

Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования
Кафедра инженерной психологии и эргономики
Учебная дисциплина «Пользовательские интерфейсы
информационных систем»

Отчет
«Основы Git»

Выполнила: Бакун А.А.

гр. 210901

Проверил: Давыдович К. И.

Минск 2024

Часть I – Основы *Git*.

Цели курса:

- Понять, что такое *Git* и его преимущества.
- Изучить основы работы с *Git* на протяжении 30 уроков.
- Ознакомиться с концепцией системы контроля версий и её применением в разработке программного обеспечения.

Git – это система контроля версий, позволяющая отслеживать изменения в коде с течением времени.

Представление *Git* как «машины времени» для кода:

- Позволяет вернуться к предыдущим версиям кода.
- Отслеживает, кто и когда вносил изменения.
- Предоставляет возможность отменять изменения.

Git использует текстовый интерфейс, требующий работы в командной строке.

Преимущества работы с командной строкой:

- Глубокое понимание работы *Git*.
- Возможность использования *Git* на любых компьютерах и серверах.

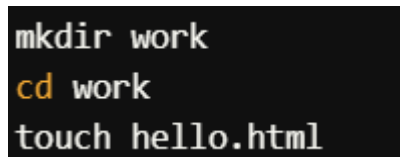
Репозиторий – хранилище проекта и его истории (локальное или удаленное). При инициализации репозитория создается скрытая директория «.git», содержащая всю информацию о репозитории.

Коммит – снимок проекта в определенный момент времени. Содержит информацию об изменениях с момента последнего коммита. Формирует цепочку истории изменений проекта.

Ветка – параллельная версия репозитория для работы над отдельными функциями. Позволяет разработчикам работать независимо от основной версии проекта. В репозитории всегда присутствует хотя бы одна ветка, обычно называемая «*main*» или «*master*».

Создание страницы «Hello, World».

Сначала была создана пустая поддиректория «*work*» в директории «*repositories*» (рис. 1).



```
mkdir work
cd work
touch hello.html
```

Рисунок 1 – пустая директория

Содержимое файла «*hello.html*»: «*Hello, World*».

После создания файла был инициализирован *Git*-репозиторий с помощью команды: «*git init*».

Файл «*hello.html*» был добавлен в репозиторий с помощью команд: *git add hello.html*; *git commit -m "Initial Commit"* (рис. 2).

```
$ git add hello.html
$ git commit -m "Initial commit"
[main (root-commit) 5836970] Initial commit
1 file changed, 1 insertion(+)
create mode 100644 hello.html
```

Рисунок 2 – результаты выполнения

Проверка состояния.

Сообщение «*nothing to commit*», «*working tree clean*» означает, что текущее состояние рабочих файлов уже сохранено в репозитории, и нет изменений, которые требуют коммита.

Команда «*git status*» будет использоваться для постоянного отслеживания состояния репозитория и рабочей директории в дальнейшем.

Внесение изменений.

Успешно освоен процесс внесения изменений в файл и отслеживание состояния рабочей директории с помощью команды «*git status*». Это ключевой навык для управления версиями и контроля изменений в проектах. В дальнейшем планируется добавление изменений в репозиторий с помощью команды «*git add*».

Отмена изменений.

Знание методов отката изменений, как до индексации, так и после коммита, стало важным элементом моего обучения. Я научилась использовать команды «*git restore*» и «*git reset*», что позволяет мне безопасно отменять нежелательные изменения и сохранять чистоту рабочего каталога.

История.

Способность просматривать историю изменений в проекте является критически важной для анализа и отслеживания его прогресса. В процессе обучения я освоила использование команды «*git log*», которая предоставляет полное представление о всех предыдущих коммитах в репозитории (рис. 3).

Команда «*git log*» позволяет увидеть список всех коммитов, сделанных в проекте. Каждый коммит отображается с уникальным хешем, датой, автором и сообщением коммита. Это дает возможность быстро оценить, какие изменения были внесены, и кем они были сделаны.

Я также научилась использовать различные параметры команды «*git log*», которые позволяют фильтровать и форматировать вывод. Например, использование «*git log --oneline*» показывает краткий обзор коммитов в одной строке, что делает его более удобным для быстрого анализа.

Параметр «*--grep*» позволяет искать коммиты по содержимому сообщений. Это полезно, когда необходимо быстро найти конкретные изменения или исправления, связанные с определенной задачей.

Знание о том, как просматривать историю изменений, помогает мне возвращаться к предыдущим версиям проекта при необходимости. Например, если я обнаруживаю, что недавние изменения привели к ошибкам,

я могу использовать хеш коммита для возврата к стабильной версии, используя команду «git checkout».

