# An Empirical Study on the Detection of Test Smells in Test Codes Generated by GitHub Copilot

**Victor Anthony Alves**[1], **Carla Bezerra**[1], **Ivan Machado**[2]

[1]Federal University of Ceará (UFC) - Quixadá, Ceará, Brazil

[2]Federal University of Bahia (UFBA) - Salvador, Bahia, Brazil

victorpa@alu.ufc.br, carlailane@ufc.br, ivan.machado@ufba.br

***Abstract.*** *Various techniques for automatically generating unit tests have been studied. The use of Large Language Models (LLMs) has recently emerged as a popular approach for automatic test generation from natural language descriptions. This study aims to measure the quality of the test codes produced by LLMs by detecting test smells in the test cases generate. To do this, we proposed an empirical study and a quality assessment methodology to be performed for each LLM that generates code. In our preliminary results, we applied these procedures with GitHub Copilot and obtained significant data on the quality of test codes. These findings indicate that although GitHub Copilot can generate valid unit tests, quality violations are still frequently found in these codes.*

## 1. Introduction

The creation of unit test code is often overlooked by developers, predominantly due to its complexity and the time-consuming nature of the task [Beller et al. 2015, Daka et al. 2015, Runeson 2006]. The manual generation of tests is not only labor-intensive but also costly [Li 2022]. Consequently, there is a growing emphasis within the development community on developing tools and methodologies that automate the generation of unit tests [Serra et al. 2019].

Recent research by Schäfer et al. 2024 and El Haji et al. 2024 highlights the substantial capabilities of Large Language Models (LLMs) in generating test code. These studies commonly employ a strategy where LLMs are pre-trained using natural language prompts. This method capitalizes on the models' human-like text comprehension and generation abilities, thereby facilitating the creation of customized test code for distinct use cases [Yu et al. 2023, Yetistiren et al. 2022]. LLMs' depth of knowledge and contextual understanding enables them to transcend traditional testing limitations, probing diverse scenarios and identifying potential issues that conventional methods might overlook.

While LLMs are increasingly indicated as the future of code generation, a growing body of research is focused on evaluating the code quality they produce [Hansson and Ellréus 2023, Yetiştiren et al. 2023]. However, in the domain of test code generation, although numerous studies have addressed the generation capabilities of LLMs, a significant research gap remains concerning the quality of test code produced by these models [El Haji et al. 2024, Siddiq et al. 2024]. Traditional testing methodologies often encounter issues with test cases, commonly known as *test smells*, which may signal deficiencies in test design or implementation. These shortcomings can lead to decreased efficiency in detecting failures or validating software behavior [Tufano et al. 2016]. Test smells can manifest in various forms, such as poorly structured test code or complex logic, which complicates both understanding and maintenance of the code [Kim et al. 2021].

The main objective of this project is to conduct an empirical study on the quality of test code generated by LLMs, focusing on the detection of test smells. To conduct this study, we formulated three research questions and defined a series of steps to answer them. These steps include selecting the LLM to be analyzed, generating test codes for open source projects using that LLM, and finally detecting test smells with the help of tools and industry professionals. Our preliminary results evaluated these aspects with GitHub Copilot in Python. This study provides findings on the frequency of detection of test smells, the most common types of test smells and identifiable patterns, as well as practitioners' perceptions of the quality of these codes.

## 2. Background

### 2.1. Unit Testing & Test Smells

Unit testing is a technique designed to detect defects and validate the functionality of the smallest testable parts of software, such as modules, objects, and classes, which can be examined in isolation [Graham et al. 2021]. This practice is instrumental in preventing programming errors and identifying issues early in the development cycle [Peng et al. 2021, Khorikov 2020]. In this context, certain problems, known as test smells, may affect the quality of the tests. These issues generally result from poor design choices made during the implementation of test cases, affecting test effectiveness [Palomba et al. 2018, van Deursen et al. 2001]. Test smells often originate when test code is initially committed to a repository and are likely to persist, adversely affecting the software's maintainability and directly impacting its quality [Tufano et al. 2021, Santana et al. 2020, Kim 2020].

