

Системы контроля версий (VCS)

Курс «Разработка ПО систем управления»

Кафедра управления и информатики НИУ «МЭИ»

Весна 2018 г.

Проблема: управление кодом

- Написан новый код, проект перестал работать.
 - Как вернуться к прежней версии?
 - Резервные копии, в которых трудно искать и легко запутаться.
 - Как узнать, что именно изменилось (файлы и строки)?
 - Вручную («глазами»), программой `fc` (для каждой пары файлов).
- Двое пишут разные части одной программы.
 - Как проверить, что они не изменили одно и то же?
 - Как совместить изменения?
 - Вручную — сложно, долго, есть риск ошибиться.
- Код пишут несколько человек долгое время.
 - Как обмениваться версиями?
 - Флэшки, почта, «облака» — всё сложно.
 - Как уследить за тем, какие были версии и что менялось?
 - Почта и т. п., файл-журнал — разобщенно, ненадежно, трудоемко.

Решение: система контроля версий

- Написан новый код, проект перестал работать.
 - Как вернуться к прежней версии?
 - Есть перечень версий (история, журнал), можно вернуться к любой.
 - Как узнать, что именно изменилось (файлы и строки)?
 - Можно сравнить версии и представить изменения наглядно.
- Двое пишут разные части одной программы.
 - Как проверить, что они не изменили одно и то же? (См. выше).
 - Как совместить изменения?
 - Автоматически; когда невозможно — будет ясно, что и почему.
- Код пишут несколько человек долгое время.
 - Как обмениваться версиями?
 - Есть общее хранилище всех версий и средства синхронизации.
 - Как уследить за тем, какие были версии и что менялось?
 - У версий есть дата создания, автор, примечания.

Почему нужно специальное решение, а не «облако»?



- Автоматическое ведение истории любых изменений.
- Автоматическая синхронизация.
- Одновременное редактирование в реальном времени.

Разработка программ

- Раздельное редактирование.
- Явное создание пунктов истории:
 - составление набора сохраняемых изменений;
 - комментарии.
- Явное совмещение наборов изменений.

Git — конкретная СКВ

- Сам Git — консольная программа.
 - В поставке для Windows работает в Git Bash, более удобном терминале, чем `cmd.exe`.
 - Многие работают с Git из терминала:
 - Одинаково на любой ОС и любом компьютере.
 - Удобно делиться рецептами, копируя команды.
- TortoiseGit — расширение «Проводника» Windows для работы с Git; есть в лаборатории.
- Отдельные программы: SourceTree, GitG, ...
- Внутри IDE: Visual Studio, CLion, QtCreator.

Основные понятия VCS

- **Хранилище (repository, сокр. репо),** или **репозиторий,** — место хранения всех версий и служебной информации.
- **Версия (revision),** или **ревизия,** — состояние всех файлов на определенный момент времени, сохраненное в репозитории, с дополнительной информацией
- **Коммит (commit; редко переводится как «слепок»)** —
 - 1) синоним версии;
 - 2) создание новой версии («сделать коммит», «закоммитить»).
- **Рабочая копия (working copy или working tree)** — текущее состояние файлов проекта, основанное на версии из хранилища (обычно на последней).
- **Индекс (index, staging area)** — промежуточный буфер, где хранятся коммиты до попадания в хранилище

Основы Git: инициализация

- Будем работать в пустом каталоге `project`:

```
$ mkdir project
```

```
$ cd project
```

- Инициализируем хранилище:

```
$ git init
```

Initialized empty Git repository in `/tmp/project/.git/`

Теперь `project/` — рабочая копия (пустая).

- Просмотрим содержимое каталога:

```
$ ls -A
```

```
.git/
```

← Это хранилище. Содержимым управляет Git, не следует ничего там менять.

Основы Git: просмотр состояния

\$ git status

On branch master

Initial commit ← История не содержит еще коммитов.

nothing to commit (create/copy files and use
"git add" to track) ↗

1. В новый (первый) коммит ничего не планируется добавить.
2. В рабочей копии нет ничего, что можно было бы добавить.


Основы Git: пустой коммит

Обычно Git не допускает пустых коммитов,
но в данном случае именно это и нужно.



```
$ git commit --allow-empty
```

```
[master (root-commit) 5179fab] начало
```



Хэш коммита — его уникальный идентификатор,
по которому коммит можно найти:


```
$ git show 5179fab
```

```
commit 5179fab9f3344f6963f474da399fc9c1fe76e41a
```

```
Author: Dmitry Kozliuk <Dmitry.Kozliuk@gmail.com>
```

```
Date: Sun Mar 12 15:12:46 2017 +0300
```

начало



Берется из настроек Git,
подробнее на ЛР.

Основы Git: обычный коммит

```
$ echo 'первая строка' > file.txt
```

```
$ git status
```

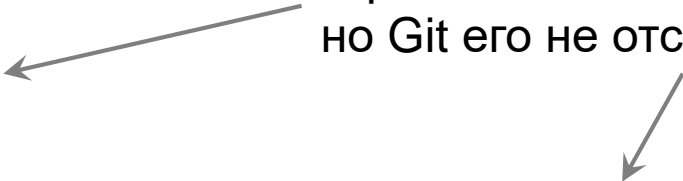
On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

file.txt

В рабочей копии замечен файл,
но Git его не отслеживает.



nothing added to commit but untracked files present
(use "git add" to track)

Основы Git: занесение под СКВ

`$ git add file.txt` ← Файл добавляется в **индекс (index)** — набор изменений для будущего коммита.

