Katedralen og basaren

af Eric S. Raymond

Dette er en oversættelse af Eric S. Raymonds The Cathedral and the Bazaar. Den engelske original findes på http://www.catb.org/~esr/writings/cathedral-bazaar/. Oversættelsen er foretaget af Jesper Laisen og Ole Michaelsen. Tak til Lotte Kristiansen, redaktør af DKUUG-Nyt, for mange gode rettelser samt til Peter Makholm og Donald Axel. Yderligere forslag, kommentarer, rettelser og kritik er yderst velkommen.

Jeg dissekerer et succesfuldt open source-projekt, Fetchmail, der med vilje blev kørt som en test af nogle overraskende teorier om udvikling af software, som kom fra Linux' historie. Jeg diskuterer disse teorier ud fra to fundamentalt forskellige udviklingsmetoder, 'katedral-modellen', som det meste af den kommercielle verden bruger, mod 'basar-modellen' fra Linux-verdenen. Jeg viser, at disse modeller stammer fra modstridende forudsætninger om det at finde fejl i software. Jeg fremsætter så et støttende argument fra Linux-erfaringen om, at 'med et tilstrækkeligt antal øjne er alle fejl banale', hvilket antyder produktive analogier med selviske repræsentanters selvkorrigerende systemer, og afslutter med en udforskning af denne indsigts betydning for fremtiden for software.

1. Katedralen og basaren

Linux er undergravende. Hvem ville have troet selv for fem år siden (1991), at et operativsystem i verdensklasse på magisk vis kunne forene flere tusinde udviklere, som spredt over hele planeten hackede på deltid, og som kun var forbundet via Internets fine tråde?

I hvert fald ikke mig. Da Linux kom ind på min radar i begyndelsen af 1993, havde jeg allerede arbejdet med Unix og udvikling af open source i 10 år. I midten af 1980'erne var jeg en af de første bidragere til GNU. Jeg havde frigivet en hel del open source-software på nettet, udviklet eller været med til at udvikle adskillige programmer (Nethack, Emacs VC og GUD modes, Xlife og andre), som stadig bruges flittigt i dag. Jeg troede, jeg vidste hvordan, det skulle gøres.

Linux vendte meget af det, jeg troede, jeg vidste, på hovedet. Jeg havde prædiket Unix-troen på små værktøjer, hurtige prototyper og udviklingsprogrammering i årevis. Men jeg troede også , at der var et bestemt kritisk kompleksitetsniveau, hvor en mere centraliseret, a priori tilgang var på krævet. Jeg troede, at det vigtigste software (operativsystemer og rigtig store værktøjer som Emacs programmeringseditoren) skulle bygges som katedraler, omhyggeligt formet af individuelle troldmænd eller små grupper af magikere, som arbejdede i fuldstændig isolation, og hvor ingen beta blev frigivet før tiden.

Linus Torvalds udviklingsmodel — frigiv tidligt og ofte, deleger alt, hvad du kan, vær åben på grænsen det promiskuøse — kom som en overraskelse. Ingen rolig og ærbødig opbygning af katedraler — snarere synes Linux-samfundet at ligne en stor, vrøvlende basar med forskellige dagsordner og fremgangsmåder (passende symboliseret af Linux-arkiverne, som modtog bidrag fra hvem som helst) ud fra hvilken et sammenhængende og stabilt system tilsyneladende kun kunne fremkomme ved en række mirakler.

Det faktum, at denne basar-model syntes at fungere og fungere godt, kom som et udtrykkeligt chok. Efterhånden som jeg kom ind i tingene, arbejdede jeg hårdt ikke bare på individuelle projekter men også på at prøve at forstå , hvorfor Linux-miljøet ikke alene ikke faldt fra hinanden under forvirringen, men syntes at gå fra sejr til sejr med en hastighed, som byggere af katedraler ikke engang kunne forestille sig.

I midten af 1996 troede jeg, at jeg var begyndt at forstå . Tilfældet havde givet mig en perfekt må de at teste min teori i form af et open source-projekt, som jeg bevidst kunne prøve at køre efter basar-modellen. Det gjorde jeg så — og det var en betydelig succes.

Dette er historien om det projekt. Jeg vil bruge den til at foreslå nogle aforismer om effektiv udvikling af open source. Ikke alt er ting, som jeg først lærte i Linux-verdenen, men vi vil se, hvordan Linux-miljøet understregede dem. Hvis jeg har ret, vil de hjælpe dig med at forstå nøjagtig, hvad det er, der gør Linux-miljøet til et springvand af god software — og måske hjælpe dig til at blive mere produktiv selv.

2. Posten skal ud

Siden 1993 har jeg stået for den tekniske side af en lille gratis ISP kaldet Chester County InterLink (CCIL) in West Chester, Pennsylvania, USA. Jeg var medstifter af CCIL og kodede vores unikke software til flerbruger bulletin-boards — du kan checke med telnet til locke.ccil.org. I dag er der næsten 3.000 brugere på 30 linier. Jobbet gav mig 24 timers adgang til nettet på CCIL's 56K linie — jobbet krævede det faktisk!

Derfor var jeg blevet helt vant til at få min email øjeblikkelig. Jeg syntes, det var irriterende, at jeg jævnligt skulle bruge telnet til locke for at læse min post. Jeg ville have, at min post blev leveret til snark, så jeg fik besked, når den kom, og at jeg kunne håndtere den med mine sædvanlige værktøjer.

Internets almindelige protokol til videresendelse af post, SMTP (Simple Mail Transfer Protocol) ville ikke virke, fordi det virker bedst, når maskiner altid er på nettet, mens min personlige maskine ikke altid er på nettet og ikke har en statisk IP-adresse. Jeg behøvede et program, som kunne række ud over min SLIP-forbindelse og hente min post, så den kunne leveres lokalt. Jeg vidste noget sådant eksisterede, og at de fleste af dem brugte en simpel aplikationsprotokol kaldet POP (Post Office Protocol). POP er nu bredt understøttet af de fleste postklienter, men på det tidspunkt var den ikke bygget ind i det postprogram, som jeg brugte.

Jeg havde brug for en POP3-klient. Så jeg fandt en på nettet. Faktisk fandt jeg tre eller fire. Jeg brugte Pop-perl et stykke tid, men der manglede en oplagt funktion, muligheden for at hacke adressen på den hentede post, så svarene fungerede ordentligt.

Dette var problemet: forestil dig, at en eller anden kaldet 'joe' sendte mig post på locke. Hvis jeg hentede posten til snark og så prøvede at svare på den, ville mit postprogram forsøge at sende svaret til den ikke-eksisterende 'joe' på snark. Det blev hurtigt en pinsel at rette adressen til med '@ccil.org'.

Det var helt klart noget, som computeren burde gøre for mig. Men ingen af de eksisterende POP-klienter vidste hvordan! Og det giver os vores første lektion:

> *1. Ethvert godt stykke software starter med at afhjælpe en udviklers personlige kløe.*

Dette burde måske have været det mest åbenlyse (et gammelt ordsprog siger, at 'Nød lærer nøgen kvinde at spinde'), men alt for ofte slider softwareudviklere for lønnens skyld med programmer, som de hverken behøver eller synes om. Men ikke i Linux-verdenen — hvilket kan forklare, hvorfor gennemsnitskvaliteten af software, som stammer fra Linux-miljøet, er så høj.

Kastede jeg mig så straks vildt over kodning af en helt ny POP3-klient, der kunne konkurrere med de eksisterende? Under ingen omstændigheder! Jeg undersøgte grundigt de POP-muligheder, som jeg havde for hånden, og spurgte mig selv, 'hvilken en er tættest på det, som jeg vil have?'. Fordi

> *2. Gode programmører ved, hvad de skal skrive. Fremragende programmører ved, hvad de skal skrive om (og genbruge).*

Selv om jeg ikke hævder at være en fremragende programmør, prøver jeg at efterligne en. Et vigtigt træk ved de store er konstruktiv dovenskab. De ved, at de kan få et 13-tal ikke for flid men for resultater, og at det næsten altid er nemmere at starte med en god delvis løsning end fra bunden.

Eksempelvis skrev Linus Torvalds faktisk ikke Linux fra bunden. I stedet startede han med at genbruge kode og ideer fra Minix, et lille Unix-lignende OS til PC-kloner. I sidste ende forsvandt al Minix-kode eller blev skrevet fuldstændigt om — men mens det var der, tjente det som kravlegård for spædbarnet, der til sidst blev til Linux.

I samme ånd søgte jeg efter et eksisterende POP-program, som var rimelig godt kodet, til at bruge som udgangspunkt.

Traditionen med deling af kildekode inden for Unix-verdenen har altid været positiv indstillet overfor genbrug af kode (det er derfor GNU-projektet valgte Unix som grund-OS, på trods af alvorlige betænkeligheder ved selve styresystemet. Linux-verdenen har ført denne tradition næsten til dens teknologiske grænse; der er terabytes af frit tilgængelige open source-programmer. Den tid, du bruger på at søge efter andres programmer, der er næsten gode nok, vil oftere give gode resultater i Linux-verdenen end andre steder.

Og det gjorde den for mig. Med dem jeg fandt tidligere, endte jeg efter min anden søgning op med 9 kandidater — Fetchpop, PopTart, Get-mail, Gwpop, Pimp, Pop-perl, Popc, Popmail og Upop. Den første jeg bestemte mig for, var Fetchpop, skrevet af Seung-Hong Oh. Jeg lagde min rutine til omskrivning af brevhovedet ind i programmet og lavede forskellige andre forbedringer, som forfatteren accepterede i sin version 1.9.

Men nogle få uger senere faldt jeg over koden til Popclient af Carl Harris og blev klar over, at jeg havde et problem. Selvom Fetchpop indeholdt nogle gode og originale ideer (eksempelvis dets daemon-metode), så kunne det kun håndtere POP3, og det var kodet af en amatør (Seung-Hong var en kvik men uerfaren programmør, og begge træk kunne ses). Carls kode var bedre, ganske professionel og solid, men hans program manglede adskillige vigtige funktionaliteter fra Fetchpop, som var vanskelige at implementere (også dem, jeg selv havde kodet).

Beholde eller skifte? Hvis jeg skiftede, ville jeg miste koden, som jeg allerede havde lavet. Fordelen ville være et bedre udgangspunkt for videre udvikling.

Et praktisk motiv til at skifte var understøttelse af flere protokoller. POP3 er den mest brugte protokol til post-servere, men det er ikke den eneste. Fetchpop og de andre konkurrenter kunne ikke klare POP2, RPOP eller APOP, og for sjov legede jeg allerede med tanken om at tilføje IMAP (Internet Message Access Protocol, den nyest designede og mest anvendelig protokol).

