Engineer

ユニットテスト

✎ コードの小さな一部分を単独で確かめる自動テスト

どのようなものか

ユニットテストは、コードの一部分を単独で確かめる、小さな自動検査です。選んだ入力で関数、クラス、小さなモジュールを呼び出し、返り値が期待どおりかをアサーションで確かめます。通常は、対象コードと同じプロセスで素早く動き、ネットワーク、データベース、ブラウザーを必要としないよう設計します。これらは資源と分離に関する有用な選択であり、範囲が狭いだけで保証される性質ではありません。このようなテストを書くことをユニットテスティングと呼び、プロジェクト内の集まりをユニットテストスイートと呼びます。

何をユニットとするかは、名前から想像するほど定まっていません。*Software Engineering at Google*のユニットテストの章で、Erik Kueflerは、一つのクラスやメソッドなど、比較的狭い範囲のテストをこの言葉で表しています。その上で、ユニットは一つの関数から、関連する複数のパッケージまで広がり得るとしています。境界を考えるには、外から呼ばれる部分に注目すると役立ちます。外部の誰かが頼るインターフェースを提供するものがユニットであり、そのテストも、呼び出し側と同じようにインターフェースを使います。

現代のテストフレームワークには、共通の祖先があります。Kent Beckの論文*Simple Smalltalk Testing: With Patterns*は、フィクスチャ、テストケース、検査、スイートからなる小さな仕組みを記述し、開発者自身がクラスごとにユニットテストを書くよう勧めました。Pythonの `unittest` も、最初はJUnitに着想を得たと文書に記し、同じ四つの考え方を保っています。テストフィクスチャは必要な準備と後片付け、テストケースは個々の検査単位で、ほかにテストスイートとテストランナーがあります。両者が区別する失敗とエラーは、分けて考える価値があります。失敗は検査結果が期待と違うこと、エラーはコードの奥で投げられた例外など、テストがまったく想定していなかったことです。

テストの範囲を広さで並べると、ユニットテストは狭い側に位置します。同じ本のAdam Benderによるテストの章は、範囲で分類しています。狭いものは通常ユニットテスト、いくつかの部品を組み合わせる中程度のものは結合テスト、システム全体を動かす広いものはエンドツーエンドテストです。

概念を判断に生かす

「ユニットテストとは？」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

第一の理由は速さです。小さく分離したユニットテストは対象コードと同じプロセスで動くため、エンジニアは通常の一日の仕事で何千件も実行でき、原因となる変更が画面に残っているうちに誤りが分かります。Googleの指針は、許す操作を明確にしています。小さなテストは、スリープ、I/O、ネットワークやディスクへのアクセスをしてはなりません。その制約は、速く繰り返し実行できるテストを作りやすくします。毎回同じ答えを返すスイートなら、チームも信頼するようになります。

第二は診断です。狭いテストが失敗すれば、誤りの場所はコードの小さな部分で、多くは直前に変えた箇所です。システム全体を動かすテストなら、ブラウザーからデータベースまでのどこかかもしれません。そのため、*Software Engineering at Google*は、大まかな目安としてピラミッドを示しています。約80%を狭いテスト、15%を結合テスト、5%をエンドツーエンドテストにする配分です。

第三は変更です。公開インターフェース越しに振る舞いを確かめるスイートがあれば、その内側のコードを安心して組み替えられます。テストが通り続ければ、呼び出し側が必要とする振る舞いも保たれているからです。Kueflerの理想は、要件が変わらなければ変更不要のテストです。純粋なリファクタリングなら、テストには一切触りません。

ユニットテストは、文書にもなります。振る舞いに基づく名前を付け、分かりやすい入力と期待する結果を省略せずに明記したテストは、次の読者に、コードの呼び出し方と約束する内容を示します。コメントと違い、それが事実でなくなれ

ば失敗します。

実際の活用例

設定ファイルから `1h30m` のような時間の長さを読み、秒数に変える関数を考えます。

```
# durations.py
UNITS = {"h": 3600, "m": 60, "s": 1}


def to_seconds(text: str) -> int:
    """Turn "1h30m" or "45s" into a whole number of seconds."""
    total, digits = 0, ""
    for char in text:
        if char.isdigit():
            digits += char
        elif char in UNITS and digits:
            total += int(digits) * UNITS[char]
            digits = ""
        else:
            raise ValueError(f"not a duration: {text!r}")
    if digits or not text:
        raise ValueError(f"not a duration: {text!r}")
    return total
```

pytestで書いたユニットテストは、受け入れるものと拒否するものの、二つの振る舞いを調べます。

```
# test_durations.py
import pytest

from durations import to_seconds


@pytest.mark.parametrize(
    "text,expected",
    [("45s", 45), ("2m", 120), ("1h30m", 5400), ("0s", 0)],
)
def test_counts_each_unit(text, expected):
    assert to_seconds(text) == expected


@pytest.mark.parametrize("text", ["", "90", "h", "1x", "1.5h"])
```

```
def test_refuses_what_it_cannot_read(text):
    with pytest.raises(ValueError):
        to_seconds(text)
```

