

Лекция №1

Введение в разработку ПО

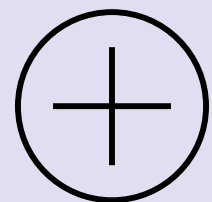
Бобряков Дмитрий
Senior разработчик



Меня хорошо видно и слышно?



Проверить, идет ли запись



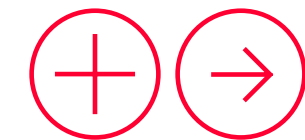
Ставим «+», если все хорошо
«-», если есть проблемы

Дмитрий Бобряков

Senior разработчик

- 7 лет в IT
- Senior-разработчик в BigData Streaming
- Experience: NetCracker, AlphaBank, MWS
- Java-ментор в MWS
- Преподаватель в МФТИ

Содержание лекции



- Что такое разработка?
- Почему важна командная разработка
- Системы контроля версий
- CI/CID

Что такое разработка?

Разработка – это автоматизации определенного бизнес-процесса, решение какой-то проблемы.

Часть 1. Языки программирования

Почему ЯП так много?

- Разные языки заточены под разные задачи?
- Парадигмы программирования отличаются
- Доступность open-source библиотек и фреймворков
- Требование к платформе



- Статическая/динамическая типизация

Статическая типизация гарантирует проверку типов во время компиляции программы.

Динамическая типизация — приём, используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

```
class Main {  
    public static void main(String[] args) {  
        int a = 45;  
        int b = 55;  
        System.out.println(sum(a, b));  
    }  
  
    public static int sum(int a, int b) {  
        return a + b;  
    }  
}
```

Динамическая типизация в Python

```
def sum(a, b):  
    return a + b
```

```
if __name__ == '__main__':  
    a = 45  
    b = 55  
    print(sum(a, b))
```


Плюсы статической типизации

- Типы гарантированно проверяются на этапе компиляции: мы уверены в значениях, которые получаем
- Меньше вероятности допустить ошибку из-за жесткой системы типов
- Ускорение разработки при помощи IDE: подсказки кода

Плюсы динамической типизации

- Простота создания универсальных коллекций типов
- Легкость в освоении

Вопрос к аудитории: лучше статическая или динамическая типизация?

В разработке промышленных решений чаще всего применяют языки со **статической** типизацией

Почему выбирают статическую типизацию?

- Несоответствие типов гарантирует, что программа не запустится. В динамических языках выполнение дойдет до точки, где есть ошибка.
- Безопасность- не придёт число/строка/объект в обработчик
- Сложность предметной области.
- Большое количество кода: `autocomplete` помогает.

Почему именно Java?



Вакансии

Резюме

Компании

🔍

java

3 251 вакансия «java»

Уровень дохода	₽ ▾
<input checked="" type="radio"/> Не имеет значения	
<input type="radio"/> от 100 000 ₽	398
<input type="radio"/> от 200 000 ₽	294
<input type="radio"/> от 300 000 ₽	161
<input type="radio"/> от 400 000 ₽	77
<input type="radio"/> от 500 000 ₽	22
<input type="radio"/> Своя зарплата	

Вакансии

Резюме

Компании

🔍

c#

1 247 вакансий «c#»

Уровень дохода	₽ ▾
<input checked="" type="radio"/> Не имеет значения	
<input type="radio"/> от 90 000 ₽	255
<input type="radio"/> от 180 000 ₽	178
<input type="radio"/> от 270 000 ₽	90
<input type="radio"/> от 355 000 ₽	36
<input checked="" type="radio"/> от 445 000 ₽	16
<input type="radio"/> Своя зарплата	

