ソフトウェアテストシンポジウム 2026
JaSST'26: Japan Symposium on Software Testing 2026

LLM を用いたユニットテスト生成における実行時エラー抑制のためのプロンプト手法の提案

山田 泉樹
ソニーグローバルマニュファクチャリング&オペレーションズ株式会社

片山 徹郎
宮崎大学

高橋 寿一
株式会社マネーフォワード

あらまし 生成 AI によるユニットテスト生成は，テストの実行時エラーが実用化の障壁となっている．本稿では，依存関係が複雑なクラスのテスト生成に焦点を当てる．このようなテストで頻発する Mock オブジェクトの不適切な利用に起因した実行時エラーを防ぐため，頻出エラーの分析に基づいた「予防的指示」をプロンプトに組み込む手法を提案する．本提案手法は，LLM が自己修正を苦手とする問題に対し，予防的指示でエラーを抑制することを目的とする．産業界の実プロジェクト（81 メソッド）での評価で，テスト実行成功率を 76%から 91%へ，ブランチカバレッジを 23%から 54%へと向上させた．これは，予防的指示がテスト実行可能性を高め，エラーにより測定できなかった領域のコードカバレッジ計測を可能にすることを示している．本提案手法は，実行可能なテストコードを安定的に生成する実践的アプローチとしての有用性が高いことを示す．
キーワード ソフトウェアテスト，ユニットテスト生成，LLM，プロンプトエンジニアリング，エラー抑制

A Prompting Method for Suppressing Runtime Errors in Unit Test Generation using LLMs

Abstract Runtime errors during test execution remain a significant barrier to the practical application of unit test generation using Generative AI. This paper focuses on unit test generation for classes with complex dependencies using mock frameworks. We propose a method that incorporates "preventive instructions" into prompts based on an analysis of frequently occurring errors to prevent such errors caused by improper usage of mock objects. This method aims to suppress errors proactively through preventive instructions, addressing a key weakness of LLMs: their difficulty with self-correction. In an evaluation on an industrial project (81 methods), the proposed method improved the test execution success rate from 76% to 91% and branch coverage from 23% to 54%. These results demonstrate that preventive instructions enhance test executability, enabling code coverage measurement for regions that were previously unmeasurable due to errors. The proposed method demonstrates high practical value as a reliable approach for consistently generating executable test code.
Keyword Software Testing, Unit Test Generation, LLM, Prompt Engineering, Error Suppression

1. 背景

近年のソフトウェア開発において，アジャイル開発や CI/CD（継続的インテグレーション/継続的デリバリー）の浸透に伴い，迅速なリリースサイクルと品質保証の両立が不可欠となっている[1][2]. その中でユニットテストは，機能追加やリファクタリング時のリグレッション（デグレード）を早期に検知し，コードの品質を維持するための最も重要なセーフティネットとして位置づけられている．品質の高いユニットテスト群は，開発者が自信を持ってコードを修正・拡張することを可能にする．これにより，プロダクト全体の持続的な成長を支える土台となる．

その重要性とは裏腹に，ユニットテストの実装は多くの課題を抱える．我々はソフトウェア開発において品質向上を支援する立場から，多くのプロジェクトでこの課題を目の当たりにしてきた．ユニットテストの実装には，コーディング時間の約 16%が費やされるという報告がある[3]．我々の経験でも，プロダクトコードとユニットテストコードの実装工数が 2:1 程度となり，約 30%を占めることも珍しくない．さらに，効果的なテスト設計を行うには専門スキルが必要である．そのため，リソース制約の厳しいプロジェクトではテスト実装が後回しにされる傾向にあり，結果として技術的負債の原因となっている．

ユニットテスト効率化の試みは従来から存在したが，テンプレートやスタブの自動生成に留まり，テストの意図を汲んだ入力値の網羅や，人間が保守しやすい可読性の高いコードの生成は困難であった．このような状況を打開する技術として我々が着目したのが，大規模言語モデル（Large Language Model, LLM）に代

