Kapitel 5: Kunsten at få en computer til at adlyde (II): Softwareudvikling

Henrik Kragh Sørensen Mikkel Willum Johansen

22. april 2022

Indhold

5.1 Softwareudvikling som teori og praksis

Hvis vi også vender tilbage til karakteriseringen af datalogi i kapitel 1, så noterer vi os, at der blandt datalogiens „vidtfavnende“ områder er der både teoretisk viden om computeres og beregningers natur og mere teknologisk viden om, hvordan man kan bringe IT i anvendelse. I det foregående har vi beskæftiget os med de indre forbindelser mellem en algoritme og den software, som implementerer den. Men en mindst lige så stor praktisk og filosofisk udfordring i de datalogiske videnskaber er den *proces*, hvorved et problem identificeres og løses ved hjælp af software. Her er der altså tale om *softwareudvikling* som både en praksis og en række videnskabsteoretiske refleksioner over samme. Og dette emne er vigtigt både for dets indre værdi som en væsentlig del af datalogens værktøjskasse og for de filosofiske, etiske, økonomiske, juridiske og politiske aspekter, som softwareudvikling også spænder over.

For at få hold på softwareudvikling både som praksis og som del af videnskabsfaget datalogi, er det nyttigt at udforske nogle forbindelser, analogier og kontraster. Man kan med god ret tænke på softwareudvikling som *anvendt* og *teknologisk videnskab*, der er *handlingsorienteret* (se også Kragh, 2003, s. 177–180). Heri ligger, at teknologiske videnskaber går ud over at ønske blot at *forstå* forhold i deres domæne, men også gerne vil angive metoder og teknikker til at *kontrollere* og *manipulere* forhold i domænet med (menneske-centrerede)

formål (se også kapitel 2). I det omfang softwareudvikling er en teknologisk videnskab, handler den altså om at teoretisere over 'metoder og processer til udvikling af software' og herunder at udlede 'best practices', som kan komme udviklere til gavn. Men selvom det kunne være oplagt at understrege forskellen mellem denne form for teknologisk videnskab og en mere *grundvidenskabelig del* af datalogien, så er forskellene sværere at fastholde en som så: Både grundvidenskab og teknologisk videnskab stræber efter at producere ny viden, begge former udøves typisk i de samme kontekster (universitetsinstitutter eller private forskningslaboratorier), begge former hører til uddannelseskanonen for datalogistuderende, og ofte udføres begge former af de samme individer eller forskningsgrupper.

En anden dimension af sammenligning kunne være at fokusere på de metoder, som bruges i softwareudvikling. I den praktiske udførelse af softwareudvikling er der ofte tale om en mere induktiv metode, baseret i nogen grad på *trial-and-error*. Dette står i kontrast til den teoretiske datalogis brug af formelle metoder, og er noget, vi skal udforske i afsnit 5.5. Og i studiet af softwareudvikling inddrages kvalitative, sociologiske metoder for at kunne komme til at forstå aktørernes interesser og handlinger. I ret direkte forlængelse heraf kunne en sidste, foreløbig form for sammenligning handle om de nærmest tilgrænsende felter. Hvor teoretisk datalogi og programmering måske ligger tæt på matematik og sprogteori i det akademiske landskab, så trækker softwareudvikling både på studier af menneskelig adfærd, designvidenskab og organisationsteori.

Hvis man forsøger at identificere og beskrive forskellige overordnede positioner i debatterne om software-engineering — både de faglige og de filosofiske debatter — så kan man i overensstemmelse med Gruner (2011, s. 284) pege på i hvert fald tre forskellige *parter*:

1. En *formalistisk* position, der tager udgangspunkt i, at software-udvikling handler om matematiske objekter, og derfor argumenterer for brug af matematiske metoder til at sikre god software, fx igennem formel verifikation eller model-verifikation.

2. En anden *ingeniør-centreret* position, som fremhæver stringente produktionsprocesser og workflow-modeller, som de kender fra fx mekaniske fabrikker, til at sikre god softwareudvikling.

3. En tredje *humanistisk* position, der fremhæver de sociale interaktioner undervejs i udviklingen af software og de konsekvenser, software har i det omgivende samfund.

Et nuanceret, filosofisk syn på softwareudvikling bør tage alle disse perspektiver med i betragtning og kende til deres respektive fokuspunkter og begrænsninger.

I dette kapitel skal vi derfor studere softwareudvikling som videnskaben om de processer, der indgår i at fremstille og anvende software. Det skal vi gøre ud fra en *systemisk tilgang til software*, en *inkrementiel tilgang til udvikling* og en *bruger-orienteret tilgang til kvalitet*.

5.2 Hvis vi ikke kender historien, gentager vi den

Det berømte citat fra den spanske filosof og essayist George Santayana (1863–1952) — „Those who cannot remember the past are comdemned to repeat it“ (Santayana, 2011, s. 172) — har en dobbelt betydning inden for softwareudvikling: På den side er det altid vigtigt at have et historisk blik for, hvordan vores moderne ideer om softwareudvikling er blevet udviklet igennem de sidste 50 år (Northover m.fl., 2008), siden man begyndte aktivt at tænke over, hvad *software engineering* kunne og skulle være. Men på den anden side

er indholdet af moderne softwareudvikling også på mange måder formet af gentagne *læringsmomenter*, hvor fiaskoer af forskellige former har fået dataloger og andre til at tænke over, hvordan software kan og bør udvikles. Nogle af disse episke fiaskoer handler om overskridelser af budgetter og tidsplaner, nogle af dem handler om, at man ikke kan forudse teknologiens fremtidige udvikling, og nogle af dem handler om, at softwarens begrænsninger ikke kunne indses på forhånd, men måtte opdages og læres — af historien!

Et af de mest indflydelsesrige *læringsmomenter* var, da International Business Machines (IBM) næsten brækkede halsen på det nu notoriske operativsystem *OS/360*. Siden umiddelbart efter Anden Verdenskrig havde IBM domineret markedet for store computere til myndigheder, banker og store virksomheder. Pointen med *OS/360* var, at IBM ønskede at udvikle deres computere og deres software i retning af kompatibilitet, således, at kunder skulle kunne opskalere deres hardware uden at være nødt til at udvikle og adaptere deres software, sådan som man ellers havde været nødt til. Man skal huske, at man i 1950'erne og 1960'erne — af grunde, som vi bl.a. skal vende tilbage til i kapitel 7 — *ikke* anså software som et selvstændigt produkt. Derimod opfattede man software som noget, der blev leveret og evt. vedligeholdt enten af hardwareproducenten eller af en inhouse computer-afdeling i de største virksomheder (Haigh, 2002).

Men det storslåede projekt med at lave en fælles platform i form af et delt *operativsystem*, som skulle sætte forskellige stykker hardware i stand til at køre *samme* stykke software, var lige ved at gå helt galt for IBM. Lederen af projektet, Fred Brooks beskrev efterfølgende sine erfaringer i en bog, der kom til at sætte dagsordenen for en del af det, der var galt med den tidlige, før-professionaliserede form for software-udvikling (Brooks, 1995): Blandt Brooks' konklusioner er i hvert fald en lille håndfuld i dag gået over som folklore iblandt dataloger, fx 1. 'no mythical man-month', 2. 'no silver bullet', 3. 'second system effect' og 4. 'irreducible number of errors' (se figur 1). Brooks observerede også, at projekter 'bliver forsinket en dag ad gangen' i takt med at planerne skrider fx ved at nye features bliver foreslået implementeret. Det fik ham til at insistere på, at software skulle ses som en opsplitning mellem arkitektur og implementation: Arkitekturen er den samlede begrebslige beskrivelse af systemet, og der skal stræbes efter høj grad af klarhed og integritet. Hvordan arkitekturen så implementeres er en anden sag, og her kan alle mulige former for lokal kompleksitet komme til at vise sig nødvendig — men det skal ikke 'forurene' den arkitektoniske integritet. I stedet skal implementationen selv være læselig, dokumenteret og afprøvet.

