

# Современные технологии разработки ПО

## Командная разработка ПО

Кафедра ИС, Петров А.А. [petrov.a@kubsau.ru](mailto:petrov.a@kubsau.ru)

# Содержание

- Системы управления версиями
- Git
- Внешние репозитории кода, GitHub/GitLab
- CI/CD и цикл разработки
- Примеры

# Системы управления версиями\*

- Version Control System, VCS
- Программное обеспечение, обеспечивающее версионирование текстовых документов в многопользовательской среде
- Позволяет:
  - хранить несколько версий одного и того же текстового документа;
  - возвращаться к более ранним версиям;
  - определять, кто и когда сделал то или иное изменение;
  - и многое другое.

# Централизованные системы управления версиями\*

- Репозиторий расположен на сервере
- Централизованное управление
- Управление коллизиями: блокировка файла
- Дополнительно:
  - поддержка ветвей;
  - пометка версий тегами.
- Цикл работы:
  - рабочий каталог,
  - checkout,
  - checkin.
- Примеры централизованных систем:
  - Concurrent Versions System (CVS), Subversion, Mercurial;
  - Team Foundation Server (TFS), Visual Studio Team System (VSTS), Azure DevOps Server (с 2019 года).

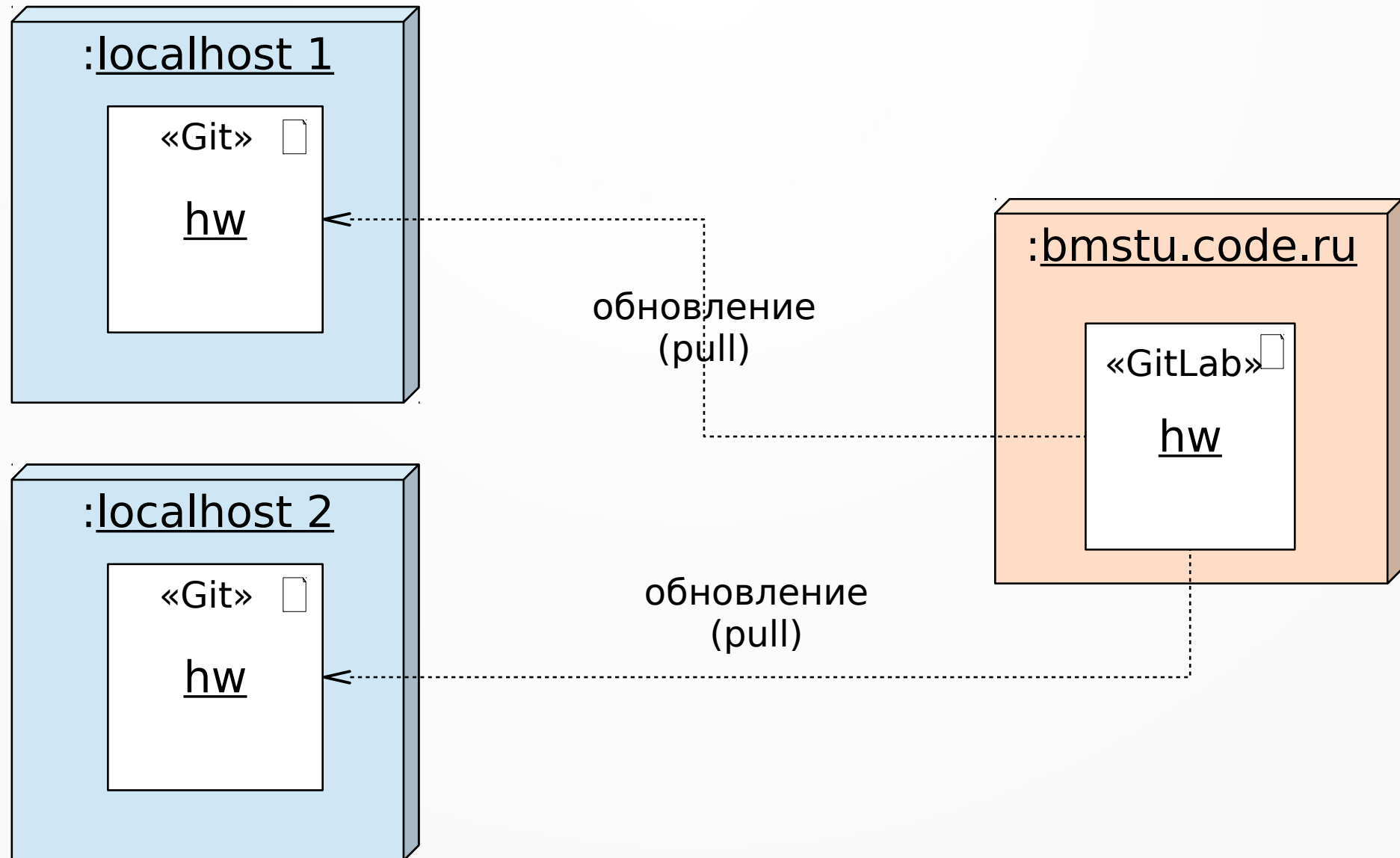
# Распределённые системы управления версиями\*

- Distributed Version Control System, DVCS
- Вся история изменений храниться в локальном репозитории на компьютере разработчика
- Фрагменты истории локального хранилища синхронизируются с аналогичным хранилищем на другом компьютере
- Управление коллизиями:
  - слияние различий (merge),
  - ручное решение конфликтов.