表される生成 AI である．近年の LLM は，人間が書いたような自然で意図を汲んだコードを生成する能力を示していることから，従来のツールが抱えていた壁を越えられると期待した．

本研究は，生成 AI によるユニットテスト自動生成の実用化へ向けた第一歩として，Java 言語を対象に，複雑な依存関係を持つクラスのテストにおいて，Mock フレームワークを用いたユニットテスト生成に取り組む．Mock とは，テスト対象が依存する外部コンポーネントを模倣したオブジェクトであり，テストの独立性を確保するために用いられる．

本研究の目的は，頻出エラー分析に基づく予防的指示をプロンプトに組み込むことで，実行可能なユニットテストを安定的に生成する手法を確立することである．具体的には，Mock オブジェクト（以下，Mock）の不適切な利用といった問題を防ぐ「予防的指示」をプロンプトに組み込む手法を提案する．なお，本研究では Mock を使わないシンプルなユニットテストは対象外とし，依存関係の複雑さに起因するエラーの抑制に注力する．

2. 課題と先行研究

2.1. 実践から見えた生成 AI の根本課題

我々はソフトウェアの品質と生産性を向上させる技術の研究開発に取り組んでいる．長年の課題であったユニットテストの非効率性に対し，生成 AI，特に LLM の活用に着目し，その実用化に向けた研究を開始した．

先行研究の動向から，当初は比較的シンプルなプロンプトでも実用的なテストコードが生成されると期待していた．しかし，実用レベルには達しなかった．具体的には，生成されたテストコードはコンパイルエラーや実行時エラーが多発し，ほとんどが実行にすら至らなかったのである．この経験から，生成 AI によるテスト生成における根本的な課題は，コードカバレッジの最適化以前に，まず「実行可能なテスト」をいかにして安定的に得るかにあるという重要な気づきを得た．

この初期の試行錯誤と失敗分析を通じて，我々は実用化を妨げる課題を整理した．

1. 実行可能性の低さ:
   我々が直面した最も深刻な課題は，生成されたテストコードがそもそも実行できないことであった．特に，複雑な依存関係を持つクラスのテストにおいて，Mock の利用が不適切であることに起因する実行時エラーは，LLM 自身による修正が困難なケースが多かった．
2. プロンプト設計の非体系性:
   高品質な出力を得るためのプロンプトの工夫は，非形式知となりがちである．特に，特定の頻出エラーを体系的に防ぐための予防的なプロンプトの設計指針は確立されていない．

2.2. 先行研究と本研究の位置づけ

我々が直面した課題，特に Mock 化の難しさについては，複数の先行研究でも指摘されている．例えば，LLM が Mock を適切に扱えず，テストの品質を低下させる問題が報告されている[4][5].

これらの課題に対するアプローチとして，エラーメッセージを LLM にフィードバックして自己修復を促す手法が提案されている[6][7]. また，プロンプトのコンテキスト情報に依存関係を追加することで成功率を向上させるアプローチも示されている[8].

これらの先行研究は重要な示唆を与えるが，我々の初期経験では，特に Mock 関連のエラー修正は，ベースラインとなる単純なプロンプトを用いた場合，事後的なエラーメッセージのフィードバックのみで修正を試みても困難なケースが多かった．また，コンテキスト情報を単に拡充するだけでは，LLM がそれをどう解釈し，どうエラーを回避すべきかを制御しきれないという課題があった．

本研究は，これらの先行研究が明らかにした課題に対し，より能動的なアプローチを取る．具体的には，LLM が自己修正困難な特定のエラーパターンを事前に分析し，それを体系的に抑制するための予防的指示を構造化プロンプトに組み込むという工学的な手法を実践する．これは，事後的なエラーフィードバックや一般的なコンテキスト拡充に留まっていた先行研究のアプローチに対し，頻出エラーを事前に防ぐ予防的指示を初期プロンプトに組み込むことで自己修正プロセスの成功率を向上させる点に独自性がある．