Men jeg havde desuden en mere teoretisk grund til at synes, at det ville være en god grund til at skifte, noget jeg lærte længe før Linux.

> *3. 'Planlæg at smide en væk; du kommer alligevel til det'. (Fred Brooks, 'The Mythical Man-Month', kapitel 11).*

Eller for at sige det på en anden måde, ofte forstår du alligevel ikke problemet, før efter du har implementeret en løsning. Den anden gang ved du måske nok til at gøre det rigtigt. Så hvis du vil gøre det rigtigt, skal du være parat til at begynde forfra mindst en gang [JB].

Tja (sagde jeg til mig selv), ændringerne i Fetchpop havde været mit første forsøg. Så jeg skiftede.

Da jeg sendte mit første sæt af rettelser til Carl Harris den 25. juni 1996, opdagede jeg, at han i grunden allerede havde mistet interessen for Popclient. Koden var en smule støvet med mindre udestående fejl. Jeg havde mange rettelser, så vi blev hurtig enige om, at det mest logiske for mig var at overtage programmet. Uden jeg egentlig opdagede det, havde projektet eskaleret. Jeg overvejede ikke længere mindre rettelser til en eksisterende POP-klient. Jeg overtog fuldstændig vedligeholdelse af en, og jeg vidste, at de spirende ideer i mit hoved, ville betyde store ændringer. I en softwarekultur, der opfordrer til deling af kode, er det en naturlig måde for et projekt at udvikle sig på . Jeg var et eksempel på :

> *4. Hvis du har den rette indstilling, løber du ind i interessante problemer.*

Men Carl Harris indstilling var endnu mere vigtig. Han forstod, at

> *5. Når du mister interesse for et program, er det din sidste forpligtelse at overgive det til en kompetent efterfølger.*

Uden nogensinde at behøve diskutere det vidste Carl og jeg, at vi havde det fælles mål, at den bedste løsning fandtes derude. Det eneste spørgsmål for os var, om jeg kunne godtgøre, at jeg var kompetent til at overtage opgaven. Da jeg havde gjort det, handlede han med elskværdighed og hast. Jeg håber, at jeg vil handle ligeså , når det bliver min tur.

3. Nødvendigheden af at have brugere

Og så arvede jeg Popclient. Ikke mindre vigtigt arvede jeg Popclients brugerbase. Brugere er en vidunderlig ting at have, og ikke bare fordi de demonstrerer, at du har udfyldt et behov, at du har gjort noget rigtigt. Hvis du dyrker dem rigtigt, kan de blive medudviklere.

En anden styrke ved Unix-traditionen, en som Linux udnytter til det ekstreme, er, at mange brugere også er hackere. Når kildekoden er tilgængelig, kan de være effektive hackere. Det kan være overordentlig brugbart til hurtigere fejlfinding. Med lidt opmuntring vil dine brugere diagnosticere problemer, foreslå rettelser og hjælpe med at forbedre koden endnu hurtigere, end du selv kunne uden hjælp.

> *6. At behandle dine brugere som medudviklere er den mindst besværlige vej til hurtig forbedring af koden og effektiv fejlfinding.*

Det er let at undervurdere betydningen af denne effekt. Faktisk undervurderede stort set alle os i open source-verdenen drastisk, indtil Linus Torvalds beviste det modsatte, nemlig hvor godt effekten skalerede med antal brugere i forhold til systemets kompleksitet.

Faktisk mener jeg, at Linus' vigtigste hack ikke var konstruktionen af selv Linux-kernen, men snarere hans opfindelse af Linux' udviklingsmodel. Da jeg engang gav udtryk for denne holdning i hans tilstedeværelse, smilede han og gentog stille noget, han ofte har sagt: 'Jeg er egentlig en meget doven person, som kan lide at få æren for ting, som andre faktisk har lavet'. Doven som en ræv. Eller, som Robert Henlein så fremragende skrev om en af sine karakterer, for doven til at fejle.

Når du ser tilbage, kan du finde et fortilfælde for Linux' metoder og succes i udviklingen af GNU Emacs Lisp-biblioteket og Lisp kodearkiverne. I modsætning til katedral-byggestilen omkring Emacs' C-kerne og de fleste andre FSF-værktøjer, var udviklingen af Lisp-koden flydende og meget brugerorienteret. Ideer og prototyper blev ofte omskrevet tre eller fire gange, før de nåede en endelig og stabil form. Og et løst forbundet samarbejde, som var muliggjort af Internet, i Linux-stil, var almindeligt.

Faktisk var mit eget mest succesfulde hack før Fetchmail sandsynligvis Emacs VC (version control), et Linux-lignende samarbejde med tre andre mennesker via email, hvor jeg til dato kun har mødt en af dem (Richard Stallman, forfatteren af Emacs og grundlægger af FSF). Det var en brugergrænseflade til SCCS, RCS og senere CVS i Emacs, som tilbyder en 'enkeltklik'-version af kontrolmulighederne. Det udviklede sig fra en lille, grov sccs.el, som en anden havde skrevet. Og udviklingen af VC lykkedes, fordi Emacs' Lisp-kode kunne gå igennem generationerne af frigivelse/test/forbedring meget hurtigt i modsætning til Emacs selv.

4. Frigiv tidligt, frigiv hyppigt

Tidlige og hyppige frigivelser er en vigtig del af Linux' udviklingsmodel. De fleste udviklere (mig selv inklusive) troede, at det var en dårlig politik for større projekter, da tidlige versioner per definition altid er fulde af fejl, og da du ikke ønsker at opbruge dine brugeres tålmodighed.

Denne tro var forstærket af den generelle tilslutning til katedralen som udviklingsstil. Hvis den overvældende målsætning var, at brugerne skulle se så få fejl som muligt, så burde du kun frigive en version hver sjette måned (eller sjældnere) og arbejde som en hest på at finde fejl mellem frigivelserne. Emacs' C-kerne blev udviklet på denne måde. Lisp-biblioteket blev det faktisk ikke — fordi der var aktive Lisp-arkiver uden for FSF's kontrol, hvor du kunne finde nye versioner af kode under udvikling uafhængig af Emacs' frivgivelsesmønster [QR].

Det vigtigste af disse, the Ohio Sate Elisp Archive, foregreb ånden og mange af mulighederne i nutidens store Linux-arkiver. Men få af os tænkte særlig meget over, hvad vi gjorde, eller over, hvad eksistensen af det arkiv antydede af problemer med FSF's udviklingsmodel med katedral-bygning.

Jeg gjorde et seriøst forsøg i 1992 på formelt at fåen stor del af Ohio-koden ind i det officielle Emacs Lisp-bibliotek. Jeg løb ind i politiske problemer og havde ingen synderlig succes. Men et år senere, da Linux blev bredt kendt, blev det klart, at noget anderledes og meget sundere foregik der. Linus' åbne udviklingspolitik var modsætningen til katedral-bygningen. Sunsite- og tsx-11-arkiverne bugnede og adskillige distributioner kom frem. Og alt dette var drevet af en uhørt frekvens af frigivelse af nye Linux-kerner.

Linus behandlede sine brugere som medudviklere på den mest effektive måde:

> *7. Frigiv tidligt. Frigiv hyppigt. Og lyt til dine kunder.*

Linus' fornyelse bestod ikke så meget i, at han gjorde ekspederede hurtige frigivelser, hvor han indkorporerede masser af bruger feedback (noget lignende havde været en del af Unix-traditionen længe), men at han skalerede princippet op til et intensitetsniveau, der matchede kompleksiteten af det, som han udviklede. Dengang (omkring 1991) var det ikke ualmindeligt, at han frigav en kerne hyppigere end en gang om dagen! Det virkede, fordi han dyrkede sin base af medudviklere og brugte Internet til samarbejde meget mere end nogen anden.

Men hvordan virkede det? Og var det noget, jeg kunne gøre efter, eller skyldtes det Linus Torvalds' unikke geni?

Det troede jeg ikke. Jeg medgiver gerne, at Linus er en pokkers god hacker (hvor mange af os kunne konstruere en fuldstændig, produktionsklar kerne til et operativsystem fra bunden?) Men Linux repræsenterede ikke nogen frygtindgydende begrebsmæssigt spring fremad. Linus er ikke (eller i det mindste ikke endnu) et innovativt design-geni, ligesom Richard Stallman eller James Gosling (NeWS og Java) er det. Snarere synes Linus at være et geni til udvikling, med en sjette sans til at undgå fejl, udvikling i forkert retning og sand evne til at finde vejen mellem A og B med den minimale indsats. Hele Linux' opbygning stråler i sandhed af den kvalitet, der afspejler Linus' i grunden konservative og forenklede udviklingsstil.

Så hvis hurtig frigivelse og fuldt brug af Internet som medie ikke var et tilfælde men integrerede dele af Linus' udviklings-genis forståelse for vejen med den minimale indsats, hvad var det så han maksimerede? Hvad var det han hev ud af maskineriet?

Når det udtrykkes sådan, besvarer spørgsmålet sig selv. Linus holdt sine hackere/brugere konstant stimulerede og belønnede — stimulerede af udsigten til en del af æren til at tilfredsstille egoet og med udsigt til konstante (ja daglige) forbedringer af deres eget arbejde.

Linus forsøgte direkte at maksimere antallet af mandetimer, der blev brugt til at finde fejl og til at udvikle, selv om det betød ustabil kode og udbrændte brugere, hvis en alvorlig fejl viste sig at være umedgørlig. Linus opførte sig som om, han troede på sådan noget som dette:

> *8. Med en stor nok base af betatestere og medudviklere, vil næsten ethvert problem blive hurtigt beskrevet og rettelsen være åbenlys for en eller anden.*

Eller mindre formelt, 'Med nok øjne er alle fejl banale'. Jeg kalder det: Linus' lov.

Min originale formulering var, at ethvert problem 'vil være gennemskueligt for en eller anden'. Linus indvender, at den person, som først forstår og retter problemet, ikke nødvendigvis — eller ikke engang som regel — er den, som først beskrevet det. 'En eller anden finder problemet', siger han, 'og en helt anden forstår det. Og jeg vil gerne citeres for at sige, at det at finde problemet er den største udfordring'. Men pointen er, at det har en tendens til at ske hurtigt.

Jeg tror, at her er den fundamentale forskel på katedral-stilen og basar-stilen. Når en katedralbygger programmerer, er fejl og udvikling vanskelige, lumske, grundlæggende fænomener. Det tager måneder med grundig overvejelse for de pligttro at tro på, at de har fundet alle fejlene. Derfor er der lange intervaller mellem frigivelser, og den uundgåelige skuffelse, når den længeventede frigivelse ikke er perfekt.

