Assignment - Part 2
Executing the tests

Done by:

Thanh Pham Vu Van: 152768892

Vizhányó Marcell: 153812860

Course: COMP.SE.200 - Software Testing

Academic year: 2024-2025

Github: https://github.com/noxfilet/software-testing
Coveralls: https://coveralls.io/github/noxfilet/software-testing?branch=main

Table of contents

1. Definitions, acronyms and abbrevations

AI: Artificial Intelligence

CI-pipeline: continuous integration pipeline

Npm: Node Package Manager, which developers can use to find, build and manage code packages

Severity: indicates how serious or impactful a bug is

Frequency: describes how often the issue occurs under the same conditions

2. Introduction

This document provides unit testing report for 10 library functions consisting of its testing process, continuous integration pipeline, final findings and conclusions. As we utilize AI in testing, we also discuss AI's affect on the test. By reading this document, team members should be able to understand the testing process and decision whether E-commerce application is ready for production.

3. Implementation of CI-pipeline and tests

3.1. Alterations to the original test plan:

As documented in the test plan, we are going to test 8 selected library functions and add 2 libraries compared to test plan which we found them being called often in our end-to-end scenarios. We utilize GitHub Actions CI to implement continuous integration, Jest framework as a testing framework and to create coverage data and Coveralls to provide coverage report.

Selected libraries for testing include:

1. add.js
2. capitalize.js
3. countBy.js
4. drop.js
5. eq.js
6. filter.js *(is added after test plan)*
7. get.js
8. isEmpty.js
9. keys.js *(is added after test plan)*
10. words.js

While testing, even though we implemented test cases designed in test plan for *countBy, add, drop, get* functions, many unhappy cases are added using illegal values or edge values such as NaN, null, boolean, and different types of inputs to ensure system behaves correctly under unexpected conditions.

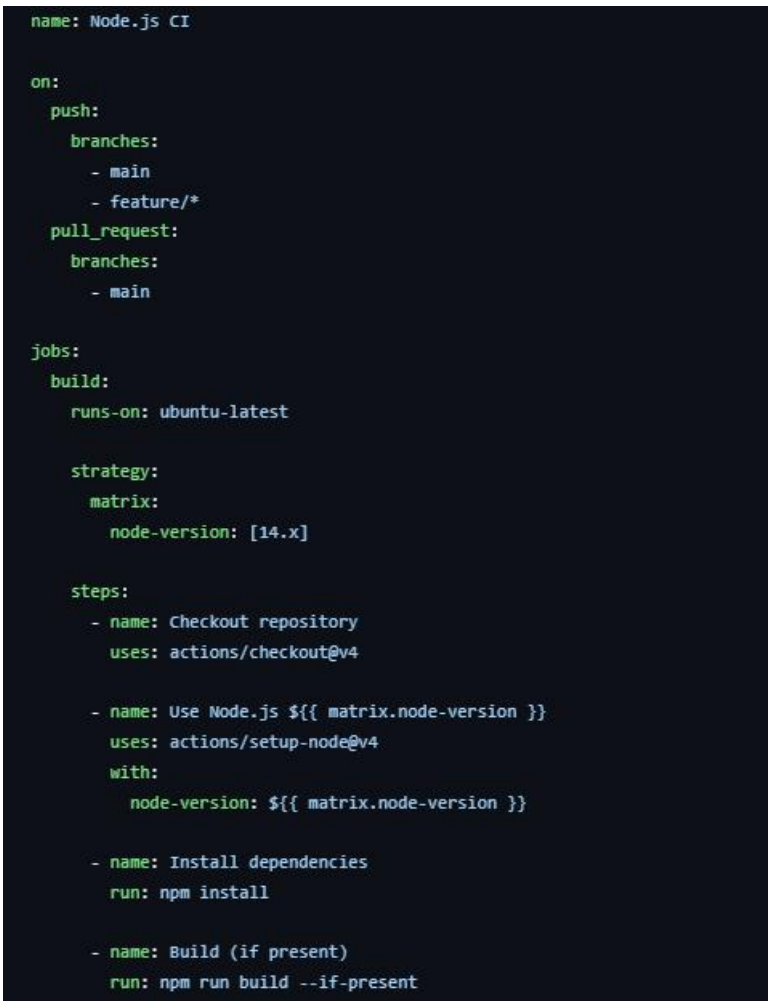
Moreover, we use severity and frequency to evaluate a bug in bug report. Theirs criteria is defined bellow:

Severity: 1-unlikely to cause harm, 2-could cause minor issue, 3-could cause some problems but not critical, 4-significant risk, 5-critical risk.

Frequency: Low-rarely occur, Medium-sometimes occur but not always, High-occur almost every time or always.

