

Командная разработка ПО. Использование Git и GitHub

Отличия крупных проектов от учебного кода

- Разрабатываются, как правило, группой людей.
- Живут долго.

Особенности разработки командой

- Каждый участник может знать не всё.
- Люди приходят и уходят — знания должны сохраняться и передаваться.
- Весь проект разбивается на задачи, которые делятся между участниками.
- Проблема одновременного редактирования кодовой базы.

Особенности долгоживущего кода

- Наличие фазы сопровождения в жизненном цикле ПО.
- Поддержание чистоты кода — снижение стоимости сопровождения ПО.

Жизненный цикл ПО

- Сбор требований.
- Проектирование.
- Кодирование.
- Тестирование и отладка.
- Эксплуатация и сопровождение.

Жизненный цикл глазами программиста

- Написание первой версии системы с нуля:
 - проектирование,
 - кодирование,
 - отладка.
- Поддержка текущей версии и добавление новых функций:
 - проектирование,
 - кодирование,
 - отладка.



Инструменты и методологии

- Поддержка чистоты кода.
- Средства управления проектом.
- Системы управления версиями.
- Код как справочник.
- Система непрерывной интеграции.

Поддержка чистоты кода (1)

- Единый стиль кодирования, которого обязаны придерживаться все.
 - <http://learn.javascript.ru/coding-style>
- Следование хорошим практикам при написании кода (например, SOLID для ООП).
- Ревью перед каждой публикацией кода.
- Автоматизированные регрессионные тесты, включая юнит-тесты.
- Система непрерывной интеграции.

Поддержка чистоты кода (2)

- Рефакторинг — улучшение внутренней структуры кода без изменения его внешнего поведения.
- Технический долг — факт наличия незавершённой работы в прошлом, либо откладывание такой работы на будущее.
- Автобусный фактор — мера сосредоточенности информации среди отдельных членов проекта.

Поддержка чистоты кода (3)

- Ошибка в программе — несоответствие ожидаемого и фактического поведения программы. Формально: несоответствие фактического поведения и описанного в документации.

Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живёте.

Стив Макконнелл,
автор книги «Совершенный код»

Средства управления проектом (баг-трекеры, таск-трекеры) (1)

- Примеры: Bugzilla, Jira, Redmine, встроенные средства в GitHub.
- Используются для упорядочивания деятельности программистов:
 - постановка задач,
 - контроль их выполнения,
 - накопление и сохранение знаний (постановка задачи, переписка в комментариях, прикреплённые файлы, связанные коммиты).

Средства управления проектом (баг-трекеры, таск-трекеры) (2)

- Задача (task, bug, issue, ticket) имеет некоторый номер, может иметь статусы (открыта, в работе, на ревью, на тестировании, закрыта).
- Другие атрибуты задачи: проект, тип (задача/ошибка), оценка времени, затраченное время, метки для поиска, кому назначена, автор задачи...
- Развитые таск-трекеры позволяют организовывать иерархию задач (выпуск, темы, задачи, подзадачи), отношения между ними (связана, блокирует, блокируется...)

Средства управления проектом (баг-трекеры, таск-трекеры) (3)

- Любой код, который пишет программист, должен выполняться в рамках назначенной на него задачи. В идеале, вообще любая деятельность.

Системы управления версиями

- СУВ — программное обеспечение для облегчения работы с изменяющейся информацией.
- Мы будем рассматривать только СУВ для исходных кодов программ.
- Виды СУВ:
 - Локальные: RSC, SCCS.
 - Централизованные: CVS, SVN, MS Team Foundation.
 - Распределённые: Git, Mercurial.

Принципы систем управления версиями (1)

- Есть некоторое централизованное *хранилище (репозиторий)*, с которым синхронизируются разработчики.
- Данные в СУВ организованы в виде *фиксаций (commit, коммитов)* — снимков состояния рабочего каталога в определённые моменты времени. Фиксация имеет автора и временной штамп.
- *Ветки* — поддерживается возможность создания нескольких независимых параллельных ветвей, развивающихся независимо после точки расхождения.
- Также предоставляется возможность слияния ветвей — добавление изменений, сделанных в одной ветви в другую.
- *Метки (tags, теги)* — на отдельные фиксации можно поставить метки версии.

Принципы систем управления версиями (2)

- Любые рабочие, тестовые и демонстрационные версии собираются только из главного хранилища.
- Последняя фиксация в главной ветке всегда должна быть корректна.
- Любое значимое изменение — отдельная ветвь.
- Версии проекта помечаются тегами. Выделенная и помеченная ветвь не изменяется.
- Коммиты должны быть удобны для восприятия, каждый коммит должен решать одну задачу:
 - или добавление новой функции,
 - или исправление ошибки,
 - или рефакторинг/стилевую правку,
 - ...

Преимущества СУВ перед их отсутствием

- Бекап с удобной возможностью отката.
 - Следствие: исходный код не засоряется закомментированным старым кодом.
- Удобные средства просмотра различий и слияния изменений.

Что хранить в репозитории?

- Общий принцип:
 - Хранятся файлы, которые создаются программистами вручную.
 - Не хранятся те файлы, которые создаются программно.
- Хранят:
 - Исходные тексты программ (.c, .java, .js...)
 - Внутреннюю документацию для программистов.
 - Исходные тексты документации (.tex, .md...).
 - Спецификации, по которым генерируется код.
 - Списки зависимостей (package.json, package-lock.json...).
- Не хранят:
 - Скомпилированные файлы (.exe, .jar, .pdf...).
 - Объектные файлы (.o, .obj, .class...).
 - Сгенерированные файлы (y.tab.c).
 - Проекты IDE (.idea).
 - Автоматически загружаемые зависимости (node_modules).

Код как справочник (1)

- Далее я перескажу идеи поста gaperton'a «Миф о документации. Продолжение»: <http://gaperton.livejournal.com/60632.html>
- Виды «документов», которые пишут:
 - «Договор» — пишется для того, чтобы синхронизировать взаимопонимание. Пишется один раз и не сопровождается.
 - «Справочник» — служит эталоном, должен поддерживаться актуальным.
 - «Учебник» — все любят читать, никто не любит писать.
- «Учебников» по архитектуре конкретной системы никто не пишет.

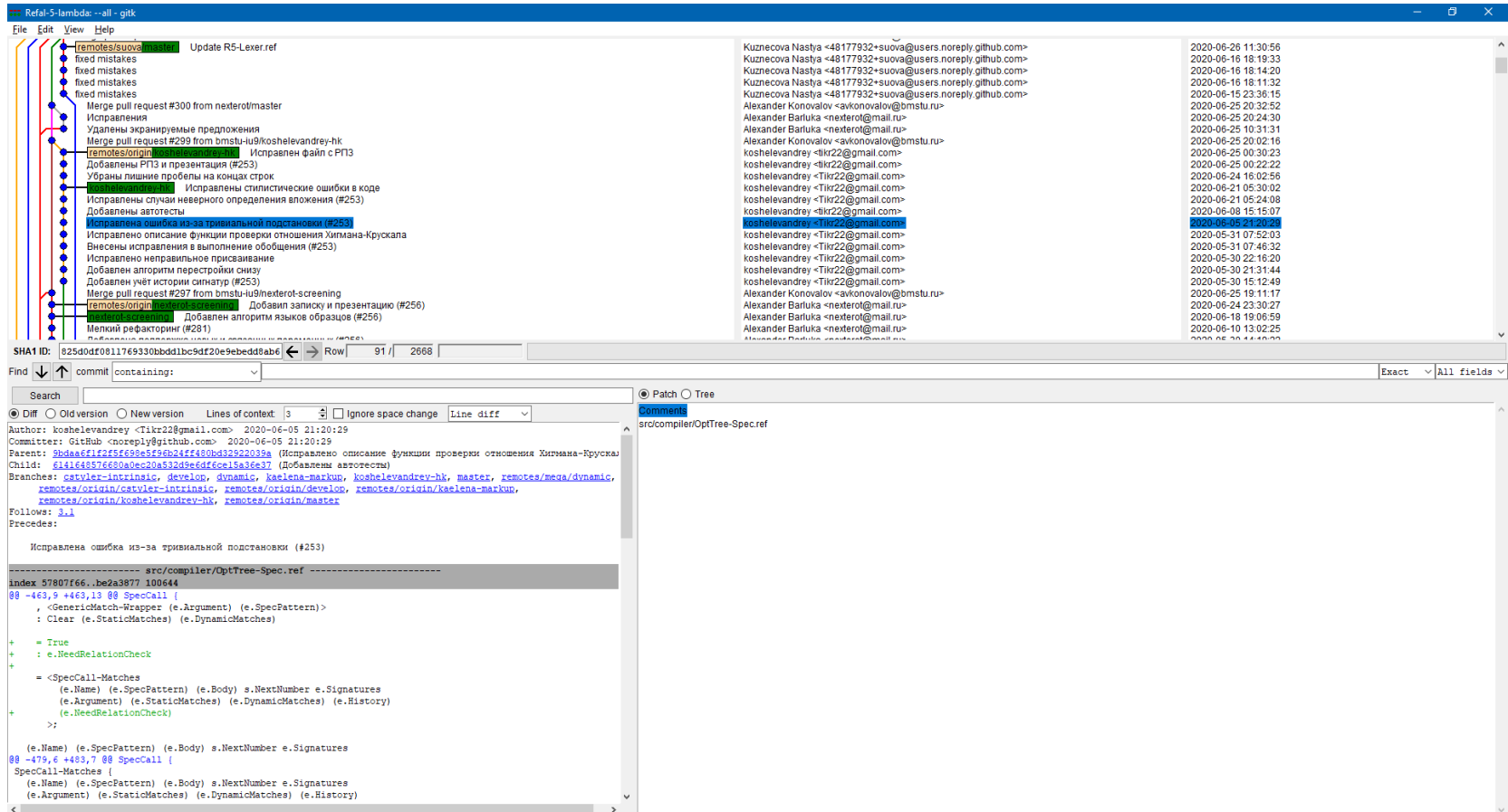
Код как справочник (2)

- Хорошая новость: справочником можно сделать исходный код!
- Для этого:
 - Исходный код нужно содержать в чистоте, проводить ревью перед публикацией кода. Ревьюер разделяет ответственность за код!
 - Использовать таск-трекеры, переписку по задаче вести в комментариях к ней.
 - Писать ясные комментарии к коммитам, в комментариях обязательно указывать номер задачи.
- Тогда:
 - СУВ позволяют для каждой строки кода узнать коммит, которым она была внесена (`svn annotate`, `git blame`...)
 - Найдя коммит, можно прочесть его комментарий и уже что-то понять.
 - По номеру задачи в коммите открыть задачу, а там и постановка задачи, и обсуждение в комментариях, и другие связанные коммиты.