`$ git status`

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: file.txt ← Git начал отслеживать файл.

`$ git commit -m 'добавлена первая строка'`

[master **6c81949**] добавлена первая строка

1 file changed, 1 insertion(+)


create mode 100644 file.txt

← Сводная статистика изменений.

Основы Git: просмотр истории

`$ git log --stat`

Новые коммиты
идут в начале




```
commit 6c81949f524b0ecad60fe02222d9642db0b99120
Author: Dmitry Kozliuk <Dmitry.Kozliuk@gmail.com>
Date: Sun Mar 12 17:43:18 2017 +0300
```

добавлена первая строка

```
file.txt | 1 +
```

```
1 file changed, 1 insertion(+)
```

Ключ `--stat` показывает
затронутые коммитом файлы.



```
commit 5179fab9f3344f6963f474da399fc9c1fe76e41a
Author: Dmitry Kozliuk <Dmitry.Kozliuk@gmail.com>
Date: Sun Mar 12 15:12:46 2017 +0300
```

начало

Можно фильтровать по дате, автору,
тексту сообщения, затронутым файлам.

Что такое «branch master»?

короткий формат вывода

показывать ветки

отрисовывать ветки

`$ git log --oneline --decorate --graph`

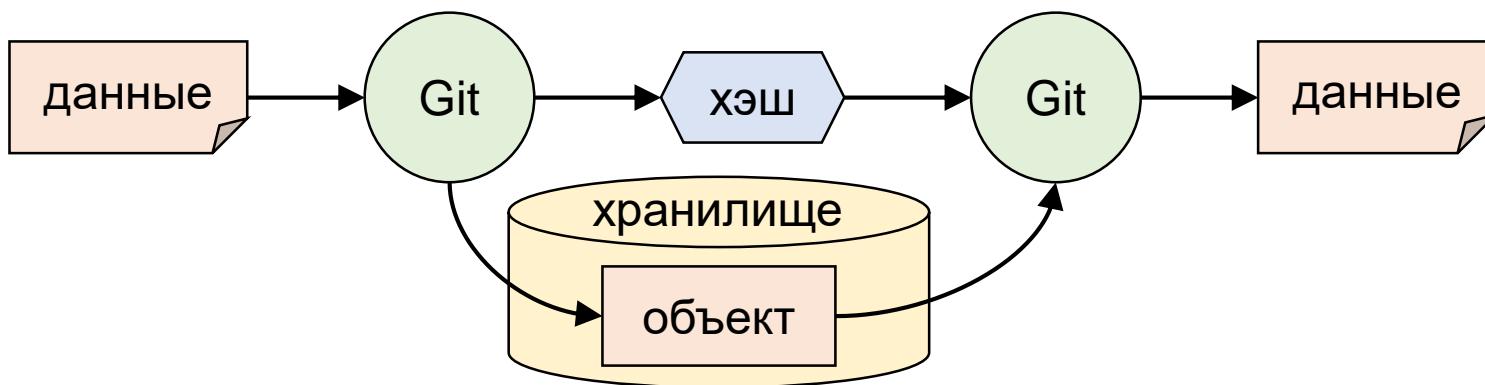
* 6c81949 (HEAD -> master) добавлена первая строка

* 5179fab начало

- **Ветвь (branch)** — это линейный участок истории.
- Ветвь по умолчанию называется **master**.
- О том, зачем нужны другие ветви и как история может быть нелинейной, см. далее.
- В выводе `git log` отмечен конец ветви.

Устройство Git: объекты и хэши

- Git позволяет сохранить **объект (object)** и получить **хэш (hash)**, по которому его затем можно извлечь.
 - Объекты могут быть связаны (объект-коммит и объект-файл).
 - Хэш подобен адресу в C++.

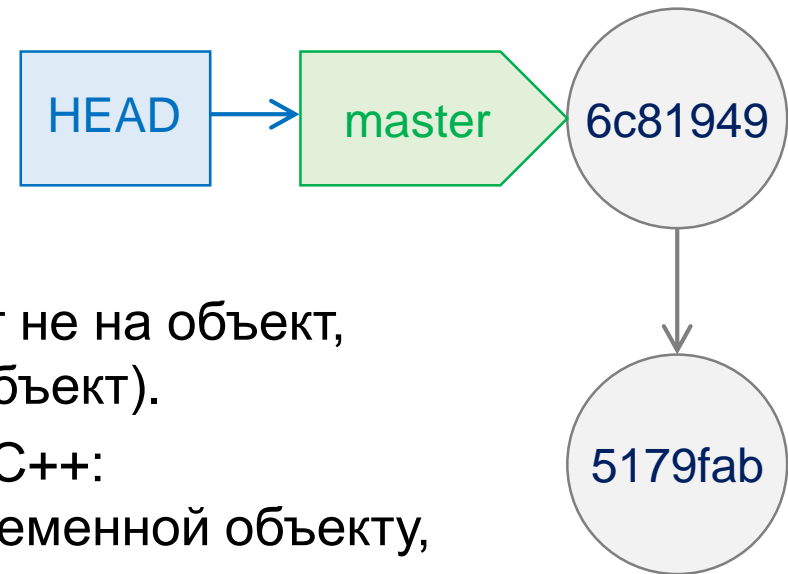


- **Хэш-функция** принимает данные любого размера, а возвращает число фиксированного (SHA1 в Git: 40 символов)
- **Коллизия** — случай, когда для разных данных значения хэш-функции совпадают.
 - Хэш-функция устроена так, что вероятность коллизии мала.