3.2. GitHub Actions Workflow:

We implemented automatic tests run, coverage collecting and uploading it to Coveralls. The triggers run on every *push request* to *main* and *feature/** branches, and every *pull reques*t from *main*. The expected Node.js version to run the test is version 14 but other versions related to 14.x are accepted. Then it starts to install all dependencies, which we defined in package.json. If the build script is created before, CI will run it or it starts to run all Jest tests.



```yaml
name: Node.js CI

on:
  push:
    branches:
      - main
      - feature/*
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [14.x]

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Build (if present)
        run: npm run build --if-present
```

After testing, there might be some failed tests that could stop the pipeline, but we expected failed tests to be inevitable in software testing, so these failures are just logged in CI console and the CI continues to send coverage to Coveralls. We can

successfully upload coverage using *"if: always()"* because Jest actually created a coverage report during the test and saved it in *coverage/lcov.info*.

```
      - name: Run tests
        run: npm test

      - name: Send coverage to Coveralls
        if: always()
        uses: coverallsapp/github-action@v2
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
          path-to-lcov: ./coverage/lcov.info
```

As a result, we could receive the newest Coveralls' coverage report and CI fails notification at the same time.

3.3. Run test and get coverage locally:

The unit tests can also be run locally simply by following these commands:

1. Clone the repository:

    git clone https://github.com/noxfilet/software-testing/

    cd software-testing

2. Install dependencies:

    npm install

3. Run tests:

    npm run test // runs all the tests

    npm run test:manual // run self-designed test suites only

    npm run test:ai // run AI-assisted test suites only

After the command "npm run test", you can see the result for each test and the total coverage report. To access Coveralls website for more detailed information, you can browse this.

3.4. Comparison between AI-assisted and self-designed test suites

In our Github repository, in folder *test,* there are two different types of test suites: AI-assisted and self-designed test suites. Test suites which we made on our own are named ***.self.test.js**, and test suites which relied on AI assistance are named ***.ai.test.js.**

With self-designed test suites, we created test cases using pairwise tool. While with AI assistance, we created test suites with respective prompts as below:

- add.js : “write a test suite for a function to add two numbers, test illegal values such as NaN, null, false, and other types of inputs to increase amount of unhappy cases.”
- capitalize.js : “write a test suite for a function to convert the first character of "string" to upper case and the remaining to lower case. In case of non-string input, convert it to string first. Then, test illegal values such as NaN, null, false, and other types of inputs to increase amount of unhappy cases.”
- isEmpty.js : “Write a Jest test suite for the isEmpty() function from a utility library. Focus purely on unusual edge cases that normal unit tests might overlook. Include tests for string edge cases (empty, whitespace, String objects), primitive values, numbers, NaN, booleans, typed arrays, arguments objects, objects created via Object.create(null), inherited vs own properties, symbol keys, non-enumerable properties, objects that look array-like, sparse arrays, built-in objects (Date, RegExp), functions with and without custom properties, Map and Set behavior, and prototype objects.”
- keys.js : “Generate a Jest test suite for the keys() function from the utility library. Focus specifically on edge cases, unusual input types, and behaviors that a typical developer might forget to test. Include tests for prototype properties, symbol keys, non-enumerable properties, sparse arrays, arguments objects, objects created via Object.create(null), and unusual array-like structures.”

For simple functions like “add” and “capitalize”, AI provided similar test suites to our self-designed test suites in only one second. With specific case (for example, non-string input) in “capitalize” function, we have to define clearly in the prompt “In case of non-string input, convert it to string first” to get expected result. In this case, using AI definitely reduces the workload of testing.

However, when we tried with more complex functions like “isEmpty” and “keys”, there were huge differences between the two test suites. The test suites created with AI had many more test cases than our own test suites. For instance, our self-designed test suite had 6 while AI-assisted test suite had 38 test cases for “keys” function. And we had to review all the test cases to make sure they were all correct. This revision increased the amount of time to read and fix rather than to create them ourselves.

After reviewing all the AI-assisted code, we realized there were some test cases we had ignored unintentionally, some of them were so comprehensive that we did not even think about. So we could learn a lot from AI and fill in the lack in our test.

Some test cases we had ignored:

```
test('treats objects with length: 0 and indexed props as non-empty key list', () => {
  const obj = { 0: 'a', length: 0 };

  const result = keys(obj).sort();

  expect(result).toEqual(['0', 'length']);
});

test('treats objects with length: 0 and no indexed props as having only "length"', () => {
  const obj = { length: 0 };

  const result = keys(obj);

  // According to Object.keys({ length: 0 }) -> ['length']
  // If implementation treats array-like differently, this may fail.
  expect(result).toEqual(['length']);
});
```

Some test cases we did not think about:

```
test('object with both string and symbol keys only includes string keys', () => {
  const sym = Symbol('id');
  const obj = { a: 1 };
  obj[sym] = 123;

  const result = keys(obj).sort();

  expect(result).toEqual(['a']);
});
```

```
test('ignores symbol keys and only returns string-keyed properties', () => {
  const sym = Symbol('secret');
  const obj = { visible: 1 };
  obj[sym] = 99;

  const result = keys(obj);

  expect(result).toEqual(['visible']); // symbol key should not be included
});
```

In “isEmpty” function, at first we could not cover line 59 in source code, so we defined specifically a test case for “prototype” using AI and reached expected coverage percentage.

