Om afprøvning

Mads Rosendahl

6. april 1999

Denne note drejer sig om den særlige form for afprøvning man kan bruge når man i en rapport skriftligt skal dokumenterer den udførte afprøvning. Når man i første omgang afprøver sit program er det normalt med det formål at finde så mange af de fejl der måtte findes i programmet. Man kan sige at man her har en destruktiv indstilling idet man søger at få frembragt fejl i programmet. Den skriftlige fremstilling har derimod et ganske andet formål. Her skal man søge at give læseren tiltro til pålideligheden af programmet. Til det formål er det ikke nok at vifte lidt med hænderne og med stor overbevisning påstå at man skam har fundet en hel masse fejl i programmet.

1 Udvikling af programmer

Udviklingen af et større stykke programmel kan ofte inddeles i en række separate faser. Det er sker ikke sjældent at faserne kan overlappe tidsmæssigt og at man på grund af erfaringer fra en senere fase skal ændre i arbejdet fra en tidligere fase. Alligevel kan det være nyttigt at tænke på en sådan opdeling under arbejdet.

- Analyse/design. Problemstillingen afklares, algoritmer vælges.
- Programmering. En løsning udtrykkes i det valgte programmeringssprog.
- Indkøring. Fejlfinding. Man overbeviser sig selv om programmets pålidelighed.
- Afprøvning. Struktureret test af programmet med henblik på dokumentation over for aftagere af programmet.

Denne note drejer sig særligt om den sidste fase, men man kan med fordel have afprøvningen for øje allerede på et tidligt tidspunkt i udviklingsforløbet.

Der er ofte noget forvirring om begreber som “indkøring”, “afprøvning” og “test” og ordene bruges mange gange i flæng. Vi vil her bruge begreberne som ovenfor. Indkøring er så normalt den destruktive process — man søger at få programmet til at “gå ned” ved at køre programmet med en række mulige inddata. Afprøvningen er derimod konstruktiv — man skal søge at overbevise en læser om et programs pålidelighed.

2 Afprøvningsstrategi

Når man går igang med et afprøvningsforløb er det værd at overveje hvilken grad af pålidelighed man stræber efter. Hvis man er ved at lave et program som man kun selv skal bruge og som

man kun skal bruge et par gange, ja så er pålidelighed måske ikke så vigtigt. Skal det derimod bruges af andre og skal det bruges mange gange, så er pålidelighed væsentligt vigtigere. En del af afprøvningen er derfor at fastlægge hvilken grad af pålidelighed man stræber efter.

Mange datalogi-projekter drejer sig om at vise at et eller andet er muligt og det sker ofte ved at konstruere et prototype-program. Tanken er at man så senere kan udvikle et egentligt program der gør det hele ordentligt — med ordentlige fejlmeddelelser, kontroller etc. I en sådan situation kan man ofte argumentere for at pålideligheden af prototypen ikke er så vigtig så længe programmets muligheder kan vises. Enhver datalog med respekt for sig selv bør imidlertid være istand til at gennemføre en grundig og veldokumenteret afprøvning af et program og det bør man vise i et projektforløb i løbet af sin uddannelse — også selvom alle projekterne omhandler prototype-programmer. Tid er ofte et problem i et projektforløb og man kan derfor vælge at begrænse sig til overveje og vælge en afprøvningsstrategi. Man beskriver så hvordan programmet kan afprøves, men undlader at dokumentere selve afprøvningen.

3 "White-box" versus "Black-box" afprøvning

Afprøvninger kan i formen deles i "white-box" og "black-box" afprøvninger afhængig af om man i afprøvningen baserer sig på kendskab til den faktiske programkode eller ej. "White-box" afprøvninger baserer sig altså direkte på programmets tekst og består typisk i at vise at alle programmets dele er rimelige og gør det forventede. I "black-box" baserer man sig i stedet på programmets funktion og man skal så vise at programmet også gør det man påstår det gør.