No mythical man-month	Brooks' måske mest provokerende erfaring var, at programmører ikke er udskiftelige og skalerbare ressourcer i softwareudviklingen. Man kan ikke ved at ansætte dobbelt så mange programmører få produktionen af software til at gå dobbelt så hurtigt. Så fraværet af den *mytiske mandemåned*, han refererer til, gik imod gængs tænkning om effektivisering og ledelse i fx *Fordismen*.
No silver bullet	En anden provokerende erfaring var, at Brooks ikke mente, at et enkelt greb — som i en enkelt strategi eller filosofi — omkring softwareudvikling ville kunne garantere pålidelige systemer, der overholdt budgetterne. I stedet skulle softwareudvikling selv anses som en kompleks proces med behov for styring og tilpasning af mange forskellig slags.
Second system effect	En af de konkrete udfordringer, som Brooks var stødt på med *OS/360* var en gradvist stigende kompleksitet af opgaven i takt med, at designere og programmører fik øjnene op for endnu flere smarte muligheder, de kunne bygge ind i systemet. Så selvom Brooks ligesom andre anbefalede, at den første udgave af ethvert stykke software blev anset som en pilot-version, der kunne kasseres helt, så identificerede han også et modsatrettet træk i retning af, at version 2 bliver *for* kompleks — og mere generelt et teknologisk *pull* i retning af, hvad der er muligt nærmere end hvad der er ønskeligt (eller bestilt).
Irreducible number of errors	En yderligere grund til softwarens kompleksitet og fordyring kommer, mener Brooks, af selve processen med at gøre software pålidelig: Hver gang vi retter en fejl risikerer vi nemlig at snige *nye* fejl ind i software — enten lokalt, hvor vi rettede, eller et andet sted i systemet, hvor vi ikke havde tænkt over, at rettelsen kunne have konsekvenser. Pessimistisk udtrykker han dette fænomen som, at der er en nedre men positiv grænse for, hvor få fejl, et stykke software kan have. Vi kan aldrig, siger han, opnå fejlfri software!

Figur 1: Opsummering af Brooks' erfaringer fra *OS/360* baseret på hans bog *The Mythical Man-Month* (Brooks, 1995).

Erfaringerne fra *OS/360* og en række andre storslåede fiaskoer i at planlægge og styre softwareudvikling var en medvirkende årsag til, at NATO i 1968 afholdt den første af en række konferencer om softwareudvikling. Denne konference i Sonnenbichl i Garmish i Sydtyskland er siden blevet anset som fødselsstedet for begrebet *software engineering*, og nogle af de store diskussioner om programmeringssprog og deres betydning for udvikling af god software (se kapitel **??**) fandt sted under og i kølvandet på dette møde.

At NATO havde en så stærk interesse i softwareudvikling i 1960'erne er værd at bemærke: Foruden universiteter, de allerstørste private virksomheder, banker og finansvæsenet var militæret nogle af de få, der havde eget computerudstyr at udnytte og vedligeholde. Og i regi af militæret og rumudforskningen var nogle af de største datalogiske gennembrud sket.

En af de interessante cases handler om et af de tidligste realtidssystemer med grafisk brugerinterface, som det amerikanske militær begyndte udviklingen af i årene umiddelbart efter Anden Verdenskrig i samarbejde med forskere fra Massachusetts Institute of Technology (MIT). Formålet var at bygge en computer, kaldet *Whirlwind*, der kunne hjælpe med at simulere flyvning. Til den brug udviklede man mange nye teknologiske gennembrud, blandt andet en ny måde at opbygge computerens hukommelse på og muligheden for at vise data på en CRT-skærm. Projektet var altså en teknologisk succes i den forstand, at det bidrog til mange ting, vi i dag tager for givet. Men det var voldsomt dyrt og slugte det meste af flådens forskningsbudget. Så i begyndelsen af 1950'erne *forvandlede* man *Whirlwind computeren* til brug for et andet prestige-projekt, kaldet *SAGE*, som var et tidligt semi-automatisk sporingssystem til luftforsvaret (Smith, 1976). Og i den form blev grafiske interfaces til en uomgængelig del af at *semi-automatisere* kritiske processer. Men lektien om, at *Whirlwind* var blevet så dyrt hang alligevel ved i militære kredse, hvor pålideligheden af softwaren i forvejen var af høj prioritet.

5.3 Hvad er god software?

De foregående nedslag i softwareudviklingens historie har vist nogle af de interesser, der kan være på spil for at sikre god software. Og flere af disse hensyn spiller stadig en afgørende rolle, så det er værd at begynde overvejelserne over, hvordan man skaber *god* software med at reflektere over, hvilke kvaliteter, software egentlig kan og bør have.

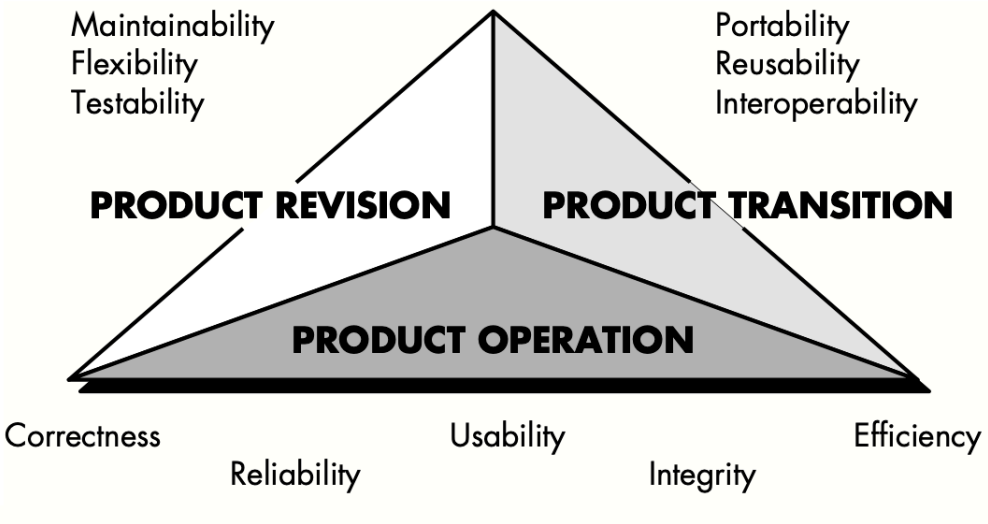
Software — i forstanden programmer og systemer, som brugere kan efterspørge, anskaffe, bruge og evt. videreudvikle — kan med nogen ret betragtes som *artefakter*: De er skabt af mennesker for at tjene et formål. Og på den måde kan man stille nogle af de samme kriterier op for kvaliteter ved software, som man kan for andre *artefakter* som biler, skruetrækkere eller computere (se fx Floridi, Fresco og Primiero, 2015, s. 1200). Mange af disse kvaliteter, som vi efterspørger om fx computere vil være, at de fungerer pålideligt — for eksempel, at de tænder, når vi trykker på knappen, og altid fremviser konsistent opførsel, således at vi kan stole på, at de virker også i fremtidige situationer. Andre kvaliteter, som vi er vant til at efterspørge, er anvendelighed til at løse de opgaver, vi støder ind i, således at teknologien gør det *nemmere* for os at opnå *vores* mål (som brugere). En stor del af disse kvaliteter er *designet ind i artefakten*, således, at skruetrækkeren fx har et solidt håndgreb, men som vi så i kapitel 2, er der også mange af kvaliteterne, der i virkeligheden afhænger af det eller de *teknologiske systemer*, som den enkelte *artefakt* indgår i: Der er ingen kvalitet i en stjerneskruetrækker, hvis jeg ikke kunne købe stjerneskruer at skrue i med den.