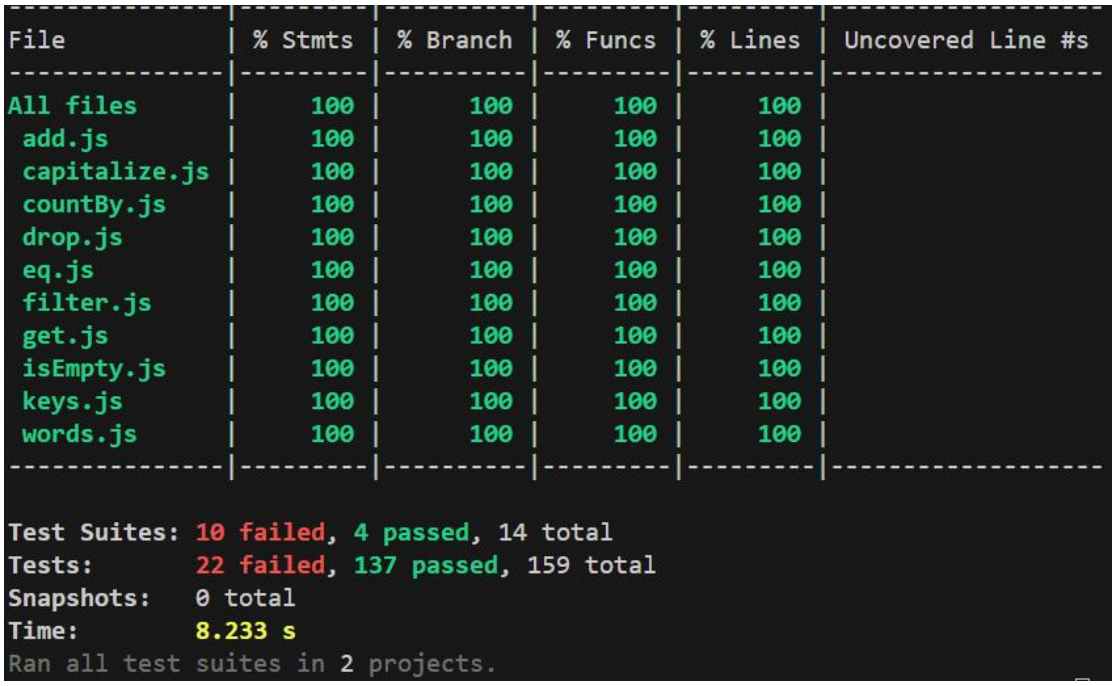
In summary, with AI assistance, test suites are created faster than creating on our own. We do not have to worry about the syntax or typo errors which usefully supports for testing. However, AI-assisted test cases usually skipped edge cases if we

had not defined clearly in the prompts. More important, the correctness is unsure and we have to check carefully for each test case and we need to give proper instructions in the prompts for specific cases. This might take time depending on whether or not tester properly understands the function.

For both types of test suites, we used manual calculation, expected behavior from the source code and general e-commerce rules as our oracle.

4. Findings and conclusions

We created in total 14 test suites with 161 test cases for 10 utility library functions. All test cases can be found in appendices 2 and 3. The picture below indicates the overview of our testing report.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
add.js	100	100	100	100	
capitalize.js	100	100	100	100	
countBy.js	100	100	100	100	
drop.js	100	100	100	100	
eq.js	100	100	100	100	
filter.js	100	100	100	100	
get.js	100	100	100	100	
isEmpty.js	100	100	100	100	
keys.js	100	100	100	100	
words.js	100	100	100	100	

```
Test Suites: 10 failed, 4 passed, 14 total
Tests:       22 failed, 137 passed, 159 total
Snapshots:   0 total
Time:        8.233 s
Ran all test suites in 2 projects.
```

Pass percentage reached 86.16% (137/159) for test cases, while test coverage was 100% for branch coverage, statement coverage, function coverage and line coverage, which ensured:

- No line of code is left untested.
- All lines of code were executed.
- All functional units were tested.
- All logical paths were tested.

Note that this coverage was measured based on selected functions only and not the whole library.

