Engineer

テスト駆動開発

コードより先に、小さなテストを一つずつ書く開発手法

どのようなものか

テスト駆動開発は、ある振る舞いを実現するコードより先に、その振る舞いを確かめる小さなテストを一つずつ書くプログラミングの進め方です。自分の関わりを「再発見」と説明するKent Beckは、プログラミングのワークフローと呼んでいます。2023年、用語の意味についてプログラマーの理解が一致していないことに気付き、Canon TDDという短い記事で流れを整理しました。その記事では、ほぼ一貫して略称のTDDを使っています。

ワークフローは五段階です。

- 変更で対応すべきケースを一覧にします。
- 一覧からちょうど一つを選び、実行できる具体的なテストにします。
- それまでのテストを壊さずに、そのテストを通します。作業中に見つかった新しいケースは、すべて一覧に加えます。
- 実装の設計を改善できるなら、ここでリファクタリングします。この段階は任意です。
- 一覧が空になるまで、段階2に戻ります。

  一覧は振る舞いの分析であり、設計書ではありません。基本のケース、その変化形、変更によって既存の動作を壊してはいけないケースの順に考えます。段階2で書くのは、準備、呼び出し、アサーションを備えた本物の自動テストです。対象の振る舞いはまだ実装されていないため、最初の実行では失敗します。Beckは段階3を、失敗する赤から成功する緑へ進むことと呼んでいます。

この流れに含まれないものも、理解しておくと役立ちます。コードより先にすべてのテストを書くのが嫌だ、というのはTDDへのよくある批判です。Beckの答えは明確で、それはTDDではなく、批判も別のものに向いていると述べています。逆方向にも寛容です。違う方法で良い結果が出ているチームは、その方法を続けてよく、段階を厳密に守っても、ご褒美の金の星がもらえるわけではない、としています。

概念を判断に生かす

「テスト駆動開発とは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 3社の間で受け渡す代わりに、1つのチームがアイデアのスケッチから、お客様がログインして使える製品まで担当します。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 GIS地図アプリの開発を主導し売上を10倍に — 10x 地図製品の開発を主導して実現した売上成長

GIS地図アプリの開発を主導し売上を10倍に お問い合わせ →

なぜ重要なのか

Beckは、この流れの目的も示しています。システムの振る舞いを変更するプログラマーは、既存の動作を保ちながら新しい要件を満たし、コードが次の変更に対応できるようになり、その三点について本人も同僚も確信を持てる状態を目指します。変更がまだ小さく記憶に新しいうちに各テストを書くため、安全網の整備も後回しにならず、コードと同じ速さで進みます。

二つ目の理由は設計です。Beckは、振る舞いをどう呼び出すかと、システムがどう実現するかを、別の設計判断として扱います。テストは何かを呼び出し、その戻り値を示す必要があるため、先に書くとインターフェースの判断が必要になります。実装の判断は、通るテストに守られたリファクタリングの段階まで待ちます。テストが書きにくいことは、インターフェースが使いにくくなる早い兆候です。

三つ目の理由は二重の確認です。テストの期待値は人が要件から求め、コードは別の経路で同じ答えに到達します。両者が一致すれば、独立した二つの道筋が同じ答えを出したことになります。Beckは、TDDの検証手段としての価値の多くが、この二重確認から生まれると述べています。

やめどきの目安もあります。Beckは、コードの動作への不安が退屈に変わるまで、テストと実装を続けるよう勧めています。この習慣は記事よりはるかに古いものです。自身による再発見の時期を、コンサルティングの依頼主のためにSUnitを書いた1994年から、AustinのOOPSLA会議でWard CunninghamにTDDを実演した1995年10月までの間としています。論文Simple Smalltalk Testingは Smalltalk向けの簡単なテストフレームワークを説明しており、Pythonのunittest ドキュメントも、同じパターンを説明したBeckの原論文として今も参照しています。

実際の活用例

サイトの文章に含まれる、使用禁止の流行語を報告するコンテンツチェックを考えます。まず、単なるメモとしてテスト一覧を書きます。

```
- 禁止語が報告される
- 大文字を含む禁止語が報告される
```

段階2では、ちょうど一項目をテストにします。最初の項目を先にするのには理由があります。既知の禁止語をチェックに渡し、その語が報告されることを確かめます。チェックがその語を見つけられるようになるまで、テストは失敗します。`test_wording.py` というファイルに置くと、名前が `test_*.py` に一致するのでpytestが収集します。

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

`pytest -q` を実行すると、インポートする `find_banned` が存在せず、すぐに失敗します。この失敗が狙いです。一度失敗することを確認しているからこそ、後の成功に意味が生まれます。今は、通るようにする最も単純なコードで十分です。`wording.py` に書きます。

