

3

🔥 0

0

💩 0

Разработка через тестирование на простом примере

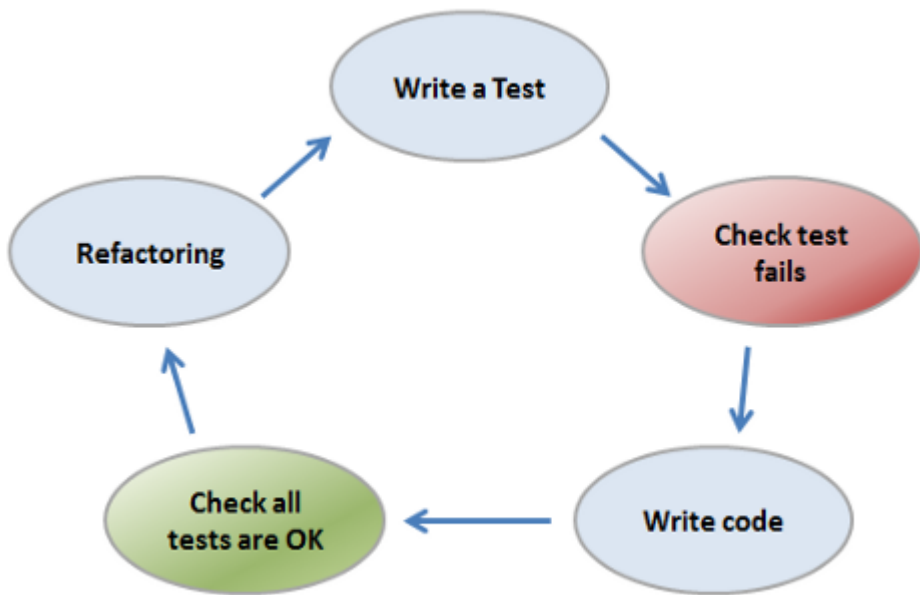
Разработка через тестирование начинается с юнит-тестов, а не с кода. Из части **Agile** она доросла до самостоятельной дисциплины.

♥ 3

 Обсудить

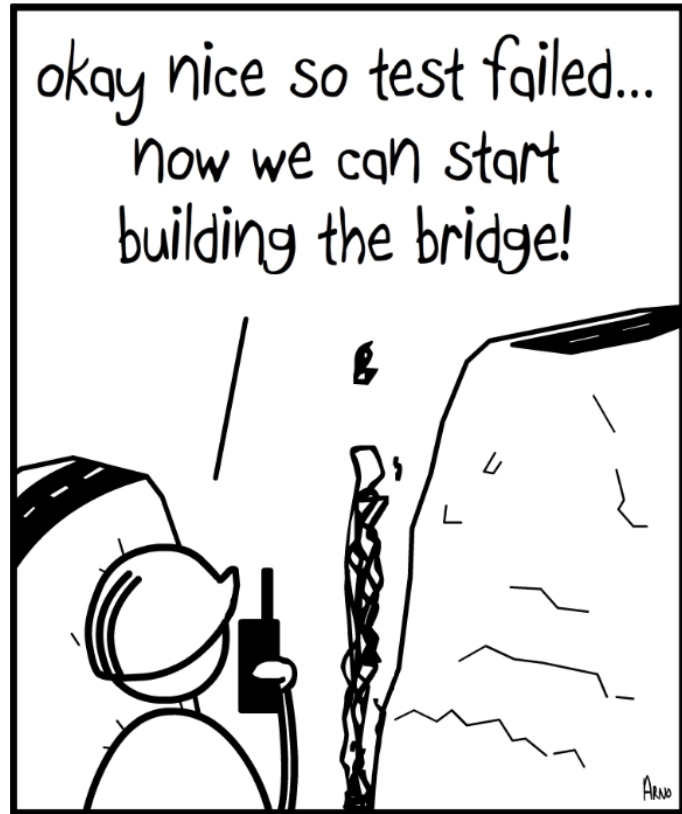
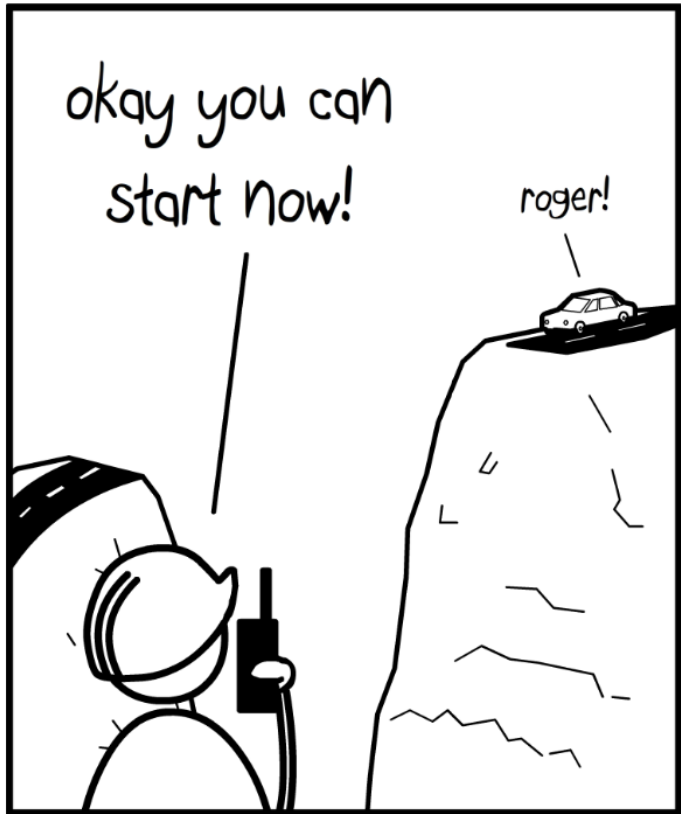
3