På den anden side er holdningen i basaren, at du går ud fra, at fejl i al almindelighed er banale — eller i det mindste bliver de hurtig banale, når de eksponeres til tusinder af villige medudviklere, der hamrer løs på enhver ny frigivelse. Følgelig frigiver du hyppigere for at få flere rettelser, og som en god sidegevinst er der mindre at miste, hvis et lejlighedsvis makværk slipper ud af døren.

Og det er det. Det er nok. Hvis 'Linus' lov' er falsk, så burde ethvert system så komplekst som Linux-kernen, der er blevet hacket af så mange, på et eller andet tidspunkt være brudt sammen under vægten af uforudset dårligt sammenspil og 'grundlæggende' fejl, der ikke er fundet. På den anden side — hvis den er sand, er det tilstrækkeligt til at forklare Linux' relative mangel på fejl og dens fortsatte høje oppetid, der strækker sig over måneder og år.

Og måske burde det ikke have været sådan en overraskelse. Sociologer opdagede for år siden, at den gennemsnitlige mening hos en gruppe af lige dygtige (eller lige ignorante) iagttagere, er en del mere pålidelig end en enkelt tilfældigt udvalgt af iagttagerne. De kaldte det 'delfi-effekten'. Det virker som om, at det, Linus har vist, er, at det også gælder om det at finde fejl i et operativsystem — at delfi-effekten kan tæmme udviklingens kompleksitet selv ved et kompleksitetsniveau som med en operativsystem-kerne.

En særlig egenskab ved Linux-situationen, som klart hjælper sammen med delfi-effekten, er det faktum, at bidragsyderne til et hvilket som helst projekt er selv-valgte. En tidlig kritiker gjorde opmærksom på, at bidrag ikke modtages fra en tilfældig stikprøve, men fra folk, som er tilstrækkeligt interesseret til at bruge softwaren, lære hvordan den virker, forsøge at finde løsninger på de problemer, som de støder på, og faktisk producere en tilsyneladende fornuftig løsning. Det er overordentlig sandsynligt, at enhver, der kommer igennem disse filtre, kan bidrage med noget brugbart.

Jeg står i gæld til Jeff Dutky (dutky@wam.umd.edu), som gjorde opmærksom på, at Linus' lov kan omformuleres til 'Fejlfinding kan paralleliseres'. Jeff observerer, at selvom fejlfinding kræver, at fejlfinderne kommunikerer med en eller anden koordinerende udvikler, så kræver det ikke nogen betydelig koordinering mellem fejlfinderne. Det falder ikke som bytte for den samme kvadratiske kompleksitet og de ledelsesudgifter, som gør det problematisk at øge antallet af udviklere.

I praksis synes det teoretiske tab af effektivitet, som skyldes duplikeringen af fejlfindernes arbejde, aldrig at være et problem i Linux-verdenen. En effekt af 'politikken om at frigive tidligt og hyppigt' er at minimere sådan duplikering ved at hurtigt sprede rettelser, der stammer fra feedback [JH].

Brooks (forfatteren af The Mythical Man-Month) kom endda med en henkastet bemærkning om Jeff: 'Den totale omkostning ved at vedligeholde et bredt anvendt program er typisk 40 procent eller mere af omkostningen ved at udvikle det. Det er overraskende, at de omkostninger er påvirket af antallet af brugere. Flere brugere finder flere fejl' (min fremhævning).

Flere brugere finder flere fejl, fordi flere brugere betyder flere måder at stresse programmet på. Den effekt er forstærket, når brugerne er medudviklere. Enhver af dem griber opgaven med at beskrive fejl an med et lidt anderledes begrebssæt og og andre analytiske værktøjer, en anden vinkel på problemet. Delfi-effekten synes at virke præcis pågrund af denne variation. Specifikt med hensyn til at finde fejl har variationen også tendens til at reducere duplikeringen af indsatsen.

At have flere betatestere reducerer således ikke nødvendigvis kompleksiteten af forhåndenværende grundlæggende fejl, men det øger sandsynligheden for, at en eller andens arbejdsmetode bliver stillet overfor problemet på en sådan måde, at fejlen bliver banal for denne person.

Linus helgarderer også sit væddemål. Hvis der er alvorlige fejl, er en version af Linux-kernen nummereret på en sådan måde, at en potentiel bruger kan vælge enten at køre den sidste version, som er 'stabil', eller afprøve det nyeste nye og risikere fejl for at få nye muligheder. Denne taktik er endnu ikke formelt efterlignet af de fleste Linux-hackere, men måske skulle den være det; faktum er, at det at begge valg er mulige, gør begge valg mere attraktive [HBS].

5. Hvornår er en rose ikke en rose?

Da jeg havde studeret Linus' adfærd og formuleret en teori om, hvorfor den var succesfuld, tog jeg en bevidst beslutning om at afprøve hans teori på mit nye projekt (som jeg indrømmer er meget mindre komplekst og ambitiøst).

Men det første jeg gjorde var at reorganisere og forenkle Popclient en hel del. Carl Harris' implementeringer var meget solide, men bar præg af en unødvendig kompleksitet, som er kendetegnende for mange C-programmører. Han behandlede koden som om, den var det centrale i sig selv, og datastrukturerne som understøttelse for koden. Resultatet var, at koden var smuk, men designet

af datastrukturerne var tilfældigt og temmelig grimt (i det mindste efter den gamle LISP-hackers høje standarder).

Udover at forbedre koden og datastrukturerne havde jeg dog en anden grund til at omskrive koden. Det var at udvikle det til noget, som jeg fuldstændigt forstod. Det er ikke særligt morsomt at rette fejl i et program, som du ikke forstår fuldt ud.

I den første måned, cirka, fulgte jeg simpelthen betingelserne i Carls grundlæggende design. Den første alvorlige ændring, jeg lavede, var at tilføje understøttelse af IMAP. Jeg gjorde dette ved at reorganisere indretningen af protokollerne til en standard-driver og tre metode-tabeller (til POP2, POP3 og IMAP). Dette og de forrige ændringer illustrerer et generelt princip, som programmører gør klogt i at huske, specielt med hensyn til sprog som C, der ikke naturligt arbejder med dynamisk skrivning:

> *9. Smarte datastrukturer og dum kode virker meget bedre end det modsatte.*

Brooks, kapitel 9: 'Vis mig din [kode] skjul dine [data strukturer], og jeg vil fortsat være mystificeret. Vis mig dine [data strukturer], og jeg vil sandsynligvis ikke behøve din [kode]; den vil være indlysende'.

Faktisk sagde han 'oversigts-diagrammer' og 'tabeller'. Men hvis man tager højde for tredive års skift i terminologi/kultur, er det næsten samme pointe.

På dette tidspunkt (tidligt i september 1996, omkring seks uger efter år nul) begyndte jeg at tænke på, at et navneskift ville være passende —- når alt kommer til alt, var det jo ikke mere bare en POP klient. Men jeg tøvede, fordi der endnu ikke var noget virkelig nyt i designet. Min version af Popclient manglede stadig at udvikle en selvstændig identitet.

Det ændredes radikalt, da Fetchmail lærte at videresende post til SMTP-porten. Jeg kommer til det om et øjeblik. Men først: jeg sagde ovenfor, at jeg bestemte mig til at bruge dette projekt til at teste min teori om, hvad Linus Torvalds havde gjort rigtigt. Hvordan (kan du sagtens spørge) gjorde jeg det? På disse måder:

(a) Jeg frigav tidligt og hyppigt (næsten aldrig mindre end hver tiende dag; en gang om dagen i perioder med intens udvikling).

(b) Jeg udvidede min beta-liste ved at tilføje enhver, som kontakte mig angående Fetchmail.

(c) Jeg sendte sludrende meldinger til beta-listen hver gang, jeg frigav, hvor jeg opfordrede folk til at deltage.

(d) Og jeg lyttede til mine betatestere, spurgte dem om beslutninger om design og roste dem hver gang, de sendte rettelser og feedback ind.

Gevinsten fra disse simple forholdsregler var øjeblikkelig. Fra projektets begyndelse fik jeg fejlrapporter af en kvalitet, som de fleste udviklere ville dræbe for, ofte vedhæftet rettelser. Jeg fik opmærksom kritik, jeg fik fanpost, jeg fik intelligente forslag til faciliteter. Hvilket fører til:

> *10. Hvis du behandler dine betatestere, som om de er din mest værdifulde ressource, vil de reagere ved at blive din mest værdifulde ressource.*

Et interessant mål for Fetchmails succes er bare størrelsen af beta-listen, vennerne af Fetchmail. I skrivende stund har den 249 medlemmer, og der kommer to eller tre til hver uge.

Faktisk er der en interessant grund til, at listen er ved at miste medlemmer fra højdepunktet tæt på 300, da jeg reviderer i maj 1997. Adskillige folk har bedt mig slette dem, fordi Fetchmail virker så godt for dem, at de ikke længere har behov for at være på listen! Måske er det den normale livscyklus for et modent projekt af basar-stilen.

6. Popclient bliver til Fetchmail

Det virkelige vendepunkt i projektet var, da Harry Hochheiser sendte mig sin skrabede kode til at videresende post til klientmaskinens SMTP-port. Jeg blev straks klar over, at en pålidelig implementering af denne facilitet ville gøre alle de andre leveringsmåder praktisk talt forældede.

I mange uge havde jeg kun rykket gradvist frem med Fetchmail, selv om jeg syntes, at designet af brugergrænsefladen var brugbart men groft — uelegant og med for mange ubetydelige muligheder

stikkende frem alle vegne. Muligheden for at smide den hentede post ned i en postkasse-fil eller til standard uddata generede mig især, men jeg kunne ikke finde ud af hvorfor.

(Hvis du ikke er interesseret i den tekniske side af hvordan elektronisk post på Internet fungerer, kan du uden problemer springe de næste to afsnit over)

Det jeg så, når jeg tænkte over videresendelse med SMTP var, at Popclient havde forsøgt at gøre for mange ting på en gang. Den var blevet designet til både at være en posttransportagent (MTA) og en lokal leverings-agent (MDA). Med videresendelse med SMTP kunne den komme væk fra MDA-området og blive en ren MTA, der afleverer posten til andre programmer, som så håndterer den lokale levering — ligesom Sendmail gør.

Hvorfor besvære sig med kompleksiteten i konfigurering af MDA eller med at sætte en lås-og-tilføj på en postkasse, når port 25 til at begynde med næsten med sikkerhed findes på enhver platform med understøttelse af TCP/IP? Især når det betyder, at hentet post med sikkerhed vil se ud som SMTP-post fra en normal afsender, hvilket i virkeligheden var det, som jeg alligevel ville.

(Tilbage til et højere niveau...)

