修士学位論文

題目

コーディングスタイルがLLM-APRの精度に与える影響の調査

指導教員
肥後 芳樹 教授

報告者
高田 智生

令和8年2月2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 ソフトウェア工学講座

令和 7 年度 修士学位論文

コーディングスタイルが LLM-APR の精度に与える影響の調査

高田 智生

内容梗概

人間が介在することなくプログラムのバグを修正する技術である自動プログラム修正（Automated Program Repair: APR）について，近年では大規模言語モデル（Large Language Model: LLM）を用いた LLM-APR の手法が提案されている．また，より高精度な LLM-APR の実現に向けて，入力として与えるプロンプトの改善やコードそのものが LLM のコード関連タスクの性能に与える影響も研究されている．一方で現在のソースコードは主に人間にとっての可読性を目的としたコーディングスタイルで記述されており，LLM-APR を行う上で適しているコードであるとは限らない．そこで本研究では人間にとって可読性を高めると知られているコーディングスタイルを対象に，それらのコーディングスタイルが LLM-APR の精度にどのような影響を与えるのか調査を行った．

調査の結果，変数宣言位置をクラスやメソッドの上部にまとめる書き換えやインデントおよび改行を削除する書き換えを行うことで実バグに対する LLM-APR の修正成功率がわずかに向上し，人間にとって可読性が高いとされる変数名などの情報のコード上での配置が，必ずしも LLM-APR に適しているとは限らないことを確認した．また，変数名のマスキングは LLM に対して，変数名の情報に基づいて偏った箇所に対して修正行うのではなくプログラム全体の文脈から均等にバグを特定して修正を行うように促すことを確認した．一方で変数のマスキングは，変数宣言位置の移動などの他の書き換えを行った一般的でないコーディングスタイルのコードに対して，性能を悪化させることを確認した．

また，これらの結果を踏まえて，変数宣言位置の移動とインデント・改行を削除する書き換えを行うことで修正成功率が約 7%改善し，さらに LLM-APR のコストにおいても入力トークン数を約 12%削減し，書き換えがより高い精度の LLM-APR を低コストで実現することを確認した．

主な用語

自動プログラム修正
大規模言語モデル

コーディングスタイル

目 次

1　はじめに

自動プログラム修正（Automated Program Repair: APR）は人間が介在することなくプログラムのバグを修正する技術である [8]．APR はソフトウェア開発においてデバッグに割かれる時間やコストを削減し，効率的で信頼性の高いソフトウェアの開発・保守を行うために必要である [33].

近年では大規模言語モデル（Large Language Model: LLM）が登場し，APR をはじめとした様々なコード関連タスクにおいて，LLM が従来の手法と比較してより高い修正能力を示すことが報告されている [7]．LLM を用いた APR（LLM-APR）も活発に研究されており，より高精度な LLM-APR の実現に向けて様々な手法が提案されている [50]．また，LLM-APR をはじめとしたコード関連タスクに対する LLM の性能を引き出すための研究としてプロンプトの改善が行われており，設計や追加情報の工夫により性能の向上がみられることが示されている [24][26].

また，LLM-APR をはじめとしたコード関連タスクにおける性能向上の要因は，プロンプト設計に限られないことも指摘されている．タスクの対象として LLM に与えるコードそのものが性能に与える影響も研究されており，例えばコード補完タスクにおいては入力トークン数の削減を目的にインデントや改行を削除した場合でも性能が変化しないことが報告されている [32]．LLM のコード関連タスク性能が変数名やメソッド名に依存することも示されている [43].

一方で，ソースコードの品質を保証する重要な指標には人間の理解の容易さとしての可読性が用いられており，実際に GitHub のプルリクエストにも可読性を向上させるための変更が多く含まれている [3][6]．また，可読性の向上を目的とした静的解析ツールも提案され，エディターの機能として普及しているツールも多い [16]．このように，現在のコーディングスタイルは，人間による保守を前提として，人間にとっての可読性を重視して設計されている．一方で，コードを自然言語と同様にトークン列として処理する LLM にとって，現在のコーディングスタイルが最適であるとは限らない.

本研究は，より精度が高く効果的な LLM-APR 技術の活用を目的に，人間にとっての可読性の向上を目的とする現在のコーディングスタイルが LLM-APR にも適しているのかについて調査を行った．コーディングスタイルを次の 2 種類に分類し，それぞれの形式で実際に普及しているいくつかのコーディングスタイルを対象に，LLM-APR においてどのようなコーディングスタイルが適しているのか考察を行った.

- コーディング規約等のコードを書く際に従うべき方針となる，コーディングにおける留意事項としてのコーディングスタイル

- エディターの補助ツール等として実装されている，可読性向上に貢献する自動的な静的解析や書き換え機能で扱われるコーディングスタイル

また調査結果を踏まえて，既存のコードを LLM-APR に適した形へ書き換えることで，修正成功率が約 7%向上し，入力トークン数を約 12%削減できることを確認した.

2　背景

本節では，本研究の背景として自動プログラム修正，大規模言語モデル，プログラムの可読性について説明する．

2.1　自動プログラム修正

自動プログラム修正（APR）は人間が介在することなくプログラムのバグを修正する技術であり，デバッグを支援するためのツールとして様々な手法やツールが公開されている [8]．

デバッグはソフトウェア開発において多くの時間やコストを要する工程である [33]．APR はデバッグに必要なコストを削減し，開発プロセスの迅速化することを目的としている．また，信頼性が高く高品質なソフトウェアを効率的に開発・保守するために期待される技術でもある [8]．

2.1.1　従来の APR 手法

従来の APR 手法はテストを使用し，テストに通過する修正を見つけるように設計されている [13]．例としては，修正空間を探索してテストで通過する修正を見つける探索ベースの手法 [48] や，修正パッチのテンプレートを用いて修正を行うテンプレートベースの手法 [22]，プログラムの意味的制約や仕様条件を満たす修正を探し出す制約ベースの手法 [27] などが挙げられる [14]．

しかし，従来の APR 手法は特定のパターンや既存のコードの再利用での修正が可能な一部のバグには有効であるものの，複雑なバグや多様なバグの修正は難しい [13][1]．また大規模なプログラムへの適用や計算効率の面でも，実用化には課題が残されている [23]．

2.1.2　近年の APR 手法

近年ではニューラルネットワークを活用して修正パターンを学習する学習ベースの手法が多く研究されており [49]，特に LLM を用いた APR 手法が注目されている [47]．これらの APR 手法はコンテキストに基づいてコードを生成できるため，従来の手法では修正困難な複雑なバグに対しても対処できる可能性が示されている [40]．また，これらの手法はテストを必要としない場合が多いため，従来の APR 手法の適用が困難であったテストが不十分な状況においても，有効なアプローチである。[47]．

2.1.3 Defects4J

Defects4J[15] は Java プログラムにおける実際のバグを収集した代表的なデータセットである．実際のオープンソースプロジェクトから収集されたバグを含むプログラムが含まれており，そのバグに対応するテストと実際に行われた修正のパッチが含まれている．

また，Defects4J は代表的な Java のオープンソースプロジェクトから収集されたコードから構成されており，活発に開発・保守されている高品質なコードのデータセットである．

このような特徴から APR の研究において，Defects4J は性能を評価するためのベンチマークとして広く用いられている [49].