We found 12 bugs (more details in appendix 5), including:

- 2 bugs are considered high severity and high frequency, which are significant issues and need to be fixed before next stage.
- 7 bugs are considered medium severity or medium frequency, which are not critical but still need to be fixed for better performance.
- 3 bugs are considered low severity and low frequency, which should be fixed later.

In summary, **coverage meets expectations** of 100% branch coverage, 80% statement coverage, 80% function coverage and 80% line coverage. However, **86.16% of test cases passing has failed the requirement of 90% pass rate.** And **there are 2 high-severity defects** which also do not meet exit criteria in our test plan.

As the tested libraries failed for unit test, they are not ready for production. These libraries are core functions which are used for E-Commerce application that we evaluated using three most-used scenarios, which leads to the conclusion that E-Commerce application is neither ready for production. Besides, despite having 100% test coverage, the libraries do not mean they are "fully tested". We still need extensive unit testing to make sure they do not reveal new bugs after fixing existing bugs or after any upcoming edge case tests. Therefore, before proceeding to production, E-Commerce application needs to satisfy unit tests' exit criteria first and then complete integration testing, acceptance testing and performance testing.

5. AI and testing

AI tools used: ChatGPT 4.o, Microsoft Copilot (GPT-5)

In part one, we have discussed that AI could be used in different stages. During this unit testing stage, we utilized AI to assist in creating test suites and solved untested code branches as we mentioned above in section 2. We also used AI to check if we lacked any test cases or debug if we ran into any error. For further tests, we believe AI can continue to help with data generation for testing, test design, and defect analysis.

AI is a useful tool if we use it wisely. Compared to phase one, our opinions do not change. AI supported us and saved a considerable amount of time in searching and debugging. Sometimes, when we felt down or discouraged, we could confide to AI as a friend. AI was excellent at being an encourager. However, technically AI could provide incorrect information, we must accept this and check for its accuracy before using it.

6. Course feedback & Learning reflection

Feedback from Thanh Pham:

As a total newbie in software testing, this course taught me a lot. Everything was new knowledge for me to discover. I learned that there were different kinds of testing in each phase, and in each testing, we used different tools and methods and had different requirements. I learned to use models such as the V-model and pyramid model to decide what testing should be implemented and in what order.

I liked the idea of pre-tasks for lectures every week so that I had to read the materials before class to understand at least the general idea of the following lecture. The slides were filled with information that I can use to complete the pre-tasks. However, there wasn't anything for me to recall the knowledge I learned that week, and I almost forgot all of them until I reviewed all the slides preparing for exam. I think practical exercises suited well for this purpose but practical exercises were usually held 2 or 4 weeks after the respective lectures, so when the teaching assistant asked about relevant knowledge, I couldn't answer right away.

On the other hand, practical exercises provided many exciting tools that I can use to test software, and they also helped in final assignment. I would prefer if practical exercises and lectures were close and interlaced with each other rather than separate sessions as now.

About the final assignment, I was almost lost in part 1 because I did not know what to do exactly to meet the requirements of a test plan. I felt that my test plan was good before receiving feedback, but it turned out to lack many details that I could not think about. I hope there was a session where the professor explained the requirements, and we can ask to make sure what we should do. Part 2 of the assignment was supported a lot by practical exercises. From my point of view, I had to learn everything from the start, even how to use npm and code in JavaScript, which were many, but I believed the workload was acceptable.

Feedback from Vizhányó Marcell:

The course was very well put together overall, with some pain points. The lectures and pre-tasks every week were handled very well in my opinion, although I found the lack of sample questions or even a sample exam a bit hard in my preparation for the final. The slides were very informational, maybe even a bit too dense. The lectures were taught well, and the guest lectures were always interesting.

I can't give feedback on the practicals though, since I couldn't take part in them (due to timetable issues, and due to me being an exchange student not having access to linux-desktop.tuni.fi in time).

I found the amount of work the assignment took just right, especially since we could split it up. For part 1, I worked about 6-8 hours on my side, and for part 2 8-9 hours. The most difficult part of it was understanding the format, and the expectations for our submission. The general idea of having an e-commerce store was nice, but we had to skip over a lot of specifics that would actually be implemented in such a store (databases, payment handling and calls, general UI/UX details, authentication and

authorization, performance and scalability) and the testing of these parts, which was quite jarring.

Before this, my only scope of testing in assignments and personal work were simple unit tests for backend logic (and some front-end JS, of course). This course has taught me a lot about good testing practices and workflow, and how it would generally work in the industry. I've also been preparing for my ISTQB CTFL exam before this, and this course has given me sufficient skills for it.