Selvom du ikke fulgte den foregående tekniske jargon, så er der adskillige lektioner her. For det første var denne ide om videresendelse med SMTP den største enkelte gevinst, jeg fik ved bevidst at efterligne Linus' metoder. En bruger gav mig denne fremragende ide — det eneste jeg behøvede at gøre var at forståimplikationerne.

> *11. Det næstbedste efter at få gode ideer er at genkende gode ideer fra dine brugere. Nogle gange er det sidste bedre.*

Interessant nok finder du hurtigt ud af, at hvis du er hudløst ærlig med hensyn til, hvor meget du skylder andre mennesker, så vil verdenen som helhed behandle dig som om, du selv har opfundet hver en detalje, og som om du bare er passende ydmyg med hensyn til dit medfødte geni. Vi kan alle se, hvor godt det virkede for Linus!

(Da jeg holdt dette foredrag på Perl-konference i august 1997, sad Larry Wall på forreste række. Da jeg kom til den sidste af ovenstående linier, råbte han i religiøs vækkelsesstil, 'Sig det, sig det, broder'! Hele publikummet grinede, fordi de vidste, at det også havde virket for opfinderen af Perl)

Efter i nogle få uger at have kørt projektet i den samme ånd, begyndte jeg at få lignende ros ikke bare fra mine brugere men fra andre folk, som havde hørt rygtet. Jeg har gemt nogle af de emails; jeg vil tage dem frem igen, hvis jeg nogen sinde begynder at tvivle på, om mit liv har været noget værd :-)

Men der er yderligere to fundamentale, upolitiske lektioner her, som gælder for alle former for design.

> *12. Ofte stammer de mest slående og nyskabende løsninger fra en erkendelse af, at din forståelse af problemet var forkert.*

Jeg havde forsøgt at løse det forkerte problem ved at forsætte med at udvikle Popclient som en kombineret MTA/MDA med alle slags smarte lokale leveringsmåder. Fetchmails design trængte til at blive genovervejet fra grunden som en rendyrket MTA, som en del af den normale postvej på Internet med SMTP.

Når du rammer muren under udvikling — når du har problemer med at komme gennem næste rettelse — er det ofte på tide at spørge, ikke om du har det rigtige svar, men om du stiller det rigtig spørgsmål. Måske trænger spørgsmålet til at blive omformuleret.

Ok, jeg havde omformuleret mit problem. Tydeligvis var det rigtige at gøre, (1) at hacke understøttelse af videresendelse med SMTP ind i standard-driveren, (2) at gøre det til standardmåden og (3) i sidste ende smide alle de andre leveringsmåder væk, især mulighederne for levering som fil og levering som standard uddata.

Jeg tøvede nogen tid med hensyn til skridt 3, da jeg var bange for at gøre gamle brugere af Popclient,som var afhængige af vekslende mekanismer til postleveringer, vrede. I teorien kunne de med det samme skifte til .forward-filer eller deres tilsvarende non-sendmail for at få samme effekt

(for andre postservere end Sendmail) for at have samme mulighed. I praksis kunne overgangen blive noget skidt.

Men da jeg gjorde det, viste fordelene sig at være enorme. Den tungeste del af koden til driveren forsvandt. Konfigurationen blev voldsomt enklere — ikke flere problemer med MDA-systemet og brugerens postkasse, ikke flere bekymringer om hvorvidt det underliggende OS understøttede fillåsning.

Desuden forsvandt den eneste måde, man kunne miste post. Hvis du valgte aflevering som fil, og din disk løb fuld, mistede du posten. Det kan ikke ske ved videresendelse med SMTP, da postserveren ikke returnerer et OK med mindre beskeden kan blive leveret eller i det mindste sendt til en midlertidig køfor senere levering.

Derudover blev ydelsen forbedret (dog ikke sådan, at du ville bemærke det i første omgang). En anden ikke uvæsentlig fordel af denne ændring var, at den manuelle side blev enklere.

Senere måtte jeg genindføre levering via brugerspecificeret MDA for at tillade håndtering af nogle obskure situationer med dynamisk SLIP. Men jeg fandt en meget simplere måde at gøre det på.

Moralen? Tøv ikke med at kaste overudviklede funktioner bort, når du kan gøre det uden at miste effektivitet. Antoine de Saint-Expry (som var pilot og flydesigner, når han ikke forfattede klassiske børnebøger) sagde:

> *13. 'Perfektion (med hensyn til design) opnås ikke, når der ikke er mere at tilføje, men snarere når der ikke er mere at tage bort'.*

Når din kode er ved at blive bedre og enklere, så *ved* du, at det er rigtigt. Og i processen fik Fetchmails design sin egen identitet, som var forskellig fra forgængeren Popclient.

Det var tid til en navneforandring. Det nye design lignede Sendmail meget mere end den gamle Popclient; begge er MTA'er, men hvor Sendmail sender før levering, så henter den nye Popclient før levering. Så to måneder ude af starthullerne omdøbte jeg den til Fetchmail.

Der er en mere almindelig lektie af lære fra denne historie om hvordan, SMTP-levering blev indarbejdet i fetchmail. Det er ikke kun fejlfinding, der kan paralleliseres; det kan udvikling og (i en måske overraskende grad) udforskning af designrummet også. Når din udviklingsmetode hurtigt gentages, kan udvikling og forøgelse blive særlige tilfælde af fejlfinding — at rette 'udeladelsesfejl i softwarens originale anvendelighed og koncept.

Selv på et højere designniveau, kan det være meget værdifuldt, at masser af tænkende medudvikleres tilfældigt gennemgår designrummet omkring dit produkt. Se til eksempel den måde en vandpyt finder et afløb, eller endnu bedre hvordan myrer finder mad: i grunden udforskning gennem spredning efterfuldt af udforskning formidlet gennem en skalerbar kommunikationsmekanisme. Det virker rigtig godt; som med Harry Hochheiser og jeg er det muligt, at en af dine udforskere finder et stort bytte i nærheden, som du var bare lidt for snævert fokuseret til at se.

7. Fetchmail vokser op

Der var jeg med et godt og innovativt design, kode, som jeg vidste virkede godt, fordi jeg brugte den hver dag, og en spirende beta-liste. Det gik gradvist op for mig, at jeg ikke længere var involveret i et trivielt, personligt hack, som måske ville være brugbart for nogle få andre. Jeg havde fat i et program, som enhver hacker med en Unix-boks og en SLIP/PPP postforbindelse virkelig havde brug for.

Med faciliteten til SMTP-videresendelse rykkede det så langt foran konkurrenterne, at det blev en potentiel 'kategoridræber', et af de klassiske programmer, der udfylder sin niche så fuldstændigt, at alternativerne ikke bare kastes væk men næsten glemmes fuldstændigt.

Jeg tror ikke, at du kan sigte mod eller planlægge et resultat som dette. Du skal næsten trækkes ind i det af ideer om design, der er såkraftfulde, at resultaterne bagefter virker uundgåelige, naturlige, næsten forudbestemte. Den eneste måde at sigte efter sådanne ideer er ved at have masser af ideer — eller ved at have evnen til at bringe gode ideer videre, end deres ophavsmænd troede, de kunne komme.

Andrew Tanenbaum fik den originale id at bygge en simpel enkeltstående Unix til en 386-platformen til brug som undervisningsredskab (han kaldte det Mimix). Linus Torvalds drev formentlig Minix-projektet længere, end Andrew troede det kunne komme — og det voksede til noget vidunderligt. På samme måde (bare i mindre målestok) tog jeg nogle af Carl Harris' og Harry Hochheisers ideer og pressede dem videre. Ingen af os var 'originale' i den romantiske forstand, som folk synes er genialt. Men på den anden side er det meste videnskab, ingeniørarbejde og softwareudvikling ikke udført af originale genier, uanset hvad hacker-mytologien siger.

Resultaterne var alligevel temmelig voldsomme — faktisk var det den slags succes, som alle hackere lever for! Og de betød, at jeg skulle sigte endnu højere. For at gøre Fetchmail så godt, som jeg nu så det kunne blive, måtte jeg nu skrive ikke bare efter mine egne behov men også inkludere og understøtte faciliteter, som var nødvendige for andre uden for min kreds. Og at gøre det mens programmet fortsat forblev enkelt og robust.

Den første og overvældende vigtige facilitet, jeg skrev, efter jeg havde indset det, var understøttelse af levering til mange brugere — muligheden for at hente post fra postkasser, der opsamlede al post for en gruppe af brugere og så sende hver enkelt stykke post til de enkelte modtagere.

Jeg besluttede at tilføje understøttelse af levering til mange brugere, delvist fordi nogle brugere råbte op om det, men mest fordi jeg mente, at det ved at tvinge mig til at håndtere adressering fuldstændig generelt ville fremprovokere fejl i koden, som var skrevet med henblik på levering til en enkelt bruger. Og det gjorde det. At fåRFC 822-fortolkeren til at virke tog mig bemærkelsesværdig lang tid, ikke fordi nogen enkelt del af det var svært, men fordi det havde betydning for en bunke af gensidigt afhængige og forvirrende detaljer. Men levering til mange brugere viste sig også at være en fremragende design-løsning. Det vidste jeg fordi:

> *14. Ethvert værktøj bør være anvendeligt på den forventede måde, men et i sandhed enestående værktøj kan bruges til ting, som du aldrig havde forventet.*

Den uventede brug af Fetchmails levering til mange brugere er styring af postfordelingslister og alias-ekspansion udført på klientsiden af SLIP/PPP-forbindelsen. Det betyder, at man fra en personlig maskine koblet op gennem en ISP, kan administrere en postfordelingsliste uden konstant adgang til alias-filerne hos ISP'en.

En anden vigtig ændring, som mine betatestere forlangte, var understøttelsen af 8-bit MIME. Det var temmelig nemt at gøre, da jeg var omhyggelig med at holde koden i ren 8-bit. Ikke fordi jeg havde forventet krav om den facilitet, men i overensstemmelse med en anden regel:

> *15. Når du skriver gateway software, skal du sørge for at forstyrre datastrømmen så lidt som muligt — og* **aldrig** *smide informationer væk, med mindre modtageren tvinger dig til det!*

Hvis jeg ikke havde fuldt denne regel, ville understøttelsen af 8-bit MIME have været svær og fejlbehæftet. Faktisk behøvede jeg kun at læse RFC 1652 og tilføje en triviel mængde logik til generering af brevhoveder.

Nogle europæiske brugere plagede mig om at tilføje en mulighed for at begrænse antallet af beskeder, som blev hentet per session (så de kunne kontrollere omkostningerne ved deres dyre telefonforbindelser). I lang tid strittede jeg imod dette, og jeg er fortsat ikke helt glad for det. Men når man skriver til hele verdenen, skal man lytte til sine kunder — det ændrer sig ikke bare fordi, de ikke betaler med penge.

