

5. Введение в тестирование

- Обзор
- Проблема
- Юнит тестирование
- TDD
- BDD
- Интеграционное тестирование
- Уровень покрытия тестами
- Итог

Обзор

Привет 🙋👨! В этой части конспекта мы рассмотрим проблемы, являющиеся причинами необходимости тестировать код, а также самые популярные методики тестирования 🐛.

К делу 🦊!

Проблема

Не зависимо от того пишем мы личный 🧑🏠 проэкт или работаем в команде 🏢, возникает вопрос — «как быть уверенным, что написанный мною код будет работать хорошо?».

В случае участия в команде разработчиков на проэкте интернет-магазина, представь, что ты работаешь над модулем «Корзина». Пока бэкенд разработчики готовят API, ты строишь модель UI для подключения ее к API когда оно станет доступно, временно работая с заглушкой 📺.

Вскоре бэкенд разработчик информирует о готовности API, и ты совершаешь миграцию на готовую инфраструктуру, делаешь пулл последних изменений, и... все ломается, и может быть тысяча причин, почему происходит именно так 🤔.

Такие ситуации часто возникают по причине того, что один из разработчиков мог случайно внести изменения в код, не явно (или явно), касающийся модуля «Корзина». Как правило, так происходит по случайности: все разработчики — обычные люди, и людям свойственно допускать ошибки 🙋.

В случае с персональным проэктom ситуация обстоит немного иначе, но в случае, когда ты долго не заглядывал в код, и спустя некоторое время решил совершить рефакторинг, на пример — высока вероятность случайно сломать что-то в программе. Достаточно рискованно рефакторить код, с которым давно не работал, и на такой случай нужна

страховка 🧑.

🔍 Заметка:

Рефакторинг — это дисциплинированный путь реструктуризации написанного кода с целью улучшения его качества 💎.

Подобные ошибки иногда могут приводить к часам их отслеживаний и фиксов, однако существует надежный и главное, **автоматизированный** подход сохранения предсказуемости изменений, внесенных в проэкт — **тестирование кода** 📝.

Тестирование кода — это процесс выполнения программы с целью выявления **багов**.

Баг 🐛 — это дефект, ошибка компьютерной программы или системы, приводящая к некорректному результату ее выполнения. **Баг** также можно описать как несходство актуального поведения программы с ожидаемым 🧑.

Цель тестирования кода — отлов **багов**, созданных программистом.

Существует несколько фундаментальных методик **тестирования кода** — о них мы и поговорим 🧑.

Юнит тестирование

Юнит (англ. «unit», единица) — это изолированный функциональный блок кода. Как правило — функция 🧑.

Чаще всего, **юнит тест** на практике — это **простейшая функция, проверяющая правильность работы другой простейшей функции** 📖.

Хороший **юнит тест** должен соответствовать следующим критериям:

- Простота;
- Быстрота написания;
- Быстрота работы.

Это означает, что **юнит тестов** как правило, бывает много.

Преимущество, несущее в себе написание **юнит тестов** в том, что при добавлении новых фич в программу, или ее рефактинге разработчик сохраняет уверенность в том, что остальные части программы, покрытые тестами — будут работать хорошо. А если возникнет ошибка — провалившийся тест сообщит об этом разработчику 🛡️.

Юнит тест должен быть **изолированным**, то есть, не зависеть ни от каких внешних факторов — от сети интернет, баз данных или внешних программных модулей.



Автор:

— Эй Оскар! Что ты думаешь о юнит тестах?



Оскар:

— Привет! Отношусь отлично, юнит тесты помогают мне быть уверенным в моем коде! Совсем недавнее я создал фэнтезийную RPG, и покрыл ее юнит тестами для обеспечения стабильности данной программы 🧑🏻‍💻.



Пример кода 14.1:

```
export default class Character {
  constructor (level = 0) {
    this.level = level;
  }

  levelUp () {
    this.level += 1;
  }

  getLevel () {
    return this.level;
  }
}

const character = new Character(0);

character.levelUp();

if (character.getLevel() !== 1) {
  throw new Error('Character level should be 1.');
```

В примере кода 14.1 на строке кода 15 создается экземпляр класса `Character`, с изначальным значением свойства `level` — `0`. Метод `levelUp` значение свойства `level` увеличивает на единицу. Метод `getLevel` возвращает текущее значение свойства `level`.

