Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter

Citation for the original published paper (version of record):
Gay, G. (2024). Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 14415 LNCS: 140-146.
http://dx.doi.org/10.1007/978-3-031-48796-5_11

(article starts on next page)

Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter

Gregory Gay[1][0000−0001−6794−9585]

Chalmers and University of Gothenburg, Gothenburg, Sweden
greg@greggay.com

Abstract. A major challenge in automated test generation is the *readability* of generated tests. Emerging large language models (LLMs) excel at language analysis and transformation tasks. We propose that improving test readability is such a task and explore the capabilities of the GPT-4 LLM in improving readability of tests generated by the Pynguin search-based generation framework. Our initial results are promising. However, there are remaining research and technical challenges.

Keywords: Automated Test Generation · Search-Based Test Generation · Readability · Large Language Models · Generative AI

1 Introduction

Software testing is a crucially important, yet labor-intensive, stage in the software development lifecycle. Automation could partially relieve the burden of test creation. Search-based test generation—the use of optimization algorithms to produce tests [2, 4]—is both efficient and effective at fault-detection [2].

However, *humans must work with generated tests* to interpret and debug program behavior. Therefore, a major challenge is the *readability* of generated tests. Generated tests are often difficult to understand—e.g., lacking documentation, incorporating uninterpretable actions, or checking trivial assertions [7]. Partial solutions have been proposed, e.g., synthesizing informative test names or documentation [3, 6]. However, comprehensive solutions do not yet exist.

Large language models (LLMs), machine learning models trained on massive corpora of text—including natural language and source code—are an emerging technology with great potential for *language analysis and transformation tasks* such as translation, summarization, and decision support [5]. We propose that improving generated test readability can be viewed as a similar task—the transformation of a test into a form with identical semantic meaning, but presented in a manner easier for human testers to interpret.

In this study, we explore the capabilities of LLMs to improve generated test readability[1]. In particular, we apply the state-of-the-art GPT-4 LLM [5], through the ChatGPT interface with the Code Interpreter plug-in, to Python unit test suites generated by the Pynguin framework [4].

[1] LLMs can generate tests, which have been found to be more readable than alternative approaches [8].We instead focus on improving readability of already-extant tests.

2 Background and Related Work

Search-Based Unit Test Generation: Test creation can be viewed as a optimization problem where, given a limited time budget, we seek tests that best meet testing goals [2]. Metaheuristic optimization algorithms sample possible test inputs to identify those that maximize or minimize fitness functions representing those goals. In this study, we focus on generation of unit tests targeting individual modules within broader systems [2, 4].
Factors Affecting Readability: Through a mapping study, Winkler et al. identified readability factors [7] including meaningful test and identifier names, enforcement of a predictable test structure, avoidance of too many or meaningless assertions, avoidance of too many dependencies and interdependent tests, understandable input values, test documentation (summaries and in-line comments), and other language properties such as consistent identifier styles. Our approach targets all of these factors, except for dependencies and test data, as those should be targeted during generation rather than refactoring.
Automated Readability Improvement: Approaches have been proposed to address the factors above. For example, test documentation has been generated using deep learning models [6] and natural language processing and code summarization techniques [3]. Our approach is the first to apply a general LLM, GPT-4, to improving readability of pre-generated tests. An advantage of a general LLM is that it can target many readability factors simultaneously. However, it could be outperformed in individual aspects by targeted tools. In future work, we will compare LLM performance to targeted solutions.

3 Approach

Our approach transforms already-generated tests as part of a series of prompts to GPT-4, a state-of-the-art LLM that has shown human-competitive performance on certain language tasks [5]. Prompts are applied manually through the ChatGPT interface[2] to take advantage of the Code Interpreter plug-in[3]—a ChatGPT plug-in that enables file uploads and execution, as well as enhanced analysis capabilities, of Python code. We target `pytest`-formatted tests generated by Pynguin [4]. However, this approach should be applicable to tests generated by other frameworks, e.g., [2], or created by humans. An example of a transformed test case is shown in Figure 1.
Enhancements to Readability: We identified potential readability enhancements to readability by examining past literature. In particular, we were influenced by Winkler et al. [7]. We also prompted GPT-4 for a definition of test readability, and drew on its suggestions. We make the following transformations:

- **Use Meaningful Test and Variable Names:** Rather than generic names like `int_0`, we apply names related to the test or variable usage context.
- **Define Variables for Constants:** Rather than using hard-coded values, we define constants at the beginning of each test with descriptive names.