# Git\*

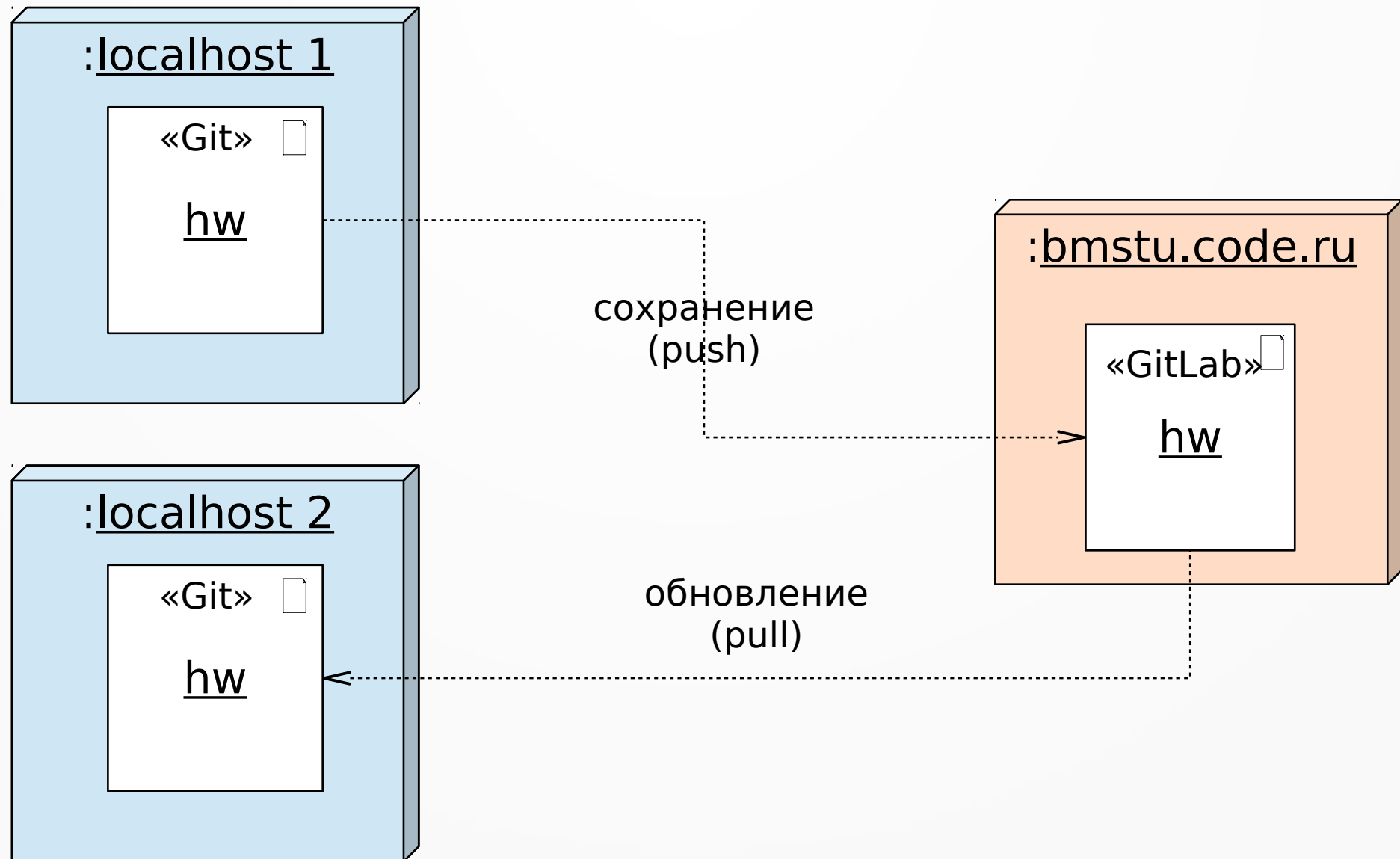
- Распределённая система управления версиями исходного кода ПО
- Возможности:
  - распределённая:
    - предоставляет каждому разработчику локальную историю разработки,
    - не требует захватывать файлы для выполнения изменений;
  - быстрое разделение и слияние версий;
  - предоставляет несколько протоколов для доступа к удалённому репозиторию (GitHub, GitLab и пр.).
- Аналоги:
  - Bazaar (bzd), Launchpad.

