Sniffing Out Test Smells in JavaScript: An Open-source Detection Tool

Holm Smidt
Dr. Anthony Peruma
MS Plan B / ICS 699, Spring/Summer 2023

Introduction

The quality of a software project is determined by its ability to implement functional requirements according to its specifications. Quality assurance encompasses techniques that help teams ensure high-quality software in a project. Amongst these techniques, unit testing is a widely used technique to verify that the production code meets functional requirements [1].

As the production code evolves, unit tests can be used as a quality gate in the continuous integration process to ensure that unit tests cover production code and that incremental code changes do not break any existing unit tests. Design and maintainability of test code are thus just as important as the design and maintainability of production code itself; yet, less time and consideration is given to the implementation and refactoring of test code [2].

A test smell refers to any symptom in the test code that indicates a deeper problem [3]. Automated test smell detection tools can help identify common anti-patterns in the test code.

Motivation

Much work has been done on classifying test smells and developing open-source test smell detection tools [4]. There is a lack of research examining test smells for JavaScript [5], despite JavaScript being the most widely used programming language in 2022 [6].

The lack of open-source automated JavaScript test smell detection tools also hinders research in this area and provides no support for industry practitioners in maintaining their code. This work thus aims to provide an extensible open-source automated JavaScript test smell detection tool to be used by researchers and practitioners. With the Jest testing framework being the most used JavaScript testing framework [7], this work focuses on the Jest framework.

JavaScript Test Smells

This research study targets test smells in JavaScript unit tests utilizing the Jest framework. Following are the test smells:

- Assertion Roulette
- Missing Await/Return
- Forgotten Test.Only
- Missing Test Fixture

Assertion Roulette

Since the Jest framework does not provide the option for assert messages, we propose that one should only use one assertion per test as it would otherwise be difficult to identify which assertion is failing.

```
// anti-pattern
describe('test basic arithmetic', () => {
  it('ensure addition works', () => {
    expect(2 + 2).toEqual(4);
    expect(2.5 + 3).toEqual(5.5);
  });
});

// recommended implementation
describe('test basic arithmetic', () => {
  it('ensure integer addition works', () => {
    expect(2 + 2).toEqual(4);
  });
  it('ensure decimal addition works', () => {
    expect(2.5 + 3).toEqual(5.5);
  });
});
```

Listing 1: Sample code illustrating Assertion Roulette.

Missing Await/Return

Asynchronous code is commonly encountered in JavaScript. The Jest framework thus recommends always using an `await/return` for promises as the test may otherwise finish before the promise resolves or rejects.

```
// anti-pattern
test('the data is peanut butter', async () => {
  expect(fetchData()).resolves.toBe('peanut butter');
});
test('the fetch fails with an error', async () => {
  expect(fetchData()).rejects.toMatch('error');
});


// recommended implementation
test('the data is peanut butter', async () => {
  return expect(fetchData()).resolves.toBe('peanut butter');
});
test('the fetch fails with an error', async () => {
  await expect(fetchData()).rejects.toMatch('error');
});
```

Listing 2: Sample code illustrating Missing Await/Return.

Forgotten Test.only

Jest offers and recommends the `test.only` construct to isolate a test in a test suite for debugging purposes. Test suites containing this construct beyond the development process are problematic as they cause remaining tests in a suite to be excluded.

```
// anti-pattern
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

// recommended implementation
test('this test will not run', () => {
  expect('A').toBe('A');
});
```

Listing 3: Sample code illustrating Forgotten Test.Only tests.

Missing Test Fixture

Test fixtures are essential to setting the state prior to test execution. Setup and Teardown helper functions can reduce code redundancy. They can further eliminate non-fixture code in describe handlers, which can ultimately reduce flaky tests due to Jest's order of code execution.

```
// antipattern
initializeCityDatabase();
test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});
clearCityDatabase();

// recommended implementation
beforeEach(() => {
  initializeCityDatabase();
});
afterEach(() => {
  clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

Listing 4: Sample code illustrating Missing Test Fixtures.

Automated Test Smell Detection Tool

The tool is implemented in Typescript, uses the Factory Method design pattern to make it extensible for future development, and targets test code implemented in the Jest framework.

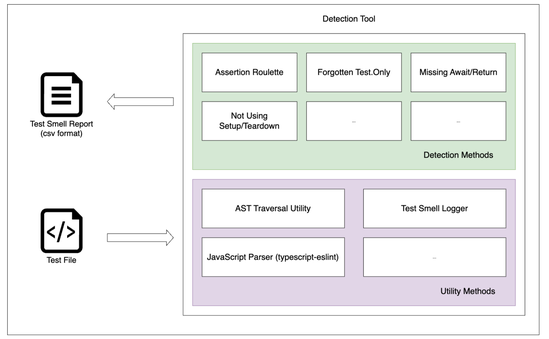

Figure 1: Architecture of the test smell detection tool.

Conclusion

Writing high-quality test code is paramount to the maintainability and efficacy of an automated test suite. Like production code, test code is susceptible to anti-patterns, i.e. test smells. To help researchers study test smells in JavaScript projects and assist practitioners in avoiding test smells, an automated JavaScript test smell detection tool was developed that analyzes unit tests using the Jest framework.

Future Work

Future work includes the evaluation of the correctness of the developed tool. A sample of open-source JavaScript projects will be used to verify its precision via manual introspection of reported test smells.

References

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in Future of Software Engineering (FOSE '07), 2007, pp. 85–103.
[2] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: Patterns, beliefs, and behavior," IEEE Transactions on Software Engineering, vol. 45, no. 3, pp. 261–284, 2019.
[3] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 1–12.
[4] W. Aljedaani, A. Peruma, A. Aljohani, M. et al., "Test smell detection tools: A systematic mapping study," in Evaluation and Assessment in Software Engineering, ser. EASE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 170–180.
[5] D. Jorge, P. Machado, and W. Andrade, "Investigating test smells in javascript test code," in Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing, ser. SAST '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 36–45.
[6] "Stack overflow developer survey 2022." [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof
[7] "State of JS survey 2022." [Online]. Available: https://2022.stateofjs.com/en-US/