На `линии кода 19` описан простой `юнит тест`. В данном случае, если метод `getLevel` вернет `не ожидаемое` значение — `юнит тест` можно расценивать как `провалившийся`. Если метод `getLevel` вернет `ожидаемое` значение — `юнит тест` можно расценивать как `успешный` 🏆.

Рассмотрев данный пример, можно вывести, что `юнит тестинг` — это процесс разработки программного обеспечения, когда малейшие функциональные элементы программы, или `юниты`, подвергаются независимой проверке в условиях изоляции.

Ранее мы говорили о том, что в функциональные компоненты React просты в тестировании. Теперь у нас есть более глубокое понимание причины этого. Ведь функциональный компонент — это `чистая функция`, каждый раз возвращающая один и тот-же результат, будучи вызванной с одними и теми-же аргументами.

`Тестируемость` — это одна из `хороших` сторон функциональных компонентов.

Пример кода 14.2 🖥️:

```
const Character = ({ name }) => {  
  
  return <h1>Greetings, I am { name }!</h1>;  
}
```

`Юнит тест` данного функционального компонента — простая проверка возвращаемой им разметки. Мы рассмотрим механизмы тестирования React компонентов в следующих частях конспекта 🔑.

TDD

`TDD` (Test-Driven Development) — это методика процесса разработки программного обеспечения, основанная на первичном написании тестов, и последующем написании кода, основанного на уже написанных тестах, с целью `удовлетворить требования написанного теста`, и отрефакторить написанный код после достижения этой цели 🧙.

Непосредственная составляющая методики `TDD` — `юнит тестирование`, рассмотренное нами в предыдущей секции этого урока.

На пример, для создания простой функции по методологии `TDD`, необходимо:

- Продумать логику выполнения тела функции и принимаемые ею аргументы при вызове;
- Продумать возможные «подводные камни» и «уязвимости» функции;
- Написать тест для будущей функции учитывая описанные пункты выше;
- Запустить тест — на этот момент времени тест сломается, учитывая отсутствие имплементации будущей функции;
- Написать имплементацию функции в минимальном виде, покрывающем требования

теста;

- Запустить тест, и убедиться в его успешности;
- Опционально отрефакторить написанный код.

👉 Совет:

Процесс **TDD** не подразумевает написания **всех** тестов перед написанием **всего** кода. **TDD** больше похож на инкрементальные и итеративные шаги во время процесса разработки, сравнимого с восхождением альпиниста на гору.

Важно также соблюдать условие **изолированности** кода в условиях теста, что означает отсутствие привязок к внешним зависимостям. Примером зависимости может быть Интернет 🌐.

Например, чтобы протестировать функцию, содержащую «сайд-эффект» в виде запроса к удаленному API, и соблюдать условие **изолированности** — нам могут пригодиться **моки** (англ. **«mock»**) 📄.


Мок — это инструмент, имплементирующий «подмену» **не предсказуемого** «сайд-эффекта» (такого, как запрос к удаленному серверу), **предсказуемым** результатом.

Например, тест функции с настоящим запросом к удаленному API может произвести обманчивый результат, если запустить тест в условиях отсутствия Интернет. Ведь в таком случае причиной провала теста не является баг в тестируемой функции — а зависимость ее от «сайд-эффекта», дающего обманчивый результат.

Поэтому использование **моков** в написании тестов является естественным выбором по умолчанию 🛠️.

BDD

BDD (Behavior-Driven Development) — это подход процесса разработки программного обеспечения, основанного на создании приложения из описания его поведения по отношению к конечному пользователю 👤.

Принцип тестирования в **BDD** можно описать на примере **поведенческого теста** банкомата .

- Изначально банкомат располагает балансом в **100\$** ;
 - При этом, карточка пользователя — **валидна** ;
 - При этом, у автомата — **достаточно средств для выдачи** ;
- Когда пользователь **запрашивает 20\$** ;
- Банкомат должен **выдать 20\$** ;
 - В результате, баланс банкомата должен обрести значение **80\$** ;
 - А карточка должна быть возвращена пользователю.