### 2.2. Automatic test generation by LLMs

Automated test generation facilitates the production and execution of numerous inputs that thoroughly test software units [Xie and Notkin 2006]. LLMs have assumed a role in test generation driven by Natural Language Processing (NLP) [Yu et al. 2023]. The advent of automatic code generation via LLMs offers substantial potential to reduce the time and costs associated with manual coding [Hansson and Ellréus 2023]. By training on human language inputs, models such as OpenAI's Codex[1] can generate code snippets, documentation, and even repair bugs. This functionality has been extended in applications like GitHub Copilot and ChatGPT, which has been extensively studied for its capability in test code generation [El Haji et al. 2024, Schäfer et al. 2024]. Despite these advancements, there remains a gap in the evaluation of the quality of code generated by these models.

## 3. Related Work

El Haji et al. 2024 conducted an experiment with the Codex version of GitHub Copilot, investigating the usability of different test cases generated using command prompts. They found that a comment combining instructive natural language with an example of code usage resulted in more usable test generations. Similarly, Yu et al. 2023 analyzed ChatGPT's ability to generate mobile test scripts. This study's findings indicate that

[1]`https://openai.com/index/openai-codex/`

ChatGPT can generate useful tests when provided with sufficient context and information about the project architecture. Although several studies [Siddiq et al. 2024, Schäfer et al. 2024] have found promising results on the capability and limitations of LLMs, they did not evaluate the design quality of the generated test code, focusing mainly on the validity of these codes. For this study, we used the test generation methods of the two works [El Haji et al. 2024, Schäfer et al. 2024].

Several studies have already been conducted by researchers to assess its quality for production code. Yetistiren et al. 2022 examined the quality of the code produced by GitHub Copilot, looking at aspects such as efficiency and design. They observed that although Copilot is able to generate valid code, many of them still have problems related to efficiency and design. In complementary research, Yetiştiren et al. 2023 investigated the presence of code smells in the codes generated by Amazon's CodeWhisperer, GitHub Copilot and ChatGPT. They found that certain code smells tend to recur in the generated code and that the LLMs themselves have the ability to fix these maintenance problems. This pattern of repeating problems that impact code maintainability was also identified by Hansson and Ellréus 2023, but the authors emphasized that ChatGPT and Copilot have proven to be more effective in generating quality code. Thus, although there have been several evaluations of the quality of code produced by LLMs, these analyses have focused mainly on production code and have not specifically addressed test code. From this works we were able to extract code quality analysis techniques and adapt them to test code.

## 4. Study Design

### 4.1. Goals and Research Questions

This study aims to investigate the detection of test smells in test codes generated by LLMs, both in the perception of professionals and through the use of detection tools. Python was chosen for this investigation due to its extensive support by Codex [Chen et al. 2021]. To achieve these goals, three research questions were formulated:

**RQ$_1$:** *How often do detection tools and industry professionals identify test smells in the test codes generated by LLMs?* This question aims to compare the incidence and frequency of test smells in test codes generated by LLMs, as identified by detection tools and industry professionals.

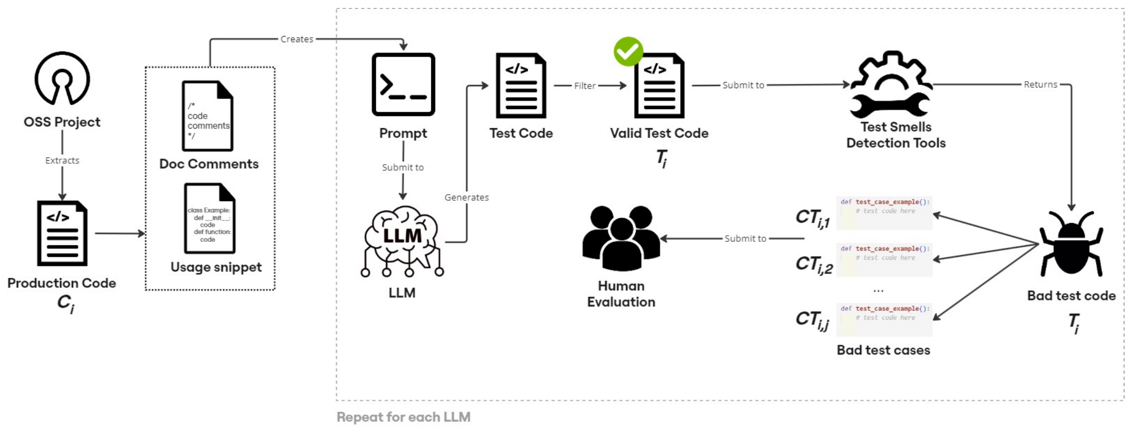
**RQ$_2$:** *What are the most common test smells identified by detection tools in test codes generated by LLMs?* This question seeks to identify the most common types of tests smells carried out by detection tools and the professionals in the test codes generated by LLMs.