2.2 大規模言語モデル

大規模言語モデル（LLM）は大量のテキストデータを用いて次のトークンを予測するという自己教師あり学習によって訓練された言語モデルであり，自然言語の理解や生成を高精度に行うことが可能である [51]．代表的な LLM には，OpenAI が開発した GPT[34]，Meta が開発した Llama[11]，Google が開発した Gemeni[41] などが挙げられる．これらの LLM は高い会話能力を有しており，社会全体においても様々なサービスとして普及しつつある [21][36].

また，LLM は自然言語処理分野に留まらずプログラミング支援やソフトウェア工学といった分野においても活用が進んでいる [7].

2.2.1 GPT

GPT（Generative Pre-trained Transformer）は OpenAI によって開発された大規模言語モデルの系列であり，文脈中の過去のトークン列を条件として次のトークンを生成する形式のモデルにより自然な文章生成が可能である [34]．GPT の特徴は二段階学習であり，大規模な自然言語のデータを用いた事前学習（Pre-training）により汎用化能力を持つとともに，微調整（fine-tuning）により特定タスクに特化したモデルへの応用が可能である [31]．これにより対話に特化したモデルである ChatGPT[25] や，コード特化型モデルである Codex[45] といった，用途ごとに特化したモデルが展開されている．

GPT はモデルの世代に伴ってパラメータ数や学習データ量が拡大され，より高度な言語理解および生成能力を獲得している [51]．2025 年 8 月に公開された GPT-5 は従来のモデルを大きく上回る性能を持つことが報告されている [37].

また，GPT は自然言語タスクのみならずソースコードを含むテキストデータを学習している．これにより GPT はコードの理解や生成といったコード関連タスクにおいても高い能力を有し，自然言語とプログラムを統合的に扱うことが可能である [46].

2.2.2 LLM によるコード関連タスク

LLM は自然言語と同様にコードを扱うことが可能であり，コード生成やコードの要約といったコード関連タスクにおいても，LLM が高い性能を発揮することが示されている [5][39].

また，コード関連タスクの性能向上を目的とする研究として，プロンプトの改善が行われている．段階的思考の明示を要求する Chain-of-Thought[24] や少数の例を与える few-Shot[26] などで，指示や追加情報の工夫が有効であることが示されている．

2.2.3 LLM のコード関連タスク性能とコード

LLM に入力として与えるコードそのものがコード関連タスクの性能に与える影響についても研究が行われている．例えばコード補完タスクにおいて，入力トークン数の削減を目的にインデントや改行を削除した場合でも性能が変化しないことが示されている [32]．一方で，変数名やメソッド名の命名 [43] や，コードの複雑さ [35] といったコーディングスタイルやコードメトリクスによって，LLM のタスク性能が変化することも報告されている．

2.2.4 LLM を用いた APR

APR においても LLM を用いた手法（LLM-APR）が活発に研究されている [50]．LLM-APR に使用される LLM は OpenAI の GPT 系列のモデルが用いられることが多く [50]，より有効な LLM-APR に向けた戦略として，タスク固有の学習を用いた追加の学習を行う fine-tuning[18] や少数の例を示す few-shot[28]，タスクの説明のみを示す zero-shot[44] などが採用されることも多い．近年ではより大規模な基盤を持つ LLM の登場により，例を示すことなく修正を指示する zero-shot が用いられることが多くなりつつある [20].

また，近年では LLM が環境の認識，現在の状態の推論，修正を達成するための一連の行動を自律的に行うエージェントとして使用される LLM-APR も提案されている [50]．代表的な研究としては，ChatGPT[25] に反復的なデバッグ行動を管理するステートマシンを統合し，自律的に修正を命令する RepairAgent が挙げられる [2].

2.3 プログラムの可読性

プログラムの可読性はソースコードを人間がどれだけ容易に理解することができるのかという性質であり，コードの意味や構造全体の理解と関連するため，ソフトウェアの保守性や品質に影響する [3]．可読性向上に貢献するものとして，コーディング規約やフォーマットや命名規則などのスタイルを統一する静的解析ツールが研究されている [19][29].

コードの可読性を高めることはソフトウェアの品質を保証する上で重要な役割を持ち，実際に GitHub のプルリクエストにも可読性を向上させるためのコードの変更が多く含まれて

いる [6]．また，可読性の向上を目的とした静的解析ツールも提案され，エディターの機能として普及しているツールも多い [16]．

3　調査

本研究では，人間にとっての可読性を向上させるためのコーディングスタイルがLLM-APRにとっても同様に適したコーディングスタイルであるか，ということを明らかにするための調査を行った．現在のコーディングスタイルは人間の可読性を高める目的で設計されており，LLM-APRにとって適したコーディングスタイルであるとは限らない．より効果的なLLM-APR技術の活用に向けLLM-APRに適したコーディングスタイルの特定が必要であり，本研究ではその足掛かりとして現在のコーディングスタイルがLLM-APRに適しているのか検証を行う．

なお，本研究においてLLMはGPT-5.2を使用した．GPT-5.2はLLM-APRの分野で最も頻繁に使用されるGPT系列モデルの2026年1月時点での最新モデルであり，かつ同様に最新のモデルであるGPT-5.2proと比較して高コストではないモデルである．

また，本研究で調査の対象とするコーディングスタイルを考えるにあたり，現在普及しているコーディングスタイルをコードに適用される状況により2つに分類し，調査を行う．

コーディングにおける留意事項としてのコーディングスタイル

変数名のコーディング規約等のコードを書く際に従うべき方針として与えられるコーディングスタイル．例えば，変数の定義と参照の位置を近づけることで可読性が向上すること [52] や，複雑な長い文を避けることも可読性の向上に貢献することが示されている [29]．実際にGoogle Style Guides[10] やSEC CERT C Coding Standard[38] などの複数のコーディング規約やスタイルガイドにおいて，変数宣言は最も狭いスコープで，最初の使用位置の近くで，変数の初期化は同時に行うべきであるとされている．また複雑な文についても，中間変数を用いた分解が必要であるとされている一方で，不要な一時変数も避けるべきであると述べられている．LLM-APRに適したコードと現在の一般的なスタイルのコードが一致するかを調べることで，LLM-APRを技術を活用するためにコードを記述する際にどのような規則に従うべきであるかを考察する．

自動的な静的解析や書き換え機能で扱われるコーディングスタイル

エディターの補助ツール等として普及している，可読性を向上させ，開発者の補助やプログラムの保守に貢献する機能によって実現されるコーディングスタイル．例えばインデントや改行といったレイアウトを適切に記述することや，変数名を適切に設定することも重要であることが示されている [4][17]．実際に多くの統合開発環境にはコーディングスタイルを整えるための機能が実装されている [42]．LLM-APRに適したコードの書き換えについて考えることで，LLM-APRを行う際の自動的な書き換えで性能が変化する可能性について考察する．

