Programmeringssprog

Sidst ændret: 07/19/2018 14:11:07

"…"

- …

Først nogle generelle betragtninger om det at lære at programmere. Dernæst ser vi på de fire sprog-generationer og hvordan et program kommer fra kildetekst til udførelse. Vi gennemgår en række mål vi har med programmer og slutter af med at introducere begreberne syntaks og semantik.

Forudsætninger:

Tålmodighed.

Når man står over for at skulle lære at programmere, er man ofte i den samme situation som en nyfødt der skal lære at tale.

Autodidakt

Foregår det autodidakt er der sjældent nogen systematisk indlæring. Man ser andres programmer og høster egne erfaringer ved at lave programmer man selv har brug for eller lyst til at lave. Det betyder at ens programmeringsevner bliver baseret på talent og behov. Man tillægger sig en række uvaner og får ofte et urealistisk indtryk af egne evner til at programmere i almindelighed.

Undervisning

Anderledes stiller det sig når man i en undervisningssituation skal lære at programmere. Her er det en ekstern person og ikke én selv, der stiller kravene til programmerne, og der laves en række øvelser der skal give en alsidig og dybere forståelse af programmering.

Det væsentlige er *hvad* man siger

Det minder om undervisning i engelsk eller fransk. Man laver en lang række øvelser, der skal give et ordforråd og en række sætninger man kan bruge i praktiske situationer. Det er ikke med det samme meningen, at det man siger skal være særlig bevinget. Senere i forløbet opnår man at kunne formulere tanker mere frit og det endelige mål er at kunne tænke i sproget, så man er tilbage ved det væsentlige: *Hvad* man siger og ikke på hvilket sprog man siger det.

Tænke algoritmisk

Programmeringsprog adskiller sig fra almindelige sprog. Når vi i skolen lærer et nyt sprog har vi allerede lært vort modersmål, vi har lært at tænke i et sprog. Anderledes stiller det sig med et programmeringssprog. I programmering kommer det nemlig ind i billedet, at det primært drejer sig om at lære at tænke i den logik og med den systematik, der ligger i sproget. Det er derfor det ikke er Java, men det algoritmiske der er det væsentlige. Om man kan huske at stave til **switch** kan være evig lige gyldigt, hvis man kan tænke algoritmisk skal man nok klare sig.

Programmering er meget svært

Tag ikke fejl, det er givetvis sværere at lære at programmere end (næsten) alt andet. Datalogistudiet ved universiteterne er det studie hvor færrest gennemfører af alle studier. Det største problem ved at lære at programmere ligger i den anderledes tankegang man skal vende sig til. Jeg mindes en studerende der under en øvelse med while-løkker beklagede sig: "Jeg ved godt hvordan en while-løkke fungerer, men hvad skal jeg gøre?". Det er et meget tydeligt symptom på at det er tankegangen man lærer til sidst - at det er den der er det sværeste. Hvordan løser man så dette problem. Man må gøre det med eksempler. Eksempler der fører den studerende igennem tankeforløb der fører frem til løsningen, det er utilstrækkeligt altid kun at vise løsningen uden kommentarer. Den studerendes udbyttet af at se løsningen er mindre end udbyttet ved at får forklaret *hvordan man når frem til den*.

Sprog repræsenterer kulturer

Det siges, at en stor del af et folks kultur ligger i sproget, på samme måde er det med programmeringssprog. De repræsenterer også en bestemt kultur. Lige som der findes forskellige sprogstammer, der kan være vidt forskellige i deres opbygning, er der inden for programmering også forskellige retninger, der repræsenterer forskellige idéer mht. hvordan en computer skal programmeres. Java hører til den nyeste og mest dominerende af disse sprogstammer: den objektorienterede stamme.

1. Do you speak Jaba?

Når man har lært et programmeringssprog vil man måske undre sig over at computeren faktisk ikke forstår det. I computerens barndom i 50'erne skulle de programmeres vha. tallene 1 og 0. Man var nød til at gå computeren helt i møde, og indrette sig fuldstændig på dens betingelser.