Вакансии

Резюме

Компании

🔍

c++

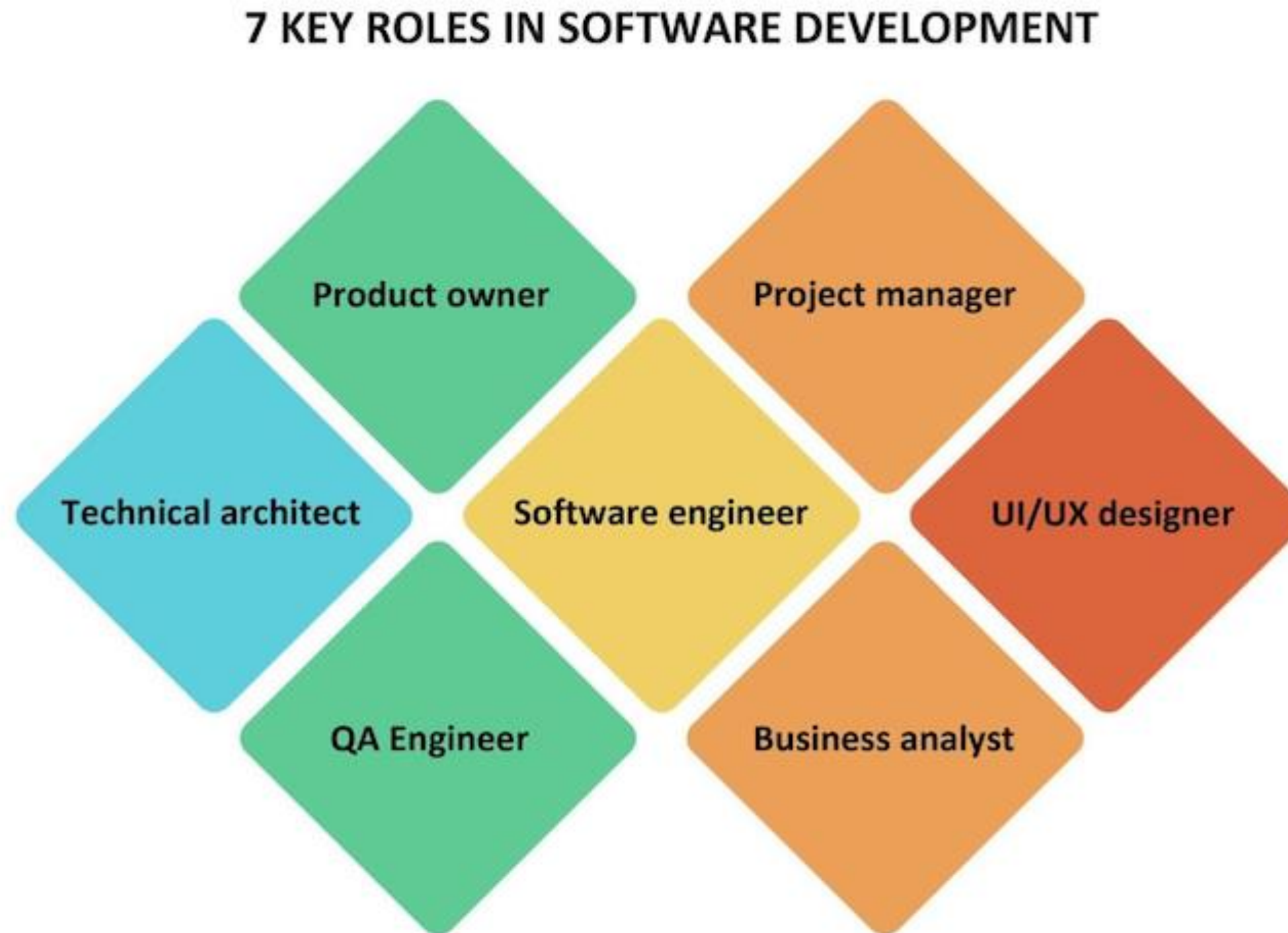
1 616 вакансий «c++»

Уровень дохода	₽ ▾
<input checked="" type="radio"/> Не имеет значения	
<input type="radio"/> от 95 000 ₽	390
<input type="radio"/> от 195 000 ₽	264
<input checked="" type="radio"/> от 290 000 ₽	105
<input type="radio"/> от 390 000 ₽	49
<input type="radio"/> от 485 000 ₽	19
<input type="radio"/> Своя зарплата	

Почему именно Java?