8. Nogle yderligere lektioner fra Fetchmail

Før vi vender tilbage til generelle emner vedrørende udvikling af software, skal vi lige overveje yderligere et par specifikke lektioner fra Fetchmail-projektet. Ikke-tekniske mindede læsere kan roligt springe dette afsnit over.

Syntaksen i rc-filen indeholder valgfri 'støj'-nøgleord, som ignoreres fuldstændigt af fortolkeren. Den engelsklignende syntaks, som tillades, er betragteligt mere læsevenlig end den traditionelle stringente kombination af nøgleord og tilhørende værdier, som man får ved at fjerne dem helt.

De begyndte som et sent natte-eksperiment, da jeg lagde mærke til, hvor meget deklarationen af rc-filen begyndte at ligne et imperativt minisprog (det er derfor, jeg ændrede det oprindelige Popclient nøgleord 'server' til 'poll').

Det forekom mig, at hvis jeg gjorde det imperative minisprog mere som engelsk, ville det måske gøre det lettere at bruge. Selvom jeg er en overbevist tilhænger af 'gør det til et sprog'-designskolen, som Emacs og HTML er eksempler på, er jeg normalt ikke fan af engelsklignende syntakser.

Traditionelt har programmører haft tendens til at favorisere kontrolsyntakser, som er meget præcise og kompakte, og som ikke har nogen redundans overhovedet. Det er en kulturel arv fra dengang, hvor computerressourcer var dyre, så fortolkerniveauer skulle være såbillige og simple som mulige. Engelsk med omkring 50% redundans, lignede en meget uegnet model dengang.

Det er ikke årsagen til, at jeg normalt undgår engelsklignende syntaks; jeg nævner det kun her for at modsige det. Med billig cpu-tid og datakraft bør enkelthed ikke være et mål i sig selv. Nu om dage er det mere vigtigt, at et sprog er bekvemt for mennesker, end at det er besparende for computeren.

Der er derimod gode grunde til at være varsom. En grund er omkostningen ved kompleksiteten af fortolkerniveauet — du bør ikke håndhæve det til et niveau, hvor det i sig selv bliver en kilde til fejl og brugerforvirring. En anden grund er, at når du prøver at gøre et sprogs syntaks mere engelsklignende, kræver det ofte, at det engelske, som bruges, bliver så fordrejet, at den overfladiske lighed med et naturligt sprog bliver lige så forvirrende, som en traditionel syntaks ville have været (du ser det ofte i de såkaldte 'fjerde generations'-sprog og kommercielle database-sprog).

Syntaksen, der styrede Fetchmail, synes at undgå disse problemer, da sprogområdet er ekstremt begrænset. Det er ikke engang i nærheden af et generelt anvendeligt sprog; indholdet er ganske enkelt ikke særligt kompliceret, så der er ikke særlig stor chance for forvirring ved mentalt at bevæge sig mellem et mindre undersprog til engelsk og det faktiske kontrolsprog. Jeg tror, at der er en generel lektie her:

> *16. Når dit sprog ikke engang er i nærheden af turingstandard, kan syntaktisk sukker være din ven.*

En anden lektie drejer sig om sikkerhed gennem uigennemskuelighed. Nogle Fetchmail-brugere bad mig om at ændre softwaren, så adgangskoder blev opbevaret krypterede i rc-filen, såsnushaner ikke ville være i stand til at se dem ved en tilfældighed.

Det gjorde jeg ikke, da det ikke rigtigt giver nogen yderlig sikkerhed. Hvem som helst, som har tilegnet sig tilladelse til at læse din rc-fil, vil alligevel være i stand til at starte Fetchmail som dig - og hvis det er din adgangskode, de er ude efter, vil de være i stand til at få den nødvendige dekoder ud af selve Fetchmail-koden for at fåadgangskoden.

Det eneste en kryptering af adgangskoder til .fetchmailrc ville have gjort, var at give en falsk følelse af sikkerhed for folk, som ikke tænker sig om. Den generelle regel her er:

> *17. Et sikkerhedssystem er kun så sikkert som dets hemmelighed. Pas påpseudohemmeligheder.*

9. Nødvendige forudsætninger for basar-metoden

Tidlige anmeldere og testlæsere af dette skrift stillede vedholdende spørgsmål om forudsætningerne for udvikling efter basar-metoden, inklusive både projektlederens kvalifikationer og kodens tilstand, når du offentliggører projektet og begynder at opbygge et fællesskab af meduddviklere.

Det er temmelig åbenlyst, at du ikke kan kode fra starten efter basar-metoden [IN]. Du kan teste, fejlsøge og forbedre med basar-metoden, men det vil være meget svært at starte et nyt projekt med basar-metoden. Linus prøvede det ikke. Det gjorde jeg heller ikke. Dit spirende udviklerfællesskab har brug for noget at lege med, der virker og kan testes.

Når du begynder at opbygge et fællesskab, har du behov for at kunne præsentere et sandsynligt løfte. Dit program behøver ikke at virke særlig godt. Det kan være ufærdigt, fejlbehæftet,

ufuldstændigt og dårligt dokumenteret. Det skal dog være i stand til at a) kunne køre, og b) og det skal dog kunne overbevise potentielle medudviklere om, at det kan udvikles til noget rigtigt godt inden for en overskuelig fremtid.

Linux og Fetchmail blev begge offentliggjort med et stærkt og tiltalende grundlæggende design. Mange, der har tænkt over basar-modellen, som jeg har præsenteret den, betragter ganske rigtigt dette som centralt, og har så draget den konklusion, at en veludviklet sans for intuitivt design og begavelse hos projektlederen er uundværlig.

Men Linus fik sit design fra Unix. Jeg fik mit design fra den fædrene Popclient (selv om det senere skulle ændre sig temmelig meget, proportionalt meget mere end Linux har gjort). Er det virkelig nødvendigt, at lederen/koordinatoren af en indsats efter basar-metoden har et exceptionel talent for design, eller kan han nøjes med at løfte andres talent?

Jeg mener ikke, at det er centralt, at koordinatoren er i stand til at skabe design af exceptionel karakter, men det er absolut centralt, at koordinatoren er i stand til genkende andres gode design-ideer.

Både Linux- og Fetchmail-projekterne viser tegn på det. Linus, selv om han (som tidligere gennemgået) ikke er en spektakulær, original designer, har vist et stærkt håndelag for at godt design og for at integrere det i Linux-kernen. Og jeg har allerede beskrevet, hvordan den allervigstigste design-ide i Fetchmail (videresendelse med SMTP) kom fra en anden.

Tidlige læsere af dette skrift komplimenterer mig ved at foreslå, at jeg er tilbøjelig til at undervurdere originalt design i basar-projekter, fordi jeg selv har masser af det og derfor tager det for givet. Der kan være noget sandhed i det; design (i modsætning til kodning og fejlsøgning) er bestemt min stærkeste egenskab.

Men problemet med at være smart og original i forbindelse med design af software er, at det bliver en vane — det bliver en vane at gøre ting smarte og komplicerede, hvor du i stedet skulle lade dem være robuste og enkle. Jeg har ødelagt projekter, fordi jeg gjorde den fejl, men det lykkedes mig at undgå det med Fetchmail.

Så jeg tror, at projektet Fetchmail blev en succes, delvist fordi jeg beherskede min tendens til at være smart; det er et argument (i det mindste) mod, at originalt design er essentielt for et succesfuldt basar-projekt. Tag Linux for eksempel. Forestil dig, at Linus Torvalds have forsøgt at gennemføre fundamentale nyskabelser indenfor design af operativsystemer under udviklingen; virker det så overhovedet sandsynligt, at den resulterende kerne ville være blevet så stabil og succesfuld, som den vi har?

Selvfølgelig er et grundniveau af design- og programmeringsevner som minimum nødvendige. Jeg regner med, at næsten enhver, der tænker på at starte en basar-indsats, allerede er over det minimum. Open source-samfundets indre marked for omdømme lægger et subtilt pres påfolk om ikke at starte en udviklingsindsats, som de ikke er kompetente til at gennemføre. Hidtil ser det ud til at have virket temmelig godt.

Der er en anden egenskab, som ikke normalt er forbundet med udvikling af software, som jeg mener er ligeså vigtig som smart design — og det er måske mere vigtigt. En koordinator eller leder af et basar-projekt må have gode sociale og kommunikationsevner.

Dette burde være indlysende. For at opbygge et udviklingssamfund, har du brug for at tiltrække folk, gøre dem interesseret i, hvad du laver, og gøre dem glade for den indsats, de yder. Teknisk brillans bringer dig langt mod opnåelsen af dette, men det er langt fra det hele. Den personlighed, som du udstråler, betyder også noget.

Det er ikke et tilfælde, at Linus er en flink fyr, som får folk til at synes om sig og få lyst til at hjælpe ham. Det er ikke et tilfælde, at jeg er energisk og udadvendt, holder af at underholde, og at jeg har noget af den indlevelse og de instinkter, som en stand up-komiker har. For at få en basar-model til at virke, hjælper det enormt, hvis du i det mindste har en smule evne til at charmere folk.

10. Open Source-software i en social sammenhæng

Det er så sandt som det er sagt: at de bedste hacks begynder som personlige løsninger til forfatterens dagligdags problemer og spreder sig, fordi det viser sig, at problemerne er typiske for en

18. For at løse et interessant problem skal du starte med at finde et problem, der interesserer dig.

stor klasse af brugere. Det fører os tilbage til emnet for regel 1 måske skrevet mere anvendeligt:

Sådan var det for Carl Harris og det fædrende Popclient, og sådan var det med mig og Fetchmail. Men det har været kendt i lang tid. Det interessante, som historierne om Linux og Fetchmail synes at kræve at vi fokuserer på, er det næste stadie — udviklingen af software i store og aktive miljøer af brugere og medudviklere.

I 'The Mythical Man-Month' bemærkede Fred Brooks, at programmørers tid ikke er ombyttelig; når der bliver tilført udviklere til et forsinket software-projekt, bliver det mere forsinket. Han mener, at kompleksiteten og omkostningerne ved et projekt vokser med kvadratet på antallet af udviklere, men at det udførte arbejde kun vokser lineært. Denne påstand er siden blevet kaldt for 'Brooks lov' og anses i vide kredse for at være en selvindlysende sandhed. Men hvis Brooks lov var hele sandheden, ville Linux have været en umulighed.

Gerald Weinbergs klassiske 'The Psychology Of Computer Programming' giver set i bagklogskabens lys en væsentlig korrektion af Brooks. I hans diskussion af 'jegløs programmering' bemærker Weinberg, at i forretninger, hvor programmører ikke er territoriale med hensyn til deres kode, og hvor de opfordrer andre til at lede efter fejl og potentielle forbedringer i den, sker fremskridt dramatisk hurtigere end ellers.

