

# 修士学位論文

題目

## LLM を用いたプログラム修正に対する ソースコードの意味的情報の調査

指導教員

楠本 真二

報告者

堀 翔太

令和7年1月28日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 6 年度 修士学位論文

LLM を用いたプログラム修正に対する  
ソースコードの意味的情報の調査

堀 翔太

## 内容梗概

ソフトウェア開発におけるデバッグ支援の一つとして、大規模言語モデル (LLM) を用いた自動プログラム修正 (LLM-APR) が注目されている。LLM はプロンプトの設計によってタスク性能が大きく変動するため、LLM-APR の性能改善の為のプロンプトエンジニアリングは重要な研究課題である。本研究では LLM-APR の性能改善を目的として、ソースコードの意味的情報について調査する。意味的情報とはソースコードのコンパイル・実行には影響を与えない、ソースコード理解のための要素であり、JavaDoc に記載された仕様やメソッド名などが該当する。本稿では仕様、メソッド名及び変数名の 3 つを意味的情報として扱う。これらの意味的情報は同じ内容であっても様々な表現での記述が可能であり、開発者のソースコード理解に大きく影響する。よってソースコードにどのような意味的情報を持たせるかは LLM-APR の性能にも大きく影響する可能性があり、また、LLM-APR にとって適切な意味的情報も存在すると考えられる。調査では 3 種類の意味的情報の有無が LLM-APR の性能に影響を与えるのか、及び LLM-APR にとって適切な意味的情報について GPT モデルを対象に調査する。調査の結果、3 種類の意味的情報が LLM-APR の性能向上に寄与することを確認し、さらに LLM-APR にとって適切な意味的情報についても確認した。

## 主な用語

大規模言語モデル (LLM)、自動プログラム修正 (APR)、プロンプトエンジニアリング、意味的情報、ChatGPT

## 目次

1	はじめに	1
2	準備	3
2.1	大規模言語モデル (LLM) . . . . .	3
2.2	自動プログラム修正 (APR) . . . . .	3
2.3	プロンプトエンジニアリング . . . . .	3
2.4	LLM を用いたプログラム修正 (LLM-APR) . . . . .	4
3	Research Questions	6
4	実験と結果	7
4.1	RQ1: ソースコードの意味的情報は LLM-APR の性能に影響を与えるのか? . . . . .	7
4.2	RQ2: LLM-APR にとって適切な識別子名とは? . . . . .	12
4.3	RQ3: LLM-APR にとって適切な仕様とは? . . . . .	15
5	考察	19
5.1	人工バグと実バグに対する意味的情報の影響の違い . . . . .	19
5.2	意味的情報の有効性とソースコードの行数の関係 . . . . .	19
5.3	各ミュータント演算子と修正成功率の関係 . . . . .	21
5.4	LLM を用いたソースコードの自動要約の分析 . . . . .	21
6	妥当性の脅威	24
7	おわりに	25
	謝辞	26
	参考文献	28

## 目次

1	3 種類全ての意味的信息を持つプロンプトパターン “Ps/m/v” の具体例 . . . . .	8
2	3 種類全ての意味的信息を持たないプロンプトパターン “P-/-/-” の具体例 . . . . .	8
3	8 種類のプロンプトパターンごとの修正成功率 . . . . .	10
4	Single Letter に従ったソースコードの具体例 . . . . .	13
5	Abbreviation に従ったソースコードの具体例 . . . . .	13
6	Full Word に従ったソースコードの具体例 . . . . .	14
7	修正対象メソッドの行数と修正成功率の関係 . . . . .	20
8	バグありソースコードの自動要約の具体例：Summary_b . . . . .	23
9	図 8 における修正対象のバグありソースコード . . . . .	23

## 表目次

1	RQ1 で用いるバグデータセットの概要 . . . . .	9
2	実験で用いるミュータント演算子 . . . . .	9
3	それぞれの識別子名における修正成功数 . . . . .	15
4	それぞれの仕様における修正成功数 . . . . .	18
5	各ミュータント演算子と修正成功数/成功率の関係 . . . . .	21

## 1 はじめに

ソフトウェア開発におけるデバッグ支援の一つとして、大規模言語モデル (LLM, Large Language Model) を用いた自動プログラム修正が注目されている。LLM とは大量のテキストデータを用いて学習した言語モデルの一種であり、様々な自然言語タスクに適用可能である。学習データにソースコードを含めた LLM も存在しており、ソースコードを対象とした様々なタスクにも応用できる。その応用の一つとして、LLM を用いた自動プログラム修正 (APR, Automated Program Repair) が挙げられる。本稿ではこの手法を LLM-APR と略す。既存の APR 手法 [1][2][3] の多くはソースコードの性質を考慮したプログラム修正手法であるのに対し、LLM-APR は大量のデータに基づく学習と推論を基本とする点で実現手段が決定的に異なる。LLM-APR は自然言語による自然かつ対話的なプログラム修正が可能という利点だけでなく、その修正精度も既存 APR 技術と同等かそれ以上であると報告されている [4][5][6]。

社会に大きなインパクトを与えた ChatGPT の公開が 2022 年 11 月という点から、根本的に LLM-APR 研究は黎明期にあり、様々な性能改善の可能性が残されている。性能改善に向けたアプローチの一つがプロンプトエンジニアリングである。ChatGPT のような生成型 AI においては、プロンプトの設計がその性能に大幅な影響を与える [7] ことが知られている。Parasaram ら [8] は、GitHub Issues や Error Information 等のバグに関する 7 つの情報をプロンプトに加えたときの LLM-APR の性能改善について調査している。結果として、7 つの情報がそれぞれ性能向上に寄与したと報告している。一方で、7 つの情報を組み合わせてプロンプトに与えたとき、加える情報の数と性能向上は単調増加でなかったとも報告している。すなわち LLM-APR において、プロンプトに加える情報の量と性能は比例せず、情報量の多さにより性能が低下する可能性も存在する。よって、“どんな情報をどれだけ” プロンプトに加えるかは、その性能を左右する重要な要素であるといえる。

本稿では、LLM-APR への入力データとなるソースコードが持つ意味的信息に着目する。ここでの意味的信息とはコンパイルや実行には一切寄与しない、開発者によるソースコード理解のための付帯情報のことを指す。最も重要な意味的信息として考えられるのは、Java における JavaDoc や Python における docstring などの仕様である。これらのドキュメントには、メソッドの目的や振る舞い、利用方法などの意味や解釈が記載されており、対象メソッドの理解に寄与する。さらに、ソースコードの内部に着目すると、メソッド名も意味的信息の一種とみなせる。メソッド名は仕様ほど多くの情報を含まないが、メソッドの責務を極めて端的に表現しているためである。同様に、変数名はメソッド全体の意味や解釈を表現しないが、対象変数の持つ値の意味や役割が付与された意味的信息だといえる。他にも様々な意味的信息が考えられるが、本稿ではこの 3 つを意味的信息として扱う。これらの意味的信息はプログラム実行には影響を与えないものの、開発者のソースコード理解に大きく影響すると報告されている

[9][10]. しかし、これらの意味的情報が LLM のソースコード理解に影響するかは不明である.

本研究の目的は LLM-APR の性能改善であり、そのためにソースコードに付帯する意味的情報の効果を実験的に確かめる. 実験では 3 種類の意味的情報の有無が LLM-APR に与える影響について、及び LLM-APR にとって適切な意味的情報について調査した. 結果として、3 種類の意味的情報の存在により LLM-APR の性能が向上することを確認し、LLM-APR にとって適切なメソッド名と変数名及び仕様についても確認した.