3. 課題解決へのアプローチ

本研究では，2.1 節で述べた課題である「実行可能性の低さ」を解決するため，その背景にある「プロンプト設計の非体系性」という問題に取り組み，エラー分析に基づく構造化プロンプト手法を実践した．

本手法は，実行可能なテストを再現性良く生成するために必要な情報をテンプレート化したものである．

本手法は，以下の 2 つの原則に基づき設計している．

1. エラーパターンの分析と予防的指示:
   LLM によるコード生成は確率的であり，常に成功するとは限らない．そこで，どのようなエラーが頻出するかを特定するため，予備実験を行った．予備実験では，産業界の実プロジェクトから抽出した 1 パッケージ（14 クラス，81 メソッド）を対象とした．ベースラインとなるプロンプトを用いて各メソッドに対しフレームワークを

独立して 6 回実行した．その上で，6 回の試行がすべて失敗した特に困難なケースに注目してエラー分析を行った．その結果，失敗の多くが，2 種類のエラーに起因することが判明した．1 つは，Mock 化すべき依存オブジェクトが初期化されず，null 参照により発生する NullPointerException である．もう 1 つは，Mockito フレームワークの API を正しく使用できていないなど，作法に反したコードが生成されることに起因する mockito.exceptions 関連の例外である．

この経験から，事後的な修正のみに頼るのではなく，確率的に失敗しやすいエラーを未然に防ぐ「予防的指示」が重要であるという結論に至った．予防的指示により，反復的な修正プロセスの成功率を高めることができる．そこで，これらの頻出エラーパターンを回避するため，「テストに必要な依存コンポーネントは，Mock フレームワークの作法に従って適切に初期化すること」といった，正しいテストの書き方を具体的に指示する予防的指示をプロンプトに組み込んだ．

2. プロンプトのテンプレート化:
   テスト対象コード以外の，テストの前提条件や制約条件（使用するテストフレームワーク，コーディング規約，前述の Mock 化ルールなど）をテンプレートとして定義する．これにより，エラーを防ぐためのノウハウを形式知化し，誰でも一定品質の出力を安定して得られるようにする．これは，プロンプト設計の非体系性という課題に対する直接的な解決策となる．

4. 実装

提案手法は，テスト対象のソースコードを静的に解析し，LLM への指示を構造化プロンプトとして生成する Python スクリプトとして実装した．本章では，その実装の詳細を述べる．

4.1. 反復的なテスト生成・修正フレームワーク

本フレームワークは，図 1 に示すように，テストコードの生成と修正を反復的に実行する．このサイクルは，テストが成功するか，試行回数が上限（本研究では初回生成を含め 6 回）に達するまで繰り返される．

1. 構文解析: テスト生成対象の Java ソースファイルを構文解析し，コンテキスト情報（import 文，クラス構造，依存関係など）を抽出する．
2. プロンプト生成: 抽出したコンテキスト情報と，あらかじめ定義された指示（ベースライン，または予防的指示を含む提案手法）をテンプレートに埋め込み，構造化プロンプトを生成する．