**RQ$_3$:** *What other quality violations did the professionals find in the test codes generated by the LLMs?* This question aims to identify other quality violations encountered by professionals that are not yet formally categorized as test smells.

### 4.2. Study Settings

For this study, it is necessary to follow some steps to answer the research questions, as detailed in Figure 1. First, open source projects were selected. Then, LLM must be configured to generate test code for these projects. In this study, the experiment was carried out with GitHub Copilot, so the methodological procedures were performed only once. But during our future work, these methods will be performed for each LLM evaluated. In the third phase, test smell detection tools must be applied to identify possible quality violations. Finally, an evaluation of these codes must be carried out by professionals in the field to collect their perceptions about the quality of the tests generated, thus answering the research questions.

**Figura 1. Methodological procedures**



### 4.2.1. Project Selection

For this study, we selected 4 open source Python projects from GitLab and GitHub presents in Table 1. Furthermore, we chose projects that already had unit tests to facilitate the identification of the most testable parts of the production code. However, unit tests must be disregarded for training LLMs. This approach aims to ensure a more complete analysis of the test codes generated by LLMs without influence from those tests already present in the project. The training must be carried out individually for each project, aiming to minimize information conflicts and the occurrence of hallucinations [Martino et al. 2023] during test generation.

**Tabela 1. List of open-source projects used in this study**

| Project | Domain | # LOC | # Classes | # Functions |
|---|---|---|---|---|
| python-lottie | File manipulation | 20770 | 442 | 1573 |
| click | CLI | 10157 | 67 | 508 |
| pyflunt | Domain Driven Design (DDD) | 2969 | 14 | 200 |
| brutils-python | Python Library | 2849 | 25 | 140 |

### 4.2.2. Test Code Generation

As shown in Figure 1, Python files containing classes, methods or functions with testable returns were considered. Thus, for each production code file $C_i$, a test Python file $T_i$ was generated. For each pair $(C_i, T_i)$, LLMs must generate the test case $CT_{i,j}$, with $j$ being able to vary from 5 to 20 test cases per file. The libraries used to generate the test codes were pytest[2] and unittest[3]. To generate each test file $T_i$, a natural language prompt

[2]`https://docs.pytest.org/en/8.2.x/`
[3]`https://docs.python.org/3/library/unittest.html`

must be sent so that the LLM can generate the tests. Next, the test cases must be filtered, excluding those with errors in the interpretation of Python, such as syntax errors, invalid references or hallucinations. Tests that fail, even without interpretation errors, will also be discarded. No manual adjustments should be made to the codes generated by the LLMs; only the filtered cases will be used in the quality assessment.

### 4.2.3. Test smells detection tools

Once the test cases had been generated and filtered, a series of Python-specific tools must be selected to detect test smells in the code. The tools that provide the most support for detecting test smells in Python are: Pynose [Wang et al. 2022], TEMPY [Fernandes et al. 2022] and pytest-smell [Bodea 2022]. Each of them has its own techniques and approaches for identifying and categorizing test smells, providing a broader view of the quality of the test code generated by LLMs. We call *bad test codes* those Python $T_i$ files in which at least one test smell has been identified in the corresponding test cases, which we call *bad test cases*. Table 2 lists the test smells that were addressed in this study so far in test codes generated by GitHub Copilot and the tools that identified them.

**Tabela 2. Test smells that have been covered in this study so far**

| Test smell | Tools | Description |
|---|---|---|
| Assertion Roulette | Pynose, pytest-smell | Several asserts without any explanation or message [Wang et al. 2022] |
| Magic Number Test | Pynose, pytest-smell | Existence of literal numeric values in a test [Wang et al. 2022] |
| Unknow Test | TEMPY | Tests without assertions [Fernandes et al. 2022] |
| Conditional Test Logic | Pynose, pytest-smell, TEMPY | Tests with control statements (if, for, while...) [Wang et al. 2022] |
| Eager Test | pytest-smell | Tests that invoke multiple methods of production code [Bodea 2022] |
| Duplicate Assert | pytest-smell | Duplicate assertions in the same test [Bodea 2022] |
| Test Maverick | Pynose | If the test suite has a fixture with setup, but a test case in this suite does not use this setup [Wang et al. 2022] |