## 2 準備

### 2.1 大規模言語モデル (LLM)

大規模言語モデル (LLM, Large Language Model) とは、大量のテキストデータを用いて学習された言語モデルの一種である。LLM は翻訳、要約、分類などの多くの自然言語処理タスクを実行できる。代表的な LLM として、OpenAI 社によって開発された GPT[11] や、Google 社によって開発された Gemini<sup>\*1</sup> や BERT[12] 等がある。これらの LLM は、従来の畳み込みニューラルネットやリカレントニューラルネットを用いた深層学習と異なり、Transformer モデルを用いて学習を行っている。

LLM はソフトウェア工学の分野でも大きな注目を集めており、コード要約 [13][14]、コード生成 [15][16] 及びバグ検出 [17][18] など様々なソフトウェアタスクに応用されている。LLM を用いたソフトウェア開発支援の一つとして OpenAI Codex<sup>\*2</sup> がある。Codex とは GPT-3 をソースコードに関するタスクにファインチューニングしたモデルである。Jesse ら [19] は Codex が補完するコードの品質を調査している。調査の結果、Codex のコード補完タスクでは、正解のソースコードより多くバグを含むソースコードを生成しており、未だ精度改善の余地があると確認されている。

### 2.2 自動プログラム修正 (APR)

自動プログラム修正 (APR, Automated Program Repair) とは、バグを含むプログラムから全自動でバグを取り除く技術である。既存の APR 手法として、探索ベースの手法 [1][20] やテンプレートベースの手法 [3]、意味論ベースの合成手法 [21] などが挙げられる。これらの手法は、バグを含むプログラムとテストを入力として与え、全てのテストケースを通過するプログラムを目標としてプログラム修正する。いずれもソースコードの静的・動的解析やプログラムの修正事例などを用いた、ソースコード特化の手法であるといえる。

### 2.3 プロンプトエンジニアリング

プロンプトエンジニアリングとは、LLM から目的の出力を生成させるためにプロンプトを設計し最適化するアプローチである。LLM はプロンプトの設計によってタスク性能が大きく変動することが知られている [7][22]。また LLM をソフトウェア工学タスクに適用する際の、プロンプトエンジニアリング手法についても多くの研究がなされている [17][23][24]。

代表的なプロンプトエンジニアリングのテクニックとして、few-shot 学習や、chain-of-thoughts (CoT) が存在している。few-shot 学習とは各タスクの例題をプロンプトに与えることで、タスク性能

---

\*1 <https://deepmind.google/technologies/gemini/>

\*2 <https://openai.com/index/openai-codex>



を向上させる方法である。CoT とはタスクに対して論理的な推論の過程を実施させることで、タスク性能を向上させる方法である。Xia ら [5] は実際に few-shot 学習の効果を確かめており、その性能向上を確認している。

## 2.4 LLM を用いたプログラム修正 (LLM-APR)

LLM-APR とは LLM を利用した APR 技術であり、バグを含むソースコードとバグ修正の指示文を LLM に与えることで、修正されたソースコードを得る。一般的な APR とは異なり、LLM-APR では自然言語文章を用いた自然かつ対話的なプログラム修正が可能である。また LLM-APR の修正性能は既存 APR 技術と比較しても高く現在多くの研究がなされている [4][5][6][25]。

ChatGPT に代表されるサービス型の生成系 AI の登場に伴い、LLM-APR の実現可能性を確かめる研究が進められるようになってきた。Sobania ら [4] は ChatGPT を用いた LLM-APR のプログラム修正能力を実験的に検証している。この研究では、簡単な指示文章とバグを含むソースコードを LLM への入力に用いている。実験ではプログラミングコンテストから得られたバグデータセットである QuixBugs [26] を用いており、40 個のバグに対し 19 個の修正の成功を確認している。より大規模な LLM-APR 調査として、Xia ら [5] は複数の LLM モデルと複数のバグデータセットを用いた実験を行っている。結果として、QuixBugs データセットに対しては 37 個の修正に成功したと報告している。LLM を用いない従来の APR 手法を用いた別の研究 [27] によると、全 10 種類の APR 手法を統合しても QuixBugs に対する修正成功数は 16 個であるとされている。よって、現時点でも LLM-APR の性能は従来の APR を大幅に上回っているといえる。

また、LLM の活用においてはプロンプトの改善がその性能に大きく影響する [7] ことが知られている。Parasaram ら [8] は GitHub issues のようなバグに関する 7 つの情報が LLM-APR の性能に与える影響について調査している。この研究で用いられた 7 つの情報は、Buggy Class, Used Method Signatures, Failing Test, Error information, Runtime Information, Angelic Forest 及び GitHub Issue である。実験では、それぞれの情報を LLM-APR のプロンプトに加えたときの影響について調査しており、各々の情報は性能向上に寄与したと報告している。一方で、7 つの情報を組み合わせてプロンプトに与えたとき、加える情報の数と性能向上は単調増加でなかったとも報告している。具体的には、3 つの情報の組み合わせで最も高い性能を記録し、4 つ以上の組み合わせでは性能が低下したと報告している。すなわち、LLM-APR におけるプロンプトに加える情報は、情報量を多くすると必ず性能向上するわけではなく、むしろ情報量が多いことにより性能が低下する可能性もあると分かる。よって、LLM-APR に対するプロンプトにどんな情報をどれだけ加えるかはその性能を左右する重要な要素であるといえる。

ここで、LLM-APR における修正対象のソースコードにどのような情報を持たせるべきかについて

は未だ明らかではない。ソースコードは自然言語の文章と同様、高い自由度を持っており、同じ内容であっても様々な表現での記述が可能である。例えば、メソッド名や変数名は開発者の考えや癖に強く依存する要素である。これらの識別子名はソースコード理解に強く寄与することが知られている [9][10]。また JavaDoc 記載の仕様はそのメソッドの責務の把握に役立つ。しかしながら、仕様や識別子名が LLM によるバグ修正にも貢献するのか、あるいはどのような仕様や識別子名が LLM に対して有用かについては明らかではない。

### 3 Research Questions

本研究の目的は、LLM-APR の性能改善である。そのために、プロンプトに含まれるソースコードの意味的情報の影響について調査する。ソースコードの意味的情報とは、開発者のソースコード理解の為の情報で、コンパイルや実行には一切影響しない要素のことである。本稿では、仕様、メソッド名及び変数名の3つを意味的情報として扱う。また、調査のために以下の Research Questions を定める。

#### **RQ1：ソースコードの意味的情報の有無が LLM-APR の性能に影響を与えるのか？**

RQ1 では、ソースコードの意味的情報の有無が LLM-APR の性能に影響を及ぼすかを調査する。2 節で述べた通り、LLM-APR に対するプロンプトにどんな情報をどれだけ加えるかはその性能を左右する重要な要素であるといえる。よって、そもそも意味的情報が LLM-APR に影響を及ぼすのかを調べる必要があり、本研究の最も基本となる調査項目である。

#### **RQ2：LLM-APR にとって適切な識別子名とは？**

RQ2 では、LLM-APR にとって適切な識別子名について調査する。識別子名の命名方法は多岐にわたり、その品質は開発者のソースコード理解に大きく影響する [9][10][28]。よって開発者にとって適切な識別子名が存在するように、LLM にとって適切な識別子名も存在すると考えられ、その結果は LLM-APR の性能改善につながる。

#### **RQ3：LLM-APR にとって適切な仕様とは？**

RQ3 では、LLM-APR にとって適切な仕様について調査する。仕様も識別子名と同様に、その記述方法は様々であり開発者の癖や考え方に依存する要素である。よって開発者にとって適切な仕様があるように、LLM にとっても適切な仕様が存在すると考えられる。さらに、LLM にとって適切な仕様の発見は LLM-APR の性能改善につながると考えられる。

## 4 実験と結果

### 4.1 RQ1：ソースコードの意味的情報は LLM-APR の性能に影響を与えるのか？

#### 4.1.1 実験設計

##### LLM へのプロンプトの設計

LLM への入力となるプロンプトの基本構造は指示文章とソースコードのペアである。指示文章には “*The next method has a bug. Please fix it.*” のような、ソースコードにバグが含まれている旨、及びタスクがバグ修正である旨が含まれる。よって想定する利用シーンは、開発者がバグの存在に気づいており、かつその修正方法が不明という状況である。

プロンプトの取り得る種類は 3 種類の意味的情報の有無の組み合わせである 8 通りとなる。以降では、この 8 通りのプロンプトをプロンプトパターンと呼び、P-/-/- の形式で表現する。これら 8 種類のプロンプトパターンの修正結果を比較することで、LLM-APR における意味的情報の影響を調査する。なおプロンプトに与えるソースコードの粒度はメソッドであり、意味的情報の一つである仕様には各メソッドの JavaDoc をそのまま使用する。