- Сборщик мусора (Garbage Collector)
- Поддерживаемость кода
- Огромное количество библиотек и фреймворков
- Кроссплатформенность

Часть 2. Почему командная разработка важна?

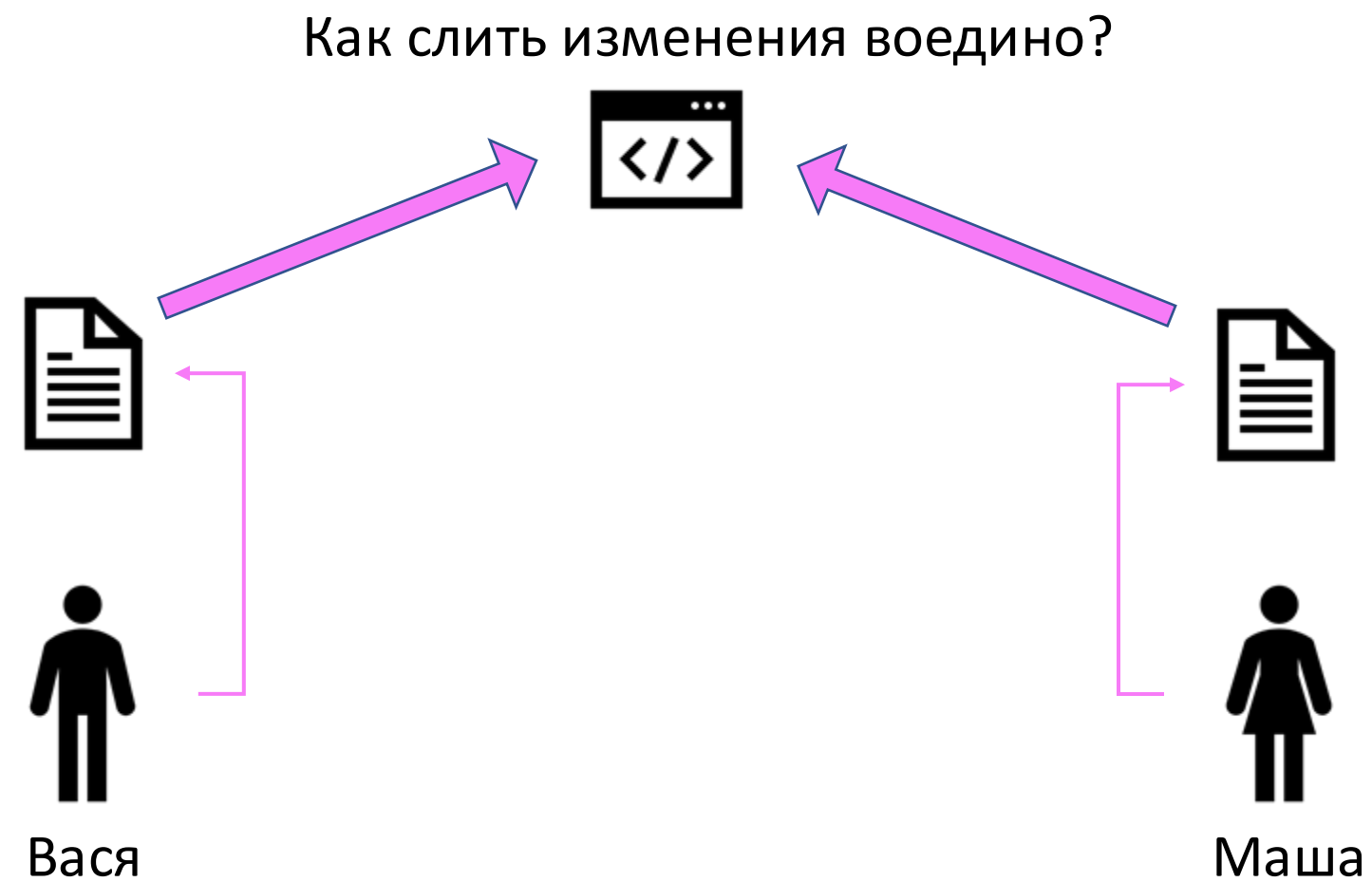


Каждый сделал свою часть, но работа не стыкуется



Часть 3. Системы контроля версий

- Зачем нужны системы контроля версий?



