# Дисциплина «Проектирование информационных систем»

# Лабораторная работа № 9 Системы контроля версий. Git.

Цели работы:

1. Изучить теоретические сведения о системах контроля версий.

2. Получить практические навыки по ведению контроля версионности проекта.

3. Получить практические навыки работы с Git.

## Система контроля версий

Система контроля версий – это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета, система контроля версий — 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 системы контроля версии также значит, 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.

## Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (см. рис.1). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели. Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные систему контроля версий с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий. Одной из популярных систем контроля версий была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.


Рис.1 Локальное управление версиями

## Централизованная система контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий. Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища (см. рис.2).

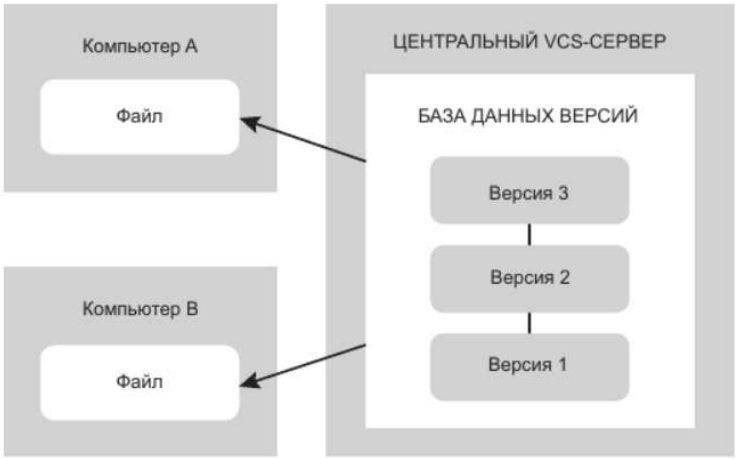



Рис.2 Централизованное управление версиями

Такой подход имеет множество преимуществ, особенно перед локальными системами контроля версий. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать централизованную систему контроля версий, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, 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, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились

на локальных машинах разработчиков. Локальные система контроля версий страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

### Распределенная система контроля версий

В распределённых системах контроля версий (таких как Git, Mercurial, Bazaar или Darcs) 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, перестанет функционировать, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных (см. рис.3).

Более того, многие распределенный системы контроля версий 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.

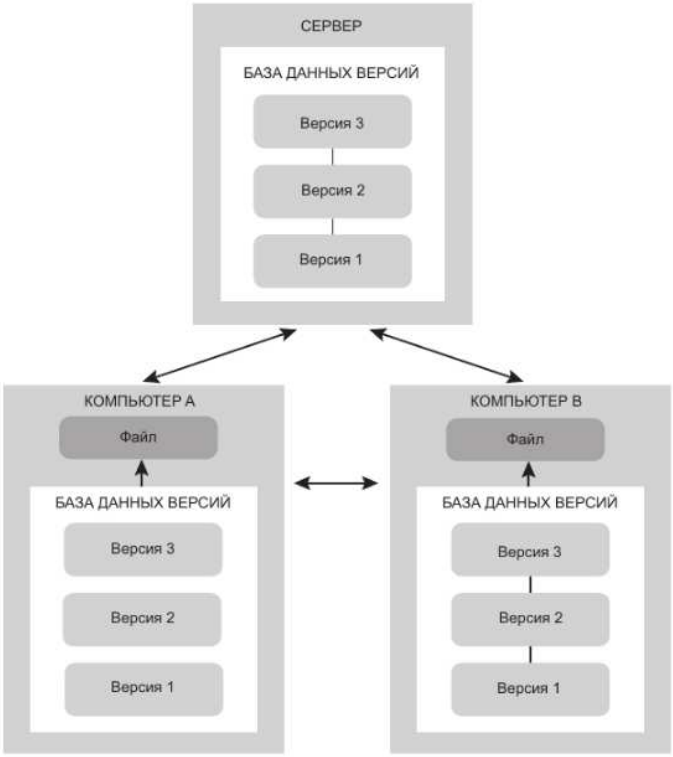



Рис.3 Распределенное управление версиями

### Git

Сегодня Git - это наиболее часто используемая система контроля версий, которая быстро становится стандартом для контроля версий. Git - это распределенная система контроля версий, что означает, что ваша локальная копия кода представляет собой полный репозиторий контроля версий. Эти полностью функциональные локальные репозитории позволяют легко работать в автономном или удаленном режиме. Вы фиксируете свою работу локально, а затем синхронизируете свою копию репозитория с копией на сервере.