Устройство Git: ссылки

- Хэшу можно дать имя (например, master). Оно называется **ссылкой (reference, или ref)**.

- Ссылка подобна переменной в C++: понятное имя для хэша.

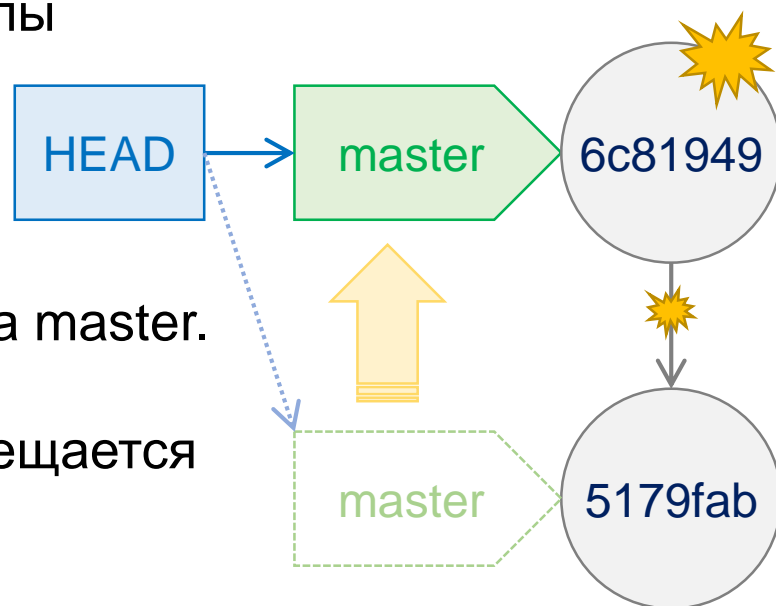


- Особая ссылка **HEAD** указывает не на объект, а на другую ссылку (а та — на объект).
 - HEAD подобна указателю в C++: позволяет обращаться к переменной объекту, не зная его имени.
- **master** всегда указывает на вершину ветки
- **HEAD** всегда указывает на коммит, после которого будет добавлен новый коммит (а заодно, меняет состояние рабочей копии). В идеале, HEAD всегда должен указывать на вершину какой-либо ветки.

Устройство Git: операции

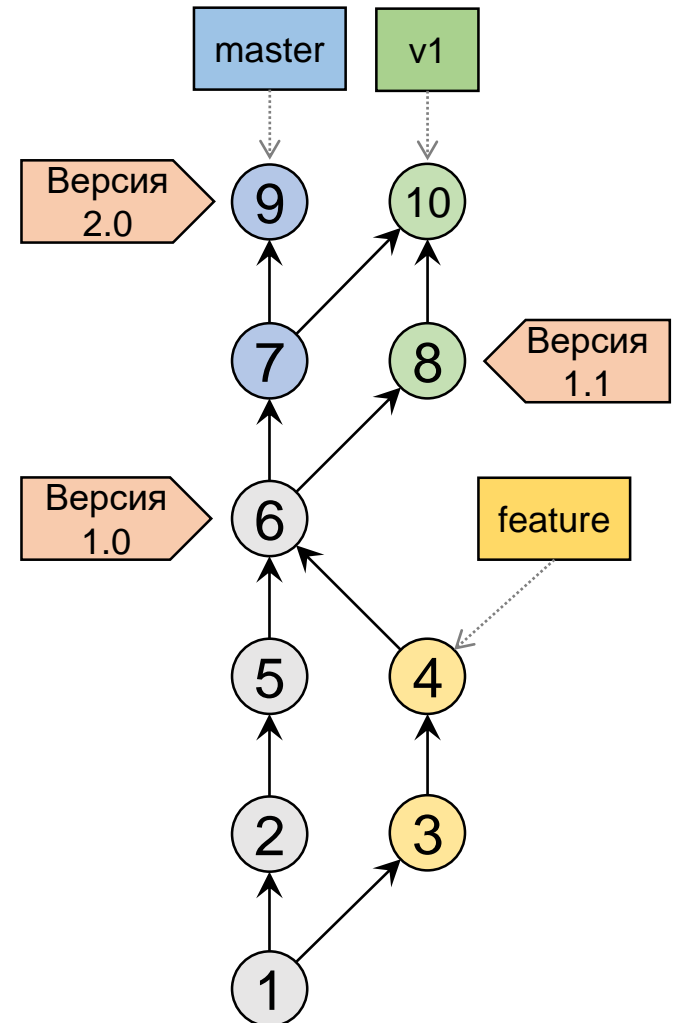
\$ git commit

- Создать и связать объекты-файлы и объект-коммит.
- «Передвинуть» ссылку-ветку, на которую указывает HEAD.
- HEAD по-прежнему указывает на master.
- Ветка — ссылка, которая перемещается при коммитах.
- Если до коммита нельзя добраться (по стрелкам) ни от какой ссылки, он «потерян» (невидим).
 - Но доступен по хэшу, пока не сделано `git gc`.



Ветки и метки

- Зачем **нужны несколько веток?**
 - Параллельная разработка разных частей программы.
 - Альтернативные варианты развития:
 - экспериментальные наработки;
 - правки в старых версиях.
 - Обычно есть главная ветка (master), или ствол (trunk).
- **Метка (git tag)** — отмеченная версия. Отличие от ветки в том, что тэг не перемещается при добавлении коммитов.