Weinbergs valg af terminologi har måske forhindret hans analyse i at få den accept, som den fortjente — man må smile af ideen om Internet-hackere som 'jegløse'. Men jeg mener hans argument virker mere tiltalende end nogen sinde.

Unix' historie burde have forberedt os på, hvad vi er ved at lære af Linux (og hvad jeg har bekræftet eksperimentelt i mindre målestok ved bevidst at kopiere Linus' metoder [EGCS]). Det vil sige, at selv om kodning basalt set er en ensom opgave, så kommer de virkelig store hacks ved at anvende opmærksomheden og hjernekraften fra hele samfund. Udvikleren, som kun bruger sin egen hjerne i et lukket projekt, vil falde bagud i forhold til udvikleren, som ved, hvordan han skaber en åben, evolutionær kontekst, hvor fejlfinding og forbedringer udføres af hundreder af mennesker.

Men den traditionelle Unix-verden var af mange årsager forhindret i at anvende denne fremgangsmåde optimalt. En årsag var lovmæssige begrænsninger som følge af licenser, forretningshemmeligheder og kommercielle interesser. En anden (i bagklogskabens lys) var, at Internet ikke var særlig godt endnu.

Før Internet blev billigt, var der geografisk tætte miljøer, hvor kulturen opfordrede til Weinbergs 'jegløse' programmering, og en udvikler kunne let tiltrække masser af dygtige kibitzere og medudviklere. Bell Labs, MIT AI Lab, UC Berkely — alle blev hjemsted for innovationer, der er legendariske og stadigt potente.

Linux var det første projekt, som var et bevidst og succesfuldt forsøg på at bruge hele verdenen som talentmængde. Jeg tror ikke, det er et tilfælde, at Linux' drægtighedsperiode faldt sammen med fødslen af World Wide Web, og at Linux overstod sin barndom i den samme periode i 1993-94, hvor ISP-industrien tog fart og eksplosionen i den almene interesse i Internet. Linus var den første, som lærte at spille efter de nye regler, som det vidt udbredte Internet muliggjorde.

Selv om billigt Internet var en nødvendig betingelse for at Linux-modellen kunne udvikles, tror jeg ikke, at det alene i sig selv var en tilstrækkelig betingelse. En anden væsentlig faktor var udviklingen af en lederstil og af samarbejdsmåder, som tillod udviklere at tiltrække medudviklere og få maksimalt udbytte af mediet.

Men hvad er den lederstil, og hvordan er disse samarbejdsmåder? De kan ikke være baserede på magtforhold — og selv om de kunne, ville ledelse ved tvang ikke producere de resultater, som vi ser. Weinberg citerer med stor effekt fra den russiske 1900-tals anarkist Pyotr Alexeyvich Kropotkins selvbiografi 'Memoirs of a Revolutionist' om det emne:

'Efter at være vokset op i en familie med livegne begyndte jeg et aktivt liv med, som alle andre andre unge mænd i min tid, en stor portion tiltro til nødvendigheden af befaling, ordre, udskældning, straf og lignende. Men da jeg på et tidligt tidspunkt skulle lede seriøse virksomheder og gøre forretning med (frie) mennesker, og hvor enhver fejl med det samme ville have alvorlige konsekvenser, begyndte jeg at værdsætte forskellen mellem at handle efter principper som befaling

og disciplin og at handle efter princippet om almindelig forståelse. De første virker fortrinligt i en militærparade, men er intet værd i det virkelige liv, og målet kan kun nås gennem en stor indsats fra mange konvergerende viljer'.

Denne 'store indsats af mange konvergerende viljer' er præcis, hvad et projekt som Linux kræver — og 'princippet om befaling' er praktisk umulig at anvende blandt frivillige i det anarkistiske paradis, som vi kalder Internet. For at fungere og konkurrere effektivt må hackere, som vil lede samarbejdende projekter, lære at rekruttere og motivere de nødvendige og effektive interessemiljøer på en måde, der svagt antydes i Kropotkins 'princip om forståelse'. De må lære at følge Linus' lov. [SP]

Tidligere henviste jeg til 'delfi-effekten' som en mulig forklaring af Linus' lov. Men bedre analogier til adaptive systemer indenfor biologi og økonomi trænger sig uimodståeligt på. Linux-verdenen opfører sig påmange måder som det frie marked eller et økosystem, en samling selviske agenter, som forsøger at maksimere nytte, en proces, der medfører selvkorrigerende og spontan orden med mere fuldendelse og effektivitet, end en hvilken som helst grad af central planlægning ville have opnået. Det er altså her, man skal søge efter 'princippet om forståelse'.

'Nyttefunktionen', som Linux-hackere maksimerer, er ikke klassisk økonomisk, men det er den uhåndgribelige tilfredsstillelse af ego og anseelse blandt andre hackere (man kunne kalde deres motivation 'uegennyttig', men det ville være at ignorere det faktum, at uegennyttighed er en form for tilfredsstillelse af ego for den uegennyttige). Frivillige kulturer, der fungerer på denne måde, er faktisk ikke ualmindelige; en anden kultur, hvor jeg har været med, er science fiction, som i modsætning til hacker-kulturen, udtrykkeligt anerkender 'egoboo' (forøgelse af ens anerkendelse blandt andre fans) som drivkraft for frivillig aktivitet.

Linus har vist en fin fornemmelse for Kropotkins 'princip om fælles forståelse' ved succesfuldt at placere sig som vogter for et projekt, hvor udviklingen for det meste udføres af andre, og ved at støtte interessen for projektet indtil den blev selvunderstøttende. Denne halvøkonomiske anskuelse af Linux-verdenen viser os, hvordan denne forståelse anvendes i praksis.

Vi kan se Linus' metode som en måde at skabe et marked for egoboo — at knytte enkelte hackeres selviskhed så tæt sammen som muligt omkring et vanskeligt formål, der kun kan nås ved vedholdende samarbejde. Med Fetchmail-projektet har jeg vist (omend i mindre målestok), at hans metode kan kopieres med gode resultater. Måske har jeg gjort det en smule mere bevidst og systematisk, end han gjorde.

Mange mennesker (især dem, som politisk er mistroiske over for det frie marked) ville forvente, at en kultur med selvstyrende egoister ville være fragmenteret, territorial, præget af splid, hemmelighedsfuld og fjendtlig. Men denne forventning er klart modbevist af (bare for at give et eksempel) den slående varians, kvalitet og dybde af Linux' dokumentation. Alle ved, at programmører hader at at dokumentere; hvordan kan det så være, at Linux-hackere genererer så meget af det? Åbenbart er Linux' frie marked for egoboo bedre til at producere dydig adfærd, der er styret af andre end de kommercielle software-producenters massivt financierede dokumentationsforretning.

Både Fetchmail- og Linuxkerne-projektet viser, at ved at belønne mange andre hackeres ego ordentligt, kan en stærk udvikler/koordinator bruge Internet til at opnå fordelene ved at have mange udviklere uden at projektet kollapser i kaotisk rod. Så til Brooks' lov har jeg følgende modforslag:

> *19: Under forudsætning af at udviklingskoordinatoren har et medium, som er mindst så godt som Internet, og forstår at lede uden tvang, er mange hoveder uundgåeligt bedre end et.*

Jeg tror, at open source i fremtiden i stigende grad vil tilhøre folk, som ved, hvordan man spiller Linus' spil, folk, som forlader katedralen og lader sig indrulle i basaren. Det betyder ikke, at individuelle visioner og åndrighed ikke vil have nogen betydning; jeg tror snarere, det mest banebrydende open source-software vil komme fra folk, som starter med en individuel vision og åndrighed og såforstærker det ved at opbygning af frivillige interessefællesskaber.

Og i fremtiden måske ikke kun open source-software. Ingen udvikler af lukket kildekode vil kunne måle sig med den talentmasse, som Linux-miljøet retter mod et problem. Meget få vil have

råd til at ansætte bare mere end de to hundrede (1999: 600. 2000: 800), som har bidraget til Fetchmail!

Måske vil open source-kulturen til sidst triumfere ikke på grund af, at samarbejde er moralsk korrekt, eller at 'hamstring' af software er moralsk forkert (under forudsætning af, at du mener det sidste, hvilket hverken Linus eller jeg gør), men simpelthen fordi en verden med lukket kildekode ikke kan vinde et evolutionært oprustningkapløb mod open source-miljøer, som kan sætte store mængder af kvalificeret tid ind på at løse et problem.

11. Om ledelse og Maginotlinien

Den oprindelige 'Katedralen og basaren' sluttede med ovenstående vision — at glade netværksforbundne horder af programmører/anarkister udkonkurrerede og overvældede det lukkede hierarki i den konventionelle software-verden.

En del skeptikere blev dog ikke overbevist; og deres spørgsmål fortjener at blive taget alvorlige. De fleste af indvendingerne mod basar-argumenterne drejer sig egentlig om, at fortalerne herfor har undervurderet den produktivitetsforøgende effekt ved konventionel ledelse.

Traditionelt indstillede ledere af softwareudviklere hævder, at den skødesløshed, projektgrupper skabes, ændres og opløses med, ophæver en betydelig del af de tilsyneladende fordele i antal, som open source-miljøet har overfor den enkelte udvikler af lukket kildekode. De vil hævde, at med hensyn til udvikling af software er det i virkeligheden vedholdende indsats over tid og det, at kunderne kan forvente en fortsat investering i produktet, som betyder noget, ikke bare hvor mange mennesker, der har smidt et ben i gryden og såladet den stå at simre.

Der er klart noget om dette argument; faktisk har jeg i The Magic Cauldron* arbejdet på den ide, at den forventede værdi af service i fremtiden er nøglen til økonomien bag udvikling af software.

Men dette argument har også et stort skjult problem; dets implicitte antagelse af at udvikling af open source ikke leverer en sådan vedholdende indsats. Faktisk har der været open source-projekter, der har fastholdt en sammenhængende retning og et effektivt vedligeholdelsesniveau over ganske lange perioder uden den slags tilskyndelsestrukturer og institutionelle kontroller, som konventionelle ledere finder nødvendige. Udviklingen af editoren GNU Emacs er et ekstremt og lærerigt eksempel; den har indarbejdet indsatsen fra hundrede af bidragsydere gennem femten år til en samlet arkitektonisk vision, på trods af høj gennemstrømning og det faktum, at kun en person (forfatteren) vedvarende har været aktiv i al den tid. Ingen editor med lukket kildekode er nogen sinde kommet i nærheden af denne holdbarhedsrekord.