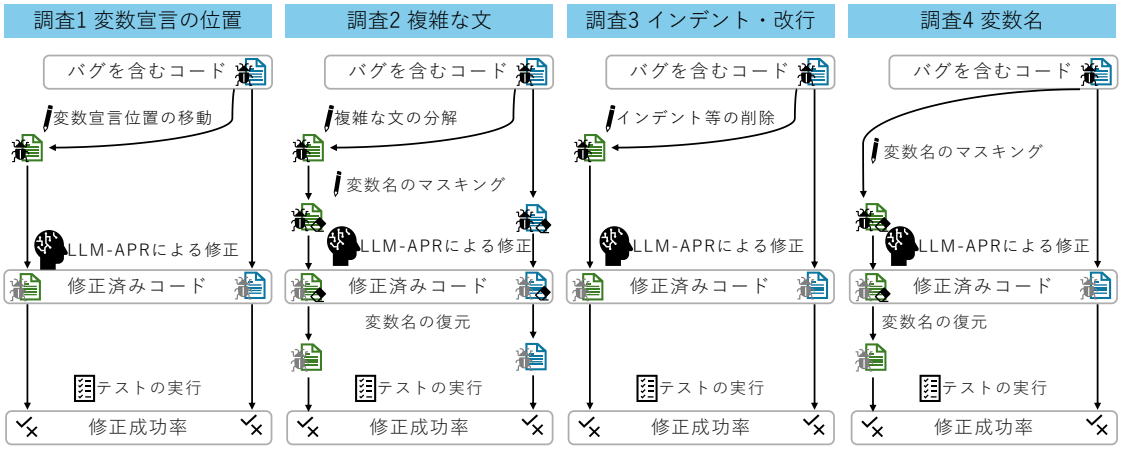

図 1: 4 つのコーディングスタイルが LLM-APR の精度に与える影響の調査

これらの 2 種類のコーディングスタイルからそれぞれ 2 つずつ，計 4 つの可読性に影響を与えるとされているコーディングスタイルの要素を対象に調査を行う．

また，調査結果に基づき，LLM-APR に適したコーディングスタイルへの書き換えにより LLM-APR の精度向上を試みる追加実験を行う．

3.1 調査の概要

コーディングスタイルが LLM-APR に与える影響の調査として，同じ機能を持ち異なる記述方法で書かれたコード組を対象に，LLM-APR を行った際の精度の違いを確認した．

調査は次の手順で行った．また，調査全体の手順を図 1 に示す．

1. バグを含むコードに対して書き換えを実施し，同じ機能を持ち，異なるコーディングスタイルで書かれたコード組を作成する
2. コード組のそれぞれのコードに LLM-APR による修正を行う
3. 修正後のコードに対してテストを実行し，その修正がもっともらしい修正[1]であるかどうかを基準として，修正の成否を判定する．

[1]仕様を満たすかどうかを考慮せず、与えられたテストスイートに対してすべて合格する修正

4. LLM の回答のランダム性を考慮し，LLM-APR による修正からテストによる判定までを 3 回繰り返す

5. 平均修正成功率に基づき，コード組のどちらのコーディングスタイルが LLM-APR に適しているのかを比較する

3.2 調査に用いるコード

LLM-APR の修正対象となるバグを含むコードとして，Defects4J をベースとする 2 種類のコードを用いた．ただし本研究では単一ファイルバグ[2]のみを対象としている．

実際のバグを含むコード

Defects4J に含まれる実際のバグを含むソースコードより，単一ファイルバグの実バグデータセットを抽出した．複数行にわたる修正を要するバグを含むデータセットを用い，実際のコーディングで LLM-APR を用いる場合の精度を計測した．

ミューテーションを実施したコード

Defects4J に含まれる修正済みのバグを含まないコードを使用し，ミューテーションを実施することで，人工バグデータセットを作成した．ランダムな箇所に生成した単純なバグを含む大規模なデータセットを用いることで，LLM がコード全体からバグを特定し，修正する能力を計測した．

3.3 調査の内容

コードを記述する際の留意事項として扱われる，次の 2 つのコーディングスタイルについて調査を行う．

- 変数の宣言から呼び出しまでの距離が短くなるように記述するコーディング
- 複雑な文を避け，単純な文で記述するコーディング

自動的な静的解析や書き換え機能で扱われる，次の 2 つのコーディングスタイルについて調査を行う．

- レイアウトを適切に記述したコーディング
- 変数名を適切に命名するコーディング

[2]プロジェクト中の 1 つのファイル変更のみで修正が可能なバグ

3.3.1 調査１：変数宣言の位置が与える影響

次の 2 つの記述方法の比較を行い，変数宣言の呼び出しからの距離が LLM-APR の精度に対してどのような影響を与えるのかを調査した．

- Defects4J に含まれるコード（例：ソースコード 1）．ただし，Defects4J に含まれるコードは活発に保守が行われているオープンソースのコードであり，全てのコードにおいて，変数宣言の位置が十分適切に設定されていることを確認した [15]
- 変数宣言が最初の呼び出しから遠くになるように，変数宣言文をクラスやメソッドの最初にまとめて行うように変更を加えたコード（例：ソースコード 2）

ソースコード 1: 変数宣言文を移動する前の Closure_91 中のメソッド

```
1  private JSDocInfo getFunctionJsDocInfo(Node n) {
2    JSDocInfo jsDoc = n.getJSDocInfo();
3    Node parent = n.getParent();
4    if (jsDoc == null) {
5      int parentType = parent.getType();
6      if (parentType == Token.NAME || parentType == Token.ASSIGN) {
7        jsDoc = parent.getJSDocInfo();
8        if (jsDoc == null && parentType == Token.NAME) {
9          Node gramps = parent.getParent();
10         if (gramps.getType() == Token.VAR) {
11           jsDoc = gramps.getJSDocInfo();
12         }
13       }
14     }
15   }
16   return jsDoc;
17 }
```

ソースコード 2: 変数宣言文を移動した Closure_91 中のメソッド

```
1  private JSDocInfo getFunctionJsDocInfo(Node n) {
2    JSDocInfo jsDoc;
3    Node parent;
4    int parentType;
5    Node gramps;
6    jsDoc = n.getJSDocInfo();
7    parent = n.getParent();
8    if (jsDoc == null) {
9      parentType = parent.getType();
10     if (parentType == Token.NAME || parentType == Token.ASSIGN) {
11       jsDoc = parent.getJSDocInfo();
```

```
12        if (jsDoc == null && parentType == Token.NAME) {
13          gramps = parent.getParent();
14          if (gramps.getType() == Token.VAR) {
15            jsDoc = gramps.getJSDocInfo();
16          }
17        }
18      }
19    }
20    return jsDoc;
21  }
```

例えばソースコード 1 では変数 *parentType* が 6 行目の呼び出しの直前である 5 行目に，*gramps* が 10 行目の呼び出しの直前である 9 行目に宣言され，初期値の代入が行われている．これを書き換えることで，ソースコード 2 のようにメソッドの最初である 5,6 行目に変数宣言を行い，初期値の代入は元のコードと同じ位置で行う．また，ソースコード 1 で 2，3 行目に行われている変数 *jsDoc*，*parent* についても変数宣言を分離し，メソッドの最初に移動する．

このような書き換えにより，ソースコード 2 ではメソッド全体で使用する変数の宣言がメソッドの冒頭でまとめて行われる．

3.3.2 調査２：複雑な文が与える影響

次の 2 つの記述方法の比較を行い，コードに含まれる複雑な文が LLM-APR の精度に対してどのような影響を与えるのかを調査した．

ただし，文の分解により生成される一時変数の識別子名が LLM-APR の精度に与える影響を考慮し，LLM-APR の前後に変数名のマスキングと復元を行った．また，プロンプトにおいても変数名を変更しないように指示する旨を追加した．