7. References

Pairwise tool: https://www.onlytests.io/tools/pairwise

8. Appendix

Appendix 1. Test plan
Software-testing-test-plan.pdf

Appendix 2. All test cases using pairwise tool
testcases.xlsx

Appendix 3. Example of using pairwise tool to generate test cases for words.js

pattern	string
default pattern	simple ASCII sentence
word characters	simple ASCII sentence
digits only	simple ASCII sentence
null	simple ASCII sentence
""	simple ASCII sentence
default pattern	alphanumeric
word characters	alphanumeric
digits only	alphanumeric
null	alphanumeric
""	alphanumeric
default pattern	unicode characters
word characters	unicode characters
digits only	unicode characters
null	unicode characters
""	unicode characters
default pattern	""
word characters	""
digits only	""
null	""
""	""
default pattern	null
word characters	null
digits only	null
null	null
""	null
default pattern	undefined
word characters	undefined
digits only	undefined
null	undefined
""	undefined

Appendix 4. Overalls report

Appendix 5. Bug reports

All of the following bugs are found using test environment and tools as below:

- OS: GitHub Actions Ubuntu runner (CI) + local Windows development environment
- Node.js: v14.21.3
- Test framework: Jest (with jest.config.cjs)

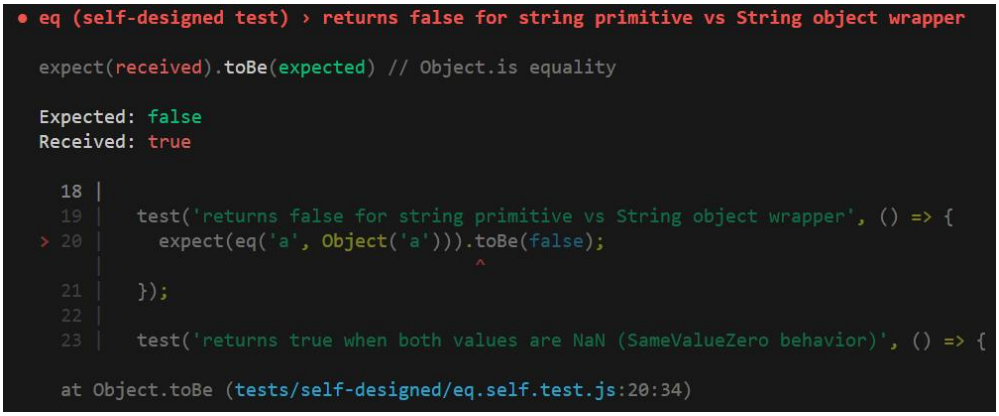
Bug ID or title	01. CountBy first occurrence of each key is 1
Time, date	2025-11-18
Reporter	Vizhányó Marcell

Severity	Issue type	Frequency
4	Calculation Error	High

Description	countBy() under-counts occurrences by 1 for each key. The first occurrence of each key is counted as 0 instead of 1.
Approach	Create a list of items ['shoes', 'shoes', 'bags'] and use the function to count the item by name
Expected result	shoes: 2, bags: 1
Found failure	shoes: 1, bags: 0
Possible cause	First visit to a key uses the else branch, which initializes result[key] to 0 and does not increment.
Attachment	

Bug ID or title 02. eq comparison returns incorrectly for Object
Time, date 2025-11-18
Reporter Vizhányó Marcell

Severity	Issue type	Frequency
3	Logic Error	Medium

Description	eq('a', Object('a')) returns true, but documentation and SameValueZero semantics expect false
Approach	Compare value ‘a’ and Object(‘a’)
Expected result	false
Found failure	true
Possible cause	Implementation uses loose equality (==), causing primitive 'a' and String object Object('a') to be considered equal
Attachment	

Bug ID or title 03. words splits alphanumeric words into numbers and words
Time, date 2025-11-18
Reporter Vizhányó Marcell

Severity	Issue type	Frequency
2	Calculation Error	Medium

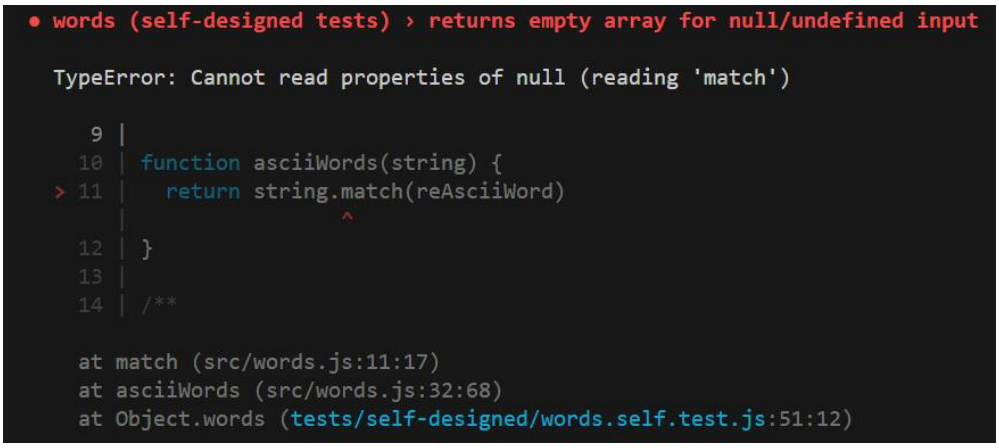
Description	Function splits alphanumeric words into numbers and words in default mode where each alphanumeric chunk is treated as a word

