

14. Введение в тестирование

Содержание урока

- Обзор;
- Проблема;
- Юнит-тестирование;
- TDD;
- BDD;
- Интеграционное тестирование;
- Уровень покрытия тестами;
- Подведём итоги.

Обзор

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

К делу! 🦊

Проблема

Независимо от того, пишем ли мы личный 🧑‍💻 проект или работаем в команде, 💣 возникает вопрос: «как быть уверенным, что написанный мною код будет работать хорошо?»

В случае участия в команде разработчиков на проекте интернет-магазина, представь, что ты работаешь над модулем «Корзина». Пока бэкенд-разработчики готовят API, ты строишь модель UI для подключения её к API, когда оно станет доступным, 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, неявно (или явно) касающийся модуля «Корзина». Как правило, так происходит по случайности: все разработчики — обычные люди, и людям свойственно допускать ошибки. 🙋‍♂️

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

🔍 Хозяйке на заметку:

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 проект — **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**, состоит в том, что при добавлении новых фич в программу или её 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: сети интернет, баз данных или внешних программных модулей.



Автор :

— Эй, Оскар! Как ты относишься к `юнит-тестам`?



Оскар :

— Привет! Отношусь отлично, `юнит-тесты` помогают мне быть уверенным в моём коде! Совсем недавно я создал фэнтезийную `RPG` и покрыл ее `юнит-тестами` для обеспечения стабильности данной программы. 📌



Пример кода 14.1:

```
1  export default class Character {
2      constructor (level = 0) {
3          this.level = level;
4      }
5
6      levelUp () {
7          this.level += 1;
8      }
9
10     getLevel () {
11         return this.level;
12     }
13 }
14
15 const character = new Character(0);
16
17 character.levelUp();
18
19 if (character.getLevel() !== 1) {
20     throw new Error('Character level should be 1.');
```

```
21 } else {
22     console.log('Test successful: Character level is 1.');
```

```
23 }
```

В примере кода 14.1 на строке кода 15 создаётся 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 строке кода 19 описан простой `юнит-тест`. В данном случае, если метод `getLevel` вернет `неожиданное` значение — `юнит-тест` можно расценивать как `провалившийся`. Если метод `getLevel` вернет `ожидаемое` значение — `юнит-тест` можно расценивать как `успешный`. 🛡️

Рассмотрев данный пример, можно заключить, что **юнит-тестинг** — это процесс разработки программного обеспечения, когда малейшие функциональные элементы программы (**юниты**) подвергаются независимой проверке в условиях изоляции.

Ранее мы говорили о том, что 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:

```
1  const Character = ({ name }) => {  
2  
3      return <h1>Greetings, I am { name }!</h1>;  
4  }
```

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.


Например, 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 вышеописанные пункты;
- 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 бывалых:


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 рамках 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» **непредсказуемого** «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 является не 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** ;
 - 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 — **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 банкомата должен обрести значение 80\$
И банкомат 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 должна быть проверка **поведения**.

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

- Экземпляр класса **Character** создаётся 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:

```

1 export default class Character {
2   constructor (level = 0) {

```

```

3         this.level = level;
4     }
5
6     levelUp () {
7         this.level += 1;
8     }
9
10    getLevel () {
11        return this.level;
12    }
13 }
14
15 const character = new Character(24);
16
17 const expectedCharacterLevel = character.getLevel() + 1;
18
19 character.levelUp();
20
21 if (character.getLevel() !== expectedCharacterLevel) {
22     throw new Error('When level-up, a Character\'s level should be
increased by \'1\'.');
23 } else {
24     console.log('Tested successfully.');
```

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

Эту особенность можно выразить ещё в одном примере. Если вдруг требования к классу Character поменяются, и персонаж будет начинать не 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 разработки 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 вместе работают правильно.



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` :

```
1 export default class Tavern {
2   constructor(...characters) {
3     this.characters = characters;
4   }
5
6   getCharacters () {
7     return this.characters;
8   }
9
10  getCharacterByName (name) {
11    return this.getCharacters().filter(character =>
12      character.getName() === name)[0];
13  }
```

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

Слегка модифицированная имплементация `Character` :

```
1 export default class Character {
2   constructor (name = 'Hero', level = 0) {
3     this.name = name;
4     this.level = level;
5   }
6
7   levelUp () {
8     this.level += 1;
9   }
10
11  getLevel () {
12    return this.level;
13  }
14
15  getName () {
16    return this.name;
```



```

17     }
18
19     static create (name, level, startingPoint) {
20         const character = new Character(name, level);
21
22         if (startingPoint) {
23             startingPoint.push(character);
24         }
25
26         return character;
27     }
28 }

```

Обновлённая имплементация `Character` представляет собой:

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

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

 Пример кода 14.4:

```

1  import Tavern from './Tavern';
2  import Character from './Character';
3
4  const robert = Character.create('Robert the Nimble', 0);
5
6  const tavern = new Tavern(robert);
7
8  const jack = Character.create('Jack the Mighty', 0, tavern.characters);
9  const jane = Character.create('Jane the Wise', 0, tavern.characters);
10
11  jack.levelUp();
12
13  // Tests
14
15  if (tavern.getCharacters().length !== 3) {
16      throw new Error('Tavern should contain three characters.');
```

Данный `интеграционный тест` имплементирует две проверки:

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

Стилистика данного `интеграционного теста` не учитывает деталей имплементации `Tavern` или `Character`. Вместо этого тест проверяет `правильное поведение взаимодействия двух сущностей`. Если обе сущности ведут себя правильно в определённые моменты времени при определённых условиях, можно считать, что `интеграционный тест` прошёл успешно. 🧑🏻‍🔬

Уровень покрытия тестами

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

`Уровень покрытия тестами` — это процентное соотношение написанного разработчиком кода и тестов, написанных для этого кода. Распространённое 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%`, которые могут обеспечить достаточный уровень безопасности и уверенности в написанном коде.

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

Подведём итоги

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 виде её **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.

Спасибо, что остаёшься с нами! 🙌 В следующей части конспекта мы рассмотрим удобный инструмент, с помощью которого можно писать тесты для программного кода с намного более высоким уровнем эффективности. До встречи! 🐙

Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту hello@lectrum.io.