Alligevel adskiller software sig på nogle puntker fra mere traditionelle artefakter, fx ved at software ikke *forgår* — i hvert fald ikke på samme måde, som fx kærven på min skruetrækker bliver slidt (Floridi, Fresco og Primiero, 2015). Og derfor er diskussioner om softwarekvalitet også på nogle områder forskellig fra til tilsvarende diskussioner om hardwarekvalitet, selvom det var et oplagt sted at hente inspiration for kvalitetssikring (Pressman, 2001, s. 212).

En anden forskel ligger måske i softwarens forhold til de mere 'eksterne teknologiske systemer' og arbejdsprocesser, som den skal indgå i. Hvor der er meget lang tradition for hamre og søm og en klar kontinuitet fra hestevogne til biler, så er indførelsen af software i mange teknologiske systemer en mere brat og omdefinerende begivenhed. Det giver mening, at *datamatisering* og i dag mere udbredt *digitalisering* både er blevet set som det store rationelle fremskridt og effektivisering og som fremmedgørende og angstfremkaldende i dele

af befolkningen: For hele arbejdsgange og professioner er blevet afhængige af, at software er en medspiller og ikke en modspiller i den teknologiske udvikling.

Så hvis vi skal indfange kvaliteter ved god software, kan vi tage udgangspunkt i den omfattende liste af faktorer i god software, som blev udarbejdet af eksperter fra General Electric Company (GE) i slutningen 1970'erne på bestilling af det amerikanske luftvåben. Ved et stort litteratur-survey fandt forskere frem til 14 faktorer, der sammen med kvaliteten af dokumentation, tidsforbrug og pris på softwaren indgår i vurderinger af softwarens kvalitet (McCall, Richards og Walters, 1977, bd. I, s. 2.7). Listen er siden blevet bearbejdet yderligere og forsimplet, således at den fx i Pressman (2001, s. 509) kan sammenfattes i tre overordnede komponenter: softwarens operation, transition og revision (se figur 2).



Figur 2: McCall's 14 kriterier anskueliggjort som hørende til softwarens operation, transition og revision (Pressman, 2001, s. 509).

Mange af faktorerne er ret nemt genkendelige: For eksempel omfatter *korrekthed*, at programmet opfylder specifikationerne og indfrier kundens formål; *pålidelighed* indfangede for Jim A. McCall og hans kolleger, at man kan stole på, at programmet opfylder sit mål med den givne præcision; *integriteten* er udtryk for, at adgangen til programmet og dets data kan kontrolleres, så herunder hører privacy og sikkerhed; og *brugbarhed* måler den indsats, en bruger skal investere for at kunne lære og bruge programmet. Men det mest interessante ved figuren er nok, at softwarekvalitet i denne betragtning rækker (langt) ud over kvaliteten af brugen af produktet og også omfatter vedligeholdelse og transision af produktet. Det er faktorer, som lægger sig tættere op ad en ingeniørmæssig tænkning om systemer, der skal kunne vedligeholdes, opdateres og skaleres. Og selvom kataloget af faktorer er omfattende, er der også andre aspekter, som kunne tænkes med ind i softwarekvalitet, fx den produktionsforøgelse og relevans, som produktet medfører (Landauer, 1995) — det vil være en anden form for effektivitet end den, McCall anlægger, som handler om maskinressourcer — eller det stykke menneskelig og social 'engineering', der skal lægges, for at programmet bruges godt i organisationer og arbejdsgange (Glass, 1998). Disse kvalificeringer peger på, at softwarekvalitet er en meget bredere kategori end blot korrekthed og pålidelighed.

I en interessant sammenligning mellem nogle af de allerførste refleksioner i begyndelsen af 1960'erne over softwareudvikling som proces og de praktiske erfaringer 20 år senere (Boehm, 1987) får man et indtryk af, at nogle af de udfordringer, som stadig er relevante, har en lang historie og måske faktisk har en mere fundamental karakter:

- Specifikationernes præcision og testbarhed: Det bør være klart, hvad softwarens specifikationer skal være — og dette bør være tænkt, beskrevet og aftalt på forhånd. Og det skal være muligt at afgøre, om et krav i specifikationen er opfyldt eller ej.
- Softwarens grænseflader: Grænsefladerne — både i forhold til andre systemer og i forhold til brugeren — er en del af specifikationen, der er særligt vigtigt at få aftalt og godt dokumenteret. Filformater var længe en stor udfordring for integration af processer og udveksling af data, og vigtigheden af gennembrud som markup-language skal ses i det lys.
- Topstyret design og „lean staffing“ i projektets tidlige faser: W. A. Hosier beskrev det karakteristisk for den arkitektoniske, designmæssige tilgang til software: „The designers should not be saddled with the distracting burden of keeping subordinates profitably occupied“ og „Quantity is no substitute for quality; it will only make matters worse“ (Hosier, 1987, s. 321).

5.4 Hvordan organiserer man udvikling af god software?

En af de interessante videnskabsteoretiske vinkler på softwareudvikling kommer af at betragte software som arbejdsprocesser, både i produktionen af software og i de sociale og erhvervsmæssige sammenhænge, hvor software skal bruges (Ensmenger og Aspray, 2001). Dermed stiller der sig både interessante spørgsmål til ledelse og inddragelse og til den professionalisering af software, som også er et fremtrædende karakteristikum ved de sidste 50 år (se fx Ensmenger, 2010).