Assemblere

50'ernes 0'er og 1'ere blev hurtigt afløst af assembler. Assembler kaldes også symbolsk maskinkode. Selve betegnelsen maskinkode knytter sig til 0'er og 1'ere, og det symbolske ligger i at man bruger forkortelser for disse maskinkoder. Senere blev assembler mere avanceret og det er ikke altid, at der er en direkte en-til-en sammenhæng mellem et symbol og en maskinkode. Betegnelsen "symbolsk maskinkode" er derfor ikke længere så brugt.

Generationer

Disse to kaldes 1. og 2. generation, og der skulle komme flere generationer

Generation	Betydning
1	Maskinkode
2	Assembler (Symbolsk maskinkode)
3	Højniveau-sprog (Lisp, Prolog, C++, Java)
4	4GL = 3GL + DB

Tabel 1: *Sprogenes generationer*

Højniveau-sprog

3. generation er de såkaldte højniveau-sprog som f.eks. Java tilhører. Højniveauet ligger i at der er meget langt fra sproget til maskinkode - der er på ingen måde nogen en-til-en sammenhæng.

Embedded database

4. generation føjer en database til. Man betegner denne database som "embedded" fordi selve sproget understøtter arbejdet med en database.

Hvordan realiserer man disse sprog, for computeren forstår jo kun maskinkode.

Tolke oversætter for os

Tidligt fik man den idé at lave programmer der virkede som **tolke** mellem mennesket og maskinen, og siden da har vi bevæget os bort fra maskinen og stillet større og større krav til tolken om at kunne **oversætte** vores tanker til et sprog som maskinen kan forstå. Desværre er det meget vanskeligt at lave disse tolke på menneskets præmisser. Der er stadig i høj grad tale om sprog, der er meget afhængige af, at det er en computer der er i den anden ende. Det vil vare end og rigtig mange år før tolkene vil være så udviklede, at man ved talegenkendelse kan forelægge dem et problem, og efter måske at have svaret på et par supplerende spørgsmål, få dem til at løse det. Vi må derfor stadig møde computeren langt henne ad vejen.

Compiler

Den største gruppe af tolke er **compilere**. Compilere fungerer ved at de oversætter hele den samlede algoritme før den forelægges computeren. Oversættelsen sker til maskinkode og det oversatte program kan umiddelbart køres på computeren.

