Using LLM-Generated Summarizations to Improve the Understandability of Generated Unit Tests

Enhancing Unit Test Understandability: An Evaluation of LLM-Generated Summaries

Natanael Djajadi[1]

Supervisor(s): Andy Zaidman[1]**, Amirhossein Deljouyi**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Natanael Djajadi
Final project course: CSE3000 Research Project
Thesis committee: Andy Zaidman, Amirhossein Deljouyi, Asterios Katsifodimos
DOI: 10.5281/zenodo.12510836

Abstract

Since software testing is crucial, there has been research on generating test cases automatically. The problem is that the generated test cases can be hard to understand. Multiple factors play a role in understandability and one of them is test summarization, which provides an overview of the test of what it is exactly testing and sometimes highlights the key functionalities. There already exist some tools to generate test summaries that use template-based summarization techniques. Limitations of generated summaries include that they can be lengthy and redundant, and that it is best to use them in combination with well-defined test names and variables. There is a tool developed named `UTGen`, which combines Evosuite and Large Language models to increase understandability which includes improving the test names and variables, but does not have a summarization functionality yet. In this research, we extend `UTGen` using LLM-generated summaries.

We investigate to what extent the understandability of a test case can be influenced by LLM-generated test summaries in terms of context, conciseness, and naturalness. For this reason, we do a user evaluation with 11 participants with a software testing background. They will judge LLM-generated summaries and compare them to existing summarization tools. The LLM-generated summaries scored overall higher than the template-based summaries and were also more favorable by the participants.

1 Introduction

Software is used everywhere around us. Examples are registering participants for an event, or doing online transactions. This means that the reliability and accuracy of software are important which can be verified through testing, but it can be very tedious and time-consuming. This is why there is already a lot of research into generating those test cases automatically, of which `EvoSuite` is an example [6]. With `EvoSuite` you can generate test cases that achieve a high code coverage, but it can be hard to understand because of the names of the variables for example. This can be seen in Listing 1, where it uses names that are not descriptive like `rational1`. In an interview study [9], the participants agreed that when there are large test cases that maximize code coverage, they are hard to understand and maintain. A systematic mapping study from 2024 by Winkler et al. [21] identified factors that play a role in understanding test cases, which include test names, identifier names, comments, test summaries, and some other aspects.

Our research focuses specifically on elevating the understandability of test cases by using test summaries generated with a large language model (LLM). In a survey about automatically documenting unit test cases, almost all the participants agreed that the quality of the system, good documentation of the test cases, and maintaining those tests are important [11]. Nevertheless, some limitations of generated summaries include that they can be lengthy and redundant, and that it is best to use them in combination with well-defined test names and variables [18]. There already exists some research on test summaries. A developed tool is `TestDescriber` [16], which also takes into account the part of the code that is executed. This leads to including code coverage in the test summary, next to only using static code to generate the summaries. One of the main aspects of that study was that they looked at the impact of summaries on bug-fixing. These summaries improved the comprehensibility of the generated test cases for developers, resulting in them finding twice as many bugs than without this tool. Another research was about `DeepTC-Enhancer` [18], which uses a two-stage approach. It first uses a template-based approach to generate high-level summaries, and then all the identifiers will be renamed by deep learning to increase understandability. One of the results of this research was that the participants preferred `DeepTC-Enhancer` over `TestDescriber`.

```
@Test
public void test3() throws Throwable {
    Rational rational0 = new Rational(1L, 3215L);
    Rational rational1 = rational0.abs();
    assertEquals(1L, rational0.numerator);
    assertEquals(3215L, rational0.denominator);
    assertEquals(3.11041E-4F, rational1.floatValue(),
    0.01F);
}
```

Listing 1: Example of a generated test of Evosuite [16])

In recent years, there has been a significant advancement in LLM technology. As such, more and more people are using LLMs for their professional workflows and everyday activities. Large language models themselves can be used for a lot of purposes in the context of coding, for example explaining code for students to understand the code more effective [10], or generating tests [4, 22]. LLMs can understand the question that was prompted and the context, and generate output in natural language, which would be relevant to our research. In our case, we can ask the LLM to generate a summary for a given test. `TestDescriber` and `DeepTC-Enhancer` both do not use LLMs in their tools. A fairly new tool developed named `UTGen` combines Evosuite with LLMs to increase the understandability [5]. `UTGen` enhances the understandability of generated tests by improving the test names and identifiers names, contextualizing test data, and adding explanatory comments.

Our research extends `UTgen` using LLM-generated summaries, which are not (yet) included in this tool. It is interesting to see the impact of summaries generated by LLMs and how that would differ from template-based summaries. We also look at the difference in output between a local and a commercial LLM. Our study has the main question: **to what extent can the understandability of a test case be influenced by Large Language Model-generated test summaries in terms of context, conciseness, and naturalness?**

This main question is broken down into three sub-questions:

RQ1 *What is the impact on the understandability of a test case by using LLM-generated test summaries using prompt engineering, few-shot, or context-aware summarization?*
Since prompt engineering, few-shot, and context-aware summarization are different techniques to generate a prompt, we want to see how that would differ from only asking the LLM plainly to generate a test.

RQ2 *To what extent can the understandability of a test case be influenced using Large Language Model generated test summaries* ***in contrast to existing tools like TestDescriber and DeepTC-Enhancer*** *in terms of context, conciseness, and naturalness?*
We want to compare the understandability of LLM-generated summaries with existing tools by looking at the context (if the content is complete and correct), conciseness (if there is no irrelevant information), and naturalness (how natural the language that was generated is). This we do with a user evaluation.

RQ3 *What are the differences in characteristic elements between LLM-generated test summaries, TestDescriber, and DeepTC-Enhancer that influence test case understandability?*
Characteristic elements are individual parts of the summary that contribute to its overall understandability, like the length of the summary, and using a numbered list. By answering this question we can explain how these differences impact the understandability of LLM-generated summaries in contrast to the existing tools which was answered in **RQ2**.