Introduction to Test Driven Development



Упомянутое движение прогрессировало в последние 15 лет. Оно вело к новым прагматичным практикам быстрого создания продукта. Традиционные методы предлагают проводить юнит-тестирование в конце. Такой подход показал неэффективность: тесты адаптируются под код, а не наоборот, или вообще не пишутся.

Тогда зачем тратить на них время?



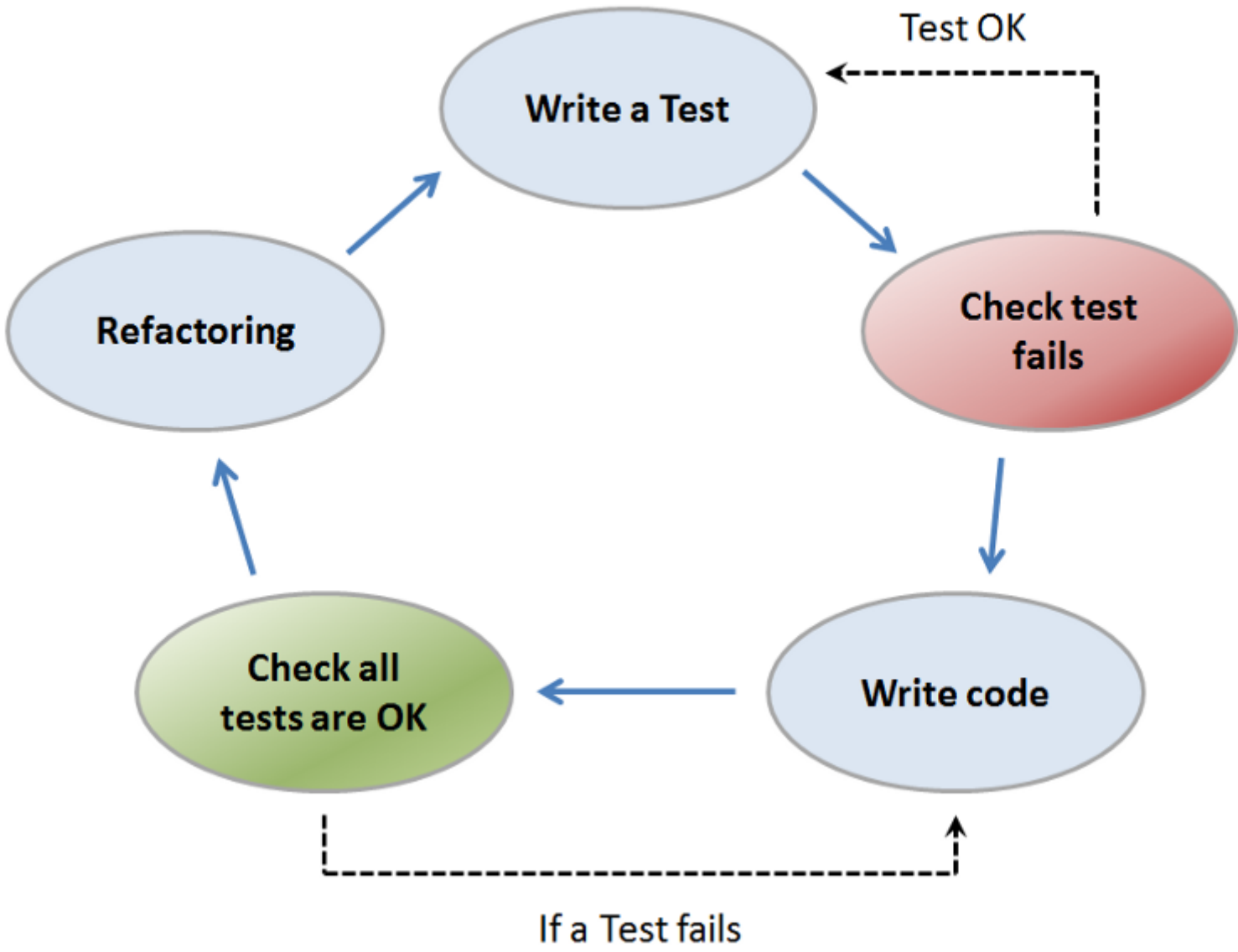
Для метода **TDD** ответ очевиден: сегодняшние вложения в тесты дадут вознаграждение завтра при добавлении новой функциональности и **рефакторинге**. Метод предлагает писать юнит-тесты перед кодом. Идея появилась в середине 1990-х, а в 2003 опубликовали книгу **Экстремальное программирование**. Она объясняет понятие непрерывного рефакторинга для улучшения кода продукта.