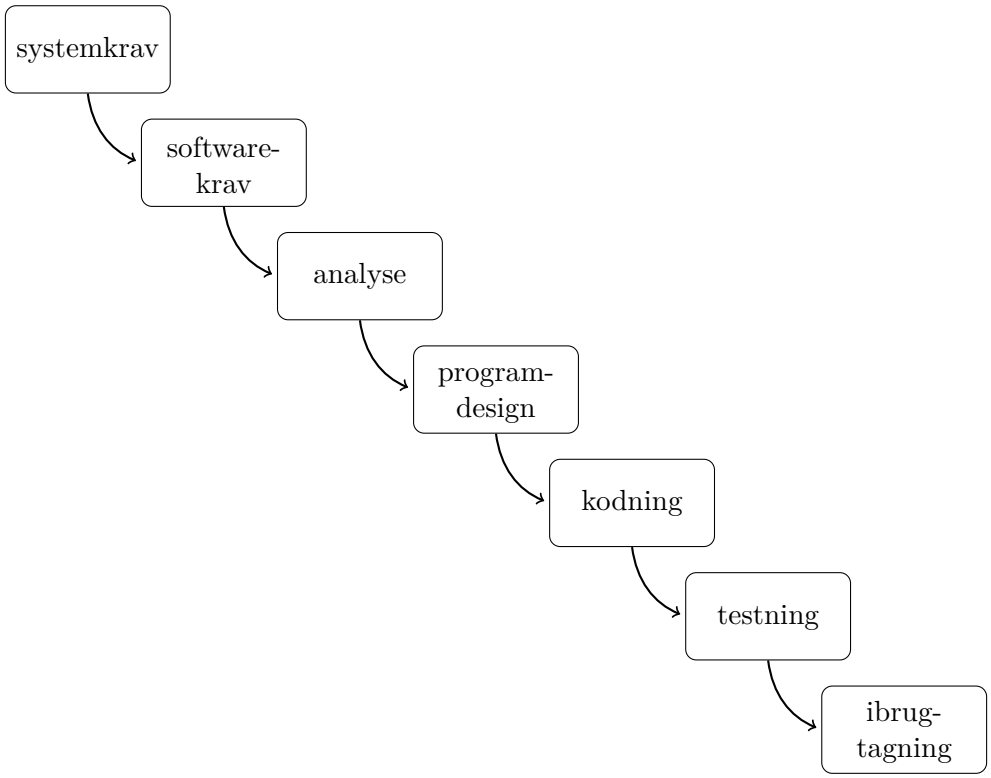
Som allerede beskrevet har organiseringen af god softwareudvikling været et særdeles dybtfølt behov både i civile og militære kredse, men det har ikke været uden problemer at finde frem til gode organisationsmodeller. For softwareudvikling er en krævende branche, som historisk alligevel var præget af lav social indtrædelsestærskel og en række idealer om autonomi, som var svært forenelige med sædvanlig amerikansk erhvervskultur (Ensmenger, 2003). Og hvor enkeltpersoner eller små, sammentømrede hold i starten kunne udvikle brugbare stykker software, er software blevet 'big business', både målt i omsætning og i antallet af beskæftigede på de enkelte projekter. Og både anekdotisk viden og forsøg på systematiske undersøgelser viser, at det er langt billigere at træffe de rette designvalg tidligt end at skulle ud og ændre efter release (Pressman, 2001, s. 14 taler om en vækst i forandringsomkostninger fra 1 i designfasen til mellem 1,5 og 6 i udviklingsfasen og helt op til mellem 60 og 100 efter release). En klassisk kritk af de meget topstyrede organisationsformer, kaldet 'katedralerne', i modsætning til mere demokratiske og flade strukturer, kaldet 'basarer', kan findes i Eric Steven Raymonds forsvar for Open Source Software (Raymond, 2001), men de relevante videnskabsteoretiske diskussioner er sådan set ikke betinget af forretningsmodeller.

For at komme til nogle af de videnskabsteoretiske refleksioner over udviklingsprocesser og organisering deraf vil vi først meget kort præsentere tre karakteristiske modeller for softwareudvikling: vandfaldsmodellen, model-drevet udvikling og agil udvikling. De tre tilgange repræsenterer en klassisk ingeniørmæssig og korporativ udviklingsmodel, en meget

formalistisk-orienteret model og en mere bruger-inddragende og i den forstand humanistisk tilgang. Der findes mange andre modeller og tilgange (se fx Northover m.fl., 2008, s. 87), særligt for objekt-orienteret analyse, design, implementation og test, men disse er udvalgt fordi de repræsentere væsentlige videnskabsteoretiske forskelle, og de kan i virkeligheden bringes i anvendelse i tænkning om (næsten) al slags software.

Vandfaldsmodellen

Den såkaldte *vandfaldsmodel*, som i sin kanoniske version blev formuleret fx af Winston W. Royce (1929–1995) fra Lockheed, som har store kontrakter med det amerikanske militær, bliver ofte trukket frem og kritiseret for dens rigide adskillelse af de forskellige processer, som indgår i softwareudvikling. Men historisk er det lidt ufortjent, for Royces version har (mindst) to vigtige kvaliteter: 1. Den sætter ord på (nogle af) de forskellige processer, som traditionelt har indgået i softwareudvikling (se figur 3): specifikation, analyse, implementation, test og ibrugtagning, og 2. Royces eget formål med artiklen var faktisk at diskuere og nuancere modellens opbygning — men det går sommetider glemt. I sin karikerede version er *vandfaldsmodellen* en *topstyret* (top-down) udviklingsproces, hvor de mest abstrakte trin skal foretages først og er forudsætninger for, at 'vandet kan spilde over' til næste trin i processen. Den er derfor blevet skældt ud for at argumentere for, at der ikke må skrives kode, før hele specifikationen er på plads. Men selvom denne designproces kunne virke tillokkende for mere traditionelle produktionsvirksomheder, viste allerede Brooks' erfaringer fra IBM, at softwareudvikling i praksis synes at være en meget mere flydende — næsten kaotisk — proces.



Figur 3: Royces simpleste vandfaldsmodel (Royce, 1987, s. 329).

Vandfaldsmodellen taler til en vision af, at software kan specificeres, designes, programmeres og testes i den rækkefølge og som adskilte skridt. Men allerede Royce talte for tidligt design for at afprøve muligheder, behovet for dokumentation også undervejs og imellem trinene og en filosofi om 'to-it-twice', fordi de første forsøg på fx implementation sjældent var optimale og lappeløsninger var (og blev anset for at være) en af kilderne til den enorme kompleksistet og fordyrelse, som software har været plaget af siden 1960'erne.

En del af kritikken og videreudviklingen af *vandfaldsmodellen* har, ligesom Royces formål, sigtet på at tage højde for, at information og beslutninger fra underliggende trin kan påvirke tidligere trin, uden at det betyder, at hele processen skal starte forfra. Et eksempel kunne være den rolle, som test spiller i processen (se også afsnit 5.5, nedenfor), hvor den karikerede model vil sigte mod at bruge tests til at opdage fejl ved systematisk afprøvning af ellers givne og opfyldte specifikationer. Men mødet med virkeligheden — og især med brugeren — har vist sig (også) at være en produktiv kilde til at finde mangler i de specifikationer, som vandfaldsmodellen vil forudsætte, at man har opstillet til at begynde med (se fx kapitel 3).

I sine kommentarer argumenter Royce også for at inddrage kunden undervejs i processen, men hans arguemnt er alligevel sigende for den topstyrede, kontraktfikserede tilgang, for kundeinddragelse skulle primært være med til at forebygge konflikter under overdragelsen af det færdige produkt. Så der er ikke så meget tænkt på samskabelse eller co-design som på informationsflow fra producent til kunde og milestones for ændringer i bestillingen.