We summarize the core contributions of our work as follows:

- We extended UTGen with LLM-generated summaries with prompt engineering and/or few-shot.
- A user evaluation was conducted with 11 participants who all had experience in software testing to judge the summaries on their understandability.
- The results show that LLM-generated summaries scored higher and were preferred over template-based summaries.

2 Background

In this section, we will go through the background in more detail. First, we look at template-based summaries that existing tools use. Then we are going through two prompting techniques which are used before.

2.1 Template-based Summaries

As earlier stated, `TestDescriber` [16] and `DeepTC-Enhancer` [18] both use template-based summaries. This means that they are using pre-defined templates of natural language sentences. Then they fill it with the output of SWUM, which is a technique that uses nouns, verbs, and prepositional phrases to represent program statements [12]. TestDescriber generates three types of summaries: a class summary, a method summary, and a fine-grained statement summary. `DeepTC-Enhancer` on the other hand, focuses on its test summarization mostly to filter out redundant information. It analyzes all the statements, reduces the redundant steps which are for example multiple lines of the same operations, and aggregates the remaining statements.

2.2 Prompting Techniques

Prompt Engineering

Given that each model has its own favored input structures, complexities, and drawbacks, there is not a universal solution that fits all scenarios in prompt engineering. As in the paper of `UTGen` [5], they use guidelines to prompt the LLM. Action words, adopting a persona, and using Chain of Thought are all techniques that are used. A study by Wei et al. [20] shows that using Chain-of-Thought prompting improves the performance of LLM for complex reasoning. Using this technique, in the prompt there will be a series of intermediate reasoning steps present.

Few-shot Summaries

Few-shot is another technique that can be used to prompt the LLM. Here, one will first give a few examples or demonstrations similar to the task that will be performed, and then prompt the LLM to do the task. An existing work, which is called `Cedar`, uses a strategy to automatically retrieve the demonstration selection [14]. They show in their study that the method they developed surpasses the most advanced existing models of assertion and program repair, showing improvements of up to 11% and 5% respectively.

3 LLM-generated Summarizations

3.1 Prompting Techniques Used for Summarizations

We created four prompts using different prompting techniques to generate summaries with the LLM.

Simple/Baseline

With this prompt, we asked the LLM to simply generate a summary with the test method provided. This will be our baseline to see if using different prompting techniques will affect the output of the LLM. Tags like `[CODE]` and `[/CODE]` will be used to make it clear for the LLM where the code starts and ends. The prompt used is shown in Listing 2.

```
Please generate a summary of the unit test given between the
[CODE] and [/CODE] tags. Please put your answer between
[SUMMARY] and [/SUMMARY] tags.

[CODE]
<Insert Unit Test>
[/CODE]
```

Listing 2: Simple prompt which serves as the baseline

Prompt Engineering

As earlier mentioned, there is no one-size-fits-all solution for prompt engineering. Since we are using the LLM `Codellama:7b-instruct` which was also used in `UTGen` [5] and build on that tool, we can take inspiration from how they engineered their prompt. We are aware that the goals of `UTGen` are different since it prompts to improve aspects like the test name and variables, and not summarizations. Still, as they argue in their paper, prompt engineering can set guidelines for the LLM for the output generated. This can be useful for our output since we want the summaries to concisely explain what the test does in a readable way.

Since using Chain of Thought improves the performance of LLMs for complex reasoning [20], we will be also using the Chain-of-Thought prompting. In step 4 as seen in Listing 3, we ask the LLM to make the summary as succinct and concise as possible in a bullet point format. It is not a significant concern when the output is not in a bullet point format, which sometimes happens with `codellama:7b-instruct`.

This is because we implement prompt healing, inspired by Dagan et al.'s token healing technique [3]. While their approach fixes issues in the generated text by reducing bias in the output, we use prompt healing to fix layout and information issues in our generated summaries. Layout issues include not using the bullet point format we asked for in our prompt and summaries in code blocks. Irrelevant information refers to auto-generated introductions like "*Here is a suggested test summary for the provided Java code:*". Prompt healing omits irrelevant parts and adds newlines for non-bulleted output.

```
[INST] As a detail-oriented developer focused on enhancing the
clarity of a test suite, your task is to analyze the provided
Java code and generate a descriptive and concise summary of
the unit test. Follow these
steps:
   1. Carefully read the Java code between the [CODE] tags.
   2. Identify the primary functionality or purpose of the test.
   3. Analyze actions performed in the test.
   4. Formulate a test method summary that succinctly and
   concisely captures what the test case does in a bullet point
   format for easy, quick reading
   5. Place your complete suggested test summary between the
   [TESTSUMMARY] and [/TESTSUMMARY]  tags, ensuring it is clear
   and precise without unnecessary descriptions or information.
   Remember, your focus is on clarity and precision. Use your
   expertise to provide a meaningful and appropriate summary.
[/INST]
```

Listing 3: The instructions part for prompt engineering

Context-Awareness

For context-awareness, inspiration was taken from `TestDescriber` [16]. `TestDescriber` takes the statements and branches being tested into account to include the code coverage in the test summary. In our case, next to the test method which was already present in our simple prompt, we also include the method that is being tested. We hypothesize that the method that is being tested will be explained in some way in the summary. `The method under test is between the [METHOD_TESTED] AND [/METHOD_TESTED] tags.`

Few-shot

In our few-shot prompt, we use demonstrations from summaries that developers in the industry have written. After going through open-source repositories on GitHub, the projects from `Spring` [17], `CoreNLP` [15], and `Dagger` [7] were chosen with the classes Spr8510Tests.java, StanfordCoreNLPServerITest.java, and LongCycleTest.java respectively. These summaries were chosen because they were summaries for unit tests, and were short, concise, and clear. A summary generated with few-shot is shown in Listing 4. The full prompt including the demonstrations is in the replication package on GitHub[1].

3.2 Large Language Models

Initially, the plan was that we would only use `codellama:7b-instruct` [1] as the LLM, since `UTGen` already uses this model. This model uses 7 billion parameters. But since we are running this locally and not with as many parameters as `GPT3.5`, which has an estimated parameter count of 175 billion, we hypothesize that `codellama:7b-instruct` performs worse. We have found during testing different prompts that the generated summaries do not always follow the instructions given. It can be mostly seen during few-shot testing, where it would generate very long summaries, while the demonstrations given were only one line long. For this reason, we tried both `codellama:7b-instruct` and `GPT3.5` and `GPT4o` to generate summaries.