Первое, что можно заметить из вышеописанного — тест написан на человеческом языке



Второе — что поведенческий тест составлен из трех секций:

Контекст	Событие	Результат
Банкомат располагает балансом в 100\$	Когда пользователь запрашивает 20\$	Банкомат должен выдать 20\$
И карточка пользователя — валидна		В результате, баланс автомата должен обрести значение 80\$
И автомат располагает достаточным количеством средств		А карточка должна быть возвращена пользователю

- Контекст — это изначальное состояние, все условия, описанные в контексте должны быть соблюдены ;
- Событие — это действие (как правило пользователя) для получения результата ;
- Результат — это ожидаемый результат события .

Хорошая сторона поведенческого тестирования — это условие наличия прямолинейного описания поведения программы.

В отличие от подхода TDD, в BDD тесты могут быть написаны в любое время — перед, во время или после разработки кода 🕒.

Еще одно отличие BDD от TDD подхода в том, что целью теста не должна быть проверка имплементации. Целью должны быть проверка поведения.

Например, в примере кода 14.1, тест класса Character полностью зависит от факта, что уровень персонажа всегда начинается с 0:

- Экземпляр класса Character создается со значением свойства level в виде 0 ;
- Метод levelUp всегда увеличивает значение свойства level на 1 .

Факт того, что свойство level обретает значение 0 при создании экземпляра класса Character — деталь имплементации. Единственная причина по которой тест был написан именно так — необходимость протестировать имплементацию, не поведение.

BDD предусматривает тестирование поведения — поэтому, вместо того, чтобы думать об имплементации, стоит задуматься о поведении.

Учитывая данный вектор мы можем протестировать класс Character следуя практике BDD.

 Пример кода 14.3:

```
export default class Character {
  constructor (level = 0) {
    this.level = level;
  }

  levelUp () {
    this.level += 1;
  }

  getLevel () {
    return this.level;
  }
}

const character = new Character(24);

const expectedCharacterLevel = character.getLevel() + 1;

character.levelUp();

if (character.getLevel() !== expectedCharacterLevel) {
  throw new Error('When level-up, a Character\'s level should be increased by \'1\'');
} else {
  console.log('Tested successfully');
}
```

В примере кода 14.3 мы больше тестируем деталь имплементации. Проверка строке кода 21 направлена на тест поведения с учетом того, что мы больше не зависим на хардкод в виде стартового уровня персонажа в виде 0.

Эту тонкость можно выразить еще в в одном примере. Если вдруг требования к классу Chracter поменяются, и персонаж будет начать на с 0 уровнем а скажем, с 10 — поведенческий тест останется валидным, так как таковым является поведение, чего вовсе не скажешь о варианте теста, написанного по подходу TDD.

Интеграционное тестирование

Интеграционное тестирование — это фаза тестирования в процессе разработке программного обеспечения, в которой программные модули тестируются в единой сессии группой.

! Важно:

Интеграционное тестирование подразумевает наличие готовых юнит тестов.

Задача интеграционного тестирования — убедиться, что отдельные части приложения работают правильно вместе.



Автор:

— Эй, Оскар! Давай напишем интеграционный тест на твою RPG!



Оскар:

— Давай! В моей игре у Character есть некоторая стартовая точка, с которой он начинается — это Tavern. Эту деталь мы и используем в тесте. К делу 🦊!

Модуль Tavern.

```
export default class Tavern {
  constructor(...characters) {
    this.characters = characters;
  }

  getCharacters () {
    return this.characters;
  }

  getCharacterByName (name) {
    return this.getCharacters().filter(character => character.getName()
=== name)[0];
  }
}
```

Tavern (таверна) содержит массив потенциальных characters (персонажей), которые могут жить в ней, и обладает не сложным API, позволяющим получить список всех персонажей или звать их по-одному 🏠.