Dette antyder, at der er en grund til at stille spørgsmål ved fordelene ved konventionelt ledet udvikling af software, der er uafhængig af resten af argumenterne omkring katedral- overfor basar-måden. Hvis det er muligt for GNU Emacs at fremvise en konsistent arkitektonisk vision gennem femten år, eller for et operativsystem som Linux at gøre det samme gennem otte år med voldsomt skiftende teknologi med hensyn til hardware og platforme; og hvis der (som det faktisk er tilfældet) har været mange open source-projekter med god arkitektur og en levetid på mere end fem år — så er vi berettigede til at undre os over, hvad — hvis noget — vi får for de enorme faste omkostninger forbundet med konventionelt ledet udvikling.

Hvad det end er, så indebærer det i hvert fald ikke troværdig overholdelse af deadlines, budget eller faciliteter i specifikationen; det er sjældent, at et 'ledet' projekt overholder bare et af disse krav endsige alle tre. Det synes heller ikke at være evne til at tilpasse sig forandringer i teknologi eller økonomisk sammenhæng; open source-miljøet har vist sig meget mere effektivt i den henseende (som det for eksempel klart bevises ved en sammenligning af Internets tredive-årige historie med de proprietære netværksteknologiers halvliv — eller omkostningen ved Microsoft Windows skift fra 16-bit til 32-bit og Linux' næsten lette opmigrering i den samme periode, ikke bare inden for udviklingen af Intel-linien, men også inden for mere end et dusin andre hardware-platforme også inklusive 64-bit Alpha).

Mange mennesker tror, at de på den traditionelle måde får en, de kan holde juridisk ansvarlig og eventuelt hente kompensation fra, hvis projektet går galt. Men det er en illusion; de fleste software-licenser indeholder forbehold for garantien af holdbarhed og endda ydelse — og tilfælde af succesfuld

* Se `http://www.laisen.dk/Den_magiske_kedel.1094.0.html`.

erstatning for manglende ydelse er forsvindende få. Selv hvis erstatninger var almindelige, ville det være forfejlet at være beroliget af følelsen af at kunne sagsøge nogen. Du ønskede ikke et søgsmål; du ønskede software, der virkede.

Så hvad får du for alle de faste omkostninger?

For at forstå det, må vi undersøge, hvad ledere af softwareudvikling tror, de gør. Jeg kender en kvinde, som virker som om, hun er meget god til det her job, og som siger, at projektledelse af software har fem funktioner:

At *definere mål* og holde alle rettet i den samme retning.

At *holde øje med* og sikre at afgørende detaljer ikke bliver sprunget over.

At *motivere* folk til at udføre kedeligt men nødvendigt slavearbejde.

At *organisere* anvendelsen af folk for at opnå den bedste produktivitet.

At *styre de nødvendige ressourcer* for at opretholde projektet.

Tilsyneladende alle værdige mål; men under open source-modellen og dens sociale sammenhæng begynder de at virke underligt irrelevante. Vi gennemgår dem i den modsatte rækkefølge.

Min ven fortæller, at meget *styring af ressourcer* i bund og grund er defensiv; når du har dine folk, maskiner og kontorpladser, skal du forsvare dem overfor dine med-ledere, der kæmper om de samme ressourcer, og dem højere oppe, der prøver at allokere den mest effektive brug af en begrænset gruppe.

Men open source-udviklere er frivillige, selvvalgte med både interesse for og evne til at bidrage til projekterne, de arbejder på (og det gælder hovedsagelig stadig, selv om de får løn for at hacke open source). Den frivillige etos har tendens til automatisk at tage sig af 'angrebssiden' af ressourcestyringen; folk bringer selv deres egne ressourcer til bordet. Og der er kun lidt eller intet behov for en leder til at 'spille i forsvaret' i den konventionelle forstand.

Under alle omstændigheder viser det sig konstant, i en verden med billige PC'er og hurtige forbindelser til Internet, at den eneste virkeligt begrænsede ressource er faglært opmærksomhed. Når open source-projekter mislykkes, gør de det i bund og grund aldrig på grund af mangel på maskiner, forbindelser eller kontorplads; de dør kun, når udviklerne selv mister interessen.

Når det er tilfældet, er det dobbelt så vigtigt, at open source-hackere *organiserer sig selv* for at opnå maksimal produktivitet gennem selvudvælgelse — og det sociale miljøvælger hensynsløst pågrundlag af kompetence. Min ven, der er bekendt med både open source-verdenen og store lukkede projekter, mener, at open source delvist har været succesfuld, fordi kulturen kun accepterer de cirka 5% mest talentfulde af programmørerne. Hun bruger det meste af sin tid på at organisere anvendelsen af de sidste 95%, og hun har derfor førstehåndserfaringer med den velkendte forskel i form af en faktor hundrede i produktivitet mellem de dygtigste programmører og de almindeligt kompetente.

Størrelsen af denne forskel har altid rejst et pinligt spørgsmål: ville individuelle projekter, og faget som sådan, være bedre tjent uden mere end 50% af de mindst dygtige? Fornuftige ledere har længe forstået, at hvis ledelse af konventionel software kun drejer sig om at ændre de mindst dygtige fra et netto tab til en marginal gevinst, så er indsatsen måske ikke besværet værd.

Open source-miljøets succes gør spørgsmålet mere aktuelt med sit klare bevis på, at det ofte er billigere og mere effektivt at rekrutere selvudvalgte frivillige på Internet, end det er at lede bygninger fulde af folk, som hellere ville lave noget andet.

Hvilket bekvemt bringer os videre til spørgsmålet om *motivation*. En lignende og ofte hørt måde at fremsætte min vens pointe er, at traditionel udviklingsledelse er en nødvendig kompensation for utilstrækkeligt motiverede programmører, der ellers ikke ville udføre et godt stykke arbejde.

Dette svar kommer sædvanligvis sammen med en påstand om, at man kun kan regne med, at open source-miljøet vil udføre arbejde, der er 'sexet' eller teknisk interessant; intet andet vil blive udført (eller udført dårligt), med mindre det er fabrikeret på stribe af daglejere, der er motiveret af penge, i små kontorer med ledere, der svinger pisken over dem. Jeg skriver om de psykologiske og sociale grunde til at være kritisk over for denne påstand i 'Homesteading the Noosphere'. Men til dette formål mener jeg, at det er mere interessant at påpege følgerne af at acceptere påstanden som værende sandfærdig.

Hvis den konventionelle stærkt ledede måde at udvikle software med lukket kildekode kun

forsvares af en slags Maginotlinie af problemer, som bidrager til kedsommelighed, så vil det kun være holdbart for hver enkelt applikation så længe, at ingen synes problemerne er særligt interessante, og ingen andre finder en måde at komme uden om dem på. For i det øjeblik, at der opstår en open source-konkurrence om et stykke 'kedeligt' software, vil kunderne vide, at det i sidste ende var løst af en, som valgte at løse det problem, fordi problemet var fascinerende i sig selv — hvilket inden for software som inden for andre slags kreativt arbejde er en langt mere effektiv motivator end pengene alene.

At have en konventionel ledelsesstruktur udelukkende for at motivere er sikkert god taktik men dårlig strategi; en kortsigtet gevinst men i det lange løb et mere sikkert tab.

Indtil videre ser konventionel udviklingsledelse på to punkter (styring af ressourcer og organisering) nu ud som en dårlig satsning overfor open source, og den lever på lånt tid med hensyn til til en tredie (motivation). Og den stakkels belejrede konventionelle leder vil ikke få undsætning fra spørgsmålet om at holde øje med afgørende detaljer; det stærkeste argument, open source-miljøet har, er, at en decentraliseret gennemgang blandt ligemænd vinder over alle konventionelle metoder i at sikre, at detaljer ikke overses.

Kan vi bruge *definering af mål* som en berettigelse for de faste omkostninger ved ledelse af konventionel udvikling af software? Måske; men for at gøre det, har vi behov for en god grund til at tro, at ledelsesgrupper og virksomhedernes faste måder at gøre ting på er bedre til at definere værdige og bredt accepterede mål end de projektledere og stammeældste, som har den tilsvarende rolle i open source-verdenen.

Det er en svær pille at sluge uden beviser. Og det er ikke så meget open source-siden af argumentet (Emacs' holdbarhed eller Linus Torvalds' evne til at mobilisere horder af udviklere med tale om 'verdensdominans'), der gør det svært. Snarere er det den forfærdelige demonstration af de konventionelle mekanismer til definition af mål for software-projekter.

Et af de bedst kendte, folkelige teoremer om softwareudvikling er, at 60% til 75% af konventionelle software-projekter enten aldrig bliver færdiggjorte eller bliver forkastet af deres kommende brugere. Hvis det bud er bare i nærheden af at være sandt (og jeg har aldrig mødt en leder med nogen erfaring, som bestrider det), så er flertallet af projekter rettet mod mål, der er enten (a) ikke realistisk opnåelige eller (b) fuldstændigt forkerte.

Det er mere end noget andet grunden til, at det inden for softwareudvikling nu om dage sandsynligvis løber en koldt ned af ryggen, når blot vendingen 'ledelsesgruppe' nævnes — selv (eller måske især) hvis den, der hører det, selv er leder. De dage, hvor kun programmører brokkede sig over dette mønster, er ovre; 'Dilbert'-striber hænger nu over *chefers* skriveborde.

Så vores svar til den traditionelle leder af softwareudvikling er enkelt — hvis open source-miljøet virkeligt har undervurderet værdien af konventionel ledelse, *hvorfor viser så mange af jer så foragt for jeres egen proces?*

Endnu en gang skærper eksistensen af open source-miljøet dette spørgsmål — for vi har det sjovt, mens vi gør det. Vores kreative leg har skabt tekniske succeser med en markedsandel og offentligt kendskab i et forbavsende antal. Vi beviser, at vi ikke bare kan lave bedre software, men at *glæde også er en fordel.*

To et halvt år efter den første version af dette essay er den mest banebrydende tanke, jeg kan afslutte med ikke længere en vision om en open source-domineret software-verden; det ser for tiden, når alt kommer til alt, ud til at være en mulighed for en masse besindige mennesker i jakkesæt.

Jeg vil snarere foreslå, hvad der vil være en dybere lektion om software (og måske om enhver slags kreativt og professionelt arbejde). Mennesker er generelt glade for en opgave, når den falder i en slags optimal udfordringszone; ikke for let til at være kedelig, ikke for svær til at løse. En tilfreds programmør er en, der hverken bliver udnyttet tilstrækkeligt eller tynget af dårligt formulerede mål og stressede arbejdsprocesser. *Morskab går forud for effektivitet.*

Med hensyn til din egen arbejdssituation med frygt og afsky (selv i den fortrængte og ironiske måde i form af ophængning af Dilbert-striber) burde derfor i sig selv betragtes som et tegn på, at processen har fejlet. Glæde, humor og leg er sandelig en fordel; det var ikke bare for sjov, jeg skrev 'glade horder' ovenfor, og det er ikke mere en vittighed, at Linux' maskot er en sød, neotenisk pingvin.