[1] https://github.com/NaelDj/UTGen-Summarization-Extention

```
/**
* Tests the absolute value function of the Rational class to
* ensure it correctly handles positive values and returns the
* expected float representation.
*/
@Test
public void test3() throws Throwable {
    Rational rational0 = new Rational(1L, 3215L);
    Rational rational1 = rational0.abs();
    assertEquals(1L, rational0.numerator);
    assertEquals(3215L, rational0.denominator);
    assertEquals(3.11041E-4F, rational1.floatValue(),
    0.01F);
}
```

Listing 4: Summary generated with few-shot using the LLM `ChatGPT` from the `Rational` class [16]

3.3 Implementing in UTGen

There were two places where we had to alter the code to implement the summary functionality in `UTGen`. The first place is in the `UTGen` code itself, where summaries would be added on top of the tests. The other place was in the LLM server. Here the prompt was located and would generate the summaries by connecting it to the LLM. The model will be connected either locally with Docker for `codellama:7b-instruct` or via the official API of `GPT3.5`.

Prompt healing was also implemented which was described earlier as being inspired by token healing [3]. In the LLM server, it would then omit unnecessary parts out of the summary generated like "*Here is your generated test:*" and when the LLM puts the summary in a code block, prompt healing will extract the summary out of the code block. In `UTGen`, every sentence of the summary will start on a new line. This is to prevent very long summaries without any newlines, where the user would have to scroll horizontally to read the summary.

4 Experimental Setup

In this section, the methodology of our research will be described. We will investigate the following research questions:

RQ1 *What is the impact on the understandability of a test case by using LLM-generated test summaries using prompt engineering, few-shot, or context-aware summarization?*

RQ2 *To what extent can the understandability of a test case be influenced using Large Language Model generated test summaries* ***in contrast to existing tools like TestDescriber and DeepTC-Enhancer*** *in terms of context, conciseness, and naturalness?*

RQ3 *What are the differences in characteristic elements between LLM-generated test summaries, TestDescriber, and DeepTC-Enhancer that influence test case understandability?*

4.1 Set-up for user-evaluation (RQ1)

We will explore the understandability of the summaries in terms of **context**, **conciseness**, and **naturalness** using two test methods that were included in the `DeepTC-Enhancer` [18] external developer's surveys. `TestDescriber` [16] and `DeepTC-Enhancer` [18] have both already generated summaries for these tests. By generating summaries with the LLMs on these summaries, we could then already compare template-based summarizations with LLM-generated summarizations before the user evaluation. This is because we do not use a pilot.

We use three different LLMs: `codellama7b-instruct`, `GPT3.5`, and `GPT4o`. An LLM is non-deterministic, which means that the same prompt can produce different results. Therefore, it was important to run it multiple times on the same prompt. Since we have three different LLMs, two tests to generate summaries on, four prompt techniques, and we run each prompt three times because of the non-deterministic nature, we will then have 72 prompts in total to analyze.

These prompts will be analyzed by ourselves because due to this project's time limitations, it is not possible to do two user evaluations. This can be done for further research. For now, these 72 prompts will be analyzed using two types of metrics: length of output and understandability. Understandability, which includes judging the context, conciseness, and naturalness, is very subjective. We chose to analyze understandability for this research question and RQ2 in terms of context and conciseness, because of similar work done before [13, 16, 18]. Context is about if the content is complete and correct. Conciseness is about if there is no irrelevant information. Those works also had a third aspect, readability. However, since our research is about large language models, it would be more interesting to see how natural the language is of the generated summaries because these models are designed to understand and generate human-like text. Rules were implemented to make this evaluation less subjective. A specific summary starts with 5 points on a Likert scale. It can have points deducted if it is wrong, like missing information or containing unnecessary information. It can stack up, for example when in two separate places in the summary it contains unnecessary information.

The rules for context are: 1) It should include what it is testing, and how it is testing. If either of these is missing, there will be a point deduction. 2) If any information is missing or 3) if any information is incorrect, there will also be a point deduction. There is only one rule for conciseness: if the summary contains unnecessary information, there will be a point deduction. For naturalness, there will be a point deduction if: 1) brackets or quotes are used unnecessarily, or 2) the flow and tone are not appropriate. Flow refers to the logical progression of ideas and how well sentences connect with each other. Tone refers to maintaining a formal, objective style with proper grammar and punctuation.

4.2 User Study (RQ2)

We conduct a controlled experiment to answer the research question of comparing LLM-generated tests with existing tools. Participants were recruited via our network and in total there were 11 participants. They were given a survey and judge summaries.

Experience	Industry	Academic
0-1 years	6 (60%)	3 (30%)
2-3 years	3 (30%)	1 (10%)
4-5 years	2 (20%)	7 (70%)
Total	11 (100%)	11 (100%)

Table 1: Experience of Participants

1) Participants: We recruited participants with software testing experience. Eight of them were Bachelor students, and three doing Masters. They were volunteers and there was no reward for participation in this research. Their experience in Java in the industry (practical) and academic are shown in Table 1. All the participants had at least one year of experience in Java academically.

2) Survey: The survey starts with an informed consent. After that, the survey consists of two parts with a total of 20 questions. The first part is about their background. The second part is about judging summaries on their understandability. This part consists of four rounds, which means that four different summaries were judged and compared by the participants. The first two summaries were chosen based on the external surveys from `DeepTC-Enhancer` [18], and the last two were chosen based on the replication packages that was on GitHub of `TestDescriber` [16] and `DeepTC-Enhancer`, such that we did not have to run those tools. We made sure to minimize bias by not ordering the summaries based on their tools in the same order but randomized the summaries generated by the tools in each round. For judging the understandability of the test cases in terms of context, conciseness, and naturalness, the participants used a 5-point Likert scale. There were also optional text boxes included after each question where participants could elaborate on their answers.