# Git. Обновление проекта из удалённого репозитория\*



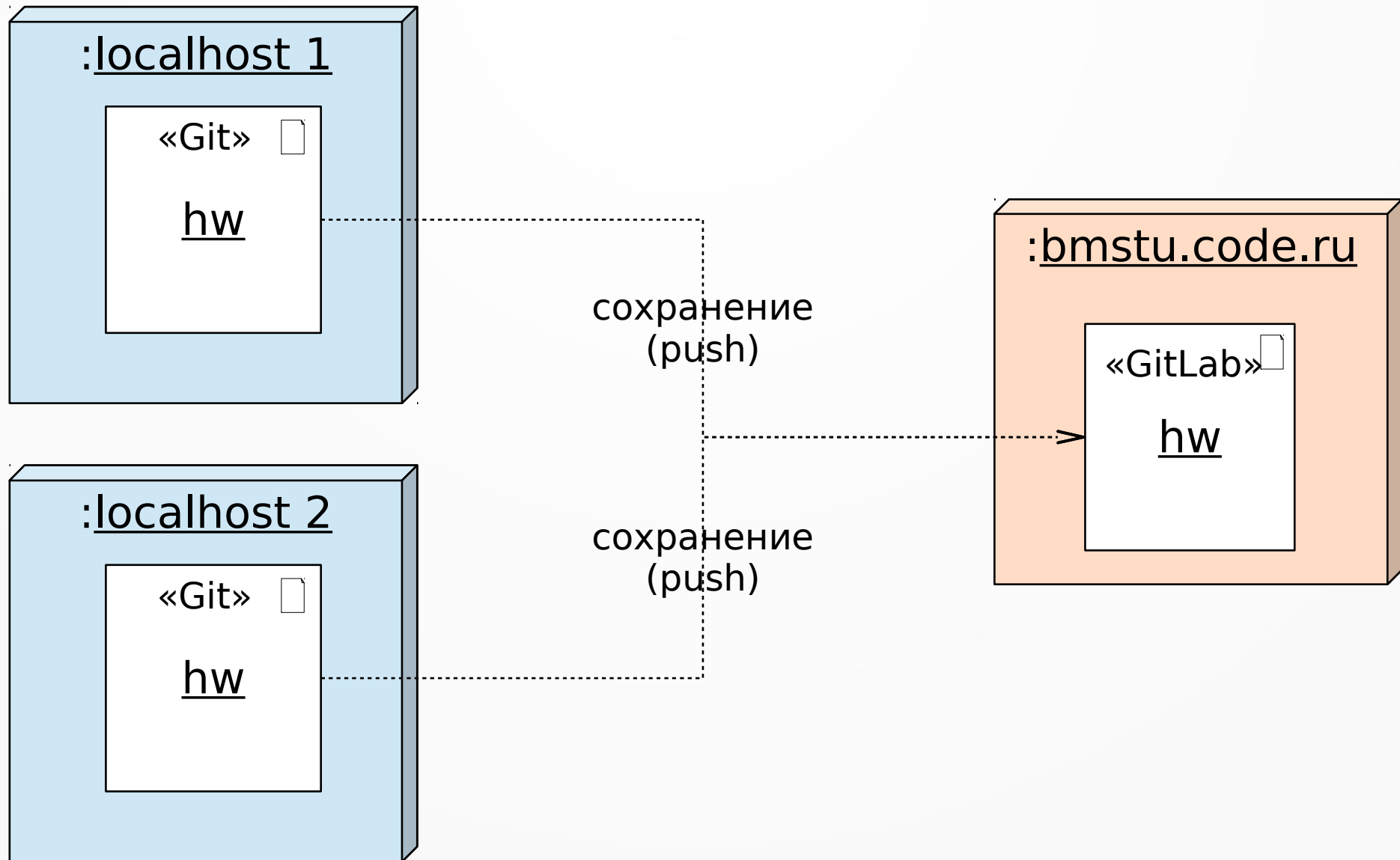


# Git. Обновление проекта из удалённого репозитория (вариант 2)\*





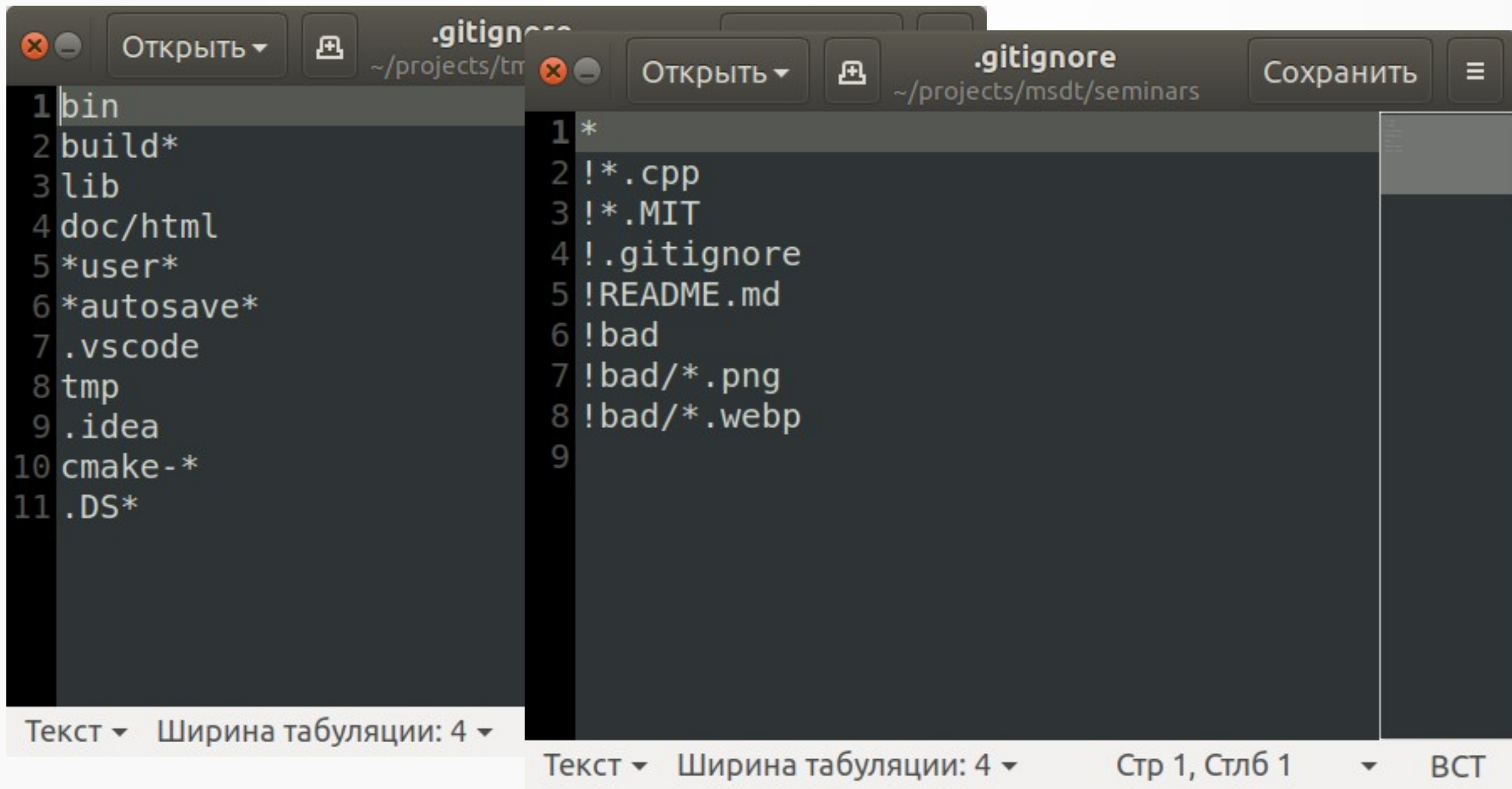
# Git. Сохранение проекта в удалённый репозиторий\*