Принципы разработки через тестирование

Методология представляет собой структурированную практику. Она позволяет получить чистый код и модифицировать его благодаря совмещению программирования, юнит-тестирования и рефакторинга. У методологии есть три фазы:

- **Красная.** Код не компилируется? Пишем юнит-тест.
- **Зелёная.** Реализация пишется в сжатые сроки. Появилось чистое и простое решение? Выполняйте его. В другом случае продукт будет улучшаться пошагово. Главная цель – получить зелёный цвет для юнит-тестов.
- **Рефакторинг.** Не пренебрегайте данным этапом – он устраняет повторения и вводит возможность изменять архитектуру. Фаза не затрагивает поведение программы.

Три ступени реализуются пятью этапами.



Цикл занимает до 10 минут и повторяется до покрытия функциональности юнит-тестами. Кажется, что всё просто. Однако шаги должны выполняться с предельной строгостью для использования преимуществ методологии. Соблюдайте правила, и получите структурированный код. Продукт будет соответствовать необходимым принципам (KISS --Keep it simple, stupid) без реализации ненужных функций (DRY – Don’t Repeat Yourself) благодаря непрерывному рефакторингу.

Чистые тесты

TDD – это не чудо, ведущее к оптимальному набору юнит-тестов без усилий. Помните, что в этой практике код продукта и тесты одинаково важны!

Чистый тест соблюдает 5 правил:

- **Скорость:** он работает быстро для частых запусков.
- **Независимость:** не зависят друг от друга.
- **Повторность:** воспроизводится в любой среде.
- **Самопроверка:** возвращает результат (Неудача или Успех) для быстрого и лёгкого заключения.
- **Своевременность:** пишется в подходящий момент.

Поменяйте мышление

Разработка через тестирование – отдельная парадигма. Во время обучения растут навыки программиста и преимущества подхода. Рассматривайте методику как вклад в будущее. Изменения затрагивают документацию приложения и юнит-тестов, представляющих исполняемые спецификации. Тесты используются для проверки исполнения требований и описывают их. Большую трудность для

программиста составляет создание дорожной карты для сложной функциональности в форме запланированных тестов.