- Defects4J に含まれる複雑な文を含むコード（例：ソースコード 3）
- 複雑な文をすべて単純な式に分解したコード（例：ソースコード 4）

ソースコード 3: 複雑な文を分解する前の Math_51 の一部

```
1 if ((x >= -threshold) && (x <= threshold)) {
2   double inverse = 1 / FastMath.sqrt(y * y + z * z);
3   return new Vector3D(0, inverse * z, -inverse * y);
4 } else if ((y >= -threshold) && (y <= threshold)) {
```

ソースコード 4: 複雑な文を分解した Math_51 の一部

```
1 double $282 = threshold;
```

```
2  double $167 = -$282;
3  boolean $41 = (x >= $167);
4  double $168 = threshold;
5  boolean $42 = (x <= $168);
6  double $283 = threshold;
7  double $169 = -$283;
8  boolean $45 = (y >= $169);
9  double $170 = threshold;
10 boolean $46 = (y <= $170);
11 if ($41 && $42) {
12   double $284 = y * y;
13   double $285 = z * z;
14   double $171 = $284 + $285;
15   double $43 = FastMath.sqrt($171);
16   double inverse = 1 / $43;
17   double $286 = inverse;
18   double $172 = $286 * z;
19   double $351 = inverse;
20   double $287 = -$351;
21   double $173 = $287 * y;
22   org.apache.commons.math.geometry.Vector3D $44 = new Vector3D(0,
         $172, $173);
23   return $443;
24 } else if ($45 && $46) {
```

例えば，ソースコード 3 の 1 行目の条件式 $(x >= -threshold)\&\&(x <= threshold)$ や $1/FastMath.sqrt(y*y+z*z)$ は複雑であり，一時変数を用いて分解される．これを次のような 1 つの操作の要素ごとに分解し，ソースコード 4 のそれぞれ 1 行目から 5 行目や 12 行目から 16 行目のような形に書き換えを行う．

- 変数の参照
- 変数の正負の入れ替え
- 比較演算の結果を boolean 変数に代入
- メソッド呼び出し

3.3.3 調査３：レイアウトが与える影響

次の 2 つの記述方法の比較を行い，コードに含まれるインデント・改行が LLM-APR の精度に対してどのような影響を与えるのかを調査した．

- Defects4J に含まれるコード（例：ソースコード 5）．ただし，Defects4J に含まれるコードは活発に保守が行われているオープンソースのコードであり，全てのコードにおいて，インデントや改行が十分適切に設定されていることを確認した [15].
- インデントと改行を全て削除したコード（例：ソースコード 6）

ソースコード 5: インデントと改行を削除する前の Jsoup_5 の一部

```
1 package org.jsoup.parser;
2
3 import org.jsoup.helper.Validate;
4 import org.jsoup.nodes.*;
5
6 import java.util.LinkedList;
7
8 public class Parser {
9   private static final String SQ = "'";
```

ソースコード 6: インデントと改行を削除した Jsoup_5 の一部

```
1 package org.jsoup.parser; import org.jsoup.helper.Validate; import
    org.jsoup.nodes.*; import java.util.LinkedList; public class
    Parser { private static final String SQ = "'";
```

ソースコード 6 のように，インデント・改行を削除することで 1 行のコードへの書き換えを行う．

3.3.4 調査 4：変数名の命名が与える影響

次の 2 つの記述方法の比較を行い，変数名が LLM-APR の精度に対してどのような影響を与えるのかを調査した．また，プロンプトにおいても変数名を変更しないように指示する旨を追加した．

- Defects4J に含まれるコード（ソースコード 7）．ただし，Defects4J に含まれるコードは活発に保守が行われているオープンソースのコードであり，全てのコードにおいて，変数名が十分適切に命名されていることを確認した [15].
- 変数名を全てマスキングしたコード（ソースコード 8）

ソースコード 7: 変数名をマスキングする前の Lang_23 の一部

```
1 fmtCount++;
2 seekNonWs(pattern, pos);
```

```
3  int start = pos.getIndex();
4  int index = readArgumentIndex(#blue#pattern#>, next(pos));
5  stripCustom.append(START_FE).append(index);
6  seekNonWs(#blue#pattern#>, pos);
7  Format format = null;
8  String formatDescription = null;
9  if (c[pos.getIndex()] == START_FMT) {
10   formatDescription = parseFormatDescription(#blue#pattern#>,next(pos));
11   format = getFormat(formatDescription);
12   if (format == null) {
13     stripCustom.append(START_FMT).append(formatDescription);
14   }
15 }
```

ソースコード 8: 変数名をマスキングした Lang_23 の一部

```
1  LOC_VAR5++;
2  seekNonWs(MET_PAR0, LOC_VAR3);
3  int LOC_VAR6 = LOC_VAR3.getIndex();
4  int LOC_VAR7 = readArgumentIndex(MET_PAR0, next(LOC_VAR3));
5  LOC_VAR2.append(FLD_VAR6).append(LOC_VAR7);
6  seekNonWs(MET_PAR0, LOC_VAR3);
7  Format LOC_VAR8 = null;
8  String LOC_VAR9 = null;
9  if (LOC_VAR4[LOC_VAR3.getIndex()] == FLD_VAR4) {
10   LOC_VAR9 = parseFormatDescriptionMET_PAR0,next(LOC_VAR3));
11   LOC_VAR8 = getFormat(LOC_VAR9);
12   if (LOC_VAR8 == null) {
13     LOC_VAR2.append(FLD_VAR4).append(LOC_VAR9);
14   }
15 }
```

ソースコード 8 のように，マスキングにより全ての変数名を別の名前に置き換えたコードを作成する．

4　調査の準備

4.1　実バグデータセットの作成・テストの生成

テストの生成には Java コードのテストケースの自動生成を行う EvoSuite[9] を用いた．Defects4J の実バグを含むコードにはそのバグを検出するテストが付属しているが [15]，ミューテーションによる様々なバグにテストに依存することなく対応するために EvoSuite によって生成したテストを採用した．

実バグデータセットは Defects4J に含まれる単一ファイルバグ 702 件のうち，次の条件を満たす 107 件のバグに対応する，バグを含むコードのファイルを抽出した．

- EvoSuite によるテストの生成に成功したコード組
- 生成されたテストが不安定なテストを含まず[3]，かつ実際のバグを検出できたコード組
- GPT-5.2 にプログラム修正の指示を安全行うことができる長さで，ファイル全体の文字数が 15,000 文字以内のコード

また LLM が，自然言語の説明等に依らずに，コードそのものを理解・修正する能力を評価するために，コードに含まれるコメントは Javadoc 等も含めて全て削除した．

4.2　人工バグデータセットの作成

人工バグデータセットは 604 件のバグを含み，実バグデータセット 107 件に対応する修正済みコードのファイルに対して Mutanerator[4]を用いたミューテーションを行うことで作成した．Mutanerator は次のような操作により，単純なバグをランダムに 1 つ発生させる．

- 二項演算子の入れ替え
- インクリメント・デクリメントの入れ替え
- 正負の入れ替え
- void 型メソッドの呼び出しの削除

各ファイルに対して 10 通りのミューテーションを生成した 10 個の人工バグを生成した．ただし，ミューテーションが可能な箇所が十分でないコードに対して生成される人工バグは 10 個未満である．また，生成した人工バグのうち EvoSuite のテストによる検出が不可能であったものを除外し，人工バグデータセットはテストで検出可能なバグのみで構成した．

[3]テストを 30 回実行し，テスト結果が変化しないコード組のみを選択した
[4]https://github.com/kusumotolab/Mutanerator