Даже если вы всего лишь один разработчик, контроль версий поможет вам оставаться организованным при исправлении ошибок и разработке новых функций. Контроль версий

сохраняет историю вашей разработки, так что вы можете легко просмотреть и даже вернуться к любой версии вашего кода.

Система управления версиями (СУВ) Git позволяет контролировать изменения файлов в выбранных папках на своем компьютере и согласовывать эти изменения с изменениями файлов на компьютерах членов команды, совместно работающих над каким-либо проектом. Возможно согласование изменений с хранилищем файлов проекта (репозиторием) на выделенном сервере, в том числе на серверах общедоступных сервисов https://github.com и https://bitbucket.org.

После загрузки программного обеспечения Git с сервера https://git-scm.com и его установки на компьютере в списке установленных программ появляется папка Git с ссылками на приложения Git Bash, Git CMD и Git GUI.

Git Bash –интерпретатор командной строки bash ОС UNIX/Linux перенесенный в Windows и интегрированный с Git. Интеграция с Git позволяет интерпретатору выполнить команды Git из любой текущей папки и хранить все настройки программы для использования при последующих вызовах.

Git CMD – интерпретатор cmd командной строки Windows, также понимающий команды Git. Для выполнения команд Git в окне Git CMD необходимо командой cd сделать текущей папкой папку с контролируемыми файлами.

Git GUI – графический интерфейс пользователя программы Git. С его помощью можно выполнить основные операции Git по управлению файлами проекта в локальном и удаленном (remote) репозиториях без необходимости знания синтаксиса команд Git. Однако в некоторых случаях его возможностей оказывается недостаточно для более сложных случаев совместной работы над проектами.

После установки Git на компьютере в контекстном меню правой кнопки мыши появляются команды запуска Git Bash Here и Git GUI Here, позволяющие запустить интерпретатор команд и графическую оболочку с привязкой к открытой папке с контролируемыми файлами проекта или папке, которую нужно сделать рабочей папкой проекта.

## Настройка Git

1. Откройте папку, содержащую файлы проекта одной из ранее выполненных лабораторных работ по ООП. Из контекстного меню правой кнопки мыши выберите Git Bash Here и в открывшемся окне введите команду:

   *$ git init.*

   В результате в папке с проектом появится папка с именем «.git» ,содержащая все необходимые файлы локального репозитория.
2. Авторизуйтесь для внесения последующих изменений, указав свое имя и адрес электронной почты, например:

   *$ git config user.name "Максим Лунин"*

   *$ git config user.email max.lunin@mail.ru*

   Если вы собираетесь постоянно работать с проектами на данном компьютере, то добавьте к командам авторизации ключ global:

   *$ git config --global user.name "Сергей Васин"*

   *$ git config --global user.email serge_vasin@yandex.ru*

Команда git --help - выводит общую документацию по git.

Если введем git log --help - он предоставит нам документацию по какой-то *определенной команде (в данном случае это - log)*

3. Создайте в папке с проектом файл «.gitignore» для постоянного исключения из числа файлов с отслеживаемыми изменениями временные и автоматически создаваемые файлы компиляторов

и компоновщиков. При создании файла с именем «.gitignore» средствами контекстного меню мыши возможен отказ системы Windows сохранить файл без указания имени перед точкой. В этом случае сохраните его с произвольным именем, например, «a.gitignore», а затем в командной строке cmd Windows и Git CMD переименуйте его командой ren или в Git Bash командой mv, предварительно сделав папку с файлом текущей:

*>ren a.gitignore .gitignore (для cmd Windows, Git CMD)*

*$ mv a.gitignore .gitignore (для Git Bash)*

Создайте в файле «.gitignore» список файлов и папок, отслеживать которые не нужно, например: *obj/Win32/Debug/.

Символ «*» в указании имени заменяет произвольное количество символов, а символ «/» после имени обозначает папку.