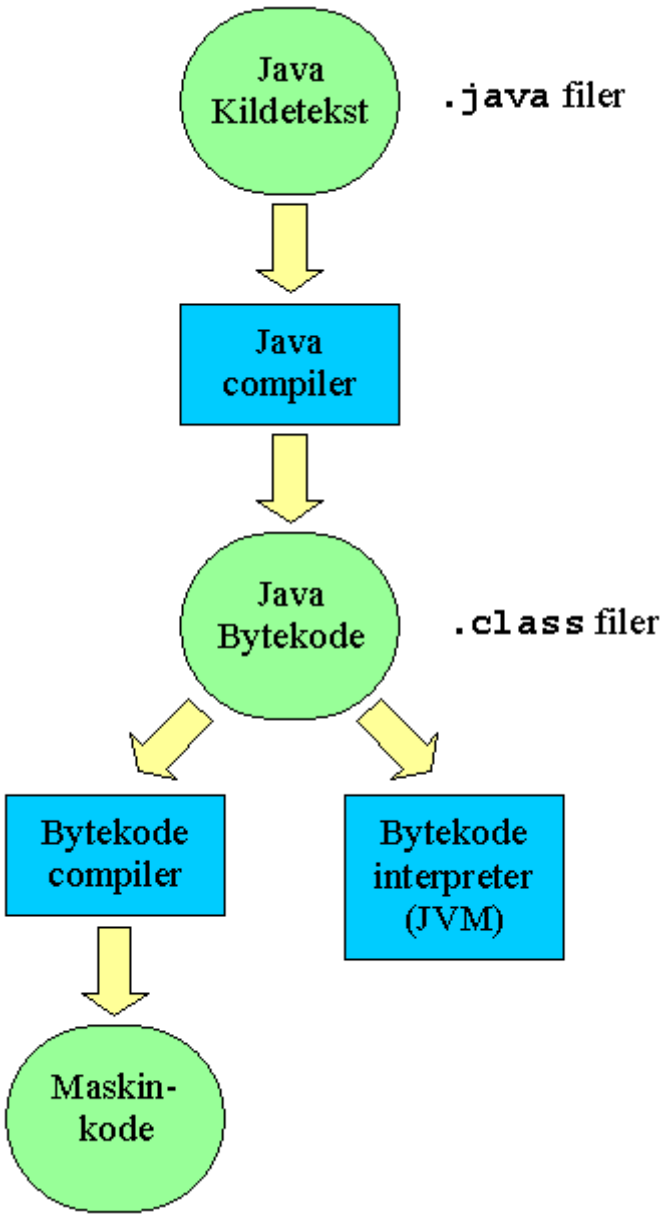
Interpreter

Den anden gruppe af tolke er **interpretere**. De fortolker/oversætter løbende programmet mens det udføres. Dvs. at den samme kode bliver "oversat" igen og igen. En interpreter simulerer at den er en computer der umiddelbart forstår den kode som programmøren har lavet. Et program der interpretes er derfor normalt langsommere end et program der er compileret.

Java kan både compileres og interpretes

Hvad så med Java - er det compileret eller interpretet? Ja! både og. Det har nemlig ikke noget som helst med sproget at gøre. Java kan compileres og det kan interpretes, det kommer an på tolken, bare der sker det der står i programmet. Man vil nok ofte støde på folk der siger at Java er langsomt, men det er rent sludder. Sun, der har udviklet Java lægger op til at Java skal afvikles på en virtuel maskine, en interpreter, men det er ren politik - det har intet med sproget at gøre! Sun lægger mere præcist op til at Java compileres til "byte-kode", der så interpretes af et andet program, der kaldes Java Virtual Machine (JVM). Med andre ord Java er både compileret og interpretet.

Lad os se en oversigt over hvordan et Java-program kan komme vejen fra programmørens kildetekst til udførelse:



Figur 1:
Vejen fra kildetekst til udførelse

Interpretere bidrager til at gøre programmer

Som man ser, i bunden af figuren, kan man ende med enten at compilere eller interprete programmet. Hvis man compilerer programmet er det lige så hurtigt som ethvert andet program der er compileret. Men hvorfor bliver de fleste Java-programmer interpretet? Det gør de for at være

platforms-uafhængige

platformsuafhængige. Platformsuafhængig betyder at programmet kan afvikles på enhvert computer. Hvis det skal være muligt at afvikle det samme Java-program på en hvilken som helst computer kræver det at der findes en interpreter til enhver slags computer. Det gør der i vidt omfang, og har man lavet et Java-program og oversat det til bytekode kan man kører denne bytekode på enhver computer der har interpreteren installeret.

Write once - run everywhere

Man kalder denne egenskab ved Java-programmer: "Write once - run everywhere". En egenskab der har været voldsom opreklameret fordi en softwareudvikler kan nøjes med at udvikle et program for at kunne sælge det til enhver computerejer.

Java er selv en platform

Man har fra forskellig side sagt at Java ikke er platformsuafhængig - Java ér en platform. Det er på sin vis også rigtigt, men det kan siges om enhver interpreter. At Java selv er en platform har skabt visse stridigheder mellem Sun og Microsoft. Microsoft har lavet verdens mest udbredt operativsystem, Windows, og efterhånden som JVM har udviklet sig er den blevet mere og mere en komplet platform, et operativsystem i sig selv; hvilket skaber balladen: Hvem har kontrollen over operativsystemet. Det er den almindelige opfattelse af Microsoft har skabt sin markedsdominans på baggrund af at det var dem der havde kontrollen med operativsystemet. Bill Gates har gjort dette så godt at han i visse kredse er lagt for had (næsten), men personligt kan jeg dårligt bebrejde ham for noget jeg ikke ville have gjort meget anderledes. Dog måtte Microsofts produkter gerne være mere stabile. Nogen gange bliver man i tvivl om det rent faktisk er den blå skærm der er firmaets logo :-)

2. Mål

Et "godt" program

Når vi vælger et programmeringssprog og konstruerer programmer har vi nogle mål - nogle kvaliteter vi søger opfyldt. I første række er det at opfylde kravene til programmet - hvad det skal kunne. Det man kalder kravspecifikationen. Ud over dette er der er dog en række grundlæggende mål, som er væsentlige at holde sig for øje; hvis man vil sikre sig at grundlaget for et "godt" program skal være til stede.

