

Лекция 1.

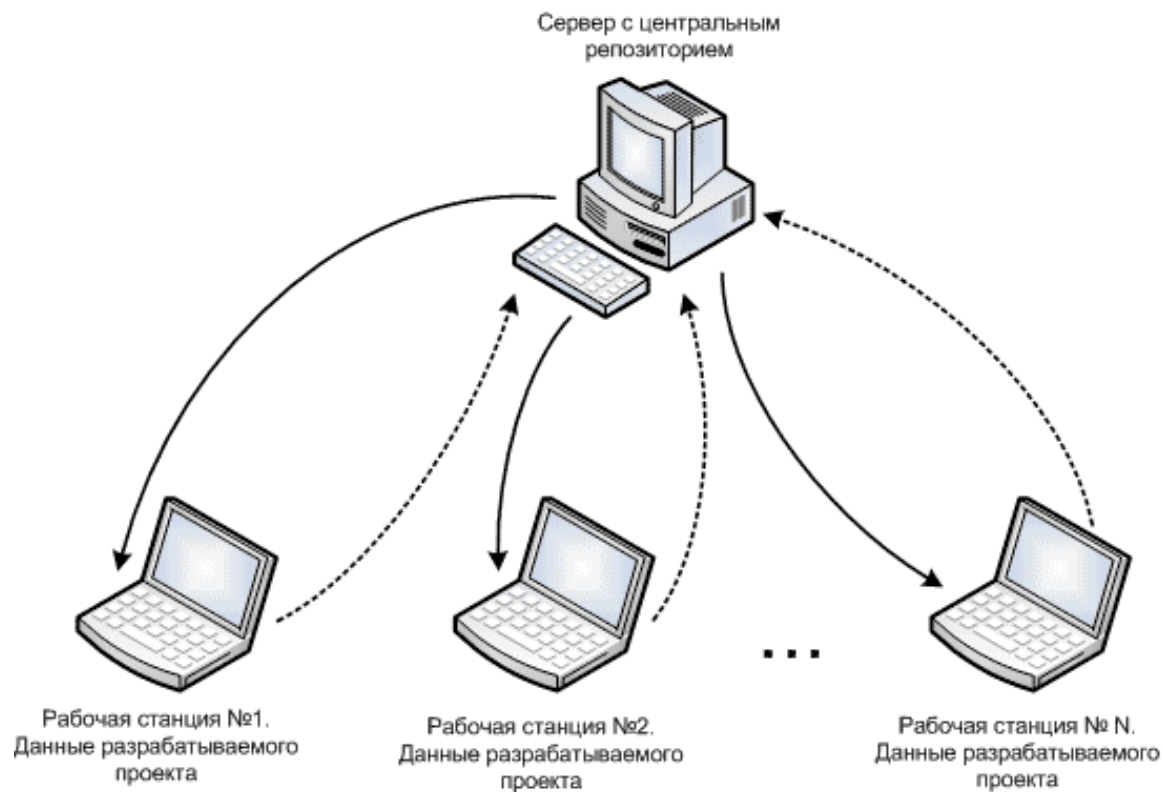
Системы контроля версий

Система контроля версий

- **Система контроля версий** - это система, которая предназначена для хранения файлов, позволяет менять их, с возможностью возврата к любой из предыдущих версий
- В IT-компаниях такие системы применяются для хранения кода проекта и документации
- На англ. - **version control system (VCS)**

Зачем они нужны?

- Хранилище, из которого любой разработчик может получить последнюю (или любую другую) версию проекта
- Необходимая вещь при совместной (да и одиночной) работе

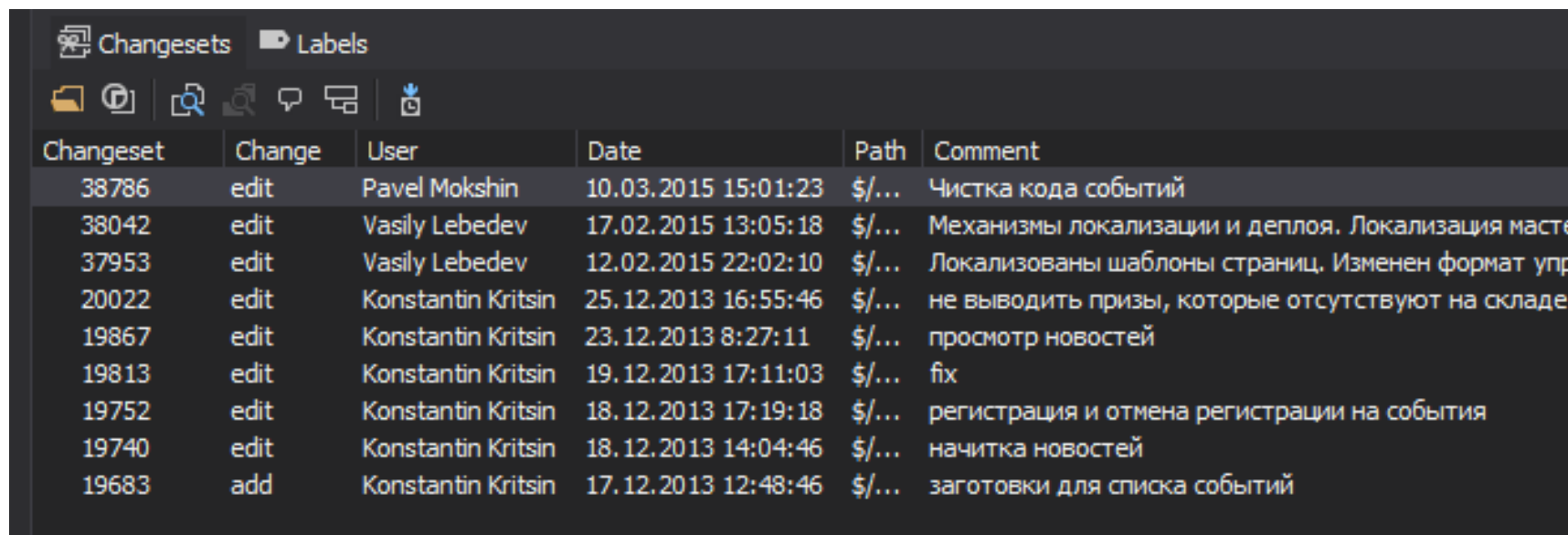


Зачем они нужны?

- Возможность вернуться к любой версии проекта
 - Допустим, в нашей программе после последних изменений обнаружен критический баг
 - Благодаря системе контроля версий мы можем легко откатить эти ошибочные изменения, либо просто вернуться к любой конкретной версии проекта
 - Можно вернуть удаленные файлы

Зачем они нужны?

- Возможность узнать информацию кто, когда, зачем и какие изменения вносил в файлы проекта
 - Например, можно понять, кто знает что делает код в этих файлах
 - По комментарию можно понять что именно менялось
 - По каждому изменению можно посмотреть отличия



The screenshot shows a version control interface with a table of changesets. The interface includes tabs for 'Changesets' and 'Labels', and a toolbar with icons for file operations, search, and commit. The table lists changesets with columns for Change ID, Change type, User, Date, Path, and Comment.

Changeset	Change	User	Date	Path	Comment
38786	edit	Pavel Mokshin	10.03.2015 15:01:23	\$/...	Чистка кода событий
38042	edit	Vasily Lebedev	17.02.2015 13:05:18	\$/...	Механизмы локализации и деплоя. Локализация масте
37953	edit	Vasily Lebedev	12.02.2015 22:02:10	\$/...	Локализованы шаблоны страниц. Изменен формат упр
20022	edit	Konstantin Kritsin	25.12.2013 16:55:46	\$/...	не выводить призы, которые отсутствуют на складе
19867	edit	Konstantin Kritsin	23.12.2013 8:27:11	\$/...	просмотр новостей
19813	edit	Konstantin Kritsin	19.12.2013 17:11:03	\$/...	fix
19752	edit	Konstantin Kritsin	18.12.2013 17:19:18	\$/...	регистрация и отмена регистрации на события
19740	edit	Konstantin Kritsin	18.12.2013 14:04:46	\$/...	начитка новостей
19683	add	Konstantin Kritsin	17.12.2013 12:48:46	\$/...	заготовки для списка событий

Зачем они нужны?

