользователям с описанием задачи, которую необходимо решить; Pull Request — не являясь владельцем репозитория и не имея возможности напрямую отправлять изменения в репозиторий (или же просто в основную master ветку), у пользователя github есть возможность запросить осуществление команды merge <ваша\_репозиторий/ваша\_ветка> <целевая\_ветка\_репозитория> у привилегированного пользователя, имеющего полный доступ к репозиторию. В случае положительного ответа, предложенные Вами изменения «вольются» в основной проект.

А теперь подробнее.

Issues - это способ общения между пользователями кода и разработчиками. Очень похож на форум: люди создают топики (они и называются issue), оставляют комментариями, могут закрывать и открывать обратно issue. Основная цель - репортить ошибки в коде (собственно, из-за этого issues и получили свое называние), обсуждать пути решения, назначать исполнителей (assignee), помечать тегами. Помимо этого в Issues еще можно предлать новую функциональность, обсуждать направления разработки и просто приводить примеры оформления других issue. Issue может открыть любой пользователь.

Pull request - предложение влить свой код в какую-то ветку. Это похоже на git merge, который был описан выше, но есть дополнения:

1. Pull request (PR) можно открыть как в пределах репозитория (одну ветку влить в другую), так и из форка в основной (например, влить ветку dev из своего форка в ветку dev оригинального репо).
2. Pull request не сразу же вливает код. Это делает вручную человек с нужными полномочиями. На PR можно также навесить защиты: запретить принимать PR, пока не пройдут тесты или его не одобрят как минимум N человек (настраиваемо).
3. Pull requsts можно и нужно обсуждать. Люди могут смотреть все предлагаемые изменения **построчно** и построчно же комментировать их. Это очень удобно на code review. Комментарии можно также оставлять ко всему PR в целом.
4. Pull requests можно связывать с Issue, при этом issue автоматически закрывается (помечается как "Closed") после принятия PR.
5. **Индивидуальные задания**

**ЧАСТЬ А**

1. Зарегистрироваться в github, если Вы ещё не зарегистрированы
2. Разбиться на группы по два человека для выполнения семестрового проекта
3. Один человек из каждой группы создаёт приватный репозиторий для проекта на java (см. рисунок ниже)
4. Владелец репозитория даёт доступ к нему второму студенту и преподавателю
5. Каждый студент создаёт свою ветку, в которой ведёт дальнейшую разработку своей части проекта до конца семестра, отправляя в master рабочие версии файлов.

**ЧАСТЬ Б**

1. Создать три файла: 1.txt, 2.txt, 3.txt.
2. Проиндексировать первый файл и сделать коммит с комментарием “add
3. 1.txt file”.
4. Проиндексировать второй и третий файлы.
5. Удалить из индекса второй файл.
6. Перезаписать уже сделанный коммит с новым комментарием “add 1.txt
7. and 3.txt”
8. Создать аннотированную метку с названием v0.01.
9. Создать легковесную ветку указывающую на первый коммит в репозитории.

**ЧАСТЬ С**

1. Создать папку temp в своем репозитории.
2. Создать папку log и добавить в нее 2 файла: main.html и some.tmp.
3. Создать файл .gitignore и добавить в игнорирование папку temp и файлы
4. с расширением .tmp из папки log.
5. Закоммитить добавление файла .gitignore.
6. Внести изменения в файл my\_first\_file.txt, добавив строчку “row to
7. index”, проиндексировать данные изменения. Еще раз внести изменения в файл, добавив строчку “row no index”.
8. Посмотреть индексированные и неиндексированные изменения используя команду git diff.
9. Удалить файл my\_first\_file.txt, зафиксировать данное удаление.
10. Переименовать файл my\_second\_file.txt в my\_first\_file.txt, зафиксировать изменение