Я также научилась использовать команды «git diff» вместе с «git log», чтобы сравнивать изменения между разными коммитами. Это позволяет мне видеть, какие именно строки были добавлены или удалены в каждом коммите, что облегчает анализ изменений.

Понимание истории изменений и умение работать с «git log» значительно повысили мою способность анализировать разработку проекта.

Это знание позволяет мне:

- Легко отслеживать прогресс работы и видеть, как проект менялся со временем.
- Быстро идентифицировать и исправлять ошибки, возвращаясь к предыдущим стабильным версиям.

Индексация изменений и коммит, коммит изменений.

Успешно освоена процедура индексации изменений с помощью команды «git add» (рис. 3). Это важный шаг перед выполнением коммита, который позволяет подготовить изменения для сохранения в репозитории. В дальнейшем планируется завершить процесс коммита для фиксации изменений.

```
$ git add hello.html
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
       modified:   hello.html
```

Рисунок 3 – результат выполнения

Индексация в *Git* позволяет разделять большие изменения на маленькие и логически связанные коммиты. Это важно для поддержания чистоты истории изменений и облегчения понимания их сути.

Представьте, что вы помыли машину и одновременно залили жидкость для очистки стекла. Эти два изменения независимы, и лучше зафиксировать их отдельно. Если объединить их в один коммит, то в истории будет запись «Помыл машину», что не отражает сути изменений.

Разделяя индексацию и коммиты, вы получаете возможность точно настроить, что будет включено в каждый коммит. Это позволяет создавать более понятные и логически обоснованные записи в истории изменений, что упрощает работу с проектом в будущем.

Для завершения процесса и фиксации проиндексированных изменений в репозитории была выполнена команда: «git commit».

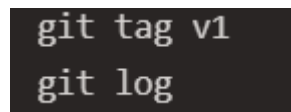
Сообщение «nothing to commit», «working tree clean» подтверждает, что все изменения успешно закоммичены, и рабочая директория чиста.

Создание тегов версии.

В процессе обучения я освоила создание тегов в *Git*, что стало важным шагом в управлении версиями и поддержании порядка в проекте. Теги позволяют отмечать значимые достижения и версии кода, делая процесс релиза гораздо более организованным и понятным.

Теги в *Git* представляют собой специальные метки, которые прикрепляются к определенным коммитам. Они служат для обозначения важных вех в развитии проекта, таких как релизы, стабильные версии или ключевые изменения. В отличие от веток, теги не изменяются и остаются привязанными к конкретному коммиту.

Я научилась создавать теги с помощью команды «*git tag*» (рис. 4).



```
git tag v1
git log
```

Рисунок 4 – выполнение команды для создания тегов первой версии

Существует два основных типа тегов:

- Легкие теги (*lightweight tags*) – это просто указатели на коммит, которые не содержат дополнительной информации.
- Аннотированные теги (*annotated tags*) – это полноценные объекты в *Git*, которые содержат имя автора, дату, сообщение и могут быть подписаны. Они являются предпочтительными для использования в большинстве случаев, так как предоставляют больше информации.

Создание тегов делает процесс релиза более организованным. Например, при завершении работы над новой функцией или исправлением критической ошибки я могу создать тег для обозначения этого изменения. Это делает процесс релиза более прозрачным.

Зная о тегах, я могу быстро возвращаться к предыдущим версиям проекта. Например, если новая версия оказалась нестабильной, можно легко переключиться на стабильную версию с помощью команды «*git checkout*». Это позволяет мне быстро реагировать на проблемы и поддерживать стабильность проекта.

Создание ветки.

Ветки в *Git* представляют собой независимые линии разработки, которые позволяют работать над различными задачами одновременно. Это помогает избежать конфликтов и сохраняет основную версию проекта в стабильном состоянии (рис. 5). Каждая ветка может содержать свои изменения и коммиты, которые не влияют на другие ветки, пока они не будут объединены.

```
$ git switch -c style
Switched to a new branch 'style'
$ git status
On branch style
nothing to commit, working tree clean
```

Рисунок 5 – результат создание ветки

Я научилась создавать новые ветки с помощью команды «*git branch*». Это позволяет мне отделить работу над новой функцией или исправлением от основной версии проекта (обычно ветки называются «*main*» или «*master*»).

Для того чтобы начать работу в новой ветке, я использую команду «*git checkout*». Это позволяет мне переключаться между ветками и работать над различными функциями или исправлениями без риска повредить основную версию проекта. Также я освоила использование команды «*git switch*», которая делает процесс переключения более интуитивным.