We choose the LLM-generated summaries for the survey based on our results from RQ1. After answering the previous research question, we can choose the summaries we think are best suited for the survey from `codellama:7b-instruct` and `ChatGPT`. In the first two rounds, they will have to compare all four tools, and in the last two rounds, they compare the two LLM-generated summaries with either `DeepTC-Enhancer` or `TestDescriber`, thus comparing three summaries. This survey is also published in the replication package on GitHub.

3) Experimental Procedure: To minimize bias even more, we also were present during the user evaluation. When participants had questions, we would be able to elaborate. At the same time, we could make sure they filled it in seriously and thoughtfully and did not just fill it out without consideration. This approach helps us avoid the uncertainty that comes with online surveys, where it is harder to verify if they have not chosen the options at random.

4) Analysis Method: We use the Kruskal-Wallis H test using a significance level of $\alpha = 0.05$, which can be applied to see if there are significant statistical differences for more than two groups. We chose the Kruskal-Wallis H test since the Shapiro-Wilk test shows that our results were not normally distributed. We will also use the Dunn's test which serves as a post-hoc test, to see pairwise which tools have statistical differences. To measure the effect size, we use Cohen's d to measure the effect size, categorizing it as negligible

```
/**
 * Creates a new instance of the "Rational" class with numerator 1
 * and denominator 3215.
 * Calls the "abs()" method on the created rational number, which returns a
 * new rational number with the absolute value of the original number.
 * Asserts that the new rational number has a numerator equal to 1.
 * Asserts that the new rational number has a denominator equal to 3215.
 * Converts the new rational number to a float value and asserts that it is
 * approximately equal to 3.11041E-4F, with a tolerance of 0.01F.
 */
@Test
public void test3() throws Throwable {
    Rational rational0 = new Rational(1L, 3215L);
    Rational rational1 = rational0.abs();
    assertEquals(1L, rational0.numerator);
    assertEquals(3215L, rational0.denominator);
    assertEquals(3.11041E-4F, rational1.floatValue(),
    0.01F);
}
```

Listing 5: Summary generated with prompt engineering using `codellama:7b-instruct` from the `Rational` class [16])

Prompt Technique	LLM	Avg Sentences	Avg Words	Words/Sentence
Simple	Codellama:7b-instruct	6,33	124,33	19,63
Simple	GPT3.5	3,83	67,3	17,57
Simple	GPT4o	4,67	75,5	16,18
Context-Awareness	Codellama:7b-instruct	4,83	87	18
Context-Awareness	GPT3.5	5	88,83	17,77
Context-Awareness	GPT4o	4,83	91	18,83
Prompt Engineering	Codellama:7b-instruct	4,67	81,33	17,43
Prompt Engineering	GPT3.5	5,5	56,33	10,24
Prompt Engineering	GPT4o	5	55,5	11,1
Few-shot	Codellama:7b-instruct	7,17	148,5	20,72
Few-shot	GPT3.5	1,5	32,5	21,67
Few-shot	GPT4o	1	20,5	20,5

Table 2: Comparison of length of LLMs across different techniques

($d < 0.2$), small ($0.2 \leq d < 0.5$), medium ($0.5 \leq d < 0.8$), and large ($d \geq 0.8$) [19].

4.3 Elements to explain the results (RQ3)

This last research question will be answered in two ways. The first way was by finding the characteristic elements ourselves. When we refer to characteristic elements, we are referring to individual parts of the summary that contribute to its overall understandability, like the length of the summary, and using a numbered list. To find the elements ourselves, we study summaries generated with the existing tools and with the LLMs first. Then, we identified common elements, for example using a numbered list. After this, we tested these elements on other summaries to see if the elements served as characteristic aspects that could contribute to the understandability. Lastly, we refined the list of elements, for example by splitting elements into more specific ones. These were given as examples in the survey to the participants, but they were encouraged to come up with their own. This is the second way to come up with aspects. Participants wrote aspects they liked the most and the least in two text boxes. By having this distinction we can also immediately see their preferences.

5 Results

The research questions will be answered one by one.

5.1 RQ1: *Impact on understandability by using different prompt techniques*

The impact on understandability using different prompt techniques depends on the LLM used. We will go through each aspect of understandability (context, conciseness, and naturalness) to see the differences between the techniques.

Context

As seen from Figure 1, few-shot has a lower score compared to the other prompting techniques simple prompt, context-awareness, and prompt engineering. This is because the summaries generated by using few-shot are usually one-liners that explain the main idea of the test, but they do not explain how it works. The other prompting techniques go through the code explaining almost each step. Since it is missing this information, by using our predefined rules few-shot has a score of around 4.

Other prompting techniques next to few-shot had on average a complete summary without much incorrect information. However, `codellama:7b-instruct` for context-awareness made on average more mistakes than `ChatGPT`, or the other techniques simple prompt or prompt engineering. This can be seen in Figure 1. We hypothesized that by using context-awareness, the method that is being tested would also be explained in the summary. However, we found out that `codellama:7b-instruct` never explained the method being tested in the summary for the class `ArrayIntList`. `ChatGPT` explained it only 1 out of the 6 times. For the class `Rational` the LLMs explained the method always. Surprisingly, summaries generated by simple prompt and prompt engineering also explained the method for the `Rational` class four of the six times using `codellama:7b-instruct`. The prompt's content, whether it includes or excludes the tested method, appears to have minimal effect on the results.

Conciseness

As for conciseness, `codellama:7b-instruct` scored the worst using few-shot and second-worst using the simple prompt. As seen from Table 2, `codellama:7b-instruct` uses the most amount of sentences and words on average using these two prompting techniques. This is because for few-shot, `codellama:7b-instruct` does not use the demonstrations given and explains the code mostly with multiple paragraphs. However, `ChatGPT` used the demonstrations given which were one-liners, outputting very concise summaries. This is why it then scored on average the highest compared to the other prompting techniques.

It is worth mentioning that the LLM using prompt engineering generates summaries with bullet points in contrast to simple prompt and context awareness, which explain the summaries in paragraphs. The downside of using bullet points is that they are more prone to containing repetitive lines of the same operations. This can be seen in Listing 5, where it has multiple lines of assertions. This explains why conciseness has a score of 4 using `codellama:7b-instruct`.

