Engineer

Testdrevet utvikling

å skrive testen før koden, én liten test om gangen

Hva det er

Testdrevet utvikling er en måte å programmere på der testen for en bit atferd skrives før koden som leverer atferden, én liten test om gangen. Kent Beck, som selv kaller sin andel i metoden en gjenoppdagelse, beskriver den som en arbeidsflyt for programmerere. I 2023 hadde han erfart at folk ikke var enige om hva begrepet dekker, så han skrev ned arbeidsflyten i et kort innlegg med tittelen Canon TDD. Der bruker han nesten overalt forkortelsen TDD, av det engelske test-driven development.

Arbeidsflyten har fem trinn:

- Skriv en liste over testscenarioene du vil dekke.
- Gjør nøyaktig ett punkt på listen om til en konkret test som kan kjøres.
- Endre koden slik at denne testen og alle de tidligere går gjennom, og før opp nye punkter på listen etter hvert som du oppdager dem.
- Forbedre eventuelt designet av implementasjonen ved å refaktorere.
- Gå tilbake til trinn 2 så lenge det står punkter på listen.

  Listen er analyse, men av atferd og ikke av design: først grunntilfellet, så variasjonene og til slutt måtene endringen ikke må ødelegge det som allerede virker på. Testen fra trinn 2 er en ekte automatisert test med oppsett, et kall og påstander (assertions), og den feiler første gang den kjøres, fordi atferden den beskriver, først skal skrives. Beck kaller trinn 3 å gå fra rødt til grønt, altså fra en test som feiler, til en som består.

Det er like nyttig å vite hva arbeidsflyten ikke er. Å skrive alle testene før man skriver en eneste kodelinje er en vanlig innvending mot TDD, og Beck svarer rett ut: Det er ikke TDD, og den som kritiserer det, kritiserer noe annet. Den andre

veien er han like avslappet. Et team som jobber annerledes og får gode resultater, kan gjerne fortsette med det, og som han skriver, deles det ikke ut noen gullstjerne for å følge trinnene til punkt og prikke.

Bruk begrepet i en beslutning

Bruk «Hva er testdrevet utvikling?» 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

Beck skriver selv hva arbeidsflyten skal føre til. En programmerer som endrer atferden til et system, skal ende opp med et system der alt som virket før, fortsatt virker, den nye atferden virker som forventet, koden er klar for neste endring, og programmereren og kollegene er trygge på alle tre delene. Fordi hver test skrives mens endringen er liten og fersk i minnet, vokser sikkerhetsnettet i samme tempo som koden i stedet for å bli lovet til senere.

Det andre argumentet er design. Beck skiller mellom to slags designbeslutninger: hvordan en bit atferd kalles, og hvordan systemet implementerer den. Når testen skrives først, tvinger den fram beslutningene om grensesnittet, for testen må kalle noe og si hva som kommer tilbake. Beslutningene om implementasjonen venter til refaktoreringstrinnet, der bestått tester beskytter dem. En test som er tungvint å skrive, er et tidlig tegn på at grensesnittet blir tungvint å bruke.

Det tredje argumentet er dobbeltkontrollen. Den forventede verdien i en test regner et menneske ut fra kravet, og koden kommer fram til den en annen vei. Når de to gir det samme, har to uavhengige veier nådd samme svar, og ifølge Beck er det nettopp dobbeltkontrollen som gir TDD en stor del av verdien som kontroll.

Til slutt gir arbeidsflyten et signal om når man er ferdig. Becks råd er å fortsette å teste og kode til frykten for hvordan koden oppfører seg, har blitt til kjedsomhet. Vanen er mye eldre enn innlegget. Beck plasserer gjenoppdagelsen mellom 1994, da han skrev SUnit for en konsulentkunde, og oktober 1995, da han demonstrerte TDD for Ward Cunningham på OOPSLA-konferansen i Austin.

Artikkelen hans Simple Smalltalk Testing beskriver et enkelt testrammeverk for Smalltalk, og Pythons dokumentasjon av unittest viser fortsatt til den som Becks opprinnelige artikkel om mønsteret unittest deler.

Slik ser det ut i praksis

Tenk deg en kontroll som skal finne forbudte moteord i tekstene på et nettsted. Arbeidet begynner med testlisten, skrevet som vanlige notater:

```
- et forbudt ord blir rapportert
- et forbudt ord med stor forbokstav blir rapportert
```

Trinn 2 gjør nøyaktig ett punkt om til en test. Det første punktet står først av en grunn: Det gir kontrollen et kjent forbudt ord og sjekker at ordet blir rapportert. Testen feiler til kontrollen kan finne det. Testen legges i en fil som heter `test_wording.py`, og pytest plukker den opp 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"]
```

Kjører man `pytest -q`, feiler testen straks, fordi importen av `find_banned` mislykkes. Den feilen er hele poenget: Man har sett testen feile, så når den består senere, betyr det noe. Den enkleste koden som får den til å bestå, holder foreløpig, i `wording.py`:

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

Testen består. Neste punkt blir neste test:

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

Den feiler, og fordi pytest viser mellomverdiene i en påstand, viser utdataene en tom liste der `["Leverage"]` var forventet. Å sammenligne `word.lower()` i stedet for `word` får den til å bestå. Underveis oppdager programmereren at `text.split()` lar tegnsettingen henge ved ordet, så «leverage,» med komma ville sluppet gjennom. Det havner på listen og ikke i koden, og blir den tredje testen:

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

Den feiler, og én linje til får alle tre til å 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]
```

Trinn 4 er valgfritt, og her er det lite å forbedre: to linjer med hver sin oppgave. Hvert punkt på listen har nå en test som består, så arbeidet er gjort. De forventede verdiene i testene er alle skrevet ut fra punktet på listen og aldri kopiert fra en kjøring.