Система управления версиями Git

- Это распределённая система управления версиями.
- Работа с репозиториями осуществляется командой `git`. Её синтаксис:
`git <команда> <аргументы...>`
- Для получения справки по команде:
`git <команда> --help`
`git help <команда>`
`man git-<команда>`

Утилита gitk



Утилита `gitk`

- Утилита `gitk` позволяет просматривать содержимое репозитория в графическом режиме.
- Её запуск в папке репозитория открывает дерево коммитов текущей ветки.
- На Linux или macOS её нужно запускать в фоне — с `&` на конце командной строки.
- Запуск с ключом `--all` отображает все ветви репозитория.

<code>gitk --all</code>	(на Windows)
<code>gitk --all &</code>	(на unix-like)

- На Windows она устанавливается вместе с Git, на Linux её нужно устанавливать отдельно (`sudo apt install gitk`).

Git-репозитории

- Репозиторий может располагаться как на удалённом сервере, так и быть локальным.
- Локальный репозиторий — скрытая папка `.git` в рабочем каталоге.
- Папку можно сделать репозиторием при помощи команды
`git init`
- Можно клонировать имеющийся репозиторий:
`git clone <путь-к-репозиторию>`
- Например:
`git clone https://github.com/bmstu-iu9/scheme-labs`

Начальная настройка репозитория

- После установки Git нужно выполнить следующие команды:

```
git config --global user.name "Ваше имя"  
git config --global user.email "Ваш email"  
git config --global core.editor <команда>
```

- Без этих настроек не получится создавать коммиты.
- Email нужно указывать тот же, что и при регистрации на GitHub.
- При использовании gitk на Windows:

```
git config --global gui.encoding utf-8
```

Коммиты в Git

- *Коммит* — это снимок текущего состояния рабочего каталога.
- Коммит идентифицируется хешом от своего содержимого.
- Коммит включает в себя содержимое всех файлов рабочего каталога, ссылку на предка, сообщение коммита, имя автора и временной штамп.
- У самого первого коммита в репозитории ссылки на предка нет. У коммитов-слияний может быть несколько предков.

Коммиты в Git

- Коммит создаётся командой

```
git commit
```

- При выполнении этой команды откроется окно текстового редактора. В этом окне нужно ввести комментарий к коммиту. Git не даёт возможности создавать коммиты без комментариев.
- Если комментарий состоит из одной строки, можно использовать команду

```
git commit -m "Комментарий к коммиту"
```

- На самом деле, всё немного сложнее, но об этом позже.

Ветки в Git

- *Ветка* — перемещаемая ссылка на коммит.
- В каждый момент времени текущей является одна из веток.
- При создании нового коммита, ссылкой на предка становится предыдущий коммит, на который указывала ветка.
- После создания нового коммита ветка сдвигается на новый коммит.
- Ветки создаются командой

```
git branch <имя-ветки>
```

Новая ветка будет ссылаться на текущий коммит текущей ветки.

- Для переключения на другую ветку используется команда

```
git checkout <имя-ветки>
```

- Ветка по умолчанию — `main`.
- `git checkout -b <ветка> =
git branch <ветка> + git checkout <ветка>`

Слияние ветвей

- Для слияния двух ветвей используется команда

```
git merge <имя-другой-ветки>
```

- При её выполнении создаётся коммит с двумя предками — верхушками сливаемых ветвей, коммит помещается в текущую ветку.
- При слиянии возможны конфликты, если одни и те же участки файла по-разному менялись в разных ветвях.

Разрешение конфликтов при слиянии ветвей (1)

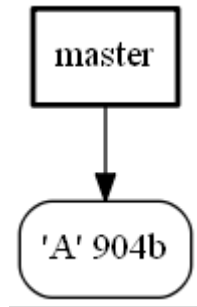
- При выполнении команды `git merge` возможны различные ситуации.
 - Если текущий коммит является непосредственным предком вершины коммита другой ветки, то происходит перемотка (`fast forward`) — коммит-слияние не создаётся, указатель ветки перемещается на вершину второй ветки.
 - Если конфликтов не произошло — изменения в обеих ветках не затрагивали одних и тех же мест одних и тех же файлов — создаётся коммит-слияние с сообщением по умолчанию.
 - Если же возникли конфликты, слияние прерывается на полпути.
Что в этом случае делать — узнаем чуть позже.

Ветки в Git (пример)

```
git init
```

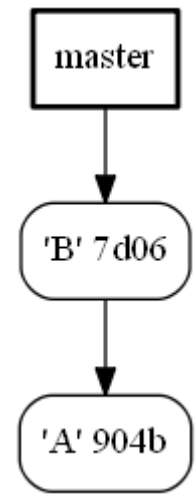
Ветки в Git (пример)

```
git init  
git commit -m "A"
```

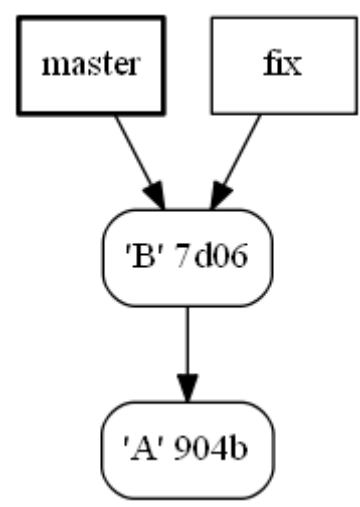


Ветки в Git (пример)

```
git init  
git commit -m "A"  
git commit -m "B"
```

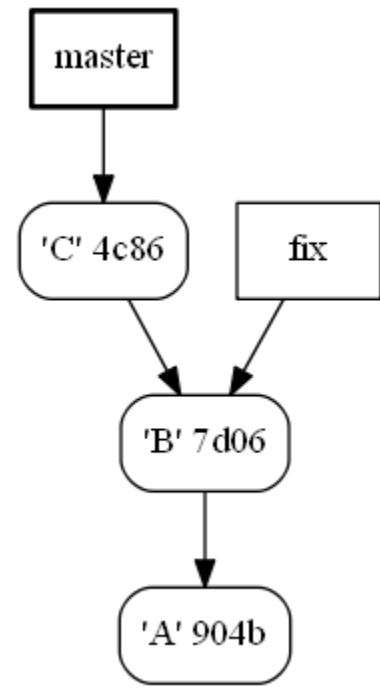


Ветки в Git (пример)



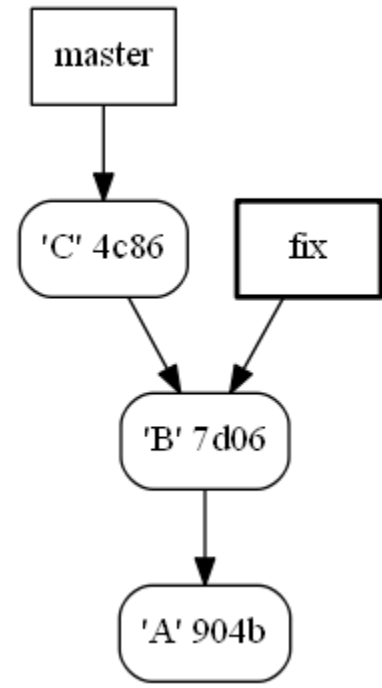
```
git init
git commit -m "A"
git commit -m "B"
git branch fix
```

Ветки в Git (пример)



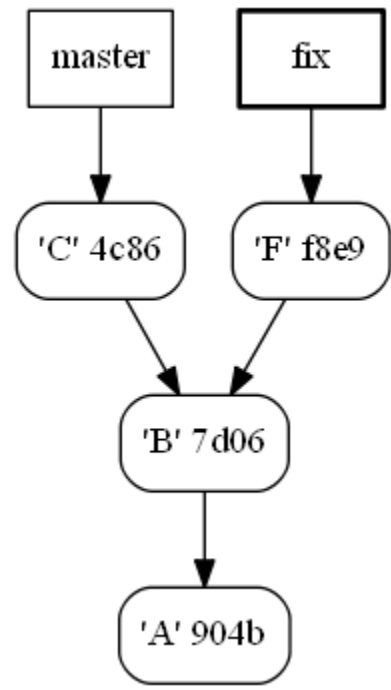
```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
```

Ветки в Git (пример)



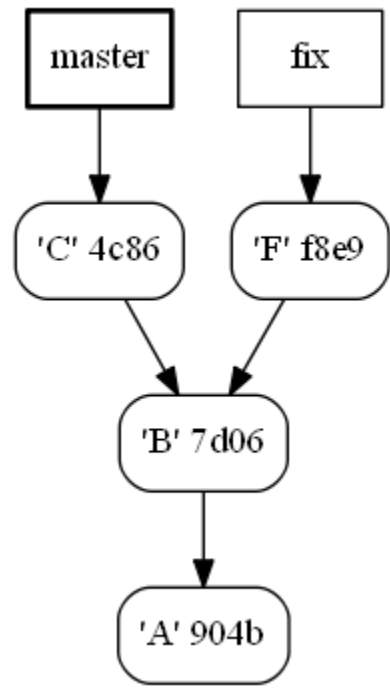
```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
git checkout fix
```