Методология обнаруживает баги на ранних стадиях, что снижает затраты на поиск решения. 80% – это минимум покрытия кода серией юнит-тестов. Следовательно, разработчик уверенно приступает к рефакторингу и постоянному улучшению.

Выбирайте правильные инструменты

Eclipse с нативной поддержкой JUnit – явное преимущество. Плагины **MoreUnit** и **Infinitest** рекомендуется использовать в управлении юнит-тестами. Последние выполняют тесты при каждом изменении кода автоматически, что упрощает циклы обратной связи – часть непрерывного юнит-тестирования. В повторяющемся цикле методологии, использование шаблонов кода для юнит-тестов экономит время.

Разработка через тестирование в действии

Решим специфичную задачу. Возьмём проблему преобразования арабских чисел в римские.

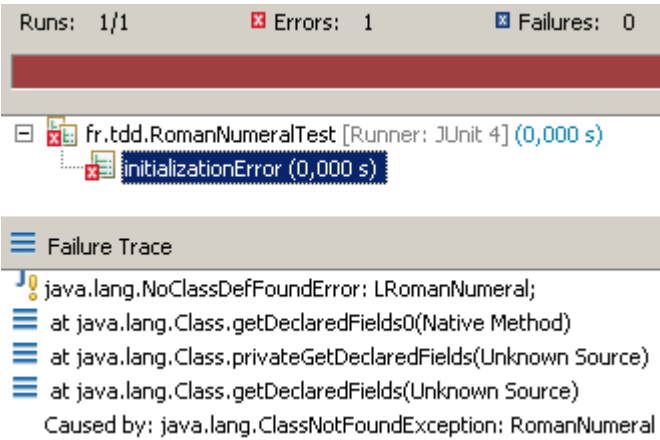
Сначала напишем класс `RomanNumeralTest`. Он содержит серии юнит-тестов программы. Первое требование – значение «1» выдаёт римскую «I»:

```
public class RomanNumeralTest {
    private static RomanNumeral romanNumeral;

    @BeforeClass
    public static void setUpBeforeClass() {
        romanNumeral = new RomanNumeral();
    }

    @Test
    public void testIntToRoman_1_is_I() {
        assertEquals(romanNumeral.toIntToRoman(1), is("I"));
    }
}
```

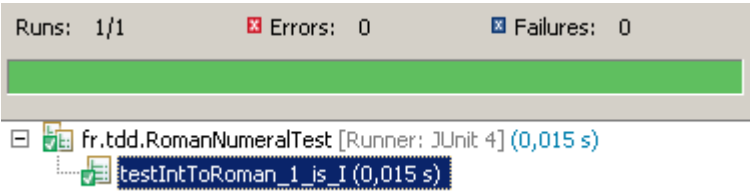
При запуске получим ошибку компиляции:



Продолжайте писать код продукта для прохождения теста. Для этого ставим курсор в месторасположение класса `RomanNumeral` и нажимаем **Ctrl+1** – сочетание горячих клавиш Eclipse. Оно предложит быстрое исправление для создания пустого класса `RomanNumeral`. Аналогичным способом пишем `toIntToRoman` – простейший метод, достаточный для возвращения значения «I»:

```
public class RomanNumeral {
    public String toIntToRoman(int arabic) {
        return "I";
    }
}
```

Тест выполнится. Двигайтесь по циклу.



Сейчас мы входим в фазу рефакторинга. Она пройдет быстро потому, что в коде нет повторений и нечего улучшать. Цикл начинается с добавления нового теста:

```
@Test
public void testIntToRoman_2_is_II() {
    assertThat(romanNumeral.toIntRoman(2), is("II"));
}
```

Для успешного прохождения измените метод `toIntRoman` класса `RomanNumeral`:

```
public String toIntRoman(int arabic) {
    if (arabic == 2) return "II";
    return "I";
}
```

Когда получаем зелёный цвет, двигаемся к рефакторингу. Предпочтительно иметь один выход для метода и использовать фигурные скобки для условия `if`:

```
public String toIntRoman(int arabic) {
    String roman = "I";

    if (arabic == 2){
        roman = "II";
    }

    return roman;
}
```

