# Лекция 1
# Введение

# Преподаватель

**Балакина Елена**

- Окончила бакалавриат и магистратуру, «Прикладная информатика» НГТУ
- Senior QA Automation Engineer, Noveo Group
- QA Ментор, solvery.io, qa.guru
- Прошла всю линейку курсов Academ IT School по тестированию и программированию на Java

# Опыт работы

## Senior QA Automation Engineer, Noveo Group, с 2020


- Автоматизация тестирования UI
  (Java/Kotlin, Selenium, JUnit, Gradle, Spring, Cucumber, GitLab)
- Тестирование API (Fuel, RestAssured, Postman)
- Разработка стратегии тестирования, тест-анализ, тест-дизайн
- Внутренний менторинг, проведение собеседований

## QA Engineer, LC Group, 2019


- Функциональное, регрессионное тестирование, UX/UI тестирование + sanity/smoke
- Анализ спецификаций, составление тестовой документации
- Автоматизация регрессионного тестирования (Selenium WebDriver, Java)

# Опыт работы

**QA Automation Engineer, EPAM Systems, 2020-2022**

- Волонтерский проект: веб-приложение для некоммерческой организации
- Построение и поддержка тестового фреймфорка (Java + Selenium, Selenide, Maven, JUnit)
- Разработка стратегии автоматизации тестирования

# План курса

1. Введение в автоматизацию тестирования.
   Система контроля версий Git.
   Среда разработки IntelliJ IDEA
2. HTML, CSS
3. Selenium IDE, Xpath/CSS локаторы
4. Основы Java
5. Selenium WebDriver
6. Тестовый фреймворк
7. Автоматизация тестирования API (Введение + Postman)
8. Автоматизация тестирования API (Postman)

# Введение

*«Test automation is any use of tools to support testing»*

Rapid Software Testing
James Bach, Michael Bolton

- **Автоматизированное тестирование –** это тестирование с использованием любых инструментов для автоматизации.
- Что можно автоматизировать:
  - подготовку тестовых окружений
  - подготовку тестовых данных
  - формирование отчётов о результатах тестирования
  - создание тестов
  - исполнение тестов

# Введение

| Анализ требований | человек |
|---|---|
| Тест-дизайн, документирование | человек |
| Подготовка тестовых данных | человек/машина |
| Ручной прогон тестов, оценка результатов | человек |
| Написание автотестов, отладка | человек |
| Создание баг-репортов | человек |
| Отслеживание покрытия автотестами | человек/машина |
| Определение графика выполнения | человек |
| Выполнение автотестов | машина |
| Оценка результатов, создание отчетов | человек/машина |
| Поддержка автотестов | человек |

- **Обычно под автоматизацией имеется в виду:**
  - Автоматизированное исполнение тестов (иногда с генерацией входных данных) и автоматизация проверок

# Зачем нужна автоматизация?

| | Преимущества | Недостатки |
|---|---|---|
| **Повторяемость** | Автоматизация дает один и тот же результат теста | «Эффект пестицида» |
| **Скорость / Затраты времени** | Тесты проходят быстро | Необходимо разрабатывать тесты специалистами |
| **Поддержка** | Поддерживать автотесты проще чем ручные тест кейсы | Сложно при плохом фреймворке и увеличении количества тестов |
| **Генерируемые отчеты** | Специальные инструменты | |
| **Автономность** | Работа тестов не требует присутствия людей | |
| **Машинная точность** | | Автотест всегда видит только то, что задано |
| **Анализ падений** | | Сложно при недостаточном логировании |

# Когда оптимальна автоматизация?

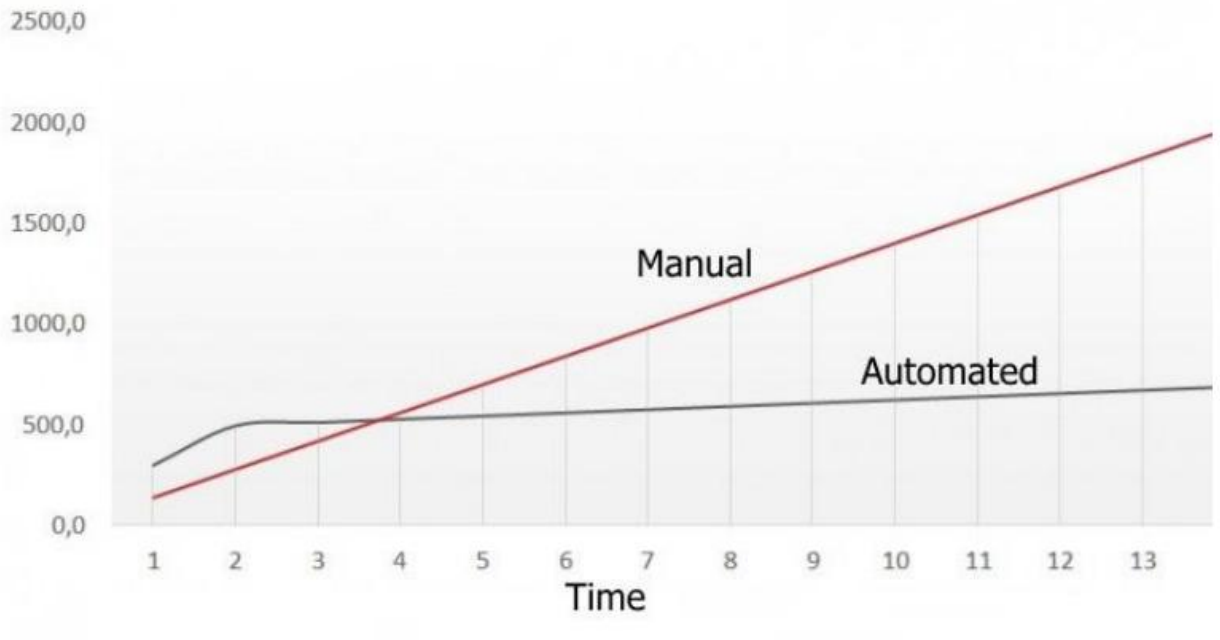
- Проект долгосрочный (от года и более)
- Проект на стадии, когда основной функционал стабилен
- Частые релизы (и следовательно частое регрессионное тестирование)
- Много рутинных проверок
- Большой процент пропуска багов (человеческий фактор)

# Когда автоматизация не нужна?

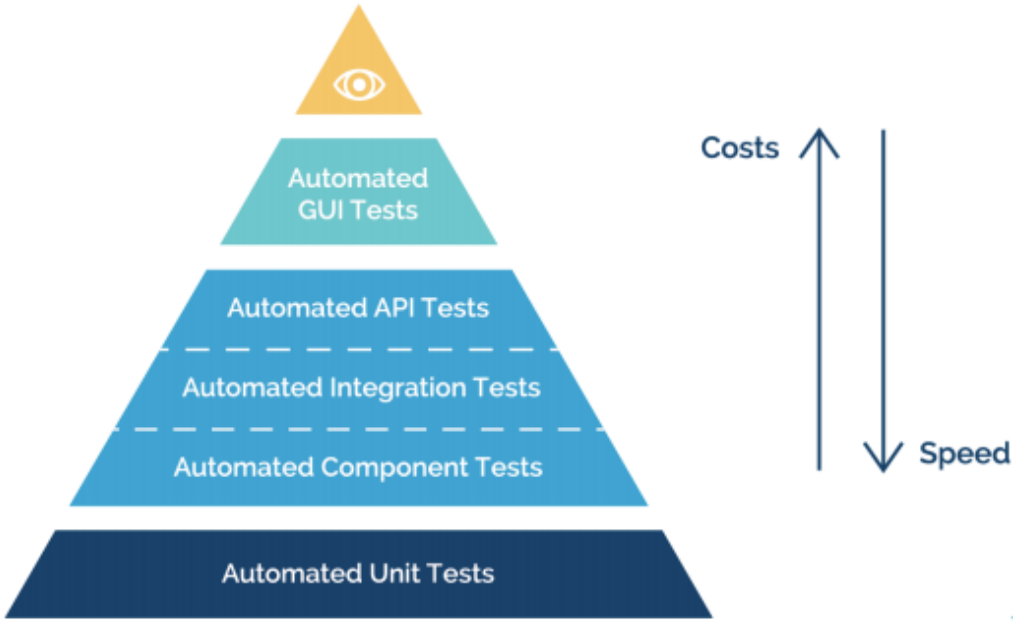
- Краткосрочный проект
- Молодой проект с часто меняющимся функционалом
- Ручных тестов мало и они проходят достаточно быстро
- Тестирование верстки, юзабилити

# Пример

- Затраты времени на ручное тестирование – 30 часов
- Затраты времени на автоматизацию – 50 часов
- Затраты времени на автоматизированное тестирование – 12 часов
- Позволяет сэкономить до 216 часов в год при ежемесячном релизе



# Пирамида тестирования




- Чем выше по пирамиде, тем тесты проходят дольше по времени
- Чем выше по пирамиде, тем тесты более дорогие в разработке и поддержке

# Unit тесты

- **Unit-тесты** (модульные тесты) – проверяют на корректность отдельные модули, компоненты исходного кода программы
- Обычно пишутся разработчиками
- Проходят за относительно быстрое время
- Запускаются регулярно при каждом изменении кода

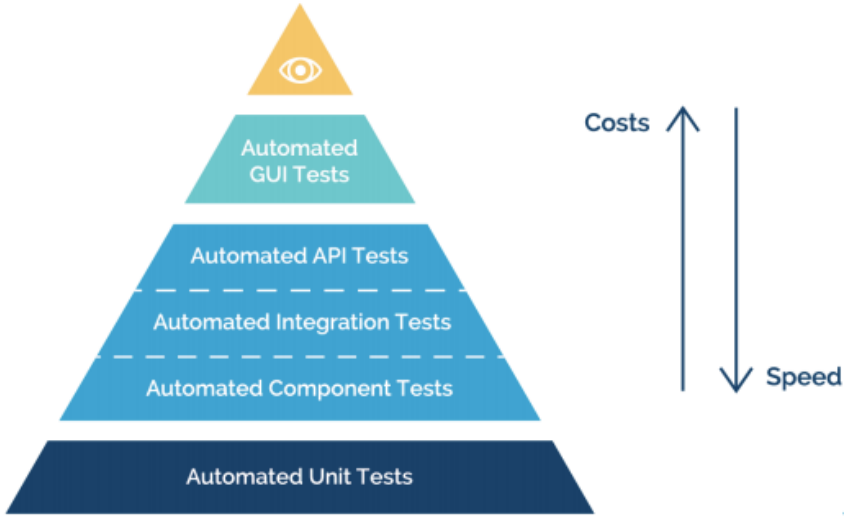
# API тесты

- **API (Application Programming Interface)** – программный интерфейс приложения
- Тестирование функциональности (бизнес-логики) приложения без использования UI
- Тесты проходят существенно быстрее, чем UI-тесты. По времени разработки тестов – обычно быстрее, чем UI-тесты
- **Инструменты:** Postman, Jmeter, SoapUI, библиотеки для автоматизированного тестирования (RestAssured, Retrofit, Fuel /и др.)