ディレクトリーで `pytest` を実行すると、 `test_*.py` または `*_test.py` という名前のファイルを集めます。二つの関数は、パラメーターの各行に対応する九つのケースに展開されます。pytestの文書は、テストを準備、実行、アサーション、後片付けに分けます。この例の準備はパラメーターの行、実行は一回の呼び出しです。関数の外には触れないので、後片付けはありません。後から誤って一分を3600秒にすると、二つのケースが失敗し、報告にはパラメーター付きの名前が出ます。 `test_counts_each_unit[2m-120]` では期待した120に対して7200、 `test_counts_each_unit[1h30m-5400]` では5400に対して111600になります。期待値は、最初にコードが返した値を写すのではなく、手で計算している点が重要です。

標準ライブラリの `unittest` で同じことを調べるなら、 `unittest.TestCase` のサブクラスを作り、 `test` で始まるメソッドから `assertEqual` と `assertRaises` を使います。 `python -m unittest` は、既定で `test*.py` に一致するファイルを見つけます。Goは、慣習をツールに組み込んでいます。 `func TestXxx(t *testing.T)` という関数を、 `_test.go` で終わるファイルに置くと、 `go test` が実行します。

規模が変わっても、変わるのは管理であり、考え方ではありません。2026年7月の仕事である981ケースを課したデータベースジェネレーターでは、383のテスト関数が、パラメーター展開後に981ケースになります。45ファイル、7,944行に分かれ、ビルドは分岐カバレッジの下限92%を強制します。実測は約95%で、警告も失敗として扱います。ページは数字の限界にも率直です。後から見つかった二つの不具合は、カバレッジ報告では実行済みのコードにありました。

問題が起きるところ

実装を写すようにテストする。 非公開の補助関数へ入り込んだり、内部呼び出しの順番そのものを検査したりすると、呼び出し側から見えるものは変わらなくても、コードを組み替えるたびに壊れます。Kueflerは、これを壊れやすいテストと呼び、相互作用を調べるテストが失敗する最も一般的な理由として、モックフレームワークへの過度の依存を挙げます。公開インターフェースを通して検査し、その状態に至るまでの呼び出しではなく結果の状態を確かめ、負担なく使える場合は本物のオブジェクトを使います。

テスト同士が依存する。 一つ目がレコードを作り、次がそれを読み、順番がそろったときだけスイートが通ります。一つだけ動かしたり、順番を変えたりすると、読者に分からない理由で失敗します。Beckの仕組みは、テストごとにデータを作って捨て、`unittest`の文書も、テストケースが完全に自己完結するよう求めています。各テストに専用のフィクスチャを与え、終わったら片付けます。

カバレッジが目標になる。 カバレッジは行が実行されたことを示すだけで、その結果を検査したかは分かりません。アサーションのないテストは、数字を増やしても何も証明しません。*Software Engineering at Google*は、80%の基準が天井になり、変更がちょうど最低値で取り込まれがちだと警告しています。数字は下限として扱い、レビューではアサーションを読みます。

メソッドごとに一つのテストを書く。 本番の各メソッドに一つずつテストメソッドを対応させると、整って見えます。しかし、分岐が増えるたびにテストも膨らみ、何を調べるのか誰にも説明できなくなります。Kueflerの助言は、代わりに振る舞いを調べることです。一つのテストで一つの振る舞いを扱い、何を示すかを名前にすれば、失敗報告は何が壊れたかを表す文になります。

時間が過ぎるのを待つ。 処理の完了を待つ`sleep()`は、毎回スイートを遅くし、それでも混雑したマシンでは失敗します。ユニットテストでは、時刻は入力です。時計やタイムスタンプを渡し、待たずに必要な時刻を設定できるようにします。

参考資料

- unittest：ユニットテストのフレームワーク — Python標準ライブラリの仕組みです。フィクスチャ、ケース、スイート、ランナー、テストの発見、失敗とエラーの区別を説明します。
- テストの構成 — pytestによる、準備、実行、アサーション、後片付けの説明です。
- 使い始める — 通常の`assert`、テストの発見、`pytest.raises`を扱う、pytest自身の入門です。
- ユニットテスト — Erik Kueflerによる*Software Engineering at Google*の章です。壊れやすいテスト、公開API、状態と相互作用、振る舞いのテストを扱います。
- Simple Smalltalk Testing: With Patterns — Kent Beckのフレームワークの論文です。クラスごとにユニットテストを書く理由と、失敗とエラーの違いを、保存された写しで読めます。
- testing — Go標準のテストパッケージです。`TestXxx`関数、`_test.go`ファイル、`go test`を説明しています。

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

「ユニットテストとは？」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/unit-test/