Университет ИТМО
Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

по дисциплине «Тестирование программного обеспечения»

Выполнил:

Студент группы P3331

Дворкин Борис Александрович

Вариант: 32091

Преподаватель:
Гаврилов Антон Валерьевич

г. Санкт-Петербург

2024 г.

Содержание

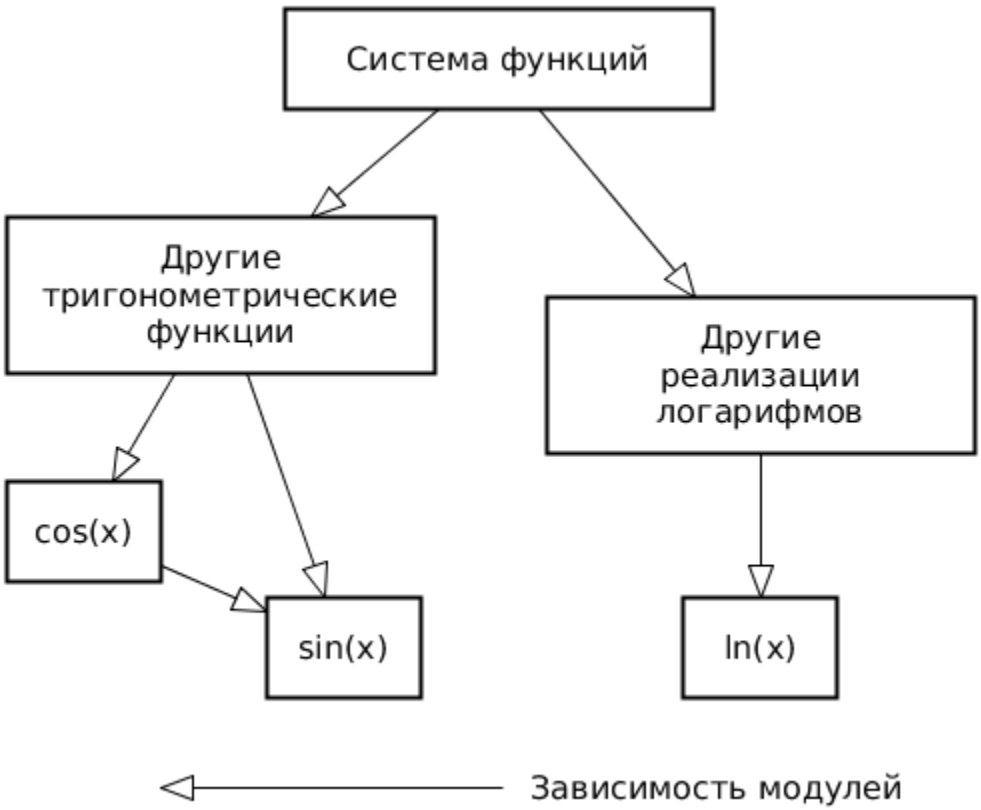
Описание задания

Провести интеграционное тестирование программы, осуществляющей вычисление системы функций:

$$\begin{cases} ((scs(x) - sec(x)) - cos(x)) & \text{if } x \leq 0 \\ \left(\frac{(\log_5(x) - \ln(x))^4 \cdot (\log_5(x) + \log_3(x))^3}{\log_{10}(x) \cdot ln(x)} \right) & \text{if } x > 0 \end{cases}$$

1. Все составляющие систему функции (как тригонометрические, так и логарифмические) должны быть выражены через базовые (тригонометрическая зависит от варианта; логарифмическая - натуральный логарифм).

2. Структура приложения, тестируемого в рамках лабораторной работы, должна выглядеть следующим образом (пример приведён для базовой тригонометрической функции sin(x)):



3. Обе "базовые" функции (в примере выше - sin(x) и ln(x)) должны быть реализованы при помощи разложения в ряд с задаваемой погрешностью. Использовать тригонометрические / логарифмические преобразования для упрощения функций ЗАПРЕЩЕНО.