- Можно просматривать историю изменения файла

```
23 self.selectedPage = ko.observable(0);
24 self.newsPerPage = ko.observable(5);
25 self.pages = ko.computed(function() {
26     var total = self.totalCount() || 0;
27     var npp = self.newsPerPage();
28     var count = Math.ceil(total / npp);
29     var result = [];
30     for (var i = 1; i <= count; ++i)
31         result.push(i);
32     return result;
33 });
34
35 self.Template = "News.Page.aspx";
36 self.RenderTo = function (selector, postAction) {
37     var pane = $(selector);
38     pane.empty();
39
40     var tplSrc = cb.getTemplate(self.Template);
41     pane.append(_.template(tplSrc));
42
43     if (self.totalCount() == 0)
44         loadData(self.categoryId, self.selectedPage(), s
45
46
47     if (typeof postAction == "function") {
48         postAction(self, pane);
49     }
50 };
51
52 self.goToAll = function(data, event) {
53     $("#page" + self.categoryId).click();
```

```
64 self.selectedPage = ko.observable(0);
65 self.newsPerPage = ko.observable(5);
66 self.pages = ko.computed(function() {
67     var total = self.totalCount() || 0;
68     var npp = self.newsPerPage();
69     var count = Math.ceil(total / npp);
70     var result = [];
71     for (var i = 1; i <= count; ++i) {
72         result.push(i);
73     }
74     return result;
75 });
76
77 self.Template = "News.Page.aspx";
78 self.RenderTo = function (selector, postAction) {
79     var pane = $(selector);
80     pane.empty();
81
82     var tplSrc = cb.getTemplate(self.Template);
83     pane.append(_.template(tplSrc));
84
85     if (self.totalCount() == 0) {
86         loadData(self.categoryId, self.selectedPage(), self
87     }
88
89     if (typeof postAction == "function") {
90         postAction(self, pane);
91     }
92 };
93
94 self.goToAll = function() {
95     $("#page" + self.categoryId).click();
```

Слияние версий (Merge)

- Допустим, два человека поменяли один и тот же файл, и решили внести в VCS свои изменения
- Один человек внес изменения, второй вносит позже, но в репозитории файл уже изменен – приходится выполнить **слияние версий (merge)**, чтобы объединить изменения от первого и второго пользователя
- VCS часто позволяют автоматически выполнить слияние версий, если изменения разных людей относятся к разным строкам в файле
- Если это не удастся, то человек должен разрешить конфликт вручную – выбрать одну из конфликтующих версий, либо обе, либо может вообще написать свой вариант конфликтующего участка файла

Классификация систем контроля версий

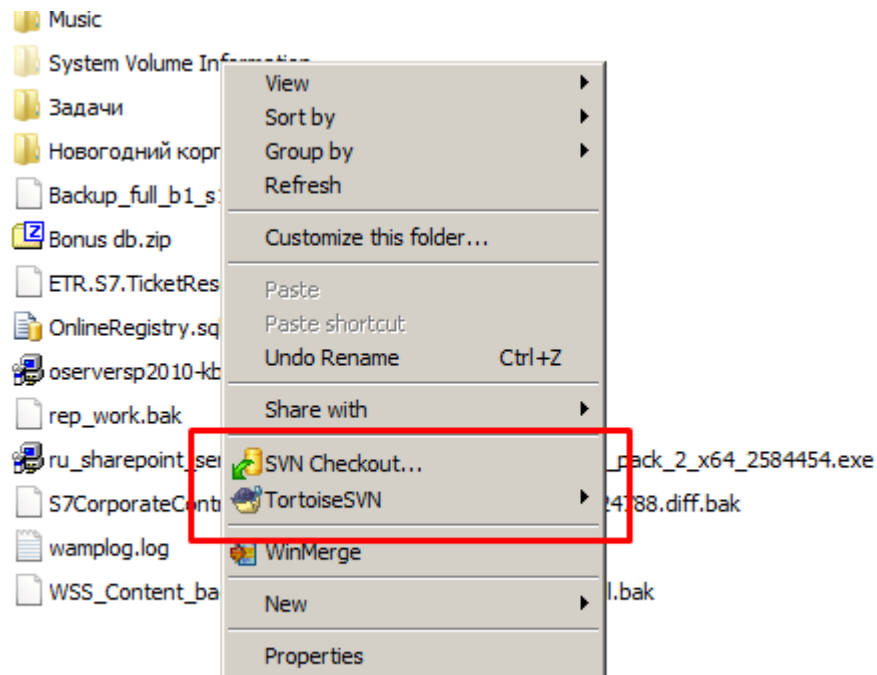
- **Централизованные системы**
 - Есть отдельный выделенный сервер, который хранит в себе **репозиторий** – хранилище файлов, с которыми работает система контроля версий
 - Остальные пользователи забирают себе копию файлов (**working copy**), работают с ними и могут вносить свои изменения в этот глобальный репозиторий
- **Распределенные системы**
 - Нет единого выделенного сервера
 - Каждый пользователь может создать полную копию репозитория, а другие пользователи могут подключаться уже к этой копии, создавать свои копии и т.д.

Классификация систем контроля версий

- **Централизованные системы**
 - SVN – некоммерческая система
 - TFS – коммерческая система
- **Распределенные системы**
 - Git
 - Mercurial

Работа с SVN

- Известная графическая система – **TortoiseSVN**
- Выполнен в виде расширения контекстного меню проводника



- **Tutorial:**
- http://tortoisesvn.net/docs/release/TortoiseSVN_ru/

Основные операции SVN

- **Checkout**
- Выкачивание себе рабочей копии репозитория, чтобы работать с ней, выполняется только первый раз
- **Commit**
- Перенос своих изменений из рабочей копии в репозиторий
- **Update**
- Перенос изменений из репозитория в свою рабочую копию
- То есть забираем изменения, сделанные другими людьми

- **Git** – очень распространенная распределенная система
- В основном это консольная утилита, но есть и графические интерфейсы
- Среды разработки тоже предоставляют свои удобные средства для работы с **Git**

Материалы по Git

- Краткая понятная статья по Git:
<https://habr.com/ru/post/437000/>
- Краткий tutorial по Git:
<https://githowto.com/ru>
- Книга про Git, на русском:
<https://git-scm.com/book/ru/v2>
- Она же в оригинале:
<https://git-scm.com/book/en/v2>

Операции git

- **Клонирование (clone)**
- Создание локального репозитория, как копии от удаленного репозитория
- **Add/remove**
- Удаление/добавление файлов под контроль git
- Нужно делать при добавлении в проект новых файлов и при удалении файлов из проекта

Операции git

- **Коммит - commit**
- Формирование набора изменений
- Выбранные изменения попадают в набор изменений (он тоже называется **commit**), но пока не вносятся в репозиторий
- Чтобы внести изменения в удаленный репозиторий, нужно выполнить команду **push**
- **Push**
- Вносит последние коммиты в удаленный репозиторий
- После этого эти изменения доступны другим

Перенос изменений на сервер

- Чтобы перенести изменения из своего репозитория в удаленный, нужно выполнить эти описанные ранее команды в следующем порядке:
 1. **add** – добавление файлов в индекс для последующего коммита
 2. **commit** – фиксация набора изменений (коммит)
 3. **push** – отправка коммитов в удаленный репозиторий

Операции git

- **Pull**
- Получение изменений из удаленного репозитория в свой локальный репозиторий

Github

- Это крупный бесплатный веб-сервис для хранения Git-репозитория
- Там можно бесплатно создавать свои репозитории
- Для управления им предоставляется удобный веб-интерфейс, а также Windows приложение
- Очень популярен для некоммерческих и open-source проектов
- Бесплатно можно создавать как публичные репозитории (доступны для просмотра всем), так и приватные
- Есть и платные тарифы
- <https://github.com/>



Bitbucket

- В целом то же самое, что **Github**
- <https://bitbucket.org/product>
- Здесь также можно бесплатно завести приватный репозиторий, доступом к которому вы можете управлять
- Бесплатные приватные репозитории на **Github** появились только с начала 2019 года, а до этого часто для приватных репозиториев использовался **Bitbucket**



