Engineer

Refaktorering

🛠️ å endre kodens struktur uten å endre hva den gjør

Hva det er

Refaktorering er å endre den indre strukturen i kode uten å endre det koden gjør for dem som bruker den. Navn, grensene mellom funksjoner og inndelingen i moduler endres; input, output og sideeffekter blir nøyaktig som de var. Verbet er å refaktorere, og én slik endring kalles også en refaktorering. På engelsk heter det refactoring.

Martin Fowler definerer begrepet i boken sin Refactoring og gjentok definisjonen på nettstedet sitt i 2004. Som substantiv er en refaktorering en endring i programvarens indre struktur som gjør den lettere å forstå og billigere å endre, uten at den observerbare atferden endres. Som verb betyr å refaktorere å omstrukturere programvare ved å bruke en rekke slike endringer, igjen uten å endre den observerbare atferden. Begge halvdelene teller. Formålet er en billigere neste endring, og betingelsen er at ingen utenfor koden kan se at noe har skjedd.

Fowler trekker grensen skarpt: «Refactoring isn't another word for cleaning up code,» skriver han på refactoring.com. Ordet betegner én bestemt teknikk. Han bruker omstrukturering (restructuring) som det bredere ordet for å omorganisere kode med alle midler, også slike som ikke bevarer atferden i hvert steg. Det som gjør refaktorering til en teknikk, er størrelsen. Hvert steg er en liten transformasjon som gjør lite i seg selv og sjelden går galt, og systemet holdes fullt fungerende etter hvert eneste ett. En lang rekke av dem kan likevel til sammen bli en stor endring i strukturen.

Stegene har navn. Katalogen på refactoring.com, som hører til bokens andre utgave, lister dem opp sammen med de eldre navnene de erstatter: Extract Function, tidligere kalt Extract Method, og det motsatte grepet Inline Function;

Inline Variable; Rename Variable; Move Function; Split Phase og dusinvis flere. Et felles navn lar en utvikler si med to ord hva en endring var, og lar den som går gjennom koden, holde den opp mot en kjent oppskrift.

Bruk begrepet i en beslutning

Bruk «Hva er refaktorering?» 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

Programvare som lykkes, fortsetter å endre seg. Ny funksjonalitet og rettinger legges oppå hverandre, og hver av dem gjør gjerne den neste litt vanskeligere, til nytt arbeid, slik Fowler beskriver det, nesten stopper opp. Refaktorering er motvekten: den former om koden slik at hvert tillegg ikke tar med seg kompleksitet det ikke trenger. Gevinsten måles i prisen på neste endring, ikke i hvordan koden ser ut.

Derfor behandler Fowler det som en del av vanlig programmering og ikke som en oppgave som står i en prosjektplan. Han nevner tre øyeblikk. Før han legger til ny funksjonalitet, sjekker han om koden har en form som gjør den lett å legge til, og hvis ikke refaktorerer han først, noe han som regel finner raskere enn å tvinge funksjonaliteten inn. Når funksjonaliteten virker, refaktorerer han resultatet til en form som en som kommer tilbake om noen uker, kan følge uten å gruble. Og når han må lese ukjent kode for å forstå den, refaktorerer han slik at neste leser ikke sliter på samme sted. Det første av de tre har sitt eget navn, forberedende refaktorering (preparatory refactoring), og Fowler gir Kent Beck æren for regelen bak det: ved hver endring skal man først forme om koden til endringen er lett, noe som kan være den vanskelige delen, og først deretter gjøre den.

Refaktorering holder også risikoen lav på en måte som arbeid med ny funksjonalitet ikke kan. Fowler gir videre enda en av Becks metaforer, de to hattene. Med refaktoreringshatten på bevarer hver endring atferden, og testene forblir grønne hele veien. Når man legger til funksjonalitet, kommer det nye

tester til, og eksisterende tester brekker, noe som er mer stressende og mer risikabelt. En utvikler som vet hvilken hatt som sitter på, vet også hva en feilende test betyr: under en refaktorering betyr den at det siste steget var feil.

Det er også slik det meste av den tekniske gjelden blir betalt ned. Artikkelen fra Google om skjult gjeld i maskinlæringssystemer nevner refaktorering av kode først blant måtene å betale ned gjeld på. Verktøy hjelper: mange språk har nå editorer som automatiserer vanlige refaktoreringer. Fowler regner slike verktøy som verdifulle, men ikke nødvendige, og der de mangler, holder han seg til små steg og hyppige tester.

Slik ser det ut i praksis

Ta en funksjon, skrevet for denne artikkelen, som summerer en ordre og skriver ut en kvittering på samme sted:

```
function printReceipt(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }

  // print the receipt lines
  console.log(`Order ${order.id}`);
  console.log(`Total: ${total}`);
}
```

Et nytt krav kommer: den samme totalsummen skal stå på en faktura. I stedet for å kopiere løkken refaktorerer utvikleren først. Første steg er Extract Function: løkken flyttes inn i en egen funksjon, oppkalt etter det den beregner, og testene kjøres. Andre steg trekker ut utskriften på samme måte, slik at et funksjonsnavn erstatter kommentaren, og testene kjøres igjen. Tredje steg er Inline Variable: variabelen `total` brukes nå bare ett sted, så kallet settes inn i stedet for den. Etter hvert steg gjør programmet nøyaktig det samme som før og kunne vært committet for seg:

```
function printReceipt(order) {
  printLines(order.id, orderTotal(order));
}

function orderTotal(order) {
```

```
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }
  return total;
}

function printLines(id, total) {
  console.log(`Order ${id}`);
  console.log(`Total: ${total}`);
}
```

Først nå kommer totalsummen på fakturaen til, og det krever én linje i fakturakoden som kaller `orderTotal`. Ingenting i de tre refaktoreringsstegene trengte en ny test, fordi ingenting observerbart endret seg; det er testene som allerede dekket kvitteringen, som beviser det.