Using context-awareness leads also to a drop for `ChatGPT` in terms of conciseness, because explains a lot more of the test which is sometimes not necessary. Table 2 shows that `ChatGPT` uses more words on average in contrast to simple prompt.

Naturalness

Naturalness for `codellama:7b-instruct` using few-shot and simple prompt has a very low rating in Figure 1. For few-shot, different aspects of prompts lowered the naturalness. Examples are using a non-appropriate tone using “we”, using a lot of the exact number 3.11041E-4, or the flow was sometimes disturbed by brackets which could have been omitted.

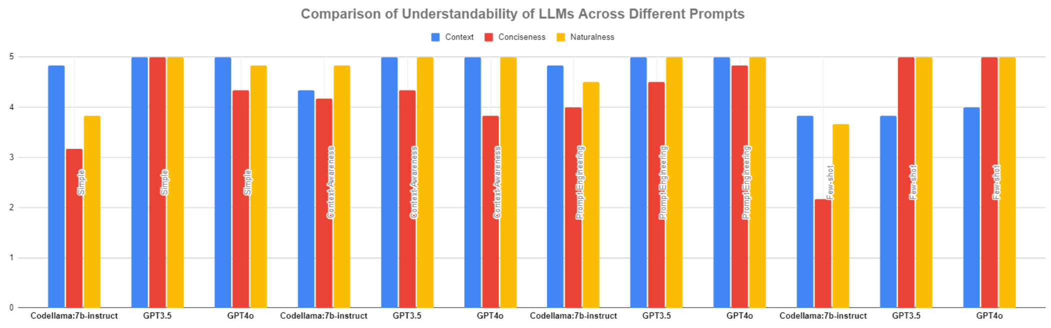

Figure 1: Comparison of the understandability of LLMs across different prompt techniques

As for the simple prompt, there were also instances where the flow was disturbed by unnecessary uses of brackets and unnecessary use of single quotes.

5.2 RQ2: *Comparative influence of LLM-generated summaries and existing tools*

The results from comparing the summaries by the participants are shown in Figure 2. Because `TestDescriber` and `DeepTC-Enhancer` are not compared to each other but only to the LLMs in rounds 3 and 4 of the survey, they are also separately compared in our results. The results indicate that 1) for context `codellama:7b-instruct` scores the highest, 2) for conciseness that `ChatGPT` scores the highest, and 3) for naturalness LLM-generated summaries were ranked higher than the template-based summaries. The average results of the aspects of understandability can be seen in Table 3. From this table, we can see that the score for the naturalness of template-based summaries is below 4, and LLM-generated summaries are greater than 4. In Table 4 we can see as average for the understandability that LLM-generated summaries overall scored higher than the template-based summaries if we assume the aspects are all weighted the same.

From a statistical perspective of comparing `TestDescriber` with the LLMs, the participants found the LLM-generated summaries `codellama:7b-instruct` and `ChatGPT` as significantly far more concise than `TestDescriber` using Dunn's test ($p = 0.01$ and $p \leq 0.01$, respectively). The effect size is large in these cases, being 0.92 and 1.41 respectively. As for naturalness between `TestDescriber` and `ChatGPT`, the participants perceived ChatGPT as significantly more natural with a p-value of 0.04 with a medium effect size of 0.60. There is no significant difference between `TestDescriber` and the LLMs for the aspect context as the result from the Kruskal-Wallis H test shows us ($p = 0.08$).

When comparing `DeepTC-Enhancer` with the LLMs, `DeepTC-Enhancer` surpasses `codellama` in terms of conciseness with a significant difference ($p = 0.03$) and a small effect size of 0.46. There are no significant differences between `DeepTC-Enhancer`'s and the LLM-generated summaries regarding the naturalness aspect ($p = 0.35$). However, between the LLM-generated summaries `ChatGPT` was perceived as more concise than `codellama` with a significant difference of $p \leq 0.01$ and a medium effect size of 0.77. Furthermore, participants ranked `codellama` significantly higher than `ChatGPT` with a difference of $p \leq 0.01$ for the context aspect, and having a large effect size of 0.94.

5.3 RQ3: *Differences in test summary elements influencing understandability*

We will first show the liked and disliked elements of each tool. The feedback from the participants was aggregated across the four test methods of the survey. The numbers mentioned represent the total time an element was liked or disliked across all tests.

`TestDescriber`

The only aspect more than one participant liked was that `TestDescriber` included comments on all main aspects in a step-by-step guide, which was mentioned four times in total by two participants. However, in general the participants disliked many elements, including that they found the summaries long and containing unnecessary information. The worst-rated aspect of this tool was the inline comments, which they say make the test more unreadable as it is too detailed. Seven participants mentioned this twelve times across the tests, and can be seen in Figure 2 where for conciseness `TestDescriber` scored low.

`DeepTC-Enhancer`

Participants liked a few aspects of `DeepTC-Enhancer`, which included a good summarization of the flow of the test what it does, and that it is concise. Five participants brought seven times the numbered list up, which has a step-by-step explanation of the test. But a few of them did not like that it describes what is happening on each line of code, which did not help to understand the test's purpose and therefore had lower scores than `codellama:7b-instruct` where people were more positive about the context which will be described in the next subsection.

