AI TDDは「動作保証」のためだけじゃなかった

TDD（テスト駆動開発）の原点「学びのループ」をAI駆動開発に取り戻す

Agile Japan 2026 (2026/11/12)

小山 功二(koji-koji)

TDDの原点「学びのループ」に立ち返ると、AI TDDの可能性が広がる。

- **AI TDDには、「動作保証」と「設計改善」の2つの使い方がある**
  - 期待する振る舞いを満たす実装を作る
  - テストと仮実装から得た学びを、設計に戻す
- **目的に応じた使い分けと、AIへの指示を紹介する**
  - 「設計AをTDDで実装して」
  - 「設計をTDDで検証して、得られた学びを設計にフィードバックしてください」
- **発展系として、2つのAI TDDを組み合わせる構想も紹介する**

自己紹介



小山 功二(koji-koji)

株式会社クオレガ｜プロダクトエンジニア

AI駆動開発の新しいあり方を探求し、
特にコンテキスト設計と品質保証に着目しています。

過去の登壇

JSConf JP 2025

AST×Reporterでテストから“観点カタログ”を自動生成し、LLMと人間の共通Specにする

登壇資料（Speaker Deck）
https://x.gd/ORlUG

従来の開発手法や知見も、AI駆動開発を進化させる要素として大事にしたい。

目次

1 動作保証のためのAI TDD

06 設計どおりに動く実装を、AIに作ってもらう
07 仕様・設計を整え、実装をまとめて任せる
08 不具合を抑えながら、実装を進められた

2 従来のTDDにあった、もう一つの価値

11 改めて、TDD（テスト駆動開発）を振り返る
12 TDDを通して、私たちは何を得ていたのか？
13 その「学び」を、AI駆動開発でも活かせているか？

3 AI TDDで、設計を検証してみた

16 実践事例：バックエンドのエラーハンドリング設計
17 エラーの内容・入力位置・責務の分担を、どう設計するか
18 AIと設計を練り、第5版まで進めた
21 実装して確かめることで、設計に足りないルールが見えた
22 これは、TDDの「学びのループ」だった