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

テストが通りました。次の項目を次のテストにします。

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

これは失敗します。pytestはアサーションの途中の値を示すため、`["Leverage"]` が期待された場所に空のリストが表示されます。比較に `word.lower()` を使い、`word` をそのまま使わなければ通ります。その変更を書いている途中で、プログ

ラマーは `text.split()` が句読点を単語に残すため、コンマ付きの「leverage,」が見逃されると気付きます。すぐコードに加えるのではなく一覧へ追加し、三つ目のテストにします。

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

これも失敗し、もう一行加えると三つとも通ります。

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

段階4は任意で、この例には改善することがほとんどありません。二行がそれぞれ一つの仕事をしています。一覧の全項目に通るテストができたので、作業は終わりです。各期待値は一覧の項目から書き、実行結果を貼り付けたものではありません。

一覧の順序は意図的なもので、このサイトの実績にも理由が表れています。2026年8月に行われたチェック自体にテストを書く取り組みでは、スペル、文体、数値の整合性などの独自チェックが何か月も成功し続けていた一方、正しい入力しか見たことのないチェックは、何も読んでいないものと区別できないと分かりました。対策は、各チェックが見つけるべき欠陥を意図的に入れることでした。スペルチェックにはわざと誤記した単語を渡し、問題なしと出たらテストを失敗させます。文体チェックには一人称の文章を渡して必ず拒否させ、流行語チェックには禁止語を渡します。その結果、二つのチェックが、ドキュメントに書かれた範囲より少ないファイルしか読んでいなかったと判明しました。リンク先のページはTDDを説明したものではありませんが、教訓はTDDが最初から組み込むものと同じです。テストに価値が生まれるのは、一度失敗することを確認してからです。

問題が起きるところ

最初にすべてのテストを書く。 一つも通らないうちに一覧の全項目をテストに変えるのが、Beckの指摘する誤解です。最初のテストで判断を見直す必要が出れば、先回りして書いたテストをすべて修正することになります。一度も通らないまま六つ目に取り組むプログラマーは、落胆や退屈に直面すると書いています。

一つを書いて通し、それから次を選びます。新しいテストによって、それまでの実装では対応できないケースが分かったら、続けるか最初からやり直すかを決めます。Beckは、ケースの順序を変えてやり直すよう勧めています。

テストにアサーションがない。 コードを呼ぶだけで何も確かめないテストは、カバレッジの数値を上げても、何も証明しません。どのテストにも、準備、呼び出し、アサーションが必要です。Beckは、ときにはアサーションから逆向きにテストを書いてみるよう勧めています。

アサーションを削除する。 通らないテストからアサーションを消せば、通ったように見えるだけです。本当に通るようにするか、項目を一覧へ戻して順序を考え直します。

期待値を出力から貼り付ける。 コードを実行し、その戻り値をコピーしてテストに貼ると、二重の確認が一重になります。確かめるのは、コードが以前と同じことをするという点だけです。必要なら手計算をしてでも、期待値を要件から求めます。

テストを通す途中でリファクタリングする。 同じ段階で構造と振る舞いを変えるのを、Beckは二つの役割を同時に担うことにたとえています。まず動かし、その後で整えます。最も単純な変更で緑にし、テストが通ってから整理します。

整えすぎる、または抽象化を急ぎすぎる。 整理は気分がよく、次のテストに向き合うより安全に感じることもあります。今回の作業に十分な状態まで整ったら止め、似た二つのコードは、パターンが明らかになるまでそのままにします。Beckの言葉は簡潔です。「重複は手掛かりであって、命令ではない」。

テストが実装を知っている。 非公開メソッドを呼んだり、何を生成したかではなくどの呼び出しを行ったかを調べたりするテストは、内部が変わるたびに壊れます。_Software Engineering at Google_のユニットテストの章で、Erik Kueflerはこれを脆いテストと呼んでいます。実際の不具合を生まない無関係な変更でも失敗するテストです。TDDでは、内部を変えても通り続けるテストがリファクタリングに欠かせないため、負担が二重になります。公開インターフェースを通してテストし、やり取りではなく結果の状態を確認し、確かめる振る舞いにちなんでテストを命名します。

参考資料

- Canon TDD — Kent Beck自身による2023年のワークフローの説明。五段階、インターフェースと実装の設計の分離、各段階の間違いを扱います。
- Simple Smalltalk Testing: With Patterns — Smalltalk向けの簡単なテストフレームワークを説明したBeckの論文の保存版。フィクスチャ、テストケース、スイートを扱います。
- ユニットテスト — _Software Engineering at Google_のErik Kueflerによる章。リファクタリングに耐えるテスト、公開APIを通じたテスト、振る舞いのテストを扱います。
- はじめに — pytestの導入資料。インストール、最初に失敗するテストとその報告、テストファイルの見つけ方を説明しています。
- unittest：ユニットテストフレームワーク — Python標準ライブラリのフレームワーク。同じパターンの原典としてBeckの論文を挙げています。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「テスト駆動開発とは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/test-driven-development/