Modelbaseret udvikling

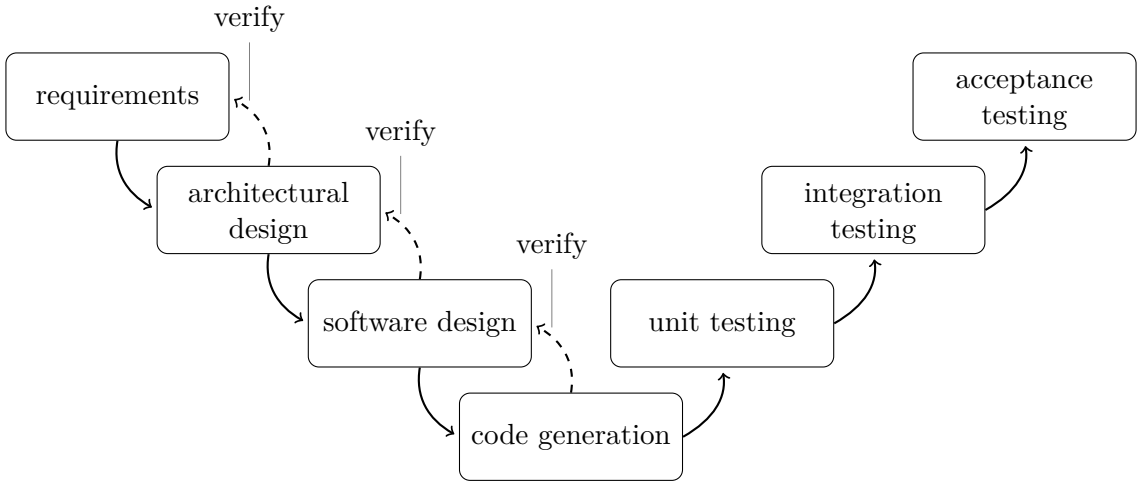
Der findes i dag adskillige forskellige videreudviklinger af den simple vandfaldsmodel, som forsøger at tage hensyn til kompleksiteterne i softwareudvikling. En af dem er såkaldt *modelbaseret udvikling* (MBD), som forsøger at kombinere mere formelle metoder med systematiske tests. Fortalere for denne tilgang arbejder ofte ud fra en V-formet udviklingsmodel som gengivet i figur 4.

Ideen med MBD er, som navnet mere end antyder, at man ønsker at udnytte nogle af de fordele, som modeller mere generelt giver, fx for at kunne tale om samme ting på forskellige abstraktionsniveauer afhængig af konteksten. Nogle fortalere (Brambilla, Cabot og Wimmer, 2017, s. 8) argumenterer for at erstatte Niklaus Wirths ellers berømte symbolske ligning

$$\text{algoritmer} + \text{datastrukturer} = \text{software} \tag{1}$$

med

$$\text{modeller} + \text{transformationer} = \text{software}. \tag{2}$$



Figur 4: Model-baseret udvikling (Bialy m.fl., 2017, s. 42).

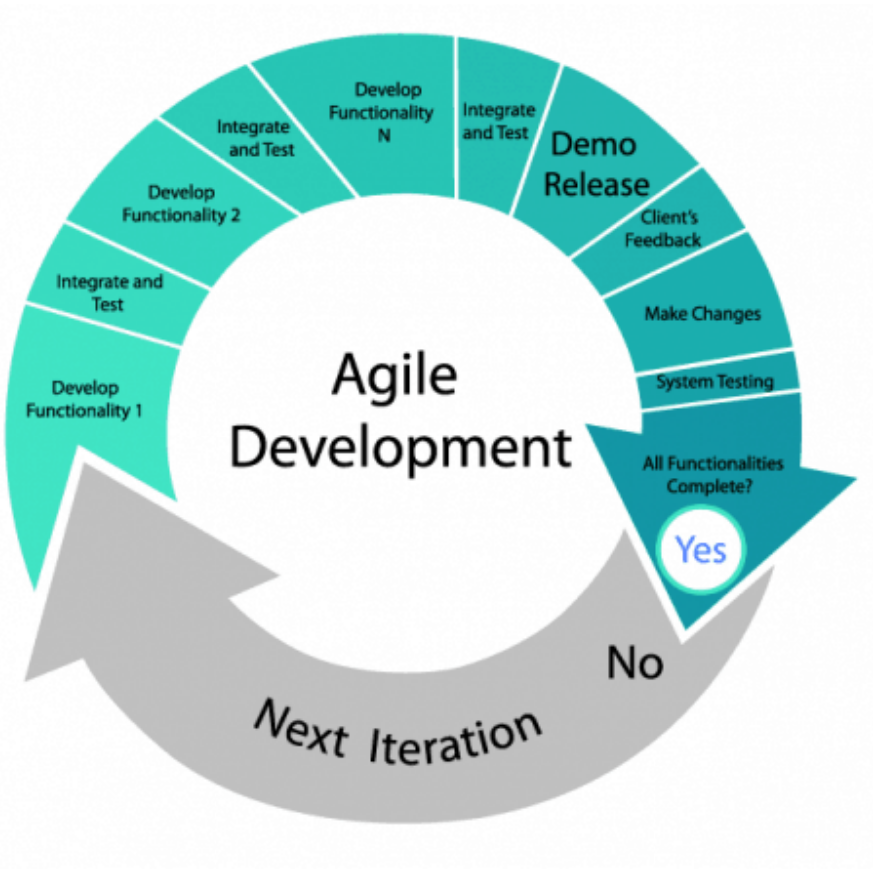
På overfladen kunne det ligne, at modeller *blot* er specifikationer, men i MBD er mekanismerne noget forskellige: Man taler om modelniveauet, hvor selve modellen er defineret, og realisationsniveauet, hvor løsninger er implementeret. Men det særlige ved denne tilgang er fokus på de afbildninger, der forbinder disse to niveauer, og som udgør et *automationsniveau* (Brambilla, Cabot og Wimmer, 2017, s. 10). Det afgørende er nu, at man kan tilføje et metaniveau til disse transformationer og dermed have en sammenhæng at argumentere om deres korrekthed inden for. Derved kombinerer MBD kaskade-elementet af vandfaldsmodellen med bevisstyrken af formel verifikation. Og vel at bemærke sker dette *i løbet af* udviklingsprocessen, således at den formelle verifikation ikke er en påklistret og voldsomt fordyrende tilføjelse til det endelige produkt.

Der findes adskillige forskellige måder at implementere denne modelbaserede tilgang på, herunder flere forskellige sprog og notationer til at udvikle og tænke om modeller i, hvoraf *UML* nok er det mest kendte. Og visionen — at man kan automatisere og formalisere store dele af softwareudviklingen — virker tillokkende i mange sammenhænge, ikke mindst i forhold til kritiske systemer.

Man kan anskue MBD som et forsøg på igennem automatisering at omgå nogle af de processer, som man har erkendt er fordyrende og tilføjer komplesitet til softwareudvikling. På den måde bliver det til et (blandt flere) eksempler på *metaprogrammering*, der er den yderste konsekvens af ideen om, at programmeringssprog skal understøtte menneskelig tænkning og skabning og ikke være bundet til den underliggende arktiektur (kapitel **??**).

Agile metoder

En radikalt anden måde at forsøge at omgå nogle af de samme omkostningstunge og forsinkende dele af vandfaldsmodellen har vundet stor indpas under navne som *agile metoder*. Hvor MBD holdt fast i en (i hvert fald i princippet) lineær udviklingsproces, så argumenterer man inden for agil udvikling for en cyklisk og iterativ proces, der fokuserer mere på at levere en strøm af synlige og brugbare prototyper end på at specificere og designe systemet til mindste detalje inden implementationen går igang (se figur 5).



Figur 5: Agile metoder illustreret som cyklisk, iterativ proces. [Figuren skal tilpasses]

Agile metoder er med andre ord baseret på et helt andet sæt værdier om softwareudvikling (citeret fra Northover m.fl., 2008, s. 91):

1. individuals and interactions over processes and tools,
2. working software over comprehensive documentation,
3. customer collaboration over contract negotiation, and
4. responding to change over following a plan.