Требования к совместному кодированию

- Процесс должен быть автоматизирован
- Каждый программист должен иметь возможность работать изолированно от других
- Старые изменения нужно сохранять, чтобы откатиться к ним в случае необходимости

История систем контроля версий

- CVS (Concurrent Version System)
- SVN (Apache Subversion)
- Git
- Mercurial

Основы Git- создание репозитория

Инициализация нового репозитория

```
$ git init my-project
```

```
$ cd my-project
```

Клонирование существующего репозитория

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

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

Основы Git- базовые команды


```
# Проверка статуса
$ git status

# Добавление файлов в staging area
$ git add file.txt
$ git add . # все файлы
$ git add *.java # все java файлы

# Создание коммита
$ git commit -m "Initial commit"

# Просмотр истории коммитов
$ git log
$ git log --oneline
$ git log --graph --oneline --all
```

Основы Git- удалённый репозиторий




```
# Добавление удаленного репозитория
$ git remote add origin https://github.com/user/repo.git

# Просмотр удаленных репозитория
$ git remote -v

# Отправка изменений на сервер
$ git push origin master

# Получение изменений с сервера
$ git pull origin master
$ git fetch origin # только загрузка изменений
```

Основы Git-работа с ветками



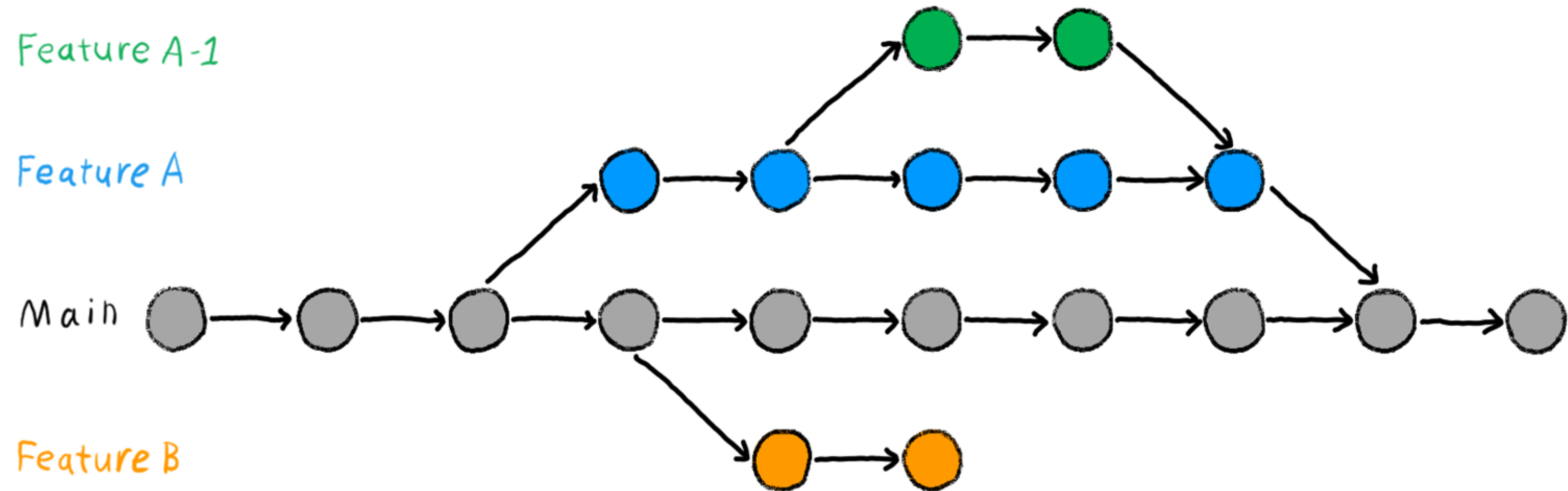
```
# Создание новой ветки
$ git branch feature-branch

# Переключение на ветку
$ git checkout feature-branch
# или
$ git switch feature-branch

# Создание и переключение за одну команду
$ git checkout -b feature-branch
# или
$ git switch -c feature-branch
```