4. Для КАЖДОГО модуля должны быть реализованы табличные заглушки. При этом, необходимо найти область допустимых значений функций, и, при необходимости, определить взаимозависимые точки в модулях.

5. Разработанное приложение должно позволять выводить значения, выдаваемое любым модулем системы, в csv файл вида «X, Результаты модуля (X)», позволяющее произвольно менять шаг наращивания X. Разделитель в файле csv можно использовать произвольный.

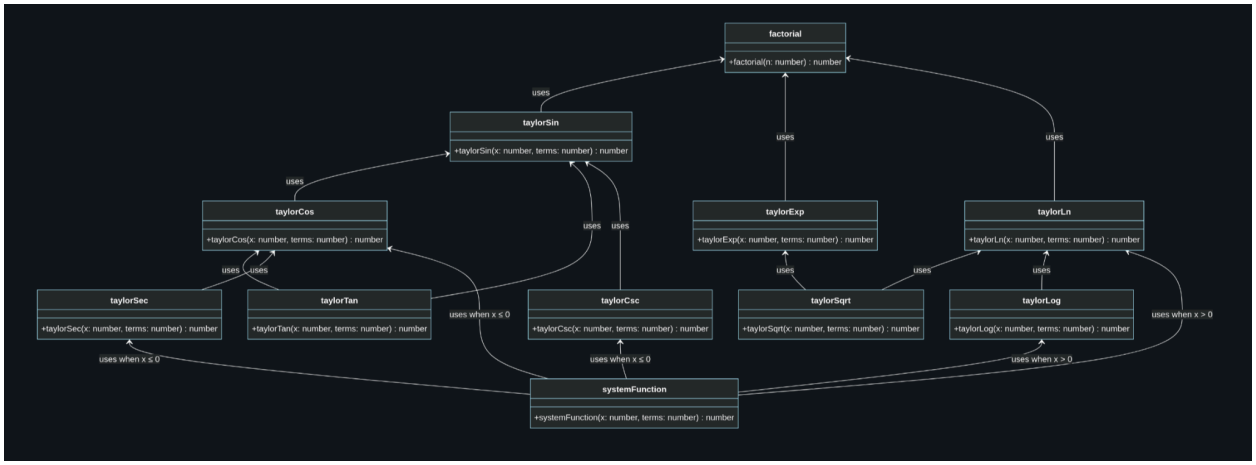
Порядок выполнения работы:

1. Разработать приложение, руководствуясь приведёнными выше правилами.
2. С помощью JUNIT4 разработать тестовое покрытие системы функций, проведя анализ эквивалентности и учитывая особенности системы функций. Для анализа особенностей системы функций и составляющих ее частей можно использовать сайт https://www.wolframalpha.com/.
3. Собрать приложение, состоящее из заглушек. Провести интеграцию приложения по 1 модулю, с обоснованием стратегии интеграции, проведением интеграционных тестов и контролем тестового покрытия системы функций.

Отчёт по работе должен содержать:

1. Текст задания, систему функций.
2. UML-диаграмму классов разработанного приложения.
3. Описание тестового покрытия с обоснованием его выбора.
4. Графики, построенные csv-выгрузкам, полученным в процессе интеграции приложения.
5. Выводы по работе.

UML диаграмма классов



Смотрите сурсы, там mermaid, поэтому удобнее, можно зумить и всё такое:
https://github.com/lmtjl/qa-fundamentals/tree/main/integration-testing#class-diagram

Описание тестового покрытия

Подход к выбору тестового покрытия

При разработке тестового покрытия для системы математических функций я руководствовался принципом эффективного выявления ошибок интеграции с минимальными затратами на тестирование. Поскольку целью лабораторной работы было именно интеграционное тестирование, я сосредоточился на проверке взаимодействия между компонентами, а не на детальном тестировании логики каждой отдельной функции.