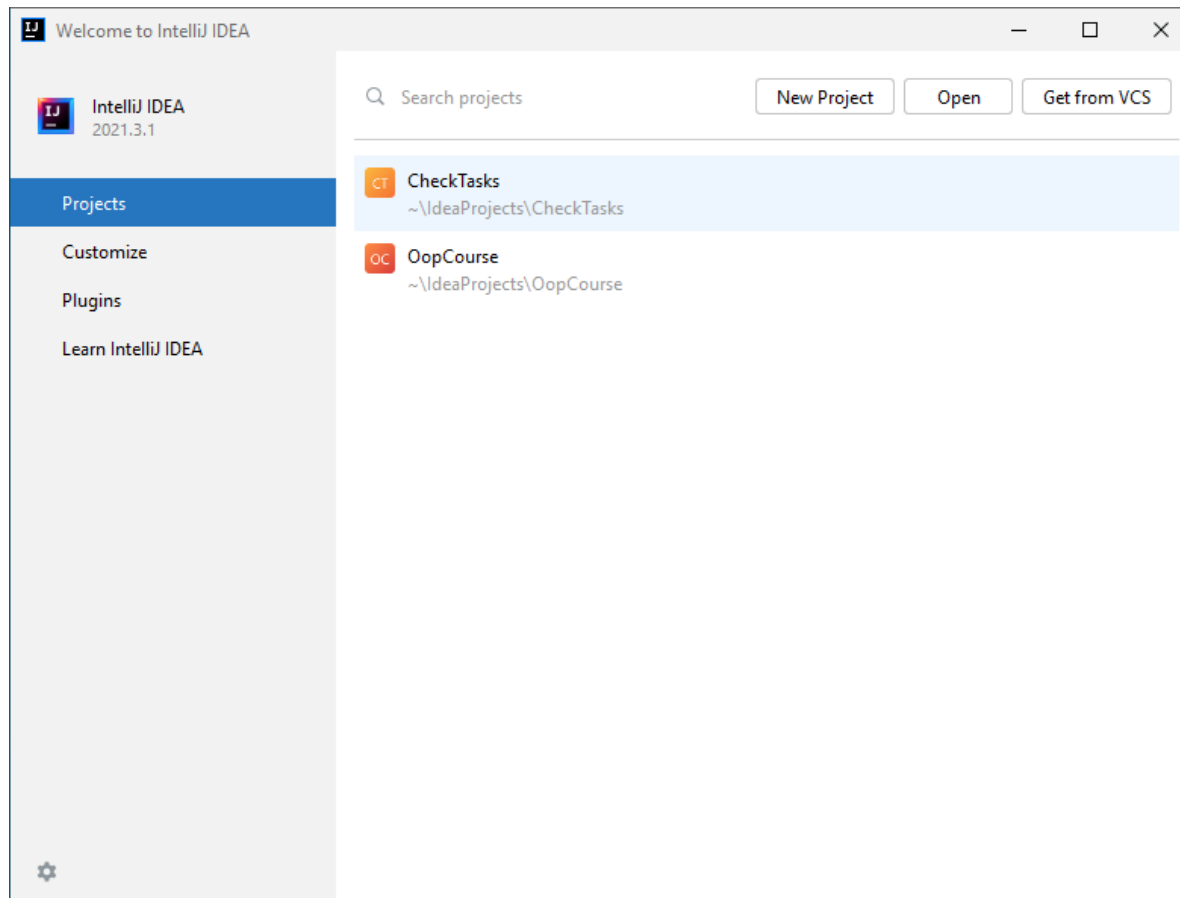
**Как создать репозиторий
в Github и создать
проект в IDEA**

Мануал по Github

1. Регистрируемся на <https://github.com>
2. Создаем новый **репозиторий** (кнопка «+», потом выбираем пункт «**New Repository**»), задаем ему информативное имя. Например, **OopCourse**
 - Для Java - в списке «**Add .gitignore**» выбираем Java
 - Для C# - не ставим чекбокс «**Add .gitignore**»
3. Скачиваем и устанавливаем **git**:
 - <https://git-scm.com/downloads>

Клонирование проекта в IDEA

1. Закрываем текущий проект: **File -> Close Project**
2. Выбираем пункт «**Get from VCS**» в верхнем правом углу

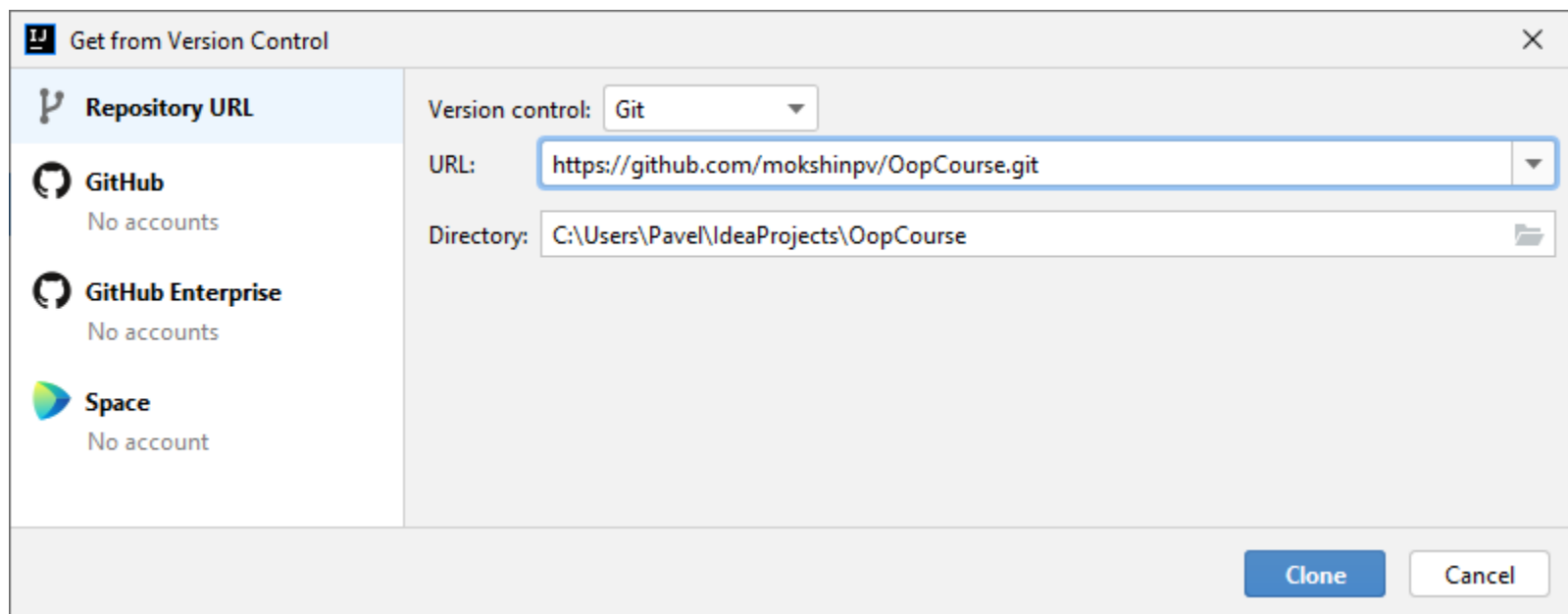


Клонирование проекта в IDEA

- Далее есть несколько вариантов, можно использовать любой из них:
 1. Вкладка **Repository URL**
 - Подходит не только для **Git** и не только для **Github**
 2. Вкладка **Github**
 - Заточена на **Github**