Det agile ved denne filosofi består altså både i en accept af, at software er og vedbliver at være en proces (og altså i udvikling og flux), og at omstillingsparathed i mødet med virkeligheden og kunden kan være vigtigere end en rigid udviklingsproces. På den måde står agil udvikling også vinkelret på mange af de bestræbelser, der er gjort på at kvantificere softwareudvikling med henblik på dokumentation og måloptimering.

Blandt de kendetegnende faktorer ved agil udvikling er hyppig brugerinddragelse og justering af både mål og prioriteter i projektet. Dette lyder meget fristende i en tid karakteriseret af hurtig udviking og hård konkurrence inden for software, men det efterlader nogle af de klassiske kvalitetsmål som korrekthed og pålidelighed med meget mindre fokus end mere klassiske tilgange.

5.5 Kan man prøve sig frem til god software?

5.6 Hvad med brugeren?

Et af de områder, hvor de forskellige udviklingsmodeller hidtil har stillet sig meget forskelligt og haft forskellige ting at bidrage er brugerens rolle, særligt i forhold til design af interfacet mellem maskine og bruger. Dette område er et selvstændigt forskningsfelt, ofte kaldet *human-computer interaction* (HCI), men også en kreativ designproces i at skabe gode *brugerinterfaces* (UI) og gode *brugeroplevelser* (UX). Og såvel i forskningsfeltet som i de kreative processer indgår der en lang række metoder, der rækker ud over de tekniske og matematiske tilgange, som vi tidligere har diskuteret, og i stedet trækker på både psykologisk og sociologisk viden. Og dertil kommer en lang række mere dogmatiske tommelfingerregler for designbeslutninger, der handler om brugerens interaktioner med maskiner og andre brugere.

Nogle af de mest succesfulde brugerdesigns har med rette fået ikonisk status i det moderne samfund — tænk på mus og cursor, drag-and-drop, swipe-right, skriveborde og skraldespande på skærmen. Og det er interessant at overveje, hvordan relationen mellem disse designede, virtuelle artefakter og den omgivende virkelighed er. For skriveborde og skraldespande er der en klar *metaforsik relation*: At lægge et dokument på skrivebordet vil være en handling, der giver fysisk mening i 1800-tallet og giver virtuel mening i dag. Noget anderledes ser det ud med nogle af de andre elementer af brugergrænsefladen, som har en meget mere normativ, definerende og systemisk natur. Der er ikke nogen metaforisk relation, der siger, at jeg udtrykker noget som helst ved at flytte min pegefinger til højre på min telefon eller trække musen henover bordfladen. Her er i stedet tale om, at brugergrænsefladen definerer ny adfærd hos mennesker, og at denne adfærd kan vise sig at række langt ud over den oprindelige motivation for den. Når det sidste sker, har vi fået tilføjet en ny mening til en handling, som så siden kan (og bør) udnyttes i andre tilsvarende situationer, hvor det giver mening.

Gennem disse refleksioner har vi nærmet os de tre 'gyldne regler', som brugergrænsefladedesigneren Theo Mandel har opstillet (Mandel, 1997, kap. 5), og som ofte refereres:

1. Place Users in Control,
2. Reduce Users' Memory Load,
3. Make the Interface Consistent.

Hvert af disse dogmer udmøntes så i en række konkrete anbefalinger, som fx (under 1) at brugerens behov for at ombestemme sig ikke skal være en uønsket undtagelse for designeren, men noget, der aktivt understøttes, (under 2) at det er bedre at forlade sig på genkendelse end på genkaldelse, så brugergrænsefladen give et kognitivt off-load, og (under 3) at brugergrænsefladen, som vi jo lige sagde var en kreativ proces, alligevel bør holdes konform og konsistent mellem situationer og produkter.

En brugerflade og en brugeroplevelse er med andre ord noget, som udvikleren designer, og Mandels regler er opstillet for at minde om, at det ikke er designerens men brugerens kognitive og sociale behov, der skal tilgodeses ved gode designs.

Dertil rejser sig så selvfølgelig spørgsmålet om, hvordan man som udvikler får adgang til brugerens behov — det være sig den enkelte slutbruger såvel som den mulige kunde. Og her kommer den systemiske vinkel igen i spil, for man skal også som udvikler have øje

for, at man leverer et stykke software, hvis funktion kan være med til radikalt at forandre menneskers liv og hverdag (se mere i kapitel **??**).

Der findes selvfølgelig en række sociologiske metoder som observationsstudier, interviews og fokusgrupper, som kan kvalificere såvel den generelle analyse af softwarens formål som mere specifikt brugerens interaktionsbehov og -muligheder. Og nogle af disse metoder og anbefalinger kan tænkes ind i selve ingeniørprocessen (Seffah og Metzker, 2004), således som det fx hyppigt er tilfældet med agile metoder.

Sammenfattende kan man sige, at selvom UI er en design-aktivitet, er den teleologisk i den forstand, at den har et klart formål, nemlig brugerens målopfyldelse. Derved adskiller UI sig alligevel fra kunstneriske aktiviteter, og UI kan ses som en artefakt, der både er del af og på linje med software.

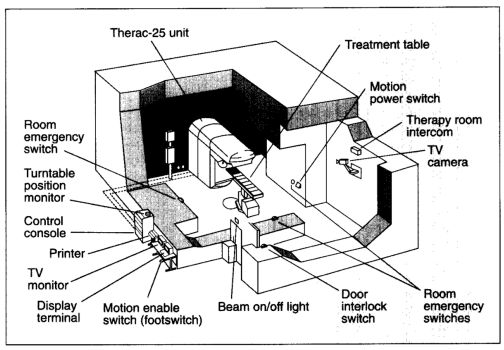
5.7 Når IT-projekter stadig fejler

Trods mere end 50 års fokus på udvikling af god software sker det stadig, at softwarerprojekter fejler ved enten at blive voldsomt forsinkede eller fordyrede eller ved at munde ud i produkter og services, som ikke bliver indoptaget af de kunder og brugere, for hvem de var udviklet. Hvis vi kort vender tilbage til *vandfaldsmodellen* kan man sige, at enhver fase deri og enhver overgang er en potentiel katastrofe, der venter på at ske.

Den danske datalog Erik Frøkjær har igennem årtier forsøgt at give sin akademiske viden om softwareudvikling videre til især offentlige software- og digitaliseringsprojekter. Blandt de grunde, som Frøkjær peger på, er der flere systemiske forhold, som han peger på, at man i princippet kunne undgå (se fx Frøkjær, 1987; Frøkjær, 2017). En af faktorerne kan kaldes utidig indblanding fra kundens side, fx ved at fremskynde ibrugtagning af systemer. Dette forhold peger jo i nogen grad på en udviklingsmodel, der reserverer tid og ressourcer til de enkelte trin i en autonom udviklingsproces. Og Frøkjærs observation peger dermed nok på, at agile udviklingsmål er dårligt forenelige med udrulning af store, definerende offentlige systemer. En anden faktor, som også kan identificeres har med effektivisering at gøre. For selvom digitalisering og software bærer løfter om automatisering og dermed effektivisering af menneskelige arbejdsgange, så er rækkefølgen bestemt ikke uvæsentlig: Udvikling, indførelse og indkøring af digitaliseringsprojekter er på den korte bane nok oftere dyrere end normal drift, men stiller så en effekt i udsigt på længere sigt. En tredje faktor har med det skel mellem styring/ledelse af projektet og udførelsen, som vi allerede berørte under professionaliseringen af softwareudvikling. For at kunne styre softwareprojekter er detailkendskab til processen nok vigtigere, end man måske oplever i andre produktionsbrancher. Hvis det er en empirisk observation, så peger den i retning af, at softwareudvikling faktisk har sin egen disciplinære identitet, som ledere af udviklingsprojekter — private som offentlige — skal have kendskab til og forståelse for (se også Limoncelli, 2019).