4.3 調査に使用する書き換えデータセットの作成

調査に使用する書き換えデータセットは，作成した実バグデータセットと人工バグデータセットをさらに書き換えることで作成した．また書き換え前後でプログラムの挙動が変化していないことを確認するために，調査に使用する全てのコードが次の 2 つの条件を満たすことを確認した．

- バグを含むコードに対して書き換えを行ったコードが，コンパイル可能でテストに不合格すること
- 同様の書き換えツールやパッチを用いて書き換えたバグを含まないコードが，コンパイル可能でテストにテストに合格すること

4.3.1 調査１：変数宣言の位置を移動する書き換え

実バグデータセットに対してソースコード 2 のような書き換えを手作業で実施した．

書き換えは変数宣言のメソッドやクラスの最上部への移動により行った．ただし，移動後の変数宣言文は元のコードの変数宣言文と同じ順番となるように並べる．また，変数宣言文と代入文が 1 行にまとめられている場合には文を分離し，変数宣言文のみを移動した．

ただし，手作業での書き換えとなるため，実バグデータセットから 40 件をランダムに選択して比較を行った．また，人工バグセットも実バグデータセットと同じ 40 件に対応する修正済みコードから生成されたもののみを使用した．

4.3.2 調査２：複雑な文を分解する書き換え

JCodeFlatter[12] を用いて，ソースコード 1 行あたりの意味量を均一にする書き換えを行った．書き換えはソースコードを抽象構文木上で行い，メソッド呼び出しや二項を含む文を対象に，1 つの文あたり 1 つの操作のみが対応するまで一時変数に分解する．

ただし一部のコードについては，ソースコード 9 からソースコード 10 への書き換えのように，コンパイル失敗やプログラムの挙動の変化が発生する．例えば，&& で結合された条件式について，左辺で変数の値が適切であるかの確認を行い，右辺でその変数を用いたメソッド呼び出しや配列の参照を行っている場合にエラーが発生する．ソースコード 9 の場合，&& の左辺で変数 $columnIndex$ の値が適切でない際には右辺は実行されない一方で，ソースコード 10 では変数 $columnIndex$ の値にかかわらずメソッド $getObject(columIndex)$ の呼び出しが実行される．

ソースコード 9: 複雑な文を分解する前の Chart_22 の一部

```
1 if (columnIndex >= 0 && row.getObject(columnIndex) != null) {
```

```
2   allNull = false;
3   break;
4 }
```

ソースコード 10: 複雑な文を分解した Chart_22 の一部

```
1 int $147 = columnIndex;
2 boolean $58 = $147 >= 0;
3 int $171 = columnIndex;
4 org.jfree.data.KeyedObjects $172 = row;
5 java.lang.Object $148 = $172.getObject($171);
6 boolean $59 = $148 != null;
7 if ($58 && $59) {
8   allNull = false;
9   break;
10 }
```

ツールを用いた文の分解後にコンパイルに失敗するものやプログラムを実行した際の挙動が変化したものは除き，41 件のみを用いて比較を行った．また，人工バグセットも実バグデータセットと同じ 40 件に対応する修正済みコードから生成されたもののみを使用した．

4.3.3 調査３：インデント・改行を削除する書き換え

インデントと改行を削除するプログラムを実装し，書き換えを行った．書き換えは改行やインデントを削除しつつ，コンパイル可能なコードを生成するために，次の書き換えを行った．

1. 各行の開始から空白文字以外の文字までを全て削除する
2. 空行を削除する
3. 残った行を空白文字つなぎで連結し，1 行のコードを作成する

4.3.4 調査４：変数名をマスキングする書き換え

実装した変数名のマスキングツールを用いて，マスキングを実施した．ただしマスキング後の変数名は次の 4 種類の変数にごとにインデックスを割り当て，VAR_5 のように変数の区分とインデックスを用いて命名する．

- ローカル変数 LOC_VAR
- フィールド変数 FLD_VAR

- メソッド引数 MET_PAR
- その他の引数 PAR

また，マスキングを行った変数名はテストの実行の際に復元する．復元はマスキング時に保存した元の変数名とマスキング後の変数名の対応に基づき，元の名前に書き換えることにより行った．

5　結果

5.1　調査１：変数宣言の位置が与える影響

変数宣言の位置を移動する書き換え前後での LLM-APR の修正成功率を表 1 に示す.

実バグに対する修正において，書き換えを行うことで修正成功率がわずかに向上した．一方で，人工バグ修正については修正成功率が低下し，修正後のコードがコンパイルエラーとなった件数も増加した.

表 1: 変数宣言の位置の移動による修正性能の変化

コード	実バグ		人工バグ	
	修正成功率	コンパイルエラー	修正成功率	コンパイルエラー
書き換え前	10.83 %	10.83 %	50.22 %	3.61 %
	4.33 /40 件	4.33 /40 件	116.00 /231 件	8.33 /231 件
書き換え後	11.67 %	11.67 %	48.20 %	5.05 %
	4.67 /40 件	4.67 /40 件	111.33 /231 件	11.67 /231 件

5.2　調査２：複雑な文が与える影響

複雑な文を分解する書き換え前後での LLM-APR の修正成功率を表 2 に示す.

実バグ，人工バグともに書き換えにより修正成功率が低下することを確認した.

表 2: 複雑な文の分解による修正性能の変化

コード	実バグ		人工バグ	
	修正成功率	コンパイルエラー	修正成功率	コンパイルエラー
書き換え前	11.38 %	8.13 %	65.23 %	6.32 %
	4.67 /41 件	3.33 /41 件	151.33 /232 件	14.67 /232 件
書き換え後	8.94 %	7.32 %	59.34 %	7.04 %
	3.67 /41 件	3.00 /41 件	137.67 /232 件	16.33 /232 件

5.3　調査３：レイアウトが与える影響

インデント・改行を削除する書き換え前後での LLM-APR の修正成功率を表 3 に示す.

実バグに対する修正において，書き換えを行うことで修正成功率がわずかに向上した．一方で，人工バグ修正については修正成功率が変化しなかった.

5.4 調査４：変数名の命名が与える影響

変数名をマスキングする前後での LLM-APR の修正成功率を表 4 に示す.

実バグに対する修正において，書き換えの前後で修正成功率が変化しなかった．一方で，人工バグにおいてはマスキングにより修正成功率が成功したものの，コンパイルエラーの発生件数は約 2 倍に増加することを確認した.

