Engineer

Enhedstest

en automatisk kontrol af et lille, isoleret stykke kode

Hvad det er

En enhedstest er en lille, automatisk kontrol af ét stykke kode for sig selv: den kalder en funktion, en klasse eller et lille modul med udvalgte input og kontrollerer, at det, der kommer tilbage, er det, der skal komme tilbage. Den slags test er som regel udformet til at køre i samme proces som den kode, de tester, blive hurtigt færdige og ikke kræve netværk, database eller browser. Det er nyttige valg med hensyn til ressourceforbrug og isolation, ikke garantier, der følger af testens snævre omfang alene. Den kaldes også en unit test, og arbejdet med at skrive dem kaldes enhedstestning. Et projekts samling af dem er dets enhedstestsuite.

Hvad der tæller som en enhed, er mindre afklaret, end navnet lover. I kapitlet om unit testing i *Software Engineering at Google* bruger Erik Kuefler betegnelsen om tests med et relativt snævert omfang, for eksempel af en enkelt klasse eller metode, og tilføjer, at en enhed kan være så lille som én funktion eller så bred som en samling beslægtede pakker. Den grænse, der er til at bruge i praksis, går ved det, andre kalder: en enhed er det, der stiller en grænseflade til rådighed, som nogen uden for den er afhængig af, og dens tests bruger grænsefladen på samme måde som kalderne.

De moderne frameworks har en fælles stamtavle. Kent Becks artikel *Simple Smalltalk Testing: With Patterns* beskrev et lille framework bygget af fixtures, test cases, kontroller og suites og anbefalede, at udviklerne selv skrev deres enhedstests, én pr. klasse. Pythons `unittest` skriver i sin dokumentation, at det oprindeligt var inspireret af JUnit, og det har de samme fire begreber: en test fixture (den forberedelse, en test kræver, og oprydningen bagefter), en test case (den enkelte testenhed), en test suite og en test runner. Begge trækker en grænse, der er værd at holde fast i: en failure er en kontrol, der fik det forkerte svar, mens en error er noget, testen slet ikke havde regnet med, for eksempel en undtagelse, der bliver kastet dybt inde i koden.

Enhedstests ligger i den snævre ende af en skala. Adam Benders kapitel om test i samme bog sorterer tests efter omfang: snævre tests kaldes almindeligvis unit tests, mellemstore tests, der dækker nogle få komponenter sammen, er integrationstests, og store tests, der driver hele systemet, er end-to-end-tests.

Brug begrebet i en beslutning

Brug »Hvad er en enhedstest?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Ét team, der tager en idé fra skitse til noget, jeres kunder kan logge ind på, i stedet for at kaste den mellem tre leverandører.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har ført en GIS-kortapp til 10x omsætningsvækst — 10x omsætningsvækst, drevet af kortproduktet

Har ført en GIS-kortapp til 10x omsætningsvækst Kontakt os →

Hvorfor det betyder noget

Hastighed er den første grund. En lille, isoleret enhedstest kører i samme proces som den kode, den tester, så en udvikler kan køre tusindvis af dem på en almindelig arbejdsdag, og en fejl viser sig, mens den ændring, der forårsagede den, stadig står på skærmen. Googles retningslinjer siger ligeud, hvad de tillader: en lille test må ikke sove, lave I/O eller røre ved netværket eller disken. De begrænsninger gør det lettere at lave hurtige test, der kan gentages med samme resultat, og et team lærer at stole på en suite, der giver samme svar hver gang.

Den anden grund er fejlfinding. Når en snæver test fejler, ligger fejlen i et lille stykke kode, som regel det, der lige er blevet ændret. Når en test, der driver hele systemet, fejler, kan fejlen ligge hvor som helst mellem browseren og databasen. Derfor beskriver *Software Engineering at Google* en pyramide som en grov rettesnor: omkring 80 % snævre tests, 15 % integrationstests og 5 % end-to-end-tests.

Den tredje er forandring. En suite, der kontrollerer adfærd gennem en offentlig grænseflade, gør det muligt at omstrukturere koden bag grænsefladen uden frygt: består testene stadig, får kalderne stadig det, de skal have. Kueflers ideal er en test, der aldrig skal ændres, medmindre kravene ændrer sig, så en ren refaktorering ikke rører en eneste test.

Enhedstests dokumenterer også. En test, der er opkaldt efter en adfærd, med enkle input og et forventet resultat skrevet helt ud, viser den næste læser, hvordan koden skal kaldes, og hvad den lover, og i modsætning til en kommentar

fejler den, så snart den ikke længere passer.

Sådan ser det ud i praksis

Tag en funktion, der læser varigheder som `1h30m` fra en konfigurationsfil og regner dem om til sekunder:

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


def to_seconds(text: str) -> int:
    """Regn "1h30m" eller "45s" om til et helt antal sekunder."""
    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
```

Funktionens enhedstests, skrevet til pytest, kontrollerer to slags adfærd: hvad den tager imod, og hvad den afviser.