`Codellama:7b-instruct`

Eight participants mentioned fourteen times across the tests that they liked the detailed description generated by `codellama:7b-instruct`. It was worded in different ways, like having a good amount of detail, an in-depth explanation, and being clear. Also, they liked the structured format, which makes it very easy to skim through. This came up five times by individual participants. As such, `codellama:7b-instruct` received high scores for the context. However, five participants raised seven times that the summary is long and contains information that is already in the test. Like using `DeepTC-Enhancer`,

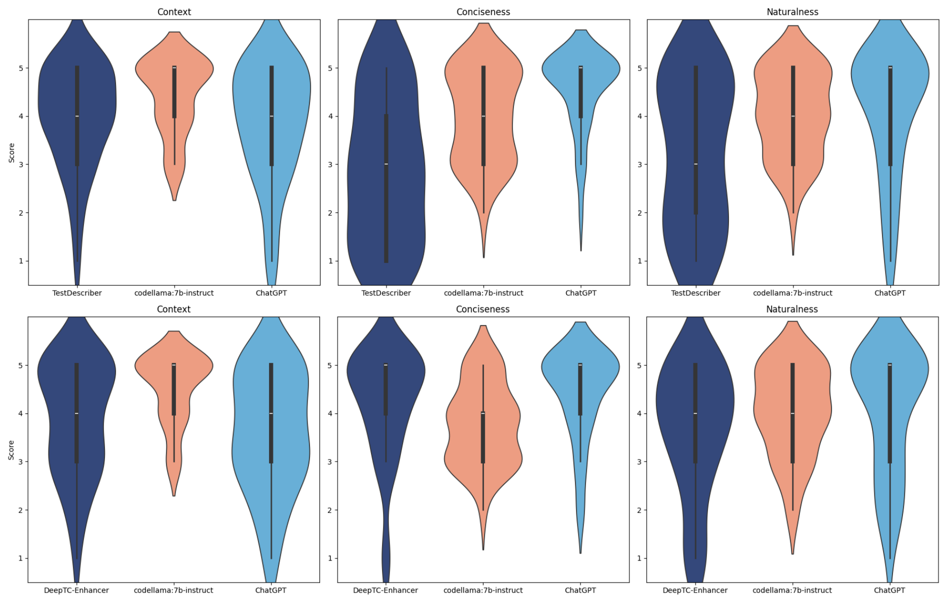

Figure 2: Results from the comparison for the summaries of `TestDescriber`, `DeepTC-Enhancer`, `codellama:7b-instruct`, and `ChatGPT` using a 5-point Likert scale

it explains in words what is happening on each line of code, which participants said they could read themselves. But it was raised that sometimes the summary looks like `DeepTC-Enhancer`, but longer. This explains why the conciseness of `codellama:7b-instruct` is lower than `DeepTC-Enhancer` and `ChatGPT`.

`ChatGPT`

Five participants stated six times that they liked the concise summarization. A slightly different group of five participants said eight times that they liked how the summary directly addresses the test's objective and explains the idea behind the test. This can be seen in Figure 2 where many participants scored `ChatGPT` high for conciseness. There was only one aspect that they did not appreciate at all and therefore was brought up seventeen times by seven participants. This aspect was that `ChatGPT` was explaining too little, which they say makes it less practical, very basic and limited, and even cryptic. This is the reason why for `ChatGPT` the context aspect scored lower than the other three tools on average.

Elements influencing preferences

As shown in Table 4, `codellama:7b-instruct` was the most preferred summary among the participants, in second place `ChatGPT`. `Codellama:7b-instruct` scored higher than `ChatGPT` for context, but it is the other way around for conciseness. This can be seen in Table 3 where the individual aspects of understandability are seen. From this, we see a possible indication that participants find context more important than conciseness. As the participants have brought up, they really liked the detailed descriptions and structured format of `codellama:7b-instruct` and disliked that `ChatGPT` was very basic and limited. A long summary with unnecessary information and in-line comments also has a lot of impact as seen in Table 4, where `TestDescriber` was favored the least.

Tool	**Context**	**Conciseness**	**Naturalness**
`TestDescriber`	3.97	2.85	3.21
`DeepTC`	3.88	4.21	3.64
`Codellama:7b-instruct`	4.48	3.93	4.09
`ChatGPT`	3.78	4.57	4.05

Table 3: Average results of the aspects (context, conciseness, naturalness) of understandability of the LLM-generated summaries

Tool	**# times favored**	**Understandability**
`TestDescriber`	7	3.34
`DeepTC-Enhancer`	12	3.09
`ChatGPT`	17	4.17
`codellama:7b-instruct`	20	4.13

Table 4: Comparison of the number of times participants preferred to use a summary generated by the tool and the average score of understandability

6 Discussion

In this section, we will discuss our results, and the potential threats to validity in terms of construct, internal, and external validity.

Revising the Research Questions

RQ1: *What is the impact on the understandability of a test case by using LLM-generated test summaries using prompt engineering, few-shot, or context-aware summarization?* We have seen that using the prompting technique context-awareness did not have a lot of impact on the summary in contrast to other techniques like the simple prompt or prompt engineering. This is because the LLM did not include the method under test, or other techniques also already explained the method being tested. To see if this is really the case, it would be good to include the method being tested for other different tests and methods, since two may not be enough to see the actual difference. Furthermore, for future research it would be better to do a user evaluation to answer this research question, which will be further explained in *Construct Validity*.

RQ2: *To what extent can the understandability of a test case be influenced using Large Language Model generated test summaries* ***in contrast to existing tools like TestDescriber and DeepTC-Enhancer*** *in terms of context, conciseness, and naturalness?* `Codellama:7b-instruct` and `ChatGPT` were rated differently in the last two tests in the survey, which is reflected in the difference in the rows of the plots in Figure 2. An example can be seen when looking at the context aspect for `ChatGPT`. When comparing `ChatGPT` to `DeepTC-Enhancer`, participants scored `ChatGPT` lower than comparing it to `TestDescriber`. We hypothesize that the difference in context for `ChatGPT` was because the third test method in the survey was testing `buildFromEncodedMap` of the `KeycloackUriBuilder` class, which could have been more difficult to understand than the other tests in the survey. Since the summary of `ChatGPT` was very concise, it did not tell the participants much about how it is testing the method which leads to a drop in understandability of the context. The difference of rating in conciseness of `codellama:7b-instruct` may be because of its individual comparison to either `TestDescriber` or `DeepTC-Enhancer`. In round three of the survey, the summary of `codellama:7b-instruct` was longer but similar to `DeepTC-Enhancer`'s summary. In round four against `TestDescriber`, participants liked the steps more of `codellama:7b-instruct`, which can be seen in Figure 2, where it scored higher for conciseness.