Слегка модифицированная имплементация Character:

```
export default class Character {
```



```

    constructor (name = 'Hero', level = 0) {
        this.name = name;
        this.level = level;
    }

    levelUp () {
        this.level += 1;
    }

    getLevel () {
        return this.level;
    }

    getName () {
        return this.name;
    }

    static create (name, level, startingPoint) {
        const character = new Character(name, level);

        if (startingPoint) {
            startingPoint.push(character);
        }


        return character;
    }
}

```

Обновленная имплементация `Character` представляет собой:

- Новое свойство `name`;
- Новый метод `getName` для получения свойства `name`;
- Новь статический метод `create`.

Теперь мы можем написать `интеграционный тест` с участием этих двух сущностей:

 Пример кода 14.4:

```

import Tavern from './Tavern';
import Character from './Character';

const robert = Character.create('Robert the Nimble', 0);

const tavern = new Tavern(robert);

```

```

const jack = Character.create('Jack the Mighty', 0, tavern.characters);
const jane = Character.create('Jane the Wise', 0, tavern.characters);

jack.levelUp();

// Tests

if (tavern.getCharacters().length !== 3) {
  throw new Error('Tavern should contain three characters.');
```

```

} else {
  console.log('Tavern capacity is calculated correctly.');
```

```

}

if (tavern.getCharacterByName('Jack the Mighty').level !== 1) {
  throw new Error('Jack the Mighty\'s level should be \'1\'.');
```

```

} else {
  console.log('The specific Character is returned by a Tavern correctly.');
```

```

}
```

Данный `интеграционный тест` имплементируем две проверки:

- Что `Tavern` привально обрабатывает подсчет персонажей, что в ней живут;
- Что `Tavern` возвращает запрошенного персонажа правильно, и что это правильный персонаж.

Стистика данного `интеграционного теста` не учитывает деталей имплементации `Tavern` или `Character`. Вместо этого тест проверяет `правильное поведение взаимодействие двух сущностей`. Если обе сущности ведут себя правильно в определенные моменты времени при определенных условиях — можно считать, что `интеграционный тест` прошел успешно 🍀.

Уровень покрытия тестами

В тестировании программного обеспечения существует некоторое абстрактное понятия `уровня покрытия программы тестами` (англ. «test coverage»).

`Уровень покрытия тестами` — это процентное соотношение написанного разработчиком кода и тестов, написанных для этого кода. Распространенное заблуждение многих разработчиков заключается в том, что чем выше `уровень покрытия тестами` — тем лучше, однако это не совсем так. Дело в том, что относительная природа механизма подсчета `уровня покрытия тестами` не говорит о `релевантности` написанных тестов ровно ничего 🤔.

Иными словами, если разработчик располагает 10 функциями, иногда достаточно написать 10 тестов (по 1 тесту на функцию), чтобы достичь необходимого уровня безопасности. 20 тестов для 10 функций могут быть не лишними, но полезность дополнительных 10 тестов резко уменьшается. При этом, порой достаточно располагать даже 5 тестами для 10 функций, что в результате даст `уровень покрытия тестами` в размере `50%`, могут дать достаточный уровень безопасности и уверенности в написанном коде.

Из вышеописанного можно вывести теорию о том, что полезность написанных тестов уменьшается прямо пропорционально их количеству. Нужно очень осторожно распределять время на написание тестов. Ведь написание тестов не подразумевает их написание ради их написания — тесты должны нести в себе `смысл`, и быть `полезными` и `релевантными`. Только в таком случае тестирование программного кода можно расценивать как полезную практику 🍌.

Итог

В этом уроке мы рассмотрели основные подходы тестирования программного кода.

`TDD` — подход разработки и тестирования программного обеспечения, в котором программист сперва пишет тесты для атомарных составляющих программы, а потом — сами составляющие, таким образом, чтобы удовлетворить уже написанные тесты.

`BDD` — подход разработки и тестирования программного обеспечения, в котором акцент смещается на соответствие требованиям программы в лице ее `поведения` по отношению к конечному пользователю.

`Интеграционное тестирование` — подход тестирования программного кода с участием нескольких модулей программы.

`Уровень покрытия программы тестами` — значение, выведенное из соотношения количества написанного кода и тестов для этого кода.

Мы будем очень признательными, если ты оставишь свой фидбек в отношении этой части конспекта: `hello@lectrum.io`.