[2] https://chat.openai.com/
[3] https://openai.com/blog/chatgpt-plugins#code-interpreter

```python
# BEFORE TRANSFORMATION
def test_case_38():
    str_0 = "X"
    bool_0 = module_0.is_palindrome(str_0)
    assert bool_0 is True
    assert (
        module_0.URLS_RAW_STRING == "([a-z-]+://)([a-z_\\d-]+:[a-z_\\d-]+@)?(www\\.)?" +
        "((?<!\\.)[a-z\\d]+[a-z\\d.-]+\\.[a-z]{2,6}|\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\" +
        "d{1,3}|localhost)(:\\d{2,})?(/[a-z\\d_%+-]*)*(\\.[a-z\\d_%+-]+)*" +
        "(\\?[a-z\\d_+%-=]*)?(#\\S*)?" )
    assert (
        module_0.EMAILS_RAW_STRING == "[a-zA-Z\\d._\\+\\-'`!%#$&*/=\\?\\^\\{\\}\\|~\\\\]+" +
        "@[a-z\\d-]+\\.?[a-z\\d-]+\\.[a-z]{2,4}" )
    assert len(module_0.CREDIT_CARDS) == 6
    assert len(module_0.PRETTIFY_RE) == 8

# AFTER TRANSFORMATION
def test_single_character_palindrome():
    # Test to check the is_palindrome function in the module for single character strings.
    # Setup: Define a test string with a single character
    test_string = "X"
    # Execution: Check if the string is a palindrome
    is_palindrome = module_0.is_palindrome(test_string)
    # Assertion: The string is a single character, so the function should return True
    assert is_palindrome is True
```

Fig. 1: Example of a readability transformation for a test for the `validation` module in the `python-string-utils` project.

- **Separate Test Steps into Setup, Execution, and Assertion Phases**
- **Remove Unnecessary or Redundant Assertions**
- **Code Formatting Following the PEP 8 Style Guide**[4]
- **Test Documentation:** Comments are added to explain the purpose of the test, actions taken, and code coverage achieved by the test case.

Prompt Structure: We apply the following prompts, in sequence[5]:

- *If a single module is imported by the generated test suite:* I will provide a Python module, then a series of unit tests targeting that module. First, here is the Python module. `[module-under-test]`
- *If the imported modules are in multiple files (M):*
  - I will provide M Python modules, then a series of unit tests targeting those modules. Here is the first Python module. `[module-under-test]`
  - *For each additional module:* Here is the next Python module. `[module]`
- I will now provide each unit test individually. After making the requested changes to each, incorporate each into a single test suite. The test suite should have the following import statements: `{import statements}`
- *For each test case:* I have written the following pytest test case for the previously uploaded Python file: `{test code}` Make the following changes to the test case, in the order they are listed. Use the resulting modified test

[4] https://peps.python.org/pep-0008/

[5] Square brackets ("[") indicate a file upload, while curly brackets ("{") indicate text that should be added directly to the prompt.

case following a change to make the subsequent change. The changes are, in order: add constants inside of the test case, separate setup, execution, and assertion, remove unnecessary assertions, add a meaningful test name, use meaningful variable names, format the code according to PEP 8. After making these changes, add comments to explain the purpose of the test, actions taken, and code coverage achieved by the modified test case. Add the final test case to the test suite.
– I have finished creating the test suite. Provide the test suite as a file.

Research and Technical Challenges: We developed the prompts iteratively. During this process, we encountered multiple challenges. Some we were able to partially or completely overcome, while others remain as future work.

– **Non-Determinism:** Different sessions can yield tests with potentially significant differences in readability, interpretation of the original test, and judgement on how to apply transformations (e.g., which assertions are unnecessary)[6]. We limit the scope of non-determinism by explicitly describing transformations, but this challenge remains unsolved.
– **Potential Semantic Changes to Test Cases:** The model interprets the semantic meaning of a test case. It then can change the test according to its inferences. In our observations, we saw the model remove method calls, assign calls to different objects, and even add method calls.
  Initially, we also experimented with removing redundant test steps. However, this yielded tests that, in some cases, were very different from the originals. Therefore, we removed this transformation. For the remaining transformations, we added clear descriptions and an explicit order. This limited the scope for changing semantic meaning. Still, some sessions yield larger reinterpretations of tests than others. Under the current prompt structure, code coverage seems to remain the same and the same outcomes are achieved. However, there is still potential for the semantic meaning to change.
– **Transformation Order:** The order that transformations are applied can affect results. For example, if comments are added first, they may reference elements that are removed or changed later. We account for this by applying an explicit transformation order.
– **Manual Application:** Prompts are applied manually, which can be very time consuming. Rather than applying transformations to all tests, this process could be used selectively for difficult tests.
– **Text Limits:** We initially uploaded the test suite and made transformations to all tests with a single prompt. However, GPT-4 can only process a finite quantity of text in a single prompt. We transitioned to performing one prompt per test and providing the test within the prompt rather than as part of a file upload. Some issues are still encountered with long individual tests (e.g., over 100 lines of code). We would like to explore suite-wide readability transformations (e.g., extraction of helper functions and grouping related test cases). However, such transformations are hindered by this limitation.

[6] A speculated, but currently unconfirmed, reason for performance variance is that OpenAI is deploying sub-models of GPT-4 with decreased computational cost [1].