# Git. Создание проекта, конфигурация\*

- `git help`
- `git init`
- `git config --local user.name "<Имя Фамилия>"`
- `git config --local user.email "<адрес эл. Почты>"`
- `git config --local -l`
- Области конфигураций:
  - `local`
  - `global`
  - `system`

# .gitignore – список исключений\*



```
1 bin
2 build*
3 lib
4 doc/html
5 *user*
6 *autosave*
7 .vscode
8 tmp
9 .idea
10 cmake-*
11 .DS*
```

```
1 *
2 !*.cpp
3 !*.MIT
4 !.gitignore
5 !README.md
6 !bad
7 !bad/*.png
8 !bad/*.webp
9
```

Текст ▾ Ширина табуляции: 4 ▾

Текст ▾ Ширина табуляции: 4 ▾ Стр 1, Стлб 1 ▾ ВСТ

# Git. Копирование проекта\*

- Команда:

```
git clone <адрес проекта>
```

- Пример:

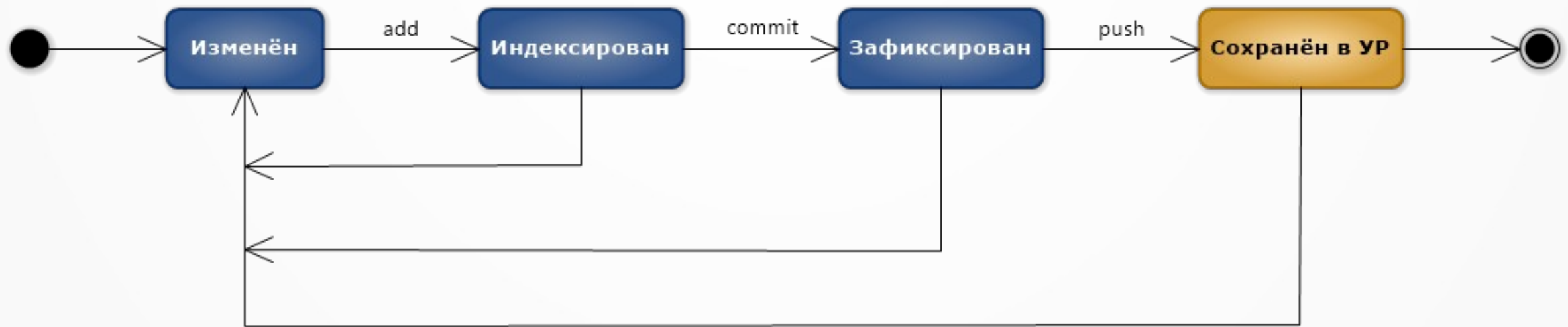
```
git clone git@bmstu.codes:lsx/msdt-study-group/homework/detach-2.git
```

```
git clone https://bmstu.codes/lsx/msdt-study-group/homework/detach-2.git
```

# Git. Сохранение изменений в проекте\*

- Индексация файла(ов):  
git add <имя файла>  
git add .
- Фиксация изменений:  
git commit -m "<описание изменения>"
- Сохранение изменений в удалённом репозитории:  
git push

# Git. Этапы сохранения изменений файла\*





# Git. Удалённый репозиторий (УР)\*

- Подключение к УР:  
git remote add origin <адрес>
- Получение проекта из УР:  
git pull  
git clone <адрес>
- Сохранение изменений в УР:  
git push -u origin master  
git push



# Git. Просмотр изменений, удаление\*

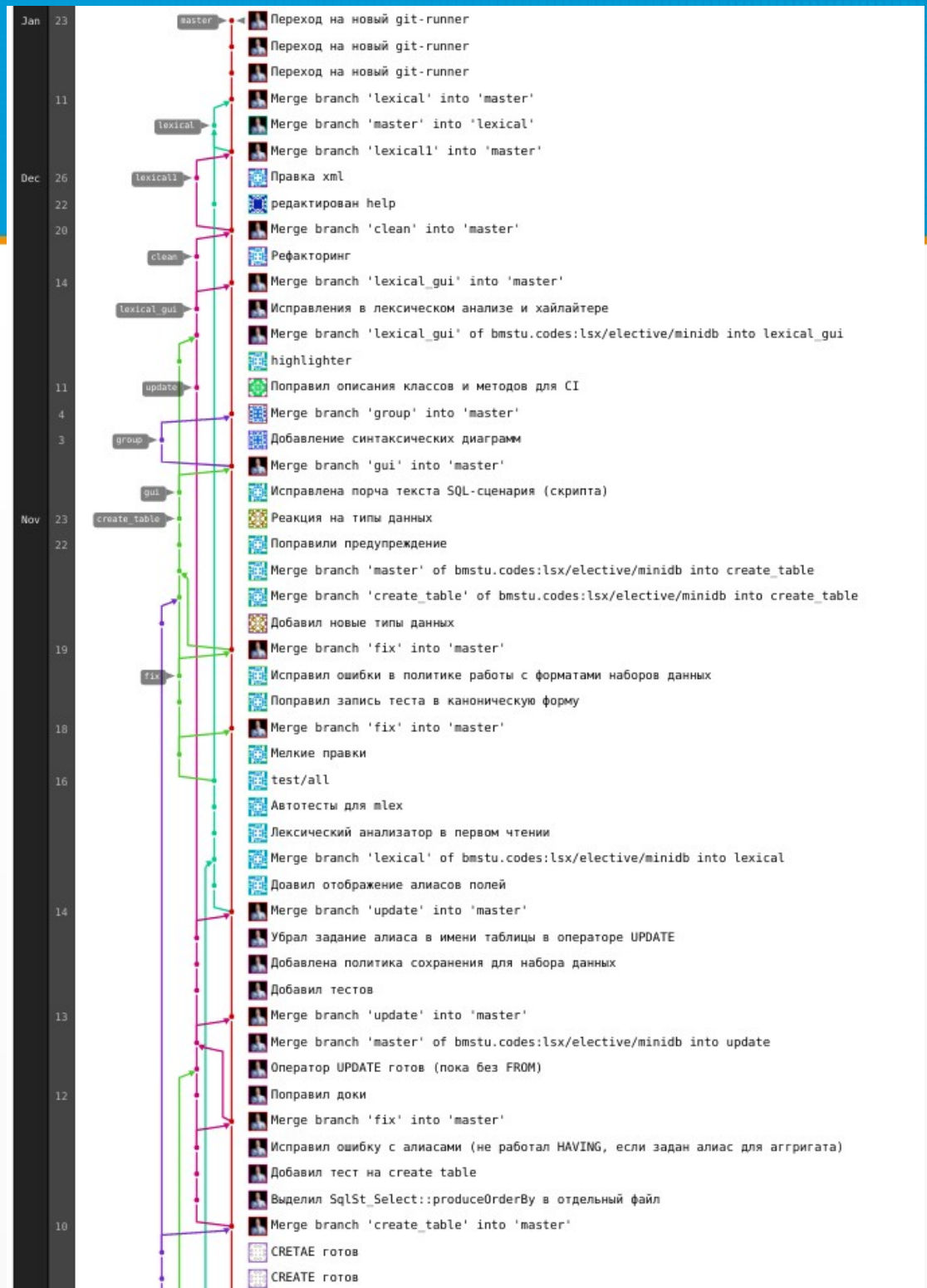
- Изменения на уровне списка файлов:
  - `git status`
- Сравнение файлов:
  - `git diff <имя файла>`
- Удаление:
  - `git rm`
  - `git rm -f`