# UI тесты

- **UI (User interface) тесты** – тестирование интерфейса приложения с помощью имитации действий пользователя (клик, ввода данных в поля формы, скроллы и т.д.)
- Наиболее длительные по времени тесты
- Часто бывают нестабильными из-за изменения функциональности приложения и из-за изменения локаторов
- **Инструменты:**
  - Selenium WebDriver
  - Test Complete
  - Рекордеры (Selenium IDE)

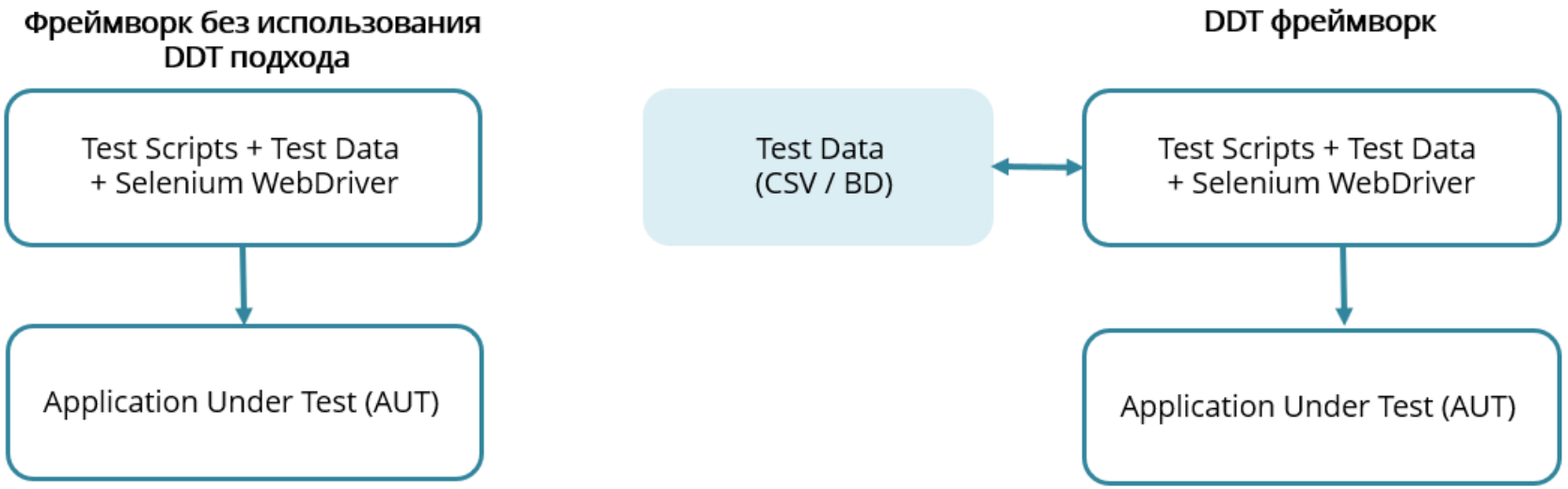
# Пирамида тестирования




- **В нашем курсе:**
  - Автоматизированные UI тесты – Selenium WebDriver + Java
  - Автоматизированные API тесты - Postman

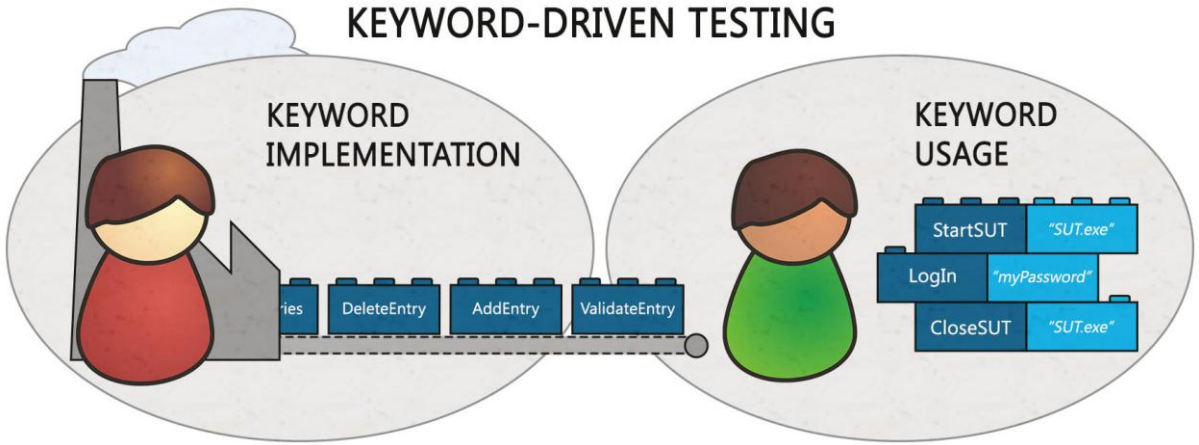
# Подходы к автоматизации тестирования

- **Data Driven Testing (DDT)** – подразумевает под собой использование меняющихся наборов тестовых данных для выполнения одного и того же теста несколько раз



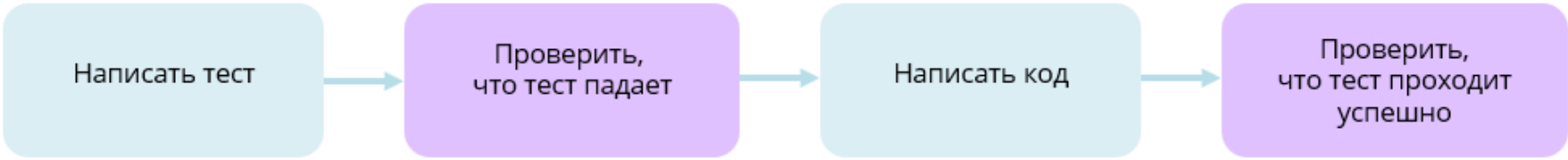
# Подходы к автоматизации тестирования

- **Keyword Driven Testing (KDT)** – подразумевает под собой использование ключевых слов, описывающих набор действий, нужных для выполнения конкретного шага тестового сценария



# Подходы к автоматизации тестирования

- **Test Driven Development (TDD)** – подход разработки через тестирование, предполагает организацию автоматического тестирования посредством написания модульных, функциональных и интеграционных тестов, определяющих требования к коду перед написанием кода



# Подходы к автоматизации тестирования

- **Behaviour Driven Development (BDD)** – это методология для разработки программного обеспечения посредством непрерывного обмена примерами между разработчиками, QA и BA
- Обычно используется Gerkin language (человеко-читаемый язык, структура которого делает его понятным и для машины; документация + acceptance criteria)

```
Feature: Serve coffee
  Coffee should not be served until paid for
  Coffee should not be served until the button has been pressed
  If there is no coffee left then money should be refunded

Scenario: Buy last coffee
  Given there are 1 coffees left in the machine
  And I have deposited 1$
  When I press the coffee button
  Then I should be served a coffee
```

- Примеры фреймворков на основе BDD подхода: https://cucumber.io, https://jbehave.org

# Подходы к автоматизации тестирования

- **Hybrid Framework** – это концепция, в которой мы используем преимущество различных подходов к автоматизации тестирования
- Например: Data Provider в рамках фреймворка TestNG - элемент DDT подхода

```
//This method will provide data to any test method that declares that its Data Provider
//is named "test1"
@DataProvider(name = "test1")
public Object[][] createData1() {
 return new Object[][] {
   { "Cedric", new Integer(36) },
   { "Anne", new Integer(37)},
 };
}

//This test method declares that its data should be supplied by the Data Provider
//named "test1"
@Test(dataProvider = "test1")
public void verifyData1(String n1, Integer n2) {
 System.out.println(n1 + " " + n2);
}
```

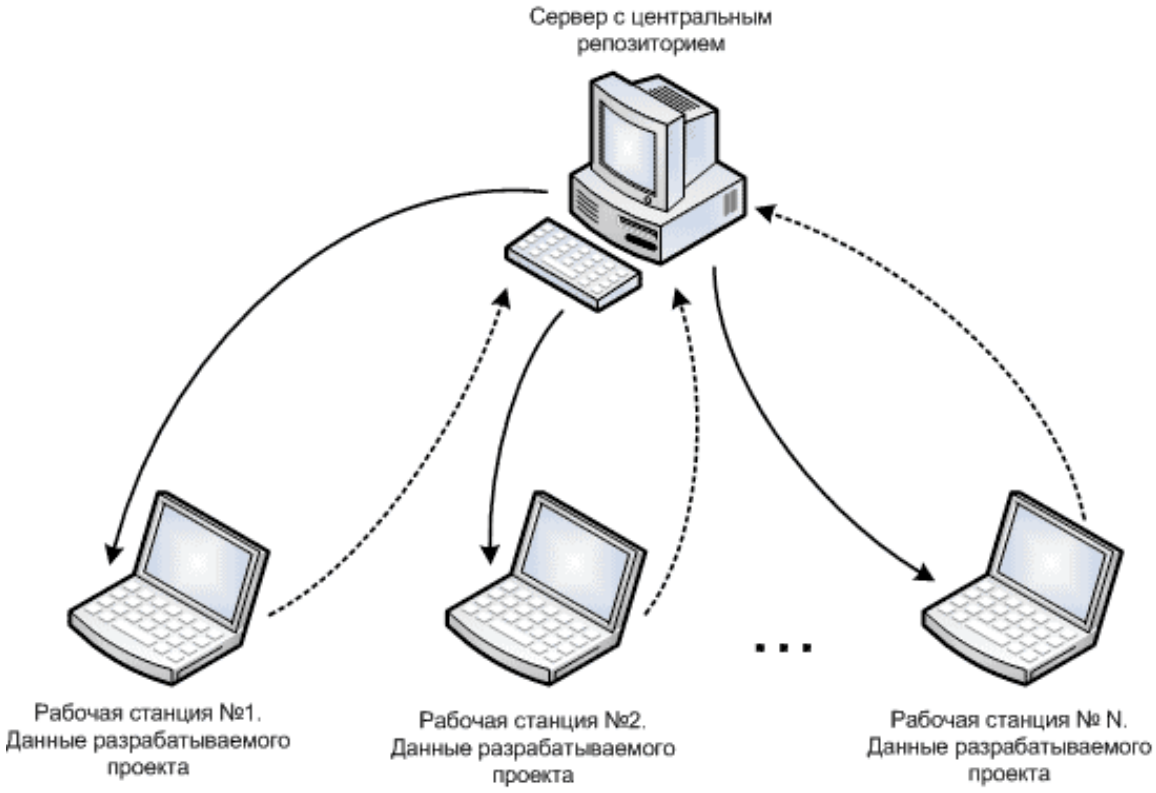
# Системы контроля версий

# 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 (также можно хранить документацию - реже)
- 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