Тесты по-прежнему имеют зелёный цвет успеха. Мы расширяем их дополнительным требованием – числом 3, которое даёт римскую «III». Это делает неудачными текущие юнит-тесты. Для проверки пишем следующее:

```
public String toIntRoman(int arabic) {
    String roman = "I";

    if (arabic == 2){
        roman = "II";
    } else if(arabic == 3) {
        roman = "III";
    }

    return roman;
}
```

На шаге рефакторинга код улучшается с помощью цикла. Он уменьшает значения арабских чисел и добавляет полосу римских:

```
public String toIntRoman(int arabic) {
    StringBuilder roman = new StringBuilder();

    while (arabic-- > 0) {
        roman.append("I");
    }

    return roman.toString();
}
```

Мы обнаруживаем, что алгоритм не поддерживает римскую X. Для этого добавим обработку арабской десятки:

```
@Test
public void testInToRoman_10_is_X() {
    assertThat(romanNumeral.toIntToRoman(10), is("X"));
}
```

Тест выполняется, а рефакторинг не нужен. Значение 10 и его римское представление XX потребуют ещё один тест. Он «сломает» все предыдущие. Пишем следующий код:

```
public String intToRoman(int arabic) {
    StringBuilder roman = new StringBuilder();

    if (arabic == 10) {
        roman.append("X");
    } else {
        while (arabic-- > 0) {
            roman.append("I");
        }
    }

    return roman.toString();
}
```

Он пройдёт серию тестов. Фаза рефакторинга позволяет нам вернуться на предыдущий шаг для оптимизации алгоритма преобразования римской X. Заметно, что использование цикла будет эффективнее условий `if / else if`. Код принимает следующий вид:

```
public String intToRoman(int arabic) {
    StringBuilder roman = new StringBuilder();

    if (arabic == 10) {
        roman.append("X");
    } else if (arabic == 20) {
        roman.append("XX");
    } else {
        while (arabic-- > 0) {
            roman.append("I");
        }
    }

    return roman.toString();
}
```

Код успешно пройдёт серию тестов. Фаза рефакторинга позволяет нам вернуться на предыдущий шаг для оптимизации алгоритма преобразования римской X. Заметно, что использование цикла будет эффективнее условий `if / else if`. В итоге код принимает следующий вид:

```
public String intToRoman(int arabic) {
    StringBuilder roman = new StringBuilder();

    while (arabic >= 10) {
        roman.append("X");
        arabic -= 10;
    }

    while (arabic-- > 0) {
        roman.append("I");
    }

    return roman.toString();
}
```

}

JUnit горит зелёным, а рефакторинг не изменил внутренне поведение метода. Рассмотрев код продукта, мы заметим, что задачи для «I» и «X» выполняются одним способом. У нас появилась идея нового дизайна алгоритма: две таблицы, связанные индексами и содержащие римские и арабские цифры соответственно.

```
public static final int[] ARABIC_DIGITS = {10, 1};
public static final String[] ROMAN_DIGITS = {"X", "I"};

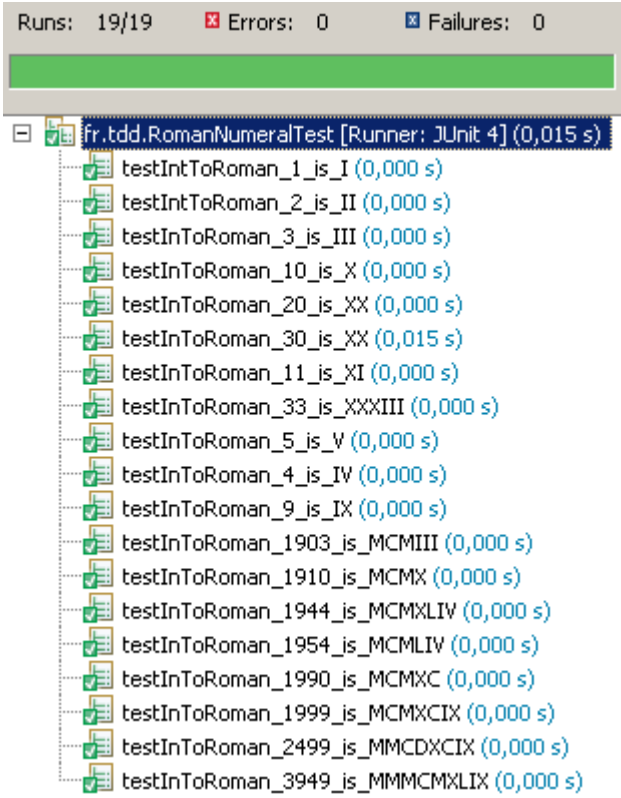
public String intToRoman(int arabic) {
    StringBuilder roman = new StringBuilder();

    for (int i = 0; i < ARABIC_DIGITS.length; i++) {
        while (arabic >= ARABIC_DIGITS[i]) {
            roman.append(ROMAN_DIGITS[i]);
            arabic -= ARABIC_DIGITS[i];
        }
    }

    return roman.toString();
}
```