4. Добавление имен удаленных репозиториев. При совместной работе над проектом нескольких разработчиков вначале необходимо получить файлы текущей версии проекта из удаленного репозитория. Для этого необходимо знать краткое имя репозитория на удаленном сервере или его URL-адрес. Знания краткого имени будет достаточно, если файлы в рабочую папку уже копировались с сервера командой git clone. Пример команды git clone:

*$ git clone https://github.com/имя_владельца/имя_папки_проекта.*

При этом по умолчанию удаленному репозиторию присваивается имя «origin». Проверить присвоенные имена серверов можно командой git remote:

*$ git remote –v .*

Ключ –v позволяет вывести полные URL-адреса серверов:

*origin https://github.com/имя_владельца/имя_папки (fetch)*

*origin https://github.com/имя_владельца/имя_папки (push)*

В скобках указаны команды скачивания (fetch) и отправки изменений (push) допустимые для этого сервера. Добавить еще одно имя сервера с удаленным репозиторием можно командой git remote add:

*$ git remote add краткое_имя_сервера URL-репозитория.*

Добавление имени сервера имеет смысл только после регистрации своего аккаунта на соответствующем сайте или после получения URL-адреса от руководителя разработки проекта. Команда git clone настраивает ваш локальный репозиторий на слежение за удаленным репозиторием. Скачать все изменения файлов удаленного репозитория с момента последней сверки можно командой git pull:

*$ git pull краткое_имя_сервера.*

**Создание собственного репозитория на сайте github.com**

Для создания собственного удаленного репозитория на веб-сервисе Microsoft github.com необходимо:

- Зайти нас сайт github.com и зарегистрироваться, указав имя (логин), e-mail и пароль для входа.
- Подтвердить регистрацию по ссылке в письме, пришедшем на указанный при регистрации e-mail адрес.
- Войти в свой аккаунт на github.com и щелчком на кнопке «New» создать новый репозиторий, указав его имя, например, MyRepo.
- В командном окне Git Bash задать краткое имя, обозначающее удаленный репозиторий на github и его URL, например:

*$ git remote add КраткоеИмя http://github.com/Логин/MyRepo*

(В качестве краткого имени по умолчанию выбирается имя origin).

- Скопировать содержимое ранее созданного локального репозитория в удаленный репозитория командой push с указанием имени ветки (master)

*$ git push КраткоеИмя -u master*

Ключ -u сохранит связь с удаленным репозиторием позволит в дальнейшем отслеживать изменения и копировать измененные и добавленные файлы командами git pull и git push без указания имени репозитория и имени ветки. Для доступа к удаленному репозиторию по запросу сервера придется ввести ваш логин и пароль. Следите за регистром ввода пароля!

- Внесите изменения или добавьте файл в удаленный репозиторий непосредственно на сервере github.com. Затем скопируйте эти изменения командой git pull в папку локального репозитория.
- Удалить репозиторий на github можно на его странице из меню "Settings" (значок «шестеренка») в блоке «Danger Zone» нажатием на кнопку «Delete this repository». При этом потребуется ввести имя репозитория и пароль владельца.

**Основные команды Git**

5. Проверка состояния рабочей папки осуществляется командой

*$ git status.*

В ответ Git Bash покажет название текущей ветви «master», присвоенное по умолчанию, и список всех неотслеживаемых файлов в рабочей папке, не подпадающих под шаблоны файла «.gitignore». Имена этих файлов будут выделены красным шрифтом.

6. Включение файлов и папок в перечень отслеживаемых выполняется командой git add:

*$ git add имена_файлов_или_шаблоны_имен*.

Например, команда

*$ git add *cpp *dfm *h.*

включает в перечень отслеживаемых файлы исходных текстов на языке C++ (*cpp), описание графической формы (*dfm) и заголовочные файлы (*h).

7. Внесите изменения в один из текстовых файлов, например, добавив комментарий к какому-либо оператору исходного текста программы на языке C++, и сохраните измененный файл. Снова выполните команду git status. Перечень отслеживаемых файлов рабочей папки, не затронутых изменениями и подготовленных к фиксации их состояния командой commit будет выведен зеленым шрифтом. Имя измененного файла и файлов, не включенных в отслеживаемые, отобразится красным шрифтом.

8. Добавьте измененный файл к файлам, подготовленным к фиксации состояния (staged files), с помощью команды

*$ git add имя_файла*.