Rekkefølgen på listen var ikke tilfeldig, og nettstedets egen portefølje viser hvorfor. Et prosjekt som skrev tester for selve kontrollverktøyene, arbeid utført i august 2026, slo fast at prosjektets egne kontrollskript, blant dem en stavekontroll, en kontroll av grammatisk person og en kontroll av tallkonsistens, hadde vært grønne i månedsvis, og at et kontrollskript som aldri har sett annet enn rene inndata, ikke kan skilles fra et som ikke leser noe i det hele tatt. Løsningen var å plante nettopp den feilen hvert kontrollskript finnes for å finne: Stavekontrollen får et bevisst feilstavet ord, og testen feiler hvis kjøringen kommer tilbake ren, kontrollen av grammatisk person får prosa i første person og må avvise den, og moteordkontrollen får et forbudt ord. To av kontrollskriptene viste seg å lese et smalere sett med filer enn dokumentasjonen deres hevdet. Siden beskriver ikke TDD, men poenget er det samme som TDD bygger inn fra første trinn: En test er først verdt noe når man har sett den feile.

Hvor det går galt

Alle testene skrives først. Å gjøre hvert punkt på listen om til en test før en eneste av dem består, er den feiltolkningen Beck trekker fram. Når den første testen tvinger en til å tenke om, må alle de spekulative testene gjøres om, og en programmerer som har kommet til test nummer seks uten å ha sett noe bestå, rammes ifølge Beck av nedstemthet eller kjedsomhet. Skriv én test, få den til å bestå, og velg så den neste. Viser en ny test at arbeidet så langt ikke klarer et tilfelle, må man bestemme seg for om man skal fortsette eller begynne på nytt; Becks råd er å begynne på nytt og velge en annen rekkefølge.

Testen har ingen påstand. En test som kaller koden og ikke sjekker noe, øker dekningsgraden og beviser ingenting. Hver test trenger oppsett, et kall og en påstand, og Beck foreslår å prøve av og til å skrive en test baklengs, med påstandene som utgangspunkt.

Påstanden slettes. Når en test nekter å bestå, er det fristende å slette påstanden, slik at testen later som om den består. Få den til å bestå på ordentlig, eller send punktet tilbake til listen og tenk gjennom rekkefølgen på nytt.

Forventede verdier kopieres fra utdataene. Å kjøre koden, kopiere det den returnerte, og lime det inn i testen gjør en dobbeltkontroll til en enkeltkontroll: Testen bekrefter nå bare at koden gjør det den gjorde. Regn ut den forventede verdien fra kravet, for hånd om nødvendig.

Refaktorering mens testen skal bestå. Å endre struktur og atferd i samme trinn er det Beck kaller å ha to hatter på. Få det til å kjøre, og gjør det så riktig: Nå grønt med den enkleste endringen, og rydd først opp når testene består.

For mye opprydding, eller abstraksjon for tidlig. Det føles godt å rydde, og det kan føles tryggere enn å ta fatt på neste test. Stopp når koden er god nok for denne økten, og la to kodebiter som ligner hverandre, være i fred til mønsteret er tydelig. Becks formulering er kort: «Duplisering er et vink, ikke en ordre.»

Testene kjenner implementasjonen. Tester som kaller private metoder eller sjekker hvilke kall koden gjorde, i stedet for hva den ga som resultat, brekker hver gang innmaten endres. Erik Kuefler kaller dem skjøre tester i kapittelet om enhetstester i *Software Engineering at Google*: De feiler ved en endring som ikke har noe med dem å gjøre og ikke innfører noen reell feil. I TDD koster det

dobbelt, for refaktoreringstrinnet forutsetter tester som forblir grønne mens innmaten flytter seg. Test gjennom det offentlige grensesnittet, sjekk tilstanden koden etterlater, heller enn samspillet, og gi hver test navn etter atferden den sjekker.

Les mer

- Canon TDD — Kent Becks egen beskrivelse av arbeidsflyten fra 2023: de fem trinnene, skillet mellom design av grensesnitt og av implementasjon og feilene ved hvert trinn.
- Simple Smalltalk Testing: With Patterns — Becks artikkel om et enkelt testrammeverk for Smalltalk med fixtures, testtilfeller og testsuiter, i en arkivert kopi.
- Unit Testing — Erik Kueflers kapittel i *Software Engineering at Google* om tester som overlever refaktorering, testing gjennom offentlige API-er og testing av atferd.
- Get Started — innføringen i pytest: installasjon, en første test som feiler, med rapporten, og hvordan testfiler blir funnet.
- unittest: Unit testing framework — rammeverket i Pythons standardbibliotek, som nevner Becks artikkel som den opprinnelige beskrivelsen av mønsteret det deler.

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 testdrevet utvikling?» 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/test-driven-development/