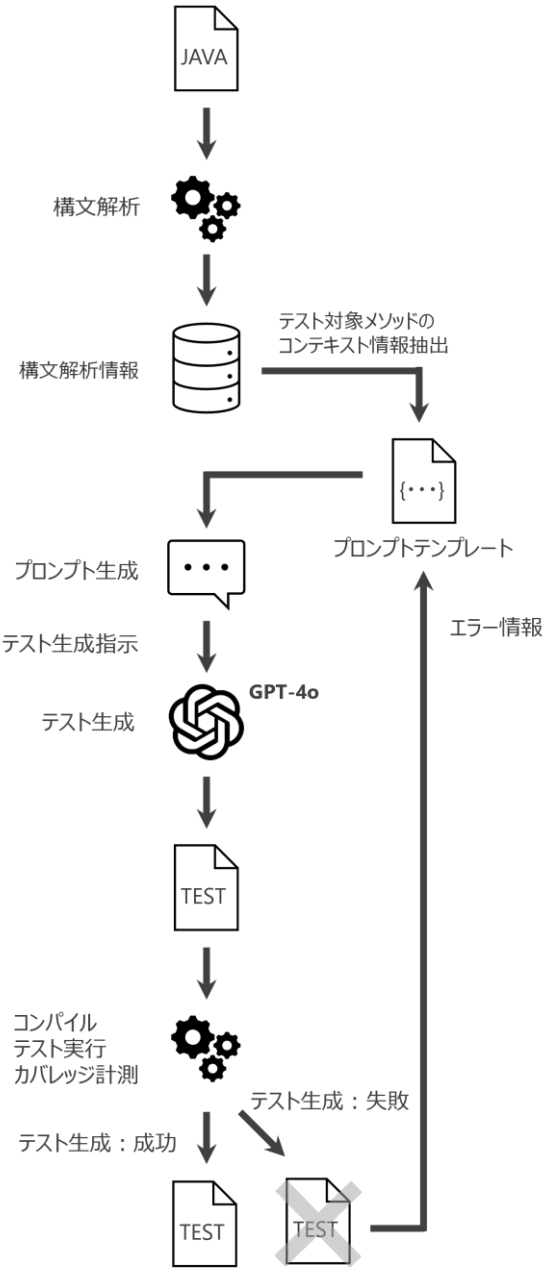

図 1: 提案フレームワークの処理フロー

3. テスト生成: 生成したプロンプトを LLM API に送信し，応答としてテストコードを受け取る．
4. テスト実行: 受け取ったテストコードをコンパイルし，実行する．
5. 成功判定: テストがエラーなく完了した場合，プロセスは成功として終了する．
6. テスト修正: コンパイルエラーや実行時エラーが発生した場合，そのエラーメッセージをコンテキストに含めた新たなプロンプトを生成し，LLM にコードの修正を指示する（ステップ 3 へ戻る）．

表 1: 使用した主な技術要素

分類	技術要素
OS	Ubuntu 22.04
システム実装	Python 3.10.12
LLM	GPT-4o[9]
テスト対象言語	Java 11
構文解析ライブラリ	Tree-sitter[10]
テストフレームワーク	JUnit 4.13.2[11]
モックフレームワーク	Mockito 2.0.9[12] PowerMock 2.0.9[13]
コードカバレッジ測定	JaCoCo 0.8.9[14]

実装に使用した主な技術要素を，表 1 に示す．LLM には，その高度なコード生成能力と長いコンテキストウィンドウを評価し，GPT-4o を選定した．また，Java コードの構文解析には，多言語に対応した構文解析ライブラリである Tree-sitter を使用した．

4.2. 構文解析によるコンテキスト抽出

LLM が，テスト対象の依存関係を正しく理解し，実行可能なテストコードを生成するためには，テスト対象のコードに関する十分なコンテキスト情報が不可欠である．本フレームワークでは，Tree-sitter を用いてソースコードの抽象構文木（AST）を解析し，以下の情報を網羅的に抽出する．

- import 文
- テスト対象クラスのシグネチャ
- テスト対象クラス内に定義されているフィールド変数およびメソッドのシグネチャ
- テスト対象メソッドのシグネチャおよびソースコード
- 依存関係にある外部クラスおよびメソッドのシグネチャ

これにより，LLM は単に 1 個のメソッドのコード片を見るのではなく，クラス全体の構造と依存関係を理解した上でテストコードを生成できる．

4.3. 構造化プロンプトの生成

次に，抽出したコンテキスト情報と，3 章で述べたエラー分析に基づく静的な「予防的指示」を，あらかじめ定義したプロンプトテンプレートに埋め込む．図 2 に，本研究で定義した構造化プロンプトのテンプレート概略を示す．