表 3: インデント・改行の削除による修正性能の変化

コード	実バグ		人工バグ	
	修正成功率	コンパイルエラー	修正成功率	コンパイルエラー
書き換え前	14.64 %	10.59 %	59.99 %	3.75 %
	15.67 /107 件	11.33 /107 件	362.33 /604 件	22.67 /604 件
書き換え後	15.58 %	10.90 %	59.82 %	3.75 %
	16.67 /107 件	11.67 /107 件	361.33 /604 件	22.67 /604 件

6 考察

6.1 LLM-APR に適したコーディングスタイル

人間にとっての可読性の向上を目的とする現在の一般的なコーディングスタイルは，必ずしも LLM-APR に適したコーディングスタイルであるとは限らないことを確認した．

変数宣言に関して，一般的なコーディングスタイルでは変数は可能な限り狭いスコープで宣言し，最初の使用位置の近くで初期化と同時に記述することが推奨されている．一方で調査の結果から，LLM-APR においてはクラスやメソッドの先頭部において変数宣言をまとめて行うことで，修正精度が向上する傾向が確認された．

また，インデントや改行といったレイアウトについても，人間の可読性向上を目的とした一般的なコーディングスタイルとは異なり，これらを削除したコードを入力とした場合に LLM-APR の精度が向上する結果が得られた．この結果は，コード補完タスクにおけるインデントと改行の有無と LLM の性能との関係を調査した Pan らの研究 [32] の報告と整合的である．

この要因としては，変数宣言をクラスやメソッドの上部にまとめることで，変数名とその型に関する情報がコードの先頭付近に集約され，LLM がコード全体の変数を把握しやすくなった可能性が考えられる．LLM はトークン間の関係性に基づいて次トークンを予測するモデルであり，変数に関する情報が先に与えられることが，後続の修正済みコードを生成する上での精度に影響すると考えられる．

このように，人間にとって可読性が高いとされるコードにおける変数名などの情報配置が，必ずしも LLM-APR に適しているとは限らず，LLM-APR の精度向上に寄与するコーディングスタイルについて，さらなる調査が必要であると考えられる．

表 4: 変数名のマスキングによる修正性能の変化

コード	実バグ		人工バグ	
	修正成功率	コンパイルエラー	修正成功率	コンパイルエラー
書き換え前	14.64 %	10.59 %	59.99 %	3.75 %
	15.67 /107 件	11.33 /107 件	362.33 /604 件	22.67 /604 件
書き換え後	14.33 %	8.72 %	67.22 %	7.56 %
	15.33 /107 件	9.33 /107 件	406.00 /604 件	45.67 /604 件

6.2 単純なバグに対する LLM-APR

人工バグが Defects4J に含まれる実バグと比較して，規模が小さく修正が容易なバグであることを確認した．人工バグの修正成功率は実バグの修正成功率と比較して約 4 倍から 5 倍高く，実バグと比較して修正しやすい単純なバグであると考えられる．

また，LLM-APR の性能に影響するコーディングスタイルについても，単純な人工バグと複雑な実バグで異なる傾向が見られた．変数宣言の位置の書き換えやインデント・改行の削除により実バグの修正では修正成功率の改善が見られる一方で，人工バグについては修正成功率がほとんど変化しない，あるいはわずかに低下することを確認した．コンパイルエラー件数についても，変数宣言の位置の書き換えにより増加した．

これは人工バグがランダムな位置に発生した規模の小さいバッグであり，一般的でないコーディングスタイルでは，バグが見逃されるケースやバグのない箇所に変更を加えてコンパイルエラーを生じるケースが増加する可能性を示すものである．

6.3 変数名のマスキングが LLM-APR に与える影響

変数名のマスキングを行った場合，人工バグに対する修正成功率は約 7%向上した一方で，コンパイルエラー件数は約 2 倍に増加した．

変数名は LLM に対して変数の役割や意味を示す情報を提供する可能性を持つ情報であり，プログラムの動作理解やバグ位置の特定に寄与する可能性がある．そのため変数名をマスキングすることで，LLM が特定の識別子名に依存せず，プログラム全体の構造や文脈に基づいてバグを探索・修正するよう促された可能性が考えられる．この結果，識別子名による偏りを受けることなく，プログラム全体から均等にバグを探索できるようになり，ランダムな位置に存在する人工バグの修正成功率が向上した一方で，バグと無関係な箇所に変更を加えてしまう可能性も高まり，結果としてコンパイルエラー件数が増加したと推測される．

一方で，Wang らの研究 [43] では，適切に命名された変数名が LLM のコード関連タスクの性能向上に寄与することが報告されている．Wang らの研究では GPT 系列のモデルが使用されているが，論文中で使用された GPT のバージョンは明記されておらず，論文公開時点である 2023 年 7 月以前のモデルが用いられていると推測される．本研究で使用した GPT-5.2 は，パラメータ数および学習データ量が拡大されたモデルであるため，モデル規模の違いにより，変数名が LLM-APR の性能に与える影響が Wang らの研究とは異なる結果となった可能性がある．

7　書き換えによる LLM-APR 精度向上の試み

調査１から調査４を通して，コーディングスタイルが LLM-APR の精度に与える影響について，以下の点が確認された．

- 変数宣言位置の移動や，インデントおよび改行を削除する書き換えは，実バグの修正性能をわずかに改善する
- 変数名のマスキングにより LLM は識別子名による偏りを除き，コード全体から均等にバグを探索するよう促される
- 複雑な文を分解する書き換えは，LLM-APR の性能を低下させる．

これらの結果を踏まえ，本研究では，複数のコーディングスタイルを組み合わせた書き換えを行った場合に，LLM-APR の性能がどのように変化するのかを明らかにするための追加調査を行う．

7.1　調査５：書き換えによる LLM-APR の精度変化

7.1.1　調査の概要

本調査では，LLM-APR の精度向上に寄与すると考えられる複数のコーディングスタイルが同時に適用された場合における修正性能の変化を確認する．これにより，実際にこれらのコーディングスタイルを組み合わせて採用した際に生じる LLM-APR の精度変化について考察を行う．

本調査では実バグのみを対象とし，以下の 3 種類のコードに対して LLM-APR の性能を計測する．

- 書き換えを行わない元のコード
- 変数宣言位置の移動およびインデントと改行の削除を行ったコード
- 変数宣言位置の移動，インデントと改行の削除に加えて，変数名のマスキングを行ったコード

7.1.2　評価指標

本調査では，実際のコードに対して LLM-APR を適用した際の性能を評価するため，修正成功率およびコンパイルエラー発生件数に加えて，LLM-APR に要するコストの計測を行う．

GPT の利用コストは入力および出力のトークン数に依存するため [30]，本研究では LLM-APR のコスト指標として入力トークン数を計測する．トークン数の計測には，OpenAI が提供するトークナイザー[5]を使用する．このトークナイザーは，GPT 系列モデル内部でのトークナイズ処理と同等の機能を提供しており，本研究で使用する GPT-5.2 に対応するバージョンを用いる．なお，実際の LLM-APR では修正対象コードに加えてプロンプトも入力として与えられるが，本研究ではコードそのものと LLM-APR 性能との関係に着目するため，修正対象コードのみの入力トークン数を比較対象とする．

7.1.3 データセット

本調査では，調査 1 から調査 4 で使用したものと同一の実バグデータセットを使用する．ただし，本調査では変数宣言位置の移動を行う必要があるため，調査 1 で使用した実バグ 40 件のみを対象とする．

また，変数名のマスキングは調査 4 と同様のツールを用いて行う．ただし，トークン数削減によるコスト削減を目的として，マスキング後の変数名には以下の略称とインデックスを組み合わせた形式（例：$LV3$）を用いる．

- ローカル変数：LV
- フィールド変数：FV
- メソッド引数：MP
- その他の引数：P

7.2 結果と考察

3 種類のコードに対する LLM-APR の性能を計測した結果を表 5 に示す．変数宣言位置の移動およびインデントと改行の削除を行ったコードでは，元のコードと比較して修正成功率が約 7%向上し，入力トークン数を約 12%削減できることを確認した．