Ps/m/v と P-/-/- の具体例をそれぞれ 図 1 と 図 2 に示す。図 1 は全ての意味的情報をもったプロンプトであり、タスクの指示文章、仕様及びバグを含むメソッドがあることが確認できる。それぞれの意味的情報から対象メソッドの理想とする振る舞いも推測できる。一方、図 2 はすべての意味的情報を持たないプロンプト例である。タスクの指示文章及びメソッド名と変数名がマスクされたメソッドが確認できる。これらの情報のみでは対象メソッドの理想とする振る舞いは推測が困難である。このように意味的情報はプログラマのソースコード理解には寄与するが、LLM のソースコード理解にも寄与するかは不明である。なお P-/-/- のようなメソッド名や変数名のマスクを施しているプロンプトパターンでは、“*do not change method name and variable names ...*” といったメソッド名や変数名を変換するなという旨の指示を追加している。これは、後述する修正成否判定の際にマスキングした識別子名を元プロジェクトの状態に戻しテストを実行するためである。

##### データセット

ミュータント生成による比較的規模の大きな人工バグデータセットと実バグデータセットの 2 種類を用いる。人工バグは豊富なバグ数を確保できるため大規模な検証が可能である。しかし、人工バグは実際に発生するバグとは異なるため、実バグにおいても同様の結果が確認できるのか検証する必要がある。よって今回は人工バグと実バグの二種類を用いることとした。それぞれのプロジェクトとバグ数について表 1 に示す。

```
<task>
This method follows the next specification.
But the method contains a bug. Please fix it.
</task>

<specification>
Remove an attribute by key. <b>Case sensitive.</b>
@param key attribute key to remove
</specification>

<method>
public void remove(String key) {
    int i = indexOfKey(key);
    if (i == NotFound)
        remove(i);
}
</method>
```

図1 3種類全ての意味的信息を持つプロンプトパターン“Ps/m/v”の具体例

```
<task>
This method contains a bug. Please fix it.
In addition, do not change method names and
variable names such as "$1" when fixing it.
</task>

<method>
public void $1(String $2) {
    int $3 = indexOfKey($2);
    if ($3 == $4)
        $1($3);
}
</method>
```

図2 3種類全ての意味的信息を持たないプロンプトパターン“P-/-/-”の具体例

・人工バグデータセット：Mutanerator<sup>\*3</sup>を用いてバグデータセットを作成する。MutaneratorはJavaプロジェクトを対象としたミュータント生成ツールである。本稿では表2に示すミュータント演算子を適用する。対象プロジェクトとして、Javaプロジェクトであるjsoup<sup>\*4</sup>とGson<sup>\*5</sup>の2つを用いる。どちらのプロジェクトもGithub上のスター数が10,000を超えており、プロジェクトとしての品質が高く、適切な意味的信息が付与されていると考えられるため実験の対象とした。さらに対象メソッドと

\*3 <https://github.com/kusumotolab/Mutanerator>

表1 RQ1 で用いるバグデータセットの概要

	対象プロジェクト	バグ数
人工バグ	jsoup <sup>*4</sup>	557
	Gson <sup>*5</sup>	383
実バグ	Defects4J.Math	52

表2 実験で用いるミュータント演算子

ミュータント演算子	変換の具体例	
	変換前	変換後
Conditional boundary	a<b	a<=b
Increments	a++	a--
Invert negatives	-1	1
Math	a+b	a-b
Negate conditionals	a==b	a!=b

して、少なくとも1行はテストケースが通過しており、かつ JavaDoc が存在するものを選定している。後述するテスト評価による妥当性確保のためと、本稿では JavaDoc をそのままメソッドの仕様として用いるためである。これらの条件に適するメソッドに対して Mutanerator を適用したところ、jsoup では557個、Gson では383個のバグデータを収集できた。

・実バグデータセット：実バグデータセットとして Defects4J[29] を用いる。Defects4J は Java プロジェクトを対象とした、開発プロセス中に実際に発生したバグを集めたデータセットである。今回は Defects4J の内バグ数が豊富であった Math プロジェクトを対象とする。さらに実験に用いるバグとして、バグ修正箇所が1メソッド内であり、かつ JavaDoc が存在するものを選定する。本実験で LLM に与えるソースコードは単一メソッドであり、また JavaDoc をそのままメソッドの仕様として用いるためである。これらの条件に適するバグとして、52個のバグを収集できた。

## LLM モデル

OpenAI 社によって公開されている GPT-3.5 Turbo を使用する。テキスト形式の入出力を行うモデルであり、自然言語とソースコード両方のタスクに対応している。GPT-3.5 Turbo には入出力トークン数の上限が存在しているが、本研究においてトークン数の上限を上回ること無かったため問題はない。

GPT-3.5 Turbo を呼び出す際のパラメータとして、temperature=0 としている。temperature は、

<sup>\*4</sup> <https://github.com/jhy/jsoup>

<sup>\*5</sup> <https://github.com/google/gson>

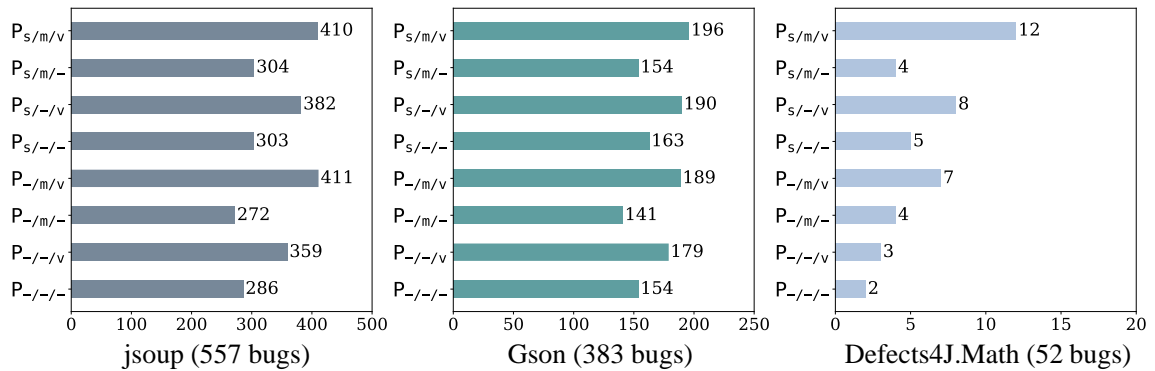


図3 8種類のプロンプトパターンごとの修正成功数

LLM の出力のランダム性に関する。GPT-3.5 Turbo では、0~1 で指定され、0 に近いほど程ランダム性が小さくなる。LLM の出力のランダム性を低くし再現性を確保するため本実験では 0 を用いる。

#### 評価方法

各プロンプトパターンごとの修正成功数により評価する。生成 AI には同様のプロンプトであってもランダム性を持つ特性が存在している [15]。これらの対策のため LLM-APR に関する他研究 [4] でも複数の試行を行っていることから、本稿でも同一のプロンプトについて 3 回の試行を行う。3 回全ての出力結果について修正成否の判定を行い、1 度でも修正成功となった場合そのバグは修正成功とする。

修正成否の判定は各プロジェクトのテストを用いる。LLM の出力結果に対して、マスキング処理を施した識別子名を元のプロジェクトの状態に戻し、それらを元プロジェクトにマージしテストを実行する。

#### 4.1.2 結果

各バグデータセットに対する修正成功数を 図 3 に示す。左のグラフが jsoup に、中央のグラフが Gson に、右のグラフが Defects4J.Math にそれぞれ対応している。以降では、P<sub>s</sub>/m/v での修正成功数を #P<sub>s</sub>/m/v と表現する。例えば、jsoup においては #P<sub>s</sub>/m/v=410 と示す。本セクションでは、各データセットごとに各意味的情報に注目して結果を述べる。

##### 人工バグ

*jsoup*: 仕様による修正成功数の結果に着目すると、#P<sub>s</sub>/m/v=410、#P<sub>-</sub>/m/v=411 であることから、ソースコードに適切な識別子名が付与されているとき、LLM-APR の性能に対する仕様の影響はほとんどないと推測される。しかしその他の仕様の有無が異なるペアを比較すると、#P<sub>s</sub>/m/-=382、#P<sub>-</sub>/m/-=359 と #P<sub>s</sub>/-/v=304、#P<sub>-</sub>/-/v=272 及び #P<sub>s</sub>/-/-=303、#P<sub>-</sub>/-/-=286 であることから、適切な識別子名が存在していない場合、仕様が LLM-APR の性能向上に寄与していると分かる。次に変数名の有無が異なるペアについて、#P<sub>s</sub>/m/v=410、#P<sub>s</sub>/m/-=304 と #P<sub>-</sub>/m/v=411、