```
# 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)
```

Kører man `pytest` i mappen, samler den alle filer, der hedder `test_*.py` eller `*_test.py`, og de to funktioner folder sig ud til ni tilfælde, ét pr. række parametre. Dokumentationen til pytest beskriver hver test som arrange, act, assert og cleanup; her er forberedelsen parameterrækken, handlingen er det ene kald, og der er intet at rydde op, fordi intet uden for funktionen bliver rørt. Kommer nogen senere til at sætte minuttet til `3600` sekunder, fejler to tilfælde, og rapporten navngiver hvert af dem ud fra dets parametre: `test_counts_each_unit[2m-120]` viser `7200` over for de forventede `120`, og `test_counts_each_unit[1h30m-5400]` viser `111600` over for `5400`. Læg mærke til, at de forventede værdier er regnet ud i hånden og ikke kopieret fra det, koden returnerede første gang.

De samme kontroller i standardbibliotekets `unittest` ville være en underklasse af `unittest.TestCase` med metoder, hvis navne begynder med `test`, og med `assertEqual` og `assertRaises`; `python -m unittest` finder som standard filer, der matcher `test*.py`. Go har bygget konventionen ind i værktøjskæden: en funktion `func TestXxx(t *testing.T)` i en fil, hvis navn ender på `_test.go`, bliver kørt af `go test`.

Skalaen ændrer bogholderiet, ikke ideen. Databasegeneratoren med 981 testtilfælde, et arbejde udført i juli 2026, tæller 383 testfunktioner, der bliver til 981 tilfælde, når de parametriserede er foldet ud, fordelt på 45 filer og 7.944 linjer. Buildet håndhæver en nedre grænse på 92 % grendækning, suiten måler omkring 95 %, og advarsler behandles som fejl. Siden er ærlig om, hvor langt tallet rækker: to af de defekter, der blev fundet senere i projektet, lå i kode, som dækningsrapporten viste som dækket.

Hvor det går galt

Testene spejler implementeringen. En test, der rækker ind i private hjælpefunktioner eller kontrollerer den nøjagtige rækkefølge af interne kald, går i stykker, hver gang koden bliver omstruktureret, selvom intet, en kalder kan se, har ændret sig. Kuefler kalder det en skrøbelig test og peger på overdreven brug af mocking-frameworks som den mest almindelige grund til, at tests af

interaktioner går galt. Test gennem den offentlige grænseflade, kontrollér den tilstand, der kommer ud af kaldet, frem for de kald, der førte dertil, og brug rigtige objekter, hvor de er billige.

Testene afhænger af hinanden. Én test opretter en post, den næste læser den, og suiten består kun i den rækkefølge. Kør en enkelt test, eller de samme tests i en anden rækkefølge, og den fejler af en grund, ingen læser kan se. Becks framework oprettede sine testdata og smed dem væk igen for hver test, og `unittest`-dokumentationen beder om, at en test case er helt selvstændig. Giv hver test sin egen fixture, og ryd op efter den.

Dækning bliver målet. Dækning viser, at en linje blev kørt, ikke at noget kontrollerede, hvad den gjorde. En test uden en eneste assertion hæver tallet og beviser intet, og *Software Engineering at Google* advarer om, at et krav på 80 % har det med at blive et loft, så ændringer lander på præcis minimum. Behandl tallet som et gulv, og læs assertions under code review.

Én test pr. metode. At give hver metode i produktionskoden sin egen testmetode ser ryddeligt ud, men testen vokser, hver gang metoden får en ny gren, indtil ingen kan sige, hvad den kontrollerer. Kueflers råd er at teste adfærd i stedet: én adfærd pr. test, opkaldt efter det, den beviser, så en fejlrapport kan læses som en sætning om, hvad der gik i stykker.

Testen venter på, at tiden går. Et `sleep()`, der skal give noget tid til at blive færdigt, gør suiten langsommere ved hver kørsel og fejler alligevel på en travl maskine. I en enhedstest er tiden et input: giv koden et ur eller et tidsstempel, så testen selv kan sætte det tidspunkt, den har brug for, i stedet for at vente på det.

Læs mere

- unittest: Unit testing framework — frameworket i Pythons standardbibliotek: fixtures, test cases, suites, runners, test discovery og forskellen på en failure og en error.
- Anatomy of a test — pytests forklaring af arrange, act, assert og cleanup.
- Get Started — pytests egen introduktion: almindelig `assert`, test discovery og `pytest.raises`.
- Unit Testing — Erik Kueflers kapitel i *Software Engineering at Google* om skrøbelige tests, offentlige API'er, tilstand frem for interaktioner og test af adfærd.

- Simple Smalltalk Testing: With Patterns — Kent Becks artikel om frameworket, om at skrive enhedstests én pr. klasse og om failures og errors, i en arkiveret kopi.
- testing — standardpakken til test i Go: `TestXxx` -funktioner, `_test.go` -filer og `go test` .

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er en enhedstest?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/unit-test/