Я выбрал комбинированный подход к интеграционному тестированию, используя элементы как восходящего (bottom-up), так и нисходящего (top-down) подходов. Такая стратегия позволила эффективно проверить взаимодействие на разных уровнях абстракции системы — базовых математических операций.

Структура тестового покрытия

1. Базовые функции и их заглушки

Первым уровнем тестирования стала проверка соответствия между реальными функциями и их заглушками. Для каждой базовой функции (factorial, sin, cos и т.д.) были реализованы табличные заглушки, возвращающие предварительно вычисленные значения для ключевых точек. Это обеспечивает стабильное и быстрое выполнение тестов, исключая влияние погрешностей вычислений на результаты тестирования.

```
test('factorial should match the stub', () => {
      for (let n = 0; n <= 5; n++) {
      const actualValue = math.factorial(n);
      const stubValue = stubs.factorialStub(n);

      expect(stubValue).toBeCloseTo(actualValue, 0);
      }
});

test('sin and cos should match the stubs', () => {
      const testPoints = [0, Math.PI/6, Math.PI/4, Math.PI/3, Math.PI/2,
Math.PI];

      for (const x of testPoints) {
      const actualSin = math.taylorSin(x, 15);
      const stubSin = stubs.sinStub(x);
```

```
      const actualCos = math.taylorCos(x, 15);
      const stubCos = stubs.cosStub(x);

      expect(stubSin).toBeCloseTo(actualSin, 1);
      expect(stubCos).toBeCloseTo(actualCos, 1);
      }
});
```

Важно отметить, что я не стремился к 100% точному совпадению реальных функций и заглушек, так как это противоречит самой идее заглушек как упрощенных заменителей. Достаточная точность (до 1 знака после запятой) обеспечивает корректное поведение для тестирования интеграции.

2. Интеграция тригонометрических функций

Следующим уровнем стало тестирование интеграции компонентов тригонометрической части системной функции. Здесь я проверял, что функция `trigFunctionStub` корректно вызывает свои компоненты (`cscStub`, `secStub`, `cosStub`) и правильно обрабатывает особые случаи.

```
test('trigFunction should integrate with stubs', () => {
      const cscSpy = jest.spyOn(stubs, 'cscStub');
      const secSpy = jest.spyOn(stubs, 'secStub');
      const cosSpy = jest.spyOn(stubs, 'cosStub');

      const x = -0.5;
      const result = stubs.trigFunctionStub(x);

      expect(cscSpy).toHaveBeenCalledWith(x);
      expect(secSpy).toHaveBeenCalledWith(x);
      expect(cosSpy).toHaveBeenCalledWith(x);

      expect(isNaN(result)).toBe(false);
});

test('trigFunction should handle special cases', () => {
      const result = stubs.trigFunctionStub(0);
      expect(isNaN(result)).toBe(true);
});
```

Данные тесты выявляют проблемы интеграции между компонентами, в частности, проверяют факт вызова зависимых функций с правильными параметрами и корректную обработку особых случаев (например, при x = 0, где sin(0) = 0, что делает csc(0) неопределенным).

3. Интеграция логарифмических функций

Аналогичный подход был применен к тестированию логарифмической части системной функции. Тесты проверяют, что функция `logFunctionStub` корректно вызывает компоненты (`log5Stub`, `log3Stub`, `log10Stub`, `lnStub`) и обрабатывает граничные случаи.

```
test('logFunction should integrate with stubs', () => {
      const log5Spy = jest.spyOn(stubs, 'log5Stub');
      const log3Spy = jest.spyOn(stubs, 'log3Stub');
      const log10Spy = jest.spyOn(stubs, 'log10Stub');
      const lnSpy = jest.spyOn(stubs, 'lnStub');

      const x = 2;
      const result = stubs.logFunctionStub(x);

      expect(log5Spy).toHaveBeenCalledWith(x);
      expect(log3Spy).toHaveBeenCalledWith(x);
      expect(log10Spy).toHaveBeenCalledWith(x);
      expect(lnSpy).toHaveBeenCalledWith(x);

      expect(isNaN(result)).toBe(false);
});

test('logFunction should handle special cases', () => {
      const result = stubs.logFunctionStub(0);
      expect(isNaN(result)).toBe(true);
});
```

Особое внимание уделено обработке граничных случаев, таких как x = 0, где логарифмические функции не определены.