このテンプレートの{Generating Mocks}といったプレースホルダー部分に，3 章の分析で得た「NullPointerException や Mockito 関連の例外を防ぐ」ための具体的な予防的指示（例：依存オブジェクトを適切に Mo-

```
Role:
  Job:
    - Software test engineer
  Responsibilities:
    - Test design
    - Software quality
  Level:
    - Specialist
  Experienced:
    - Expert in enterprise frameworks
  Skills:
    Programming Languages
      - {language}
    Frameworks:
      Unit Test:
        - {unit test framework}
      Mock:
        - {mock framework}
Tasks:
  Main:
    - Generate a unit test for the focal method
      in the focal class
  Input:
    Method:
      - {focal method name}
    Class:
      - {focal class name}
    Information:
      Imports:
        - {import}
      Source code:
        - {focal method code}
      Signature:
        - {class signature}
        - {field signature}
        - {method signature}
  Output:
    Format:
      - {format} *e.g., Java 11
    Required Imports:
      - {required for unit test framework}
      - {required for mock framework}
      - Additional dependencies from input
    Quality:
      Coverage:
        Line:
          - {target value}
        Branch
          - {target value}
    TestPattern:
      Arrange:
        Setup:
          - {Preconditions}
        Mocking:
          - {Generating Mocks}
      Act:
        Execute:
          - {Execution of the focal method}
      Assert:
        - {Assertion}
      Prohibit:
        - {Prohibited actions}
    Grammar:
        - {rule} *e.g., Google Java Style Guide
```

図 2: 構造化プロンプトのテンプレート概略

ck 化し，初期化する指示）を挿入する．

5. 評価

提案手法の有効性を定量的に検証するため，エラー分析に基づく予防的指示を持たないプロンプト（ベースライン）と，予防的指示を持つ提案手法との比較実験を行った．

5.1. 評価設計

5.1.1 評価対象

評価の現実的な妥当性を確保するため，我々が開発に携わった，産業界で実稼働している IoT バックエンドサービスを評価対象とした．対象として選定したパッケージは，外部サービスなどとの連携を含む，依存関係が複雑な業務ロジックを持つパッケージである．対象の概要は以下の通りである．

ドメイン: IoT バックエンドサービス
対象スコープ：プロジェクト内の 1 パッケージ
クラス数：14
メソッド数：81
コード行数: 約 1,200 行

5.1.2 評価アプローチ

評価対象の全 81 メソッドに対し，以下の手順で各アプローチの評価を行った．まず，1 つのメソッドにつき，4.1 節で述べたテスト生成・修正フレームワークをそれぞれ独立に 6 回実行した．フレームワークの 1 回の実行の中では，初回生成と最大 5 回の自己修正，すなわち最大 6 回の LLM によるテスト生成を行う．独立した 6 回の実行のうち，一度でもコンパイルエラーや実行時エラーのないテストコードが得られた場合に，そのメソッドは「実行成功」と判断した．

比較は，エラー分析に基づく予防的指示の有無という違いを持つ，以下の 2 つのアプローチで行った．

提案手法: 図 2 に示した，3 章で述べた予防的指示をすべて含んだ構造化プロンプトを初期プロンプトとして使用．

ベースライン: 図 2 に示したプロンプトから，予防的指示に関する部分（例：TestPattern-Arrange-Mocking 内の指示）のみを削除したプロンプトを初期プロンプトとして使用．

5.1.3 評価指標

提案手法の有効性を測るため，以下の 3 つの指標を用いた．

1. テスト実行成功率: 各メソッドに対してフレームワークを独立に 6 回実行し，そのうち少なくとも 1 回，コンパイルエラーや実行時エラーのないテストコードが得られたメソッドの割合．
2. ステートメントカバレッジ: 評価対象となる 1 パッケージ（81 メソッド）において，実行に成功したテストコード群を実行した際の，総命令数に対する実行された命令数の割合．
3. ブランチカバレッジ: 評価対象となる 1 パッケージ（81 メソッド）において，実行に成功したテストコード群を実行した際の，総分岐数に対する実行された分岐数の割合．