Typisk er "white-box" metoder bedst i indkøringsforløbet; ikke mindst i forbindelse med gruppearbejde hvor det kan være vigtigt under indkøringen at overbevise sig om at den kode de andre i gruppen har skrevet er rimelig. I afprøvningsforløbet er det normalt bedst at bruge "black-box" metoder idet en læser så kan følge argumentationen uden at kende eller forstå detaljer i den valgte implementation. Læseren kan koncentrere sig om hvad programmet skal kunne og hvad det gør og glemme detaljer om hvordan det gør det.

Vi skal senere omtale nogle "white-box" metoder men her omtale en vigtig "black-box" afprøvningsstrategi — ekstern afprøvning.

4 Ekstern afprøvning

I en ekstern afprøvning tænkes alle de mulige inddatasæt til programmet inddelt i grupper efter om man kan forvente at disse inddata i en eller anden forstand kan tænkes at blive behandlet ens af programmet. I den eksterne afprøvning køres programmet så med mindst en repræsentant for hver gruppe. Det centrale problem i en sådan afprøvning er altså at gruppere de mulige inddata på fornuftig og overbevisende vis. Metoden omtales ofte som afprøvning baseret på opdeling i ækvivalensklasser[1]

Udgangspunktet for en ekstern afprøvning er en beskrivelse af hvad programmet skal kunne. Det kan være en specifikation af mere eller mindre formel karakter eller det kan være en brugervejledning/manual. I store træk søger den eksterne afprøvning at afprøve alle påstande i denne

[1] Ækvivalensklasser er egentlig et matematisk begreb, men det bruges kun her i overført betydning. Hvis man f.eks. tænker sig mængden af tekststrenge kunne en ækvivalensrelation være at to tekststrenge er ækvivalente hvis de har samme længde. I dette tilfælde er ækvivalensklasseopdelingen af mængden så en opdeling i grupper (eller klasser) af indbyrdes ækvivalente tekststrenge. Man vil så have en klasse med alle tekstrenge på et tegn, en klasse med tekststrenge på to tegn osv. Der vil også være en ækvivalensklasse som kun består af den tomme streng.

beskrivelse af programmets funktionalitet. Det store problem i afprøvningen er imidlertid at strukturere afprøvningen så læseren med rimelighed kan overbevise sig om at man er nået ud i alle interessante hjørner.

Valg af ækvivalensklasser. Hvis man har et program som skal indlæse et tal mellem 1 og 10 og gøre et eller andet med det, så kan man formode at programmet opføre sig nogenlunde ækvivalent for alle disse tal og man kan så nøjes med en repræsentant for denne ækvivalensklasse. Derudover vil opføre sig anderledes for tal mindre en 1 og for tal større end 10. Man kan derfor også prøve med 0 og 11 for at kontrollere fejludskrifter. Er man lidt mere mistroisk vil man nok sige at tallene 1 og 10 i en eller anden forstand må blive behandlet særligt i programmet så man tester også med disse tal. Der er ikke nogen facitliste for rigtig valg af ækvivalensklasser, men man lægger en grad af mistro til hvad der kan blive behandlet forskelligt og vælger så klasser og repræsentanter for klasser ud fra dette.

Normalt vil man ikke afprøve kombinationer af inddata. Hvis et program f.eks. skal indlæse tre tal mellem 1 og 10 skal man ikke prøve alle kombinationer af grupperinger af disse tal. Gør man det vil det give alt for mange og sikkert også for uinteressante grupperinger. Var tallene dog f.eks. sidelængder i en trekant vil det være naturligt at prøve visse særlige kombinationer af længder — f.eks hvis en side er længere end summen af de to andre.

Det vanskellige i at opdele mulig inddata i ækvivalensklasser er at gøre det så det overbevisende bliver en fuld dækning af mulig inddata. De færreste læsere kan overskue mere end 7-10 klasser ad gangen så vælger man en for finkornet opdeling bliver det ikke overbevisende selvom det jo burde give en større grad af sikkerhed. Kun sjældent kan man nøjes med 7-10 klasser, men så må man opdele i nogle større grupperinger, feks. dele korrekte og fejlbehæftede inddata, eller dele op efter forskellige dele af inddata.