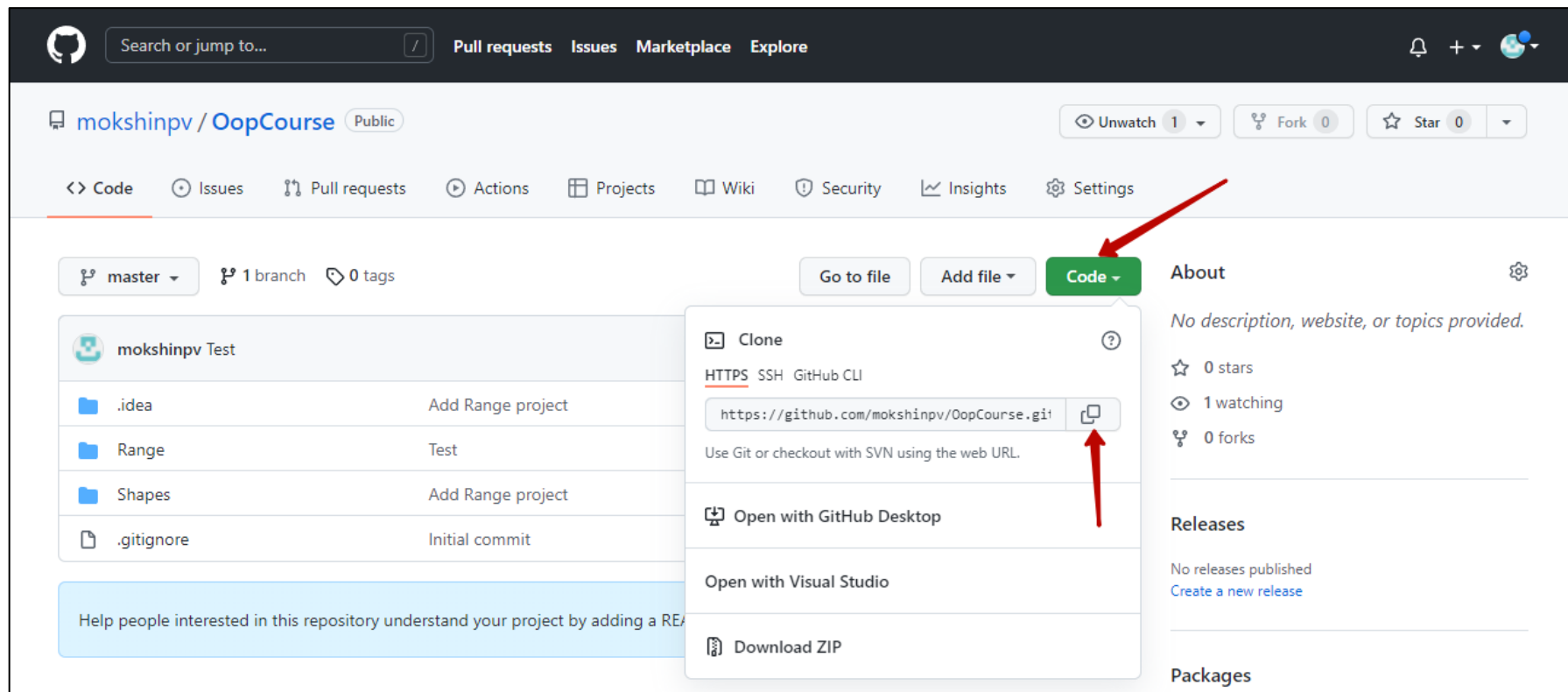
Вкладка Repository URL

- Выбираем систему контроля версий – **Git**
- Вставляем URL репозитория
- При желании можно изменить папку, куда скачивать репозиторий
- Нажимаем **Clone**



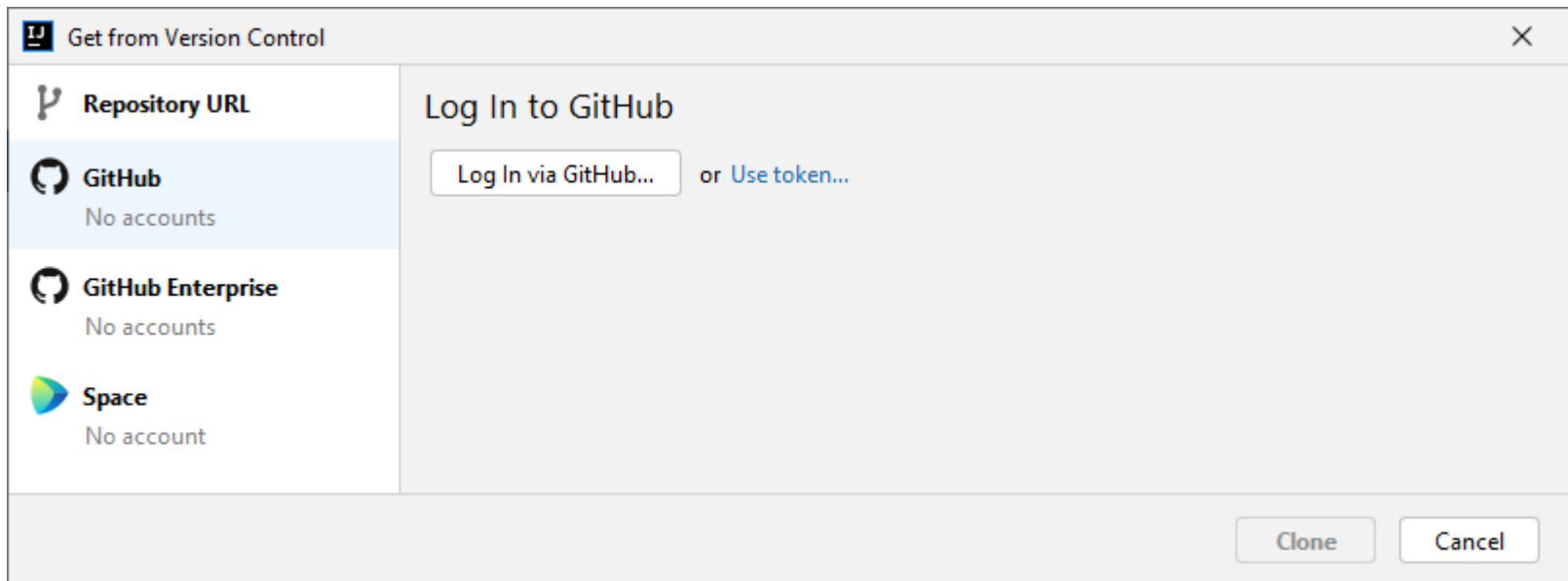
Где взять URL репозитория

- URL репозитория в **Github** можно взять здесь:



Вкладка Github

- Здесь сначала нужно настроить доступ к своему аккаунту **Github**
- Можно выбрать любой из вариантов:
 - **Log In via Github**
 - **Use token**



Вариант Log In to Github

- Вас перекинет на сайт, там нужно нажать кнопку **“Authorize in GitHub”**
- Далее нажать **“Authorize JetBrains”**
- Далее вам нужно будет ввести пароль от аккаунта **Github**
- После этого в **IDEA** можно будет просто выбрать нужный репозиторий и нажать **Clone**

Вариант Use token

- Вам нужно будет создать **токен доступа** в **Github** и вставить его в **IDEA**
- Здесь **токен** – некоторая длинная последовательность, которая используется для выдачи прав доступа
- Можно нажать кнопку **Generate**, и вас перекинет на сайт **Github**
 - На странице уже будут выбраны все нужные права доступа
 - При желании можно поменять срок истечения токена
- Нажмите **Generate token**, а дальше скопируйте его и вставьте в **IDEA**
- Далее можно просто выбрать репозиторий и нажать **Clone**

Настройка проекта

1. Создаем в проекте **модуль** для одной из задач, например, назовем его **Vector** для первой задачи
- Для каждой задачи заводите отдельный **модуль-подпапку**, давайте ему имя задачи из листа успеваемости
 - Для этого выберите пункт **File -> New module**
 - Пожалуйста, пишите комментарии ко всем коммитам. Можно писать на русском
 - Последнюю версию выкладывайте в свой репозиторий, а потом напишите мне письмо, что задачу можно смотреть