# Git. Ветвление\*

- Создание ветки:
  - `git branch <имя ветки>`
  - `git checkout <имя ветки>`
  - `git checkout -b <имя ветки>`
- Просмотр веток:
  - `git branch`
- Слияние веток:
  - `git checkout master`
  - `git merge <имя ветки>`

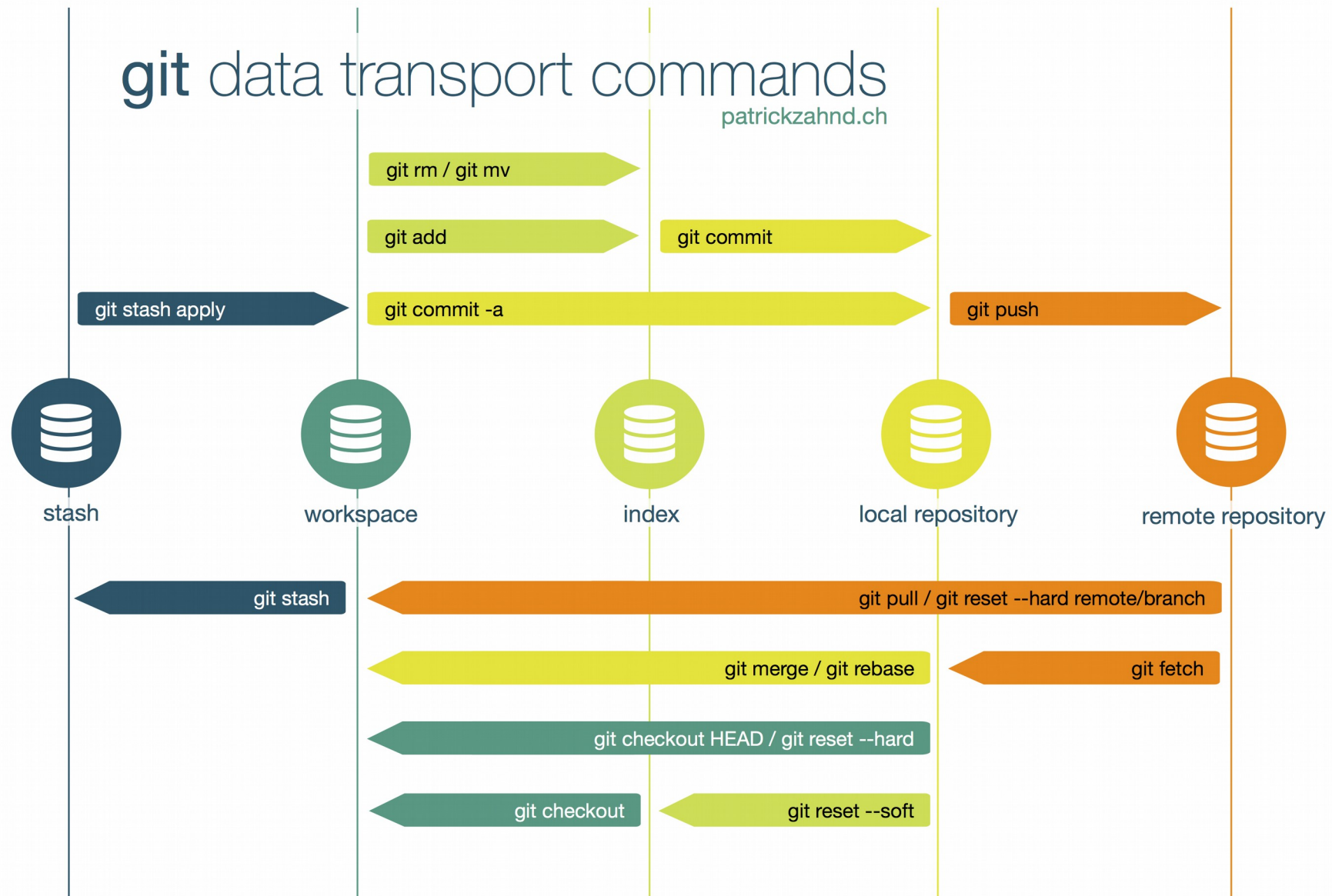
# Git. Ветвление

## Пример



# Git

## Cxema 1



# Git Cheat Sheet

by Jan Krüger <jk@jk.gs>, <http://jan-krueger.net/git/>  
Based on work by Zack Rusin

## Basics

Use `git help [command]` if you're stuck.