Opfyldelsen af disse mål beror både på programmeringssproget og på os som udviklere. Jo mere sproget selv gør for at opfylde målene, jo lettere bliver det for os at opnå dem, men vi skal selv udnytte mulighederne, ellers er de uden værdi.

2.1 Det skal være let at formulere et program

Dette mål har været en drivende kraft for udviklingen blandt programmringssprog langt hen ad vejen. Det var den indledende smerte ved maskinkode der satte udviklingen i gang. Hvis tallene 0 og 1 var det nemmeste i denne verden at programmere med, ville vi aldrig have lært ordet "programmringssprog" at kende.

Vælg moderne sprog

Programmeringssprogets bidrag til dette mål, får man bedst udbytte af ved at vælge et relativ nyt sprog. Fortran, Algol eller Cobol vil *ikke* være sagen. Det er dog ikke nok med det! Man skal også vælge et sprog der passer til den opgave der skal løses. Det er et spørgsmål om hvilket programmeringsparadigme der passe bedst til opgavens løsning. Programmeringsparadigmer er ikke et emne vi vil se på her, men man skal være varsom med at opdyrke den holdning at der er ét programmeringssprog der er det bedste - for det er der ikke. Der er måske ét man kan tillade sig personligt at regne for det bedste i de fleste sammenhænge, men man må ikke mene *altid*!

Hvad kan man selv gøre, ud over at vælge et godt sprog? Hvordan kan man selv bidrage til at gøre det lettere at formulere sig? Man må være indstillet på, at det kræver en indsats at blive god til at bruge sproget.

Verdens scene skifter og der kommer med mellemrum nye sprog på banen, som man skal arbejde med. Man starter som udgangspunkt med at programmere i det nye sprog som om det var det gamle. Man bruger mest de ting i sproget der ligner dem fra det gamle sprog, og hvis man ikke kan det, brokker man sig!

Lære sproget rigtigt

Det er afgørende at man arbejder alle sprogets muligheder igennem, at man øver sig, at man arbejder med det på sprogets præmisser. Hvis man tror man kan programmere i et nyt sprog uden at lære det rigtigt, uden at efteruddanne sig, så beder man selv om de problemer man får.

2.2 Det skal være let at forstå et program

Programmørens ansvar

Læseren skal naturligvis kende det pågældende sprog, men det skal ikke alene være muligt men også let at forstå kildeteksten. Det er i første række op til programmøren at gøre sit program forståeligt. Naturligvis er sproget lavet så dets opbygning gør det mere eller mindre forståeligt, men det falder primært tilbage på kendskabet til sproget. Hvis man kender det, betyder det ikke så meget om der er mange forkortelser og lignende. Hvad kan programmøren så gøre for at øge læsbarheden af kildeteksten?

Kommentarer øger læsbarheden

I næste kapitel skal vi se hvordan man kan lave kommentarer. Alle moderne sprog giver mulighed for at programmøren kan skrive kommentarer i kildeteksten. Sådanne kommentarer kan naturligvis formuleres i naturligt sprog, og derved kan man skrive hvad man finder nødvendigt, alt efter hvor kompliceret det er at forstå kildeteksten.

Ensartet anvendelse

En anden ting er måden man formulerer sig på. Alle sprog indeholder en række sproglige konstruktioner som man bruger utallige gange. Ved altid at skrive disse på en ensartet måde bliver det lettere at læse kildeteksten, og dermed bliver den også nemmere at forstå. Det er også et punkt vi vender tilbage til i næste kapitel.

2.3 Det skal være let at finde fejl

Opdage fejl

For at kunne finde en fejl, må man opdage den. Vi vil i næste kapitel identificerer tre forskellige fejltyper, der opstår i tre forskellige situationer. Rent teoretisk kan man diskutere om en fejl man aldrig opdager er en fejl? Hvis alle er glade og tilfredse kan det vel være lige meget, eller?

Finde årsag

Når man har opdaget en fejl kan vejen fra registrering af fejlen til indentifikation af årsagen til den være vanskelig. Det primære problem ligger i at finde det "sted" i kildeteksten hvor fejlen "er". Ordene er sat i anførselstegn, fordi det nogle gange er samspillet mellem dele af vores program, der giver problemer. I disse situationer kan det være en designfejl, som kræver mere vidtgående ændringer i programmet.