Проверьте состояние файлов командой git status и сохраните снимок текущего состояния отслеживаемых файлов командой git commit:

*$ git commit -m 'Ver 1.0'.*

Ключ «-m» позволяет сохранить сообщение (message) 'Ver 1.0' как комментарий, описывающий данную фиксацию (снимок состояния). Сообщение должно быть простым текстом на любом языке, заключенным в апострофы или двойные кавычки. В данном примере предлагается указать, что этот снимок состояния является первой версией файлов проекта. Любая фиксация требует обязательного указания строки сообщения.

Ввод команды git commit без указания сообщения запускает текстовый редактор, которым по умолчанию является стандартный редактор текста ОС UNIX/Linux Vim. Редактор Vim всегда запускается в командном режиме. Чтобы перевести его в режим вставки, то есть в режим ввода и изменения текста, необходимо ввести с клавиатуры символы «i» или «a». В первой пустой строке открывшего текста-комментария нужно ввести сообщение, описывающее данную фиксацию, а затем перейти в командный режим нажатием клавиши <Esc>. Выход из редактора с сохранением изменений выполняется командой:

*:wq* или **ZZ**.

При этом будет программа git выполнит фиксацию состояния файлов проекта с сохранением в качестве комментария строки введенного сообщения.

9. Просмотрите историю изменений с помощью команды git log:

*$ git log или $ git log --oneline*.

Ключ «--oneline» существенно сокращает объем выводимой информации. Ключ «-p» позволяет вывести различие между каждым коммитом, а ключ «-число» – ограничивает вывод количеством коммитов, заданных числом. Например, команда

*$ git log -p -3*

выведет информацию о трех последних коммитах с указанием различий в отслеживаемых файлах.

10. Создание новой ветки

Необходимость создания новой ветки в разработке проекта возникает, если предполагаются существенные отклонения от первоначального замысла или разработчикам требуется одобрение результатов работы руководителем проекта. Создайте ветку testing командой:

*$ git branch testing*.

Узнать, какая ветка является текущей можно, введя команду git log:

*$ git log или $ git log --oneline.*

В первой строке будет выведена информация о наличии двух веток

*1bdf852 (HEAD -> master, testing) текст_сообщения_commit*,

на которые указывает указатель HEAD.

Переход на новую ветку выполняется командой git checkout.

*$ git checkout testing*.

Повторный ввод команды git log --oneline покажет, что переход произошел:

*1bdf852 (HEAD -> testing) текст_сообщения_commit*.

11. Внесите изменение в один-два отслеживаемых текстовых файла новой ветки и зафиксируйте изменения командой:

*$ git commit -a -m "текст_комментария_commit"*

Ключ «-a» в команде git commit позволяет пропустить команду git add, добавляющую измененные файлы в область индексирования, подготавливая их к фиксации.

Просмотрите историю изменений ветки *testing* командой *git --oneline*.

12. Вернитесь на основную ветку master командой:

*$ git checkout master*

Внесите изменения в один-два отслеживаемых текстовых файла и зафиксируете изменения командой

*$ git commit -a -m "текст_комментария"*

13. Слияние веток. Выполните слияние ветки testing с веткой master вводом команды:

*$ git merge testing*

Если вы внесли изменения в один и тот же файл в разных ветках, и они оказались противоречивыми, то автоматическое слияние не удастся. Git Bash проинформирует о конфликте слияния сообщением:

*CONFLICT (content): Merge conflict in Text1.txt*

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

В текст конфликтных файлов будут внесены подсказки в местах противоречивых расхождений вида:

*Строка 3*

*<<<<<<< HEAD*

*Строка 4*

*=======*

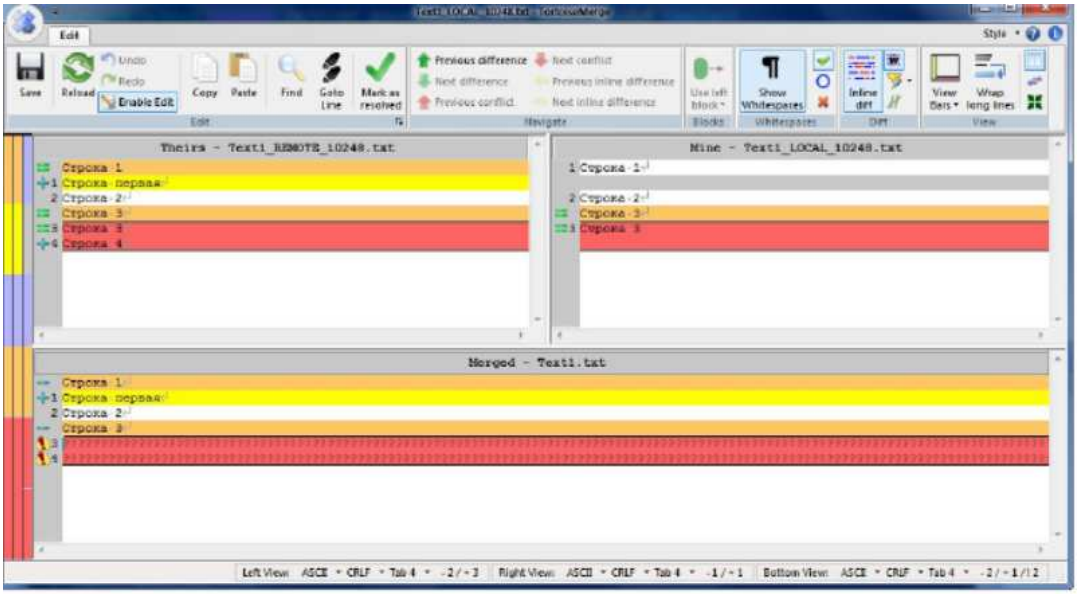
*>>>>>>> testing*

Конфликтный файл можно открыть в текстовом редакторе и внести необходимые изменения, удалив строки:
*<<<<<<< HEAD*
*>>>>>>> testing*
Затем необходимо зафиксировать изменения командой git commit и повторно ввести команду слияния *git merge testing*.

Изменения в конфликтующие файлы удобно вносить визуальным средством просмотра и редактирования различий, вызываемым командой:

*$ git mergetool*

Если выбор конкретного редактора различий не задан, то git предложит использовать программу *tortoisemerge* с чем целесообразно согласиться. Как показано на рисунке 1, окно программы *tortoisemerge* разделено на три области: текст файла в сливаемой ветке (Theirs), текст в активной ветке (Mine) и область слияния (Merged). Необходимые изменения вносятся в нижней части окна – в области слияния. Затем они сохраняются (Save) и отмечаются как разрешенный конфликт (Mark as resolved). После этого окно программы *tortoisemerge* можно закрыть.




14. Просмотр истории изменений проекта. Просмотрите историю изменений командой:

*$ git log --graph --oneline*

Ключ «--graph» позволяет графически показать историю разделения и слияния веток *master* и *testing*:

** ae208a3 (HEAD -> master) Merge branch 'testing'*
*|*
*| * 44a04e1 (testing) Исправлен Text1.txt*
*| * 5033ec5 Исправлен Text2.txt*
*|/*
** 5b12bf1 (origin/master) Добавлен Text2.txt*
** 4789148 Добавлен Test1.txt*

Посмотрите результат объединения исправленных файлов в текстовом редакторе (блокноте).

**Замечание.** Так как выход в Интернет в лабораториях университета осуществляется через прокси-сервер, выгрузить файлы проекта на удаленный сервер и загрузить с него будет проблематично. Поэтому вся практическая часть последовательно выполняется дома, для сдачи лабораторной работы предоставляется отчет с комментариями и скриншотами действий в командной строке.

Подробное описание системы контроля версий Git можно найти здесь:

1. Чакон С., Штрауб Б. Git для профессионального программиста. – СПб.: Питер, 2016. – 496 с.
2. Git за полчаса: руководство для начинающих. URL: https://proglib.io/p/git-for-half-an-hour

**Контрольные вопросы:**

1. Что такое система контроля версий?
2. Какие системы контроля версий бывают?
3. Что такое локальная система контроля версий? Какие у нее недостатки?
4. Что такое централизованная система контроля версий? Какие у нее недостатки?
5. Что такое распределенная система контроля версий?
6. Опишите процесс осуществления контроля версий.
7. Что такое коммит? Как можно просмотреть историю коммитов?
8. Что такое удаленный репозиторий? Как к нему подключиться? Как осуществляется отправка изменений на сервер?
9. Что такое ветвление? Каким образом оно осуществляется в Git? Что означает слияние веток?
10. Что такое конфликты? Когда они возникают?
11. Как удалять ветки в Git?
12. Для чего используется параметр –amend команды commit?
13. Позволяет ли Git вернуть файл к предыдущему состоянию?
14. Как можно отследить изменения, сделанные в коммитах?