master	default devel branch
origin	default upstream branch
HEAD	current branch
HEAD^	parent of HEAD
HEAD~4	great-great grandparent of HEAD
foo..bar	from branch <i>foo</i> to branch <i>bar</i>

## Create

### From existing files

```
git init
git add .
```

### From existing repository

```
git clone ~/old ~/new
git clone git://...
git clone ssh://...
```

## View

```
git status
git diff [oldid newid]
git log [-p] [file|dir]
git blame file
git show id (meta data + diff)
git show id:file
git branch (shows list, * = current)
git tag -l (shows list)
```

## Revert

In Git, revert usually describes a new commit that undoes previous commits.

```
git reset --hard (NO UNDO)
(revert to last commit)
git revert branch
git commit -a --amend
(replaces prev. commit)
git checkout id file
```

## Publish

In Git, `commit` only respects changes that have been marked explicitly with `add`.

```
git commit [-a]
(-a: add changed files
automatically)
git format-patch origin
(create set of diffs)
git push remote
(push to origin or remote)
git tag foo
(mark current version)
```

## Update

```
git fetch (from def. upstream)
git fetch remote
git pull (= fetch & merge)
git am -3 patch.mbox
git apply patch.diff
```

## Branch

```
git checkout branch
(switch working dir to branch)
git merge branch
(merge into current)
git branch branch
(branch current)
git checkout -b new other
(branch new from other and
switch to it)
```

## Conflicts

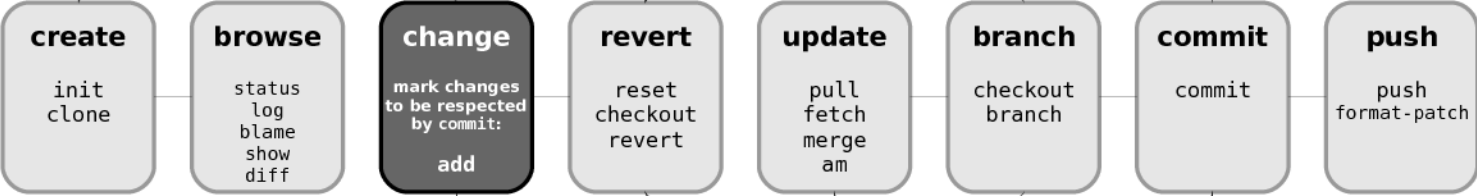
Use `add` to mark files as resolved.

```
git diff [--base]
git diff --ours
git diff --theirs
git log --merge
gitk --merge
```

## Useful Tools

```
git archive
Create release tarball
git bisect
Binary search for defects
git cherry-pick
Take single commit from elsewhere
git fsck
Check tree
git gc
Compress metadata (performance)
git rebase
Forward-port local changes to
remote branch
git remote add URL
Register a new remote repository
for this tree
git stash
Temporarily set aside changes
git tag
(there's more to it)
gitk
Tk GUI for Git
```

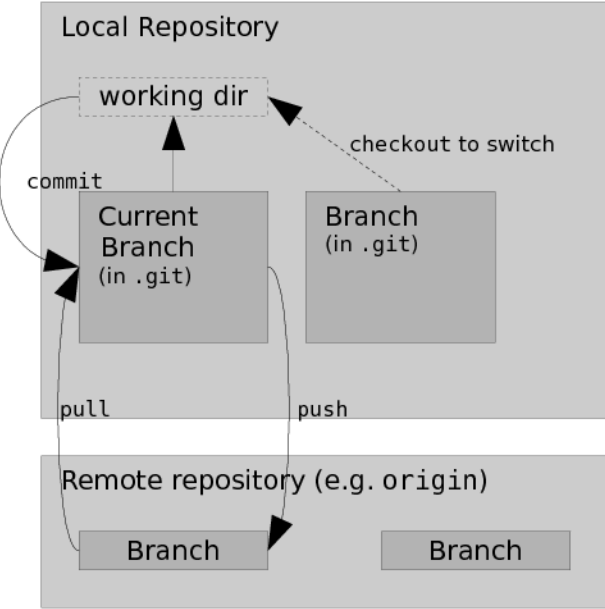
## (left to right) Command Flow



## Tracking Files

```
git add files
git mv old new
git rm files
git rm --cached files
(stop tracking but keep files in working dir)
```

## Structure Overview





# Git Cheat Sheet

git <COMMAND> --help

git config --help

## ★ Create

```
cd ~/my_project_directory
git init
git add .
```

```
git clone ~/existing_repo ~/new/repo
git clone git://host.org/project.git
git clone ssh://user@host.org/project.git
```

## ★ Show

```
git status
```

```
git diff
```

```
git diff <ID1> <ID2>
```

```
git log
```

```
git log -p <FILE> <DIRECTORY>
```

```
git blame <FILE>
```

```
git show <ID>
```

```
git show <ID>:<FILE>
```

```
git branch
star (*) marks the current branch
```

## ★ Revert

```
git reset --hard
This cannot be undone!
```

```
git revert HEAD
Creates a new commit
```

```
git revert <ID>
Creates a new commit
```

```
git commit -a --amend
(after editing the broken files)
```

```
git checkout <ID> <FILE>
```

## ★ Update

```
git fetch
(this does not merge them)
```

```
git pull
(does a fetch followed by a merge)
```

```
git am -3 patch.mbox
In case of conflict, resolve the conflict and
git am --resolved
```

## ★ Publish

```
git commit -a
```

```
git format-patch origin
```

```
git push
```

```
git tag v1.0
```

## ★ Branch

```
git checkout <BRANCH>
```

```
git checkout <BRANCH2>
git merge <BRANCH1>
```

```
git branch <BRANCH>
```

```
git checkout -b <BRANCH> <OTHER>
```

```
git branch -d <BRANCH>
```