### 4.2.4. Practitioners' Assessment

For answer the research questions of this paper, we conduct an evaluative survey with industry professionals containing six sets of test cases generated by GitHub Copilot, called $S_i$, so that professionals could assess the quality of each set. Each set contained 2 to 3 test cases from the same $T_i$ file. The professionals were given a list of characteristics corresponding to the description of each test smell detected (column *Description* of Table 2) and had to mark which of these characteristics they could identify in each set of test cases. They also had the opportunity to describe other characteristics not mentioned and suggest improvements to the code. To improve the validity of the evaluations, we do not inform practitioners that quality problems have been detected by the tools. For this paper, A total of 20 professionals, mostly Software Developers or Software Quality Assurance with 1 to 7 years of experience in the industry, took part in the evaluation.

## 5. Results

This section presents the results of the research, currently, we carried out the steps described in the previous section with GitHub Copilot and were able to extract data relating to this LLM. A sample of 30 pairs $(C_i, T_i)$ was generated by Copilot, totaling 397 test cases. After filtering, the sample was reduced to 194 valid test cases.

### 5.1. $RQ_1$: Test smell detection frequency

To determine the frequency detected by the tools, we recorded how many times a tool identified a test smell in a test case. We counted the detections of all three tools. In some cases, more than one tool identified the same test smell in the same test case. To avoid double counting, in this scenario we only considered one detection. Of the 194 test cases analyzed, 92 had at least one occurrence of test smell, representing 47.4% of the total. Of the 30 test files generated, 21 contained at least one test smell detection. From this, we observed that the presence of a single test smell per test case is more common, while the occurrence of up to 3 test smells in the same test case is less frequent. Table 3 shows the three statuses detected in the test set generated by Copilot, the detection frequency for each test case $CT_{i,j}$ and for each test file $T_i$.

**Tabela 3. Test smells detection frequency by tools**

| **Detection Status** | $CT_{i,j}$ | $\%CT_{i,j}$ | $T_i$ | $\%T_i$ |
|---|---|---|---|---|
| 1 test smell detected | 84 | 43,2% | 15 | 60% |
| 2 test smell detected | 6 | 3,1% | 4 | 16% |
| 3 test smell detected | 2 | 1,1% | 2 | 8% |
| **TOTAL** | **92** | **47,4%** | **21** | **84%** |

To determine how often professionals detected test smells, we counted how many times each practitioner pointed out the presence of these problems in each set of test cases $S_i$. We then calculated the average number of detections per status to identify which detection status was the most prevalent among the professionals. In total, twenty professionals took part in the evaluation, and their detections were counted on a unit basis, as shown in Table 4. The columns referring to $S_i$ denote the six sets of test cases evaluated and the absolute frequency with which professionals in their evaluations identified test smells. It was observed that the status with at least three test smells detected was the most frequent among the groups, with an average of 8.8. It is important to note that all the professionals identified at least one test smell in the sets analyzed.

**Tabela 4. Test smells detection frequency by practitioners**

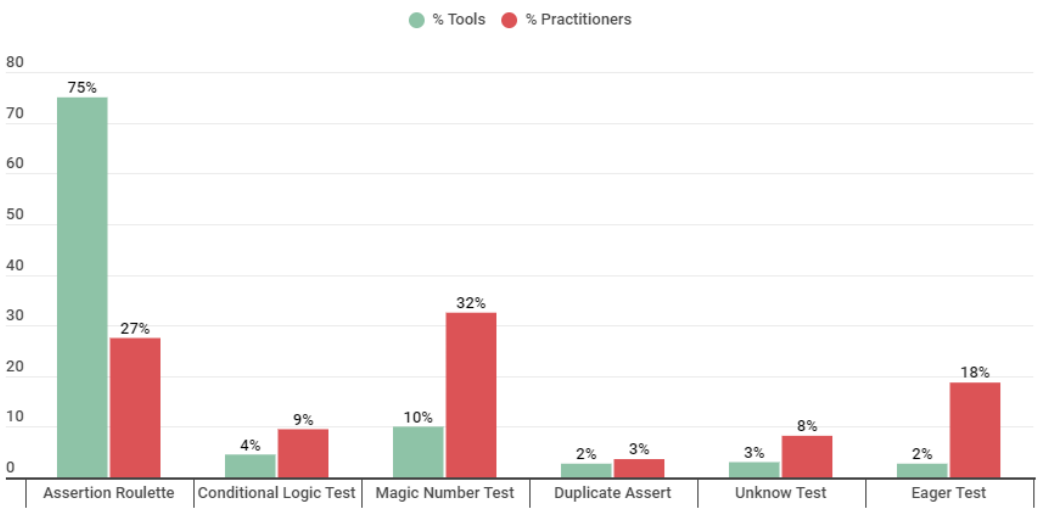
| **Detection Status** | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | **AVERAGE** |
|---|---|---|---|---|---|---|---|
| 1 test smell detected | 1 | 2 | 2 | 2 | 9 | 2 | **3** |
| 2 test smell detected | 7 | 5 | 8 | 6 | 6 | 1 | **5.5** |
| 3 test smell detected | 11 | 6 | 10 | 11 | 5 | 10 | **8.8** |
| 4 test smell detected | 1 | 7 | 0 | 1 | 0 | 7 | **2.6** |