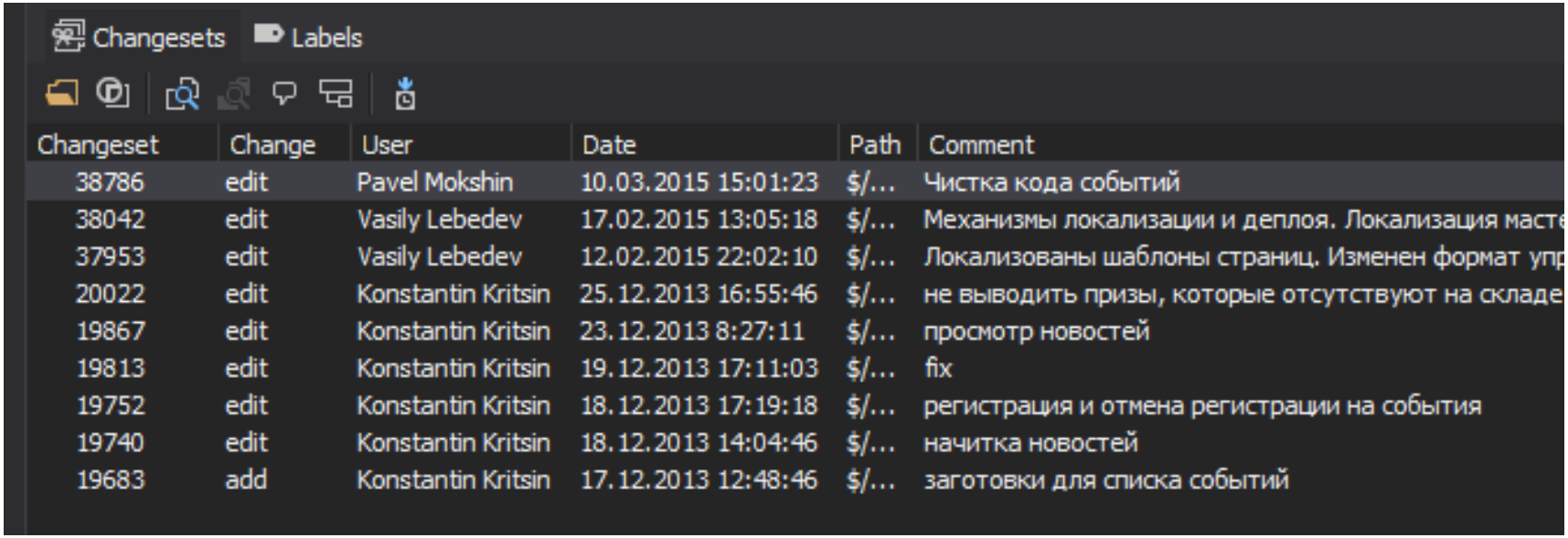


# Зачем они нужны?

- 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?

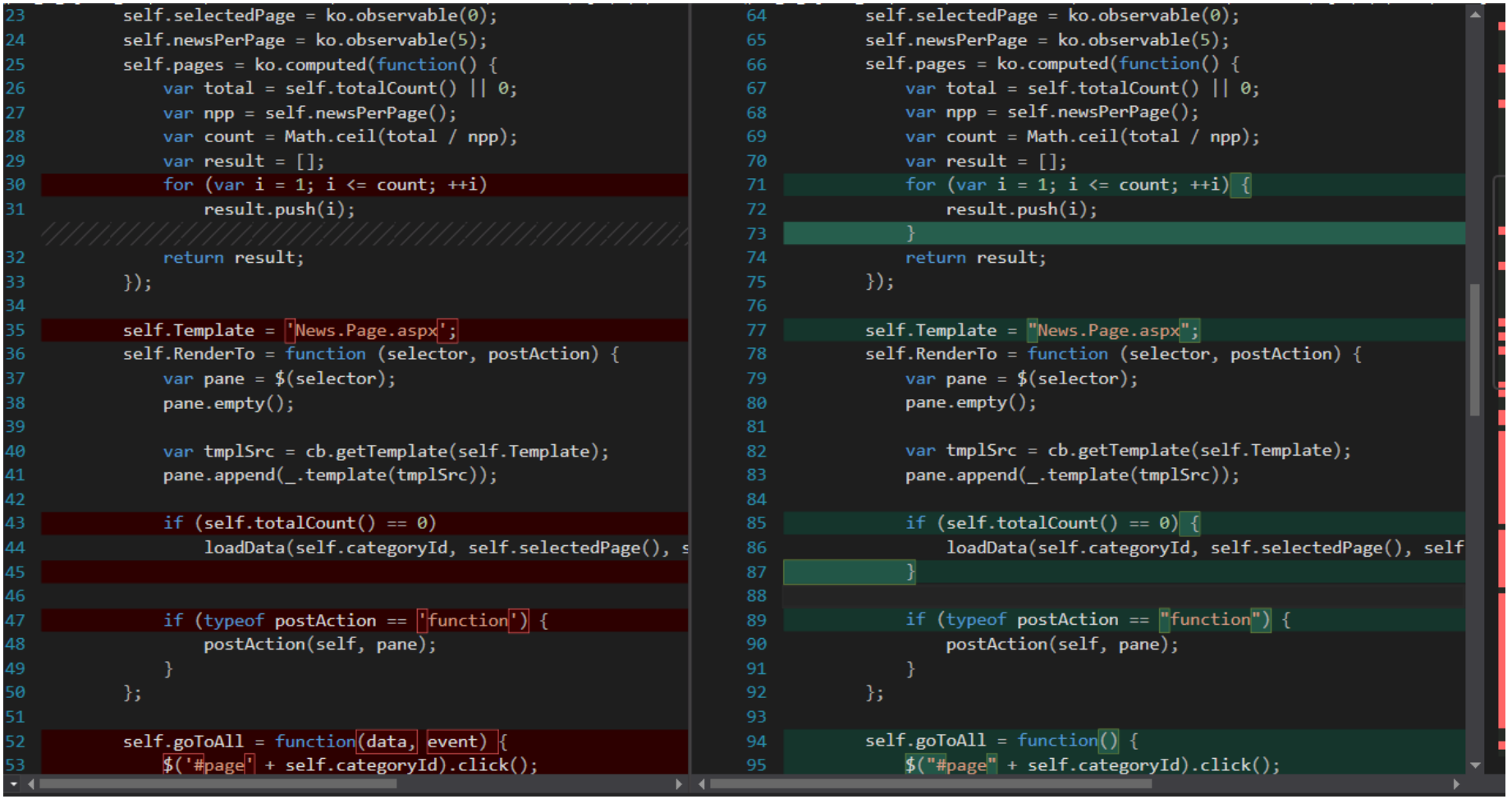
- Возможность узнать информацию кто, когда, зачем и какие изменения вносил в файлы проекта
  - Например, можно понять, кто знает что делает код в этих файлах
  - По комментарию можно понять что именно менялось
  - По каждому изменению можно посмотреть отличия




| 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 tmplSrc = cb.getTemplate(self.Template);
41          pane.append(_.template(tmplSrc));
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();
```

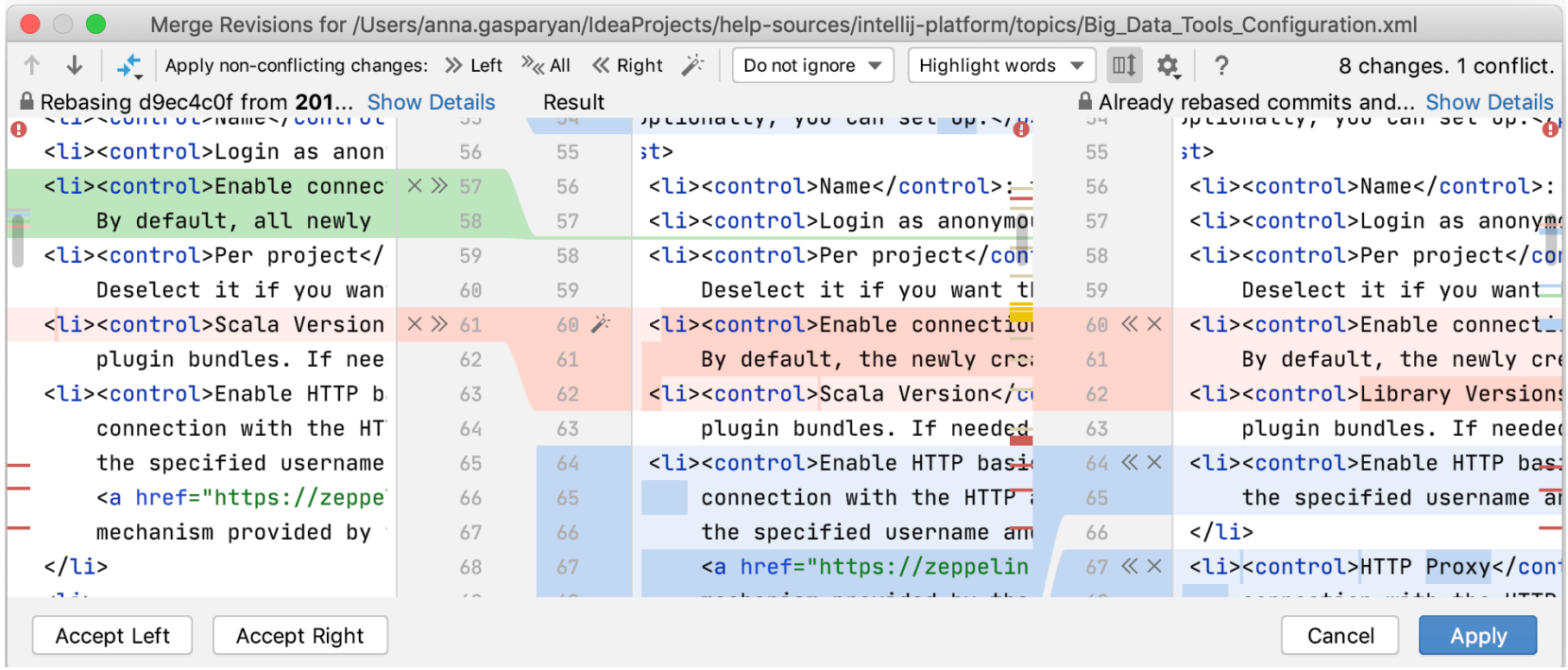
```
64      self.selectedPage = ko.observable(0);
65      self.newsPerPage = ko.observable(5);
66      self.pages = ko.computed(function() {
67          var total = self.totalCount() || 0;
68          var npp = self.newsPerPage();
69          var count = Math.ceil(total / npp);
70          var result = [];
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 tmplSrc = cb.getTemplate(self.Template);
83          pane.append(_.template(tmplSrc));
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 удается, то человек должен разрешить конфликт вручную – выбрать одну из конфликтующих версий, либо обе, либо может вообще написать свой вариант конфликтующего участка файла