1. **Main/Master** – основная ветка для production кода
2. **Develop** – ветка для разработки
3. **Feature branches** – ветки для разработки новых функций
4. **Release branches** – ветки для подготовки релиза
5. **Hotfix branches** – ветки для срочных исправлений

Git branches



Pull Request



Unwrap a validator instance of specified type contained in SpringValidatorAdapter #37119

[Code](#)

[Open](#) zpavloudis wants to merge 2 commits into `spring-projects:main` from `zpavloudis:expose-inner-validator-gh-37081`

Conversation 10 Commits 2 Checks 0 Files changed 2

+26 -0

Changes from all commits File filter Conversations

0 / 2 files viewed

[Review in codespace](#)[Review changes](#)

Filter changed files

spring-boot-project/spring-boot-a...

main/java/org/springframework/b...

ValidatorAdapter.java

test/java/org/springframework/bo...

ValidatorAdapterTests.java

14 ...ure/src/main/java/org/springframework/boot/autoconfigure/validation/ValidatorAdapter.java

Viewed

@@ -39,6 +39,7 @@

39 *

40 * @author Stephane Nicoll

41 * @author Phillip Webb

42 * @since 2.0.0

43 */

44 public class ValidatorAdapter implements SmartValidator, ApplicationContextAware, InitializingBean, DisposableBean

{

@@ -153,4 +154,17 @@ private static Validator wrap(Validator validator, boolean existingBean) {

153 return validator;

154 }

155

156 }

39 *

40 * @author Stephane Nicoll

41 * @author Phillip Webb

42 + * @author Zisis Pavloudis

43 * @since 2.0.0

44 */

45 public class ValidatorAdapter implements SmartValidator, ApplicationContextAware, InitializingBean, DisposableBean

{

154 return validator;

155 }

156

157 + @Override

158 + @SuppressWarnings("unchecked")

159 + public <T> T unwrap(Class<T> type) {

160 + if (type.isAssignableFrom(this.target.getClass())) {

161 + if (this.target instanceof SpringValidatorAdapter adapter) {

162 + return adapter.unwrap(type);

163 + }

164 + return (T) this.target;