4 2つのAI TDDと、その発展系

24 2つのAI TDDを、どう使い分けるか
25 たとえば、こんな場面で使う
27 発展系 — 2つのAI TDDで、プロジェクト全体を進める
31 振り返り

TDDの原点に立ち戻り、2つのAI TDDとその発展系までお話しします。

第1章

動作保証のためのAI TDD

06 設計どおりに動く実装を、AIに作ってもらう

07 仕様・設計を整え、実装をまとめて任せる

08 不具合を抑えながら、実装を進められた

設計どおりに動く実装を、AIに作ってもらう

設計を渡す

「設計AをTDDで実装して」

TDDで実装する

① テストを書く
② 失敗を確認する
③ テストを通す実装をする
④ コードを整理し、再テストする

小さな単位で、このサイクルを繰り返す。

期待する振る舞いを確かめる

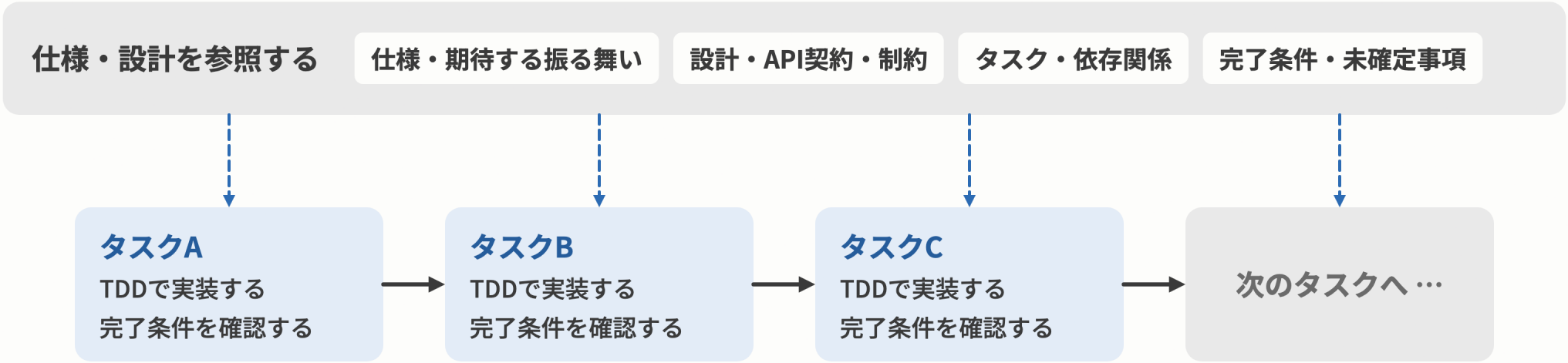
テストを通った実装

Claude Codeなら、ultracodeのような仕組みで、ハーネスを組んでいなくても簡単に試せます。ただし、トークンの消費量には注意。

設計で定めた振る舞いをテストにし、実装が満たすことを確かめる。

仕様・設計を整え、実装をまとめて任せる

継続して任せるには、AIが次の作業を判断できる情報も必要になる。



相性の良い進め方の例

仕様駆動開発 （SDD | Specification-Driven Development）

仕様を事前にドキュメント化することで、
AIが参照しながら実装・検証を進めやすくなる。

何を作るかをドキュメントで示し、できたかどうかをテストで確かめる。

不具合を抑えながら、実装を進められた

テストで動作を確かめながら、
不具合を早い段階で修正できた。

開発実績

27ページ
57 API

を3週間で実装

（実装後の多少の細かい調整はあった）

設計書の工夫と、それを支えるAI TDDが、この開発を可能にした。

でも、**TDDが持つ価値**を、
十分に活かせているだろうか？

第2章

従来のTDDにあった、もう一つの価値

11 改めて、TDD（テスト駆動開発）を振り返る

12 TDDを通して、私たちは何を得ていたのか？

13 その「学び」を、AI駆動開発でも活かせているか？

改めて、TDD（テスト駆動開発）を振り返る

テスト・実装・リファクタリングの小さなサイクルを繰り返す。



テストで振る舞いを確かめながら、小さく実装し、設計を改善する。

TDDを通して、私たちは何を得ていたのか？

期待する振る舞いが明確になる	テストの期待結果を考えると、仕様の曖昧さに気づく。
設計の改善点が見えてくる	テストを実行し、コードを見直す中で、改善の余地に気づく。
コードをよりシンプルにできる	テストで動作を確かめながら、重複や複雑さを取り除く。

動くコードとともに、設計をより良くするための学びを得ていた。

AIは、テストを書き、実装し、問題を修正している。

その途中で、設計の曖昧さや使いにくさに出会っているかもしれない。

私たちが受け取るのが「実装完了」だけだとしたら？

「実装完了」だけを受け取っていた私には、取り戻せる学びがあった。

TDDの原点に立ち返り、
AIが実装から得た学びを活かす。

それは、AI TDDの新たな可能性になり得るのでは？

第3章

AI TDDで、設計を検証してみた

16 実践事例：バックエンドのエラーハンドリング設計

17 エラーの内容・入力位置・責務の分担を、どう設計するか

18 AIと設計を練り、第5版まで進めた

21 実装して確かめることで、設計に足りないルールが見えた

22 これは、TDDの「学びのループ」だった

配属から約1か月で担当したタスク

バックエンドのエラーハンドリング

エラーレスポンスには、Problem Detailsを採用したいという要望があった。

Problem Detailsとは

RFC 9457で定められた、HTTP APIのエラー詳細を伝える共通形式。

pointerは必須項目ではなく、RFCに掲載されたバリデーションエラーの拡張例に登場するもの。

エラーレスポンスのイメージ

※理解のための例。実際の検証結果ではありません。

```
{
  "type": "https://example.com/problems/invalid-input",
  "title": "入力内容に問題があります",
  "status": 422,
  "errors": [
    {
      "detail": "数量は1以上で指定してください",
      "pointer": "#/items/1/quantity"
    }
  ]
}
```