Откат изменений

Reset («сбросить»)

- a) двигает указатель ветки;
- b) перемещает текущую ветвь;
- c) сбрасывает рабочую копию к известному состоянию.

Checkout («забрать»)

- a) двигает HEAD;
- b) меняет текущую ветвь;
- c) восстанавливает состояние отдельного файла.

■ Сделаны ненужные изменения.

- Коммита еще не было, нужно вернуться к последнему:

```
$ git reset --hard HEAD
```

- То же самое, но для одного файла:

```
$ git checkout -- file.txt
```

- Коммит был сделан, нужно вернуться к известному:

```
$ git reset --hard 6c81949
```

■ Сделан негодный коммит; нужно его убрать, но оставить изменения:

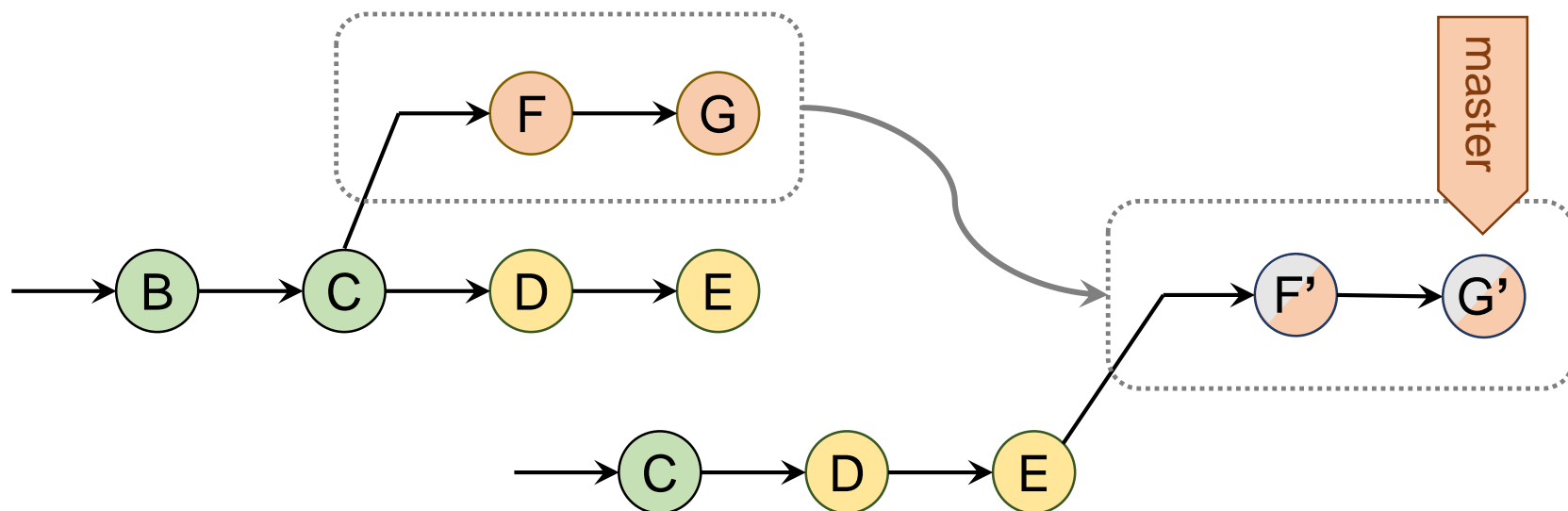
```
$ git reset HEAD~1
```

Refspec — обозначение коммита относительно ссылки. Здесь: на один коммит назад от **HEAD**

Создание веток и откат к ним

- Нужно перейти к известному коммиту, не удаляя имеющиеся.
 1. Создать ветку на нужном коммите:
`$ git branch имя-ветки хэш-коммита`
 2. Переключиться на неё:
`$ git checkout имя-ветки`
 - Создать и переключиться в одно действие:
`$ git checkout -b имя-ветки хэш-коммита`
 3. Переключиться обратно (если стояли на `master`):
`$ git checkout master`

Слияние веток: решение (I)



- История остается линейной, ветвление исчезает.
- История искажается: изменяются коммиты и их порядок.
- Возможны ошибки программиста при переносе коммитов.

Git: перенос ветви

- Текущая ветвь — `master`. Нужно перенести из `master` коммиты, которых нет в `new_branch`, поверх последней.