コードカバレッジは JaCoCo を用いて計測した．

なお，本評価ではテストの実行可能性とコードカバレッジに焦点を当てており，アサーションの妥当性やテストケースの論理的な正しさといったテストの質的側面については今後の課題とする．

表 2: 評価結果(ベースライン vs 提案手法)

評価指標	ベースライン	提案手法	改善幅
テスト実行成功率	76%	91%	+15pt
ステートメントカバレッジ	38%	71%	+33pt
ブランチカバレッジ	23%	54%	+31pt

5.2. 評価結果

評価結果を表 2 に示す．提案手法はベースラインと比較して,テスト実行成功率を 76%から 91%へと 15 ポイント向上させた．また，ステートメントカバレッジは 38%から 71%へ，ブランチカバレッジは 23%から 54%へと，それぞれ 30 ポイント以上向上した．

5.3. 考察

実験の結果から，エラー分析に基づき設計した予防的指示が，NullPointerException や不適切な Mock 化といった頻出エラーを初期段階で防ぐことで，その後の自己修正プロセスを含めた最大 6 回の試行内での成功確率を高めたことが示された．

次に，コードカバレッジの大幅な向上について考察する．この向上は，生成されるテストケースの質が向上したというよりも，むしろテストの実行可能性が高まったことに起因する．ベースラインにおいても，LLM はある程度の分岐を考慮したテストを生成していた．しかし，Mock の初期化失敗や不適切な利用といった根本的なエラーによってテストが早期に異常終了し，それらの分岐パスが実行されずにいた．提案手法は，この実行を妨げる要因を取り除くことで，元々生成されていたテストコードが最後まで実行されるようになった．これにより，これまで測定されていなかった潜在的なコードカバレッジが顕在化したものと考え

られる．

以上の結果から，提案手法はテストの実行可能性を大幅に改善することで，コードカバレッジの観点からもその有効性が実証されたと言える．

6. 経験から得られた知見

5 章の評価結果と，そこに至るまでの我々の経験から，以下の 4 つの知見を得た．

知見 1：実行可能性の向上が潜在的コードカバレッジを顕在化させる

知見 2: 失敗ケースから見る LLM の能力的限界と再現性の課題

知見 3：評価の妥当性と一般化可能性への考察

知見 4：実践的導入に向けた提言と今後の展望

以降，この 4 つの知見を具体的に説明する．

6.1. 知見 1: 実行可能性の向上が潜在的コードカバレッジを顕在化させる

本研究で最も重要な発見は，予防的指示によるテスト実行成功率の向上が，副次的にコードカバレッジを大幅に向上させるという点である．評価結果が示す通り，テスト実行成功率が 15 ポイント向上した結果，ステートメントカバレッジは 33 ポイント（38%→71%），ブランチカバレッジは 31 ポイント（23%→54%）向上した．

コードカバレッジ向上の主な要因は，テストの実行可能性が高まったことにある．これは，LLM が生成するテストケースの質が向上したことによるものではない．ベースラインの時点でも，LLM は既にある程度の分岐を考慮したテストコードを生成していた．しかし，Mock の初期化失敗や不適切な利用といった初歩的なエラーが原因でテストが早期に異常終了し，それらの分岐パスが実行される前にテストが中断していたのである．

予防的指示は，この実行ブロッカーを取り除く抑制機能として働いた．これにより，元々LLM が生成していたものの実行に至らなかったテストロジックが最後まで実行されるようになり，これまで潜在的に存在していたコードカバレッジが，測定結果として顕在化した．