#P-/m/-=272 であり、どちらも変数名を有することで成功数が 100 以上大きくなっていることが分かる。よって変数名は LLM-APR の性能向上に大きく寄与していることが確認できる。また、その他ペアについても変数名による LLM-APR の性能向上の傾向が確認できる。最後にメソッド名の有無が異なるペアに注目する。#Ps/m/v=410, #Ps/-/v=382 であり、#P-/m/v=411, #P-/-/v=359 である。どちらもメソッド名を有することで修正成功数が大きくなっていることが確認できる。しかしその差は変数名に比べると小さく、変数名よりは LLM-APR の性能に関する影響は小さいといえる。また、その他のメソッド名の有無が異なるペアでは、#Ps/m/-=304, #Ps/-/-=303 であり、#P-/m/-=272, #P-/-/=-=286 であり、適切な変数名が存在しないとき、メソッド名は LLM-APR の性能向上に寄与しないと分かる。

*Gson*: 仕様の影響について注目すると、#Ps/m/v=196, #P-/m/v=189 であり大きな差はない。またその他の仕様が異なるペアについては、少なながら仕様を有する方が結果が向上していることが確認でき、jsoup と同様に適切な識別子名が存在しない時ほど仕様が LLM-APR の性能向上に寄与していると考えられる。次に変数名の影響について注目すると、#Ps/m/v=196, #Ps/m/-=154 と #P-/m/v=189, #P-/m/-=141 であり、変数名によって修正成功数が大きく向上している。また、その他の変数名の有無が異なるペアについても変数名の性能向上の傾向が確認でき、jsoup 同様に変数名が LLM-APR の性能向上に大きな影響を持つと分かる。最後にメソッド名の影響について、#Ps/m/v=196, #Ps/-/v=190 であり、#P-/m/v=189, #P-/-/v=179 である。これら 2 つではメソッド名を持つ方が成功数が大きい。しかし残りのペアでは、#Ps/m/-=154, #Ps/-/-=163 と #P-/m/-=141, #P-/-/=-=154 であり、これも jsoup 同様に適切な変数名が存在しない場合、メソッド名は LLM-APR の性能向上には寄与せず、性能低下をもたらす可能性を示唆している。

### 実バグ

*Defects4J.Math*: 仕様の影響について比較すると、#Ps/m/v=12, #P-/m/v=7 であることから、仕様が LLM-APR の修正性能向上に寄与していることが分かる。また、その他の仕様の有無が異なるペアについても、仕様による性能向上を確認できる。次に変数名に注目すると、#Ps/m/v=12, #Ps/m/-=4 と #P-/m/v=7, #P-/m/-=4 である。また、その他の変数名の有無が異なるペアについても、変数名による性能向上が確認できる。最後にメソッド名の影響について、#Ps/m/v=12, #Ps/-/v=8 と #P-/m/v=7, #P-/-/v=3 及び、#P-/m/-=4, #P-/-/=-=2 であることから、メソッド名による性能向上が確認できる。しかし、#Ps/m/-=4, #Ps/-/-=5 のペアではメソッド名が性能向上に寄与していない。これは人工バグでも見られた傾向であり、適切な変数名がない時メソッド名による性能低下がみられる。実バグにおける意味的情報の影響の傾向は大部分が人工バグと同様であった。ただし、バグ数に対する意味的情報による性能向上の割合は人工バグと比較して大きくなっており、この原因考察については 5.1 節で述べる。



**RQ1 への回答:** 3 種類の意味的情報がほとんどの状況において性能向上に寄与した。

## 4.2 RQ2 : LLM-APR にとって適切な識別子名とは？

### 4.2.1 実験設計

#### 識別子名の命名規則

本実験で用いる識別子名は、単一文字による識別子名（以降 Single Letter）、略語による識別子名（以降 Abbreviation）及び略語を一切使わない完全な英単語による識別子名（以降 Full Word）である。これは、Lawrie らの既存研究 [30] に従った命名規則である。図 4、図 5 及び 図 6 にそれぞれのソースコードの具体例を示す。Lawrie らは、3 種類の識別子名が開発者のソースコード理解にどのように影響するのかを調査しており、Full Word が最も開発者のソースコード理解に役立ったと報告している。よって開発者のソースコード理解にとって最も優れた識別子名は Full Word であると分かったが、LLM のソースコード理解にとって最も優れた識別子名はいまだ不明である。

それぞれの識別子名の作成方法について、まずは対象の識別子名に対応する Full Word を作成する。今回対象にしたプロジェクトはソースコードの品質が高く、Full Word の単語は元の識別子名から全て推測可能であった。Single Letter については Full Word の頭文字を取ることで作成する。Abbreviation も Full Word を基に作成する。Full Word に対して一般的に知られる略語が存在する場合はその略語を使用し（例：`length` -> `len`）、一般的に知られる略語が存在しない場合には略語の傾向に従って作成する（例：`offset` -> `ofst`）。

#### LLM へのプロンプトの設計

LLM へのプロンプトの基本構造は RQ1 と同様に、指示文章とソースコードのペアである。想定する利用シナリオも RQ1 と同様、開発者がバグの存在に気づいており、かつその修正方法が不明という状況である。プロンプトの取り得る種類は 3 種類の識別子名に従った 3 通りである。これら 3 種類のソースコードに対してバグ修正成否数を比較することで、各識別子名が LLM-APR に与える影響について調査する。

#### データセット

ミュータント生成による人工バグデータセットを用いる。ミュータント生成には RQ1 と同様に Mutanerator<sup>\*3</sup>を用いる。対象プロジェクトは、jsoup<sup>\*4</sup>である。jsoup に存在するメソッドの内、行数が 25 以下で Mutanerator によって作成できるバグの数が 7 以上、さらに対応するテストが存在しているメソッドを実験対象とする。本実験では、全てのメソッドに対して識別子名の対応表を作成する必要があるため、効率よく多くのバグで実験をするためにこのような条件を設けた。これらの条件に合致したメソッドは合計 28 個あり、それらに対して Mutanerator を適用したところ 293 個のバグを生成でき

```

int nio(CharSequence s) {
    bufferUp();
    // doesn't handle scanning for surrogates
    char sc = s.charAt(0);
    for (int o = bf; o < bl; o++) {
        // scan to first instance of startchar:
        if (sc != cb[o])
            while(++o < bl && sc != cb[o]) { /* empty */ }
        int i = o + 1;
        int l = i + s.length()-1;
        if (o < bl && l <= bl) {
            for (int j = 1; i < l && s.charAt(j) == cb[i]; i++, j++) { /* empty */ }
            if (i == l) // found full sequence
                return o - bf;
        }
    }
    return -1;
}

```

図4 Single Letter に従ったソースコードの具体例

```

int nxtIdxOf(CharSequence seq) {
    bufferUp();
    // doesn't handle scanning for surrogates
    char strtChar = seq.charAt(0);
    for (int ofst = bufPos; ofst < bufLen; ofst++) {
        // scan to first instance of startchar:
        if (strtChar != charBuf[ofst])
            while(++ofst < bufLen && strtChar != charBuf[ofst]) { /* empty */ }
        int i = ofst + 1;
        int lst = i + seq.length()-1;
        if (ofst < bufLen && lst <= bufLen) {
            for (int j = 1; i < lst && seq.charAt(j) == charBuf[i]; i++, j++) { /*
                empty */ }
            if (i == lst) // found full sequence
                return ofst - bufPos;
        }
    }
    return -1;
}

```

図5 Abbreviation に従ったソースコードの具体例

```

int nextIndexOf(CharSequence sequence) {
    bufferUp();
    // doesn't handle scanning for surrogates
    char startCharacter = sequence.charAt(0);
    for (int offset = bufferPosition; offset < bufferLength; offset++) {
        // scan to first instance of startchar:
        if (startCharacter != characterBuffer[offset])
            while(++offset < bufferLength && startCharacter != characterBuffer[
                offset]) { /* empty */ }
        int i = offset + 1;
        int last = i + sequence.length()-1;
        if (offset < bufferLength && last <= bufferLength) {
            for (int j = 1; i < last && sequence.charAt(j) == characterBuffer[i]; i
                ++, j++) { /* empty */ }
            if (i == last) // found full sequence
                return offset - bufferPosition;
        }
    }
    return -1;
}