4. Интеграция системной функции

Верхний уровень интеграционного тестирования проверяет, что системная функция корректно выбирает между тригонометрической и логарифмической частями в зависимости от входного значения x.

```
test('systemFunction should use trigFunction for x ≤ 0', () => {
      const trigSpy = jest.spyOn(stubs, 'trigFunctionStub');

      const x = -0.5;
      stubs.systemFunctionStub(x);

      expect(trigSpy).toHaveBeenCalledWith(x);
});

test('systemFunction should use logFunction for x > 0', () => {
      const logSpy = jest.spyOn(stubs, 'logFunctionStub');

      const x = 2;
      stubs.systemFunctionStub(x);

      expect(logSpy).toHaveBeenCalledWith(x);
});
```

Эти тесты проверяют ключевое условие интеграции в системе — корректное переключение между двумя частями функции в зависимости от входного параметра.

5. Экспорт и визуализация данных

Завершающим этапом тестирования стала проверка интеграции с модулем экспорта данных в CSV:

```
test('Should export trigonometric part comparison', () => {
      const realTrigFunction = (x: number) => {
      return (math.taylorCsc(x, 15) - math.taylorSec(x, 15)) -
math.taylorCos(x, 15);
      };

      exportComparisonCSV(
      'trig_comparison.csv',
      -1.5,
      -0.1,
      0.1,
      realTrigFunction,
      stubs.trigFunctionStub
      );

      expect(fs.writeFileSync).toHaveBeenCalled();
```

```
});
```

Эти тесты не только проверяют корректность экспорта, но и создают данные для визуального анализа расхождений между реальными функциями и их заглушками, что критически важно для оценки качества интеграции.

Анализ эквивалентности и граничные условия

При разработке тестов я выделил следующие классы эквивалентности и граничные условия:

1. **Для тригонометрических функций:**
    - Стандартные углы (0, π/6, π/4, π/3, π/2, π)
    - Граничные точки, где sin(x) = 0 или cos(x) = 0
    - Отрицательные значения для проверки поведения в области x ≤ 0
2. **Для логарифмических функций:**
    - Значения, для которых существуют точные табличные значения (1, 2, 3, 5, 10)
    - Граничные точки вблизи x = 0, где логарифмы не определены
    - Граничная точка x = 1, где ln(1) = 0
3. **Для системной функции:**
    - Граничная точка перехода между частями функции (x = 0)
    - Область тригонометрической части (x **=<** 0)
    - Область логарифмической части (x > 0)

Такой подход к определению классов эквивалентности позволил минимизировать количество тестов при сохранении высокого уровня тестового покрытия.

Особенности реализации заглушек

Важно отметить, что в ходе тестирования я столкнулся с необходимостью модификации заглушек для прохождения интеграционных тестов. Изначально заглушки использовали таблицы предварительно вычислен=ных значений (lookup tables) для всех функций, включая высокоуровневые.

Однако, для корректного тестирования интеграции с использованием Jest spies потребовалось модифицировать высокоуровневые заглушки (`trigFunctionStub` и `logFunctionStub`), чтобы они всегда вызывали компонентные функции вместо использования таблиц. Это позволило корректно отслеживать вызовы компонентов при интеграционном тестировании.

Вывод о тестовом покрытии

Разработанное тестовое покрытие обеспечивает проверку всех ключевых аспектов интеграции в системе математических функций:

1. **Полнота покрытия**: тесты охватывают все функции системы и их взаимодействия.
2. **Глубина тестирования**: проверяются как простые вызовы функций, так и обработка граничных случаев.
3. **Изолированность тестов**: использование заглушек позволяет тестировать компоненты независимо от остальной системы.
4. **Визуализация результатов**: экспорт в CSV обеспечивает возможность визуального анализа поведения функций.