一方で，これらの書き換えに加えて変数名のマスキングを行った場合には LLM-APR の修正成功率が低下する結果となった．このことから，変数名のマスキングによる修正性能の改善は一般的な文脈を持つコードに含まれる単純なバグに対して有効であり，一般的なコードとは異なる文脈を持つ実バグの修正においては変数名のマスキングは必ずしも適切ではない可能性があると考えられる．また変数名のマスキングを行った場合，入力トークン数が増加する傾向も確認された．これはマスキング後の変数名が，例えばローカル変数が i のよう

[5] https://github.com/openai/tiktoken

な一文字の識別子であった場合でも，$LV5$ のように少なくとも 3 文字以上の名称に置き換えられるため，結果として変数名が全体的に長くなることに起因すると考えられる．

なお，入力トークン数とは異なり，出力トークン数についてはいずれの条件においても大きな変化は見られなかった．これは，LLM が出力する修正コードが，インデントや改行が整えられた形式で生成されることに起因すると考えられる．そのため出力トークン数の削減には，プロンプトにおける追加指示などさらなる工夫が必要であると考えられる．

表 5: コーディングスタイルと LLM-APR の精度

コード	修正成功率	コンパイルエラー	平均トークン数	
			入力	出力
元のコード	10.83 % 4.33 /40 件	10.83 % 4.33 /40 件	1204.3	1239.1
変数宣言位置の移動 インデントと改行の削除	17.50 % 7.00 /40 件	11.67 % 4.67 /40 件	1060.5	1223.3
+変数名のマスキング	9.17 % 3.67 /40 件	7.50 % 3.00 /40 件	1179.7	1341.5

8　妥当性の脅威

8.1　内的妥当性の脅威

8.1.1　LLM の回答のランダム性

GPT-5.2 をはじめとした LLM の回答は確率的に生成されるため，同一のコードとプロンプトを入力として与えた場合であっても実行毎に異なる修正結果が得られる可能性がある．本研究では LLM-APR は全て 3 回の施行を行い，結果は平均値を用いて評価することで LLM の回答のランダム性の影響の低減を図っている．

しかし LLM の回答のランダム性の影響を完全に排除することは難しく，本研究で計測した修正成功率やコンパイルエラー件数が LLM のランダム性の影響を受けている可能性は完全には否定されない．

8.1.2　GPT-5.2 と Defects4J の学習データ依存性

GPT-5.2 が Defects4J に含まれるバグの修正内容を学習済みである可能性を考慮し，LLM-APR によって生成された修正内容と，Defects4J における実際の修正内容との一致性について調査を行った．具体的には，書き換えを行っていない Defects4J の実バグに対して LLM-APR を 3 回実行し，少なくとも 1 回以上修正に成功したバグについて，生成された修正が実際の修正と一致するもの（same）と，相違するもの（diff）に分類した．分類結果を表 6 に示す．

その結果，修正に成功したバグの約 60%は，実際の修正内容とは異なる修正によってテストを通過していることが確認された．また，一致する修正についても，多くが単一トークンバグ[6]や単一行バグ[7]といった，修正内容が一意に定まりやすいバグであった．

これらの結果から，GPT-5.2 が Defects4J の修正内容を単純に再現していることを強く示す根拠は確認されず，本研究における評価結果が学習データの影響を強く受けているとは言い難い．ただし，LLM の学習データの詳細は公開されていないため，GPT-5.2 が Defects4J を学習済みである可能性や，学習データが本研究の評価に影響を与えている可能性を完全に否定することはできない．

[6] 1 つのトークンの変更のみで修正が可能なバグ
[7] 1 行の変更のみで修正が可能なバグ

表 6: 修正に成功したバグと実際の修正の比較

BugID	1	2	3	バグの種類
Chart21			diff	
Chart24	same	same	same	単一トークンバグ
Cli5	same	same	same	
Cli40	diff			単一行バグ
Codec18	diff	diff	diff	単一行バグ
Compress2	diff	diff		
Compress6		diff	diff	
Csv4	same	same	same	単一行バグ
Csv6	diff	diff	diff	
Csv11	same	same	same	単一行バグ
Gson12	same	same		
Gson17	diff	diff	diff	
Jsoup5	diff	diff	diff	
Jsoup16	diff	diff		
Jsoup26	diff	diff	diff	
Jsoup59	diff	diff	same	
Math35	same	same	same	同一の単一行バグのコードクローン
Math62			same	
Math101	diff	diff	diff	

8.2 外的妥当性の脅威

8.2.1 LLM・データセットへの依存

本研究ではLLM-APRを行うモデルとしてGPT-5.2，評価に用いるデータセットDefects4Jのみを用いた．そのため，本研究で得られた結果が他のGPT系列モデルや異なるLLMを用いたAPRや，他のデータセットや実際の開発におけるバグ修正においても同様であるとは限らない．

8.2.2 バグの規模・種類への依存

本研究で対象としているバグは，全て15,000文字以内の単一ファイルの実バグまたは人工バグであり，大規模なコードに含まれるバグや複数ファイルにまたがる修正を必要とするバグは含まれていない．そのため，本研究で得られた結果が実際の開発におけるバグ修正においても同様であるとは限らない．

9　おわりに

本研究では，人間にとっての可読性向上を目的とするコーディングスタイルが LLM-APR に対してどのような影響を与えるのかについて調査を行った．調査の結果，変数宣言をクラスやメソッドの上部にまとめるコーディングスタイルや，インデントおよび改行を含まないコーディングスタイルは，実バグに対する LLM-APR の修正成功率をわずかに改善することを確認した．また変数名をマスキングすることで，LLM が特定の識別子名に依存せずプログラム全体の構造や文脈に基づいてバグを探索・修正するように促され，単純なバグの修正成功率が向上した．一方で，変数名のマスキングはコンパイルエラーを増加させる要因でもあり，また，他の書き換えを行った一般的なコードとは異なる文脈を持つコードにおいては変数名のマスキングが LLM-APR の性能の悪化を招くことも確認した．

また，LLM-APR の精度向上に貢献する変数宣言位置の移動，インデントと改行の削除を行うことで，修正成功率が約 7%改善し，入力トークン数を約 12%削減できることを確認した．

以上の結果から本研究では，人間にとって一般的に可読性が高いとされるコーディングスタイルは必ずしも LLM-APR に適しているとは限らず，適切なコード中での情報の配置を行うコーディングや，LLM-APR に適した自動書き換えツールの導入により，より低コストで高精度な LLM-APR が実現できる可能性を示した．

ただし，本研究では可読性の向上を目的として一般的に用いられているコーディングスタイルのうち，変数宣言の位置，複雑な式と単純な式，インデントおよび改行の有無，変数名の有無といった限られた要素のみを対象として調査を行った．しかし，可読性の向上を目的としたコーディングスタイルやリファクタリング手法は他にも数多く存在しており，それらが LLM-APR に与える影響については十分に明らかになっていない．今後は，より多様なコーディングスタイルを対象とした調査を行うことで，LLM-APR に適したコードに対する理解を深める必要がある．

また，本研究の成果は一般的な可読性を重視したコーディングスタイルと LLM-APR に適したコーディングスタイルとの間に差異があることを確認し，適切なコーディングスタイルを選択することで LLM-APR の精度向上の可能性を示したにとどまる．実際のソフトウェア開発においては，人間による理解や保守性とのバランスを考慮する必要があり，LLM-APR に適したコーディングスタイルをそのまま適用することが常に適切であるとは限らない．加えて，本研究では限られたデータセットを対象とする評価を行っており，実際の開発環境での LLM-APR 性能の変化を調査するために，より大規模かつ多様なデータセットを用いた実験を通した検証を行う必要がある．