```

図 6 Full Word に従ったソースコードの具体例

た。よって実験ではこれらのバグを用いる。

## LLM モデル

OpenAI 社によって公開されている、gpt-4o-mini-2024-07-18（以後 gpt-4o-mini）と gpt-4o-2024-11-20（以後 gpt-4o）を用いる。RQ1 と使用したモデルが異なるのは、実験を行った際に広く使用されていたモデルが異なったためである。gpt-4o はマルチモーダルモデルであり、従来のベンチマークで高い水準に達している現在注目のモデルである。gpt-4o-mini は gpt-4o モデルの軽量モデルであり、パフォーマンスの精度は下がるものの、より軽量に早く動作するモデルである。パラメータとして、RQ1 と同様に temperature=0 としている。これは実験の再現性を高め安定した結果を得るためである。

## 評価方法

各識別子名による修正成功数により評価する。また RQ1 と同様に、LLM のランダム性を考慮して各プロンプトに対して 3 回の試行を行う。3 回全ての出力結果について修正成否の判定を行い、1 回でも対象のテストに通過した場合は修正成功とする。

修正成否の判定はテストを用いる。RQ1 と同様に、LLM の出力結果に対して識別子名を元のプロジェクトの状態に戻し、それらを元のプロジェクトにマージしてテストを実行する。

表 3 それぞれの識別子名における修正成功数

	Single Letter	Abbreviation	Full Word
gpt-4o-mini	169	<b>229</b>	221
gpt-4o	255	270	<b>277</b>

#### 4.2.2 結果

各モデルとそれぞれの修正成功数を表 3 に示す。gpt-4o-mini の結果に注目すると、Single Letter が 169、Abbreviation が 229 及び Full Word が 221 の修正成功数を示している。よって、Abbreviation が最も高い性能であり、これは開発者にとって良い識別子名とは異なることが分かる。ただし、Abbreviation と Full Word の修正成功数の差は小さく、対象ソースコードや LLM のランダム性による誤差の範囲ともいえる。対して、Single Letter に関しては大きく修正成功数が低下していると分かる。これは開発者と同様に、Single Letter の識別子名は LLM のソースコード理解に有効でないことを示している。次に gpt-4o の結果に着目すると、Single Letter が 255、Abbreviation が 270 及び Full Word が 277 の修正成功数となっている。よって、開発者のソースコード理解と同様に、Full Word が最も高い性能を示している。ただし、gpt-4o-mini と同様に Abbreviation と Full Word に大きな差はなく、誤差の範疇ともいえる。また Single Letter については他 2 つの識別子名と比較して修正成功数が低くなっている。ただし、gpt-4o-mini よりもその差は小さくなっており、より大きなモデルでは情報量が少ない識別子名であっても多くのバグ修正が可能であると分かる。全体の結果として、どちらのモデルにおいても Abbreviation と Full Word ではほとんど同じ性能を示し、Single Letter では低い性能を示した。よって、開発者のソースコード理解にとって Full Word が最も適切であることから、開発者にとって適切な識別子名と LLM-APR にとって適切な識別子名は異なる可能性を示唆している。

**RQ2 への回答:** Abbreviation と Full Word で高い性能を示し、Single Letter では低い性能を示した。これは開発者にとって適切な識別子名と LLM-APR にとって適切な識別子名は異なる可能性を示唆している。

### 4.3 RQ3：LLM-APR にとって適切な仕様とは？

#### 4.3.1 実験設計

##### 4 種類の仕様

RQ3 で用いる仕様は、JavaDoc、バグなしソースコードの自動要約（以降、Summary\_c）、バグありソースコードの自動要約（以降、Summary\_b）及びテストケースの 4 種類である。JavaDoc はメソッ

ドの引数や出力及び動作や責務などを記述した Java における一般的な仕様である。また LLM を用いたコード要約も現在大きく注目されており [13][14] 仕様的一种といえる。ただし LLM-APR のシナリオを想定すると、開発者はバグを含むソースコードから自動要約を取得する必要がある。通常バグを含むソースコードから要約をすると適切な仕様は作成できないが、LLM の自動要約において上手くバグを抽象化して表現することで、LLM-APR の性能改善につながる可能性があると考えている。よって実験では実際のシナリオを想定した Summary\_b と、理想的な要約文章を想定した Summary\_c の二つを用いる。最後にテストケースも仕様の一つだと考えられる。テストケースは自然言語でソースコードの振る舞いを表現しないが、テスト対象の挙動を検査しており、対象のもつべき振る舞いを間接的に表現しているためだ。RQ3 では 4 種類の仕様及び仕様を用いないパターンの計 5 通りのプロンプトパターンを用意し、各種仕様が LLM-APR に与える影響を比較調査する。

仕様の作成方法について、JavaDoc はプロジェクト固有のものをそのまま用いる。自動要約は対象ソースコードを LLM に与えて作成する。その際に用いる指示文章は、“*Provide a short code description of the following code*” である。これは、LLM を用いたバグ限局に関する Widyasari らの既存研究 [17] において、ソースコードの仕様を得るために使用された指示文章であり、本稿でもこれを用いた。最後に、テストケースは評価に使用する対象テストコードの内、import 等メタ的な情報を除いた class 内部のソースコードをそのまま使用する。import 等の情報は仕様としての情報が少ない一方 LLM のデータリクへの寄与が大きいと考えたためである。

### LLM へのプロンプトの設計

LLM へのプロンプトの基本構造は RQ1 と同様に、指示文章とソースコードのペアである。想定する利用シナリオも RQ1 と同様、開発者がバグの存在に気づいており、かつその修正方法が不明という状況である。ただし、Summary\_c を用いた実験は想定シナリオとは異なり、理想的なソースコードが存在している調査のためのシナリオである。プロンプトの取り得る種類は 4 種類の仕様を用いる 4 通りと、仕様を用いない 1 通りの計 5 通りである。仕様を用いないプロンプトは他の仕様による性能向上を確認するための比較用のプロンプトである。これら 5 種類のソースコードによる修正結果を比較することで、それぞれの仕様が LLM-APR に与える影響を調査する。

### データセット

ミュータント生成による人工バグデータセットを用いる。また、他 RQ と同様に、ミュータント生成には Mutanerator<sup>\*3</sup>を用い、対象プロジェクトは，jsoup<sup>\*4</sup>である。対象のメソッドの条件として、RQ2 と同様の条件と JavaDoc が存在するという条件を設けている。RQ3 においては JavaDoc が存在することが必須だからである。これらの条件に合致したメソッドが合計 15 個あり、それらに対して Mutanerator を適用したところ 160 個のバグを生成できた。よってこれらのバグを実験に用いる。

### LLM モデル

OpenAI 社によって公開されている, gpt-4o-mini-2024-07-18 (以後 gpt-4o-mini) と gpt-4o-2024-11-20 (以後 gpt-4o) を用いる. また他 RQ と同様に, パラメータとして `temperature=0` としている. これは実験の再現性を高め安定した結果を得るためである. Summary\_c と Summary\_b の作成にも同じ LLM モデルを用いる. すなわち, gpt-4o-mini を用いた実験においては自動要約の作成にも gpt-4o-mini を用いており, gpt-4o を用いた実験では自動要約の作成に gpt-4o を用いている.

#### 評価方法

各プロンプトによる修正成功数により評価する. また他 RQ と同様に, LLM のランダム性を考慮して各プロンプトに対して 3 回の試行を行う. 3 回全ての出力結果について修正成否の判定を行い, 1 回でも対象のテストに通過した場合は修正成功とする. 修正成否の判定はテストを用いる.

