Engineer

Enhetstest

en automatisk kontroll av en liten, isolert kodebit

Hva det er

En enhetstest er en liten, automatisk kontroll av én kodebit for seg selv: den kaller en funksjon, en klasse eller en liten modul med utvalgte inndata og sjekker at det som kommer tilbake, er det som skal komme tilbake. Slike tester utformes vanligvis slik at de kjører i samme prosess som koden de tester, blir raskt ferdige og ikke trenger nettverk, database eller nettleser. Det er nyttige valg når det gjelder ressurser og isolasjon, ikke garantier som følger av testens snevre omfang alene. Testen kalles også en unit test, og arbeidet med å skrive dem kalles enhetstesting. Prosjektets samling av dem er dets enhetstestsuite.

Hva som teller som en enhet, er mindre avklart enn navnet tilsier. I kapittelet om unit testing i *Software Engineering at Google* bruker Erik Kuefler betegnelsen om tester med relativt snevert omfang, for eksempel én klasse eller metode, og legger til at en enhet kan være så liten som én funksjon eller så vid som en samling beslektede pakker. Grensen som er nyttig i praksis, går der andre kaller koden: en enhet er det som tilbyr et grensesnitt noen utenfor er avhengige av, og testene bruker grensesnittet slik kallerne gjør.

De moderne rammeverkene har felles opphav. Kent Becks artikkel *Simple Smalltalk Testing: With Patterns* beskrev et lite rammeverk bygd av fixtures, testtilfeller, kontroller og suiter, og anbefalte at utviklerne skrev sine egne enhetstester, én per klasse. Pythons `unittest` skriver i dokumentasjonen at det opprinnelig var inspirert av JUnit, og har de samme fire begrepene: en test fixture (forberedelsen en test trenger, og oppryddingen etterpå), et testtilfelle (den enkelte testenheten), en testsuite og en testkjører. Begge trekker et skille som er verdt å holde fast ved: en failure er en kontroll som fikk feil svar, mens en error er noe testen ikke hadde regnet med i det hele tatt, for eksempel et unntak som kastes dypt inne i koden.

Enhetstester ligger i den smale enden av en skala. I kapittelet om testing i samme bok sorterer Adam Bender tester etter omfang: smale tester kalles gjerne unit tests, mellomstore tester som dekker noen få komponenter sammen, er integrasjonstester, og store tester som driver hele systemet, er ende-til-ende-tester.

Bruk begrepet i en beslutning

Bruk «Hva er en enhetstest?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Ett team som tar en idé fra skisse til noe kundene dine kan logge inn på, i stedet for at den sendes videre mellom tre leverandører.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Ledet en GIS-kartapp til 10 ganger mer omsetning — 10x inntektsvekst, ledet fra kartproduktet

Ledet en GIS-kartapp til 10 ganger mer omsetning Ta kontakt →

Hvorfor det er viktig

Hastighet er den første grunnen. En liten, isolert enhetstest kjører i samme prosess som koden den tester, så en utvikler kan kjøre tusenvis av dem på en vanlig arbeidsdag, og en feil viser seg mens endringen som forårsaket den, fortsatt står på skjermen. Googles retningslinjer er tydelige på hva de tillater: en liten test får ikke sove, gjøre I/O eller røre nettverket eller disken. Begrensningene gjør det lettere å lage raske tester med repeterbare resultater, og et team lærer seg å stole på en suite som gir samme svar hver gang.

Den andre er feilsøking. Når en smal test feiler, ligger feilen i en liten kodebit, som regel den som nettopp ble endret. Når en test som driver hele systemet, feiler, kan feilen ligge hvor som helst mellom nettleseren og databasen. Derfor beskriver *Software Engineering at Google* en pyramide som en grov rettesnor: omtrent 80 % smale tester, 15 % integrasjonstester og 5 % ende-til-ende-tester.

Den tredje er endring. En suite som sjekker atferd gjennom et offentlig grensesnitt, gjør det mulig å bygge om koden bak grensesnittet uten frykt: består testene fortsatt, får kallerne fortsatt det de skal ha. Kueflers ideal er en test som aldri må endres med mindre kravene endres, slik at en ren refaktorering ikke berører en eneste test.

Enhetstester dokumenterer også. En test som er oppkalt etter en atferd, med enkle inndata og et forventet resultat skrevet helt ut, viser neste leser hvordan koden skal kalles og hva den lover, og i motsetning til en kommentar feiler den så

snart den ikke lenger stemmer.