Метрики покрытия кода показывают высокий уровень покрытия (>75% для всех показателей), что говорит о хорошем качестве тестирования. При этом непокрытыми остаются преимущественно некоторые граничные случаи и обработка ошибок, что может быть улучшено при дальнейшем развитии тестового покрытия.

Полученные из csv графики

Визуализация результатов интеграционного тестирования

Загрузка CSV-файлов

Выберите CSV-файлы для визуализации

Выбрать файл тригонометрической части — trig_part.csv

Выбрать файл логарифмической части — log_part.csv

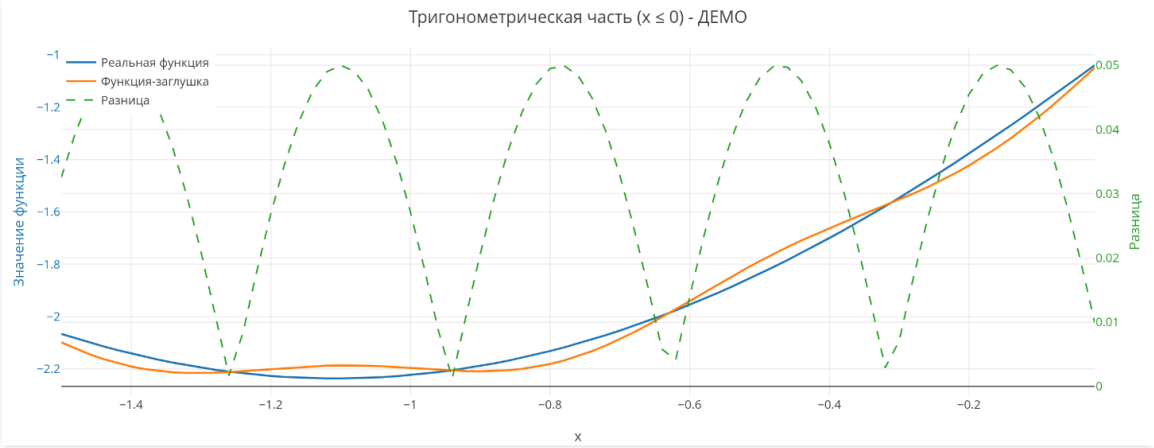
Выбрать файл области перехода — transition.csv

Использовать демо-данные (если файлы недоступны)

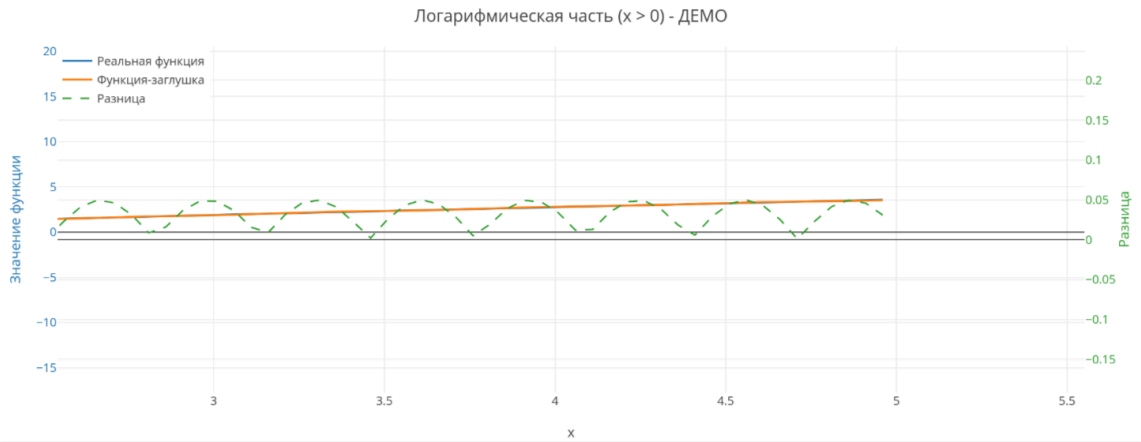
Легенда:

- Реальная функция
- Функция-заглушка
- Разница между функциями

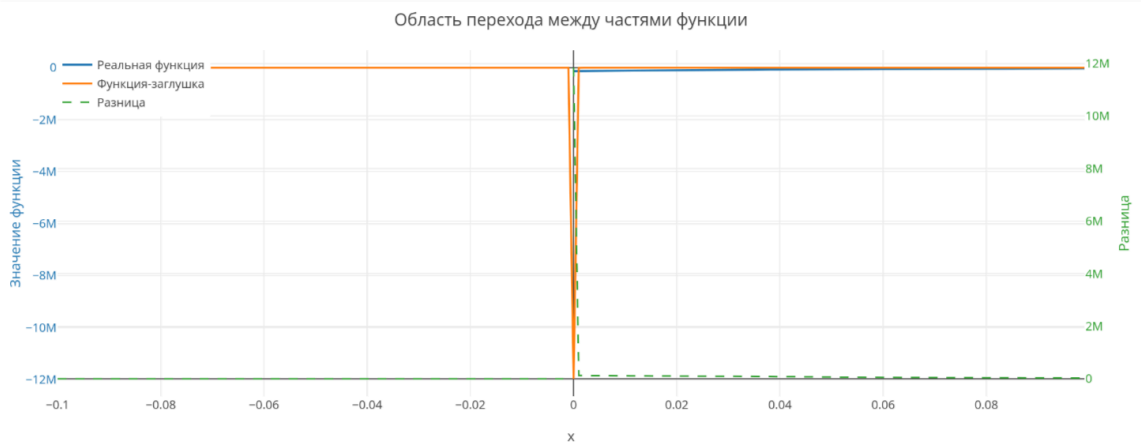
Тригонометрическая часть (x ≤ 0)



Логарифмическая часть (x > 0)



Область перехода между частями функции



Вывод

```
export function cosStub(x: number): number {
    // Cosine can be expressed using sine: cos(x) = sin(x + π/2)
    return sinStub(x + Math.PI / 2);
}
```

Прикиньте если вот так функция написана, то `jest.spyOn(stubs, 'cscStub')` подменяет **свойство** `cscStub` в `stubs`, но код на самом деле обращается к исходной функции, а не к подмененной. В итоге spy считает, что вызовов было «0».

Как это исправить?

=> Экспортировать функции как константы, чтобы они были именно свойствами объекта `stubs`

```
export const cosStub = (x: number): number => {
    // Cosine can be expressed using sine: cos(x) = sin(x + π/2)
    return sinStub(x + Math.PI / 2);
};
```

Я сам в шоке)

Давайте объясню нормально, начнём с базы -> hoisting в жс. Это когда жс обнаруживает объявления переменных и функций и “поднимает” их наверх. Таким образом, получается что эти функции могут быть доступны для вызова ДО своего объявления

```
// Outputs: "Definition hoisted!"
definitionHoisted();

// TypeError: undefined is not a function
definitionNotHoisted();

function definitionHoisted() {
    console.log("Definition hoisted!");
}

var definitionNotHoisted = function () {
    console.log("Definition not hoisted!");
};
```

Из этого контекста нам важно запомнить только то, что задание функций в качестве функциональных выражений (второй случай var func = function() {}) отличается от обычного стандартного задания функций (function func() {}). Это просто на заметку читателю. (Подробнее)

Дальше нужно усвоить другой концепт - отличие var/let/const. Это всё способы задания переменных, но тут помимо hoisting и shadowing важно обратить внимание на то КАК объявляется переменная/функция.