Пример команды для переключения:

git checkout new-feature

или

git switch new-feature

После создания и переключения на новую ветку я могу вносить изменения, добавлять новые функции или исправлять ошибки, не беспокоясь о том, что это повлияет на основную ветку.

Когда работа над функцией завершена, я могу объединить изменения из своей ветки в основную с помощью команды «*git merge*». Это позволяет мне интегрировать все изменения, которые были сделаны, и поддерживать проект в актуальном состоянии.

После завершения работы и слияния ветки, я могу удалить ее, чтобы поддерживать порядок в репозитории. Удаление веток, которые больше не используются, помогает избежать путаницы и упрощает управление проектом.

Пример команды для удаления ветки:

git branch -d new-feature

Слияние и разрешение конфликтов.

Слияние – это процесс объединения изменений из одной ветки в другую. Обычно это делается для интеграции новых функций или исправлений в основную ветку. Я использовала команду «*git merge*», которая объединяет изменения из одной ветки в текущую ветку.

Пример команды для слияния:

git checkout main

git merge feature-branch

При выполнении этой команды изменения из ветки «*feature-branch*» будут интегрированы в ветку «*main*».

Типы слияния:

- *Fast-forward merge*: Если целевая ветка (например, «*main*») не имеет новых коммитов с момента создания объединяемой ветки, *Git* просто перемещает указатель ветки на новый коммит. Это происходит без создания нового коммита слияния.

- *Three-way merge*: Если в целевой ветке есть новые коммиты, *Git* создает новый коммит слияния, который объединяет изменения из обеих веток. Это позволяет сохранить историю изменений.

Конфликты возникают, когда изменения в разных ветках затрагивают одну и ту же часть кода. *Git* не может автоматически определить, какое изменение следует сохранить, и требует от разработчика принять решение.

Когда я выполняю слияние, и возникают конфликты, *Git* сообщает об этом в терминале. Он также помечает конфликтующие файлы, добавляя специальные маркеры. Я научилась выявлять и анализировать эти конфликты, чтобы понять, какие изменения нужно сохранить.

Методы разрешения конфликтов:

- Редактирование файлов;
- Использование инструментов слияния.

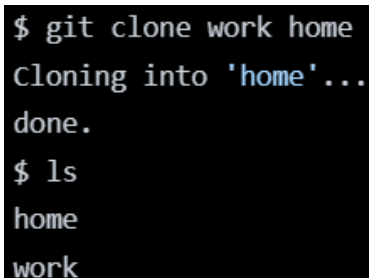
После разрешения конфликтов я сохраняю изменения и использую команду «*git add*», чтобы подготовить исправленные файлы к коммиту. Затем я завершаю слияние с помощью команды «*git commit*».

Я также изучила команду «*git rebase*», которая позволяет переносить изменения из одной ветки на другую. Это полезно для обновления ветки с последними изменениями из основной ветки без создания дополнительных коммитов слияния. Однако при использовании «*rebase*» также могут возникать конфликты, которые нужно будет разрешать аналогично процессу слияния.

Часть II – Несколько репозиториев.

Клонирование репозиториев.

Клонирование репозиториев – это один из базовых и важных шагов в работе с *Git*, который позволяет разработчикам создавать локальные копии удаленных репозиториев (рис. 6).



```
$ git clone work home
Cloning into 'home'...
done.
$ ls
home
work
```

Рисунок 6 – создан клон репозитория «*work*»

Клонирование в Git – это процесс создания полной копии удаленного репозитория на локальной машине. В результате клонирования вы получаете доступ ко всем файлам, коммитам, веткам и истории изменений, что позволяет вам работать с проектом так, как если бы он был локальным.

Для клонирования репозитория используется команда «git clone», за которой следует *URL* удаленного репозитория. Это может быть ссылка на репозиторий на платформах, таких как *GitHub*, *GitLab* или *Bitbucket*.

Пример команды: `git clone https://github.com/username/repository.git`

После выполнения этой команды *Git* создаст директорию с именем репозитория и скопирует в нее все файлы и историю коммитов.

При клонировании *Git* автоматически устанавливает «*origin*» как имя для удаленного репозитория. Это позволяет легко взаимодействовать с удаленным репозиторием, используя команды «*git push*», «*git pull*» и «*git fetch*».

Что такое origin?

В *Git* термин «*origin*» обозначает стандартное имя для удаленного репозитория, с которого вы клонируете проект. Это понятие играет ключевую роль в работе с удаленными репозиториями, поскольку оно упрощает взаимодействие с ними и делает управление изменениями более удобным.