5.8 Therac-25: Når man ikke ved, hvor meget er nok

Et af de mest omdiskuterede eksempler på et fejlende computersystem opstod i USA og Canada i midten af 1980'erne. Det drejer sig om en strålekanon til kræftbehandling, som gik under navnet Therac-25. Ialt var der installeret 11 maskiner af denne type, ligeligt fordelt nord og syd for grænsen, og de var i hyppig brug i kræftbehandling (en typisk installation er gengivet i figur 6).



(a) En typisk installation (Leveson og Turner, 1993, s. 19).

(b) Selve strålekanonen (wikipedia).

Figur 6: Strålekanonen Therac-25.

Therac-25 blev produceret af firmaet Atomic Energy of Canada Limited (AECL), som dengang var et statsdrevet selskab (se Leveson og Turner, 1993). Selve strålekanonen var en videreudvikling af tidligere, mindre kraftige modeller, som AECL havde udviklet i samarbejde med en fransk partner. Både de tidligere modeller (*Therac-6* og *Therac-20*) og Therac-25 blev styret af en DEC PDP-11 computer, som var en af de mest udbredte 'små' computere. Men hvor computeren havde spillet en mindre rolle i de tidligere versioners brug, blev Therac-25 fra starten designet til at udnytte computerstyring. Hvor man i tidligere modeller havde haft fysiske sikringsmekanismer til at overvåge strålingen, blev mange af disse på Therac-25 i stedet sikret gennem software. Denne software var i sig selv en videreudvikling af software fra *Therac-6*, og den var udviklet over en længere årrække af en enkelt programmør og skrevet i PDP-11 assembler.

Den slags hospitalsudstyr skal igennem ganske rigide sikkerhedsanalyser, før maskinen kan tages i brug. Og da Therac-25 var klar i 1982/83 blev den testet af AECL, hvor de fokuserede særligt på omfattende test af softwaren på en hardwaresimulator under brugslignende forhold. De konkluderede, at øvrige softwarefejl ikke var omfattet og analysen, at software ikke slides op, og at kommende problemer derfor måtte forudses at stamme fra fejlagtig hardware eller tilfældige hændelser forårsaget af α-stråling eller elektromagnetisk støj (Leveson og Turner, 1993, s. 21). Denne analyse ledte ingeniørerne til et fejl-træ, som angav de forskellige sikkerhedsproblemer og tildelte dem sandsynligheder. Således blev sandsynligheden for at computeren angav en forkert dosis stråling, uden videre forklaring, vurderet til 10^{-11}.

Men fra 1985 begyndte brugerne af Therac-25 at opleve problematiske behandlinger. Og indtil maskinen blev tilbagekaldt i 1987 førte den til seks tilfælde af voldsomme, livstruende overdoseringer af stråling. Patienterne fik stråleskader, som betød at en måtte gennemgå en mastektomi, flere mistede førlighed i arme, skuldre og hofter, afhængig af hvor behandlingen havde sigtet. Selvom hospitalerne var i kontakt med AECL, og der blev udført ad-hoc justeringer af maskinen, blev årsagen ikke fundet og fjernet. Derfor blev de amerikanske myndigheder U.S. Food and Drug Administration (FDA) involveret i sommeren 1986, og en stor del af vores viden om casen stammer fra udredning og sagsanlæg. Man mistænkte elektriske fejl i fx justeringen af drejebordet, men fejlen skulle vise sig at ligge i softwaren.

Faktisk var der ikke tale om blot 1 softwarefejl, men om flere — og nogle af dem delte Therac-25 med maskinen *Therac-20*, hvor de bare ikke var blevet så kritiske, fordi fysiske

forholdsregler havde stoppet maskinen i at skade patienterne. Men den mest direkte fejl i Therac-25 var forbundet til en særligt mystisk omstændighed: Det så ud som om maskinen fungerede helt fint, når den blev installeret på hospitalerne — det var først efter nogen tid, og ganske uforudsigeligt, at problemerne opstod. Alligevel var problemerne jo gentagne, så det virkede usandsynligt, at de skulle være rene tilfældigheder. Ved en nærmere analyse af den software, der kontrollere inddateringen af dosis, fandt man imidlertid frem til en årsag, der kunne forklare både mønstret af overdoseringer og det faktum, at overdoseringerne var så voldsomme. På det tidspunkt var programmøren ikke længere ansat i firmaet, og han viste sig umulig at opspore til sagsanlægene.

Problemet lå i kommunikationen mellem tastaturet og computeren. Når der blev tastet på keyboardet, blev der sendt et *interrupt* til PDP-11, som så skulle suspendere sin øvrige aktivitet, aflæse keyboardet, gemme information om den anslåede tast og vende tilbage til sin oprindelige udførsel, som var langsomt at dreje strålekanonen. Denne interruptmekanisme er i dag standard i de fleste hardware- og operativsystemer, men på PDP-11 var den implementeret direkte i hovedprogrammet. Og problemet med overdoseringerne blev nu sporet tilbage til, at hvis man trykkede på tasterne tilstrækkeligt hurtigt efter hinanden — som man som bruger af apparatet ville kunne lære at gøre, når man blev vant til det — blev det ene interrupt ikke afsluttet, før det næste løb ind. Derfor fik man en forkert afvikling af aflæsningen, og den indlæste og gemte dosering var ikke længere det, lægen havde ordineret.

Denne episode fandt sted på et tidspunkt, hvor understøttelse af interrupts i operativsystemer ikke var udbredt — og jo i hvert fald på en maskine, der ikke understøttede en interrupt-stack. Men at dette skulle være et problem for softwaren var ikke noget, man havde forudset, før man opdagede, at det kunne være et problem. Efterfølgende er det blevet et særligt opmærksomhedspunkt i al software, der skal kunne håndtere samtidige hændelser.