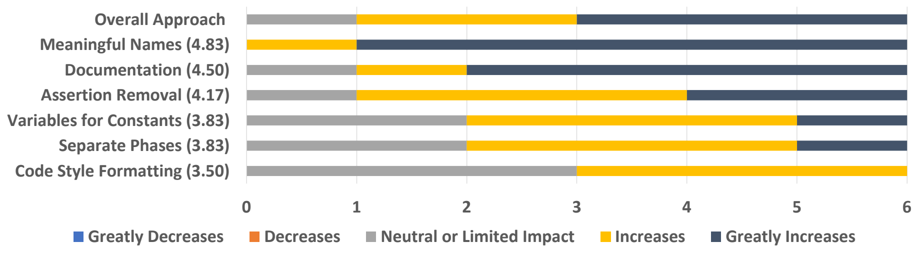

Fig. 2: Perceived impact of individual and overall readability transformations.

- **Prompt Limits:** OpenAI restricts the number of prompts in a single session[7]. If a suite contains too many tests, multiple sessions may be needed.
- **Code Interpreter Limitations:** The Code Interpreter plug-in currently only supports Python. The same transformations can be attempted on tests for other languages, but the results may be less effective. Code Interpreter also can only execute a single Python module without complex dependencies, so it cannot execute `pytest` suites currently.
- **Differences Between Screen and File Output:** The transformed tests output to the screen do not always match the versions in the file produced at the end of the session, e.g., comments may be missing.

4 Preliminary Evaluation

We make a replication package available for our study[8]. We developed and tested our approach using three case examples: a `queue`, the module `validation` from the `python-string-utils` project, and the module `sessions` from `httpie`. The latter two were selected at random from complex modules previously tested with Pynguin [4]. Generated tests were retrieved from the Pynguin documentation[9] (for `queue`) and replication package [4].

For `queue`, we used all eight tests. For the other examples, we selected 10 tests that captured a range of sizes and functionalities tested. For each test suite, we performed two transformation trials. After transformation, we manually ensured that the semantic meaning was intact. We also executed the modified suite to ensure unchanged execution results.

Three tests were broken: (1) a variable was removed but still referenced, (2) two variables had values swapped, (3) the contents of a dictionary were changed—resulting in a mismatch between result and assertion—and a method call was removed that threw an exception. Due to assertion removal, two additional tests passed when they were marked as "expected failures". **Overall, 51/56 tests (91%) were successfully transformed.**

To assess the readability improvement, each of the six transformed test suites was sent to a professional software developer (average of five years of testing experience). Only a small evaluation—six test suites, six developers—was conducted at this time to gain initial feedback. Each was asked to compare the original

[7] We encountered limits of 25 in a two hour period and 50 in a three hour period.
[8] https://doi.org/10.5281/zenodo.8296610
[9] https://pynguin.readthedocs.io/

and transformed tests and assess the overall approach and the impact of each individual transformation. The survey is included in the replication package.

The results are illustrated in Figure 2. Overall, all-but-one respondent indicated improvement in readability. Of the individual transformations, meaningful names and documentation were perceived to have the greatest impact, followed by assertion removal, variables for constants, and separation of phases. Style formatting only had a limited impact. One respondent positively noted that—after transformation—the tests could survive code review.

However, the transformations "improved a horrible test suite"—as stated by one respondent. Respondents explained that the original tests tested too many behaviors simultaneously and lacked clear rationale for the assertions included[10]. One respondents indicated that they would still need to make improvements manually—although the transformations made that improvement possible. The potential for readability transformation seems to be limited by the quality of the original tests. Respondents suggested that there would be value in applying these transformations to human-written tests.

5 Conclusions

Our results indicate that LLMs are a promising method of improving generated test readability. However, there are also significant research and technical challenges. In future work, we will further develop these initial ideas, explore automation, and conduct a full evaluation.

Acknowledgements: Vetenskapsrådet Grant 2019-05275.

References

1. L. Chen, M. Zaharia, and J. Zou. How is chatgpt's behavior changing over time?, 2023. arXiv preprint 2307.09009.
2. A. Fontes, G. Gay, F. G. de Oliveira Neto, and R. Feldt. *Automated Support for Unit Test Generation*, pages 179–219. Springer Nature Singapore, Singapore, 2023.
3. B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft. Automatically documenting unit test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352, 2016.
4. S. Lukasczyk, F. Kroiß, and G. Fraser. An empirical study of automated unit test generation for python. *Empirical Software Engineering*, 28(2), 2023.
5. OpenAI. Gpt-4 technical report, 2023. arXiv preprint 2303.08774.
6. D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 287–298, New York, NY, USA, 2021. Association for Computing Machinery.
7. D. Winkler, P. Urbanke, and R. Ramler. What do we know about readability of test code? - a systematic mapping study. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1167–1174, 2022.
8. Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2023. arXiv preprint 2305.04207.

[10] One respondent mistrusted the assertion removal, as it was not clear why some assertions were present in the first place.