## ★ Resolve merge conflicts

```
git diff
```

```
git diff --base <FILE>
```

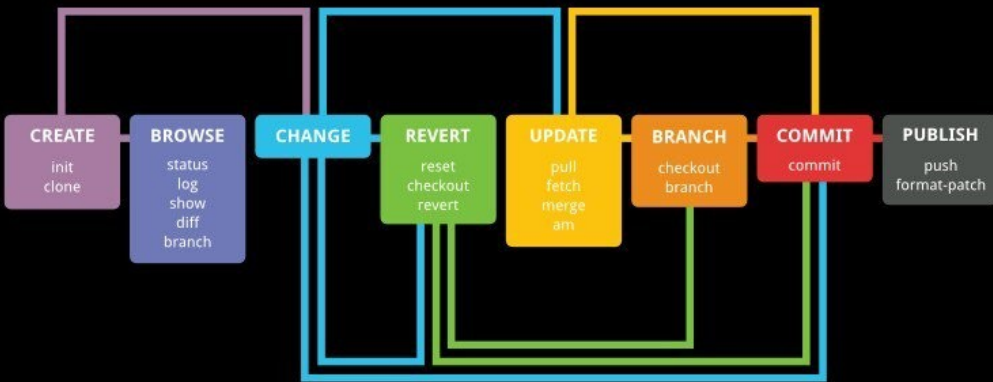
```
git diff --ours <FILE>
```

```
git diff --theirs <FILE>
```

```
git reset --hard
git rebase --skip
```

```
git add <CONFLICTING_FILE>
git rebase --continue
```

## ★ Workflow





## Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

## Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

## Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my\_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new\_branch

```
$ git branch new_branch
```

Delete the branch called my\_branch

```
$ git branch -d my_branch
```

Merge branch\_a into branch\_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

## Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

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

## Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

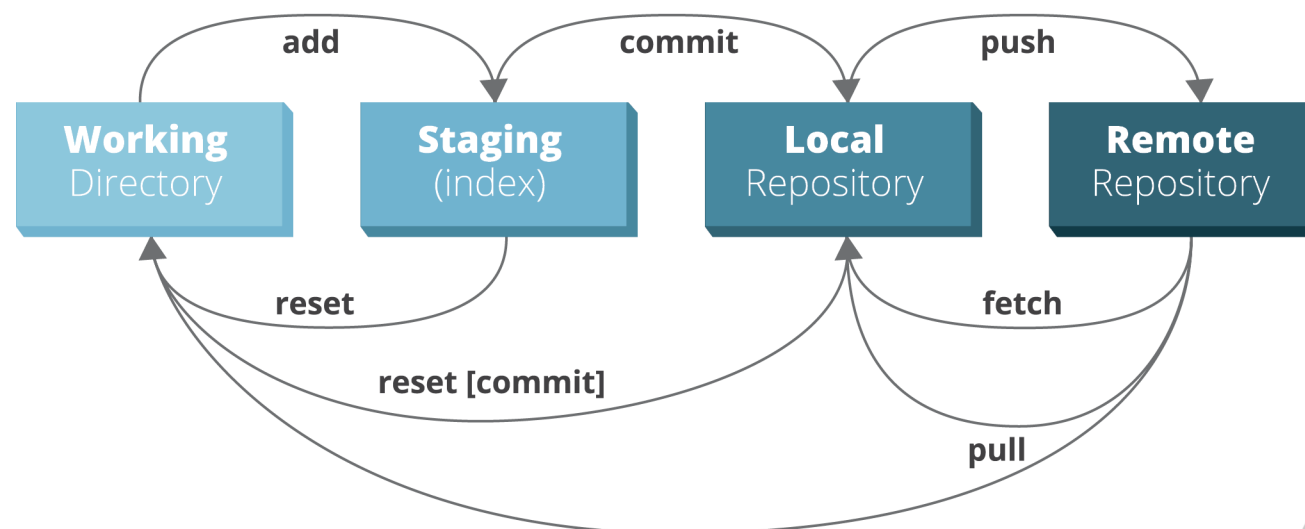
```
$ git push
```

## Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.





# GitLab\*

- Web-приложение и система управления репозиториями кода для Git
- Возможности:
  - организация публичных и частных репозиториев;
  - управление правами, группами;
  - импорт проектов, в том числе из GitHub;
  - вики;
  - API;
  - доска идей и задач;
  - лейблы, вехи, шаблоны, поиск;
  - комментирование, объединение;
  - свой CI/CD с поддержкой DevOps;
  - отслеживание изменений и прогресса;
  - отслеживание времени;
  - и др.

# Регистрация на <https://bmstu.codes>

- Регистрация
- Выслать никнейм (возможно не понадобится)

# Получение открытого ключа SSH

- Необходимо для работы с удалённым репозиторием без ввода пароля
- Проверка существования открытого ключа SSH:
  - каталог «.ssh»,
  - файл «id\_rsa.pub»;
  - содержимое этого файла и есть открытый ключ.
- Генерация открытого ключа SSH:
  - ssh-keygen

# CI/CD

## Continuous Integration\*

- Практика разработки программного обеспечения, которая заключается в:
  - постоянном слиянии рабочих копий в общую основную ветвь разработки (до нескольких раз в день) и
  - выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.
- Впервые концептуализирована и предложена Гради Бучем в 1991 году
- Является одним из основных элементов практики экстремального программирования
- Что было до?

# CI/CD

## Составляющие CI\*

- Выполняется специальной службой, как правило на отдельном сервере (runner):
  - получение исходного кода из репозитория;
  - сборка проекта;
  - выполнение тестов;
  - развёртывание готового проекта;
  - отправка отчетов.