165 + }

166 +

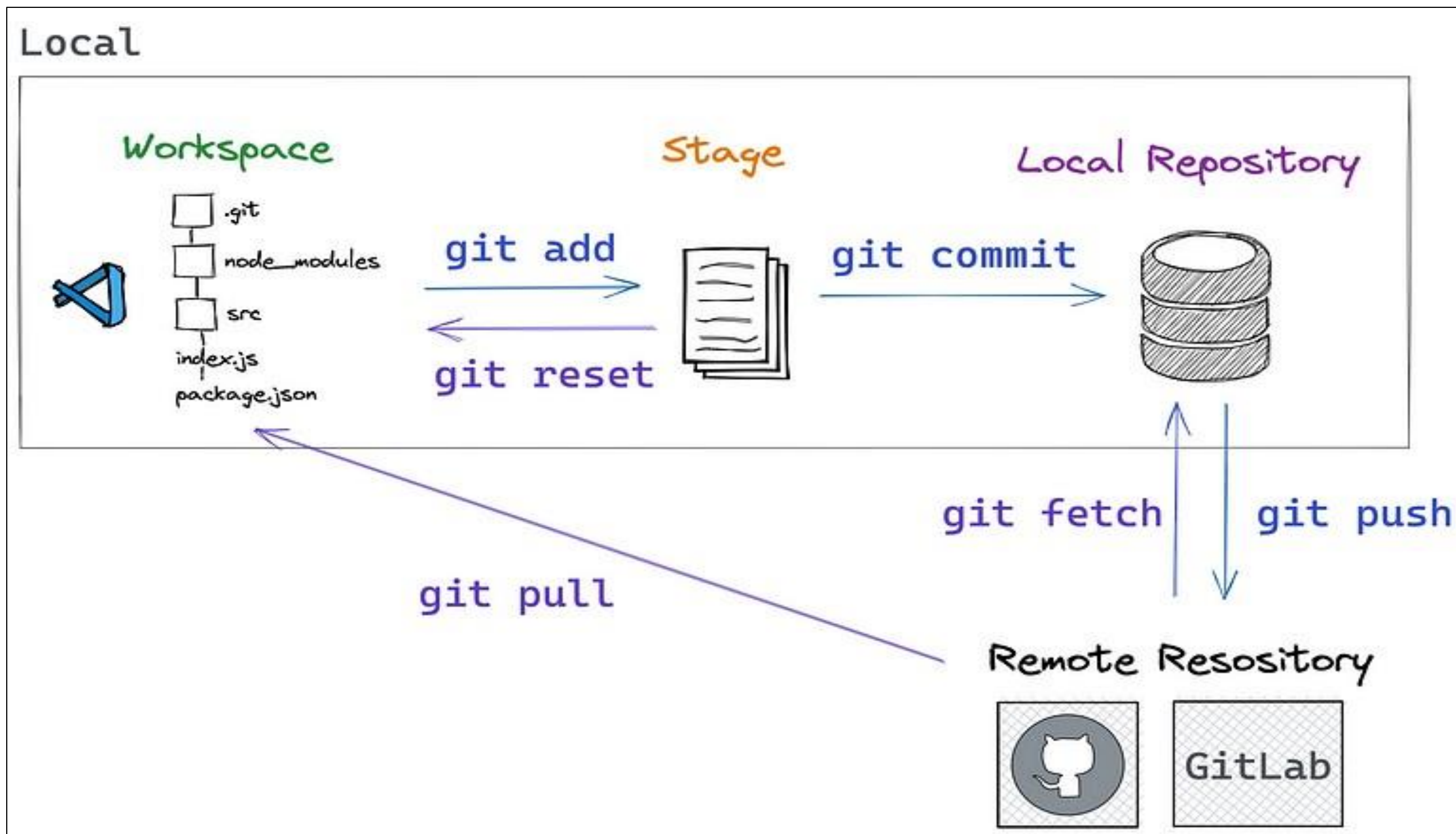
167 + throw new IllegalArgumentException("Cannot unwrap " + this.target + " to " + type.getName());

168 + }

169 +

170 }

Принцип работы Git



1. Частые маленькие коммиты
2. Осмысленные сообщения коммитов
3. Одна логическая единица изменения на коммит
4. Тестирование перед коммитом
5. Использование .gitignore

1. Коммит больших бинарных файлов
2. Force push в общие ветки
3. Неполные сообщения коммитов
4. Игнорирование конфликтов

