Engineer

Enhetstest

en automatisk kontroll av en liten, isolerad kodbit

Vad det är

Ett enhetstest är en liten, automatisk kontroll av en enda kodbit för sig: det anropar en funktion, en klass eller en liten modul med utvalda indata och kontrollerar att det som kommer tillbaka är det som ska komma tillbaka. Sådana tester utformas vanligtvis för att köras i samma process som koden de testar, bli klara snabbt och inte behöva nätverk, databas eller webbläsare. Det är användbara val som gäller resurser och isolering, inte garantier som följer enbart av testets snäva omfång. Testet kallas också unit test, och arbetet med att skriva dem kallas enhetstestning. Projektets samling av dem är dess svit av enhetstester.

Vad som räknas som en enhet är mindre givet än namnet antyder. I kapitlet om unit testing i *Software Engineering at Google* använder Erik Kuefler ordet om tester med relativt snävt omfång, till exempel en enda klass eller metod, och tillägger att en enhet kan vara så liten som en funktion eller så bred som en grupp närliggande paket. Den gräns som går att använda i praktiken går där andra anropar koden: en enhet är det som erbjuder ett gränssnitt som någon utanför är beroende av, och dess tester använder gränssnittet på samma sätt som anroparna.

De moderna ramverken har ett gemensamt ursprung. Kent Becks artikel *Simple Smalltalk Testing: With Patterns* beskrev ett litet ramverk uppbyggt av fixtures, testfall, kontroller och sviter, och rekommenderade att utvecklarna skrev sina egna enhetstester, ett per klass. Pythons `unittest` skriver i sin dokumentation att det ursprungligen inspirerades av JUnit, och det har samma fyra begrepp: en test fixture (förberedelsen ett test behöver och städningen efteråt), ett testfall (den enskilda testenheten), en testsvit och en testkörare. Båda drar en gräns som är värd att hålla fast vid: en failure är en kontroll som fick fel svar, medan en error är något testet inte hade räknat med alls, till exempel ett undantag som kastas långt inne i koden.

Enhetstester ligger i den smala änden av en skala. I bokens kapitel om testning sorterar Adam Bender tester efter omfång: smala tester kallas vanligen unit tests, mellanstora tester som täcker några komponenter tillsammans är integrationstester, och stora tester som driver hela systemet är end-to-end-tester.

Använd begreppet i ett beslut

Använd ”Vad är ett enhetstest?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Ett team som tar en idé från en skiss till något dina kunder kan logga in i, i stället för att lämna den vidare mellan tre leverantörer.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Ledde GIS-kartappen till 10 gånger högre intäkter — 10x intäktstillväxt, med kartprodukten i spetsen

Ledde GIS-kartappen till 10 gånger högre intäkter Hör av er →

Varför det spelar roll

Snabbhet är det första skälet. Ett litet, isolerat enhetstest körs i samma process som koden det testar, så en utvecklare kan köra tusentals av dem under en vanlig arbetsdag, och ett misstag syns medan ändringen som orsakade det fortfarande står på skärmen. Googles riktlinjer är raka om vad de tillåter: ett litet test får inte sova, göra I/O eller röra nätverket eller disken. Begränsningarna gör det lättare att bygga snabba tester med upprepningsbara resultat, och ett team lär sig lita på en svit som ger samma svar varje gång.

Det andra är felsökning. När ett smalt test fallerar sitter felet i en liten kodbit, oftast den som just ändrades. När ett test som driver hela systemet fallerar kan felet sitta var som helst mellan webbläsaren och databasen. Därför beskriver *Software Engineering at Google* en pyramid som en grov riktlinje: ungefär 80 % smala tester, 15 % integrationstester och 5 % end-to-end-tester.

Det tredje är förändring. En svit som kontrollerar beteende genom ett publikt gränssnitt gör det möjligt att bygga om koden bakom gränssnittet utan rädsla: går testerna fortfarande igenom får anroparna fortfarande det de ska ha. Kueflers ideal är ett test som aldrig behöver ändras om inte kraven ändras, så att en ren refaktorering inte rör ett enda test.

Enhetstester dokumenterar också. Ett test som är uppkallat efter ett beteende, med enkla indata och ett förväntat resultat utskrivet i sin helhet, visar nästa läsare hur koden ska anropas och vad den lovar, och till skillnad från en

kommentar fallerar det så fort det slutar stämma.