Slik ser det ut i praksis

Ta en funksjon som leser varigheter som `1h30m` fra en konfigurasjonsfil og regner dem om til sekunder:

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


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

Enhetstestene, skrevet for pytest, sjekker to typer atferd: hva funksjonen tar imot, og hva den avviser.

```
# 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)
```

Kjører man `pytest` i mappen, samler den alle filer som heter `test_*.py` eller `*_test.py`, og de to funksjonene foldes ut til ni tilfeller, ett per rad med parametere. Dokumentasjonen til pytest beskriver hver test som arrange, act, assert og cleanup; her er forberedelsen parameterraden, handlingen er det ene kallet, og det er ingenting å rydde opp, fordi ingenting utenfor funksjonen blir berørt. Kommer noen senere i skade for å sette minuttet til `3600` sekunder, feiler to tilfeller, og rapporten navngir hvert av dem etter parameterne: `test_counts_each_unit[2m-120]` viser `7200` mot de forventede `120`, og `test_counts_each_unit[1h30m-5400]` viser `111600` mot `5400`. Legg merke til at de forventede verdiene er regnet ut for hånd, ikke kopiert fra det koden returnerte første gang.

De samme kontrollene i standardbibliotekets `unittest` ville vært en underklasse av `unittest.TestCase` med metoder der navnet begynner med `test`, og med `assertEqual` og `assertRaises`; `python -m unittest` finner som standard filer som matcher `test*.py`. Go har bygd konvensjonen inn i verktøykjeden: en funksjon `func TestXxx(t *testing.T)` i en fil der navnet slutter på `_test.go`, kjøres av `go test`.

Skalaen endrer bokføringen, ikke ideen. Databasegeneratoren med 981 testtilfeller, arbeid utført i juli 2026, teller 383 testfunksjoner som blir 981 tilfeller når de parameteriserte er utvidet, fordelt på 45 filer og 7 944 linjer. Bygget håndhever et minstekrav på 92 % grendekning, suiten måler omtrent 95 %, og advarsler behandles som feil. Siden er ærlig om hvor langt tallet rekker: to av feilene som ble funnet senere i prosjektet, lå i kode som dekningsrapporten viste som dekket.

Hvor det går galt

Testene speiler implementasjonen. En test som griper inn i private hjelpefunksjoner eller sjekker den nøyaktige rekkefølgen av interne kall, brekker hver gang koden bygges om, selv om ingenting en kaller kan se, har endret seg. Kuefler kaller dette en skjør test og peker på overdreven bruk av mocking-rammeverk som den vanligste grunnen til at tester av interaksjoner går galt. Test gjennom det offentlige grensesnittet, sjekk tilstanden som kommer ut av kallet i stedet for kallene som førte dit, og bruk ekte objekter der de er billige.

Testene er avhengige av hverandre. Én test oppretter en post, den neste leser den, og suiten består bare i den rekkefølgen. Kjør én test alene, eller de samme testene i en annen rekkefølge, og den feiler av en grunn ingen leser kan se. Becks rammeverk opprettet testdataene og kastet dem igjen for hver test, og `unittest` -dokumentasjonen ber om at et testtilfelle er helt selvstendig. Gi hver test sin egen fixture, og rydd opp etter den.

Dekning blir målet. Dekning viser at en linje ble kjørt, ikke at noe sjekket hva den gjorde. En test uten en eneste assertion løfter tallet og beviser ingenting, og *Software Engineering at Google* advarer om at et krav på 80 % har en tendens til å bli et tak, så endringer lander på nøyaktig minimum. Behandle tallet som et gulv, og les assertions under kodegjennomgangen.

Én test per metode. Å gi hver metode i produksjonskoden sin egen testmetode ser ryddig ut, men testen vokser hver gang metoden får en ny gren, til ingen kan si hva den sjekker. Kueflers råd er å teste atferd i stedet: én atferd per test, oppkalt etter det den beviser, slik at en feilrapport kan leses som en setning om hva som gikk i stykker.

Testen venter på at tiden skal gå. En `sleep()` som skal gi noe tid til å bli ferdig, gjør suiten tregere ved hver kjøring og feiler likevel på en travel maskin. I en enhetstest er tid bare inndata: gi koden en klokke eller et tidsstempel, slik at testen selv kan sette tidspunktet den trenger, i stedet for å vente på det.

Les mer

- unittest: Unit testing framework — rammeverket i Pythons standardbibliotek: fixtures, testtilfeller, suiter, testkjørere, testoppdagelse og forskjellen på en failure og en error.
- Anatomy of a test — pytests forklaring av arrange, act, assert og cleanup.
- Get Started — pytests egen introduksjon: vanlig `assert` , testoppdagelse og `pytest.raises` .
- Unit Testing — Erik Kueflers kapittel i *Software Engineering at Google* om skjøre tester, offentlige API-er, tilstand framfor interaksjoner og testing av atferd.
- Simple Smalltalk Testing: With Patterns — Kent Becks artikkel om rammeverket, om å skrive enhetstester én per klasse og om failures og errors, i en arkivert kopi.
- testing — standardpakken for testing i Go: `TestXxx` -funksjoner, `_test.go` -filer og `go test` .

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er en enhetstest?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/unit-test/