Engineer

Testdriven utveckling

att skriva testet före koden, ett litet test i taget

Vad det är

Testdriven utveckling är ett sätt att programmera där testet för ett visst beteende skrivs före koden som ger beteendet, ett litet test i taget. Kent Beck, som själv kallar sin del i metoden en återupptäckt, beskriver den som ett arbetsflöde för programmerare. År 2023 hade han märkt att folk inte var överens om vad begreppet täcker, så han skrev ner arbetsflödet i ett kort inlägg med titeln Canon TDD. Där använder han nästan överallt förkortningen TDD, efter engelskans test-driven development.

Arbetsflödet har fem steg:

- Skriv en lista över de testscenarier du vill täcka.
- Gör exakt en punkt på listan till ett konkret test som går att köra.
- Ändra koden så att det här testet och alla tidigare går igenom, och för upp nya punkter på listan allteftersom du upptäcker dem.
- Förbättra eventuellt implementationens design genom att refaktorera.
- Gå tillbaka till steg 2 så länge det finns punkter kvar på listan.

  Listan är analys, men av beteende och inte av design: först grundfallet, sedan variationerna och till sist de sätt på vilka ändringen inte får förstöra det som redan fungerar. Testet från steg 2 är ett riktigt automatiserat test med förberedelse, ett anrop och påståenden (assertions), och det fallerar första gången det körs, eftersom beteendet det beskriver återstår att skriva. Beck kallar steg 3 att gå från rött till grönt, alltså från ett test som fallerar till ett som går igenom.

Det är lika nyttigt att veta vad arbetsflödet inte är. Att skriva alla tester innan man skriver en enda kodrad är ett vanligt klagomål på TDD, och Beck svarar rakt på sak: det är inte TDD, och den som kritiserar det kritiserar något annat. Åt

andra hållet är han lika avslappnad. Ett team som arbetar annorlunda och får bra resultat får gärna fortsätta så, och som han skriver delas det inte ut någon guldstjärna för att följa stegen till punkt och pricka.

Använd begreppet i ett beslut

Använd ”Vad är testdriven utveckling?” 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

Beck skriver själv vad arbetsflödet ska leda till. En programmerare som ändrar ett systems beteende ska sluta med ett system där allt som fungerade förut fortfarande fungerar, det nya beteendet fungerar som väntat, koden är redo för nästa ändring och programmeraren och kollegerna känner sig trygga med alla tre delarna. Eftersom varje test skrivs medan ändringen är liten och färsk i minnet växer skyddsnätet i samma takt som koden i stället för att utlovas till senare.

Det andra argumentet är design. Beck skiljer mellan två slags designbeslut: hur ett beteende anropas och hur systemet implementerar det. När testet skrivs först tvingar det fram besluten om gränssnittet, eftersom testet måste anropa något och säga vad som kommer tillbaka. Besluten om implementationen får vänta till refaktoreringssteget, då tester som går igenom skyddar dem. Ett test som är krångligt att skriva är ett tidigt tecken på att gränssnittet blir krångligt att använda.

Det tredje argumentet är dubbelkontrollen. Det förväntade värdet i ett test räknar en människa fram ur kravet, och koden når fram till det en annan väg. När de två ger samma resultat har två oberoende vägar lett till samma svar, och enligt Beck är det just dubbelkontrollen som ger TDD en stor del av dess värde som kontroll.

Slutligen ger arbetsflödet en signal om när man är klar. Becks råd är att fortsätta testa och koda tills rädslan för hur koden beter sig har förvandlats till leda. Vanan är mycket äldre än inlägget. Beck förlägger återupptäckten till tiden mellan 1994, då han skrev SUnit åt en konsultkund, och oktober 1995, då han

demonstrerade TDD för Ward Cunningham på OOPSLA-konferensen i Austin. Hans artikel Simple Smalltalk Testing beskriver ett enkelt testramverk för Smalltalk, och Pythons dokumentation av unittest hänvisar fortfarande till den som Becks ursprungliga artikel om det mönster som unittest delar.

Så ser det ut i praktiken

Tänk dig en kontroll som ska hitta förbjudna modeord i texterna på en webbplats. Arbetet börjar med testlistan, skriven som vanliga anteckningar:

```
- ett förbjudet ord rapporteras
- ett förbjudet ord med stor begynnelsebokstav rapporteras
```

Steg 2 gör exakt en punkt till ett test. Den första punkten står först av en anledning: den ger kontrollen ett känt förbjudet ord och kontrollerar att ordet rapporteras. Testet fallerar tills kontrollen kan hitta det. Testet hamnar i en fil som heter `test_wording.py`, och pytest plockar upp den eftersom namnet har formen `test_*.py`:

```
from wording import find_banned


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

Kör man `pytest -q` fallerar testet direkt, eftersom importen av `find_banned` misslyckas. Det felet är hela poängen: man har sett testet fallera, så när det går igenom senare betyder det något. Den enklaste koden som får det att gå igenom räcker för stunden, i `wording.py`:

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

Testet går igenom. Nästa punkt blir nästa test:

```
def test_reports_a_capitalised_banned_word():
```

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

Det fallerar, och eftersom pytest visar mellanvärdena i ett påstående syns en tom lista i utdata där `["Leverage"]` förväntades. Att jämföra `word.lower()` i stället för `word` får det att gå igenom. Under tiden märker programmeraren att `text.split()` låter skiljetecknen sitta kvar på ordet, så ”leverage,” med kommatecken skulle slinka igenom. Det hamnar på listan och inte i koden, och blir det tredje testet:

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

Det fallerar, och en rad till får alla tre att gå igenom:

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

Steg 4 är frivilligt, och här finns inte mycket att förbättra: två rader med var sin uppgift. Varje punkt på listan har nu ett test som går igenom, så arbetet är klart. De förväntade värdena i testerna är alla skrivna utifrån punkten på listan och aldrig kopierade från en körning.

Ordningen på listan var ingen slump, och webbplatsens egen portfölj visar varför. Ett projekt som skrev tester för själva kontrollerna, arbete utfört i augusti 2026, konstaterade att projektets egna kontrollprogram, bland dem en stavningskontroll, en kontroll av berättarrösten och en kontroll av sifferkonsistens, hade varit gröna i månader, och att ett kontrollprogram som bara någonsin har sett felfri indata inte går att skilja från ett som inte läser någonting alls. Lösningen var att plantera just det fel som varje kontrollprogram finns till för att hitta: stavningskontrollen matas med ett avsiktligt felstavat ord, och testet fallerar om körningen kommer tillbaka felfri, kontrollen av berättarrösten matas med prosa i första person och måste avvisa den, och kontrollen av modeord matas med ett förbjudet ord. Två av kontrollprogrammen visade sig läsa en snävare uppsättning filer än deras dokumentation påstod. Sidan beskriver inte TDD, men dess poäng är densamma som TDD bygger in från första steget: ett test är värt något först när man har sett det fallera.

Var det går fel

Alla tester skrivs först. Att göra varje punkt på listan till ett test innan ett enda av dem går igenom är den feltolkning som Beck lyfter fram. När det första testet tvingar fram en omprövning måste alla de spekulativa testerna göras om, och en programmerare som har kommit till test nummer sex utan att ha sett något gå igenom drabbas enligt Beck av nedstämdhet eller leda. Skriv ett test, få det att gå igenom och välj sedan nästa. Visar ett nytt test att arbetet hittills inte klarar ett visst fall får man avgöra om man ska fortsätta eller börja om; Becks råd är att börja om och välja en annan ordning.

Testet saknar påstående. Ett test som anropar koden och inte kontrollerar något höjer täckningsgraden och bevisar ingenting. Varje test behöver förberedelse, ett anrop och ett påstående, och Beck föreslår att man då och då provar att skriva ett test baklänges, med påståendena som utgångspunkt.

Påståendet tas bort. När ett test vägrar gå igenom är det frestande att ta bort dess påstående, så att testet låtsas gå igenom. Få det att gå igenom på riktigt, eller skicka tillbaka punkten till listan och tänk om kring ordningen.

Förväntade värden kopieras från utdata. Att köra koden, kopiera det den returnerade och klistra in det i testet gör en dubbelkontroll till en enkel kontroll: testet bekräftar nu bara att koden gör det den gjorde. Räkna fram det förväntade värdet ur kravet, för hand om det behövs.

Refaktorering medan testet ska gå igenom. Att ändra struktur och beteende i samma steg är vad Beck kallar att bära två hattar. Få det att köra och gör det sedan rätt: nå grönt med den enklaste ändringen, och städa först när testerna går igenom.

För mycket städning, eller abstraktion för tidigt. Det känns bra att städa, och det kan kännas tryggare än att ta itu med nästa test. Sluta när koden är tillräckligt bra för den här arbetsomgången, och låt två kodavsnitt som liknar varandra vara i fred tills mönstret är tydligt. Becks formulering är kort: ”Duplicering är en vink, inte en order.”

Testerna känner till implementationen. Tester som anropar privata metoder eller kontrollerar vilka anrop koden gjorde, i stället för vad den gav för resultat, går sönder varje gång det inre ändras. Erik Kuefler kallar dem sköra tester i

kapitlet om enhetstester i *Software Engineering at Google*: de fallerar vid en ändring som inte har med dem att göra och inte för in något verkligt fel. I TDD kostar det dubbelt, eftersom refaktoreringssteget förutsätter tester som förblir gröna medan det inre flyttas runt. Testa genom det publika gränssnittet, kontrollera det tillstånd koden lämnar efter sig snarare än samspelet, och namnge varje test efter beteendet det kontrollerar.

Läs mer

- Canon TDD — Kent Becks egen beskrivning av arbetsflödet från 2023: de fem stegen, skillnaden mellan design av gränssnitt och av implementation och misstagen i varje steg.
- Simple Smalltalk Testing: With Patterns — Becks artikel om ett enkelt testramverk för Smalltalk med fixturer, testfall och testsviter, i en arkiverad kopia.
- Unit Testing — Erik Kueflers kapitel i *Software Engineering at Google* om tester som överlever refaktorering, testning genom publika API:er och testning av beteenden.
- Get Started — introduktionen till pytest: installation, ett första test som fallerar, med sin rapport, och hur testfiler hittas.
- unittest: Unit testing framework — ramverket i Pythons standardbibliotek, som nämner Becks artikel som den ursprungliga beskrivningen av mönstret det delar.

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 testdriven utveckling?” 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/test-driven-development/