Ветки в Git (пример)



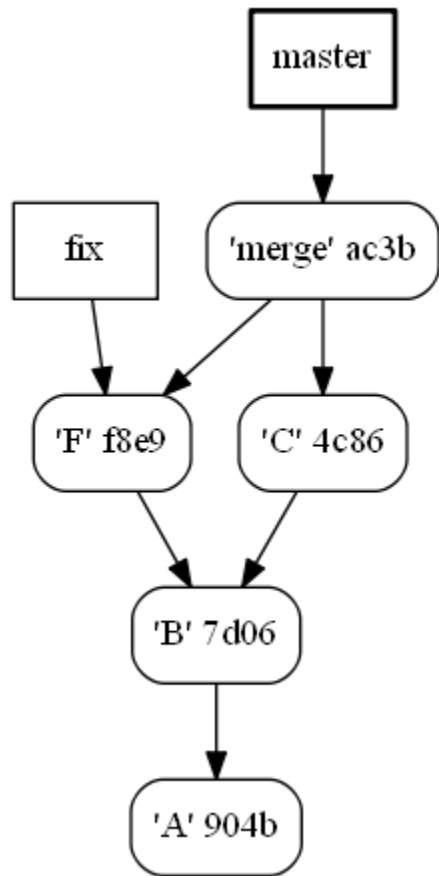
```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
git checkout fix
git commit -m "F"
```

Ветки в Git (пример)



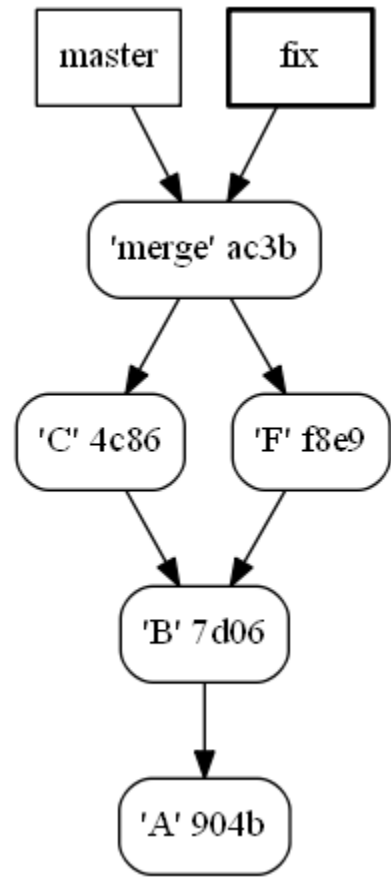
```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
git checkout fix
git commit -m "F"
git checkout main
```

Ветки в Git (пример)



```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
git checkout fix
git commit -m "F"
git checkout main
git merge fix
```

Ветки в Git (пример)



```
git init
git commit -m "A"
git commit -m "B"
git branch fix
git commit -m "C"
git checkout fix
git commit -m "F"
git checkout main
git merge fix
git checkout fix
git merge main
```


Метки в Git

- *Метка* — перемещаемая ссылка на коммит.
- Метки бывают трёх видов:
 - простые — просто ссылка на коммит,
 - аннотированные — также содержат комментарий, автора и отметку времени создания,
 - подписанные — то же, что и у аннотированных + криптографическая подпись.
- Метками обычно помечаются релизы.

Как создать коммит (1)

- HEAD — указатель на текущий коммит текущей ветки.
- *Индекс или область подготовки* — промежуточное хранилище, из содержимого которого создаётся коммит.
- Поэтому создание коммита выполняется в два этапа:
 - подготовка изменений для коммита — помещение изменённых файлов в индекс,
 - собственно, создание коммита из подготовленных файлов.

Как создать коммит (2)

- Подготовка изменений к коммиту выполняется командой

```
git add <имя-файла>
```

где <имя-файла> — файл с изменениями, которые мы хотим закоммитить.

- После чего команда

```
git commit
```

создаст коммит из подготовленных файлов

Команда `git add -p`

- Эта команда позволяет выбирать правки для коммита в интерактивном режиме.
- При запуске `git add -p` будут по очереди предлагаться непроиндексированные изменения в кодовой базе.
- Выбирать их можно, вводя с клавиатуры `y` (yes) или `n` (no).
- Выбранные изменения помещаются в индекс.
- При помощи `git add -p` можно сделать несколько разных коммитов из одного набора правок.
- Кроме того, команда позволит избежать фиксации ошибочных правок.

Состояния файлов (1)

- Вообще, файл в рабочем каталоге, с точки зрения Git, может находиться в четырёх состояниях:
 - *Неотслеживаемый* — в индексе этого файла нет, в рабочем каталоге есть.
 - *Неизменённый* — файл в рабочем каталоге совпадает с файлом в индексе и HEAD.
 - *Модифицированный* — файл в рабочем каталоге отличается от файла в индексе.
 - *Подготовленный* — файл в индексе отличается от файла в HEAD.

Состояния файлов (2)

- Для просмотра состояний файлов используется команда `git status`.

```
Z:\>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   I-am-staged.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   I-am-modified.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        I-am-untracked.txt
```

Состояния файлов (3)



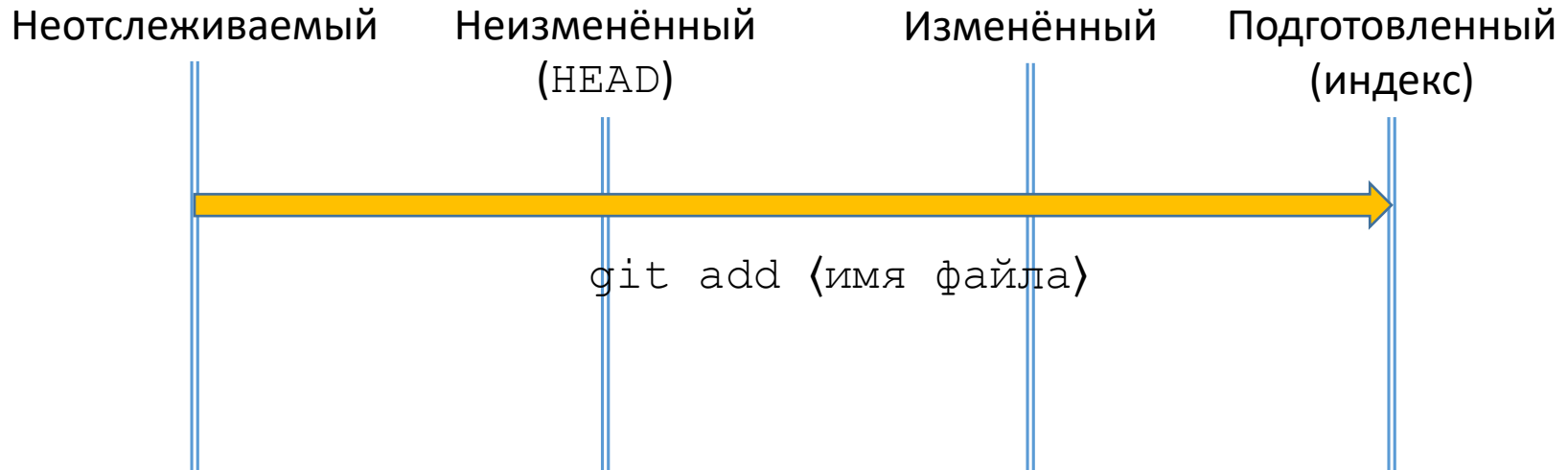
Редактирование файла делает его изменённым, он теперь отличается и от HEAD, и от индекса.

Состояния файлов (3)



Команда `git add` добавляет изменённые файлы в индекс.
Файл в рабочем каталоге теперь совпадает с содержимым индекса.

Состояния файлов (3)



Команда `git add` также добавляет неотслеживаемые файлы в индекс.

Состояния файлов (3)



Команда `git commit` создаёт новый коммит из правок в индексе. Индекс после этого совпадает с HEAD, а HEAD совпадает с рабочим каталогом.

Состояния файлов (3)



Можно сбросить содержимое индекса командой `git reset`.
Индекс после этого будет совпадать с HEAD.

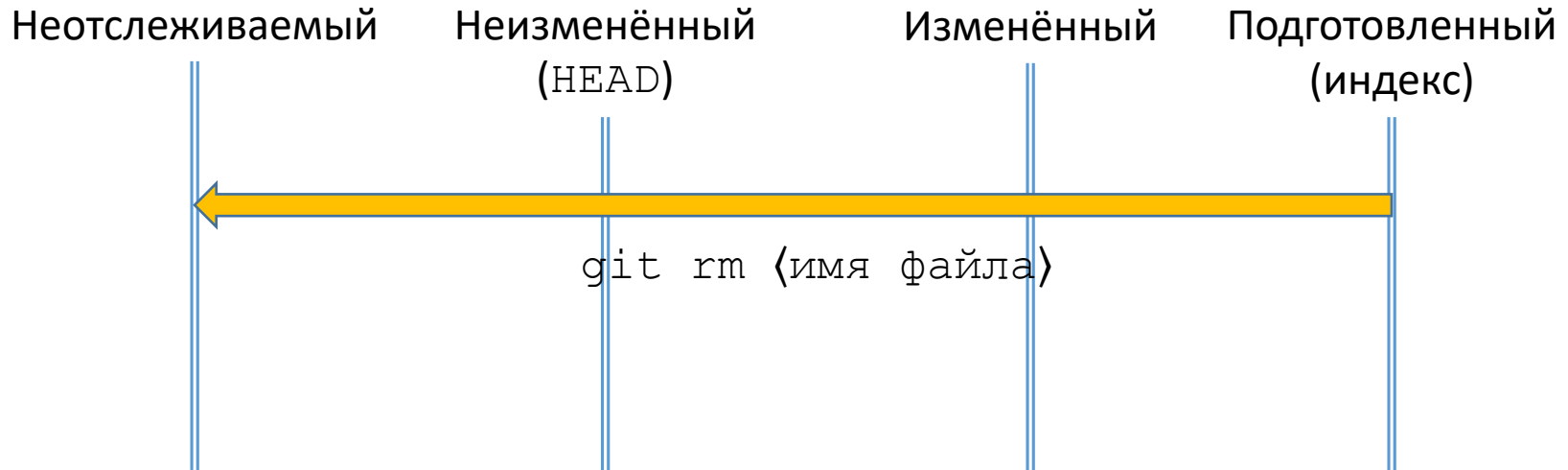
Состояния файлов (3)



Можно вернуть состояние и индекса, и рабочего каталога к содержимому последнего коммита командой `git reset --hard`.