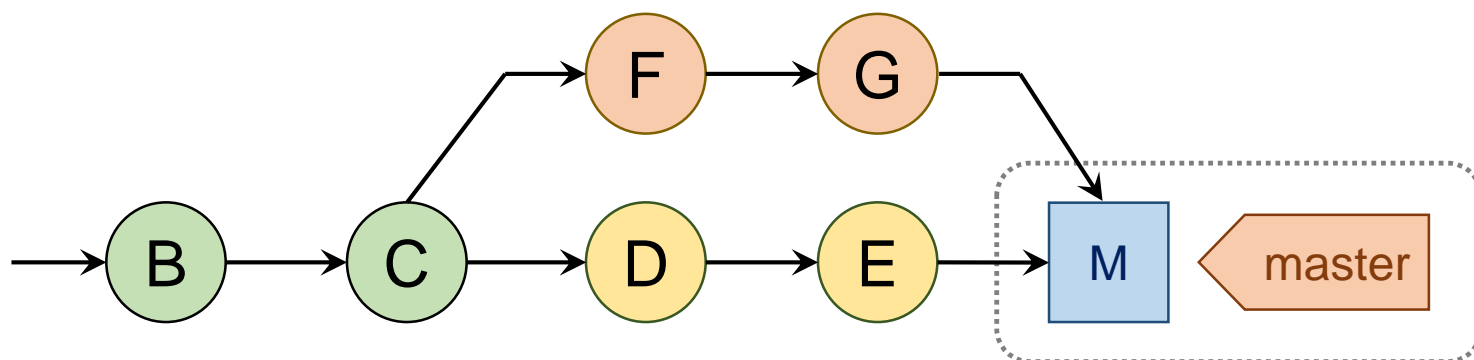
`$ git rebase new_branch`

(вывод зависит от состояния истории)

- Работает для любых ветвей, не только при обновлении.
- **Возможны конфликты (когда наложить изменения нельзя).**

`$ git rebase --abort`

Слияние веток: решение (II)



- История становится нелинейной: возникают merge commits (M).
- Если работа велась параллельно, это остается видно.
- Совмещаются только последние версии — меньше риск ошибиться программисту.
- Все версии неприкосновенны.

Git: слияние ветвей

`$ git merge new_branch`

(вывод зависит от состояния истории)

- Если можно сделать fast-forward, Git так и поступит.
- Если автоматическое слияние возможно, Git создаст новый коммит с отдельным сообщением.
- Возможны конфликты.

`$ git merge --abort`

- Или разрешение конфликта вручную, затем:

`$ git add файл-с-разрешенным-конфликтом`

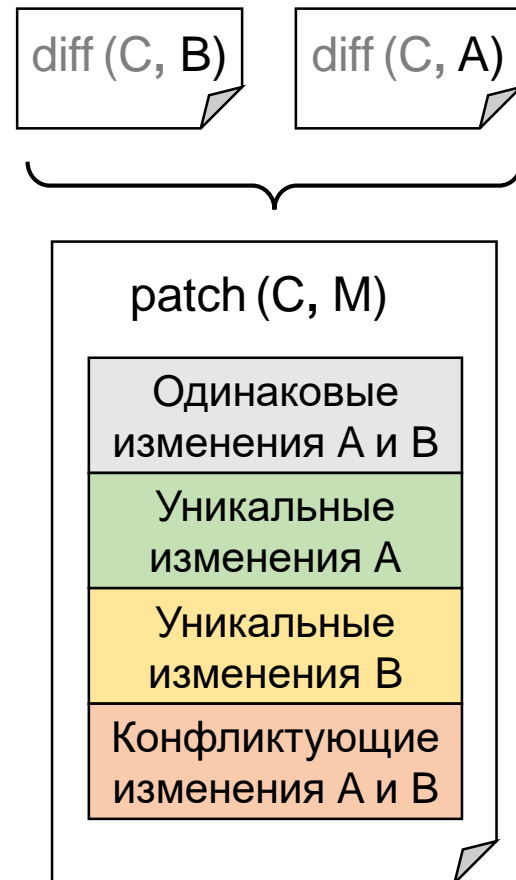
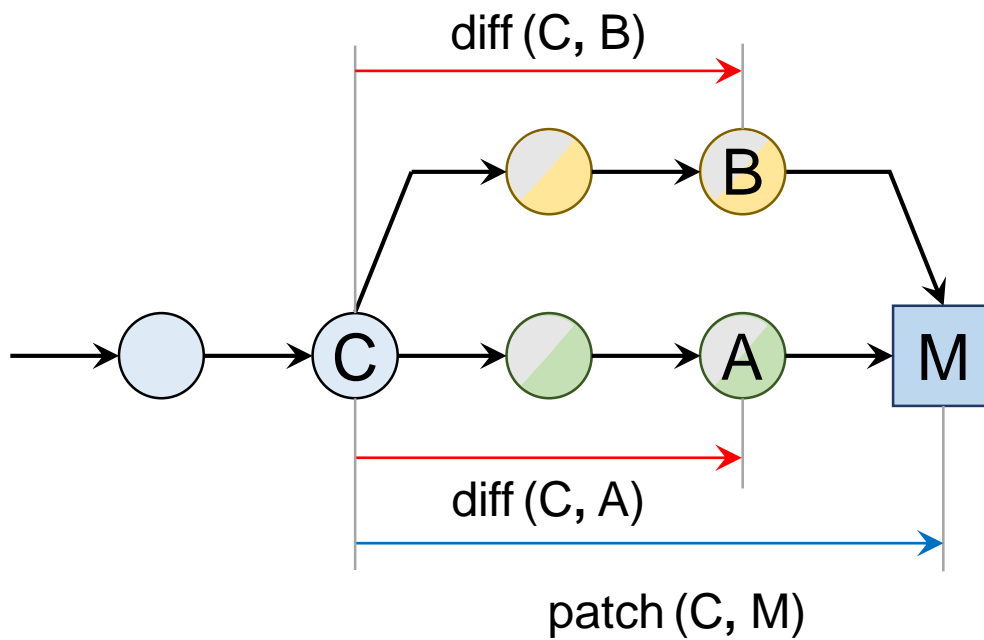
`$ git commit`

- Изучается на ЛР.