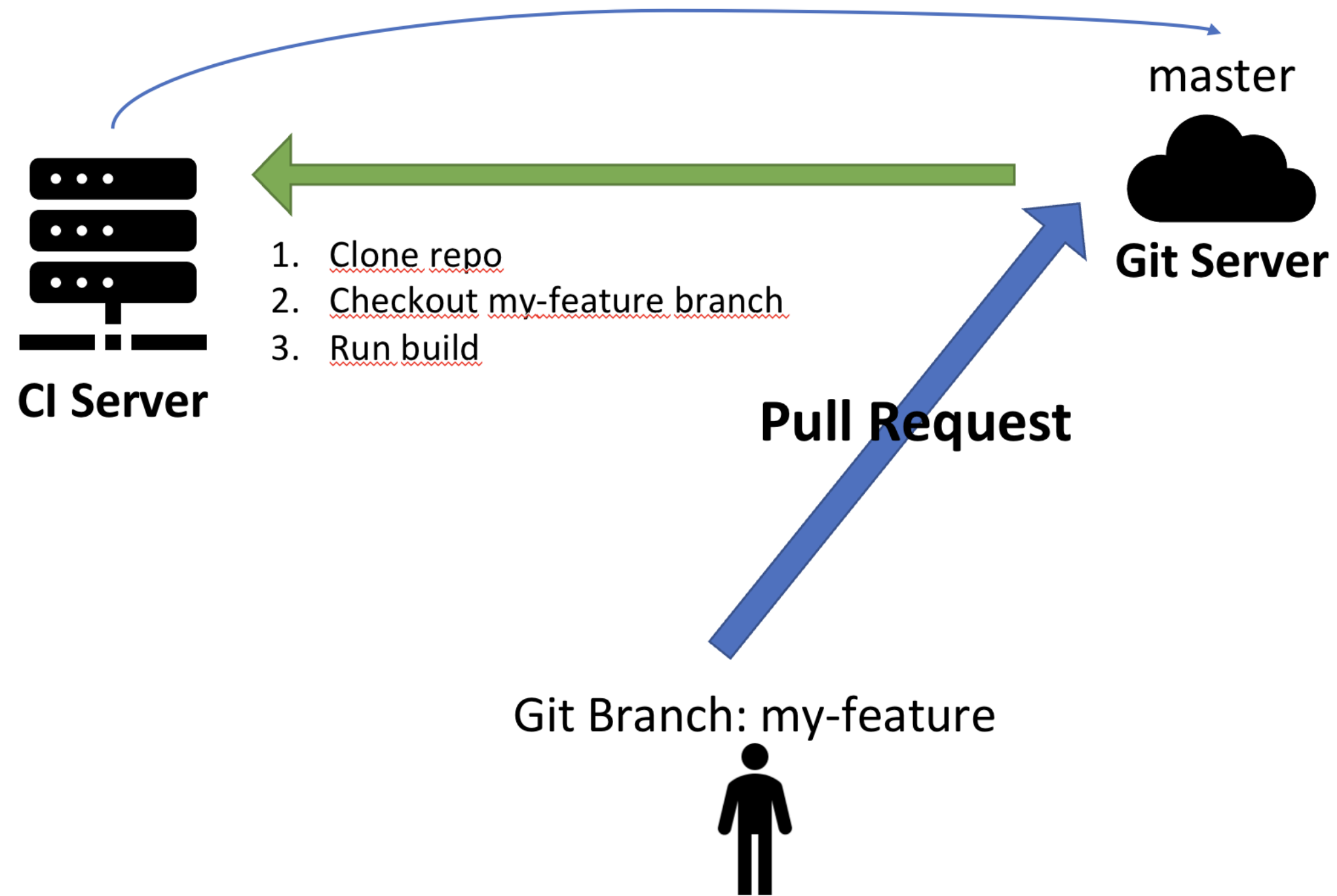
Continuous Integration

Процесс непрерывной интеграции изменений нескольких разработчиков в единое целое (вливание PR-ов в master-ветку)

Требования к CI

- Код в master всегда должен компилироваться
- Тесты должны выполняться
- Все дополнительные проверки должны быть успешны (статический анализ)

Принцип CI



Конфигурация CI на примере GitHub Actions

```
name: Simple CI job
```

```
on:  
  push:  
    branches: [ "master" ]  
  pull_request:  
    branches: [ "master" ]
```

```
jobs:
```

```
  build:
```