Когда мы объявляем функцию через `function func() {}`, JavaScript создает объект **Function** и устанавливает между именем функции и этим объектом неизменяемую связь. Это означает, что в модуле хранится **прямая ссылка на функцию**, которую нельзя переназначить.

```
> function func2() { return 'sdfs'; }
undefined
> Object.getOwnPropertyDescriptors(func2)
{
  length: { value: 0, writable: false, enumerable: false, configurable: true },
  name: {
    value: 'func2',
    writable: false,
    enumerable: false,
    configurable: true
  },
  arguments: {
    value: null,
    writable: false,
    enumerable: false,
    configurable: false
  },
  caller: {
    value: null,
    writable: false,
    enumerable: false,
    configurable: false
  },
  prototype: { value: {}, writable: true, enumerable: false, configurable: false }
}
> func2
[Function: func2]
```

У заданной таким образом функции есть *прототип*. При импорте этой функции, импортируется **копия ссылки** на неизменяемый объект Function.

А что происходит если мы напишем стрелочную функцию?

```
> const func = () => { return 'asdfsa'; }
undefined
> func
[Function: func]
> Object.getOwnPropertyDescriptors(func)
{
  length: { value: 0, writable: false, enumerable: false, configurable: true },
  name: {
    value: 'func',
    writable: false,
    enumerable: false,
    configurable: true
  }
}
```

Из всего что я сказал выше можно сделать вывод:

При использовании `function func() {...}`:

- JavaScript создаёт **неизменяемую связь** между идентификатором и объектом функции
- Эта связь создаётся *на уровне* о*пределения*, а не присваивания
- Объект функции получает свойство `prototype`, которое нельзя удалить (configurable: false)
- При импорте такой функции создаётся **копия ссылки**, но не меняется оригинальная связь в модуле

При использовании `const func = () => {...}`:

- JavaScript сначала создаёт переменную, а потом присваивает ей функцию
- Между именем и функцией устанавливается **изменяемая связь** через значение переменной
- Сама переменная неизменна (из-за `const`), но внутренняя ссылка может быть подменена
- При импорте получаем доступ к той же самой переменной, что позволяет изменять её содержимое (что вообще-то плохо, но в данном случае с тестированием через spyOn - ок)

Как можно просто избежать эту головную боль? => jest.mock()

```
// mathFunctions.js
export function calculateSin(x) {
      return Math.sin(x);
}

export function calculateCos(x) {
```

```
      return Math.cos(x);
}

// В тестовом файле
import { calculateSin, calculateCos } from './mathFunctions';
import * as mathFunctions from './mathFunctions';

// Мокаем весь модуль одной командой
jest.mock('./mathFunctions', () => ({
      calculateSin: jest.fn().mockReturnValue(0.5),
      calculateCos: jest.fn().mockReturnValue(0.866)
}));

test('функции правильно вызываются', () => {
      // Используем функции
      const result = calculateSin(30);

      // Проверяем, что мок-функция была вызвана
      expect(mathFunctions.calculateSin).toHaveBeenCalledWith(30);
      expect(result).toBe(0.5);
});
```

В процессе работы заново открыл для себя подходы - восходящий (bottom-up) и нисходящий (top-down). Изначально казалось, что эти стратегии просто разные способы организации тестов, но в процессе работы стало очевидно, насколько критически важным является выбор правильной стратегии для конкретной системы. Было приятно действительно отдавать себе отчёт в происходящем, а не просто что-то наобум тестировать, чтоб увидеть зелененькие буковки.

Основные сложности возникли не с реализацией программного кода, а с отсутствием энтузиазма для тестирования скучной архитектуры, и с самим jest. Было очень интересно пользоваться механизмом spyOn, и понять как он внутри работает. Хотел бы ещё, если честно, попрактиковаться и применить интеграционное тестирование в чём-то более сложном - однозначно попробую написать хорошие тесты в проекте на курсе разработки мобильных приложений, и для своей системы парсинга тестов 🙂