Så ser det ut i praktiken

Ta en funktion som läser varaktigheter som `1h30m` ur en konfigurationsfil och räknar om dem till sekunder:

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


def to_seconds(text: str) -> int:
    """Räkna om "1h30m" eller "45s" till ett 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 enhetstester, skrivna för pytest, kontrollerar två beteenden: vad den tar emot och vad den avvisar.

```
# 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)
```

Kör man `pytest` i katalogen samlar det in alla filer som heter `test_*.py` eller `*_test.py`, och de två funktionerna vecklas ut till nio fall, ett per rad med parametrar. Dokumentationen till pytest beskriver varje test som arrange, act, assert och cleanup; här är förberedelsen parameterraden, handlingen är det enda anropet, och det finns inget att städa upp eftersom inget utanför funktionen berörs. Råkar någon senare sätta minuten till `3600` sekunder fallerar två testfall, och rapporten namnger vart och ett efter dess parametrar: `test_counts_each_unit[2m-120]` visar `7200` mot förväntade `120`, och `test_counts_each_unit[1h30m-5400]` visar `111600` mot `5400`. Lägg märke till att de förväntade värdena är uträknade för hand, inte kopierade från det koden returnerade första gången.

Samma kontroller i standardbibliotekets `unittest` skulle bli en underklass till `unittest.TestCase` med metoder vars namn börjar på `test`, och med `assertEqual` och `assertRaises`; `python -m unittest` hittar som standard filer som matchar `test*.py`. Go har byggt in konventionen i verktygskedjan: en funktion `func TestXxx(t *testing.T)` i en fil vars namn slutar på `_test.go` körs av `go test`.

Skalan ändrar bokföringen, inte idén. Databasgeneratorn med 981 testfall, ett arbete utfört i juli 2026, räknar 383 testfunktioner som blir 981 fall när de parametriserade har expanderats, fördelade på 45 filer och 7 944 rader. Bygget upprätthåller ett golv på 92 % grentäckning, sviten ligger på ungefär 95 %, och varningar behandlas som fel. Sidan är ärlig om hur långt talet räcker: två av de fel som hittades senare i projektet låg i kod som täckningsrapporten visade som täckt.

Var det går fel

Testerna speglar implementationen. Ett test som griper in i privata hjälpfunktioner eller kontrollerar den exakta ordningen av interna anrop går sönder varje gång koden byggs om, trots att inget en anropare kan se har ändrats. Kuefler kallar det ett skört test och pekar ut överdriven användning av mockningsramverk som den vanligaste orsaken till att tester av interaktioner går snett. Testa genom det publika gränssnittet, kontrollera det tillstånd som anropet lämnar efter sig snarare än anropen som ledde dit, och använd riktiga objekt där de är billiga.

Testerna är beroende av varandra. Ett test skapar en post, nästa läser den, och sviten går igenom bara i den ordningen. Kör ett enda test, eller samma tester i en annan ordning, och det fallerar av ett skäl som ingen läsare kan se. Becks ramverk skapade sina testdata och kastade dem igen för varje test, och dokumentationen för `unittest` ber att ett testfall ska vara helt självständigt. Ge varje test en egen fixture och städa efter det.

Täckning blir målet. Täckning visar att en rad kördes, inte att något kontrollerade vad den gjorde. Ett test utan en enda assertion höjer siffran och bevisar ingenting, och *Software Engineering at Google* varnar för att ett krav på 80 % brukar bli ett tak, så att ändringar landar på exakt miniminivån. Behandla siffran som ett golv, och läs assertions under kodgranskningen.

Ett test per metod. Att ge varje metod i produktionskoden en egen testmetod ser prydligt ut, men testet växer varje gång metoden får en ny gren, tills ingen kan säga vad det kontrollerar. Kueflers råd är att testa beteenden i stället: ett beteende per test, uppkallat efter det det bevisar, så att en felrapport kan läsas som en mening om vad som gick sönder.

Testet väntar på att tiden ska gå. En `sleep()` som ska ge något tid att bli klart gör sviten långsammare vid varje körning och fallerar ändå på en belastad maskin. I ett enhetstest är tiden indata: ge koden en klocka eller en tidsstämpel, så att testet själv kan ställa in den tidpunkt det behöver i stället för att vänta på den.

Läs mer

- unittest: Unit testing framework — ramverket i Pythons standardbibliotek: fixtures, testfall, sviter, testkörare, testupptäckt och skillnaden mellan en failure och en error.
- Anatomy of a test — pytests förklaring av arrange, act, assert och cleanup.
- Get Started — pytests egen introduktion: vanlig `assert`, testupptäckt och `pytest.raises`.
- Unit Testing — Erik Kueflers kapitel i *Software Engineering at Google* om sköra tester, publika API:er, tillstånd framför interaktioner och testning av beteenden.
- Simple Smalltalk Testing: With Patterns — Kent Becks artikel om ramverket, om att skriva enhetstester ett per klass och om failures och errors, i en arkiverad kopia.

- testing — Gos standardpaket för testning: `TestXxx`-funktioner, `_test.go`-filer och `go test`.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är ett enhetstest?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/unit-test/