6.2. 知見 2: 失敗ケースから見る LLM の能力的限界と再現性の課題

本実験の過程で，特に成功確率が低く，頻繁に観測されたエラーパターンとして，主に以下の 2 点が挙げられる．これらは，より複雑なテストシナリオへの対応における，LLM の再現性の課題を示唆している．

第一に，メソッドチェーンの Mock 化である．プロンプトでは object.method1().method2()のような連鎖呼び出しに対する振る舞いを定義するよう指示しているが，LLM がこの指示を正しく解釈できず，不完全な Mock コードを生成してしまうケースがあった．

第二に，static final フィールドやメソッドの Mock 化である．これらの Mock 化には PowerMock の使用が不可欠であり，プロンプトでもその利用を明確に指示している．しかし，LLM がこの指示に従わずに Mockito を使い続けたり，PowerMock の作法に反したコードを生成したりするケースが散見された．

6.3. 知見 3: 評価の妥当性と一般化可能性への考察

本論文における評価の妥当性と，その結果が他のプロジェクトにも適用可能か（一般化可能性）について考察する．

まず，本評価はエラー分析に用いたデータセットと評価に用いたデータセットが同一である．これは，本稿で提案する予防的指示が評価データセットに対して過剰適合している可能性を意味し，本研究の一般化可能性に対する脅威である．しかし，本研究のアプローチは特定のデータセットへのチューニングを目的とするものではなく，多くの Java プロジェクトで共通して見られる Mock の初期化やその適切な利用といった根本的なエラーパターンを特定し，それに対する体系的な解決策を開発したものである．したがって，類似のエラーが発生しやすい他のプロジェクトにおいても，本手法は一定の有効性を持つと考えられる．

そのため，本手法の汎用性を確認する追試を行った．評価対象とは異なる OSS である Event-ruler[15]の 1 クラス（ByteMachine クラス，78 メソッド，約 900 行）に本手法を適用した結果，テスト実行成功率 79%，ステートメントカバレッジ 68%，ブランチカバレッジ 56%という良好な結果が得られた．このことは，本稿で提案するエラー分析と予防的指示というアプローチが，特定のプロジェクトに過度に依存するものではなく，ある程度の一般化可能性を持つことを示唆している．ただし，追試は 1 件に留まっており，より多様なプロジェクトやドメインでの検証は今後の重要な課題である．また，評価対象のコードが Java の全ての構文や設計パターンを網羅しているわけではなく，異なるプロジェクトでは未知のエラーパターンが出現する可能性は依然として残る．

6.4. 知見 4: 実践的導入に向けた提言と今後の展望

以上の知見から，本手法を実プロジェクトへ導入するにあたり，以下の提言と今後の展望を述べる．

まず，実践的導入への提言として，生成 AI によるテスト生成においても，「テスト容易性を考慮した設計」という原則が重要である．6.2 節で分析した通り，メソッドチェーンや static final フィールドといった複雑な

Mock 化を要するコードは，LLM にとってエラーの温床となる．テスト生成の成功率をさらに高めるには，こうした複雑な依存関係を避け，よりシンプルな設計を心がけることを推奨する．

今後の展望としては，6.2 節で明らかになった課題への直接的な対策が考えられる．LLM が苦手とするコードパターンに対しては，静的解析をさらに深化させ，それらを克服するためのより具体的かつ強制力の強い指示をプロンプトへ動的に追加する機構が有効と考えられる．また，LLM が指示に従わない再現性の課題に対しては，よりロバストなプロンプト設計を探求することや，正しいコード例をプロンプト内に数例含めることで精度を向上させるアプローチ（Few-shot Prompting）が，今後の重要な研究方向となると考えられる．

7. まとめ

本研究の目的は，頻出エラー分析に基づく予防的指示をプロンプトに組み込むことで，実行可能なユニットテストを安定的に生成する手法を確立することである．具体的には，生成 AI を用いたユニットテスト自動生成において，実用化の障壁となる「実行時エラー」と，それを防ぐための「プロンプト設計の非体系性」という 2 つの課題に取り組んだ．