RQ3: *What are the differences in characteristic elements between LLM-generated test summaries, TestDescriber, and DeepTC-Enhancer that influence test case understandability?* For this research question, we saw many aspects that can influence understandability in either a positive, a negative way, or even both. An example of both is when a summary is very concise like the summaries generated by `ChatGPT`. On the one hand, it is very concise which means it gets straight to the point of the test. On the other hand, participants say it is less practical since there is limited information and they have to go through the whole test themselves. However, since there are different aspects present in the same summary, it is not possible to conclude that one specific aspect is better than the other in our research. An example of this would be that `TestDescriber` was rated the worst in conciseness but has different aspects that could play a role in conciseness, which are the inline comments, the length of the summary, and the amount of unnecessary information.

Threats to Validity

Construct Validity is about how we set up our study. During the user evaluation, we were present when they were filling in the survey. However, in the first part of our study, we judged the summaries on our own which can create a bias. This we tried to mitigate by implementing rules to be more consistent. Even then, a long summary has a bigger chance of having irrelevant information than a shorter test, so the rules may still not fully mitigate this problem. Because of the time constraints of this research, another user evaluation was not possible but would be good to do as future research.

Internal Validity. In the survey, the tools that generated the summaries were not revealed. Across the several rounds, the summaries were randomized, such that summary A in round 1 does not necessarily correlate to summary A in round 2. This we have done to avoid a preference of the participants for one tool over the other. However, since we had four rounds, participants may have noticed a pattern throughout the rounds of similarities between the summaries.

Next to that, to determine if the experience of the participants (practical, academic, or combined) influenced how they ranked the aspects context, conciseness, and naturalness, we conducted a permutation test. The results indicated that no significant difference based on experience, suggesting our findings were robust and not influenced by the participants' backgrounds (for all $p \geq 0.05$).

External Validity. There is a threat to the generalizability of our results. To answer RQ1, we have used two classes that were used in the DeepTC-Enhancer paper [18]. To limit generalizing our findings of RQ2 and RQ3, we use four different tests in the user evaluation. With this, we made sure that we do not conclude results from only one or two test cases.

7 Conclusions and Future Work

Research has shown that generated test summaries can be lengthy and redundant, and that it is best to use them in combination with well-defined test names and variables. This is why we propose to extend `UTGen` [5] with LLM-generated summaries since LLMs understand the prompted question and context, and generate output in natural language. Our research question is: **to what extent can the understandability of a test case be influenced by LLM-generated test summaries in terms of context, conciseness, and naturalness?** For this, we first looked at four different prompting techniques and three LLMs. After this, the results suggested that prompt engineering using `codellama:7b-instruct` [1] and few-shot using `ChatGPT` [8] performed the best by analyzing the length of the summaries and the different aspects. After this we conducted a user evaluation to see how the LLM-generated summaries would perform in contrast to the existing tools `TestDescriber` [16] and `DeepTC-Enhancer` [18]. We found out that the LLM-generated summaries scored overall higher and were favored over the existing tools, in particular for the aspect of naturalness. Multiple aspects influence understandability, of which some has both its advantages and disadvantages like the length of the summary.

So to answer the research question, the right prompting technique in combination with a suitable LLM can have a positive impact on the understandability of a test case. With each technique different aspects will be present in the summary, having also a different impact on understandability. The shorter the test case, the more concise but also less information which affects the content. The opposite is also true, where the longer the test case, the less concise but the more information. Our results also indicate that participants prefer context over conciseness. Using an LLM increases naturalness in contrast to template-based summaries.

For future research, it would be good to do a user evaluation to look into the differences in prompting techniques of different LLMs. Next to that, it would be interesting to see how `ChatGPT` would perform with prompt engineering compared to `codellama:7b-instruct`'s prompt-engineered summaries.

8 Responsible Research

Five principles from the Netherlands Code of Conduct

As in the Netherlands Code of Conduct [2], the five principles are honesty, scrupulousness, transparency, independence, and responsibility. To be as honest as possible we document every prompt in Google Documents. This means that there is also version control, so every deletion can be seen. For scrupulousness, we have first gone through many papers to see how they have done their research. This was to get more inspiration, and then we could exercise more care in designing, conducting, and reporting our research. Transparency is important so other researchers can recheck that the research was done correctly. We have tried to write every small step in this paper what we have done, including where the data of our research came from and where it was obtained. This study was also done independently, in other words not connected to a commercial or political nature. This study was fully done to support our PhD candidate in his research. Lastly, we tried to be responsible with the participants during the user evaluation, by not flooding them with many questions that would take more than 15 minutes. If we wanted deeper results, we could do a very long survey, but this would not be good for the well-being and concentration of the participants.

Biases

Biases also play a big role in responsible research. There are different kinds of biases. For confirmation bias, it is important to record your hypotheses before starting the analysis so one can recognize their biases. Our research question was at the beginning of the research formulated in a biased way. We thought LLM-generated summaries would improve test summaries in contrast to existing tools. However, our supervisor pointed out that we do not know that yet, and the research question was altered before starting the study. Now we are more aware of this bias. There can also be selection biases, which can occur when one looks at samples that are not representative of the population. Since we are doing a small study, it was only possible to get around 10 participants. This can increase the bias if they all think in the same way. To mitigate this problem, we have Bachelor students, as well as Master students. But they are all from the same university and around the same age. To fully eliminate this bias, it would be better for a future study to include more participants who differ in more aspects like university and age.

FAIR framework

The study must be findable, accessible, interoperable, and reusable, which is the FAIR framework. We have published the prompts, results, and survey results (without the names of the participants) on GitHub[2], such that it is findable and accessible. The documentation of the code will also be updated such that it is interoperable and reusable. The only part that is a bit more difficult to reproduce is the self-evaluation of RQ1. But as described in *Construct Validity*, we tried to take this into account and mitigate it by implementing rules for others to also be able to reproduce results.