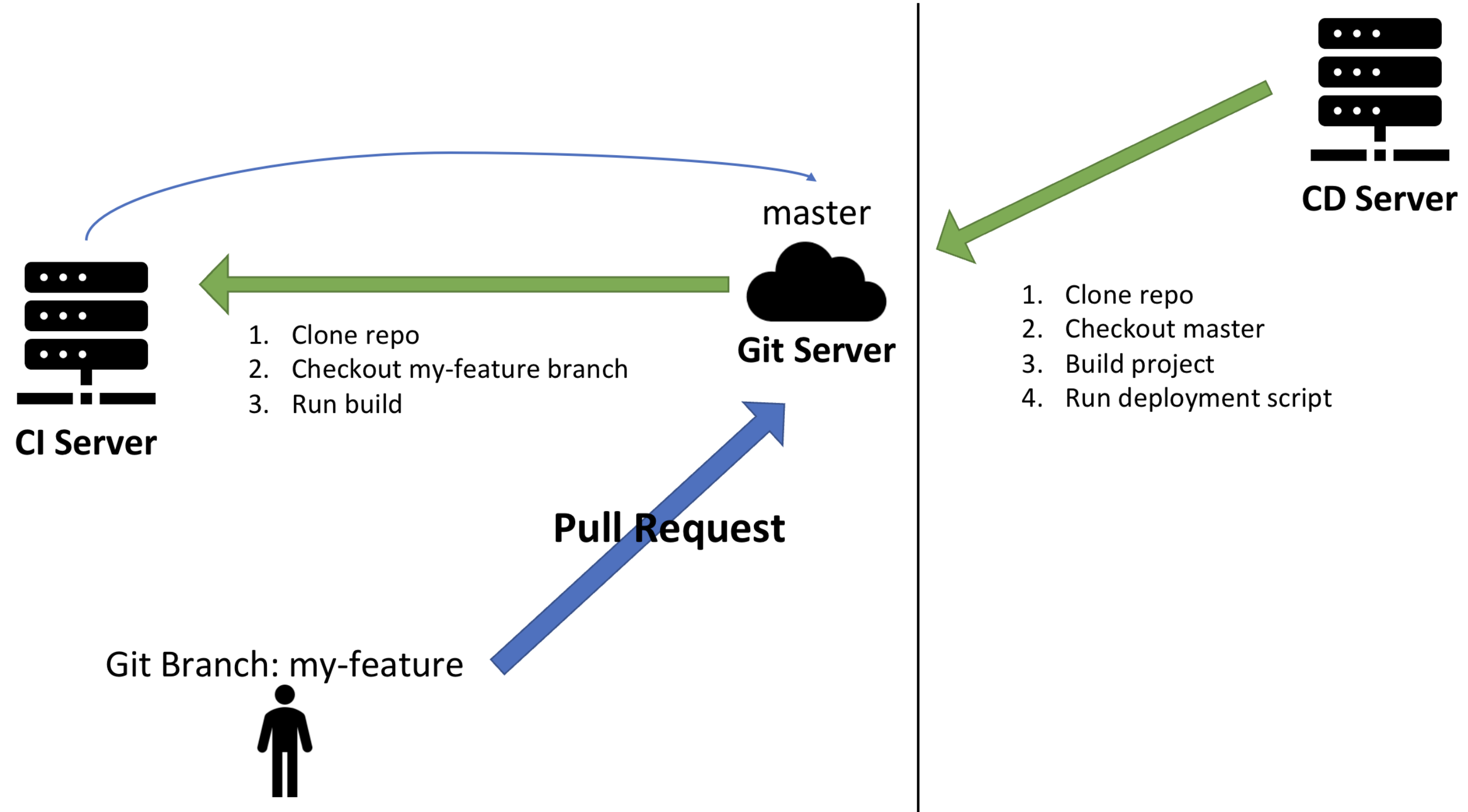
```
    runs-on: ubuntu-latest
```

```
    steps:  
      - name: Print message to console  
        run: |  
          echo "Hello, world!"  
          echo "Hello, everyone!"
```


- Создаете репозиторий. Ветка `master/main` будет создана автоматически
- Добавляете `yaml`-файл с конфигурацией CI-сборки
- Разработчики клонируют репозиторий
- Каждый создает себе отдельную ветку
- Потом создаем `pull request` из своей ветки в `master`
- Если все собралось, нажимаем `merge`, и изменения попадают в `master`

Процесс непрерывной доставки изменений на среду развертывания.

Принцип CD



Какие есть серверы CI/CD?

.GitHub Actions

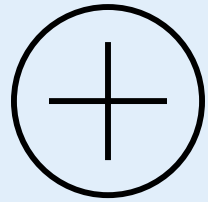
.GitLab CI

.Travis CI

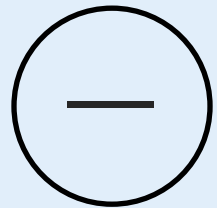
.Jenkins

- Разработка – это процесс автоматизации бизнеса
- В современном мире разрабатывать ПО без команды почти невозможно
- Системы контроля версий – ваш друг
- CI/CD – важная и неотъемлемая часть современной разработки

Вопросы?



Ставим «+»,
если есть вопросы



Ставим «-»,
если нет вопросов