Сравнение версий

Просмотр различий между версиями:

`$ git diff хэш-или-ветвь-A хэш-или-ветвь-B`



Формат unified diff

заголовок

```
--- a/main.cpp  
+++ b/main.cpp
```

Обозначение места
изменений в файле.

```
@@ -7,5 +7,6 @@ int main()
```

Измененная
функция
(для удобства
чтения).

```
cout << "Enter A and B: ";  
cin >> a >> b;  
cout << "A + B = " << a + b << '\n'
```

Контекст
(обычно
3 строки)

```
- << "A - B = " << a - b << '\n';  
+ << "A - B = " << a - b << '\n';  
+ << "A / B = " << a / b << '\n';  
}
```

Удаленные (-) и добавленные (+) строки.

Понятия VCS

- **Слияние (merge)** —

объединение двух версий в единую;
слияние ветвей — объединение их последних версий.

- **Конфликт (conflict)** —

ситуация, когда VCS не может автоматически слить внесённые изменения (т. е. когда были по-разному изменены одни и те же места в файлах).

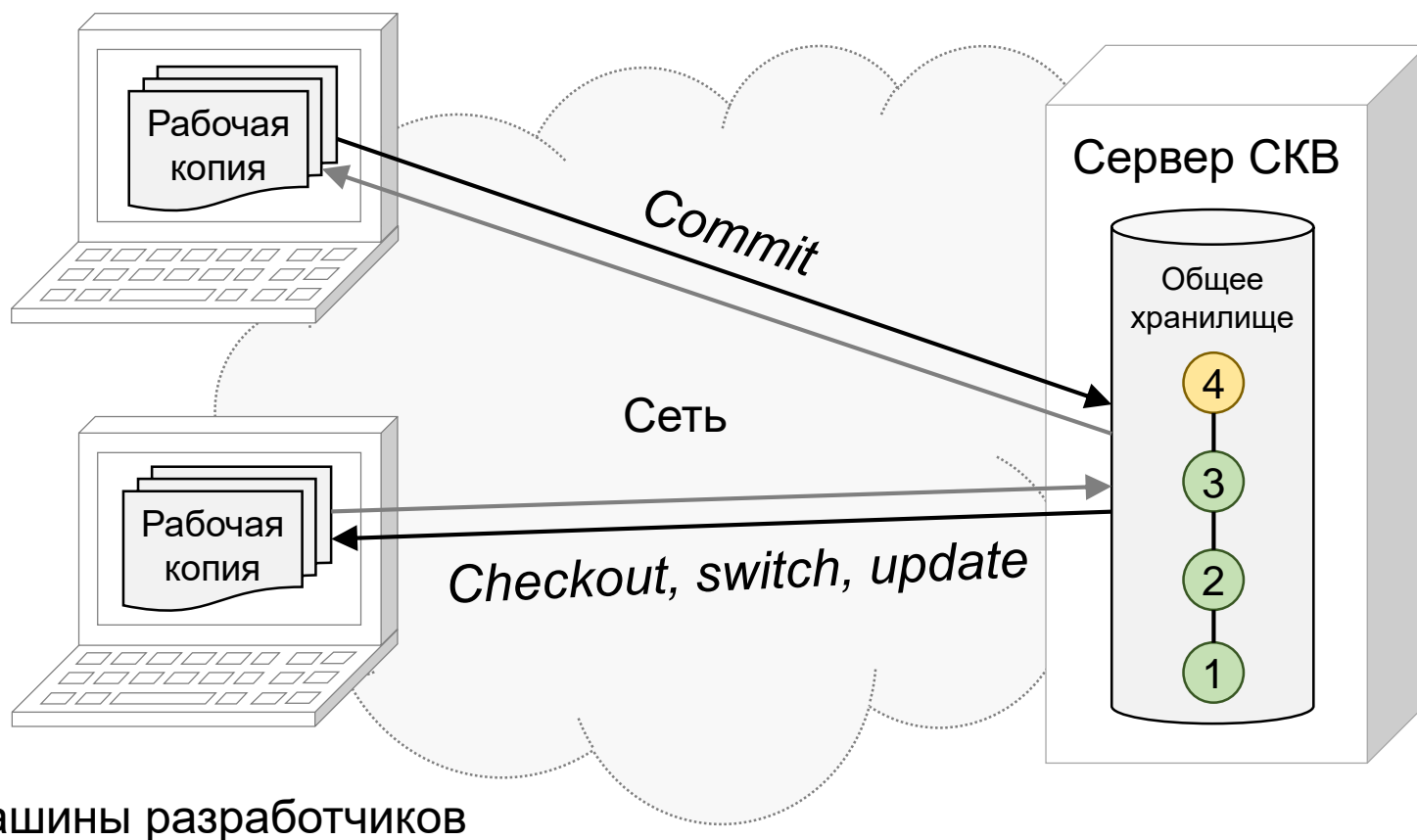
- **Разность (difference, diff)** —

построчные различия между файлами (разных версий).