Approach	Use “words” function to split alphanumeric words 'item1 item2a 123abc'
Expected result	["item1","item2a","123abc"]
Found failure	Return ["item","1","item","2","a","123","abc"]
Possible cause	ASCII regex (reAsciiWord) treats digits and letters as separate segments, not merged tokens.
Attachment	

Bug ID or title	04. words cannot handle null/undefined input
Time, date	2025-11-18
Reporter	Thanh Pham

Severity	Issue type	Frequency
2	Null/Undefined Handling	Low

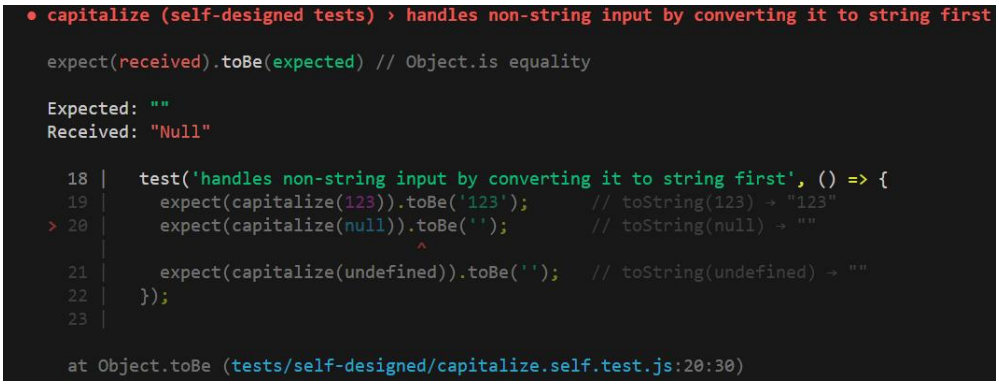
Description	Function returns error if input is null/undefined

Approach	Use “words” function to split null/undefined input
Expected result	[]
Found failure	TypeError: Cannot read property 'match' of null
Possible cause	Cannot read property 'match' of null
Attachment	

Bug ID or title 05. Capitalize function treated null value as “Null”
Time, date 2025-11-18
Reporter Vizhányó Marcell

Severity	Issue type	Frequency
2	Null/Undefined Handling	Low

Description	capitalize(null) returns "Null" instead of expected empty string "" after using toString function
Approach	Input “null” value

Expected result	“”
Found failure	“Null”
Possible cause	Behavior suggests toString(null) → "null" and upperFirst → "Null". May cause null values to show as "Null" in UI instead of blank.
Attachment	

Bug ID or title	06. Filter returns empty nested array
Time, date	2025-11-18
Reporter	Vizhányó Marcell

Severity	Issue type	Frequency
2	Edge Case Failure	Medium

Description	When filter an array in which no elements match predicate or filter an null/undefined array, the function returns empty nested array
Approach	● Filter an array when no elements match predicate, filter([1, 2, 3], n > 100) ● Filter(null/undefined, () => true)
Expected result	Empty array []
Found failure	[[]]

Possible cause	const result = [[]] in filter, which leaves a stray empty array when there are zero matches
Attachment	

Bug ID or title 07. Filter doesn't work if input null as predicate
Time, date 2025-11-18
Reporter Thanh Pham

Severity	Issue type	Frequency
2	Null/Undefined Handling	low

Description	When filter an array in which null is given as predicate, the function runs to error

Approach	Filter anything with null as predicate
Expected result	[]
Found failure	TypeError: predicate is not a function
Attachment	

Bug ID or title 08. keys function returns hole values when working with sparse array
Time, date 2025-11-18
Reporter Vizhányó Marcell

Severity	Issue type	Frequency
4	Calculation Error	High

Description	For sparse arrays where no elements exist at those positions, the function returns the “holes” value unexpectedly. In case of large index arrays, this can dramatically over-report keys
Approach	Create an empty array, then add a value to index 2 Arr[2] = ‘c’
Expected result	Only index “2”