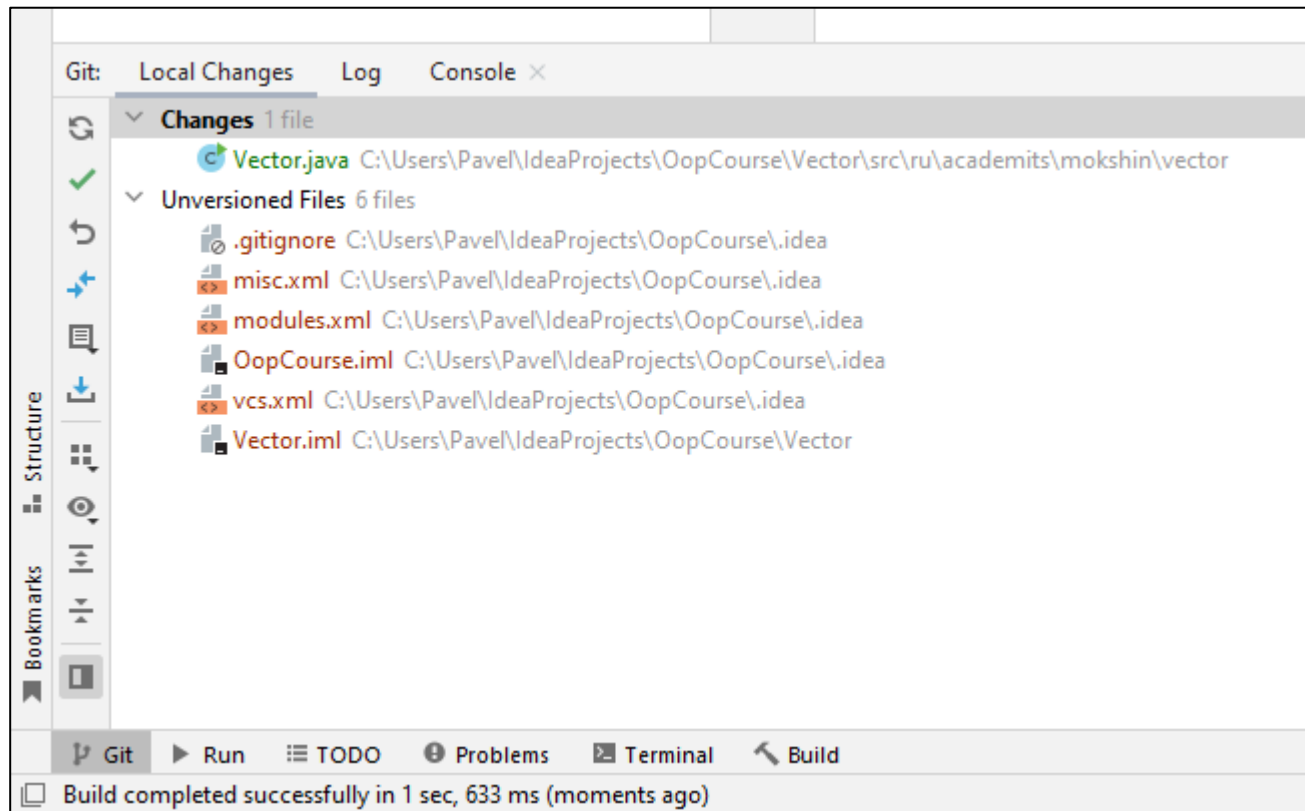
Настройка проекта

- Создайте в модуле класс с **main**'ом, и запустите программу
- После этого (либо еще до этого) IDEA создаст нужные ей файлы в папке **.idea** и в папке модуля
- Эти файлы нужно будет добавить в репозиторий, см. следующий слайд

Настройка проекта

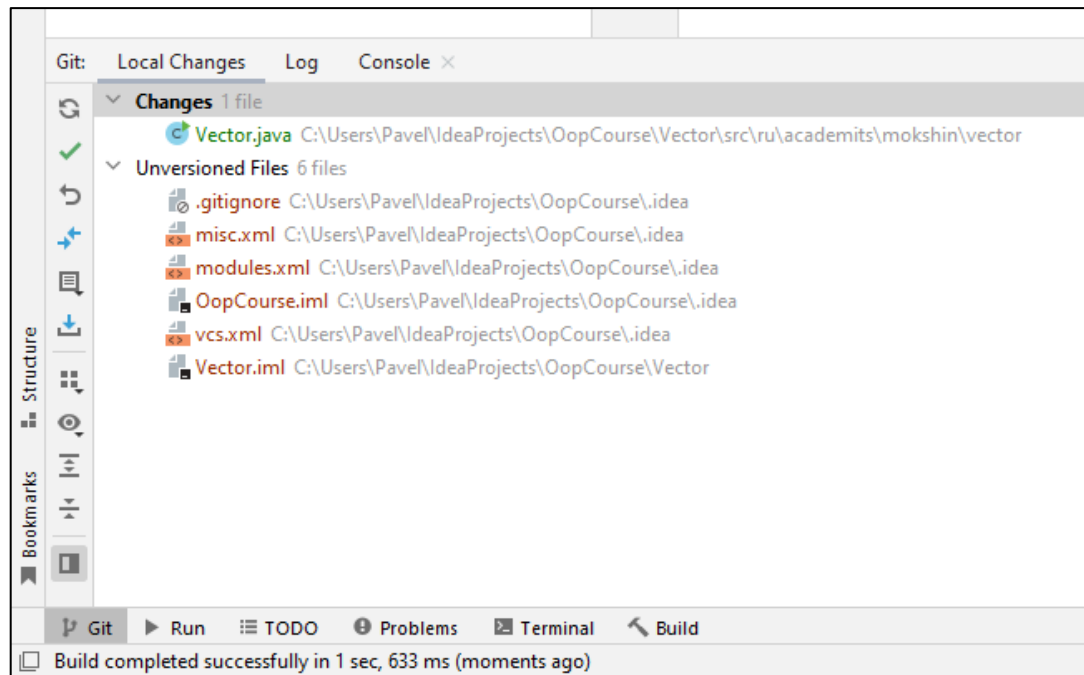
2. Нужно добавить под управление **Git** все новые файлы. Это следует делать всегда, когда в проект добавляются новые файлы

 - Сначала откройте панель **Git** в нижней части **IDEA**



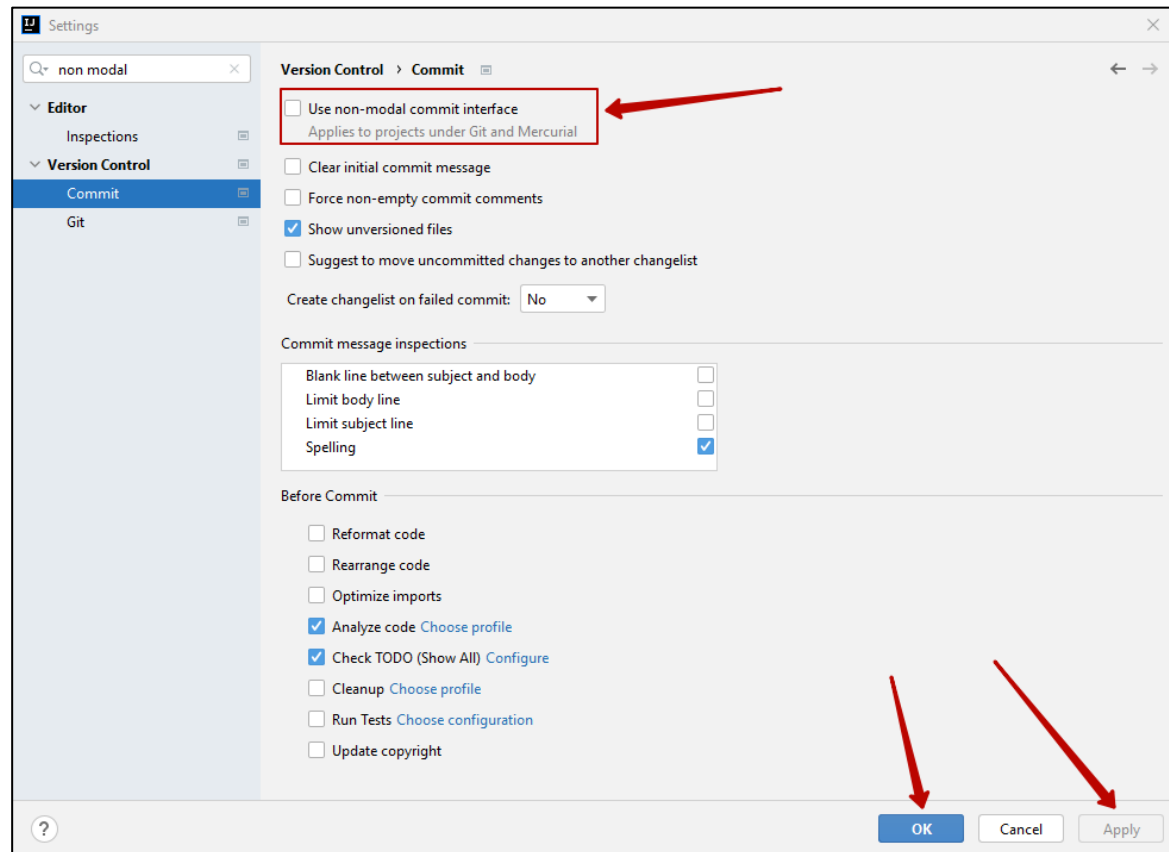
Добавление новых файлов в проект

- Если мы добавляем в **git** новые файлы, то сначала для них должна быть выполнена команда **add**, чтобы система контроля версий начала их отслеживать
- Такие файлы находятся в пункте **Unversioned files** во вкладке **Local Changes**. Выделяем их, жмем правой кнопкой, и применяем команду **Add to VCS**



Что если нет вкладки Local changes?

