Engineer

Refaktorering

🛠 att ändra kodens struktur utan att ändra vad den gör

Vad det är

Refaktorering är att ändra den inre strukturen i kod utan att ändra vad koden gör för den som använder den. Namn, gränserna mellan funktioner och uppdelningen i moduler ändras; indata, utdata och sidoeffekter förblir exakt som de var. Verbet är att refaktorera, och en enskild sådan ändring kallas också en refaktorering. På engelska heter det refactoring.

Martin Fowler definierar begreppet i sin bok Refactoring och upprepade definitionen på sin webbplats 2004. Som substantiv är en refaktorering en ändring av programvarans inre struktur som gör den lättare att förstå och billigare att ändra, utan att dess observerbara beteende ändras. Som verb betyder att refaktorera att omstrukturera programvara genom att tillämpa en serie sådana ändringar, återigen utan att ändra det observerbara beteendet. Båda halvorna räknas. Syftet är en billigare nästa ändring, och villkoret är att ingen utanför koden kan se att något har hänt.

Fowler drar gränsen skarpt: ”Refactoring isn’t another word for cleaning up code,” skriver han på refactoring.com. Ordet betecknar en bestämd teknik. Han använder omstrukturering (restructuring) som det bredare ordet för att omorganisera kod med alla medel, även sådana som inte bevarar beteendet i varje steg. Det som gör refaktorering till en teknik är storleken. Varje steg är en liten transformation som gör lite i sig och sällan går fel, och systemet hålls fullt fungerande efter vart och ett. En lång rad av dem kan ändå tillsammans bli en stor ändring av strukturen.

Stegen har namn. Katalogen på refactoring.com, som hör till bokens andra upplaga, listar dem tillsammans med de äldre namn de ersätter: Extract Function, tidigare kallad Extract Method, och dess motsats Inline Function; Inline Variable;

Rename Variable; Move Function; Split Phase och dussintals till. Ett gemensamt namn låter en utvecklare säga med två ord vad en ändring var och låter den som granskar koden hålla den mot ett känt recept.

Använd begreppet i ett beslut

Använd ”Vad är refaktorering?” 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

Programvara som lyckas fortsätter att förändras. Ny funktionalitet och rättningar läggs ovanpå varandra, och var och en gör gärna nästa lite svårare, tills nytt arbete, med Fowlers beskrivning, nästan står still. Refaktorering är motvikten: den formar om koden så att varje tillägg inte för med sig komplexitet som det inte behöver. Vinsten mäts i priset för nästa ändring, inte i hur koden ser ut.

Därför behandlar Fowler det som en del av vanlig programmering och inte som en uppgift som står i en projektplan. Han nämner tre tillfällen. Innan han lägger till ny funktionalitet kontrollerar han om koden har en form som gör den lätt att lägga till, och om inte refaktorerar han först, vilket han oftast tycker går snabbare än att tvinga in funktionaliteten. När funktionaliteten fungerar refaktorerar han resultatet till en form som någon som kommer tillbaka om några veckor kan följa utan att fundera. Och när han måste läsa obekant kod för att förstå den refaktorerar han så att nästa läsare inte kör fast på samma ställe. Det första av de tre har ett eget namn, förberedande refaktorering (preparatory refactoring), och Fowler tillskriver Kent Beck regeln bakom det: vid varje ändring ska man först forma om koden tills ändringen är lätt, vilket kan vara den svåra delen, och först därefter göra den.

Refaktorering håller också risken låg på ett sätt som arbete med ny funktionalitet inte kan. Fowler för vidare ännu en av Becks metaforer, de två hattarna. Med refaktoreringshatten på bevarar varje ändring beteendet, och testerna förblir gröna hela vägen. När man lägger till funktionalitet tillkommer nya tester och

befintliga går sönder, vilket är mer stressigt och mer riskfyllt. En utvecklare som vet vilken hatt som sitter på huvudet vet också vad ett test som fallerar betyder: under en refaktorering betyder det att det senaste steget var fel.

Det är också så det mesta av den tekniska skulden betalas tillbaka. Artikeln från Google om dold skuld i maskininlärningssystem nämner refaktorering av kod först bland sätten att amortera skuld. Verktyg hjälper: många språk har nu editorer som automatiserar vanliga refaktoreringar. Fowler räknar sådana verktyg som värdefulla men inte nödvändiga, och där de saknas håller han sig till små steg och täta tester.

Så ser det ut i praktiken

Ta en funktion, skriven för den här artikeln, som summerar en order och skriver ut ett kvitto på ett och samma ställe:

```
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}`);
}
```

Ett nytt krav kommer: samma totalsumma ska stå på en faktura. I stället för att kopiera loopen refaktorerar utvecklaren först. Första steget är Extract Function: loopen flyttas till en egen funktion, namngiven efter vad den beräknar, och testerna körs. Andra steget bryter ut utskriften på samma sätt, så att ett funktionsnamn ersätter kommentaren, och testerna körs igen. Tredje steget är Inline Variable: variabeln `total` används nu bara på ett ställe, så anropet får ta dess plats. Efter varje steg gör programmet exakt samma sak som förut och skulle kunna committas för sig:

```
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}`);
}
```

Först nu läggs totalsumman på fakturan till, och det kräver en rad i fakturakoden som anropar `orderTotal`. Inget i de tre refaktoreringsstegen krävde ett nytt test, eftersom inget observerbart ändrades; det är testerna som redan täckte kvittot som bevisar det.