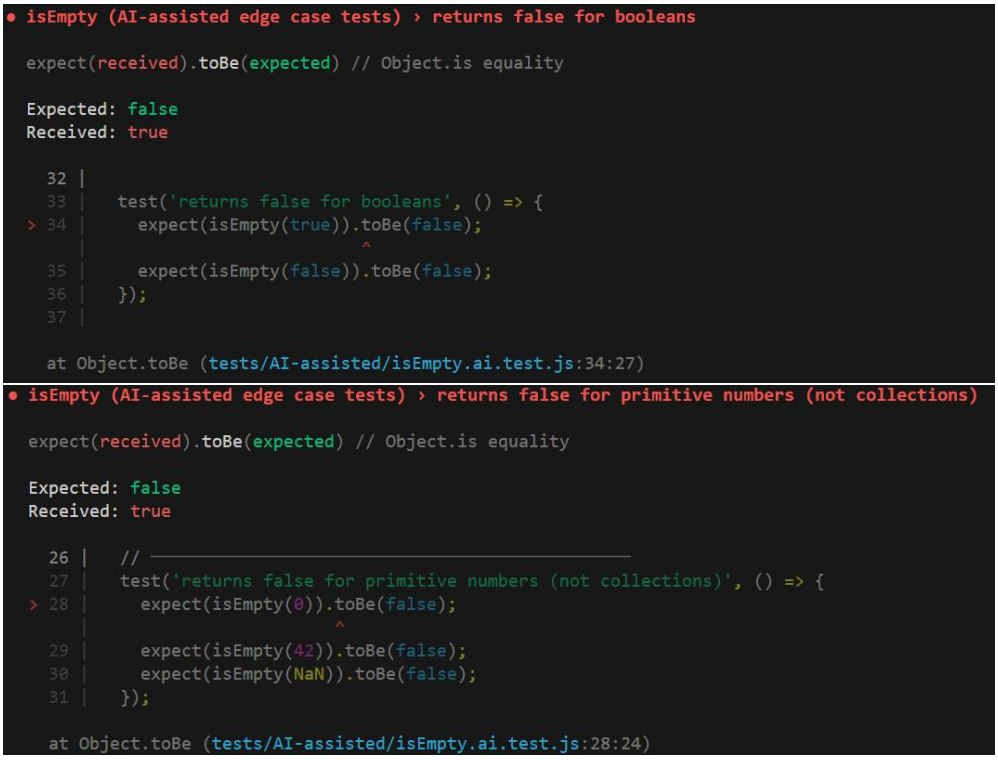
Found failure	Return ["0","1","2"] where “0” and “1” are holes
Possible cause	arrayLikeKeys() treats array holes as enumerable indices, even though they are not actual own properties. This differs from native Object.keys(), which skips sparse positions. Potential mismatch between intended behavior and underlying implementation

Attachment	 ● keys (AI-assisted edge case tests) › handles sparse arrays and only returns indices that exist
	expect(received).toEqual(expected) // deep equality
	- Expected - 0
	+ Received + 2
	Array [
	+ "0",
	+ "1",
	"2",
	]
	136
		// According to Object.keys, only index "2" should exist.
		137
			// If implementation returns ["0","1","2"], this reveals a bug.
			> 138
				expect(result).toEqual(['2']); // only index 2 is a real property
				139
					});

Bug ID or title 09. isEmpty treats all non-object non-collection values as empty
Time, date 2025-11-18
Reporter Vizhányó Marcell

Severity	Issue type	Frequency
2	Type Error	Medium

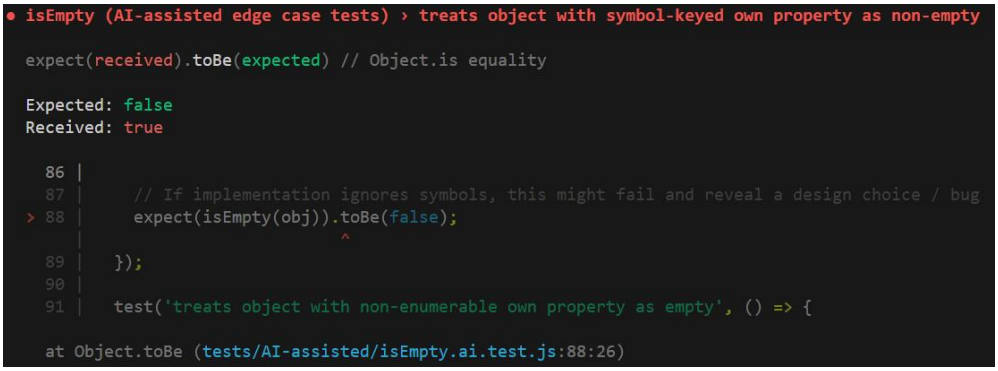
Description	isEmpty(0), isEmpty(42), isEmpty(NaN), isEmpty(true) and isEmpty(false) all return true. Expected false because primitives are not collections and should not be considered “empty”
Approach	Check value 0, 42, NaN, true, false
Expected result	false
Found failure	true
Possible cause	Implementation likely treats all non-object non-collection values as empty. This may be by design, but it contradicts the more intuitive “only collections can be empty” view