- В IDEA есть 2 варианта UI для работы с **Git**
- Для варианта из лекции нужно снять этот чекбокс



Что если нет вкладки Local changes?

- По умолчанию этот чекбокс выбран
- В этом случае в панели **Git** не будет вкладки **Local changes**
- Но слева сверху в IDEA будет боковая панель **Commit**
- В курсе можете использовать любой удобный вариант

Настройка проекта

3. Теперь сделайте **commit**, а затем **push**
 - **Commit, push, pull** и другие основные операции доступны в меню **Git**
 - При **push** вам может понадобиться получить доступ к **Github** – либо указать токен, либо залогиниться, см. предыдущие слайды про настройку доступа к **Github**
 - Всё, изменения ушли на сервер в репозиторий

Процесс работы с Git

- Если несколько разработчиков, то периодически делаем **Pull** – забираем последние изменения из репозитория
- Когда сами меняем код, то делаем **Add** для новых файлов, набираем изменения в коммиты (команда **Commit**), а потом делаем **Push** (коммиты отправляются в репозиторий)
- Иногда накапливают несколько коммитов, а потом отправляют их на сервер
- К каждому коммиту надо писать краткий, но понятный и точный комментарий – что именно изменилось

Файл .gitignore

Файл .gitignore

- В Git репозиторий можно добавлять специальный файл с именем **.gitignore**
- В этом файле указываются файлы, которые должны игнорироваться **git**'ом
- Например, это результаты компиляции, разные временные файлы среды разработки и т.д.
- Эти файлы не нужны в репозитории, потому что эти файлы не важны, но занимают место

Файл .gitignore

- В файле можно писать записи такого вида:
- *.class
node_modules
folder/file.txt
- Т.е. можно указывать конкретные пути к файлам или папкам, либо использовать специальные последовательности * и др.
- * означает любой символ, кроме /
- Т.е. здесь мы игнорируем все файлы с расширением .class, папку node_modules и файл folder/file.txt

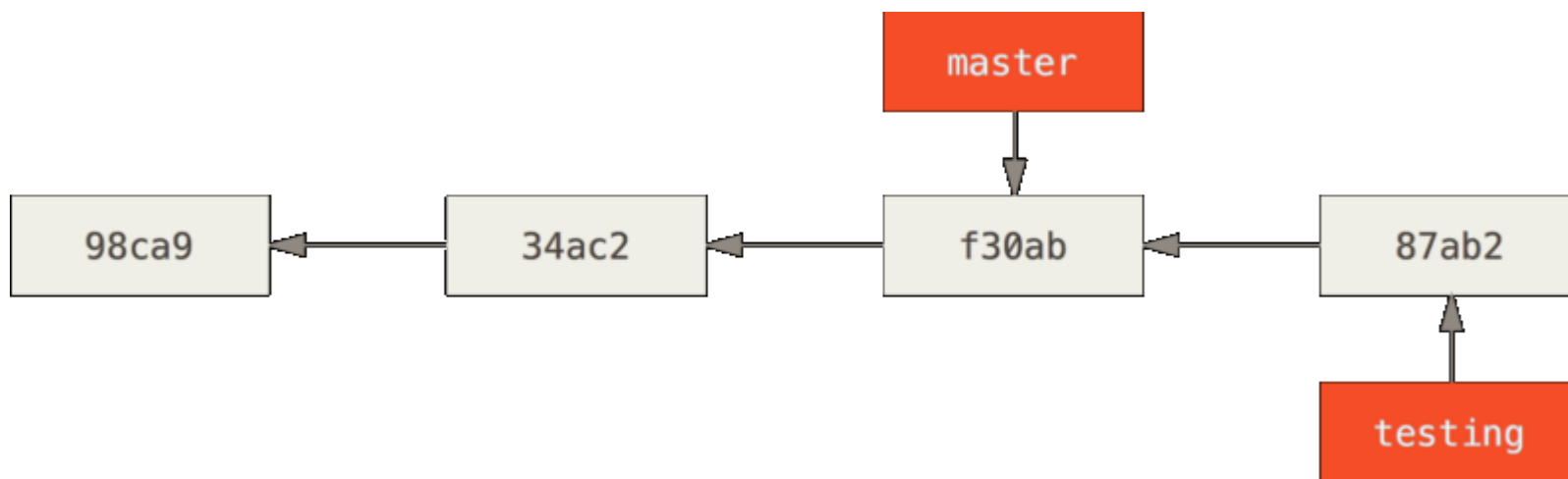
Файл .gitignore

- Действие файла **.gitignore** распространяется на папку, в которой он находится, и на все вложенные папки
- Гипотетически во вложенных папках можно добавлять еще файлы **.gitignore**, тогда они будут дополнять список игнорируемых файлов
- Документация:
- <https://git-scm.com/docs/gitignore>

Модели работы с Git

Ветки

- Чтобы понять модели работы с **Git** нам нужно понятие **ветка**
- **Ветка (branch)** – это указатель на некоторый коммит
- Каждый коммит хранит ссылку на свои родительские коммиты
- Поэтому, если у нас есть указатель на коммит, то по сути у нас есть вся история от начала до этого коммита

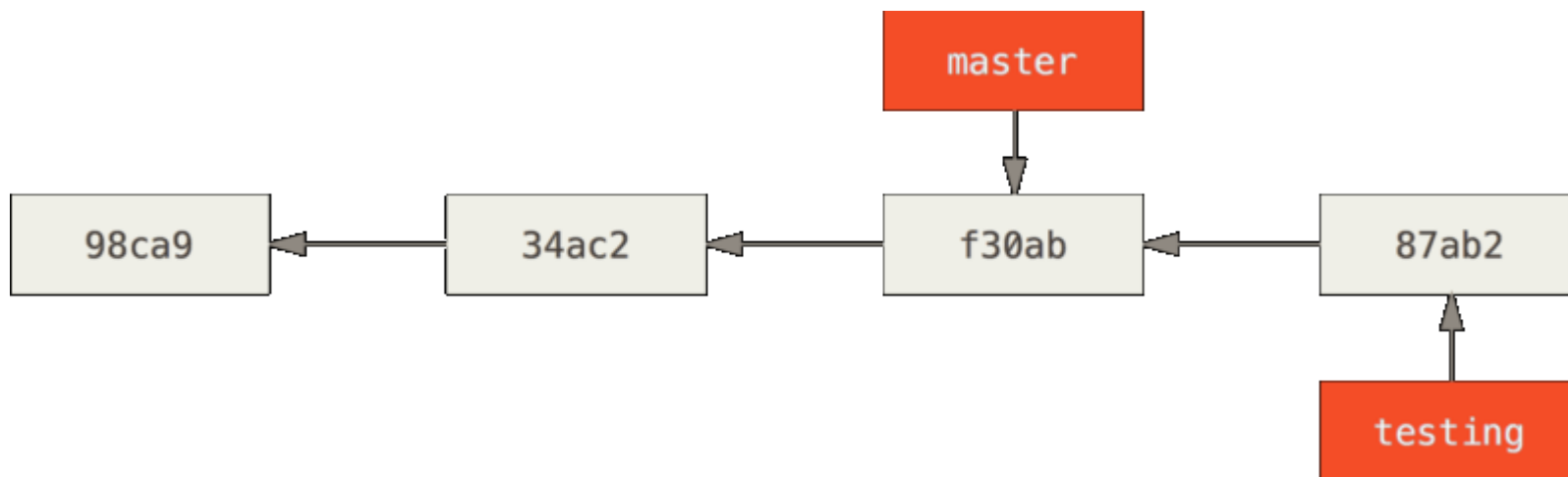


Ветка master

- Когда мы создаем репозиторий, там сразу создается 1 ветка с именем **master**
- Далее мы можем создавать новые ветки