Infrastrukturkod refaktoreras också, och där står mer på spel, eftersom namnen syftar på resurser som körs. Terraforms dokumentation varnar för att en ändrad adress för en resurs i konfigurationen som standard är en instruktion att ta bort den befintliga resursen och skapa en ny. Att byta namn från `aws_instance.a` till `aws_instance.web` för läsbarhetens skull skulle alltså ersätta en server. Sedan Terraform v1.1 registrerar ett `moved`-block i stället den gamla adressen, och nästa plan uppdaterar adressen utan att ta bort något:

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

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

I tidigare versioner hänvisar dokumentationen till kommandot `terraform state mv` för samma flytt. Hur som helst är refaktoreringen säker först när planen har lästs och visar en flytt, inte en borttagning och ett nyskapande.

Det är lättare att hålla isär sådana ändringar och arbete med ny funktionalitet när historiken säger vad som är vad. Commitstandarden som beskrivs i portföljposten här på webbplatsen om att låta fyra repon följa en och samma historikstandard låter varje commitmeddelande börja med ett konventionellt

prefix som anger typen av ändring, och en hook för commitmeddelanden upprätthåller det. Portföljposten, som gäller arbete utfört i juli 2026, räknar 26 refaktoreringar i historiken för det största repot vid sidan av 156 commits för funktioner och 133 rättningar. Den säger också ärligt att hooken kontrollerar formatet och inte innehållet. En commit som är märkt som refaktorering måste fortfarande genom sina tester och en kodgranskning visa att den har lämnat beteendet i fred.

Var det går fel

Beteendeändringar smyger med i refaktoreringen. En utvecklare städar i en funktion och rättar ett fel i den i samma commit. När ett test fallerar kan ingen se vilken av de två som var orsaken, och den som granskar koden måste läsa varje rad som en möjlig ändring av beteendet. Ha en hatt på i taget: refaktorera i en commit, ändra beteendet i nästa, och om ett test fallerar under refaktoreringen, ångra det senaste steget i stället för att lappa vidare på det.

En stor omstrukturering kallas refaktorering. En gren står i veckor med ett bygge som inte fungerar, medan koden sägs ”refaktoreras”. Det är omstrukturering i Fowlers mening, och den bär just den risk som refaktorering är gjord för att undvika. Dela upp arbetet i steg som vart och ett lämnar systemet i fungerande skick och kan slås samman var för sig; där det inte går, kalla det en omskrivning och planera det som en.

Det finns inga tester som visar att inget har ändrats. Bevarat beteende är ett påstående, och utan tester är det en gissning. Innan du rör kod som saknar tester, skriv tester som låser fast vad den gör i dag, även där det ser fel ut, och rätta egenheterna i separata ändringar efteråt. Fowlers tumregel från hans exempel på förberedande refaktorering gäller här: varje gång lusten kommer att köra koden och granska utdata med egna ögon, skriv ett test i stället.

Ett namnbyte i infrastrukturkod ersätter resursen. Ett snyggt nytt namn i Terraform ser ut som en refaktorering vid kodgranskningen, och planen föreslår i tysthet att ta bort en databas och skapa en tom. Registrera varje namnbyte och varje flytt in i eller ut ur en modul med ett `moved` -block, och betrakta varje borttagning i planen för en ändring som var tänkt som refaktorering som ett misslyckat steg.

Ett gammalt `moved` -block tas bort. Moved-block samlas på hög, och att ta bort dem ser ut som vanlig städning. Terraforms dokumentation kallar det en ändring som bryter kompatibiliteten: varje konfiguration som fortfarande står på den gamla adressen kommer då att planera att ta bort det befintliga objektet i stället för att flytta det. Behåll de historiska blocken, kedja ihop dem när ett objekt byter namn två gånger, och ta bara bort dem från en privat modul när alla som använder den har kört apply med den version som innehåller dem.

Refaktoreringen väntar på en egen fas. Ett team som skjuter upp refaktoreringen till en särskild period längre fram fortsätter att lägga till funktionalitet i kod som stretar emot, och en period som är planerad till senare är lätt att stryka när en deadline flyttas. Fowlers råd går åt andra hållet: refaktorera som en del av varje ändring, först för att göra ändringen lätt och efteråt för att lämna resultatet tydligt, så att arbetet aldrig behöver en egen plats i planen.

Läs mer

- Definition Of Refactoring — Martin Fowlers definitioner som substantiv och verb, så som han ger dem i sin bok.
- Refactoring — Fowlers webbplats för boken, om varför refaktorering hör till den dagliga programmeringen, och med hela katalogen.
- Extract Function — katalogens post för en av de vanligaste refaktoreringarna, med dess motsats och namnet från första upplagan.
- An example of preparatory refactoring — Fowler formar om sin egen kod i små steg innan han lägger till ny funktionalitet och förklarar de två hattarna.
- Refactor modules — Terraforms dokumentation om `moved` -block, namnbyten, uppdelning av moduler och varför gamla block bör stå kvar.

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 refaktorering?” 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/refactoring/