# Слияние версий (Merge)

- В IntelliJ IDEA есть встроенный редактор для решения merge-конфликтов:



# 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 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

# Git

- **Git** – очень распространенная распределенная система
- В основном это консольная утилита, но есть и графические интерфейсы (например, GitHub Desktop, Sourcetree и др.)
- https://techrocks.ru/2020/04/24/best-git-gui-for-mac-linux-windows/
- 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 туториал 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
- Расширенная статья: https://habr.com/ru/post/451662/

# Github

- Это крупный бесплатный веб-сервис для хранения Git-репозиториев
- Там можно бесплатно создавать свои репозитории
- Для управления им предоставляется удобный веб-интерфейс, а также Windows приложение (GitHub Desktop)
- 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**

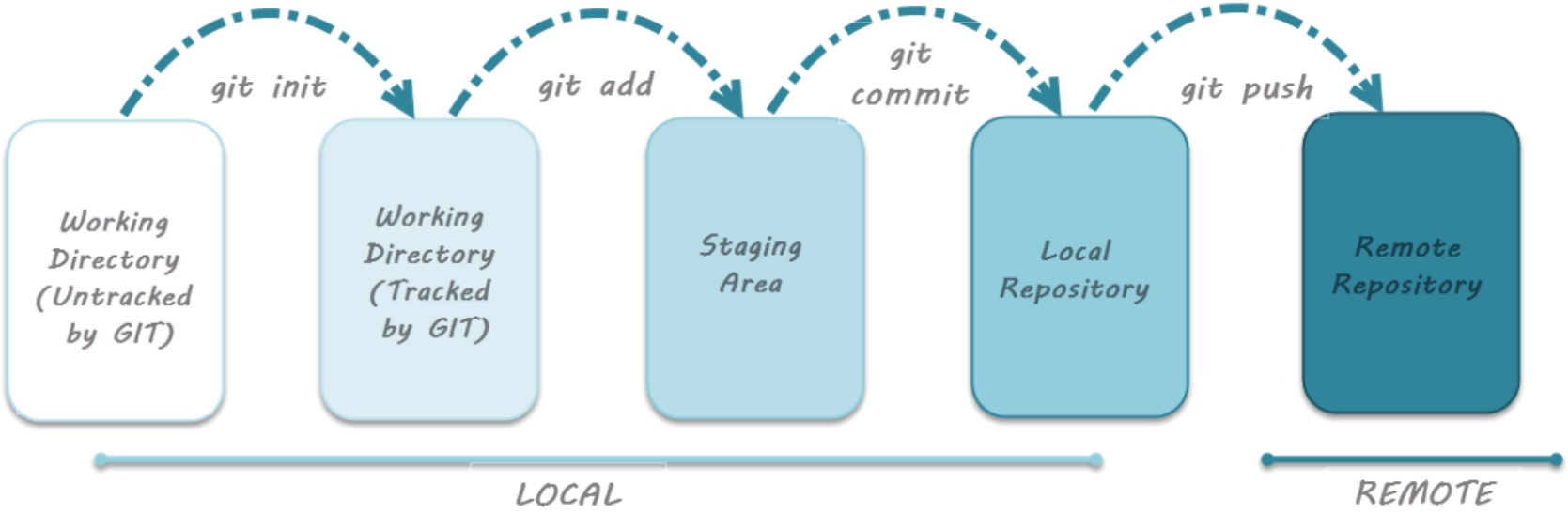
# Операции git

- **Клонирование (clone)**
- Создание локального репозитория, как копии от удаленного репозитория
- *git clone [url_удаленного_репозитория]*

- **Add/remove**
- Удаление/добавление файлов под контроль git
- Нужно делать при добавлении в проект новых файлов и при удалении файлов из проекта
- *git add [имя_файла]*
- *git add .* - добавить все измененные файлы в staging area

# Операции git

- **Зачем нужен add?**
- **Working directory** – рабочий каталог на вашем компьютере
- **Staging area** – область подготовленных файлов или рабочая область



# Операции git

- **Коммит - commit**
- Формирование набора изменений
- Выбранные изменения попадают в набор изменений (он тоже называется **commit**), но пока не вносятся в репозиторий
- *git commit –m “Сообщение”*
- Чтобы внести изменения в удаленный репозиторий, нужно выполнить команду **push**: *git 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

# Практика с git через консоль

- Демонстрация создания нового репозитория и связывания его с удаленным репозиторием

# Операции git

- **Pull**
- Получение изменений из удаленного репозитория в свой локальный репозиторий
- *git pull*
- Состоит из двух команд:

  *git fetch*

  *git merge*

# Практика с git через консоль

- Демонстрация команды git pull

# .gitignore

- В файле .gitignore указываются файлы, которые git должен игнорировать, не отслеживать в папке репозитория.
- Эти файлы не будут добавляться к staging area и, следовательно, не будут коммититься и пушиться в репозиторий
- Файл .gitignore работает для той папки, где он находится и для вложенных папок, поэтому обычно лежит в корневом каталоге репозитория

# Как создать репозиторий в Github и создать проект в IDEA

# Мануал по Github

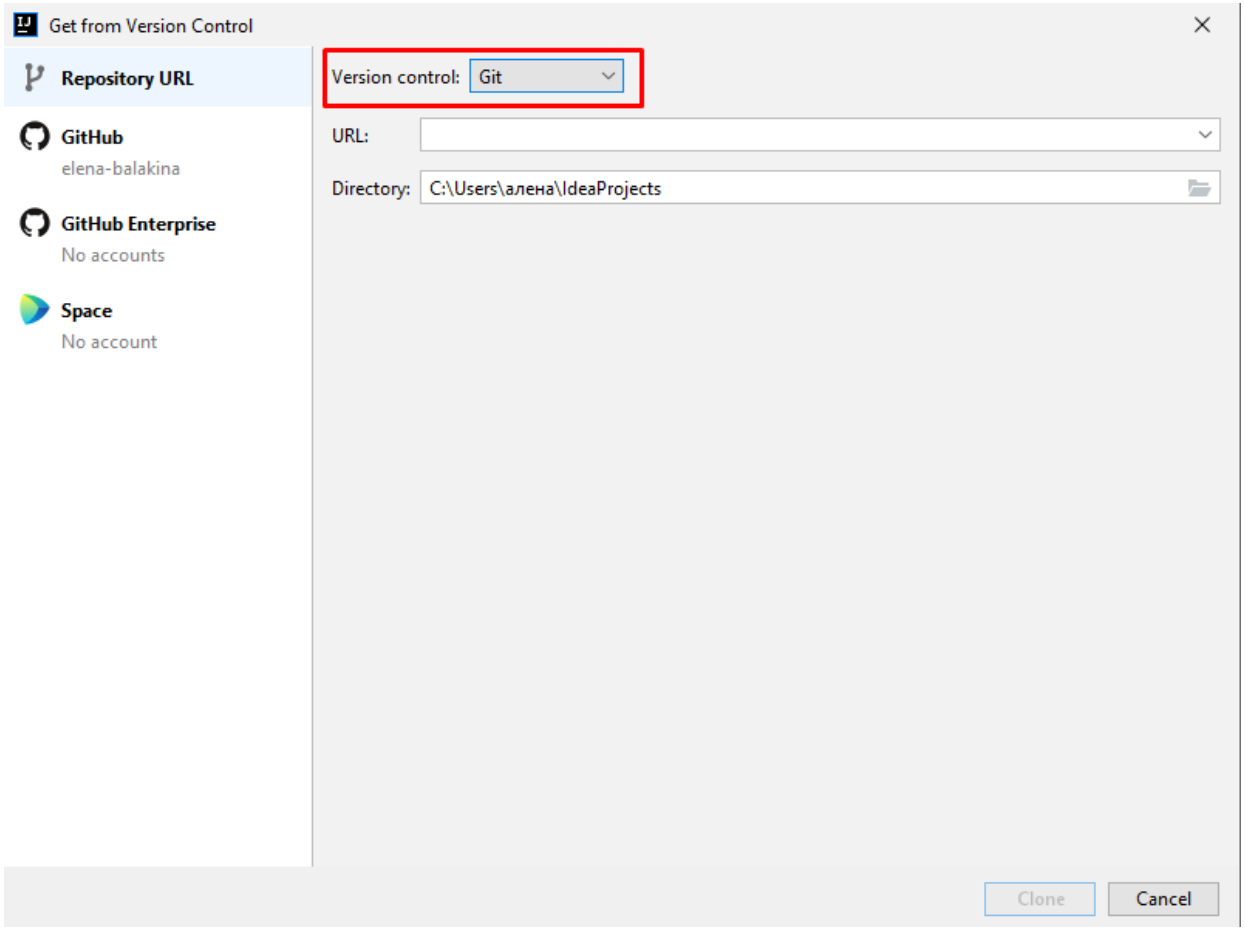
1. Регистрируемся на https://github.com
2. После регистрации и входа заходим в настройки профиля, вписываем свое имя, жмем Update Profile
   https://github.com/settings/profile
3. Создаем новый репозиторий (кнопка **+**, **New Repository**), даем ему имя **academit**.
   В списке **«Add .gitignore»** выбираем Java
4. Устанавливаем **git**

# Мануал по Github

4. Для Windows пользователей:
   - Идем на сайт https://git-for-windows.github.io/, жмем **Download**, устанавливаем все оставляем по умолчанию
- Для UNIX пользователей:
  - Ставим git:
    - sudo apt-get install git