Внимание! Эта команда удалит все несохранённые изменения.

Состояния файлов (3)



Команда `git rm <имя файла>` удаляет и из индекса, и из рабочего каталога указанный файл. Если после удаления файл с этим именем появится в рабочем каталоге, он будет неотслеживаемым.

Состояния файлов (3)



Команда `git diff` показывает разницу между рабочим каталогом и индексом.

Состояния файлов (3)



Команда `git diff --cached` (синоним `git diff --staged`) показывает разницу между HEAD и индексом.

То, что показывает `git diff --cached`, войдёт в коммит!

Состояния файлов (3)



Команда `git diff HEAD` показывает разницу между рабочим каталогом и HEAD.

Шпаргалка: как создать коммит

- Подготавливаем изменения к индексации:

```
git add -p  
git add file.txt
```

(если файл не отслеживался)

- Смотрим индекс — правки, которые войдут в коммит:

```
git diff --cached
```

- Если получается что-то не то, то сбрасываем индекс:

```
git reset
```

и начинаем заново.

- Иначе создаём коммит:

```
git commit -m "Комментарий в одну строку"  
git commit
```

(если комментарий из нескольких строк)

Разрешение конфликтов при слиянии ветвей (2)

- Если произошёл конфликт, то создание коммита-слияния прерывается.
- В файлах с конфликтами добавляются маркеры слияния, выделяющие конфликтующие правки:

```
<<<<<<< HEAD  
<текст из текущей ветки>  
=====  
<текст из другой ветки>  
>>>>>>> <имя другой ветки>
```

- Эти маркеры нужно вручную заменить на корректный текст и добавить конфликтные файлы в индекс:

```
git add <файл-с-конфликтом>
```

- После чего можно завершить создание коммита-слияния, выполнив команду

```
git commit
```

Удалённые репозитории (1)

- Ну, мы клонировали репозиторий при помощи `git clone`. А что дальше? Как обмениваться изменениями?
- Репозиторий `git` может знать об одном или нескольких удалённых репозиториях — адресах других репозиториях, с которыми можно обмениваться коммитами.
- При клонировании по умолчанию инициализируется единственный удалённый репозиторий с именем `origin` — ссылка на источник при клонировании.

Удалённые репозитории (2)

- Ветки удалённых репозиториях отображаются с префиксом имени репозитория, например, `origin/main`.
- Для локальной ветки можно задать её `upstream` — соответствующую удалённую ветку, с которой она будет синхронизироваться.
- Основные команды при работе с удалёнными репозиториями:
 - `git push` — отправить новые коммиты из текущей ветки в ветвь удалённого репозитория,
 - `git fetch` — вытянуть из удалённого репозитория все обновления ветвей `origin/...`
 - `git pull = git fetch + git merge @{u}`.

Удалённые репозитории (3)

- Задать upstream для ветки test:

```
git branch test -u origin/test
```

- Задать upstream одновременно с отправкой изменений:

```
git push -u origin test
```

- После того, как upstream задан, можно использовать короткую команду

```
git push
```

и команду

```
git pull
```

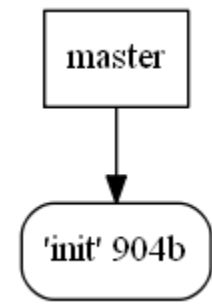
для получения изменений.