Modularitet

Der findes forskellige principper man anvende når man designer og skriver sit program. Et af disse kaldes modularitet. Ved at opbygge programmet i moduler kan man begrænse det område af kildeteksten; hvor en given fejl kan befinde sig. Vi skal se mere på moduler under objektorienteret programmering.

Læsbarhed

Endelig falder letheden ved at finde fejl også tilbage på hvor let det er at læse programmet.

2.4 Det skal være let at rette fejl

Når man har fundet fejlen - ved hvor den er, og forstår den; hvor let er det så at rette den?

Umiddelbart skulle man måske mene at det blot var et spørgsmål om at ændre lidt det pågældende sted i kildeteksten, men det kan være mere kompliceret end det.

At rette fejl

Hver gang man ændrer noget i et program kommer den øvrige del af programmet til at fungerer sammen med noget

giver nye fejl

der nu er anderledes. Det skulle gerne resultere i at det nu virker, men det kan også betyder at fejlen nu er en anden - at andre dele af programmet nu ikke virker!

Modularitet begrænser smittefare

Hver gang man ændre noget i et program løber man risikoen for at introducere nye fejl. Igen er en modulær opbygning af programmet medvirkende til at begrænse disse fejls smittefare. Hvis man ændrer i ét modul vil denne ændrings konsekvenser ofte holde sig indefor modulet, og det vil være betydelig lettere at undgå introduktion af nye fejl.

2.5 Programmet skal klare enhver situation der opstår mens det kører

Det lyder umiddelbart som en selvfølge, men det er det sjældent.

"Det virker men..."

Når en gruppe af studerende laver et projekt, og man er mere eller mindre "heldig" med at få tildelt brugerrollen, oplever man normalt én af to ting. Enten kan man sætte sig ned og få programmet til at opføre sig mærkeligt eller helt gå ned, i løbet af få minutter. Eller, hvis de studerende er lidt mere erfarne: At man hele tiden får at videre hvad man ikke må gøre, for så virker det ikke (endnu). Det er naturligvis helt i orden under udviklingsprocessen, men selv færdige programmer kan man ofte få til at bukke under.

Exceptions

En måde hvor på man kan gøre et program mere robust, er ved at bruge exceptions. Exceptions findes i de mere moderne objektorienterede sprog som C++ og Java. Vi skal i et senere kapitel se hvorledes man kan anvende exceptions til at klare fejlsituationer på en velordnet måde.

2.6 Programmet skal være hurtigt

Effektivitet

Et program skal være hurtigt - det siger sig selv. Ventetid er irriterende, og jo hurtigere et program er, jo mindre ventetid. Den hast hvor med en program løser en opgave kaldes generelt for dets tidskompleksitet eller man taler om hvor effektivt det er.

Compilerede programmer er generelt hurtigere end interpretede. I dag er computerne dog blevet så hurtige, at de har det tilstrækkelige overskud til, at kun få opgaver ikke tilfredsstillende lader sig løse af et interpretet program.

Mindre dele køres meget

Når man taler om effektivitet skal man være opmærksom på at det normalt kun er mindre dele af programmet der reelt behøver at være hurtige. De største dele af programmet har normalt ingen problemer med effektiviteten, selv om man ikke har gjort sig nogen overvejelser vedrørende deres

hastighed, da man skrev dem. Det skyldes at det typisk er mindre dele af programmet der bruger langt den største del af udførelsestiden.

Effektivitet kan svække indlæring af andre emner

Det er typisk for begyndere at de gør dem alt for mange bekymringer vedrørende effektivitet. Effektivitet er væsentligt når man laver et rigtigt systemer, men i en indlæringssituation skal man ikke bekymre sig om effektivitet med mindre det netop er emnet for det man studerer. Et godt eksempel er når man skal lære objektorienteret programmering. I den situation laver man ting der er mere eller mindre ineffektive (indirection giver altid et effektivitetstab), men hvis man begynder at bekymre sig om dét, svækker man indlæringen betydeligt. Det er først når man er god til objektorienteret programmring, at man kan begynder at gøre sig overvejelser i retning af at gå på kompromis med principperne for at øge effektiviteten - ikke et sekund før!

3. Syntaks og Semantik

Hvordan staver man til syntaks og hvad betyder semantik?