С помощью 30 юнит-тест не сломать, поэтому необязательно изменять код продукта. Для чисел 11 и 33, которые образуются римскими цифрами X и I алгоритм также остаётся функциональным. К неудаче в соответствующем юнит-тесте приведёт «V». Пора начинать цикл разработки через тестирование заново! Первое решение – добавить «V» и её арабский эквивалент в таблицы, и проверить алгоритм. Тесты проходят? Тогда это правильное решение. Во время рефакторинга возникает вопрос: не лучше ли заменить две индексированные таблицы на **Java Map**? Значения упорядочиваются путём двух взаимосвязанных циклов, текущее решение предпочтительнее потому, что оно проще соответствует принципам KISS.

Разработка через тестирование продолжается и мы завершаем серию юнит-тестов.



После 10 шага получаем следующие таблицы:

```
ARABIC_DIGITS = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
ROMAN_DIGITS = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
```

Добавление новых тестов с такими арабскими цифрами, как 1954 и 3949 не потребует никаких изменений метода `intToRoman` в коде продукта. Серия полученных юнит-тестов покрывает код максимально.

Coverage Session View							
Name	Lines	Total	%	Branches	Total	%	
All Packages (2011-12-07 16:04:51)	54	54	100,00 %	4	4	100,00 %	
fr.tdd	54	54	100,00 %	4	4	100,00 %	
RomanNumeral	11	11	100,00 %	4	4	100,00 %	
RomanNumeralTest	43	43	100,00 %	0	0	-	

Заключение

Знакомство с разработкой через тестирование показало силу этой практики. Смена парадигмы начинается с обучения и завершается ростом производительности разработчика.

Ещё одно упражнение: добавьте метод обратного преобразования в рассмотренную проблему с римскими и арабскими числами. Используйте разработку через тестирование. Практика – средство прогресса!

Что вы думаете по поводу такого тестирования?

♥ 3 💬 Обсудить 📌 3 🔥 0 💧 0 😡 0

QA



МЕРОПРИЯТИЯ

VK justtech

📅 06 марта Москва Онлайн Бесплатно

+ Показать еще

Комментарии




Оставьте свой комментарий (можно использовать markdown)

+

Отправить

ВАКАНСИИ

Добавить вакансию

-  **Middle / Senior C++ Разработчик**
Москва, до 350000 RUB
-  **Golang backend developer**
Москва, от 350000 RUB до 600000 RUB
-  **Программист PHP**
от 180000 RUB до 350000 RUB

+ Показать еще

Опубликовать вакансию

ЛУЧШИЕ СТАТЬИ ПО ТЕМЕ

👨💻💰10 онлайн-платформ для заработка на тестировании

Можно начать без заказчиков и даже сделать профессию тестировщика подработкой. Онлайн-платформы для тестирования помогут заработать на фрилансе.

Погружаемся в основы и нюансы тестирования Python-кода

Пишете код на Python? Будет полезно знать о принципах тестирования Python-кода ваших приложений. Изучайте статью и применяйте навыки в работе.

6 книг по тестированию ПО

Каждый продукт требует проверки, и ПО не исключение. Представляем подборку книг про тестирование, которая поможет вам в этом нелегком деле.

[О проекте](#)

[Реклама](#)

[Пользовательское соглашение](#)

[Публичная оферта](#)

[Политика конфиденциальности](#)

[Контакты](#)

☐ Push-уведомления

☐ Темная тема



© 2024, Proglib. При копировании материала ссылка на источник обязательна.