#### 4.3.2 結果

4 種類の仕様と仕様なしのプロンプトによる修正成功数を表 4 に示す. gpt-4o-mini の結果に着目すると, Summary\_c の 131 が最も高い成功数であった. 次に JavaDoc とテストケースで 124 と 122 を示した. 最後に Summary\_b と仕様なしが, それぞれ 115 と 116 であり低い修正成功数となった. ただし, RQ1 の結果では適切なメソッド名と変数名が存在するとき, JavaDoc はほとんど性能向上に寄与しなかったため, Summary\_c 以外の結果は有効な差ではないと考えられる. 次に gpt-4o の結果に注目すると, JavaDoc が 147, Summary\_c が 155, Summary\_b が 146, テストケースが 147 及び仕様なしが 147 であった. よって, Summary\_c のみが少し高い修正成功数を示し, そのほかでは有効な差はみられなかった. 全体の結果として, どちらのモデルにおいても Summary\_c が高い性能を示した. Summary\_b ではバグを上手く抽象化した要約を作ることで, LLM-APR の性能向上につながる可能性があると考えていたが, Summary\_b は性能向上につながらなかった. 5 節では Summary\_b が性能向上につながらなかった原因について具体例と共に分析する. また, 仕様なしと比較して仕様を追加したことにより性能が大きく低下することはなかった. よって, 実際のシナリオを想定すると, 仕様となるものが存在する場合はプロンプトに加える方が良いと考えられる.

**RQ3 への回答:** Summary\_c で高い性能を示した. JavaDoc, Summary\_b 及びテストケースでは有効な差は見られなかった.

表 4 それぞれの仕様における修正成功数

	JavaDoc	Summary_c	Summary_b	テストケース	仕様なし
gpt-4o-mini	124	<b>131</b>	115	122	116
gpt-4o	147	<b>155</b>	146	147	147

## 5 考察

### 5.1 人工バグと実バグに対する意味的情報の影響の違い

RQ1 の結果より、実バグにおいて意味的情報の性能向上への影響は人工バグと比較して大きいといえる。この理由として、人工バグはソースコードの構造的な特徴からバグを推測しやすいことが考えられる。例えば人工バグとして、“`for(int i=0; i<n; i--){...}`”を想定する。このとき、変数名の有無に関係なく“`i<n`”又は“`i--`”にバグがある可能性が高いと推測できる。なぜならこの条件式のままでは無限ループに入る可能性が高くこのような記述はしないためである。したがって、人工バグは意味的情報の有無にかかわらずソースコードの構造のみから修正可能なバグが存在すると考えられる。一方、実バグは構造的な特徴からバグを推測するのは難しい。Defects4j.Mathのバグ修正は、新しい条件分岐の追加、メソッド外で定義されるメソッドの呼び出しやメソッド内の計算の変更等が多く見られた。このようなバグはメソッドの本来の振る舞いを知らなければ、開発者であれど修正方法の推測はできない。よってLLMもソースコードの意味的情報無しに、このようなバグを修正することは困難であると考えられる。

### 5.2 意味的情報の有効性とソースコードの行数の関係

意味的情報の有無がLLM-APRの性能に与える影響は、対象ソースコードの責務や行数の多さに関係していると考えられる。RQ1のjsoupを対象とした実験における、メソッドの行数と各プロンプトパターンにおける修正成功率の関係を図7に示す。横軸に修正対象メソッドの行数、縦軸に修正成功率を示している。メソッドは5行単位で分割しており、1行から5行のメソッド、6行から10行のメソッド、11行から15行のメソッド、16行から20行のメソッド及び21行以上のメソッドの5つに分類している。P-/-/に注目すると、1行から5行のメソッドでは修正成功率が65%程度あるのに対して、21行以上のメソッドでは41%程度となっている。よって、LLM-APRにおいて修正対象メソッドの行数が増加すると修正成功率が低下することが分かる。また、そのほかのプロンプトパターンにおいても同様の傾向が確認できる。これは修正対象メソッドの行数が多くなることで、そのメソッドの責務や振る舞いが複雑になり、バグ修正の難易度自体が上がっていることに起因すると考えられる。一方、行数の少ないメソッドはその振る舞いが簡潔で推測しやすい。また、このようなメソッドは学習データに構造的に類似しているメソッドが多く存在していると考えられるため、LLM-APRにおける修正が簡単になると考えられる。

$P_s/m/v$ と $P_s/m/-$ のペアに注目すると、1から5行のメソッドにおいては $P_s/m/v$ が約88%の修正成功率であり $P_s/m/-$ が約78%の修正成功率であるため、修正成功率の差が1割程度となっている。一方、20行以上のメソッドにおいては $P_s/m/v$ が68%程度、 $P_s/m/-$ が33%程度となっておりその差



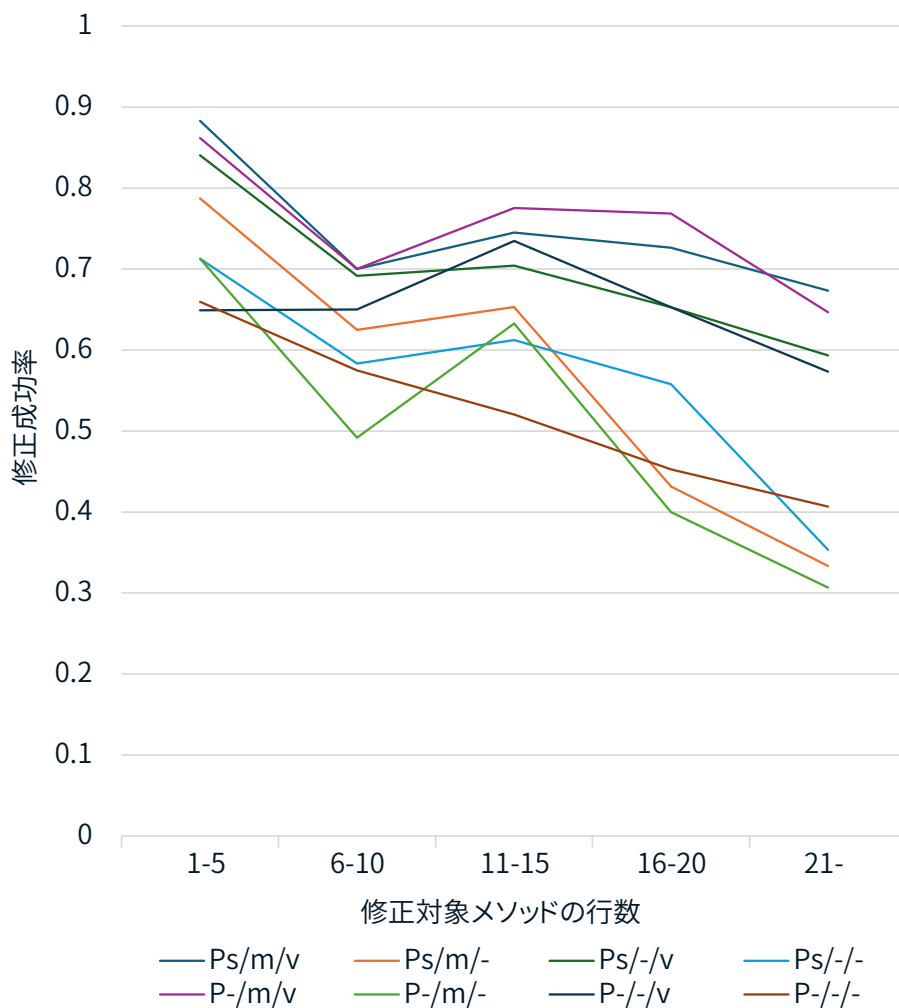


図7 修正対象メソッドの行数と修正成功率の関係

が3割以上である。よって適切な変数名は、ソースコードの行数の増加による修正成功率の低下の割合を軽減していると分かる。また、他の変数名の有無が異なるペアについても同様の傾向が確認できる。これは行数が少なく簡潔なメソッドは、ソースコードの構造的情報のみでバグ修正が可能であるのに対して、行数が多く責務の多いメソッドはソースコードの構造的情報からのバグの推測は困難であり、変数名がメソッドの理解により有効に働いたと推測できる。

次に、Ps/m/- と Ps/-/- に注目すると、1 から 5 行のメソッドにおいては Ps/m/- が約 78% の修正成功率であり Ps/-/- が約 70% の修正成功率であるため、Ps/m/- の方が少し高い修正成功率を示している。一方、16 行から 20 行のメソッドにおいては Ps/m/- が 43% 程度、Ps/-/- が 56% 程度となっておりその差が逆転している。よって行数の多いメソッドにおいて適切な変数名が付与されてい

表 5 各ミュータント演算子と修正成功数/成功率の関係

ミュータント演算子	Single Letter	Abbreviation	Full Word	合計バグ数
Conditionals Boundary	45/0.563	67/0.838	69/0.863	80
Increments	18/0.720	22/0.880	22/0.880	25
Invert Negatives	7/0.778	7/0.778	7/0.778	9
Math	22/0.537	26/0.634	25/0.610	41
Negate Conditionals	77/0.558	107/0.775	98/0.710	138