Удаленные ветки.

Удаленные ветки представляют собой ветки, которые существуют в удаленном репозитории. Они позволяют разработчикам видеть, какие ветки доступны в удаленном репозитории и какие изменения были внесены другими участниками. Удаленные ветки обычно имеют приставку «*origin/*», если репозиторий был клонирован с использованием стандартного имени «*origin*».

Пример, если в удаленном репозитории существует ветка «*feature-x*», то в вашем локальном репозитории она будет отображаться как «*origin/feature-x*».

Подтягивание изменений.

Подтягивание изменений – это процесс в *Git*, который позволяет разработчикам получать последние обновления из удаленного репозитория. Умение использовать команду «*git pull*» имеет критическое значение для синхронизации работы с командой и поддержания актуальности локальной версии проекта.

Команда «*git pull*» объединяет две операции: «*git fetch*» и «*git merge*». Сначала она загружает изменения из удаленного репозитория (с помощью «*fetch*»), а затем автоматически объединяет их с вашей текущей локальной веткой (с помощью «*merge*»).

Слияние подтянутых изменений.

Слияние (merge) – это процесс объединения двух различных веток в одну. Когда вы используете команду «*git pull*», *Git* автоматически выполняет слияние изменений из удалённой ветки в вашу текущую локальную ветку.

Когда выполняется команда «*git pull origin main*», *Git* сначала загружает изменения из удаленного репозитория (в данном случае из ветки «*main*» удаленного репозитория «*origin*»), а затем пытается объединить их с вашей текущей веткой. Если в удаленной ветке есть изменения, которые не присутствуют в вашей локальной ветке, *Git* автоматически создает новый коммит слияния, который объединяет изменения.

После добавления всех разрешенных файлов надо выполнить команду «*git commit*», чтобы завершить процесс слияния. *Git* создаст коммит слияния, который объединит изменения.

Отправка изменений.

Отправка изменений в *Git* – это процесс, позволяющий разработчикам отправлять свои локальные изменения обратно в удаленный репозиторий.

Команда «*git push*» используется для отправки коммитов из вашей локальной ветки в соответствующую ветку удаленного репозитория. Это позволяет другим разработчикам видеть ваши изменения и интегрировать их в свою работу.

Добавление удаленного репозитория.

Добавление удаленного репозитория в *Git* – это процесс, который позволяет вам связывать ваш локальный репозиторий с одним или несколькими удаленными репозиториями.

Удаленный репозиторий – это версия вашего кода, которая хранится на сервере, доступном через интернет или локальную сеть. Он позволяет командам работать совместно и делиться кодом.

Чтобы добавить новый удаленный репозиторий, используется команда «*git remote add*», за которой следует имя удаленного репозитория и его *URL*.

Чтобы увидеть список всех удаленных репозиториях, связанных с вашим локальным репозиторием, используется команда «*git remote -v*». Эта команда покажет имена удаленных репозиториях и их *URL*, а также типы операций (*fetch/push*).

Если нужно изменить *URL* для существующего удаленного репозитория, используется команда «*git remote set-url <имя> <новый_URL>*».

Если больше не нужен удаленный репозиторий, его можно удалить с помощью команды «*git remote remove <имя>*».

Вывод

Прохождение курса по *Git* предоставило глубокие знания и навыки, необходимые для эффективной работы.

Освоены ключевые команды и концепции, такие как создание локальных репозиториях, работа с ветками, коммитами и историями изменений. Это позволяет уверенно управлять проектами и отслеживать изменения в коде.

Изучено, как добавлять удаленные репозитории, подтягивать изменения и отправлять свои коммиты обратно. Это критически важно для совместной работы и поддержания актуальности кода.

Получены навыки разрешения конфликтов, возникающих при слиянии изменений. Умение правильно разрешать конфликты позволяет поддерживать качество кода и улучшает взаимодействие в команде.

Научились управлять удаленными ветками, что помогает следить за изменениями, внесенными другими разработчиками, и интегрировать их в свою работу.

Появилось понимание, как работать с несколькими удаленными репозиториями, обеспечивает гибкость в проектной разработке и позволяет эффективно справляться с изменениями в требованиях.

Курс предоставил не только теоретические знания, но и практические навыки, которые можно немедленно применять в реальных проектах. Уверенность в использовании *Git* и понимание его возможностей значительно повысит эффективность работы в команде и качество разрабатываемого программного обеспечения. Эти знания являются основой для дальнейшего профессионального роста в области разработки программного обеспечения.