Пример работы с удалёнными ветвями

Алиса

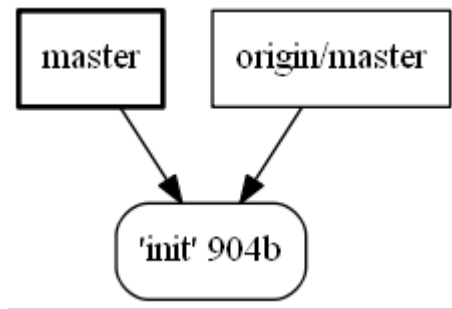
Сервер

Боб

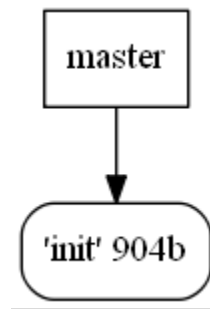


Пример работы с удалёнными ветвями

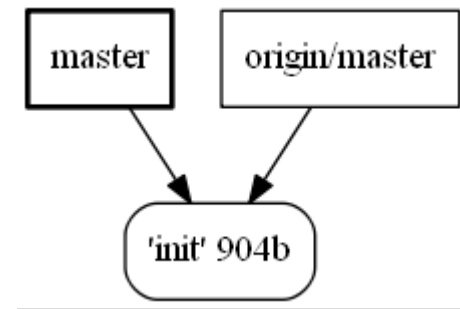
Алиса



Сервер



Боб

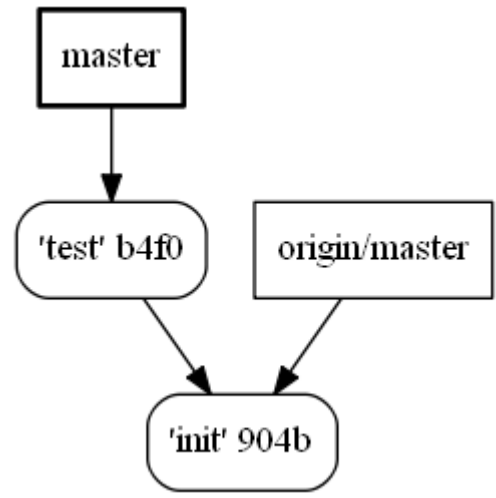


```
git clone server.git
```

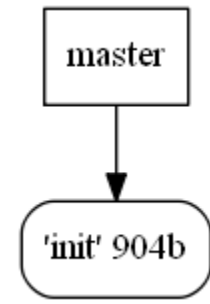
```
git clone server.git
```

Пример работы с удалёнными ветвями

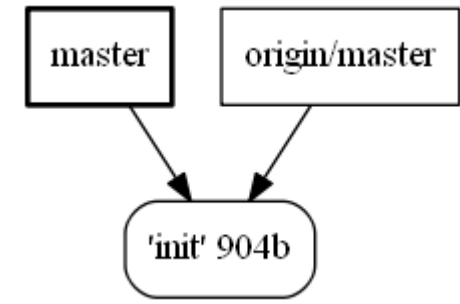
Алиса



Сервер



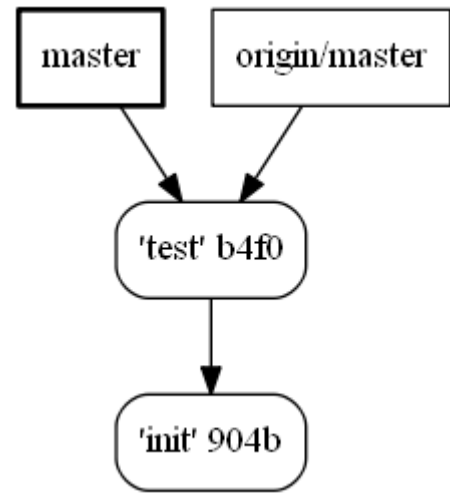
Боб



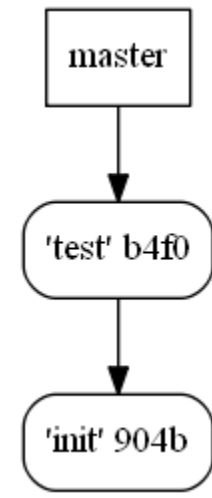
```
git commit -m "test"
```


Пример работы с удалёнными ветвями

