Engineer

Unittest

eine automatische Prüfung eines isolierten kleinen Codeteils

Was es ist

Ein Unittest ist eine kleine automatisierte Prüfung eines einzelnen Codeteils für sich: Er ruft eine Funktion, Klasse oder ein kleines Modul mit ausgewählten Eingaben auf und prüft per Assertion, ob das Ergebnis dem erwarteten entspricht. Solche Tests sind meist so ausgelegt, dass sie im selben Prozess wie der geprüfte Code laufen, schnell fertig werden und weder Netzwerk noch Datenbank oder Browser brauchen. Das sind nützliche Entscheidungen zu Ressourcen und Isolation, keine Garantien, die sich allein aus dem engen Testumfang ergeben. Die Praxis heißt meist Unit-Testing; die Sammlung solcher Tests eines Projekts ist seine Unittest-Suite.

Was als Einheit zählt, ist weniger eindeutig, als der Name vermuten lässt. Im Kapitel über Unittests in *Software Engineering at Google* verwendet Erik Kuefler den Begriff für Tests mit relativ engem Umfang, etwa einer einzelnen Klasse oder Methode. Er ergänzt, dass eine Einheit von einer einzelnen Funktion bis zu mehreren zusammengehörigen Paketen reichen kann. Sinnvoll ist die Grenze an der von außen aufgerufenen Schnittstelle: Eine Einheit ist das, was eine Schnittstelle anbietet, auf die sich jemand außerhalb verlässt. Ihre Tests benutzen diese Schnittstelle wie der aufrufende Code.

Moderne Frameworks haben gemeinsame Vorfahren. Kent Becks Aufsatz *Simple Smalltalk Testing: With Patterns* beschrieb ein kleines Framework aus Fixtures, Testfällen, Prüfungen und Suiten und riet Entwicklern, ihre Unittests selbst zu schreiben, einen für jede Klasse. Pythons `unittest` nennt in seiner Dokumentation JUnit als ursprüngliche Inspiration und behält dieselben vier Begriffe bei: Test-Fixture (die Vorbereitung für einen Test und das Aufräumen danach), Testfall (die einzelne Testeinheit), Testsuite und Test-Runner. Beide ziehen eine wichtige Grenze: Ein Fehlschlag liegt vor, wenn eine Prüfung eine falsche Antwort bekommt; ein Fehler ist etwas, das der Test überhaupt nicht vorausgesehen hat, etwa eine tief im Code ausgelöste Exception.

Unittests stehen am engen Ende eines Spektrums. Adam Benders Testkapitel im selben Buch ordnet Tests nach ihrem Umfang: Tests mit engem Umfang heißen gewöhnlich Unittests, solche mit mittlerem Umfang, die einige Komponenten gemeinsam prüfen, Integrationstests, und solche mit großem Umfang, die das gesamte System durchlaufen, End-to-End-Tests.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein Unittest?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Ein Team führt eine Idee von der Skizze bis zu einem Produkt, bei dem Ihre Kunden sich anmelden können, statt sie zwischen drei Dienstleistern weiterzureichen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird.
GIS-Kartenprodukt zu 10-fachem Umsatz geführt — 10x Umsatzwachstum durch die Leitung des Kartenprodukts

GIS-Kartenprodukt zu 10-fachem Umsatz geführt Kontakt aufnehmen →

Warum es wichtig ist

Geschwindigkeit ist der erste Grund. Ein kleiner, isolierter Unittest läuft im selben Prozess wie der geprüfte Code. Eine Entwicklerin kann daher an einem gewöhnlichen Arbeitstag Tausende davon ausführen; ein Fehler zeigt sich, während die verursachende Änderung noch auf dem Bildschirm steht. Googles Vorgaben sind bei den erlaubten Mitteln eindeutig: Ein kleiner Test darf weder schlafen noch I/O ausführen noch auf Netzwerk oder Datenträger zugreifen. Diese Grenzen erleichtern schnelle, wiederholbare Tests, und ein Team lernt, einer Suite zu vertrauen, die jedes Mal dieselbe Antwort gibt.