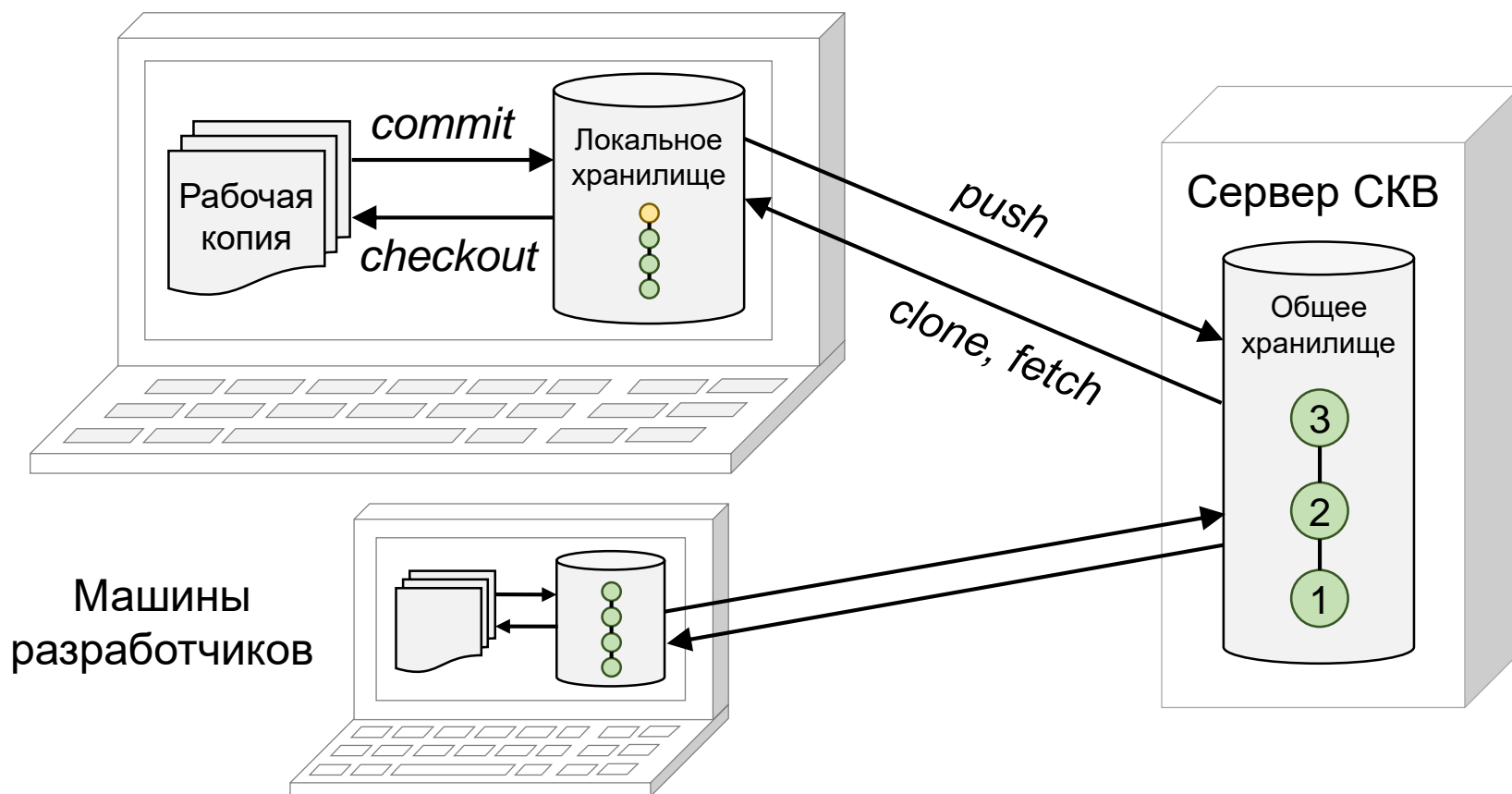
- **Заплата (patch), патч** —

файл-инструкция, какие правки нужно внести (по сути, это **diff**).

Общее хранилище: централизованная VCS



Отдельные хранилища: распределенная VCS (DVCS)



Виды систем контроля версий

Централизованные

- Простота использования.
- Вся история — всегда в едином общем хранилище.
- Нужно подключение к сети.
- Резервное копирование нужно только одному хранилищу.
- Удобство разделения прав доступа к хранилищу.
- Почти все изменения навсегда попадают в общее хранилище.

Распределенные

- Двухфазный commit:
 - 1) запись в локальную историю;
 - 2) пересылка изменений другим.
- Подключение к сети не нужно.
- Локальные хранилища могут служить резервными копиями.
- Локальное хранилище контролирует его владелец,
 - но общее — администратор.
- **Возможна правка локальной истории перед отправкой на сервер.**

Удаленное хранилище: что это и где его взять?

- Технически:
 1. Сервер с доступом по сети (HTTPS, SSH).
 2. Само хранилище (например, переименованный `.git/`) без рабочей копии (т. н. **bare repository**).
- Доступ к удаленному хранилищу может быть ограничен администратором на уровне сервера (т. е. в Git это не входит).
- Практические решения (*изучаются на ЛР*):
 - Бесплатные или арендуемые хранилища в интернете: GitHub.com, BitBucket.org, GitLab.com и другие.
 - На собственном сервере:
 - Просто Git и доступ по сети.
 - GitLab, Jira, Gogs, Gitolite, Upsource и другие.
 - Большинство решений имеют также web-интерфейс.

Git: загрузка кода в хранилище (1)

Задача. Есть код под Git и адрес хранилища, куда его нужно отправить.

- Сохранить **адрес** под именем **origin** (традиционно):

```
$ git remote add origin https://github.com/user/repo.git
```

- Отправить коммиты ветви **master** на сервер, указав, что она и на сервере будет называться **master**:

```
$ git push --set-upstream origin master
```

```
Counting objects: 5, done.
```

```
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (5/5), 462 bytes | 0 bytes/s, done.
```


```
Total 5 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/user/repo.git
```

```
* [new branch]      master -> master
```

```
Branch master set up to track remote branch master from origin.
```

Здесь может
понадобиться
ввести пароль.