これらの課題を解決するため，我々はエラー分析に基づく予防的指示を組み込んだ構造化プロンプト手法を提案した．この手法は，LLM が苦手とする，Mock の初期化やその適切な利用といった頻出エラーをプロンプトで先回りして抑制し，実行可能なテストを安定して生成することを目指すものである．

産業界の実プロジェクトから抽出した 1 パッケージ（81 メソッド）を対象とした比較評価実験により，本手法がテスト実行成功率をベースラインの 76%から 91%へと向上させた．それに伴い，ステートメントカバレッジは 38%から 71%へ，ブランチカバレッジも 23%から 54%へと大幅に改善できることを実証した．この結果は，提案手法がユニットテスト実装の初期コストを大幅に削減し，開発者のスキルに依存せず実行可能なテストコードを安定的に供給できる可能性を示すものである．ただし，本評価はテストの実行可能性とコードカバレッジに焦点を当てており，アサーションの妥当性などテストの質的側面については今後の課題として残されている．

本研究は，LLM の挙動を分析し，その弱点を体系的なプロンプトエンジニアリングによって補うアプローチの有効性を示した．また，その過程で明らかになった LLM の能力的限界や指示の不遵守といった課題は，今後の研究の方向性を示すものである．今後は，より複雑なコードへの対応力強化や，異なるプロジェクトでの有効性検証を進め，本手法をさらに実践的なものへと発展させていきたい．

文　献

[1] M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," in IEEE Access, vol. 5, pp. 3909-3943, 2017.

[2] Srungarapu Rama Krishna, Juturi Srinivasa Rao, Yenumula Venkata Durga, Lekkala Prem Venkatesh and P.S.V.S. Sridhar, "Enhancing Software Deployment Efficiency: A Comparative Analysis of Agile Application Deployment Using CI/CD Pipelines," Tuijin Jishu/Journal of Propulsion Technology, vol. 45, no. 2, pp. 2050-2061, 2024.

[3] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 201-211, Naples, Italy, Nov. 2014.

[4] M. Schäfer, S. Nadi, A. Eghbali and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," IEEE Transactions on Software Engineering, vol. 50, no. 1, pp. 85-105, Jan. 2024.

[5] Kush Jain, Gabriel Synnaeve and Baptiste Rozière, "TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark," arXiv preprint arXiv:2410.00752, 2025.

[6] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou, "Evaluating and Improving ChatGPT for Unit Test Generation," Proceedings of the ACM on Software Engineering, vol. 1, Issue FSE, no. 76, pp. 1703-1726, Dec. 2024.

[7] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," FSE 2024: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 572-576, Porto de Galinhas, Brazil, Jul. 2024.

[8] Wenhan Wang, Xuan Xie, Yuheng Huang, Renzhi Wang, An Ran Chen and Lei Ma, "Fine-grained Testing for Autonomous Driving Software: a Study on Autoware with LLM-driven Unit Testing," arXiv preprint arXiv:2501.09866, 2025.

[9] Josh Achiam et al., "GPT-4 Technical Report," arXiv preprint arXiv:2303.08774, 2024.

[10] “Tree-sitter,” 2024. [Online]. Available: https://tree-sitter.github.io/tree-sitter. (accessed 2025-06-25).

[11] “Junit,” 2024. [Online]. Available: https://junit.org.

[12] “Mockito,” 2024. [Online]. Available: https://site.mockito.org. (accessed 2025-06-25).

[13] “PowerMock,” 2024. [Online]. Available: https://powermock.github.io. (accessed 2025-06-25).

[14] “Jacoco,” 2024. [Online]. Available: https://www.jacoco.org. (accessed 2025-06-25).

[15] “Event Ruler,” 2024. [Online]. Available: https://github.com/aws/event-ruler. (accessed 2025-06-25).