Engineer

Testdrevet udvikling

at skrive testen før koden, én lille test ad gangen

Hvad det er

Testdrevet udvikling er en måde at programmere på, hvor testen for et stykke adfærd skrives før den kode, der leverer adfærden, én lille test ad gangen. Kent Beck, der selv kalder sin andel i metoden en genopdagelse, beskriver den som en arbejdsgang for programmører. I 2023 havde han erfaret, at folk ikke var enige om, hvad begrebet dækker, så han skrev arbejdsgangen ned i et kort indlæg med titlen Canon TDD. Her bruger han næsten overalt forkortelsen TDD, af det engelske test-driven development.

Arbejdsgangen har fem trin:

- Skriv en liste over de testscenarier, du vil dække.
- Gør præcis ét punkt på listen til en konkret test, der kan køres.
- Ret koden, så denne test og alle de tidligere går igennem, og føj nye punkter til listen, når du opdager dem.
- Forbedr eventuelt implementeringens design ved at refaktorere.
- Vend tilbage til trin 2, så længe der står punkter på listen.

  Listen er analyse, men af adfærd og ikke af design: først grundtilfældet, så variationerne og til sidst de måder, ændringen ikke må ødelægge det, der allerede virker. Testen fra trin 2 er en rigtig automatiseret test med opsætning, et kald og assertions, og den fejler første gang, den kører, fordi den adfærd, den beskriver, først skal skrives. Beck kalder trin 3 at gå fra rød til grøn, altså fra en test, der fejler, til en, der består.

Det er lige så nyttigt at vide, hvad arbejdsgangen ikke er. At skrive alle tests, før man skriver en eneste linje kode, er en hyppig klage over TDD, og Beck svarer direkte: Det er ikke TDD, og den, der kritiserer det, kritiserer noget andet. Den

modsatte vej er han afslappet. Et team, der arbejder anderledes og får gode resultater, må gerne fortsætte, og som han skriver, er der ingen guldstjerne for at følge trinene til punkt og prikke.

Brug begrebet i en beslutning

Brug »Hvad er testdrevet udvikling?« 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

Beck skriver selv, hvad arbejdsgangen skal føre til. En programmør, der ændrer et systems adfærd, skal ende med et system, hvor alt, hvad der virkede før, stadig virker, den nye adfærd virker som forventet, koden er klar til den næste ændring, og programmøren og kollegerne er trygge ved alle tre dele. Fordi hver test skrives, mens ændringen er lille og frisk i hukommelsen, vokser sikkerhedsnettet i samme tempo som koden i stedet for at blive lovet til senere.

Det andet argument er design. Beck skelner mellem to slags designbeslutninger: hvordan et stykke adfærd kaldes, og hvordan systemet implementerer det. Når testen skrives først, tvinger den beslutningerne om grænsefladen frem, for testen skal kalde noget og sige, hvad der kommer tilbage. Beslutningerne om implementeringen venter til refaktoreringstrinnet, hvor bestående tests beskytter dem. En test, der er besværlig at skrive, er et tidligt varsel om, at grænsefladen bliver besværlig at bruge.

Det tredje argument er dobbelttjekket. Den forventede værdi i en test regner et menneske ud fra kravet, og koden når frem til den ad en anden vej. Når de to giver det samme, er to uafhængige veje kommet frem til ét svar, og ifølge Beck er det netop dobbelttjekket, der giver TDD en stor del af dets værdi som kontrol.

Endelig giver arbejdsgangen et signal om, hvornår man er færdig. Becks råd er at blive ved med at teste og kode, indtil frygten for, hvordan koden opfører sig, er blevet til kedsomhed. Vanen er meget ældre end indlægget. Beck placerer sin genopdagelse mellem 1994, hvor han skrev SUnit til en konsulentkunde, og oktober 1995, hvor han viste TDD frem for Ward Cunningham på OOPSLA-

konferencen i Austin. Hans artikel Simple Smalltalk Testing beskriver et enkelt testframework til Smalltalk, og Pythons dokumentation af unittest henviser stadig til den som Becks oprindelige artikel om det mønster, unittest deler.

Sådan ser det ud i praksis

Forestil dig et tjek, der skal finde forbudte buzzwords i teksterne på et website. Arbejdet begynder med testlisten, skrevet som almindelige noter:

```
- et forbudt ord bliver rapporteret
- et forbudt ord med stort begyndelsesbogstav bliver rapporteret
```

Trin 2 gør præcis ét punkt til en test. Det første punkt står først af en grund: Det giver tjekket et kendt forbudt ord og kontrollerer, at ordet bliver rapporteret. Testen fejler, indtil tjekket kan finde det. Testen lægges i en fil ved navn `test_wording.py`, som pytest samler op, fordi navnet har formen `test_*.py`:

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

Kører man `pytest -q`, fejler testen straks, fordi importen af `find_banned` mislykkes. Den fejl er hele pointen: Man har set testen fejle, så når den senere består, betyder det noget. Den enkleste kode, der får den til at bestå, er nok for nu, i `wording.py`:

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

Testen består. Det næste punkt bliver den næste test:

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

Den fejler, og fordi pytest viser mellemværdierne i en assertion, kan man i outputtet se en tom liste, hvor `["Leverage"]` var forventet. At sammenligne `word.lower()` i stedet for `word` får den til at bestå. Undervejs opdager programmøren, at `text.split()` lader tegnsætningen hænge ved ordet, så »leverage,« med komma ville slippe igennem. Det kommer på listen og ikke ind i koden, og det bliver den tredje test:

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

Den fejler, og én linje mere får alle tre til at bestå:

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

Trin 4 er valgfrit, og her er der ikke meget at forbedre: to linjer med hver sin opgave. Hvert punkt på listen har nu en test, der består, så arbejdet er gjort. De forventede værdier i testene er alle skrevet ud fra punktet på listen og aldrig kopieret fra en kørsel.