I 1987 og i forbindelse med sagen beskrev Ed Miller, der var direktør i Center for Devices and Radiological Health, som er en del af FDA, hvad man kunne lære af denne ulykkelige case:

> FDA has performed extensive review of the Therac-25 software and hardware safety systems. We cannot say with absolute certainty that all software problems that might result in improper dose have been found and eliminated. However, we are confident that the hardware and software safety features recently added will prevent future catastrophic consequences of failure. (Miller 1987 citeret fra Leveson og Turner, 1993, s. 38)

Her påpegede Miller således, at *absolut sikkerhed* for fejlfrie softwaresystemer ikke er en mulighed, men samlet kan man se, at selv omfattende testning af kritisk software på hardwaresimulatorer i brugslignende situationer ikke var tilstrækkeligt til at finde fejlen i Therac-25. Fejlen lod sig kun trigge i *særlige* brugssituationer, nemlig når brugeren var blevet tilstrækkelig hurtig til at trykke på tasterne. Endvidere er man nødt til at huske, at hvis interrupt-overflow ikke er en del af testerens forventningshorisont, så vil det ikke blive samlet op og testet. På den måde er Therac-25-fejlen og opdagelsen af covert channels i *Bell-LaPadula-modellen* sammenlignelige: Ingen af dem tager højde for uforudset brug. Men hvor Therac-25 var testet empirisk, så var *Bell-LaPadula-modellen* en matematisk model, og dette siger, at ingen af metoderne er tilstrækkelige til at garantere, at man fanger uhensigtsmæssig brug før ibrugtagen. Casen var dermed også med til at fremskynde udviklingen af *system engineering* og *software engineering* som vigtige del af professionel programmering, og til at indføre rigide kvalitetstjek af software til kritiske systemer. For som Frank Houston fra det amerikanske FDA skrev:

> A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering. (Houston i 1985 citeret fra Leveson og Turner, 1993, s. 18)

Therac-25-casen er yderligere interessant, fordi den involverer så mange aktører med forskellige roller: programmøren, firmaet, hospitalerne, brugerne, patienterne, myndighederne og retssystemet. For hver af dem kan man lave en analyse af deres involvering og eventuelle ansvar — og i kapitel **??** vil vi tage den op igen i forbindelse med programmørernes professionelle ansvar.

Litteratur

Benington, Herbert D. (1983). „Production of Large Computer Programs". *IEEE Annals of the History of Computing*, bd. 5, nr. 4, s. 350–361.

Bialy, M. m.fl. (2017). „Software Engineering for Model-Based Development by Domain Experts". I: *Handbook of System Safety and Security*. Elsevier. Kap. 3, s. 39–64. DOI: 10.1016/b978-0-12-803773-7.00003-6.

Boehm, Barry W. (1987). „Software Process Management. Lessons Learned From History". I: *ICSE '87. Proceedings of the 9th International Conference on Software Engineering*. Washington, DC: IEEE Computer Society Press, s. 296–298. DOI: 10.5555/41765.41798.

Brambilla, Marco, Jordi Cabot og Manuel Wimmer (2017). *Model-Driven Software Engineering in Practice*. 2. udg. Bd. 3. 4. Morgan & Claypool. DOI: 10.2200/s00751ed2v01y201701swe004.

Brooks Jr., Frederick P. (apr. 1987). „No Silver Bullet. Essence and Accidents of Software Engineering". *IEEE Computer*, bd. 20, nr. 4, s. 10–19. DOI: 10.1109/MC.1987.1663532.

— (1995). *The Mythical Man-Month. Essays on Software Engineering*. Anniversary edition. Boston etc.: Addison-Wesley.

Ensmenger, Nathan (2003). „Letting the "Computer Boys" Take Over: Technology and the Politics of Organizational Transformation". *International Review of Social History (IRSH)*, bd. 48, nr. Supplement 11, s. 153–180. DOI: 10.1017/S0020859003001305.

— (2010). *The Computer Boys Take Over. Computers, Programmers, and the Politics of Technical Expertise*. Cambridge & London: MIT Press.

Ensmenger, Nathan og William Aspray (2001). „Software as Labor Process". I: *History of Computing: Software Issues*. Red. af Ulf Hashagen, Reinhard Keil-Slawik og Arthur Norberg. International Conference on the History of Computing, ICHC 2000. April 5–7, 2000, Heinz Nixdorf MuseumsForum, Paderborn, Germany. Berlin etc.: Springer, s. 139–165.

Floridi, Luciano, Nir Fresco og Giuseppe Primiero (2015). „On malfunctioning software". *Synthese*, bd. 192, s. 1199–1220. DOI: 10.1007/s11229-014-0610-3.

Fraser, Steven og Dennis Mancl (jan. 2008). „No Silver Bullet. Software Engineering Reloaded". *IEEE Software*, bd. 25, nr. 1, s. 91–94. DOI: 10.1109/ms.2008.14.

Frøkjær, Erik (1987). „Styringsproblemer i det offentliges edb-anvendelse". *Politica: Tidsskrift for Politisk Videnskab*, bd. 19, nr. 1, s. 31–56.

— (jun. 2017). „Finansministeriet". *Weekendavisen*, nr. 23.

Glass, R. L. (1998). „Defining quality intuitively". *IEEE Software*, bd. 15, nr. 3, s. 103–104, 107. DOI: 10.1109/52.676973.

Gruner, Stefan (feb. 2011). „Problems for a Philosophy of Software Engineering". *Minds and Machines*, bd. 21, nr. 2, s. 275–299. DOI: 10.1007/s11023-011-9234-2.

Haigh, Thomas (jan. 2002). „Software in the 1960s as Concept, Service and Product". *IEEE Annals of the History of Computing*, bd. 24, nr. 1, s. 5–13. DOI: 10.1109/85.988574.

Hosier, W. A. (1987). „Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming". I: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, s. 311–327.

Kragh, Helge (2003). „Hvad er videnskab?" I: Fink, Hans m.fl. *Universitet og Videnskab. Universitetets idéhistorie, videnskabsteori og etik*. København: Hans Reitzels Forlag. Kap. 3, s. 145–192.

Landauer, Thomas K. (1995). *The trouble with computers. Usefulness, usability, and productivity*. Cambridge og London: MIT Press.

Leveson, Nancy G. og Clark S. Turner (jul. 1993). „An Investigation of the Therac-25 Accidents". *IEEE Computer*, bd. 26, nr. 7, s. 18–41.

Limoncelli, Thomas A. (jun. 2019). „The top 10 things executives should know about software". *Communications of the ACM*, bd. 62, nr. 7, s. 34–40. DOI: 10.1145/3316776.

Mandel, Theo (1997). *The elements of user interface design*. New York: John Wiley & Sons.

McCall, Jim A., Paul K. Richards og Gene F. Walters (nov. 1977). *Factors in Software Quality. Concept and Definitions of Software Quality*. 3 bd. Final technical report RADC-TR-77-369. New York: Rome Air Development Centre.

Northover, Mandy m.fl. (2008). „Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering". *Journal for General Philosophy of Science*, bd. 39, nr. 1, s. 85–113. DOI: 10.1007/s10838-008-9068-7.

Pressman, Roger S. (2001). *Software Engineering. A Practitioner's Approach*. 5. udg. Boston m.m.: McGraw-Hill.

Raymond, Eric (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Beijing etc.: O'Reilly.

Royce, Winston W. (1987). „Managing the Development of Large Software Systems". I: *ICSE '87. Proceedings of the 9th International Conference on Software Engineering*. Washington, DC: IEEE Computer Society Press, s. 328–338. DOI: 10.5555/41765.41801.

Santayana, George (2011). *The Life Of Reason. Introduction and Reason In Common Sense*. Red. af Marianne S. Wokeck og Martin A. Coleman. Critical edition. Works of George Santayana 7. MIT Press.

Seffah, Ahmed og Eduard Metzker (dec. 2004). „The obstacles and myths of usability and software engineering". *Communications of the ACM*, bd. 47, nr. 12, s. 71–76. DOI: 10.1145/1035134.1035136.

Smith, Thomas M. (jul. 1976). „Project Whirlwind: An Unorthodox Development Project". *Technology and Culture*, bd. 17, nr. 3, s. 447–464.

Navneliste

Indeks