Præsentation af ækvivalensklasser. Når man har valgt ækvivalensklasser har man måske også allerede valgt repræsentanter for klasserne. Ellers skal man gøre det og så skal man finde en overskuelig måde at præsentere opdelingen på over for læseren. Præsentationen kan så være i form af et skema hvor man angiver klasser, hvilke repræsentanter, hvilken uddata man forventer og i hvilken kørsel det pågældende afprøves.

klasse	inddata	forventes	testkørsel
tal : 1-10	7	...	kørsel 1
grænse : 1	1	...	kørsel 2
grænse : 10	10	...	kørsel 3
ulovlig : < 1	0	...	kørsel 4
ulovlig : > 10	11	...	kørsel 5

Ind- og uddata fra kørsler præsenterer man så i appendiks — og naturligvis opdelt og nummereret så man let kan finde dem, hvis man er særlig interesseret.

Bliver et sådant skema for langt kan man, som omtalt, dele det op i nogle grupperinger så en læser kan følge med og overbevise sig om at man får en dækkende afprøvning. Det er vanskelligt præcist at sig hvor langt er for langt. Følger inddata f.eks. en grammatik med 50 produktioner vil det være naturligt at sikre sig at inddata ifølge alle produktioner prøves. I det tilfælde vil en læser nok godt kunne overskue et skema der viser i hvilke kørsler de 50 produktioner prøves.

5 Andre metoder

- Kodegennemgang. Er man en del af en projektgruppe kan man med fordel inddrage de øvrige medlemmer i fejlfindingsprocessen. En yndet metode er den såkaldte "program-inspektion" eller "kodegennemgang" hvor programmøren gennemgår sin kode med de øvrige medlemmer. Programmøren forklarer ideen i sin kode og de øvrige medlemmer prøver at finde fejl ved at stille spørgsmål. Metoden beskrives nærmere i [1] og [2].

- Intern afprøvning. En intern afprøvning skal sikre at alle linier i programmet prøves, alle valgsætninger prøves med de mulige udgange og løkker prøves med 0/1 og flere gennemløb. Herved søger man at finde brister i programmets indre logik og undersøge om der er unødvendige eller ubrugte dele af programmet. Metoden beskrives også i [2]

- Programverifikation. For mindre programmer eller programdele kan man ved brug af matematiske metoder bevise at programmet opfylder formelt udtrykte specifikationer. Metoden er særlig interessant for særlig kritisk software som f.eks. styresystemer til elevatorer el. lign.

- Afprøvning af programdele. Ved afprøvning af store programmer — ikke mindst i et objekt-orienteret eller et modulært programmeringssprog kan det være en fordel at afprøve de enkelte større programdele (klasser eller moduler) for sig idet man så kan basere afprøvningen på hvad disse programdele skal kunne. Man skriver så små drivprogrammer (eller stubbe) som kunstige omgivelser til disse programdele og afprøver ellers som var det et helt program. Se også [1].

- Brug af testudskrifter. Andre aspekter kan også have stor betydning under indkøringen. Det er vigtigt at have en konsekvent og organiseret stil mht. til indrykninger i programmer, opdeling i flere filer, brug af klasser og nedarvning. Under indkøringen er det ofte vigtigt at bruge testudskrifter med en detaljegrad der gør det let at finde fejl. Det kan være en ide at lade testudskrifter være styret af nogle betingelser så de kan slåes fra når man ikke er interesseret. Ved at bevare testudskrifterne i det endelige program — men lade dem være slået fra — kan det være nemmere senere at vedligeholde programmet.

References

[1] Programafprøvning, Pascal version. H. B. Hansen. Datalogiske noter nr. 12, RUC 1995.

[2] Kunsten at teste edb-programmer. Glenford J. Myers. Borgen 1984.