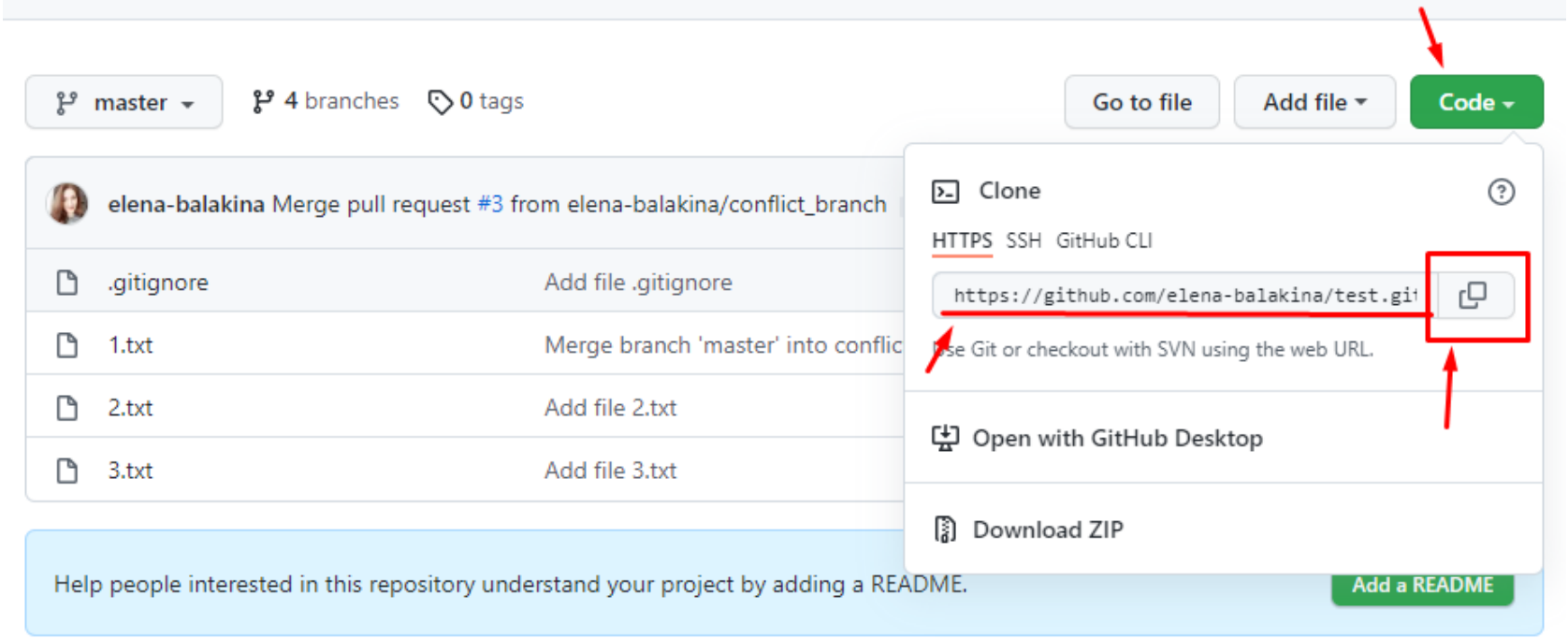
# Клонирование проекта в IDEA

1. Закрываем текущий проект: **File** -> **Close Project**
2. Выбираем **File -> New -> Project from Version Control** -> **Git**



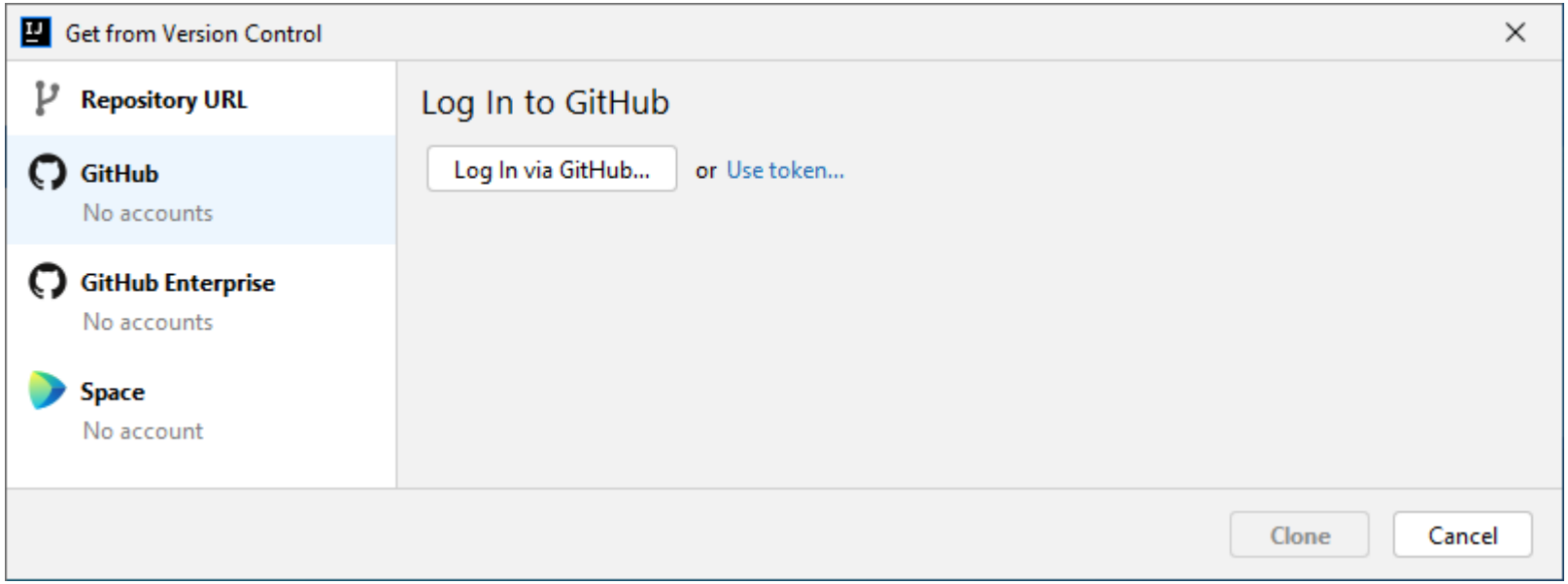
# Клонирование проекта в IDEA

3. Вводим свои логин и пароль **Github**
4. Вводим URL репозитория на **Github**, который хотим выкачать Его можно получить на Github



# Вкладка Github

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



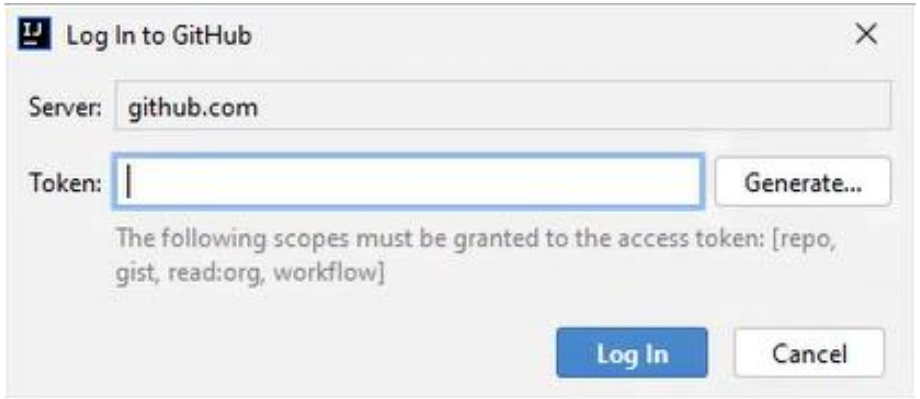
# Вариант Log In to Github

- 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**

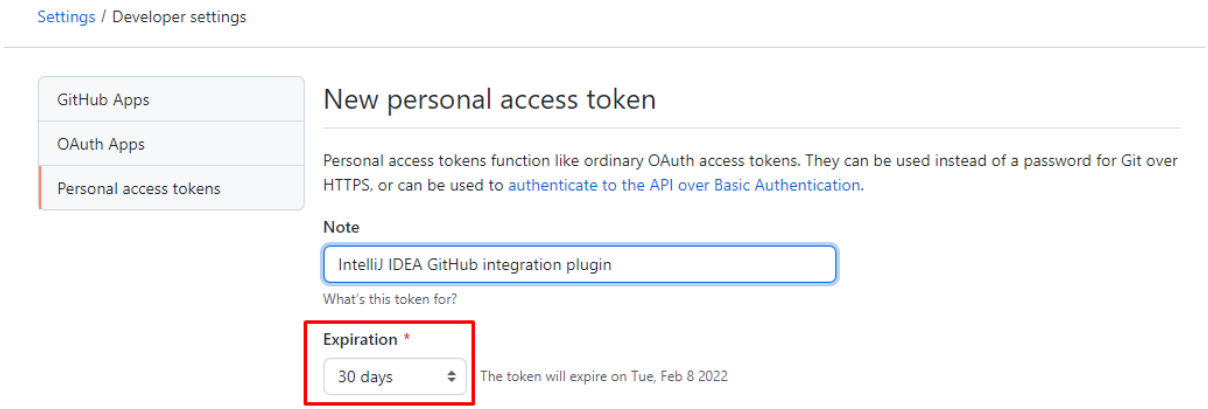
# Вариант Use token

- 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**



# Вариант Use token

- На странице уже будут выбраны все нужные права доступа
- https://github.com/settings/tokens
- При желании можно поменять срок истечения токена




- Нажмите **Generate token**, а дальше скопируйте его и вставьте в **IDEA**
- Дальше можно просто выбрать репозиторий и нажать **Clone**

# Клонирование проекта в IDEA

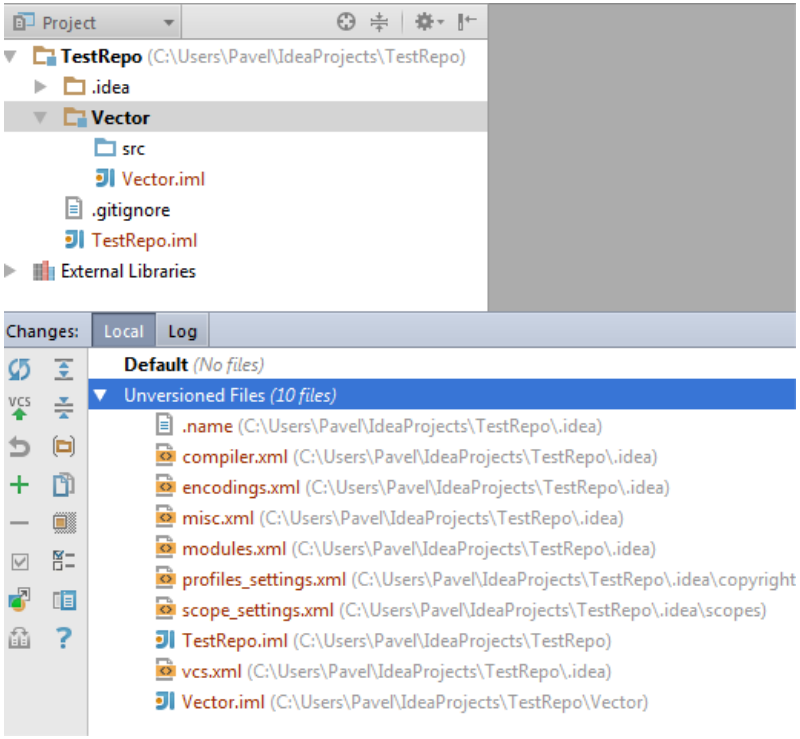
- Выбираем папку, где будет наша копия проекта
- Жмем **Clone**, затем далее, далее и т.д.