Inappropriate use of LLM-generated summaries

There is a risk that a malicious actor would manipulate the LLM to generate misleading or inaccurate test summaries. This could lead to confusion or misinterpretation of what the tests are actually doing. Next to that, since we built our summarization on the tool `UTGen`, a malicious actor could also somehow be able to generate tests that do not make sense and are irrelevant. This could lead to a drop in the quality of our approach and more ineffective testing. Lastly, since we are prompting the LLM either locally with `codellama:7b-instruct` or via the official API of `ChatGPT`, there is always a chance that somebody could overload the resources of the system or server using a lot of unnecessary large prompts for harmful purposes, which could lead to a denial of service. This is the worst-case scenario and will likely not happen with the servers of `ChatGPT`, but there is always a risk involved.

[2] https://github.com/NaelDj/UTGen-Summarization-Extention

References

[1] *CodeLlama-7b-Instruct*. https://ollama.com/library/codellama:7b-instruct. Accessed: 2024-06-01.

[2] The Dutch Research Council. *nwo.nl*. https://www.nwo.nl/sites/nwo/files/documents/Netherlands%2BCode%2Bof%2BConduct%2Bfor%2BResearch%2BIntegrity_2018_UK.pdf. Accessed 03-06-2024. 2018.

[3] Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. *Getting the most out of your tokenizer for pre-training and domain adaptation*. https://arxiv.org/abs/2402.01035. 2024. arXiv: 2402.01035.

[4] Arghavan Moradi Dakhel et al. "Effective test generation using pre-trained Large Language Models and mutation testing". In: *Information and Software Technology* 171 (July 2024), p. 107468. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2024.107468. URL: https://www.sciencedirect.com/science/article/pii/S0950584924000739.

[5] A. Deljouyi. "Understandable Test Generation Through Capture/Replay and LLMs (Paper on UTGen has not been published yet)". In: ICSE 2024 Doctoral Symposium, Apr. 2024. URL: https://dl.acm.org/doi/10.1145/3639478.3639789.

[6] G. Fraser and A. Andrea. "EvoSuite: Automatic test suite generation for object-oriented software". In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13) Szeged, Hungary, Sept. 2011. DOI: 10.1145/2025113.2025179. URL: https://www.researchgate.net/publication/221560749_EvoSuite_Automatic_test_suite_generation_for_object-oriented_software.

[7] Google. *LongCycleTest.java*. https://github.com/google/dagger/blob/9a67471586c3ed1bccb5d0e13a86af06e024cd1a/javatests/dagger/functional/cycle/LongCycleTest.java#L37. Accessed: 2024-06-01. 2024.

[8] *Introducing ChatGPT*. https://openai.com/index/chatgpt/. Accessed: 2024-06-10.

[9] P. S. Kochhar, X. Xia, and D. Lo. "Practitioners' Views on Good Software Testing Practices". In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) Montreal, QC, Canada: IEEE, Aug. 2019. DOI:

10.1109/ICSE-SEIP.2019.00015. URL: https://ieeexplore.ieee.org/document/8804445.

[10] Juho Leinonen et al. “Comparing Code Explanations Created by Students and Large Language Models”. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2023. event-place:, Turku, Finland. New York, NY, USA: Association for Computing Machinery, 2023, pp. 124–130. ISBN: 9798400701382. DOI: 10.1145/3587102.3588785. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3587102.3588785.

[11] B. Li et al. “Automatically Documenting Unit Test Cases”. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST): IEEE, July 2016. DOI: 10.1109/ICST.2016.30. URL: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/7515485.

[12] Paul W. McBurney and Collin McMillan. “Automatic documentation generation via source code summarization of method context”. In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: Association for Computing Machinery, 2014, 279–290. ISBN: 9781450328791. DOI: 10.1145/2597008.2597149. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2597008.2597149.

[13] Laura Moreno et al. “Automatic generation of natural language summaries for Java classes”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 23–32. DOI: 10.1109/ICPC.2013.6613830.

[14] Noor Nashid, Mifta Sintaha, and Ali Mesbah. “Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 2450–2462. DOI: 10.1109/ICSE48619.2023.00205. URL: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/10172590.

[15] Stanford NLP. *StanfordCoreNLPServerITest.java*. https://github.com/stanfordnlp/CoreNLP/blob/246007929ca8461804d2241b5b3ccbc9897eb1bd/itest/src/edu/stanford/nlp/pipeline/StanfordCoreNLPServerITest.java#L164. Accessed: 2024-06-01. 2024.

[16] S. Panichella et al. “The impact of test case summaries on bug fixing performance: an empirical investigation”. In: ICSE ’16: Proceedings of the 38th International Conference on Software Engineering, May 2016, pp. 547–558. DOI: 10.1145/2884781.2884847. URL: https://dl.acm.org/doi/10.1145/2884781.2884847.

[17] Spring Projects. *Spr8510Tests.java*. https://github.com/spring-projects/spring-framework/blob/35474915903970c410ea61f737ee2aeeab41e063/spring-web/src/test/java/org/springframework/web/context/support/Spr8510Tests.java#L78. Accessed: 2024-06-01. 2024.

[18] D. Roy et al. “DeepTC-enhancer: improving the readability of automatically generated tests”. In: ASE ’20: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Jan. 2021, pp. 287–298. DOI: 10.1145/3324884.3416622. URL: https://dl.acm.org/doi/10.1145/3324884.3416622.

[19] Gail Sullivan and Richard Feinn. “Using Effect Size—or Why the P Value Is Not Enough”. In: *Journal of graduate medical education* 4 (Sept. 2012), pp. 279–82. DOI: 10.4300/JGME-D-12-00156.1.

[20] Jason Wei et al. “Chain of Thought Prompting Elicits Reasoning in Large Language Models”. In: Jan. 2022. URL: https://www.researchgate.net/publication/358232899_Chain_of_Thought_Prompting_Elicits_Reasoning_in_Large_Language_Models.

[21] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. “Investigating the readability of test code”. In: *Empirical Software Engineering* 29.2 (Feb. 2024), p. 53. ISSN: 1573-7616. DOI: 10.1007/s10664-023-10390-z. URL: https://doi.org/10.1007/s10664-023-10390-z.

[22] Zhiqiang Yuan et al. *No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation*. May 2023. URL: https://www.researchgate.net/publication/370605022_No_More_Manual_Tests_Evaluating_and_Improving_ChatGPT_for_Unit_Test_Generation.