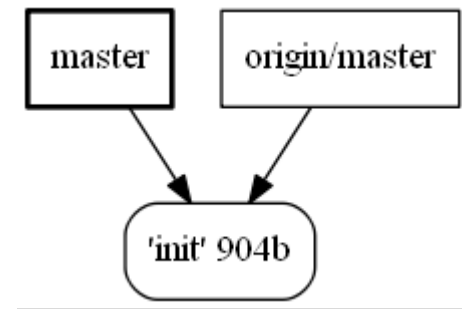
Алиса



Сервер



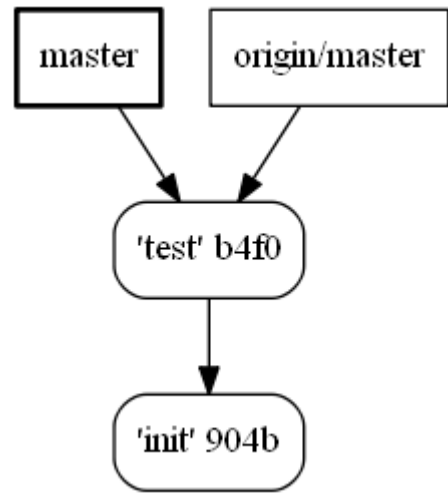
Боб



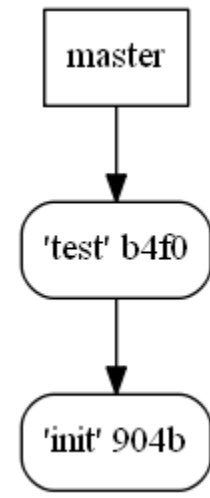
`git push`

Пример работы с удалёнными ветвями

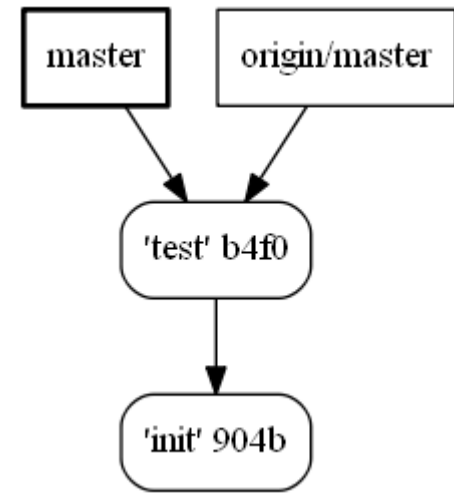
Алиса



Сервер

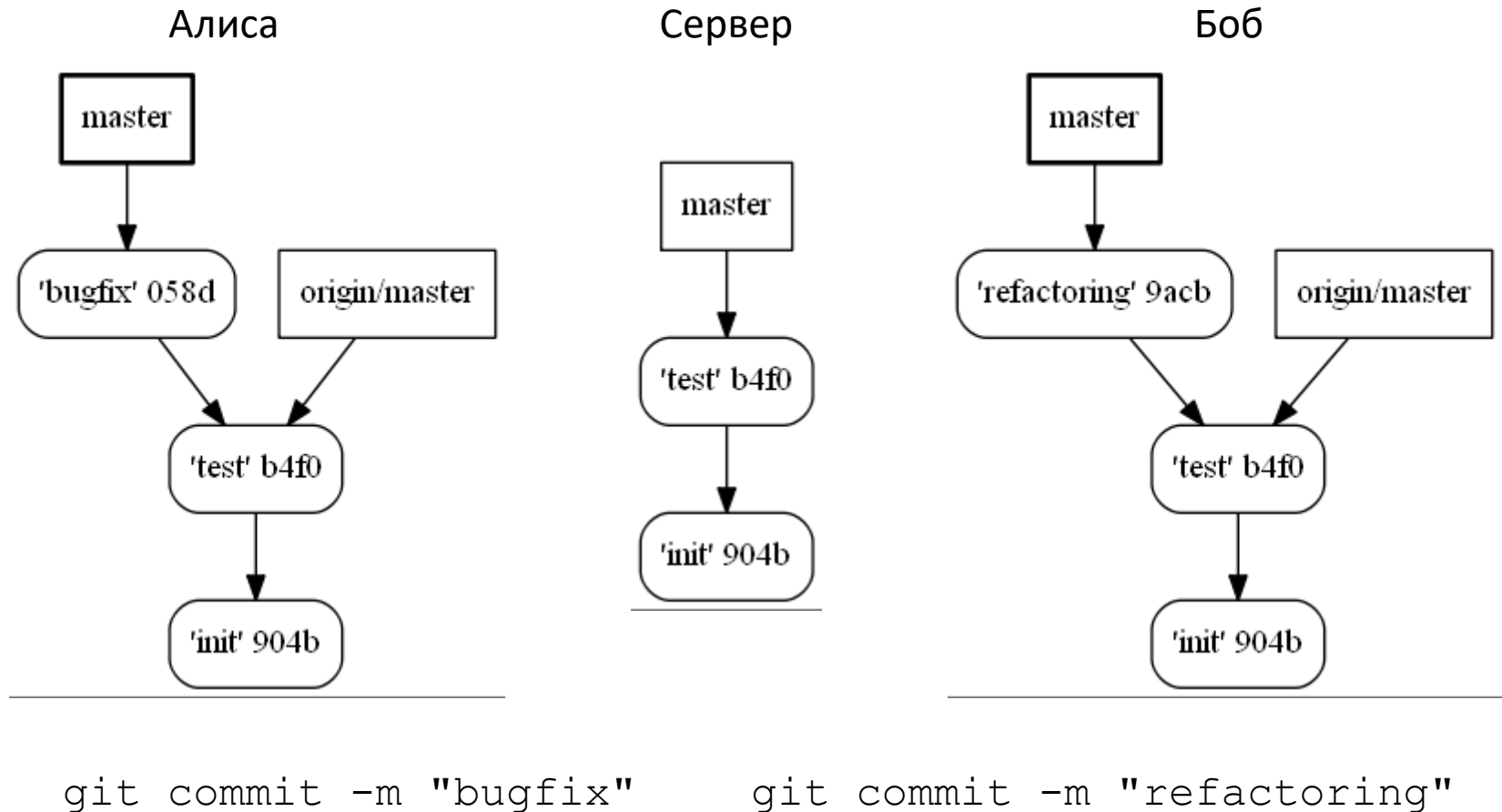


Боб

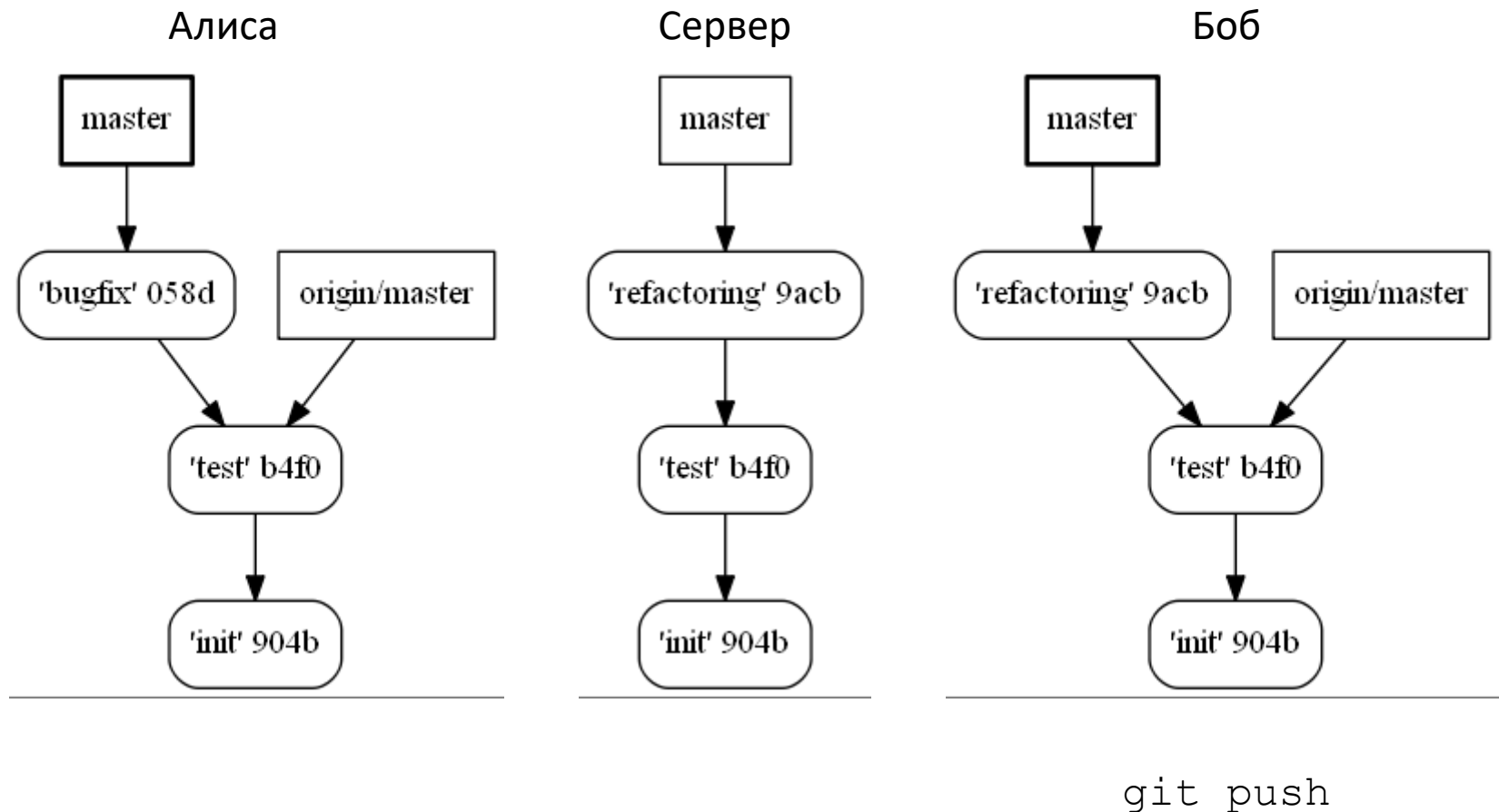


`git pull`

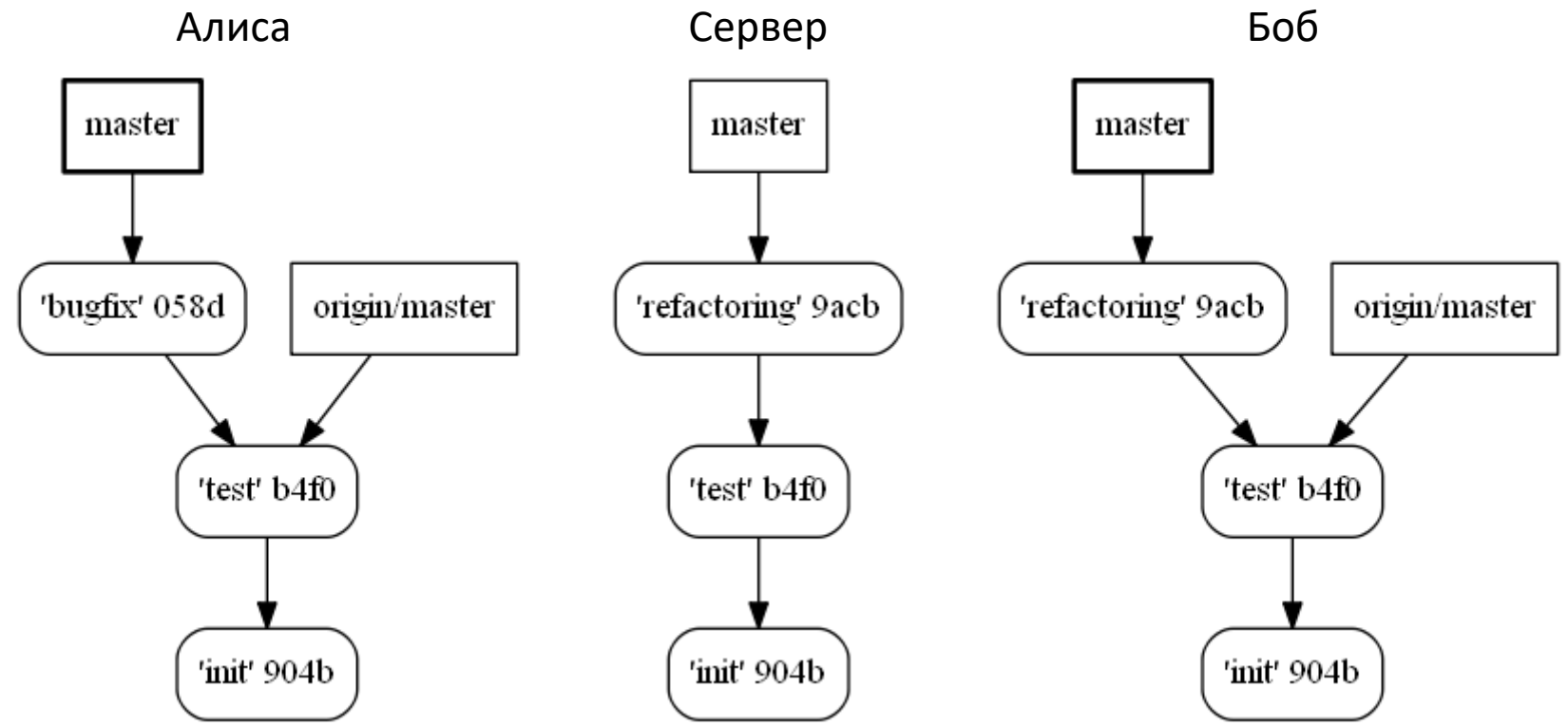
Пример работы с удалёнными ветвями



Пример работы с удалёнными ветвями

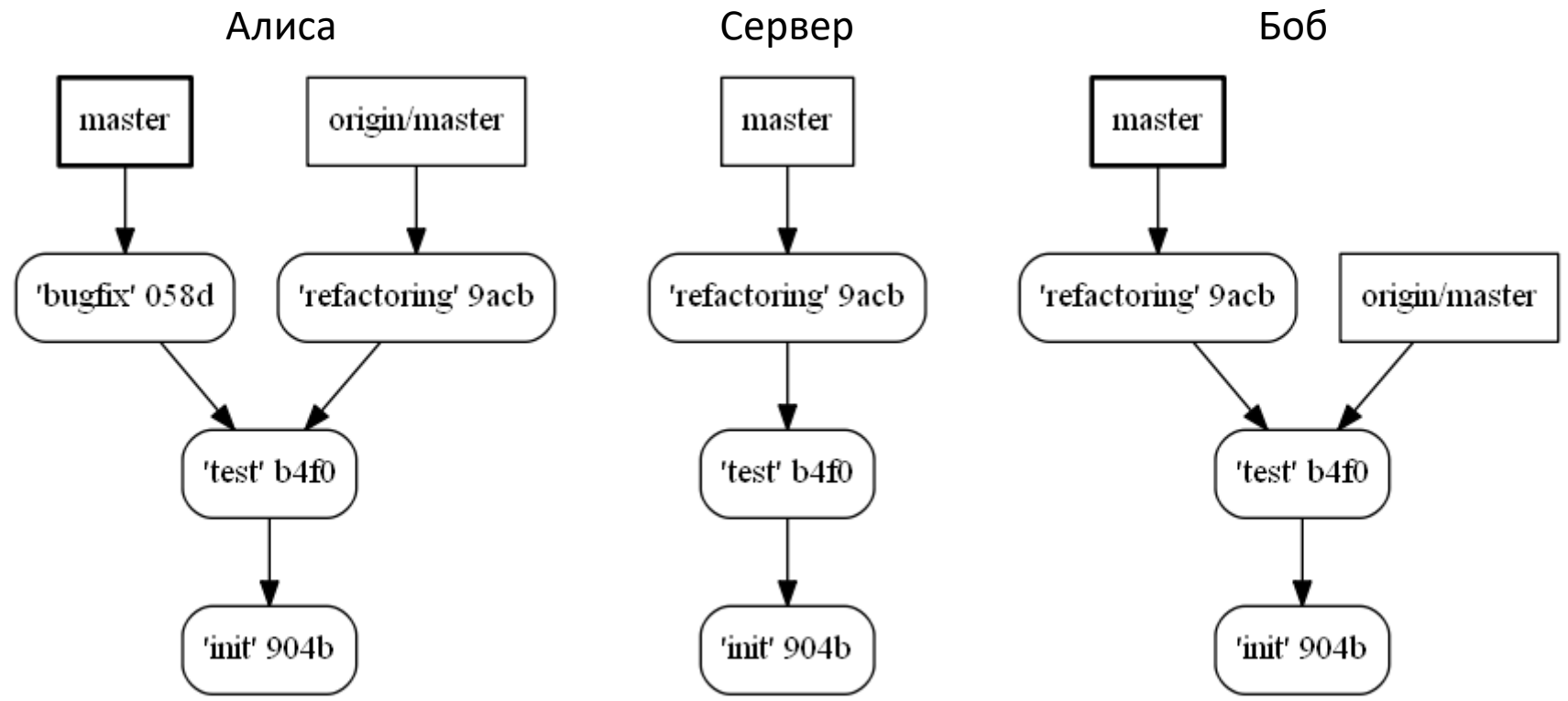


Пример работы с удалёнными ветвями



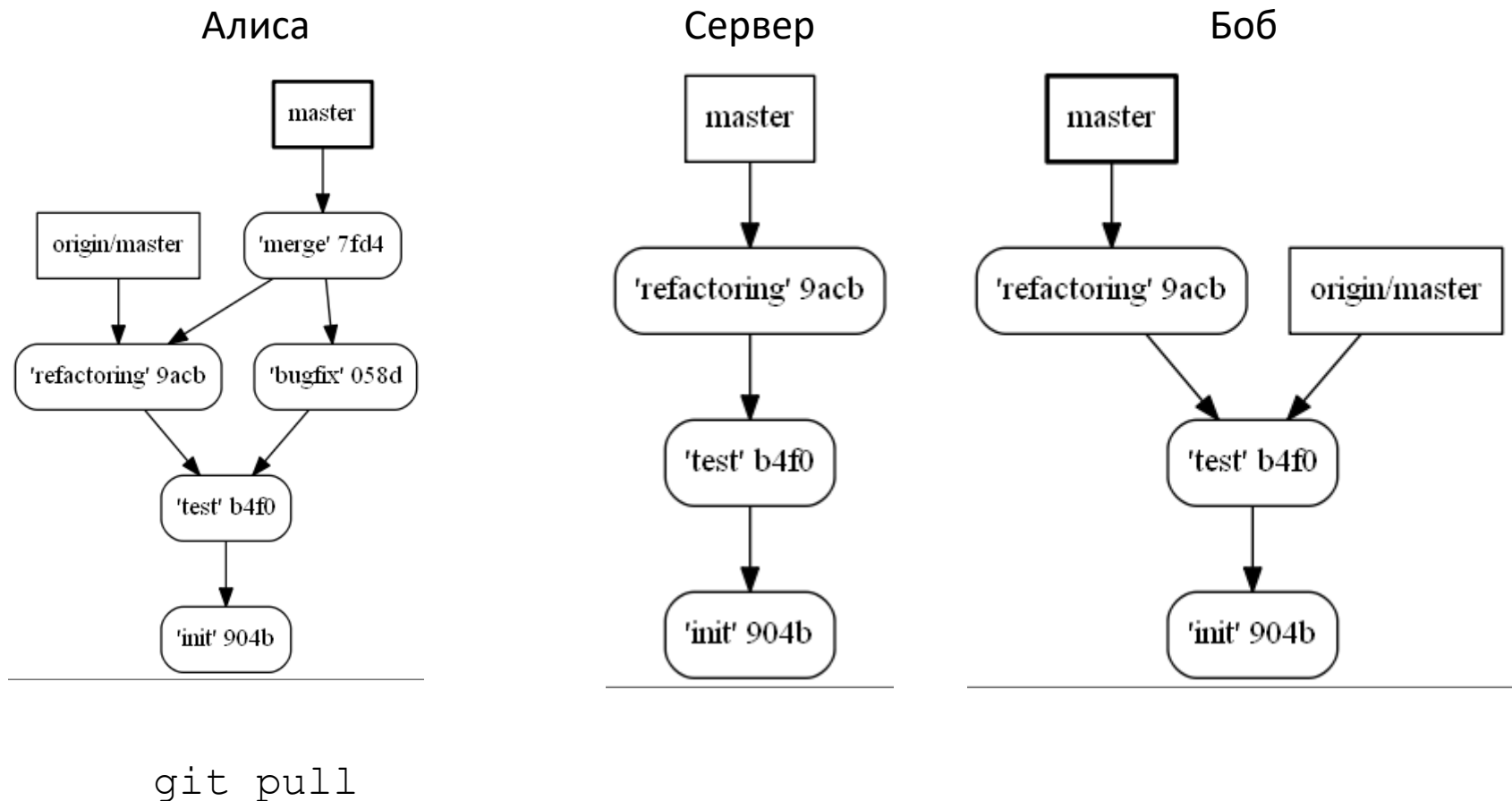
git push
! [rejected]