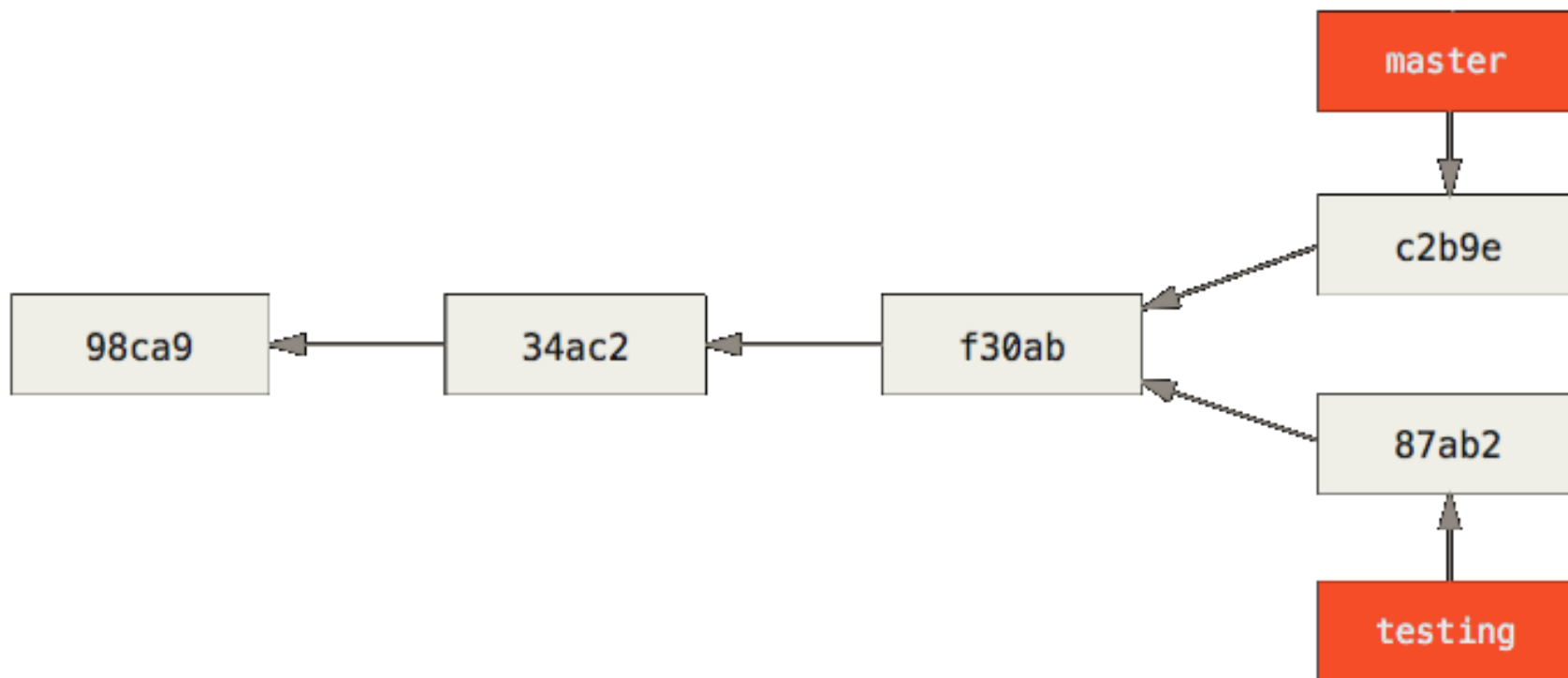
Ветки

- На данном рисунке у нас есть 2 ветки – **master** и **testing**
- Ветка **testing** была создана от ветки **master**, и туда внесли 1 КОММИТ
- Ветки позволяют независимо работать с разными версиями кода
- Ветки в Git легко создавать, легко переключаться между ними



Ветки

- А тут в **master** внесли еще 1 коммит
- Как видим, ветки могут развиваться полностью независимо друг от друга



Слияние веток

- Часто при работе в одной из веток нужно перенести изменения в другую ветку
- Это делается при помощи **слияния (merge)**
- Рассмотрим слияние **fast-forward** и **three way merge**:
- [Ссылка на статью](#)

Модели работы с Git

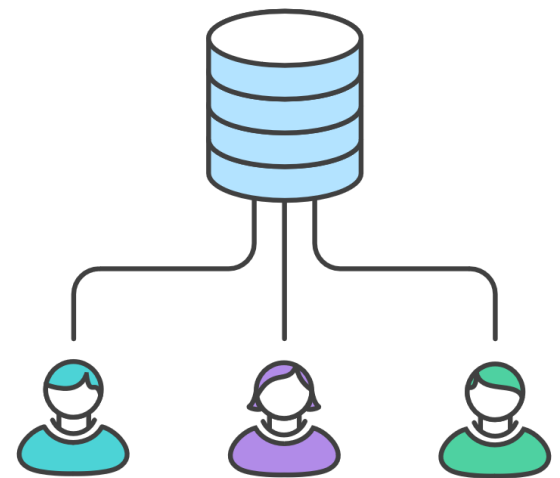
- Для совместной работы в Git сложилось несколько популярных моделей работы:
 - **Centralized Workflow** (централизованный процесс)
 - **Feature Branch Workflow** (процесс с Feature ветками)
 - **Gitflow Workflow**
- Есть и другие подходы, но мы рассмотрим эти, т.к. они самые популярные
- Подробнее можно почитать здесь:
- <https://www.atlassian.com/git/tutorials/comparing-workflows>

Выбор модели работы с Git

- При выборе модели нужно ориентироваться на следующие факторы:
 - Размер команды – не тормозит ли процесс работу команды
 - Не является ли модель переусложненной для текущей ситуации
- Сейчас мы рассмотрим модели от самой простой к самой сложной

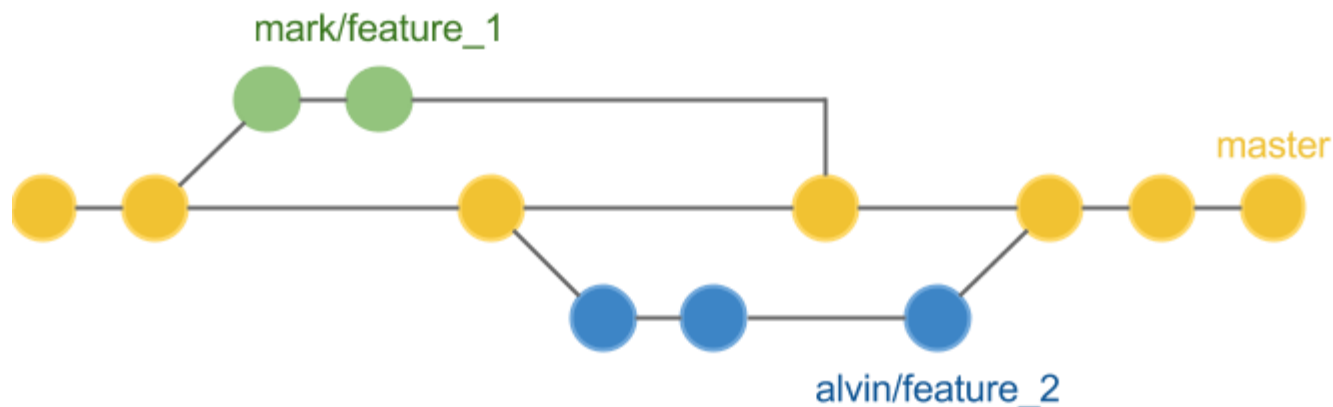
Centralized Workflow

- Все пользователи работают только с одной веткой – **master**
- Все выкачивают себе копию кода, работают с ней, и потом сразу же вносят изменения в центральный репозиторий
- Т.е. работа ведется так же, как с централизованными VCS
- В курсе мы будем работать по этой модели, она самая простая
- Подходит только для очень маленьких команд и простых проектов