pointer は、items配列の2番目の要素の quantity を指している。

エラーの内容・入力位置・責務の分担まで考慮する必要があり、設計の難しさを感じていた。

エラーの内容・入力位置・責務の分担を、どう設計するか

レスポンスの形式を選ぶだけでは、設計は完了しなかった。

1

エラーメッセージの定義

どのエラーに、どのメッセージを返すか。

メッセージをどこで定義し、組み立てるか。

2

入力パラメーターの位置の特定

内部で検出したエラーを、リクエスト上のどの項目に対応づけるか。

配列の場合は、どの要素のどの項目かも特定する必要がある。

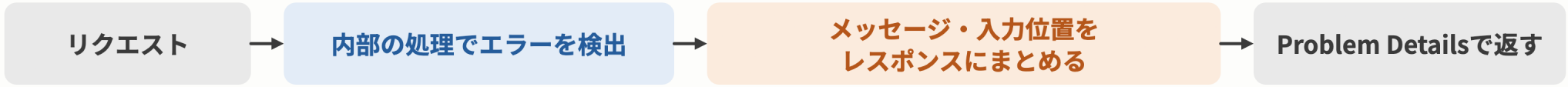
3

ヘキサゴナルアーキテクチャでの責務の分担

業務ロジックと、外部との接続部分が分離された構成。

エラーの検出、メッセージや入力位置への変換、レスポンスの生成を、どこで担うか。

検討対象の整理（概念図）

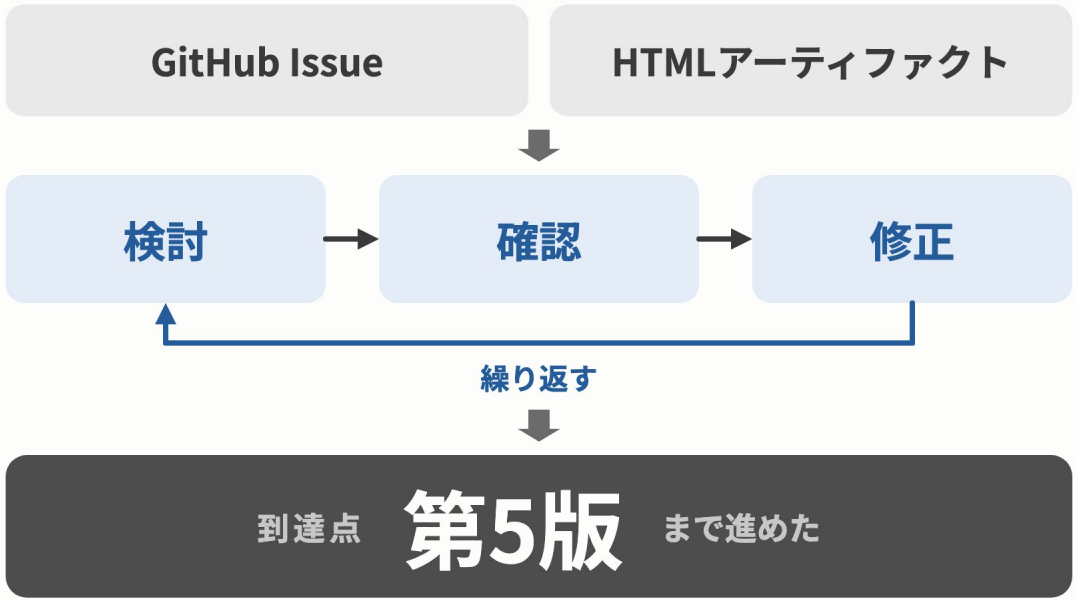


エラーの内容と入力位置を、責務の分離を保ちながら伝える設計が必要だった。

AIと設計を練り、第5版まで進めた

AIとの壁打ちに加え、GitHub IssueとHTMLアーティファクトを活用して設計を検討した。
内容を確認し、修正を重ねながら、設計を第5版まで進めた。

設計の検討に活用





HTMLアーティファクトの抜粋（設計の検討に使用）

設計は具体化できたが、実装したときに成立するか、まだ確信を持てなかった。

それでも、少し自信がなかった。

だから、実装して
学びをフィードバックしてもらった。

設計を渡し、実装で確かめてもらった

指示の例

> **この設計を、TDDで仮実装して検証してください。**
> **テストや実装を通して分かった設計の不足・矛盾・判断に迷う点を整理し、設計への修正案としてフィードバックしてください。**