Пример работы с удалёнными ветвями



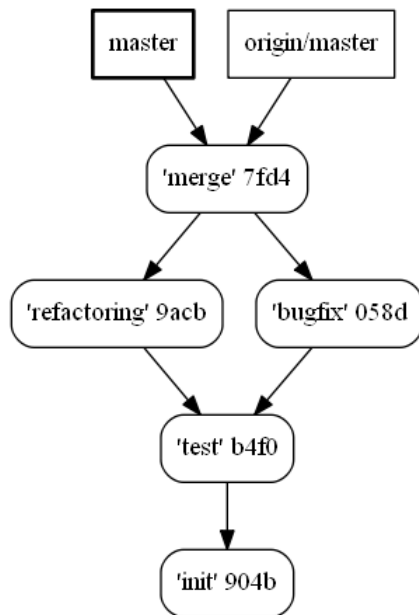
`git fetch`

Пример работы с удалёнными ветвями

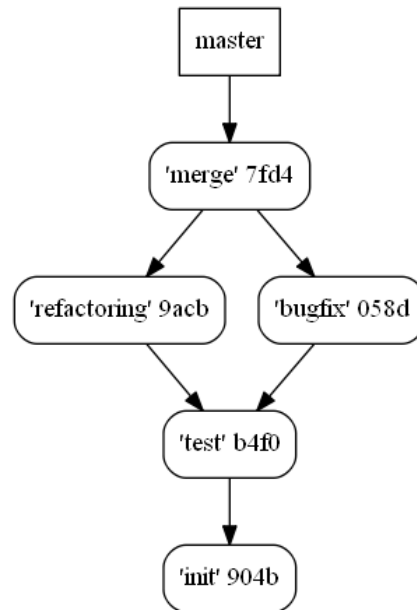


Пример работы с удалёнными ветвями

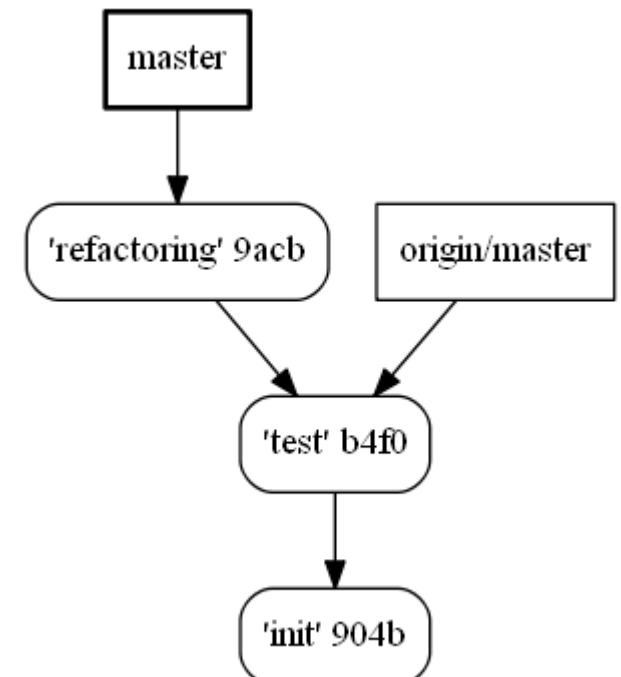
Алиса



Сервер



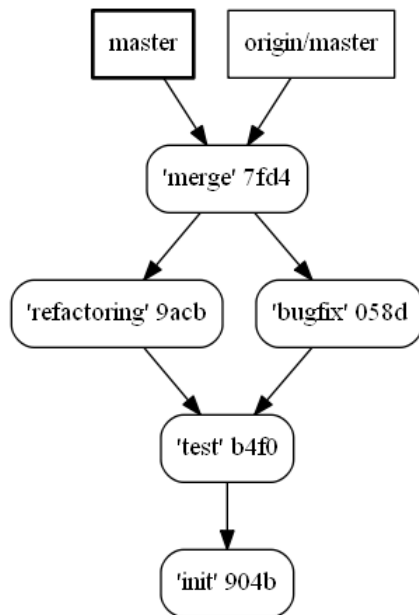
Боб



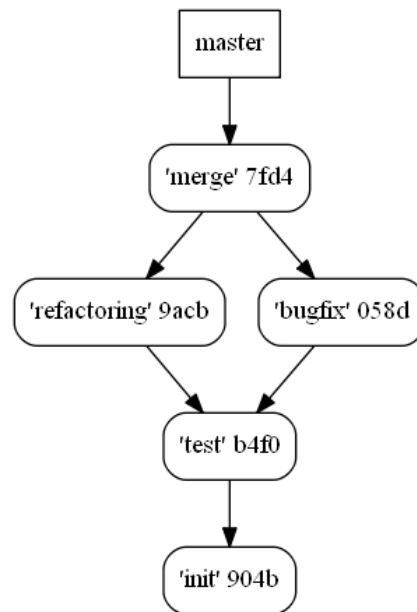
`git push`

Пример работы с удалёнными ветвями

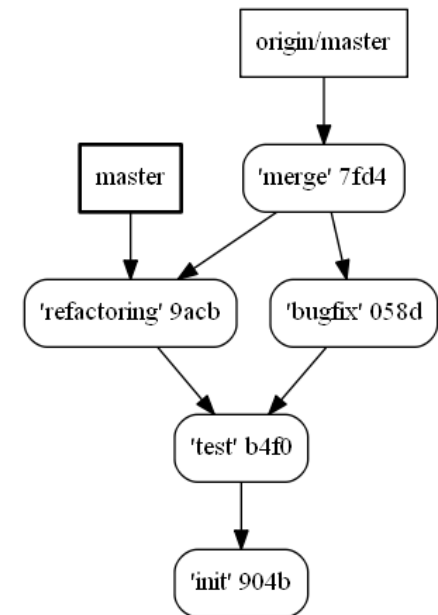
Алиса



Сервер



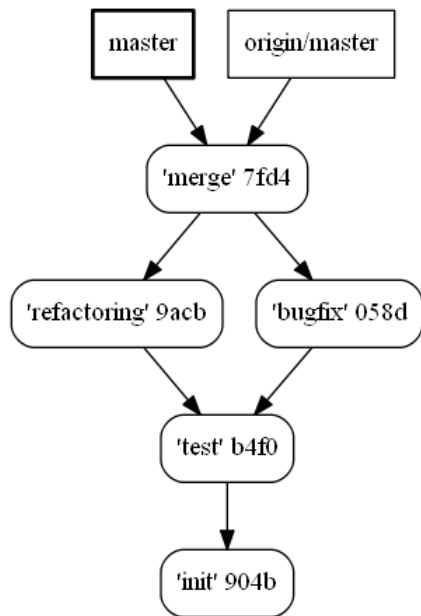
Боб



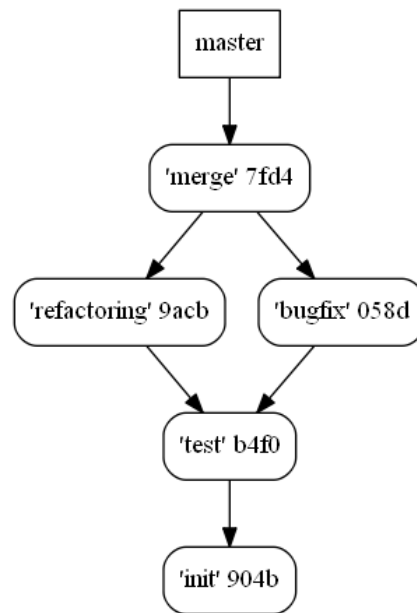
git fetch

Пример работы с удалёнными ветвями

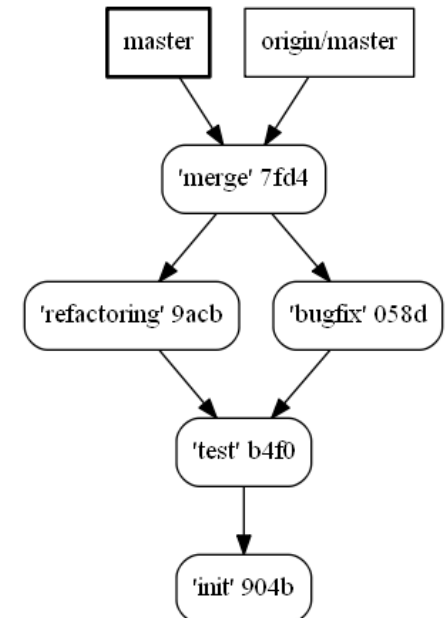
Алиса



Сервер



Боб



`git pull`

Подключение и отключение удалённых репозиториев

- Удалённых репозиториев может быть несколько.
- `git clone https://github.com/bmstu-iu9/utp2020-xxx`

эта команда создаст репозиторий с одним удалённым — `origin`.