# Работа с git через IDEA

- Нужно добавить под управление Git все новые файлы (меняется их цвет)
- Вкладка **Git**
- Настройка вида вкладки Git:
  **File -> Settings -> Version Control -> Commit -> Use non-modal commit interface**
  Добавится панель Commit слева
- Пункт меню **VCS** -> **Git**
- Вкладка Git -> **Console** (команды, которые выполняет IDEA)
- Вкладка Git -> **Log** (история коммитов, ветки)

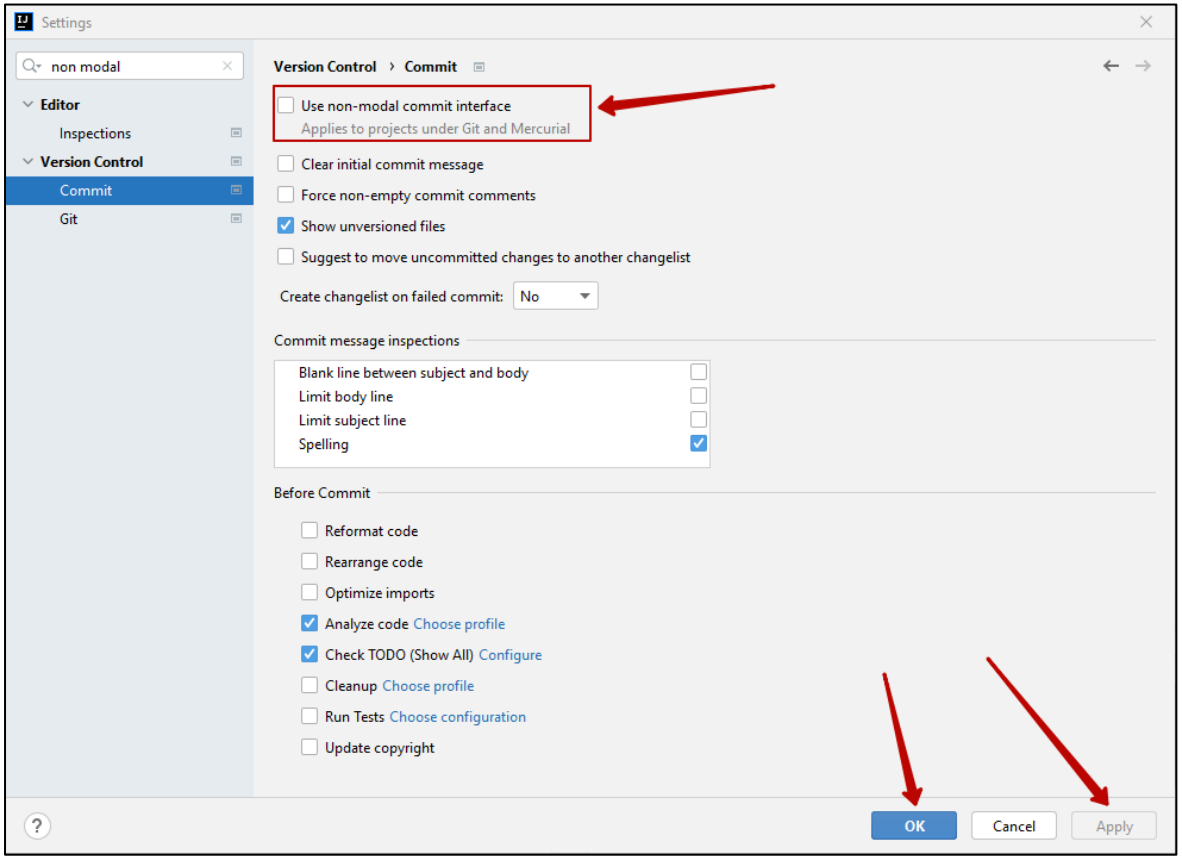
# Добавление новых файлов в проект

- 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** в окне Changes. Выделяем их 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

# Работа с git через IDEA

- 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

# Работа с git через IDEA

- Демонстрация работы с Git через IntelliJ IDEA

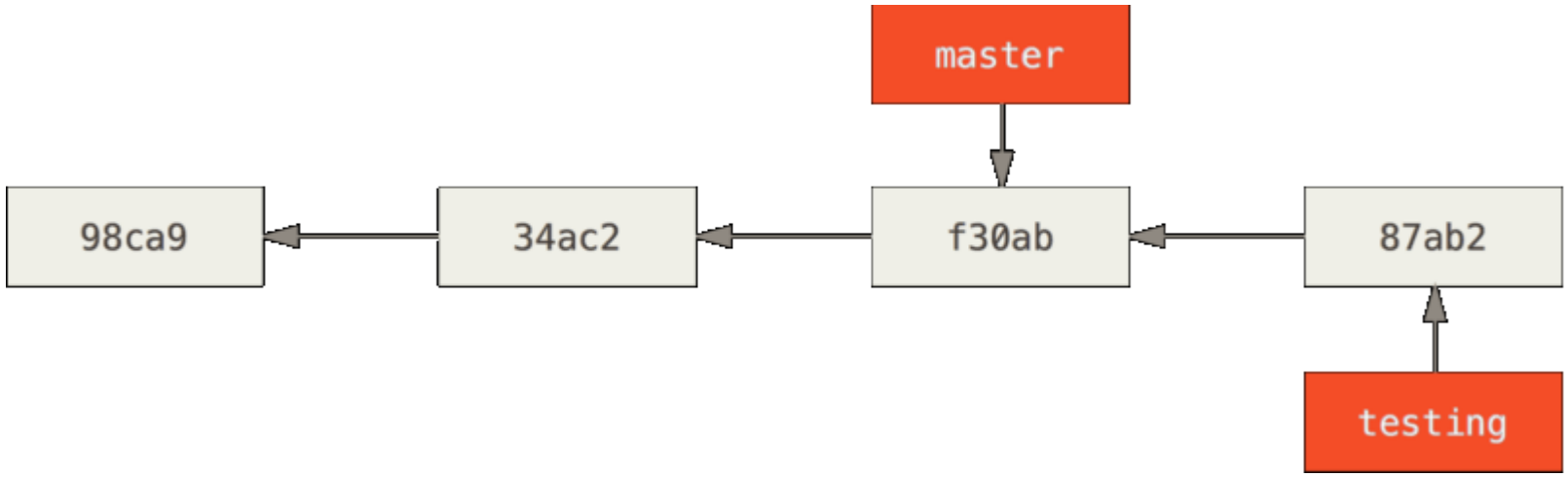
# Практическое задание 1

- Внести изменения в текстовый файл и сделать запись изменений на удаленный репозиторий.
  На github убедиться, что изменения записались.
- Отредактировать файл, закоммитить и запушить изменения.
  На github убедиться, что изменения записались.
- Внести изменения в файл на удаленном репозитории через github, применить pull для «стягивания» изменений в локальный репозиторий
  Убедиться, что файл в локальном репозитории отредактировался

# Модели работы с Git

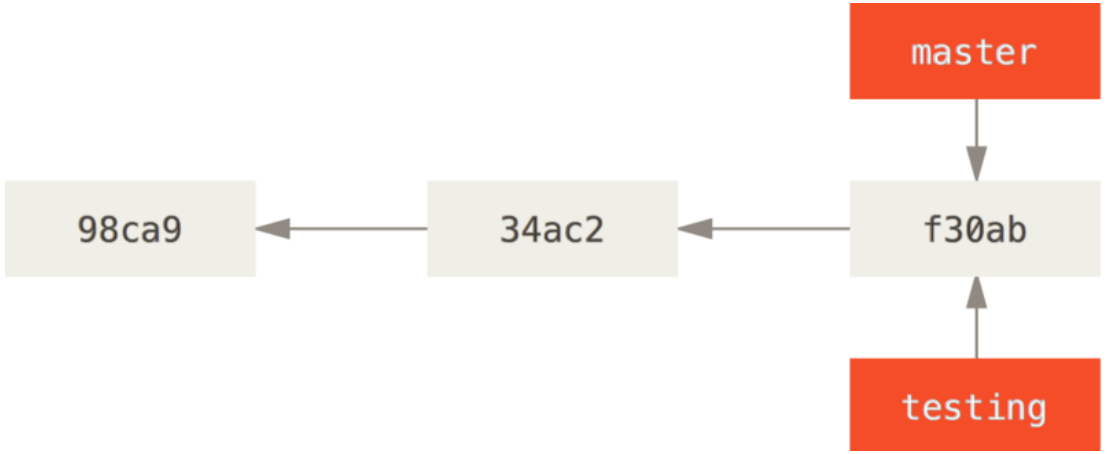
# Ветки

- 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



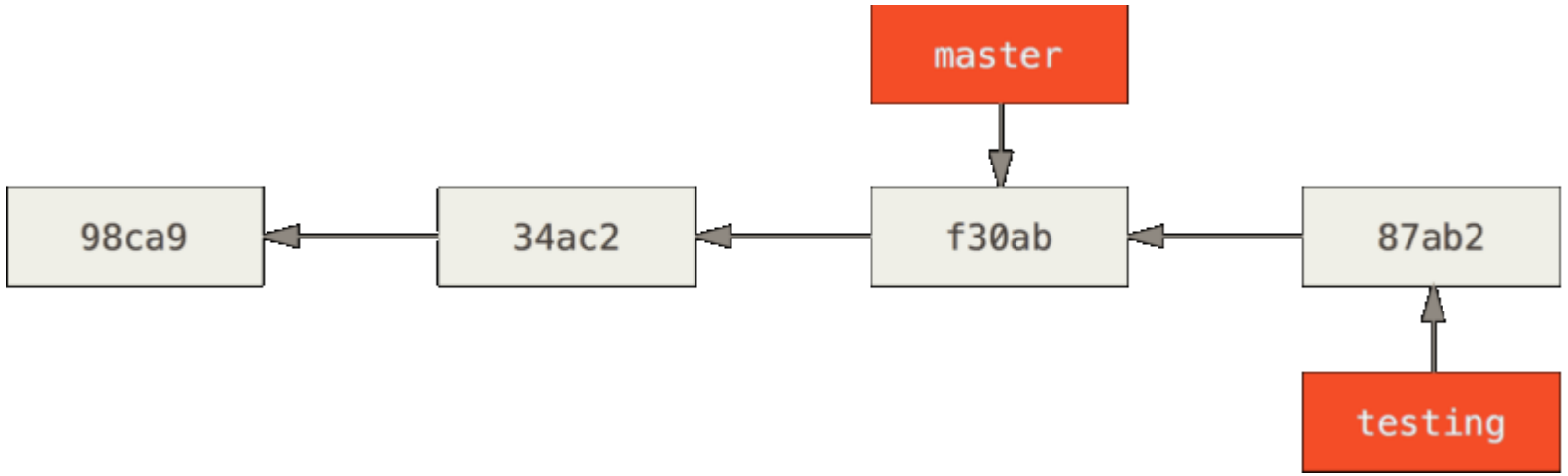
# Ветка master

- Когда мы создаем репозиторий, там сразу создается 1 ветка с именем **main (master)**
- Дальше мы можем создавать новые ветки
- Например, создаем ветку **testing** от ветки **master**
- Пока в них находятся одинаковые версии кода, так как не было ни одного нового коммита