Feature Branch Workflow

- Есть постоянная ветка **master** - в ней находится актуальная стабильная версия кода
- Для каждой задачи разработчик заводит отдельную ветку (так называемую **feature ветку**) и работает в ней
- Когда задача завершена, разработчик делает **merge (слияние)** своей ветки в **master**
 - Обычно это делается не напрямую, а через **pull request**
- После этого **feature ветку** удаляют, т.к. она больше не нужна



Pull request

- Часто для безопасности почти у всех разработчиков забирают права делать **merge** в **master**
- Но как-то ведь переносить изменения в **master** нужно
- Это можно сделать через **pull request**
- Разработчик, когда завершил задачу, создает **pull request** – запрос на **merge** в **master**
- При этом некоторому ответственному за это человеку приходит уведомление, что разработчик хочет сделать **merge** ветки в **master**

Pull request

- Ответственный человек может провести **review** кода, написать комментарии и т.д., и отклонить **pull request**
- Либо может утвердить его, тогда выполнится **merge**
- Т.е. **pull request** позволяет организовать процесс **code review** и проконтролировать что именно попадает в **master**

Feature Branch Workflow

- Эта модель хорошо подходит для маленьких и средних по размеру команд
- При этом модель довольно простая, в ней нет излишней сложности

Gitflow Workflow

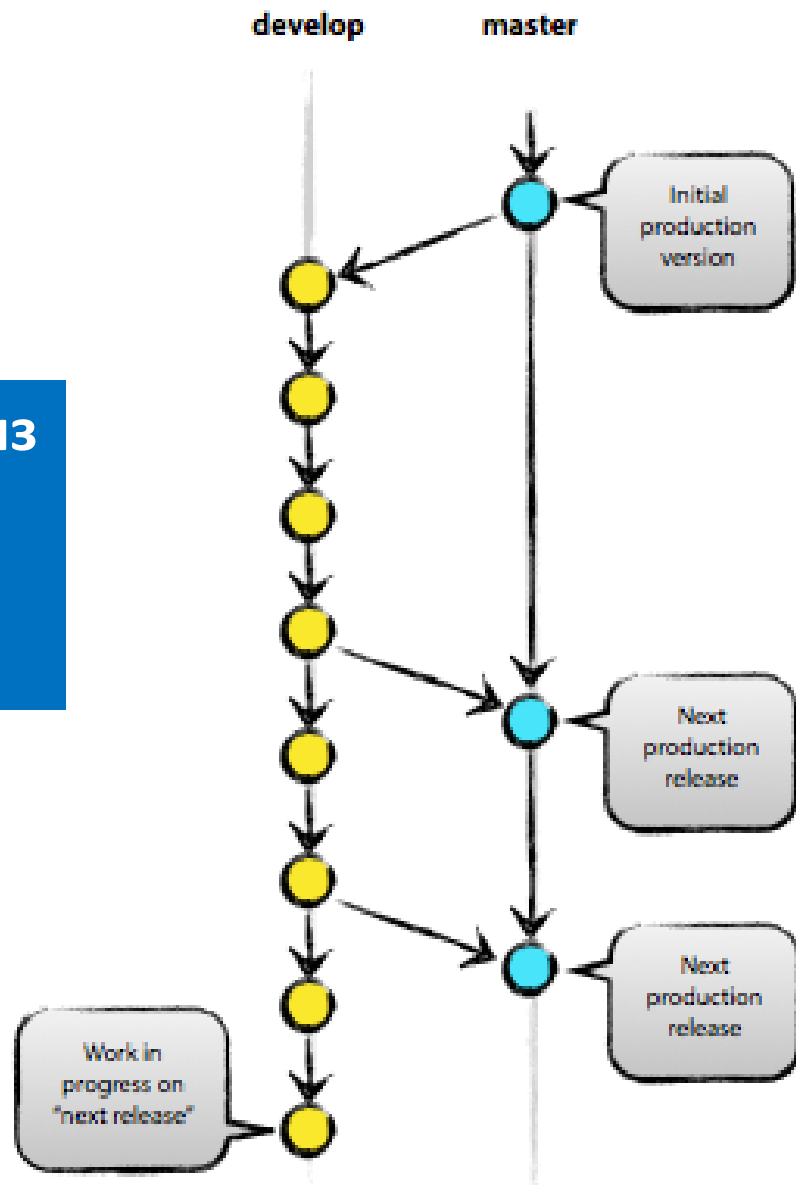
- **Gitflow** – это популярная модель использования **веток** в git
- **Gitflow** включает в себя концепции из **Feature Branch Workflow** и добавляет новые
- В целом этот процесс довольно сложный и запутанный, поэтому применять его нужно только если он действительно нужен
- Подробнее познакомиться можно здесь:
- <https://habrahabr.ru/post/106912/>
- <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

- В проекте заводятся две постоянные ветки – **master** и **develop**
- Вообще, это обычные ветки, имена сложились исторически
- **master** – содержит стабильную версию проекта. Эта ветка всегда должна быть готова к выкладыванию на **production**
- **develop** – в этой ветке самая новая версия проекта. Эта версия может быть нестабильна – могут быть баги, фичи могут быть недоделаны

Ветки master и develop

Как видно, в master изменения делаются реже

Периодически изменения из develop переносятся в master, когда фича протестирована



Ситуации при gitflow

- При работе по **gitflow** бывают следующие ситуации:
 - Нужно срочно сделать **исправление (hotfix)** и выложить на **production**
 - Нужно просто сделать **фичу**, протестировать, а потом выложить на **production**

Срочные задачи

- От **master** делается отдельная временная ветка, исправление/доработка делается в ней
- Такой ветке часто дают название по номеру задачи в багтрекере, например, **feature121** или **bug234**
- В эту ветку постоянно забираются изменения из **master** – другие разработчики тоже работают над своими задачами
- Когда всё готово и протестировано, изменения переносятся в **master** – делается **merge** (слияние)
- Всё, эти изменения рано или поздно будут выложены на **production**
- После этого нужно перенести эти изменения и в ветку **develop**. Там тоже нужно будет выполнить **merge**
- После этого временную ветку удаляют

Несрочные задачи

- От **develop** делается отдельная временная ветка, исправление/доработка делается в ней
- В эту ветку постоянно забираются изменения из **develop** – другие разработчики тоже работают над своими задачами
- Когда всё готово и протестировано, изменения переносятся в **develop** – делается **merge** (слияние)
- Когда-нибудь эти изменения будут протестированы, и будет выполнен merge **develop** в **master**, а потом изменения попадут и на **production**

Багтрекеры. Процесс разработки

Багтрекеры

- **Багтрекеры (или системы управления задачами)** – ещё один важнейший инструмент командной работы
- Обычно представляют собой сайт, в котором заводятся задачи по проекту
- Известные багтрекеры:
 - JIRA
 - Redmine
 - TFS
 - YouTrack

Параметры задачи

- Менеджер проекта заводит там задачи исполнителям
- У задач обязательно есть:
 - **Уникальный номер (ID)** – обычно целое число
 - **Тип** – task или bug. Также можно создавать свои типы задач
 - **Название** – должно быть чётким и как можно более коротким
 - **Описание** – полное описание задачи. Должно содержать всё необходимое для правильного понимания и выполнения задачи

Параметры задачи

- У задач обязательно есть:
 - **Автор задачи** – кто её создал
 - **Ответственный** – на кого назначена задача.
Ответственный может меняться в ходе работы над задачей
 - **Статус** – команда сама придумывает какие статусы могут быть у задач, каков их смысл и когда следует их менять.
 - Может быть такой набор статусов:
New, In Progress, Ready To Review, Review, Ready To Test, In Test, Done

Параметры задачи

- Дополнительно у задачи могут быть:
 - **Вложения.** Любые файлы
 - **Связи с другими задачами.** Например, отношение Дубликат, Родитель-Потомок и т.д.
 - **Теги.** Теги команда придумывает сама и может помечать ими задачи. Например, может быть тег **Deployed to prod**, который означает, что эта задача выложена на production
- Каждая задача также отслеживает и хранит **историю изменений** – кто, когда и что менял