試す方法の一例

独自のハーネスを用意していなくても、Claude Codeならultracodeのような仕組みを活用して試せます。

※トークンの消費量には注意。

実装の完成だけでなく、設計へのフィードバックを求めた。

実装して確かめることで、設計に足りないルールが見えた

本体の仕組みは、テストと仮実装で成立を確認できた。

そのうえで、設計を見直す指摘も見つかった。

重要な指摘

2件

動作の保証範囲に関わるもの

その他の改善・修正点

14件

設計ルール・テスト・文書など

※件数は指摘数。未解決事項を含む。

「見つからない」エラーにも、区別が必要だった

検証前の設計

指定したレコードが見つからない場合は、すべて「入力位置なし」としていた。

↓

テスト・仮実装での発見

原因となった入力項目を示せるケースもあった。

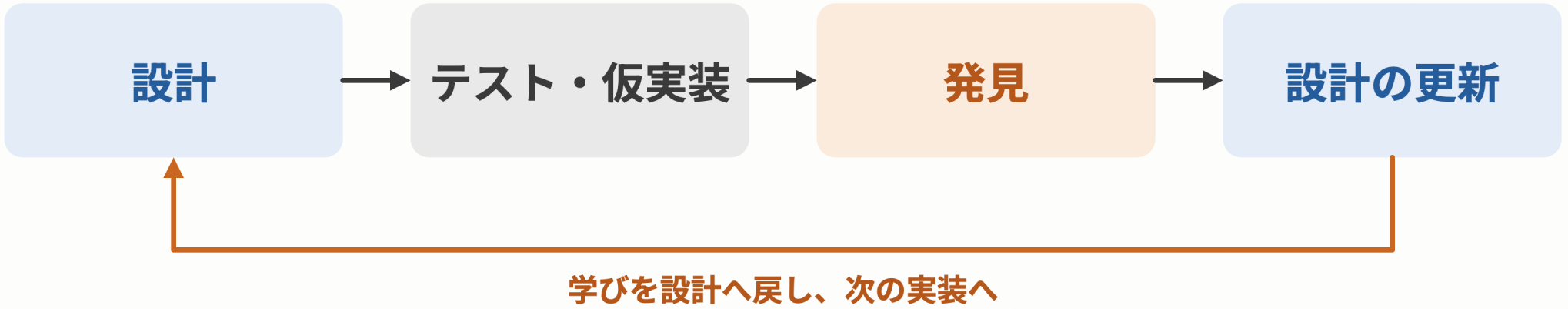
↓

設計へのフィードバック

入力項目を示す場合と、リクエスト全体のエラーとして扱う場合の判断ルールを追加した。

実装して確かめることで、設計への確信と改善の手がかりを得られた。

これは、TDDの「学びのループ」だった



AIがテストと仮実装を進める中で、設計に足りない判断が具体化した。

その発見を設計に反映することで、設計を改善できた。

AI TDDは、設計どおりに実装するためにも、設計そのものを改善するためにも使える。

第4章

2つのAI TDDと、その発展系

24 2つのAI TDDを、どう使い分けるか

25 たとえば、こんな場面で使う

27 発展系 — 2つのAI TDDで、プロジェクト全体を進める

31 振り返り

2つのAI TDDを、どう使い分けるか

設計・期待動作が明確なとき

動作保証のAI TDD

「設計AをTDDで実装して」

得たい成果

テストで確認された実装

注意点

テストの網羅性

設計や影響に不確かさがあるとき

設計改善のAI TDD

「設計AをTDDで検証して、
得られた学びを設計にフィードバックして」

得たい成果

設計への学びと改善案

注意点

トークンの消費量

期待動作が明確なら実装を進め、不確かさがあるなら検証して設計に戻す。

たとえば、こんな場面で使う

設計・期待動作が明確なとき

動作保証のAI TDD

- 仕様が決まったCRUD API
- 入出力が明確な計算・変換処理
- 既存パターンに沿った機能追加
- …

設計や影響に不確かさがあるとき

設計改善のAI TDD

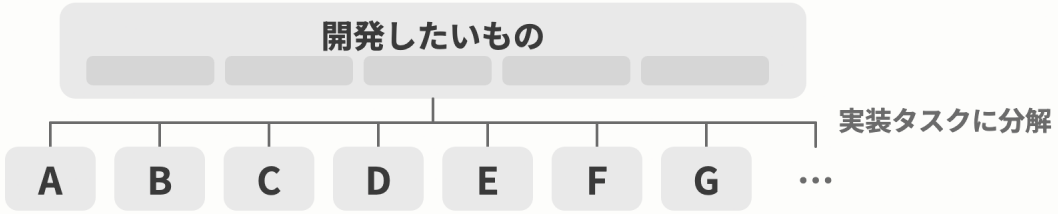
- 複数の層にまたがる責務分担の検証
- 複雑な処理のリファクタリング方針の検証
- 条件が重なる業務ルールの検証
- …