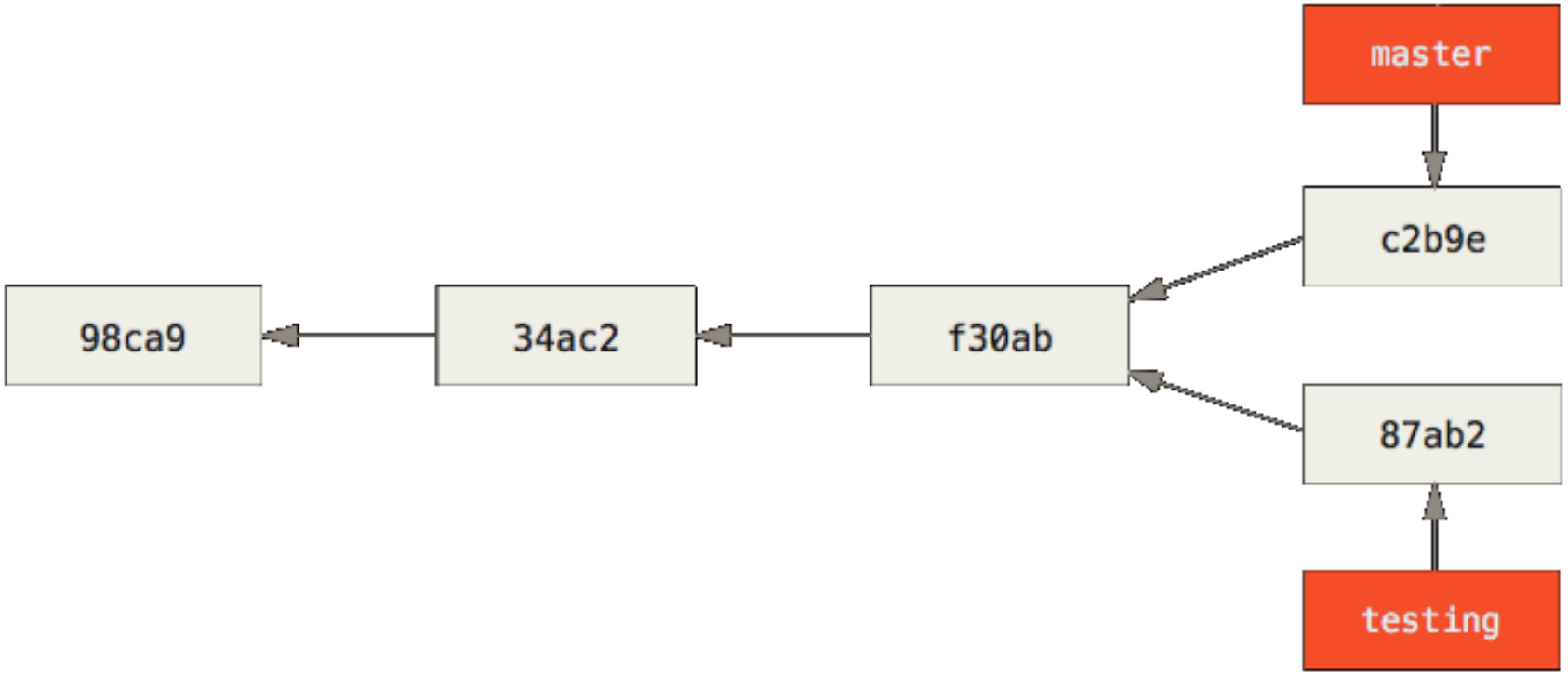
# Ветки

- 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

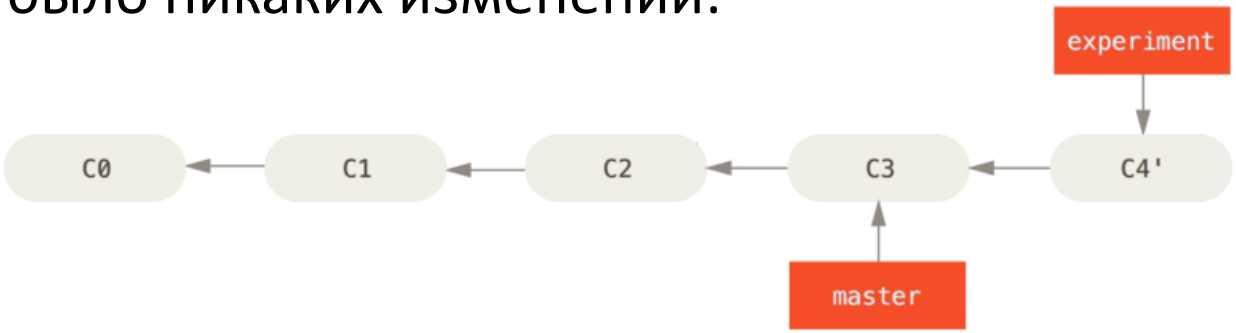


# Слияние веток

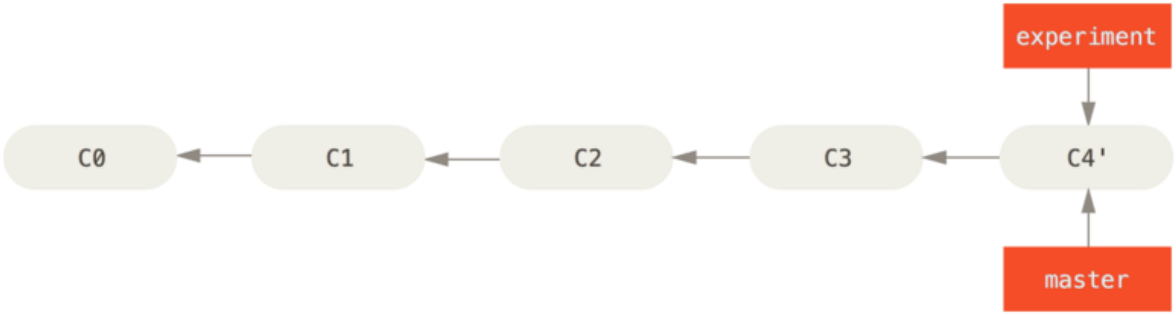
- Часто при работе в одной из веток нужно перенести изменения в другую ветку
- Это делается при помощи **слияния** (**merge**)
- Основы ветвления и слияния: https://git-scm.com/book/ru/v2/Ветвление-в-Git-Основы-ветвления-и-слияния
- Бывает 2 вида слияний: **fast-forward** и **three way merge**

# Слияние веток

- Рассмотрим слияние **fast-forward**
- Fast-forward слияние можно сделать, когда в ветке мастер не было никаких изменений:

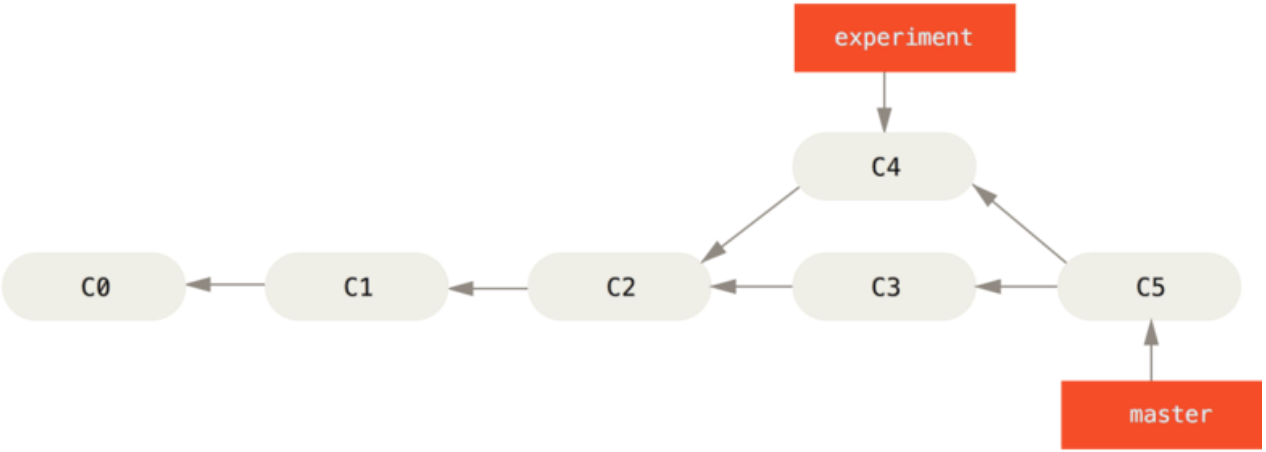



- Просто передвигаем указатель master вперед



# Слияние веток

- Рассмотрим слияние **three way merge**
- Он возможен для случаев, когда в ветке master были коммиты или когда их не было
- Создается **merge-commit** (дополнительный коммит, у него 2 родителя)