Der zweite Grund ist die Diagnose. Scheitert ein Test mit engem Umfang, liegt der Defekt in einem kleinen Codeteil, meist im gerade geänderten. Scheitert ein Test, der das ganze System durchläuft, kann der Defekt irgendwo zwischen Browser und Datenbank liegen. Deshalb beschreibt *Software Engineering at Google* eine Pyramide als grobe Richtschnur: etwa 80 Prozent Tests mit engem Umfang, 15 Prozent Integrationstests und 5 Prozent End-to-End-Tests.

Der dritte Grund ist Veränderung. Eine Suite, die Verhalten über eine öffentliche Schnittstelle prüft, lässt den Code dahinter ohne Angst umstrukturieren: Bestehen die Tests weiterhin, erhält aufrufender Code weiterhin das erwartete Verhalten. Kueflers Ideal ist ein Test, der sich nur ändern muss, wenn sich die Anforderungen ändern. Reines Refactoring berührt deshalb keinen Test.

Unittests dokumentieren auch. Ein nach einem Verhalten benannter Test mit einfachen Eingaben und vollständig ausgeschriebenem Erwartungswert zeigt der nächsten lesenden Person, wie der Code aufgerufen werden soll und was er zusagt. Anders als ein Kommentar schlägt er fehl, wenn das nicht mehr stimmt.

Wie es in der Praxis aussieht

Als Beispiel dient eine Funktion, die Zeitdauern wie `1h30m` aus einer Konfigurationsdatei liest und in Sekunden umwandelt:

```
# 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
```

Ihre mit pytest geschriebenen Unittests prüfen zwei Verhaltensweisen: was sie akzeptiert und was sie zurückweist.

```
# 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)
```

Ein Aufruf von `pytest` im Verzeichnis sammelt alle Dateien namens `test_*.py` oder `*_test.py` ein. Die beiden Funktionen ergeben neun Fälle, einen pro Parameterzeile. Die pytest-Dokumentation beschreibt jeden Test als Arrange, Act, Assert und Cleanup, also Vorbereiten, Handeln, Prüfen und Aufräumen. Hier ist die Vorbereitung die Parameterzeile, die Handlung der einzelne Aufruf, und es gibt nichts aufzuräumen, weil nichts außerhalb der Funktion berührt wird. Setzt jemand später versehentlich die Minute auf 3.600 Sekunden, schlagen zwei Fälle fehl, und der Bericht benennt jeden anhand seiner Parameter: `test_counts_each_unit[2m-120]` zeigt 7.200 statt der erwarteten 120, und `test_counts_each_unit[1h30m-5400]` zeigt 111.600 statt 5.400. Die Erwartungswerte sind von Hand berechnet und nicht aus der Ausgabe des ersten Programmlaufs übernommen.

Dieselben Prüfungen würden mit `unittest` aus der Standardbibliothek in einer Unterklasse von `unittest.TestCase` stehen, deren Methodennamen mit `test` beginnen und die `assertEqual` und `assertRaises` verwendet. `python -m unittest` findet standardmäßig Dateien, die `test*.py` entsprechen. Go verankert die Konvention in der Werkzeugkette: Eine Funktion `func TestXxx(t *testing.T)` in einer Datei mit der Endung `_test.go` wird von `go test` ausgeführt.

Mit der Größe ändert sich die Buchführung, nicht die Idee. Der Datenbankgenerator, der an 981 Testfällen gemessen wird, eine Arbeit vom Juli 2026, zählt 383 Testfunktionen, die nach Auflösung der Parametrisierung 981 Fälle ergeben, verteilt auf 45 Dateien und 7.944 Zeilen. Sein Build erzwingt eine Untergrenze von 92 Prozent Zweigabdeckung bei etwa 95 Prozent gemessener Abdeckung und behandelt Warnungen als Fehlschläge. Die Seite benennt offen die Grenze dieser Zahl: Zwei später gefundene Defekte lagen in Code, den der Abdeckungsbericht als abgedeckt auswies.