使い分けの軸は、設計や影響に不確かさがあるか。

この2つを組み合わせたら、
もっと広い範囲の開発を任せられるのでは？

ここからは、これから実践したい応用案です。

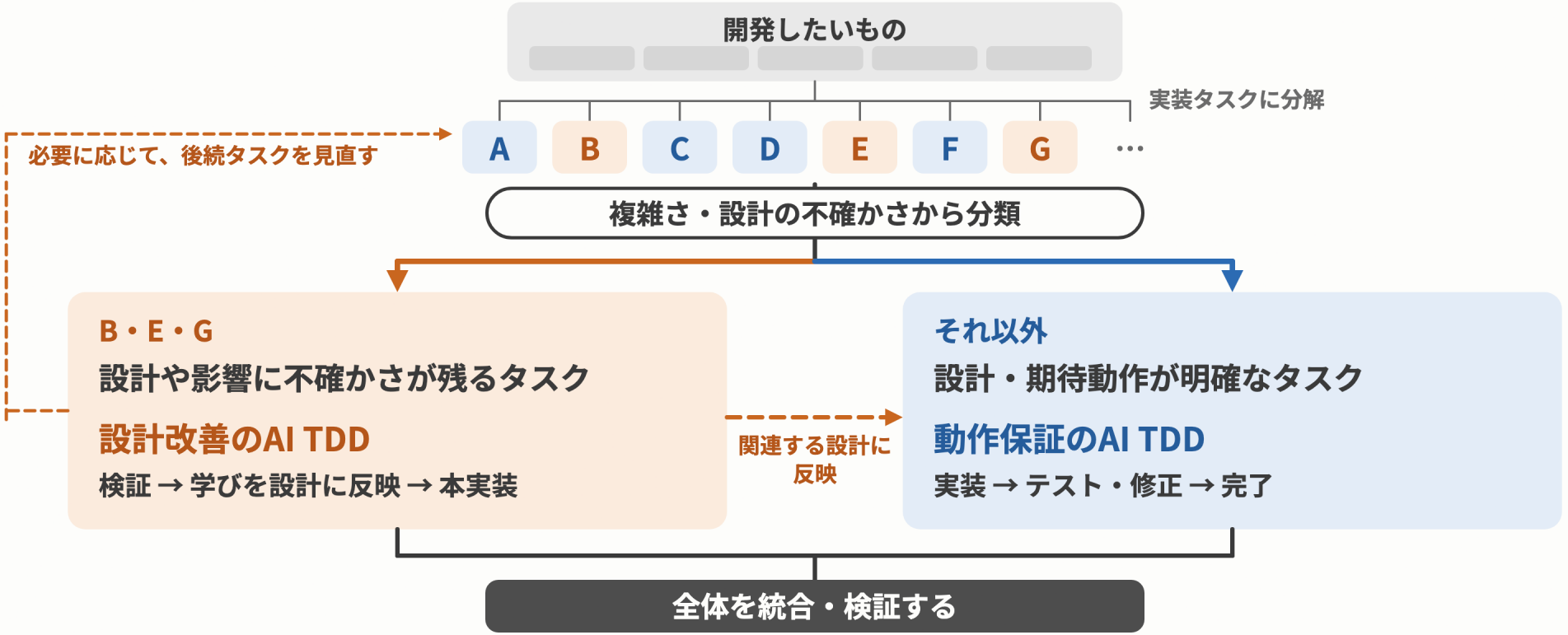
応用案 開発したいものを、小さな実装タスクに分解する。



タスクの特性に合わせて、不具合を抑えてより良い実装につなげたい。

発展系 — 2つのAI TDDで、プロジェクト全体を進める

応用案 開発したいものを、小さな実装タスクに分解する。



タスクごとに2つを組み合わせ、得られた学びを関連する設計や後続のタスクに反映する。

動作保証と設計改善のAI TDDを組み合わせることで、

プロダクト開発の進化は加速する。

振り返り

TDDの原点「学びのループ」は、AI駆動開発でも活かせる。

- **AI TDDは、不具合を抑えた実装を支えてくれた**
  - 設計書の工夫と、テストによる検証のサイクルを組み合わせた
- **実装を通して得た学びは、設計の改善にもつながった**
  - テストと仮実装で見つかった判断ルールを、設計に戻した
- **従来の手法の原点に立ち返ることが、AI活用の可能性を広げる**
  - TDDの「学びのループ」が、AIに実装を任せるだけでは得られなかった活用につながった

ありがとうございました！