いとき、メソッド名は性能を低下させる可能性がある。これは、適切な変数名がない状態でメソッド名だけが存在したためメソッド名の持つ情報量の比重が大きくなり、ソースコード理解をミスリードしたのでないかと推測している。RQ1の結果では、適切な変数名がないときメソッド名が LLM-APR の性能を低下させていたが、これが一つの要因だと考えられる。

### 5.3 各ミュータント演算子と修正成功率の関係

RQ2 の gpt-4o-mini を使った実験における、各ミュータント演算子に対応する修正成功数を表 5 に示す。各項目は修正成功数と合計バグ数に対する修正成功率を記述しており、最も右の列が各ミュータント演算子によって生成された合計のバグ数を示している。まず全体の傾向として、Increments と Invert Negatives の修正成功率が高いと分かる。また Single Letter と Abbreviation 及び Full Word を比較しても修正成功率に大きく差がないことから、これら二つのミューテーション演算子は意味的情報の質にかかわらず修正しやすく、構造的情報から推測しやすい可能性を示唆している。次にそれぞれの識別子名間の傾向に注目すると、Abbreviation と Full Word では 5 つ全てのミューテーション演算子において修正成功数の差が 10 個未満となっている。よって Abbreviation と Full Word の間に大きな差はないと分かる。対して、Single Letter では Conditionals Boundary と Negate Conditionals において、他の識別子名と比較して修正成功率が低くなっている。よってこれら二つのミューテーション演算子は、構造的情報から推測するのが難しく、意味的情報の品質が LLM-APR の性能に影響を与えていると分かる。

### 5.4 LLM を用いたソースコードの自動要約の分析

RQ3 における Summary\_b が性能向上につながらなかった原因として、要約の粒度が関係していると考えられる。Summary\_b の具体例を図 8 に、また対応する修正対象メソッドを図 9 に示す。対象とするバグの行が `Validate.isTrue(maxPaddingWidth < -1);` であり、本来 `>=` である部分が `<` に変換されている。図 8 より、Summary\_b はソースコードの内部の細かい説明までされている

ことが分かる。例えばバグに対する Summary\_b の該当部分は “*Ensures ‘maxPaddingWidth‘ is less than ‘-1‘ (though this condition seems unusual and might be an error).*” となっている。ソースコードの各行に対して細かい仕様を作成するとき、仕様の内容がバグの影響を大きく受けるのは当然である。本稿では既存研究に倣った単純なプロンプトを用いこのような仕様を作成した。しかし few-shot 学習などのプロンプトテクニックを用いるとバグに影響を受けない抽象化した仕様を作成でき、その結果 LLM-APR の性能向上につながる可能性があると考えている。より抽象化したレベルの仕様を作成させ LLM-APR の性能向上につなげられるかは今後の調査課題である。

This code defines a static method `padding` that generates a string of spaces with a specified width, subject to a maximum padding width constraint. Here's a breakdown of its functionality:

- 1. `**Input Validation**`:**
  - Ensures the `width` is non-negative (`width >= 0`).
  - Ensures `maxPaddingWidth` is less than `-1` (though this condition seems unusual and might be an error).
- 2. `**Adjust Width**`:**
  - If `maxPaddingWidth` is not `-1`, the `width` is adjusted to be the smaller of `width` and `maxPaddingWidth`.
- 3. `**Return Predefined Padding**`:**
  - If the adjusted `width` is less than the length of a predefined `padding` array, it returns the corresponding precomputed padding string from the array.
- 4. `**Generate Padding Dynamically**`:**
  - If the `width` exceeds the length of the `padding` array, it creates a new character array of spaces (`' '`) with the specified `width` and converts it to a string.

The method ultimately returns a string of spaces of the desired width, constrained by the `maxPaddingWidth` if applicable.

図8 バグありソースコードの自動要約の具体例：Summary\_b

```
public static String padding(int width, int maxPaddingWidth) {
    Validate.isTrue(width >= 0, "width must be >= 0");
    Validate.isTrue(maxPaddingWidth < -1);    // bug here ">=" -> "<"
    if (maxPaddingWidth != -1)
        width = Math.min(width, maxPaddingWidth);
    if (width < padding.length)
        return padding[width];
    char[] out = new char[width];
    for (int i = 0; i < width; i++)
        out[i] = ' ';
    return String.valueOf(out);
}
```

図9 図8における修正対象のバグありソースコード

## 6 妥当性の脅威

今回用いた LLM モデルのトレーニングデータは公開されておらず、実験対象のプロジェクトがトレーニングデータに含まれているかどうかは検証できない。このデータリークにより本稿の結果が LLM-APR の本来の精度より過剰に評価している可能性がある。しかし、RQ1 では Ps/m/v のプロンプトパターンであっても、jsoup では約 2.7 割、Gson では約 4.9 割の修正に失敗している。この事実により、本稿の結果全てがデータリークの恩恵に起因するわけではないといえる。またデータリークによる最も懸念すべき問題は、過学習による実験結果の偏りである。しかし、RQ1 における実験結果は 3 つのプロジェクトにおいて同様の傾向が得られた。よって、データリークによる LLM-APR の性能の過大評価の恐れは残るが、実験結果の傾向はデータリークの影響を受けていないと考えられる。

## 7 おわりに

本研究では、LLM-APR の性能改善を目的として、ソースコードが持つ意味的情報の影響を調査した。本稿での意味的情報とは、仕様、メソッド名及び変数名の3種類である。RQ1では、人工バグと実バグに対して実験を行い、それぞれの意味的情報の有無による性能向上を確認した。また、変数名が最も高い影響を示した。RQ2では、3種類の識別子名を用いてLLM-APRにとって適切な識別子名について調査し、AbbreviationとFull Wordによるソースコードで高い性能を確認した。RQ3では、4種類の仕様を用いてLLM-APRにとって適切な仕様について調査し、適切なソースコードによる自動要約で性能向上を確認した。

今後の課題として、他のLLMモデルでの検証やプロンプトテクニックの導入が考えられる。本稿ではGPTモデルを用いた実験しかしておらず、実験結果が全てのLLMに対して一般化可能な事実であるかが不明である。よって、他のLLMモデルを用いて実験を行い、LLMの一般的事実として結果を得る必要がある。また、プロンプトテクニックの導入も必要な課題である。本実験のプロンプトは指示文章と修正対象ソースコードという最も基本的な要素だけで構成されている。2.3節で述べた通り、CoTやfew-shot学習はLLMのタスク性能をあげるうえで重要な要素である。よって、このようなプロンプトテクニックを導入したうえで、LLM-APRにとって適切な意味的情報を調査する必要がある。

今後の展望として、LLM-APRの性能をさらに向上させる識別子名についての調査がある。Schankinらの研究[10]においては、長く説明の多い識別子名は短く端的な識別子名より開発者のソースコード理解に有効であったと報告している。RQ2では、Lawrieらの既存研究[30]に倣った3種類の識別子名を用いたが、明確にLLM-APRに適切な識別子名は確認できなかった。よって、さらに多くの識別子名の種類に対してLLM-APRを実行し、LLM-APRにとって適切な識別子名について有意な差を確認したい。

## 謝辞

本研究の遂行にあたり、多くの方々にご指導とご支援を賜りました。

2年間を通して終始多大なご指導を賜った、松本真佑准教授に深く感謝申し上げます。研究活動においては、多くの指導と経験を賜りました。毎週のミーティング、論文の執筆及び学外発表など、研究活動を通して体験した全てが私を大きく成長させました。指導は放置より多くの力を必要としますが、真摯に向き合って指導いただいたため、研究に真剣に取り組むことができました。学外発表においては、食事をご馳走になり、発表についても尽力いただきました。先生との発表練習があったため、また、後ろに先生がいるという気持ちのおかげで自信をもって発表できました。TAを通して多くのことを学びました。TAの経験は自分をすごく成長させた実感しており、それは偏に松本准教授の下で学ぶことができたためです。また、研究以外の活動においても多くを学ぶことができ、多くの思い出ができました。人としての礼儀、人に伝える力、プレゼン力及び文章作成力など社会人として重要であるが、学ぶことが困難な力を多く学びました。楠本研究室での修士2年間は自分を大きく成長させました。これは偏に松本真佑准教授のおかげです。