- Можно добавлять и другие удалённые репозитории:

```
git remote add vasya https://github.com/vasya/utp2020-xxx
git remote add masha https://github.com/masha/utp2020-xxx
```

- Имена веток в этих репозиториях будут `vasya/main`, `masha/test` и т.д.
- Для запроса коммитов из всех репозиториев используется команда

```
git fetch --all
```

- Удаление удалённых репозиториев:

```
git remote rm vasya
git remote rm masha
```

Шпаргалка: как создать новую тематическую ветку

- Переключаемся на ветку main и обновляем её:

```
git checkout main  
git pull
```

- Создаём новую ветку:

```
git checkout -b hero-rendering
```

- Отправляем ветку на сервер:

```
git push -u origin hero-rendering
```

(ключик -u настраивает upstream)

Шпаргалка: как влить изменения из main в свою ветку

- Переключаемся на ветку main и обновляем её:

```
git checkout main  
git pull
```

- Переключаемся на свою ветку:

```
git checkout hero-rendering
```

- Вливаем обновления из main:

```
git merge main
```

- Отправляем изменения на сервер:

```
git push
```

Рабочий процесс с Git

- СУВ Git очень гибкая, поэтому существует много вариантов рабочих процессов (workflow).
- Некоторые из них:
 - Придерживаться линейной истории.
Если ветка origin/main «уехала вперёд», нужно все свои ветки пересадить (git rebase) на неё.
 - GitFlow
<https://habr.com/ru/post/106912>
Хорошо известный, но очень забюрократизированный процесс.
 - GitHub Flow
<https://habr.com/ru/post/189046>
Рекомендуемый стиль разработки для GitHub.

Рабочий процесс GitHub Flow (1)

- Принципы:
 - Содержимое ветки `main` всегда работоспособно.
 - Для новой задачи ветвь ответвляется от `main`.
 - Постоянно `push`'те изменения.
 - Во-первых, сервер непрерывной интеграции будет их тестировать,
 - во-вторых, другим разработчикам будет видно, чем вы занимаетесь.
 - Когда задача завершена или нужны комментарии коллег — открывается `pull request` на вливание ветки в `main`.
 - После ревью ветка вливается в `main` и выкатывается на продакшен.

Рабочий процесс GitHub Flow (2)

- Pull request — запрос на вливание одной ветки в другую через web-интерфейс GitHub.
- Pull request позволяет вести обсуждение ветки (комментарии), а также производить ревью кода.
- Репозиторий может быть настроен таким образом, чтобы без одобряющего ревью запрещать слияние.

Как будет выглядеть ваша работа на практике

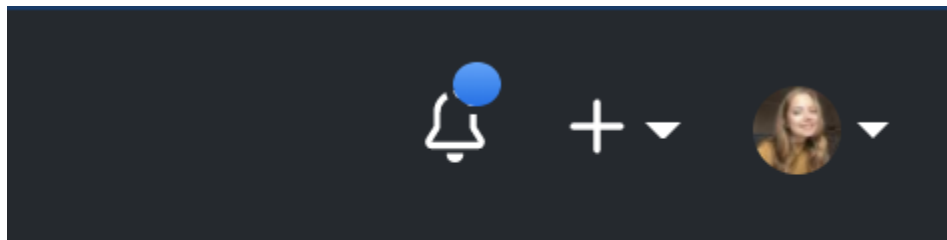
- Для каждой из команд будет создан репозиторий с именем вида

`https://github.com/bmstu-iu9/ptp2021-N-name`

- Ветка `main` в репозитории будет защищённой — `git push` в неё будет запрещён.
- Единственный способ поместить код в `main` будет `pull request`.
- `Pull request`'ы в `main` будут требовать одобрительного ревью.
- При оценке вашей работы мы будем смотреть только содержимое ветки `main` репозитория.

Получение доступа к репозиторию

- Вы регистрируетесь на GitHub и присылаете мне ник.
- Я приглашаю Вас в команду UTP-2020.
- Вы должны принять приглашение.
- Приглашение придёт или на почту, или в область уведомлений



Получение доступа к репозиторию (письмо)

[GitHub] @Mazdaywik has invited you to join the @bmstu-iu9 organization

GitHub Сегодня, 18:24
Кому: вам



GitHub



@Mazdaywik has invited you to join the
@bmstu-iu9 organization

Hi **Kuznecova Nastya!**

@Mazdaywik has invited you to join the @bmstu-iu9 organization on GitHub. Head over to <https://github.com/bmstu-iu9> to check out @bmstu-iu9's profile.

This invitation will expire in 7 days.

[Join @bmstu-iu9](#)

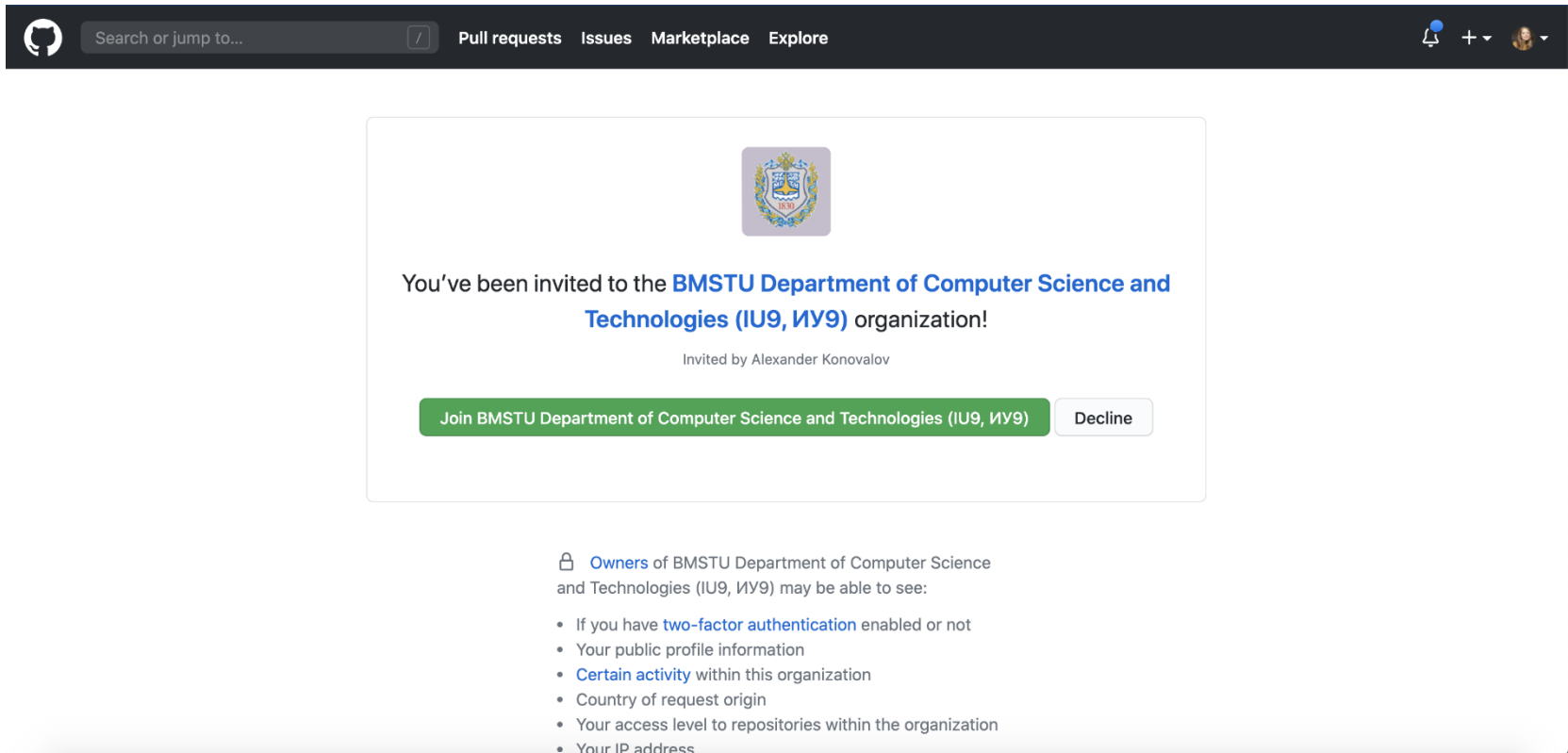
Note: If you get a 404 page, make sure you're signed in as **suova**. You can also accept the invitation by visiting the organization page directly at <https://github.com/bmstu-iu9>. If @Mazdaywik is sending you too many emails, you can [block them](#) or [report them for abuse](#).

Button not working? Paste the following link into your browser:
https://github.com/orgs/bmstu-iu9/invitation?via_email=1

You're receiving this email because @Mazdaywik invited you to an organization on GitHub.


[Opt out of future invitations from this organization.](#)
[Manage your GitHub email preferences](#)

Получение доступа к репозиторию (сайт)



The screenshot shows the GitHub interface. At the top is a dark navigation bar with the GitHub logo, a search bar, and links for Pull requests, Issues, Marketplace, and Explore. On the right of the bar are notification and user profile icons. The main content area features a white card with a blue border. At the top of the card is the BMSTU logo. Below it, the text reads: "You've been invited to the **BMSTU Department of Computer Science and Technologies (IU9, IU9)** organization!". Underneath, it says "Invited by Alexander Konovalov". At the bottom of the card are two buttons: a green "Join BMSTU Department of Computer Science and Technologies (IU9, IU9)" button and a white "Decline" button. Below the card, there is a section titled "Owners of BMSTU Department of Computer Science and Technologies (IU9, IU9) may be able to see:" followed by a list of permissions.


Search or jump to... / Pull requests Issues Marketplace Explore



You've been invited to the **BMSTU Department of Computer Science and Technologies (IU9, IU9)** organization!

Invited by Alexander Konovalov

[Join BMSTU Department of Computer Science and Technologies \(IU9, IU9\)](#) [Decline](#)

 **Owners** of BMSTU Department of Computer Science and Technologies (IU9, IU9) may be able to see:

- If you have **two-factor authentication** enabled or not
- Your public profile information
- **Certain activity** within this organization
- Country of request origin
- Your access level to repositories within the organization
- Your IP address