### 5.2. $RQ_2$: Types of test smells detected

To answer this question, we counted the occurrences of test smells according to their type and calculated the corresponding percentage for each of these types. This calculation was made both for occurrences detected by the tools and by professionals. Figure 2 shows the relative frequency of detections for each type of test smell in both contexts.

In the context of the detections made by the tools, Figure 2 reveals that the smell *Assertion Roulette* was the most prevalent, occurring in 75% of the test cases. Next, the smell *Magic Number Test* was identified in 10% of cases, followed by *Conditional Test Logic* (4%) and *Unknown Test* (3%). In the context of detections made by industry professionals, the types of test smells were more evenly distributed. The most prevalent test smell was the *Magic Number Test*, accounting for 32% of detections. It is also possible to observe a significant difference in the detection of the *Eager Test* test smell, whose percentage of detection was much higher among professionals, reaching 18%.

**Figura 2. % Occurrences of test smells by type**



These results clearly show in the tests generated by Copilot, such as the excessive use of undocumented assertions, presence of literary numeric values, complexity in conditional logic, absence of assertions, duplication of tests and excessive calls to methods in the production code. For example, the high incidence of the test smells *Assertion Roulette* and *Magic Number Test* in both contexts (tools and professionals) suggests that the test codes generated follow a specific structural pattern. This indicates that many of the tests were produced without clear documentation of expectations and with the use of static literals instead of dynamic values. This raises concerns about the suitability of automated tests in terms of maintenance, clarity and effectiveness.

## 5.3. RQ$_3$: Others practitioners' perceptions

Qualitatively, the professionals also offered opinions and suggestions on the quality of the test codes that went beyond the test smells included in the questionnaire. We noticed from the responses that some professionals identified very similar tests due to the naming of the variables in the methods being the same or showing only minimal differences. Several other problems related to readability were also identified, accompanied by suggestions for improvement. A report on this topic to illustrate:

*P9:"It is interesting to assign the positional parameters of the classes to variables, in order to facilitate the understanding of the meaning of each argument passed. Just by reading the test, you can't debug its purpose"*

*P16:"Create a function to generate and add this service being tested, so that it can be reused in both tests just by invoking it, thus avoiding code duplication."*

In addition, professionals pointed out that the tests generated by Copilot did not take advantage of most of the resources offered by the *pytest* and *unittest* libraries, which may have contributed to the problem of code readability. Some reports on these problems were left on the form:

*P11:"Pytest has some features that could make this test more efficient, and the same goes for the previous one."*

*P12:"You can use asserts such as assertTrue, assertEquals (unittest), etc. These would make the conditions simpler."*

From the perceptions shared by the professionals, it was possible to infer that, in addition to the test smells detected by the tool, other quality violations were identified in the test codes generated. This suggests that the flaws present in the tests produced by Copilot are not just limited to problems that can be automatically detected by tools, but are also noticeable through manual and critical analysis by the developers. This result underlines the importance of human review to identify quality aspects that may escape automated tools, showing that developers play a crucial role in evaluating and improving the test codes generated by LLMs.