# 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

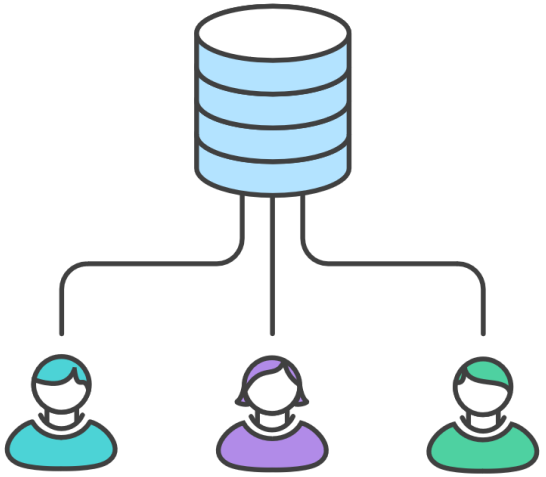
# Выбор модели работы с Git

- 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

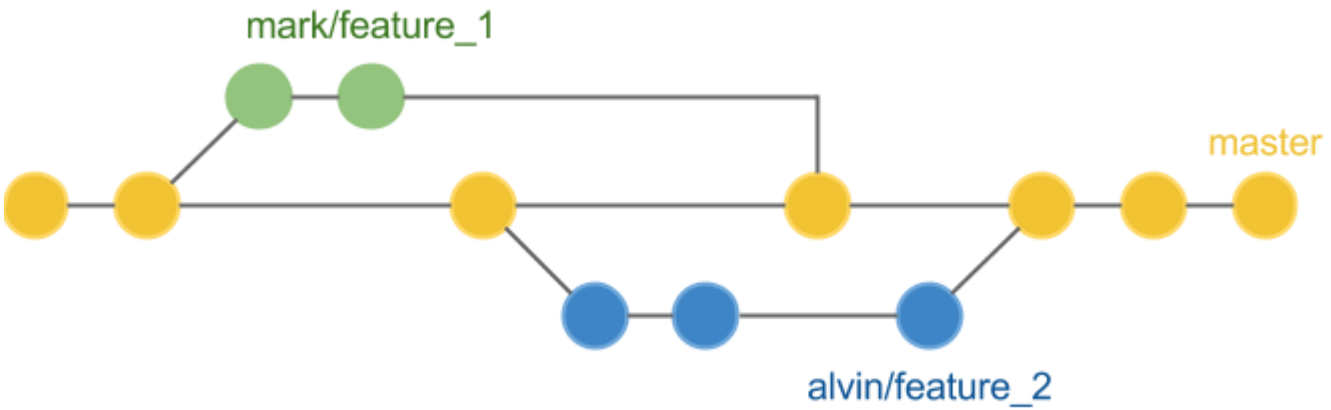
- Все пользователи работают только с одной веткой – **master**
- Все выкачивают себе копию кода, работают с ней, и потом сразу же вносят изменения в центральный репозиторий
- Т.е. работа ведется так же, как с централизованными VCS
- В курсе мы будем работать по этой модели, она самая простая

- 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 (approve), 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 Worklow

- **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

# Gitflow

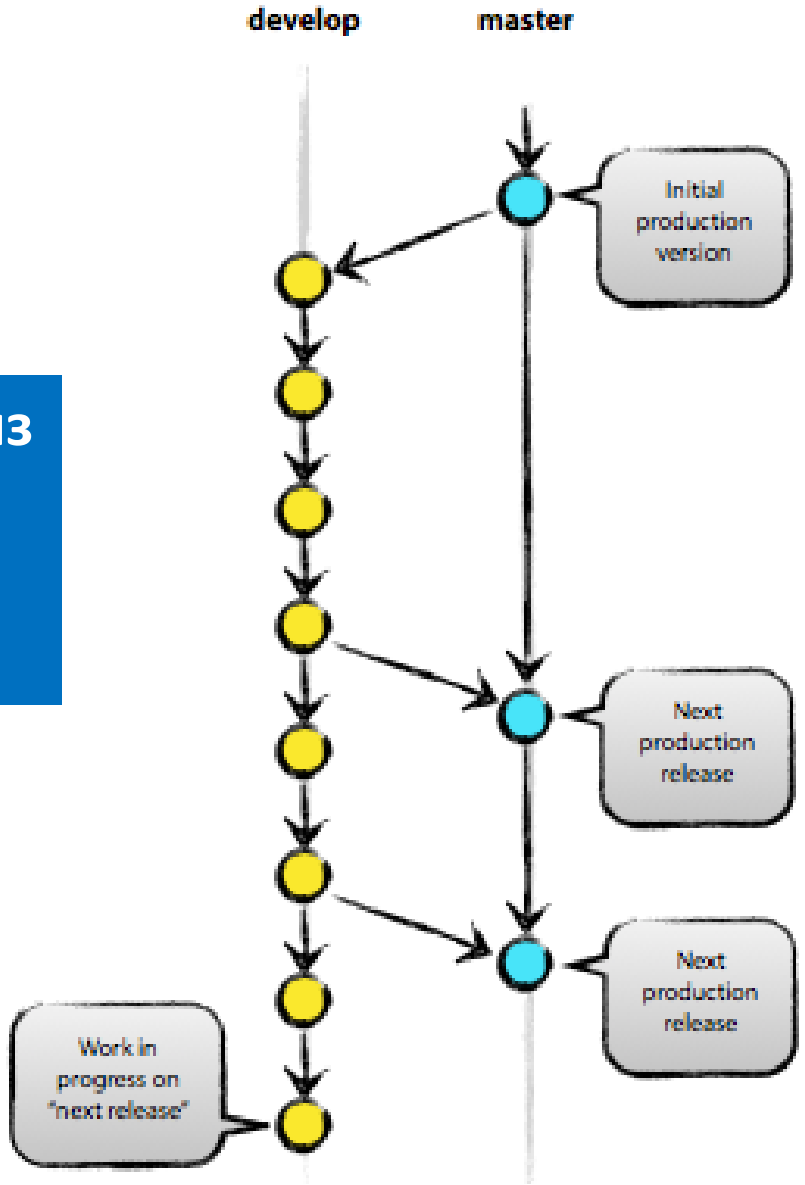
- В проекте заводятся две постоянные ветки – **master** и **develop**
- Вообще, это обычные ветки, имена сложились исторически

- **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

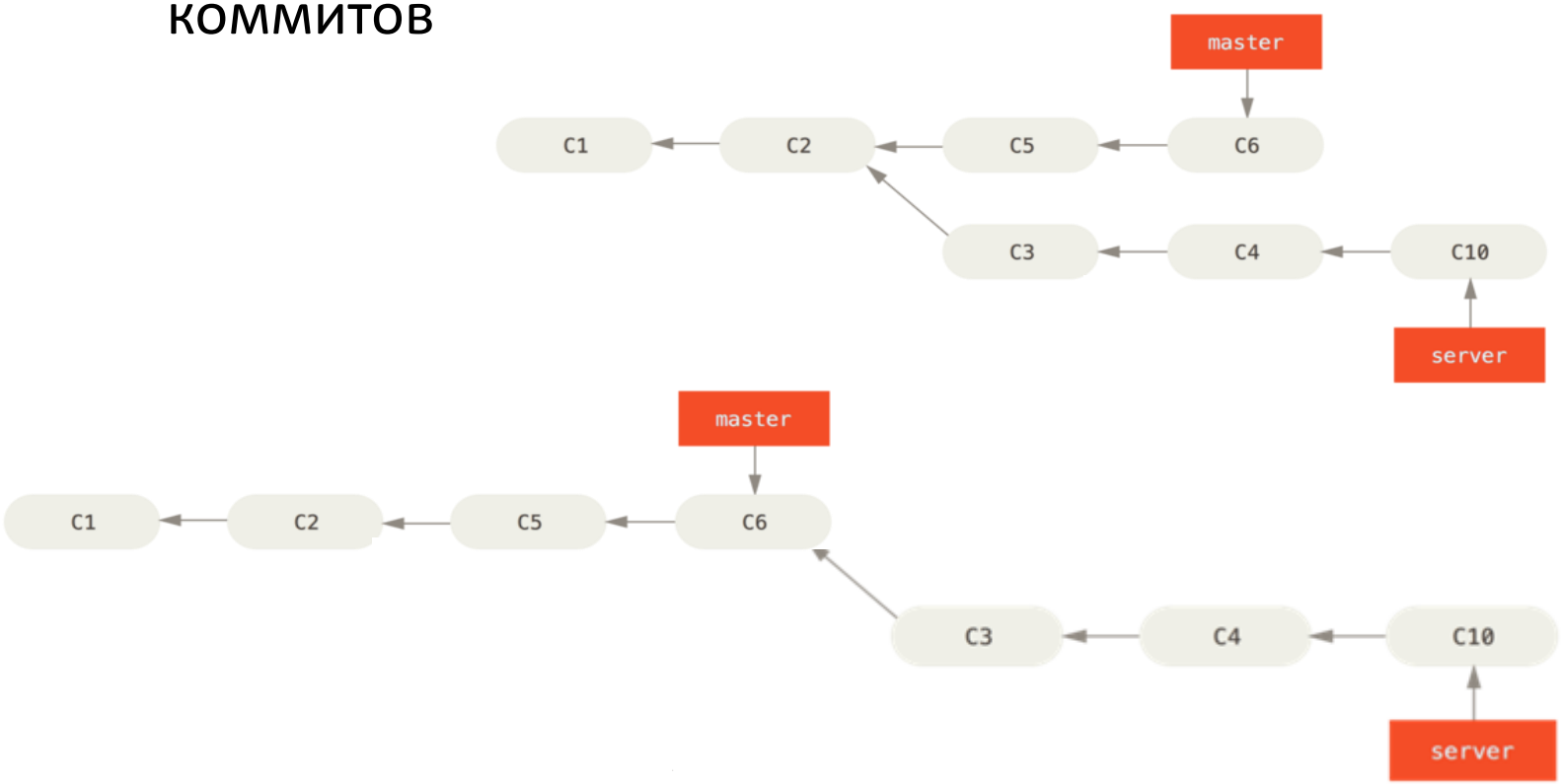
- От **master** делается отдельная временная ветка, исправление/доработка делается в ней
- Такой ветке часто дают название по номеру задачи в багтрекере, например, **feature121** или **bug234**
- В эту ветку постоянно забираются изменения из **master** – другие разработчики тоже работают над своими задачами
- 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**

# Rebase

- **Rebase** (перебазирование) – это другой способ сделать слияние веток
- Команда **rebase** позволяет переписывать историю коммитов



# Работа с git через IDEA

- Демонстрация работы с Git через IntelliJ IDEA
- Создание новых веток
- Merge веток в master

# Практическое задание 2

- Создать новую ветку develop в текущем репозитории
- В ней создать новый файл и добавить в него информацию несколькими коммитами
- Сделать push на удаленный репозиторий
- Проверить наличие второй ветки с коммитами на удаленном репозитории

- Внести изменения на удаленном репозитории, закоммитить их
- Стянуть эти изменения к себе в IDEA

# Домашнее задание «Git»

- **Часть 1. Pull request:**
- Создать репозиторий и выполнить с ним практические задания 1 и 2 с занятия
- Создать pull request на github и замержить ветку, созданную в задании 2, в master

# Домашнее задание «Git*»

**Часть 2. Решение конфликтов:**

- Создать ветку branch_with_conflicts из ветки master
- Отредактировать одну и ту же строчку кода в 1.txt в ветках master и branch_with_conflicts
- Закоммитить и запушить изменения в обе ветки
- Создать pull request для merge'а ветки branch_with_conflicts в master
- Решить конфликты, оставив только информацию из ветки branch_with_conflicts в файле 1.txt
- Завершить merge
- В качестве ответа на ДЗ отправить ссылку на репозиторий на почту academits.autotest@gmail.com