Ovenstående spørgsmål indeholder selv svaret, og er en god huskesætning [*FKJ*].

Grammatik

Syntaks er **grammatik**. Det er spørgsmålet om hvordan ord staves og hvilke regler der gælder for opbygningen af sætninger i sproget.

Betydning

Semantik er spørgsmålet om hvad det betyder. Det er det der giver sproget mening.

Når man skal beskrive et programmeringssprog kan man gøre det ved at beskrive sprogets syntaks og semantik. Ved hjælp af disse to, kan man opnå en præcis og fuldstændig beskrivelse. En sådan beskrivelse er dog sjælden god, når man skal lære at programmere. Beskrivelserne er formaliserede og de er primært tænkt som en reference man kan ty til, hvis man er i tvivl om detaljerne i et sprog.

BNF

Syntaks beskrives i almindelighed med BNF (Bachus-Naur-Form), men vi vil normalt beskrive syntaks mere eller mindre uformelt. Derfor vil vi ikke her se nærmere på BNF.

Semantik kan beskrives på mange måder, men en formel beskrivelse af semantik har primært kun teoretisk interesse, da enhver semantisk beskrivelse kan formuleres relativ kort med almindelig tekst. Vi vil derfor ikke anvende nogen formelle beskrivelser af semantik.

Repetitionsspørgsmål

1. Hvilke problemer har en autodidakt med at lære at programmere.
2. Hvor meget adskiller det at lære at programmere, sig fra det at lære f.eks. fransk eller engelsk?
3. Hvorfor er det svært at lære at programmere?
4. Hvordan kan man lære andre at tænke på en ny måde?
5. Hvad ligger der i betegnelsen "højniveau-sprog"?
6. Hvad er en compiler?
7. Hvad er en interpreter?
8. Hvorfor er et interpretet program normalt langsommere end et program der er compileret?
9. Hvad vil det sige at være platforms-uafhængig?
10. Hvordan gør man det lettere at formulere et program?
11. Hvordan gør man det lettere at forstå et program?
12. Hvordan gør man det lettere at finde fejl?
13. Hvad er det sværeste ved at rette fejl?
14. Hvordan gør man det nemmere at rette fejl?
15. Hvorfor skal programmer være hurtige?
16. Hvorfor behøver man ikke altid bekymre sig om effektivitet?
17. Hvad er syntaks?
18. Hvad er semantik?

Svar på repetitionsspørgsmål

1. Som autodidakt lærer man det man synes der er nødvendigt og man tillægger sig uvaner som ellers ville blive rettet af en underviser.

2 Når man lærer et fremmed sprog kan man formulere de samme sætninger som findes på ens modersmål. I programmering er det nogen helt anderledes sætninger man skal lære at formulere i et fremmed sprog.

3 Fordi det er svært at lære at tænke på en ny måde.

4 Ved at se mange eksempler med forklaringer af hvilke tanker der har ført frem til resultatet.

5 At sproget adskiller sig meget fra maskinkode - at der ikke er nogen en-til-en sammenhæng.

6 Et program der oversætter kildeteksten til maskinkode.

7 Et program der læser kildeteksten og simulerer udførelsen af den.

8 Fordi et compileret program kan køres direkte af computeren, mens et interpretet program skal køres af en andet program.

9 At et program umiddelbart kan køres på enhver computer.

10 Ved at vælge et moderne sprog, lærer det og øve sig meget.

11 Ved at bruge kommentarer og skrive på en ensartet måde.

12 Ved at opbygge sit program i moduler er det lettere at finde fejl, da man ofte hurtigt kan se i hvilket modul de må befinde sig.

13 At rette dem uden at der opstår andre fejl.

14 Med en opdeling i moduler vil effekten af de ændringer man foretager ved fejlretning i ét modul ikke så nemt sprede sig og skabe nye fejl i andre dele af programmet.

15 Hvis programmer er langsomme kan der opstå ventetid, enten for brugerne eller for andre dele af systemet.

16 Fordi kun mindre dele af et program kører det meste af tiden. Der er derfor store dele af programmet, hvis ineffektivitet man ikke vil mærke, og som derfor ikke behøver at være toptunede.

17 Det er de grammatiske regler, der gælder for et sprog.

18 Det er betydningen af sproget.