# CI/CD

## Архитектура удалённого репозитория\*

- (рисует)

# CI/CD

## Пример скрипта (GitLab)

```
image: ubuntu:18.04

stages:
  - compile
  - test
  - pages

# Задаём команды, которые будут выполнены до прогона скриптов
before_script:
  - apt-get -y update

# Собираем проект
compile:
  stage: compile
  script:
    # Устанавливаем пакеты
    - apt-get -y install gcc g++
    - apt-get -y install make cmake
    # Создаём рабочие каталоги
    - mkdir bin
    - mkdir build
    - mkdir lib
    # Собираем
    - cmake -Bbuild/release -H.
    - make --directory=build/release
    # Сохраняем утилиты в артефактах
  artifacts:
    paths:
      - bin/mdb
      - bin/msql
      - bin/mlcx
    expire_in: 30 minutes

# Автотесты
test:
  stage: test
  script:
    - mkdir test/tmp
    - test/test-01-mdb
    - test/test-02-msql
    - test/test-03-msql
    - test/test-04-msql
    - test/test-05-msql-special
    - test/test-06-msql-aggregation
    - test/test-07-msql-insert
    - test/test-08-msql-update
    - test/test-09-msql-create
    - test/test-10-mlex

# Используем GitLab Pages для публикации Doxygen-документации
pages:
  stage: pages
  script:
    # Устанавливаем пакеты
    - apt-get -y install doxygen
    - apt-get -y install graphviz
    # Прогоняем Doxygen
    - doxygen doc/Doxyfile.cfg
    # Копируем рисунки
    - cp -r img doc/html
    - mv doc/html/img/minidblogo.png doc/html/minidblogo.png
    # Диплоим
    - mv doc/html/ public/
  artifacts:
    paths:
      - public
  only:
    - master
```



# Continuous Delivery

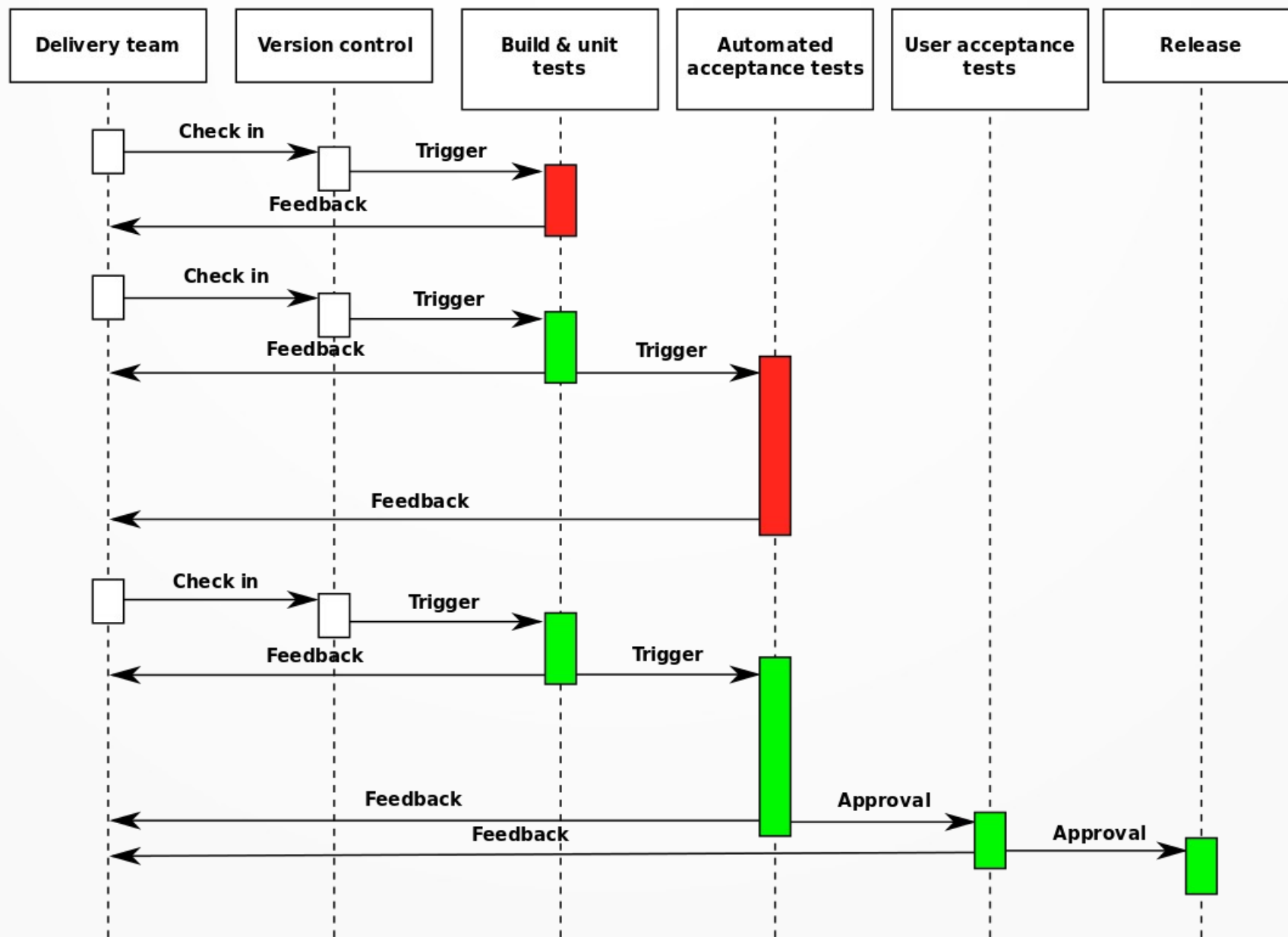
- Подход к разработке программного обеспечения, при котором программное обеспечение производится короткими итерациями, гарантируя, что ПО является стабильным и может быть передано в эксплуатацию в любое время.
- Передача его происходит вручную
- Подход позволяет уменьшить стоимость, время и риски внесения изменений путём более частых мелких обновлений в продакшн-приложение.
- Автоматическое обновление тест-сервера доступно в CI.

# Continuous Deployment

- Подход к разработке программного обеспечения, при котором функциональные возможности программного обеспечения часто предоставляются посредством автоматизированного развертывания
- (в отличие от Continuous Delivery)

# CI/CD

## Принцип работы\*



# CI/CD

## Связь с циклом разработки\*

- Полный и неполный цикл
- (рисуем)

**Вопросы?**