## 6. Conclusion and Future Work

In this paper, we suggest an empirical study on the quality of test code generated by LLMs, our preliminary results were carried out with GitHub Copilot for Python. To do this, we designed three research questions and generated a total of 194 valid test cases in 30 Python files. We analyzed how often the tools and practitioners detected smells in the sample of test cases generated by GitHub Copilot and found that approximately 47% of the test cases contained at least one test smell according to the tools. In addition, we investigated which types of test odors were most commonly found and found that *Assertion Roulette* and *Magic Number Test* were the most prevalent. Finally, we analyzed other professionals' perceptions of these test cases, revealing aspects different from those reported by the tools, such as readability problems, code repetition and failure to take advantage of the resources offered by the libraries.

As the next steps in the research, we intend to repeat the proposed methodological procedures using other LLMs with the aim of obtaining a broader data sample. With this new sample, it will be possible to carry out a detailed comparison of the quality indicators of the test codes generated by each LLM.

## Referências

Beller, M., Gousios, G., Panichella, A., and Zaidman, A. (2015). In *When, how, and why developers (do not) test in their IDEs*, ESEC/FSE 2015, page 179–190, New York, NY, USA. Association for Computing Machinery.

Bodea, A. (2022). Pytest-smell: a smell detection tool for python unit tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 793–796, New York, NY, USA. Association for Computing Machinery.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Daka, E., Campos, J., Fraser, G., Dorn, J., and Weimer, W. (2015). Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 107–118, New York, NY, USA. Association for Computing Machinery.

El Haji, K., Brandt, C., and Zaidman, A. (2024). In *Using GitHub Copilot for Test Generation in Python: An Empirical Study*, AST '24, page 11, New York, NY, USA. ACM.

Fernandes, D., Machado, I., and Maciel, R. (2022). Tempy: Test smell detector for python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 214–219, New York, NY, USA. Association for Computing Machinery.

Graham, D., Black, R., and van Veenendaal, E. (2021). *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning.

Hansson, E. and Ellréus, O. (2023). Code correctness and quality in the era of ai code generation : Examining chatgpt and github copilot.

Khorikov, V. (2020). *Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#*. Manning.

Kim, D. J. (2020). An empirical study on the evolution of test smell. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 149–151.

Kim, D. J., Chen, T.-H., and Yang, J. (2021). The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26.

Li, C. (2022). In *Mobile GUI test script generation from natural language descriptions using pre-trained model*, MOBILESoft '22, page 112–113, New York, NY, USA. Association for Computing Machinery.

Martino, A., Iannelli, M., and Truong, C. (2023). Knowledge injection to counter large language model (llm) hallucination. In Pesquita, C., Skaf-Molli, H., Efthymiou, V., Kirrane, S., Ngonga, A., Collarana, D., Cerqueira, R., Alam, M., Trojahn, C., and Hertling, S., editors, *The Semantic Web: ESWC 2023 Satellite Events*, pages 182–185, Cham. Springer Nature Switzerland.

Palomba, F., Zaidman, A., and De Lucia, A. (2018). Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322.

Peng, Z., Lin, X., Simon, M., and Niu, N. (2021). Unit and regression tests of scientific software: A study on swmm. *Journal of Computational Science*, 53:101347.

Runeson, P. (2006). A survey of unit testing practices. *IEEE Software*, 23(4):22–29.

Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., and Machado, I. (2020). Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 374–379.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Serra, D., Grano, G., Palomba, F., Ferrucci, F., Gall, H. C., and Bacchelli, A. (2019). On the effectiveness of manual and automatic unit test generation: Ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125.

Siddiq, M. L., Santos, J. C. S., Tanvir, R. H., Ulfat, N., Rifat, F. A., and Lopes, V. C. (2024). Using large language models to generate junit tests: An empirical study.

Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S. K., and Sundaresan, N. (2021). Unit test case generation with transformers and focal context.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 4–15, New York, NY, USA. Association for Computing Machinery.

van Deursen, A., Moonen, L., van den Bergh, A., and Kok, G. (2001). Refactoring test code. *Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*.

Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., and Ahmed, I. (2022). In *PyNose: a test smell detector for python*, ASE '21, page 593–605. IEEE Press.

Xie, T. and Notkin, D. (2006). Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371.

Yetistiren, B., Ozsoy, I., and Tuzun, E. (2022). In *Assessing the quality of GitHub copilot's code generation*, PROMISE 2022, page 62–71, New York, NY, USA. Association for Computing Machinery.

Yetiştiren, B., Özsoy, I., Ayerdem, M., and Tüzün, E. (2023). In *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*.

Yu, S., Fang, C., Ling, Y., Wu, C., and Chen, Z. (2023). In *LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities*, pages 206–217.