Git: загрузка кода в хранилище (2)

- **Задача.** Сделаны новые коммиты, нужно добавить их в удаленное хранилище.

\$ git push

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 325 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To <https://github.com/user/repo.git>

6c81949..10891c2 master -> master

- В соответствующей ветви удаленного хранилища не должно быть коммитов, которых нет локально. (Локальная история должна **опережать** удаленную.)

Git: загрузка кода из хранилища

- Нужно скопировать с сервера всё:

```
$ git clone https://github.com/user/repo.git
```

```
Cloning into 'repo'...
```

```
remote: Counting objects: 8, done.
```

```
remote: Compressing objects: 100% (4/4), done.
```

```
remote: Total 8 (delta 0), reused 8 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (8/8), done.
```

← Каталог, в который
будет загружена
рабочая копия
и хранилище.

- На сервере (**origin**) появились коммиты,
нужно скачать их в локальное хранилище:

- **git fetch origin**

```
remote: Counting objects: 3, done.
```

```
remote: Compressing objects: 100% (2/2), done.
```

```
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (3/3), done.
```

```
From https://github.com/user/repo
```

```
10891c2..0cfc640 master -> origin/master
```

- Для всех веток: **git fetch origin --all**.

Основные понятия DVCS

- **Загрузка изменений (fetch)** —
загрузка наборов изменений (commit-ов) из удаленного хранилища.
- **Обновление, «подтягивание» (pull)** —
загрузка изменений и немедленное слияние с локальным хранилищем (pull = fetch + merge).
- **Отправка изменений (push)** —
передача наборов изменений в удаленное хранилище с немедленным слиянием.
 - Если при слиянии возникает конфликт, происходит ошибка.
 - Возможна, но не рекомендуется, **force push** — принудительная перезапись удаленной истории (`git push --force`).

Git Stash

Несохраненные изменения можно спрятать в специальную невидимую, строго локальную область — stash. Например:

- Чтобы скачать последние изменения в процессе написания кода.
- Чтобы временно переключиться на другую ветвь.
- Сохранить изменения в stash:
`$ git stash save [возможно, примечание]`
- Извлечь изменения из stash в рабочую копию:
`$ git stash pop`
- Stash работает как стек (стопка): можно вызвать `save` несколько раз, `pop` извлекает последние изменения.
- Можно просматривать элементы stash, удалять их, извлекать не последний.

Модели ветвлений (branching models), или рабочие процессы (workflows)



Договоренность о ветках внутри команды.

- Кем, когда и зачем создаются?
 - а) по одной каждым разработчиком для своих задач
 - б) по ветке на задачу,
 - с) по ветке на группу связанных задач...
- Что содержат?
 - ветка со стабильным кодом,
 - ветка с нововведениями...
- Кем, когда, куда и по каким критериям сливаются?
 - когда код в ветке протестирован, она сливается в master,
 - раз в неделю master сливается во все ветки...
- Как называются?

Какими должны быть коммиты?

- **Гранулярность** — насколько крупными должны быть коммиты?
 - Большие сложно оценить, задачи смешаны.
 - Мелкие неудобно читать, теряется контекст.
 - Хорошая идея: размер — одно *логическое* изменение.
- Код в коммитах должен быть компилируемым.
 - Git этого не требует, но иначе неудобно на практике.
- Сообщения должны отражать цель и суть правок.
 - Что именно поменялось, покажет diff.
 - Иногда уместны объемные пояснения.

Осталось за рамками

- **Настройки Git** (изучается на ЛР), в т. ч. `.gitignore`.
- **Staging (Git):**
 - детальный контроль содержимого очередного commit;
 - можно включить в коммит часть файла (hunk).
- **Cherry-pick (Git)** —
копирование commit-ов между ветвями.
- **Pull/merge requests** —
запросы администратору (главному разработчику)
на слияние ветки в главную. Применяется в открытых
проектах и дисциплинированных командах.

Некоторые VCS и их особенности

- **Subversion (SVN):**

- одна из старейших, и потому все еще популярна;
- централизованная.

- **Git:**

- распределенная, популяризовала этот тип;
- самая распространенная на сегодня (GitHub, BitBucket).

- **Mercurial (Hg):**

- распределенная, похожа на Git, но отличается рядом аспектов;
- широкие возможности по управлению хранилищами.

- **Perforce:**

- Основана на Git, но требует центрального хранилища;
- улучшенная работа с файлами не-кода (документы и т. п.);
- ядро комплексной системы ведения проекта;
- коммерческая лицензия.

- **CVS:**

- морально устарела;
- <http://SourceForge.net>
- имела механизм locks: файл мог редактировать только один человек в момент времени.

Ресурсы к лекции

- Scott Chacon, Ben Straub. *Pro Git*.

\$ git help <команда>

- Joel Spolsky. *Hg Init: a Mercurial Tutorial*.
- Ben Collins-Sussman et al. *Version Control with Subversion*.

- Хостинги хранилищ:

- GitHub:

- самый популярный;
- больше функций, но только Git;
- бесплатно — только открытые хранилища.

- BitBucket:

- Git, Mercurial;
- есть бесплатные закрытые хранилища;

- Все ссылки есть на странице курса.