Infrastrukturkode refaktoreres også, og der står mer på spill, fordi navnene viser til ressurser som kjører. Dokumentasjonen til Terraform advarer om at det å endre adressen til en ressurs i konfigurasjonen som standard er en instruks om å slette den eksisterende ressursen og opprette en ny. Å gi `aws_instance.a` det nye navnet `aws_instance.web` for lesbarhetens skyld ville derfor erstatte en server. Siden Terraform v1.1 registrerer en `moved`-blokk i stedet den gamle adressen, og neste plan oppdaterer adressen uten å slette noe:

```
resource "aws_instance" "web" {
  # (configuration unchanged)
}

moved {
  from = aws_instance.a
  to   = aws_instance.web
}
```

I tidligere versjoner viser dokumentasjonen til kommandoen `terraform state mv` for den samme flyttingen. Uansett er refaktoreringen først trygg når planen er lest og viser en flytting, ikke en sletting og en oppretting.

Det er lettere å holde slike endringer adskilt fra arbeid med ny funksjonalitet når historikken sier hva som er hva. Commitstandarden som porteføljeoppføringen her på nettstedet om å samle fire repoer under én historikkstandard beskriver, lar hver commitmelding begynne med et konvensjonelt prefiks som angir typen

endring, og en hook for commitmeldinger håndhever det. Oppføringen, som gjelder arbeid utført i juli 2026, teller 26 refaktoreringer i historikken til det største repoet, ved siden av 156 commits med funksjoner og 133 rettinger. Den sier også ærlig at hooken sjekker formatet og ikke innholdet. En commit som er merket som refaktorering, må fortsatt vise gjennom testene sine og en kodegjennomgang at den har latt atferden være i fred.

Hvor det går galt

Atferdsendringer sniker seg med i refaktoreringen. En utvikler rydder i en funksjon og retter en feil i den i samme commit. Når en test feiler, kan ingen se hvilken av de to som var årsaken, og den som går gjennom koden, må lese hver linje som en mulig endring i atferden. Ha på én hatt om gangen: refaktorer i én commit, endre atferden i den neste, og hvis en test feiler under refaktoreringen, så angre det siste steget i stedet for å lappe videre på det.

En stor omstrukturering blir kalt refaktorering. En gren står i ukevis med et bygg som ikke virker, mens koden angivelig «refaktoreres». Det er omstrukturering i Fowlers forstand, og den bærer akkurat den risikoen refaktorering er laget for å unngå. Del opp arbeidet i steg som hvert for seg etterlater systemet i fungerende stand og kan flettes inn alene; der det ikke lar seg gjøre, kall det en omskriving og planlegg det som en.

Det finnes ingen tester som viser at ingenting er endret. Bevart atferd er en påstand, og uten tester er den en gjetning. Før du rører kode uten tester, skriv tester som låser fast hva den gjør i dag, også der det ser feil ut, og rett særhetene i egne endringer etterpå. Fowlers tommelfingerregel fra eksempelet hans på forberedende refaktorering gjelder her: hver gang lysten melder seg til å kjøre koden og se på utdataene med egne øyne, skriv en test i stedet.

Et nytt navn i infrastrukturkode erstatter ressursen. Et ryddig nytt navn i Terraform ser ut som en refaktorering i kodegjennomgangen, og planen foreslår i det stille å slette en database og opprette en tom. Registrer hvert nytt navn og hver flytting inn i eller ut av en modul med en `moved` -blokk, og regn enhver sletting i planen for en endring som var ment som refaktorering, som et mislykket steg.

En gammel `moved` -blokk blir slettet. Moved-blokker hoper seg opp, og å fjerne dem ser ut som vanlig rydding. Dokumentasjonen til Terraform kaller det en endring som bryter kompatibiliteten: enhver konfigurasjon som fortsatt står på

den gamle adressen, vil da planlegge å slette det eksisterende objektet i stedet for å flytte det. Behold de historiske blokkene, kjed dem sammen når et objekt får nytt navn to ganger, og fjern dem bare fra en privat modul når alle som bruker den, har kjørt apply med versjonen som inneholder dem.

Refaktoreringen venter på en egen fase. Et team som skyver refaktoreringen til en egen periode senere, fortsetter å legge funksjonalitet til kode som stritter imot, og en periode som er planlagt til senere, er lett å kutte når en frist flytter seg. Fowlers råd går motsatt vei: refaktorer som en del av hver endring, først for å gjøre endringen lett og etterpå for å etterlate resultatet oversiktlig, slik at arbeidet aldri trenger en egen plass i planen.

Les mer

- Definition Of Refactoring — Martin Fowlers definisjoner som substantiv og verb, slik han gir dem i boken sin.
- Refactoring — Fowlers nettsted for boken, om hvorfor refaktorering hører hjemme i den daglige programmeringen, og med hele katalogen.
- Extract Function — katalogoppføringen for en av de vanligste refaktoreringene, med det motsatte grepet og navnet fra første utgave.
- An example of preparatory refactoring — Fowler former om sin egen kode i små steg før han legger til ny funksjonalitet, og forklarer de to hattene.
- Refactor modules — dokumentasjonen til Terraform om `moved` -blokker, nye navn, oppdeling av moduler og hvorfor gamle blokker bør bli stående.

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 refaktorering?» 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/refactoring/