Wo es schiefgeht

Die Tests spiegeln die Implementierung. Ein Test, der in private Hilfsfunktionen greift oder die genaue Reihenfolge interner Aufrufe prüft, scheitert bei jeder Umstrukturierung, obwohl sich aus Sicht des aufrufenden Codes nichts geändert hat. Kuefler nennt solche Tests brüchig und die übermäßige Abhängigkeit von

Mocking-Frameworks als häufigsten Grund für problematische Interaktionstests. Getestet werden sollte über die öffentliche Schnittstelle: der entstehende Zustand statt der Aufrufe auf dem Weg dorthin, mit echten Objekten, wo ihr Einsatz wenig Aufwand verursacht.

Tests hängen voneinander ab. Ein Test legt einen Datensatz an, der nächste liest ihn, und die Suite läuft nur in dieser Reihenfolge fehlerfrei durch. Ein einzelner Test oder dieselben Tests in anderer Reihenfolge scheitern dann ohne erkennbaren Grund. Becks Framework erzeugte und verwarf seine Testdaten für jeden Test; die Dokumentation von `unittest` verlangt, dass ein Testfall vollständig eigenständig ist. Jeder Test braucht seine eigene Fixture und muss hinter sich aufräumen.

Abdeckung wird zum Ziel. Abdeckung zeigt, dass eine Zeile ausgeführt wurde, nicht, dass etwas ihr Verhalten geprüft hat. Ein Test ohne Assertion erhöht die Zahl und beweist nichts. *Software Engineering at Google* warnt, dass eine Vorgabe von 80 Prozent leicht zur Obergrenze wird und Änderungen genau beim Minimum landen. Die Zahl sollte als Untergrenze dienen; im Review sind die Assertions zu lesen.

Ein Test pro Methode. Jede Produktionsmethode einer Testmethode zuzuordnen wirkt ordentlich. Doch der Test wächst mit jedem neuen Zweig der Methode, bis niemand mehr sagen kann, was er prüft. Kuefler rät stattdessen, Verhalten zu testen: eine Verhaltensweise pro Test, benannt nach dem, was er nachweist. So liest sich der Fehlerbericht als Satz darüber, was nicht mehr funktioniert.

Der Test wartet, bis Zeit vergeht. Ein `sleep()`, das auf den Abschluss von etwas wartet, verlangsamt die Suite bei jedem Lauf und scheitert trotzdem auf einer ausgelasteten Maschine. In einem Unittest ist Zeit eine Eingabe: Eine Uhr oder ein Zeitstempel wird übergeben, damit der Test den benötigten Zeitpunkt selbst setzen kann, statt darauf zu warten.

Weiterführende Lektüre

- unittest: Unit testing framework — das Framework der Python-Standardbibliothek: Fixtures, Testfälle, Suiten, Runner, Testermittlung und der Unterschied zwischen Fehlschlag und Fehler.
- Anatomy of a test — pytest erklärt Arrange, Act, Assert und Cleanup.

- Get Started — die Einführung von pytest selbst: einfaches `assert`, Testermittlung und `pytest.raises`.
- Unit Testing — Erik Kueflers Kapitel in *Software Engineering at Google* über brüchige Tests, öffentliche APIs, Zustände gegenüber Interaktionen und das Testen von Verhalten.
- Simple Smalltalk Testing: With Patterns — Kent Becks Aufsatz über das Framework, die Begründung für einen selbst geschriebenen Unittest pro Klasse und Fehlschläge gegenüber Fehlern, in einer archivierten Fassung.
- testing — Gos Standard-Testpaket: `TestXxx`-Funktionen, `_test.go`-Dateien und `go test`.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein Unittest?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/unit-test/