実際の開発環境において LLM-APR を有効に利用するために，さらなる調査を行うとと

もに，LLM-APR と人間の可読性の両者を考慮したコーディングスタイルや書き換え手法を検討することが今後の課題である．

10　謝辞

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 肥後 芳樹 教授には，ご多忙の中，研究活動に対して多くの御指導・御助言と共に，本研究の方針に関して様々な御指導を賜りました．心より深く感謝申し上げます．

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 松下 誠 准教授には，本研究の方向性の検討から研究活動の直接の御指導や論文の執筆に関する御助言と共に，日頃より御声援の御言葉を賜りました．3 年間に渡り，研究活動の全ての場面で手厚く御指導，御助言を賜り，心より深く感謝申し上げます．

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肥後研究室の皆様には，研究における様々な場面で助言をいただくと共に，研究室での日常生活で様々な時間を共にしました．心より深く感謝申し上げます．

最後に，研究活動をはじめとした研究室での日常生活において，多くのサポートを賜りました事務職員 軽部 瑞穂 氏に心より深く感謝を申し上げます．

参考文献

[1] Aldeida Aleti and Matias Martinez. E-apr: Mapping the effectiveness of automated program repair techniques. *Empirical Software Engineering*, Vol. 26, No. 5, p. 99, Jul 2021.

[2] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. ICSE '25, p. 2188–2200. IEEE Press, 2025.

[3] Raymond P.L. Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, Vol. 36, No. 4, pp. 546–558, jul 2010.

[4] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *16th Working Conference on Reverse Engineering*, 2009. Page(s): 31-35.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, and et al. Evaluating large language models trained on code. *ArXiv*, Vol. abs/2107.03374, , 2021.

[6] Carlos Eduardo C. Dantas, Adriano M. Rocha, and Marcelo A. Maia. How do developers improve code readability? an empirical study of pull requests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 110–122, 2023.

[7] Vítor Mateus de Brito and Kleinner Farias. Understanding the role of large language models in software engineering: Evidence from an industry survey, 2025.

[8] Sena Dikici and Turgay Tugay Bilgin. Advancements in automated program repair: a comprehensive review. *Knowledge and Information Systems*, Vol. 67, No. 6, pp. 4737–4783, Jun 2025.

[9] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. ESEC/FSE '11, p. 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Google. Google style guides. `https://github.com/google/styleguide/`, 2024.

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, and et al. The llama 3 herd of models, 2024.

[12] Yoshiki Higo and Shinji Kusumoto. Flattening code for metrics measurement and analysis. *In Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution*, pp. 494–498, 9 2017.

[13] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, p. 12–23, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. A survey on automated program repair techniques, 2023.

[15] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, p. 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[16] Thomas Kanoutas, Thomas Karanikiotis, and Andreas L. Symeonidis. Enhancing code readability through automated consistent formatting. *Electronics*, Vol. 13, No. 11, 2024.

[17] Thomas E. Kesler, Randy B. Uram, Ferial Magareh-Abed, Ann Fritzsche, Carl Amport, and H.E. Dunsmore. The effect of indentation on program comprehension. *International Journal of Man-Machine Studies*, Vol. 21, No. 5, pp. 415–428, 1984.

[18] Márk Lajkó, Viktor Csuvik, and László Vidács. Towards javascript program repair with generative pre-trained transformer (gpt-2). In *Proceedings of the Third International Workshop on Automated Program Repair*, APR '22, p. 61–68, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Taek Lee, Jung-Been Lee, and Hoh In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and Its Applications*, Vol. 7, pp. 413–422, 09 2013.

[20] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In Ruslan Mitkov and Galia Angelova, editors, *Proceedings of the 14th International*

Conference on Recent Advances in Natural Language Processing, pp. 641–647, Varna, Bulgaria, September 2023. INCOMA Ltd., Shoumen, Bulgaria.

[21] Weixin Liang, Yaohui Zhang, Mihai Codreanu, Jiayu Wang, Hancheng Cao, and James Zou. The widespread adoption of large language model-assisted writing across society. *Patterns*, Vol. 6, No. 12, p. 101366, 2025.

[22] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, p. 31–42, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. ICSE '20, p. 615–627, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Ren-Biao Liu, Anqi Li, Chaoding Yang, Hui Sun, and Ming Li. Revisiting chain-of-thought in code generation: Do language models need to learn reasoning before coding? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, Vol. 267 of *Proceedings of Machine Learning Research*, pp. 38809–38826. PMLR, 13–19 Jul 2025.

[25] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Lin Zhao, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, Vol. 1, No. 2, p. 100017, September 2023.

[26] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners, 2022.

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, p. 448–458. IEEE Press, 2015.

[28] 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)*, pp. 2450–2462, 2023.

[29] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, p. 348–359. IEEE, September 2020.

[30] OpenAI. Api 料金, 2026. アクセス日: 2026-02-02.

[31] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[32] Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. The hidden cost of readability: How code formatting silently consumes your llm budget, 2025.

[33] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? ISSTA '11, p. 199–209, New York, NY, USA, 2011. Association for Computing Machinery.

[34] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[35] Melika Sepidband, Hamed Taherkhani, Song Wang, and Hadi Hemmati. 複雑性メトリクスによる llm ベースコード生成の強化：フィードバック駆動型アプローチ. *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1416–1426, 2025.

[36] Minkyu Shin, Jin Kim, and Jiwoong Shin. The adoption and efficacy of large language models: Evidence from consumer complaints in the financial industry, 2025.

[37] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, and et al. Openai gpt-5 system card, 2025.

[38] Software Engineering Institute (SEI). Sei cert coding standards. https://wiki.sei.cmu.edu/confluence/spaces/seccode/pages/88042752/SEI+CERT+Coding+Standards, 2024.

[39] Chia-Yi Su, Aakash Bansal, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. Context-aware code summary generation. *Journal of Systems and Software*, Vol. 231, p. 112580, 2026.

[40] Jiajun Sun, Fengjie Li, Xinzhu Qi, Hongyu Zhang, and Jiajun Jiang. Empirical evaluation of large language models in automated program repair, 2025.

[41] Gemini Team. Gemini: A family of highly capable multimodal models, 2025.

[42] Taiming Wang, Hui Liu, Yuxia Zhang, and Yanjie Jiang. Recommending variable names for extract local variable refactorings. *ACM Transactions on Software Engineering and Methodology*, Vol. 34, No. 6, p. 1–38, July 2025.

[43] Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, Xinzhi Luo, and Peng Liu. How does naming affect language models on code analysis tasks? *Journal of Software Engineering and Applications*, Vol. 17, No. 11, p. 803–816, 2024.

[44] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '22, p. 959–971. ACM, November 2022.

[45] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

[46] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

[47] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F. Bissyandé, Yang Liu, and Haoye Tian. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications, 2025.

[48] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067, 2020.

[49] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol.*, Vol. 33, No. 2, December 2023.

[50] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. A systematic literature review on large language models for automated program repair. *ArXiv*, Vol. abs/2405.01466, , 2024.

[51] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, and et al. A survey of large language models, 2025.

[52] 佐々木唯, 肥後芳樹, 楠本真二. プログラム文の並べ替えに基づくソースコードの可読性向上の試み. 情報処理学会論文誌, Vol. 55, No. 2, pp. 939–946, 2 2014.