楠本真二教授には、研究内容だけでなく日ごろの研究室生活を多く支えていただきました。研究室で出会った際にはいつも優しく話しかけてくださり、本当にいい環境で過ごさせていただきました。日ごろから学生を気にかけてくださる先生の姿勢のおかげで、研究に対するやる気が出ました。深く感謝申し上げます。

ソフトウェア工学講座の肥後芳樹教授には、鋭く的確な指摘をたくさん頂きました。毎月のミーティングを目標に研究を進めることで、研究のモチベーションを維持し、修士論文を書き上げることができました。また研究外の雑談でも話していただき、研究室生活を楽しく過ごすことができました。深く感謝申し上げます。

事務補佐員の橋本美砂子氏には、普段の研究室生活から出張時のサポートまで多くお世話になりました。事務書類関係の仕事で多く不手際をする中、厭うことなく手助けしていただきました。普段の生活においてもとても気にかけていただき、私が研究に集中できるよう尽力いただきました。深く感謝申し上げます。

昨年度卒業された先輩の皆様には、多く指導していただきました。研究以外でもたくさん話していただき、修士の一年間を楽しく過ごすことができたのは先輩方のおかげです。先輩方の背中を見て過ごしたからこそ、また土日祝に登校しても常に先輩がいたからこそ、1年間で大きく成長することができました。深く感謝申し上げます。

研究室の同期の皆様には、研究はもちろんのこと普段の研究室生活までたくさんの楽しい時間を頂きました。修士から入学した私に対して、入学初日からずっと助けていただき感謝しています。また、同

じく修士から入学した同期もおり、彼と入学当初からたくさん笑って話せたからこそ、研究室生活に恐れることなく馴染めたと感じています。研究活動においても気軽に相談できる同期のおかげで、悩むことなく努力を続けられました。また研究外でも、雑談を交わし食事をし旅行にいき、2年間の研究室生活の思い出がたくさんできました。深く感謝申し上げます。

研究室の後輩の皆様には、多くの学びと思い出を頂きました。後輩方が優秀であるため、常にモチベーションを維持でき、研究にしっかり向き合おうという気持ちになりました。学外発表においては常に後輩と共に発表しており、論文の添削、発表スライドの添削など尽力いただきました。共に発表した後輩がいたからこそ、学外発表はそれら全ての過程を通して大きく成長できました。また、研究外的生活においてもたくさんの思い出ができました。食事の思い出や、研究室での旅行など2年間の多くの思い出が後輩と共に過ごした日々にあります。深く感謝申し上げます。

最後に、普段の私生活において母に支えていただきました。ここまで学生を続けて、研究に没頭できたのは偏に母のおかげです。深く感謝申し上げます。



## 参考文献

- [1] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-Performance, High-Extensibility and High-Portability APR System, in *Proceedings of Asia-Pacific Software Engineering Conference*, pp. 697–698 (2018).
- [2] Martinez, M. and Monperrus, M.: ASTOR: A Program Repair Library for Java, in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 441–444 (2016).
- [3] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T. F.: TBar: Revisiting Template-Based Automated Program Repair, in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 31–42 (2019).
- [4] Sobania, D., Briesch, M., Hanna, C. and Petke, J.: An Analysis of Automatic Bug Fixing Performance of ChatGPT, in *Proceedings of International Workshop on Automated Program Repair*, pp. 23–30 (2023).
- [5] Xia, C. S., Wei, Y. and Zhang, L.: Automated Program Repair in the Era of Large Pre-trained Language Models, in *Proceedings of International Conference on Software Engineering*, pp. 1482–1494 (2023).
- [6] Jiang, N., Liu, K., Lutellier, T. and Tan, L.: Impact of Code Language Models on Automated Program Repair, in *Proceedings of International Conference on Software Engineering*, pp. 1430–1442 (2023).
- [7] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J. and Schmidt, D. C.: A prompt pattern catalog to enhance prompt engineering with chatgpt, *arXiv*, p. arXiv:2302.11382 (2023).
- [8] Parasaram, N., Yan, H., Yang, B., Flahy, Z., Qudsi, A., Ziaber, D., Barr, E. and Mechtaev, S.: The Fact Selection Problem in LLM-Based Program Repair, *ArXiv*, p. arXiv:2404.05520 (2024).
- [9] Cates, R., Yunik, N. and Feitelson, D. G.: Does Code Structure Affect Comprehension? On Using and Naming Intermediate Variables, in *Proceedings of International Conference on Program Comprehension*, pp. 118–126 (2021).
- [10] Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T. and Beigl, M.: Descriptive Compound Identifier Names Improve Source Code Comprehension, in *Proceedings of Conference on Program Comprehension*, pp. 31–40 (2018).

- [11] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language Models are Few-Shot Learners, in *Proceedings of International Conference on Neural Information Processing Systems*, pp. 1877–1901 (2020).
- [12] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *ArXiv*, p. arXiv:1810.04805 (2018).
- [13] Ahmed, T. and Devanbu, P.: Few-shot training LLMs for project-specific code-summarization, in *Proceedings of International Conference on Automated Software Engineering* (2023).
- [14] Ahmed, T., Pai, K. S., Devanbu, P. and Barr, E.: Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization), in *Proceedings International Conference on Software Engineering* (2024).
- [15] OuYang, S., Zhang, J., Harman, M. and Wang, M.: LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation, *ArXiv*, p. arXiv:2308.02828 (2023).
- [16] Joshi, H., Sanchez, J. C., Gulwani, S., Le, V., Radiček, I. and Verbruggen, G.: Repair is nearly generation: multilingual program repair with LLMs, in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence* (2023).
- [17] Widyasari, R., Ang, J. W., Nguyen, T. G., Sharma, N. and Lo, D.: Demystifying Faulty Code with LLM: Step-by-Step Reasoning for Explainable Fault Localization, in *Proceedings of International Conference on Software Analysis, Evolution and Reengineering*, pp. 568–579 (2024).
- [18] Yang, A. Z. H., Le Goues, C., Martins, R. and Hellendoorn, V.: Large Language Models for Test-Free Fault Localization, in *Proceedings of Internatinal Conference on Software Engineering* (2024).
- [19] Jesse, K., Ahmed, T., Devanbu, P. T. and Morgan, E.: Large Language Models and Simple, Stupid Bugs, in *Proceedings of International Conference on Mining Software Repositories*, pp. 563–575 (2023).
- [20] Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair, *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72 (2012).
- [21] Jha, S., Gulwani, S., Seshia, S. A. and Tiwari, A.: Oracle-guided component-based program

- synthesis, in *Proceedings of International Conference on Software Engineering*, pp. 215–224 (2010).
- [22] Mirzadeh, I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S. and Farajtabar, M.: GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models, p. arXiv:2410.05229 (2024).
- [23] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N. and Svyatkovskiy, A.: InferFix: End-to-End Program Repair with LLMs, in *proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1646–1656 (2023).
- [24] White, J., Hays, S., Fu, Q., Spencer-Smith, J. and Schmidt, D. C.: ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design, in *Generative AI for Effective Software Development*, pp. 71–108 (2024).
- [25] Huang, K., Meng, X., Zhang, J., Liu, Y., Wang, W., Li, S. and Zhang, Y.: An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair, in *Proceedings of International Conference on Automated Software Engineering (ASE)*, pp. 1162–1174 (2023).
- [26] Lin, D., Koppel, J., Chen, A. and Solar-Lezama, A.: QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge, in *Proceedings of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pp. 55–56 (2017).
- [27] Ye, H., Martinez, M., Durieux, T. and Monperrus, M.: A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark, in *Proceedings of International Workshop on Intelligent Bug Fixing*, pp. 1–10 (2019).
- [28] Alsuhaibani, R. S., Newman, C. D., Decker, M. J., Collard, M. L. and Maletic, J. I.: On the Naming of Methods: A Survey of Professional Developers, in *Proceedings of International Conference on Software Engineering*, pp. 587–599 (2021).
- [29] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).
- [30] Lawrie, D., Morrell, C., Feild, H. and Binkley, D.: What’s in a Name? A Study of Identifiers, in *Proceedings of International Conference on Program Comprehension*, pp. 3–12 (2006).