Attachment	

Bug ID or title	10. isEmpty ignores symbol keys when determining emptiness
Time, date	2025-11-18
Reporter	Vizhányó Marcell

Severity	Issue type	Frequency
3	Edge Case Failure	Meidum

Description	Implementation appears to ignore symbol keys when determining emptiness. This can hide data stored under symbols and lead to incorrect “empty” classification
Approach	Object with only a symbol-keyed own property is reported as isEmpty(obj) === true. Create: const sym = Symbol('secret'); const obj = {}; obj[sym] = 123;
Expected result	false
Found failure	Return true

Attachment	

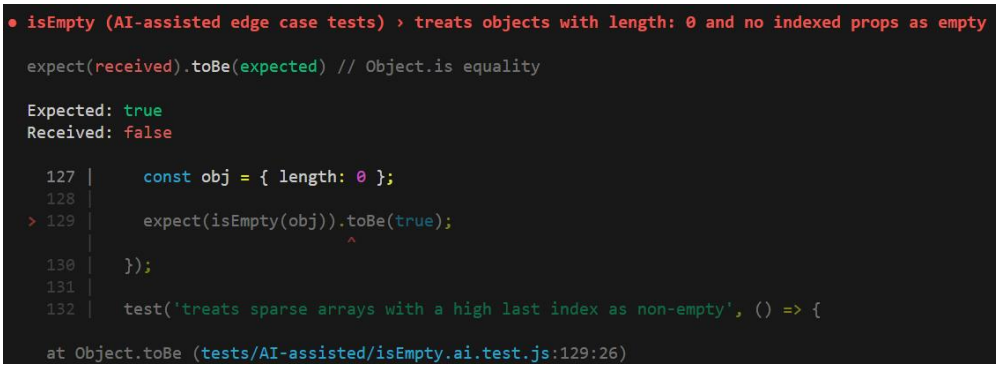
Bug ID or title 11. isEmtpy treats objects with length: 0 and no indexed props as not empty

Time, date 2025-11-18

Reporter Vizhányó Marcell

Severity	Issue type	Frequency
3	Edge Case Failure	low

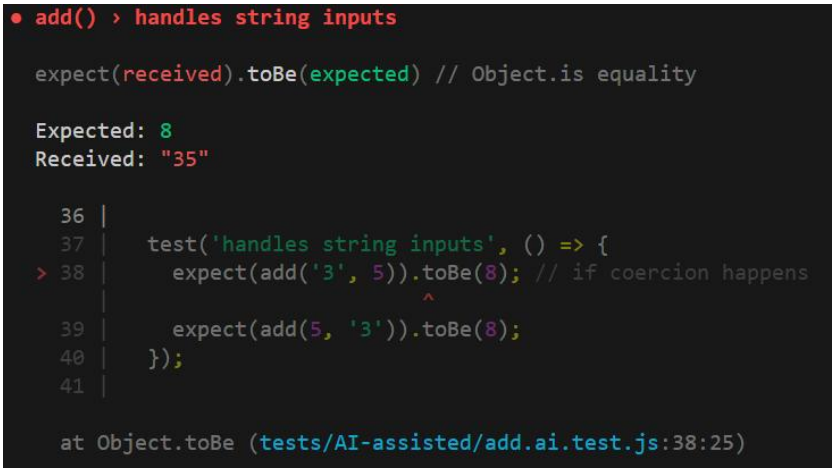
Description	For { length: 0 }, isEmpty(obj) returns false. Expected true, since there are no indexed elements and length alone doesn't represent data content
Approach	Check if objects with length: 0 and no indexed props is empty obj = { length: 0 }
Expected result	true
Found failure	Return false
Possible cause	Implementation likely treats any "array-like" object with a length property as non-empty, even when there are no indexed properties.

Attachment	

Bug ID or title 12. Add function coerces number to string and combine
Time, date 2025-11-18
Reporter Thanh Pham

Severity	Issue type	Frequency
3 (medium)	Calculation Error	medium

Description	Add function should coerce string to number before adding instead of coerce number to string
Approach	Add('3', 5)
Expected result	8
Found failure	"35"
Possible cause	JavaScript coerces number to string if trying to add number and string

Attachment	 ● add() › handles string inputs
	expect(received).toBe(expected) // Object.is equality
	Expected: 8
	Received: "35"
	36 |
	37 | test('handles string inputs', () => {
	> 38 | expect(add('3', 5)).toBe(8); // if coercion happens
	| ^
	39 | expect(add(5, '3')).toBe(8);
	40 | });
	41 |
	at Object.toBe (tests/AI-assisted/add.ai.test.js:38:25)