Rækkefølgen på listen var ikke tilfældig, og sitets egen fortegnelse viser hvorfor. Et projekt, der skrev tests af selve tjekkene, arbejde udført i august 2026, fastslog, at projektets specialbyggede tjek, blandt dem en stavekontrol, et tjek af grammatisk person og et tjek af taloverensstemmelse, havde kørt grønt i månedsvis, og at et tjek, der kun har set rene inddata, ikke kan skelnes fra et tjek, der slet intet læser. Løsningen var at plante netop den defekt, hvert tjek findes for at fange: Stavekontrollen fodres med et bevidst fejlstavet ord, og testen fejler, hvis kørslen kommer rent tilbage, tjekket af grammatisk person fodres med prosa i første person og skal afvise den, og buzzword-tjekket fodres med et forbudt ord. To af tjekkene viste sig at læse et snævrere sæt filer, end deres dokumentation påstod. Siden beskriver ikke TDD, men dens pointe er den, TDD bygger ind fra første trin: En test er først noget værd, når man har set den fejle.

Hvor det går galt

Alle tests skrives først. At gøre hvert punkt på listen til en test, før en eneste af dem består, er den misforståelse, Beck fremhæver. Når den første test tvinger en til at tænke om, skal alle de spekulative tests laves om, og en programmør, der er nået til test nummer seks uden at have set noget bestå, rammes ifølge Beck af nedtrykthed eller kedsomhed. Skriv én test, få den til at bestå, og vælg så den næste. Viser en ny test, at det hidtidige arbejde ikke kan klare et tilfælde, må man beslutte, om man skal fortsætte eller starte forfra; Becks råd er at starte forfra og vælge en anden rækkefølge.

Testen har ingen assertion. En test, der kalder koden og intet tjekker, hæver dækningsgraden og beviser intet. Hver test skal have opsætning, et kald og en assertion, og Beck foreslår en gang imellem at skrive en test baglæns, begyndende med dens assertions.

Assertionen slettes. Når en test nægter at bestå, er det fristende at slette dens assertion, så testen lader, som om den består. Få den til at bestå for alvor, eller send punktet tilbage på listen, og tænk rækkefølgen om.

Forventede værdier kopieres fra outputtet. At køre koden, kopiere det, den returnerede, og indsætte det i testen gør et dobbelttjek til et enkelt tjek: Testen bekræfter nu kun, at koden gør det, den gjorde. Regn den forventede værdi ud fra kravet, i hånden om nødvendigt.

Refaktorering, mens testen skal bestå. At ændre struktur og adfærd i samme trin er det, Beck kalder at have to hatte på. Få det til at køre, og gør det så rigtigt: Nå grønt med den enkleste ændring, og ryd først op, når testene består.

For meget oprydning, eller abstraktion for tidligt. Det føles godt at rydde op, og det kan føles tryggere end at tage fat på den næste test. Stop, når koden er god nok til denne arbejdsomgang, og lad to stykker kode, der ligner hinanden, være, indtil mønsteret er tydeligt. Becks formulering er kort: »Duplikering er et fingerpeg, ikke en ordre.«

Testene kender implementeringen. Tests, der kalder private metoder eller tjekker, hvilke kald koden foretog, i stedet for hvad den gav som resultat, går i stykker, hver gang det indre ændres. Erik Kuefler kalder dem skrøbelige tests i sit kapitel om enhedstest i *Software Engineering at Google*: De fejler ved en

ændring, der intet har med dem at gøre og ikke indfører nogen reel fejl. I TDD koster det dobbelt, for refaktoreringstrinnet forudsætter tests, der forbliver grønne, mens det indre flytter sig. Test gennem den offentlige grænseflade, tjek den tilstand, koden efterlader, frem for samspillet, og navngiv hver test efter den adfærd, den tjekker.

Læs mere

- Canon TDD — Kent Becks egen beskrivelse af arbejdsgangen fra 2023: de fem trin, skellet mellem design af grænseflade og af implementering og fejlene ved hvert trin.
- Simple Smalltalk Testing: With Patterns — Becks artikel om et enkelt testframework til Smalltalk med fixtures, testcases og testsuiter, i en arkiveret kopi.
- Unit Testing — Erik Kueflers kapitel i *Software Engineering at Google* om tests, der overlever refaktorering, test gennem offentlige API'er og test af adfærd.
- Get Started — pytests introduktion: installation, en første test, der fejler, med dens rapport, og hvordan testfiler bliver fundet.
- unittest: Unit testing framework — frameworket i Pythons standardbibliotek, som nævner Becks artikel som den oprindelige beskrivelse af det mønster, det deler.

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 testdrevet udvikling?« 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/test-driven-development/