Det vil måske vise sig, at en af de mest vigtige følger af open source' succes vil være at lære os, at leg er den økonomisk mest effektive måde at udføre kreativt arbejde på.

12. Anerkendelser

Dette skrift er forbedret gennem samtaler med et stort antal mennesker, som hjalp til med at finde fejl. Især tak til Jeff Dutky (dutky@wamumd.edu), som foreslog formuleringen 'fejlsøgning er paralleliseret', og som hjalp med at udvikle analysen, som følger deraf. Også til Nancy Lebovitz (nancyl@universe.digex.net) for hendes forslag om, at jeg efterlignede Weinberg ved at citere Kropotkin. Perspektiverende kritik kom også fra Joan Eslinger (wombat@kilimanjaro.engr.sgi.com) og Mart Franz (marty@net-link.net) fra General Technics-postfordelingslisten. Glen Vandenburg påpegede vigtigheden af, at bidragsyderne selv foretager en udvælgelse, og han antydede, at megen udvikling retter op på udeladelsessynder; Daniel Upper foreslog de naturlige analogier til dette. Jeg er taknemmelig overfor medlemmerne af PLUG, Philadelphia Linux User's Group, der var det første testpublikum af den første offentlige version af dette skrift. Paula Matuszek lærte mig om ledelse inden for inden for software. Phil Hudson mindede mig om, at den sociale organisering af hacker-kulturen er et spejl af organiseringen af dets software om omvendt. Endelig var Linus Torvalds' kommentarer hjælpsomme og hans tidlige støtte meget opmuntrende.

13. Yderligere læsning

Jeg citerede adskillige stykker fra Frederick P. Brooks klassiske 'The Mythical Man-Month', da hans indsigter på mange måder ikke er blevet forbedret. Jeg anbefaler varmt 25-årsudgaven fra Addison-Wesley (ISBN 0-201-83595-9), som også indeholder hans skrift fra 1986 'No Silver Bullet'.

Den nye udgave indeholder et uundværlig tilbageblik, der er skrevet 20 år senere, hvor Brooks åbent indrømmer de få fejlbedømmelser i den originale tekst, som ikke har holdt stik. Jeg læste først tilbageblikket efter, at dette skrift stort set var færdig, og jeg var overrasket over, at Brooks tillægger Microsoft basar-lignende metoder (i virkeligheden viste det sig at være forkert. I 1998 afslørede the Halloween Documents*, at Microsofts interne udviklingsmiljøer stærkt balkaniseret, og hvor den grad af generel adgang til kildekoden, der skal til at understøtte en basar, ikke engang er mulig)!

Gerald M. Weinbergs 'The Psychology Of Computer Programming' (New York, Van Nostrand Reinhold 1971) introducerede det ret uheldigt formulerede begreb om 'jegløs programmering'. Selv om han ikke engang var i nærheden af at være den første, der indså det nytteløse i 'princippet om befaling', var han sikkert den første til at anerkende og tale for pointen i direkte sammenhæng med udvikling af software.

Da Richard P. Gabriel betragtede Unix-kulturen fra før Linux-tiden, talte han modvilligt for en primitiv basar-lignende models overlegenhed i sit skrift 'Lisp: Good News, Bad News, and How To Win Big' fra 1989. Selv om det er forældet på nogle områder, er essayet stadig med rette værdsat blandt fans af Lisp (mig selv inklusive). En læser mindede mig om, at afsnittet med titlen 'Worse Is Better' kan læses som en forudanelse om Linux. Skriftet er tilgængeligt på World Wide Web på http://www.naggum.no/worse-is-better.html.

De Marco og Listers 'Peopleware: Productive Projects and Teams' (New York; Dorset House, 1987; ISBN 0-932633-05-6) er en perle, der ikke er tilstrækkeligt værdsat, og som jeg var glad for at se Fred Brooks citere i sit tilbageblik. Selv om kun lidt af, hvad forfatterne skriver, er direkte anvendeligt på Linux- eller open source-miljøet, er forfatternes indsigt i de nødvendige betingelser for kreativt arbejde skarpsindige og værdifulde for enhver, som forsøger at bruge nogle af basar-modellens dyder i en kommerciel sammenhæng.

Til sidst må jeg indrømme, at jeg næsten kaldte dette skrift 'Katedralen og agoraen', hvor det sidste udtryk er græsk for et åbent marked eller offentligt mødested. De skabende skrifter 'agoric systems' af Mark Miller og Eric Drexler, som beskriver de markedslignende egenskaber inden for computer-økosystemer, som er ved at opstå, hjalp mig med at tænke klart omkring analoge

* `http://www.catb.org/ esr/halloween/.`

fænomener i open source-kulturen, da Linux udpenslede det for mig fem år senere. Disse skrifter er tilgængelige på nettet påhttp:/www.agorics.com/agorpapers.html.

14. Epilog: Netscape tager basaren til sig!

Det er en underlig fornemmelse at blive klar over, at man er med til skabe historie...

Den 22. januar 1998, omkring 7 måneder efter jeg først havde offentliggjort dette skrift, annoncerede Netscape Communications, Inc. deres planer om at frigive kildekoden til Netscape Communicator*. Jeg havde ingen ide om, at det ville ske, før jeg hørte om det.

Eric Hahn, Executive Vice President og Chief Technology Officer hos Netscape sendte mig kort derefter følgende i en e-mail: 'På vegne af alle i Netscape vil jeg gerne takke dig for at hjælpe os til overhovedet at nå til dette punkt. Dine ideer og skrifter var en fundamental inspiration til vores beslutning'.

Den efterfølgende uge fløj jeg til Silicon Valley inviteret af Netscape til en dagslang strategikonference (den 4. februar 1998) med nogle af deres topchefer og tekniske folk. Vi lavede sammen Netscapes strategi for frigivelse af kildekoden og licensen.

Nogle få dage senere skrev jeg følgende:

Netscape er ved at give os en storstilet, virkelig test af basar-modellen i den kommercielle verden. Open source-kulturen står nu over for en fare: hvis Netscapes udførelse ikke virker, kan open source-konceptet blive så miskrediteret, at den kommercielle verden ikke vil forsøge det igen i det næste årti.

På den anden side er det også en spektakulær mulighed. De første reaktioner på beslutningen på Wall Street og andre steder har været forsigtigt positive. Vi har også fået en chance for at bevise vores eget værd. Hvis Netscape genvinder betydelig markedsandel gennem denne beslutning, kan det starte en revolution i software-industrien, som har ladet vente alt for længe på sig.

Det næste år vil blive en meget lærerig og interessant tid.

Og det blev det virkelig. I midten af 1999 har udviklingen af det, der senere blev navngivet 'Mozilla' kun været en begrænset succes. Netscapes oprindelige mål blev nået, som var at forhindre Microsoft i at opnå monopolstatus på browsermarkedet. Det opnåede ogsåen vis dramatisk succes (specielt frigivelsen af næste-generations renderingsmaskinen Gecko).

På den anden side har det endnu ikke fremprovokeret den massive udvikling udefra, som Mozilla-grundlæggerne oprindeligt håbede på. Problemet synes at være, at Mozilla-distributionen i lang tid faktisk brød en af de grundlæggende regler for udvikling under basar-modellen: de frigav ikke noget potentielle udviklere kunne køre og se virke (indtil mere end et år efter frigivelsen, krævede det en licens til det proprietære Motif-bibliotek for at måtte oversætte Mozilla fra kildekoden).

Værst er det (set fra resten af verdenens synspunkt), at Mozilla-gruppen endnu ikke har udsendt en browser af produktionskvalitet — og en af projektets ledere skabte noget af en sensation ved at droppe ud af projektet, klagende over dårlig ledelse og manglende muligheder. 'Open source', observerede han korrekt, 'er ikke tryllepulver'.

Og det er det ikke. Langtidsprognosen for Mozilla-projektet ser en del bedre ud nu (i august 1999), end den gjorde i tiden omkring Jamie Zawinskis afsked — men han havde ret op til det punkt, at det at frigive koden ikke nødvendigvis vil redde et eksisterende projekt, der i forvejen lider af dårligt definerede mål, spaghetti-kode eller andre af software-udviklingens kroniske sygdomme. Mozilla har præsteret at give os samtidige eksempler på hvordan open source kan føre til succes og samtidigt fejle.

I mellemtiden har open source-ideen dog haft succes og fundet støtte andre steder. 1998 og sidste del af 1999 var der en fantastisk eksplosion i interessen for open source-udviklingsmodellen, en trend både drevet af og drivende Linux-operativsystemets fortsatte succes. Den trend, Mozilla satte i gang, fortsætter nu i imponerende fart.

15. Slutnoter

* Se `http://www.netscape.com/newsref/pr/newsrelease558.html`.

[JB] *I Programming Pearls* kommenterer den kendte computeraforist Jon Bentley Brooks antagelse ved at sige 'Hvis du planlægger at smide en væk, vil du smide to væk'. Han har næsten sikkert ret. Pointen i Brooks og Bentleys antagelse er ikke kun, at du må forvente at første forsøg vil gå galt, det er ofte mere effektivt, at starte forfra med den rigtige ide end at forsøge at redde noget rod.

[QR] Der har været eksempler på open source, basar-udvikling, der er fra før Internet-eksplosionen, og som ikke er relateret til traditionerne omkring Unix og Internet. Udviklingen af info-Zipkomprimeringsværktøjet* omkring 1990-1992 især til DOS-maskiner var sådan et eksempel. Et andet var RBBS bulletin board-systemet (igen til DOS), som begyndte i 1983 og udviklede et tilstrækkeligt stærkt miljø, og der har været temmeligt regelmæssige frigivelser op til nu (midten af 1999) på trods af vældige teknologiske fremskridt med email og frem for alt fildeling på lokale BBS'er. Mens info-Zip-miljøet i nogen grad brugte email, var udviklerkulturen omkring RBBS faktisk i stand til at basere et væsentligt online-miljøpå på RBBS, som var fuldstændigt uafhængig af TCP/IP-infrastrukturen.

[JH] John Hasler har foreslået en interessant forklaring på det faktum, at fordobling af indsats ikke synes at bremse open source-udvikling. Han foreslår, hvad jeg vil kalde 'Haslers lov': omkostningerne ved dobbelt arbejde har tendens til at vokse med mindre end kvadratet på team-størrelsen — det vil sige langsommere end de faste omkostninger forbundet med den planlægning og ledelse, som ville være nødvendig for at eliminere dem.

Denne påstand er ikke i modstrid med Brooks lov. Det er måske sådan, at de samlede faste omkostninger ved kompleksitet og sårbarhed overfor fejl vokser med kvadratet på team-størrelsen, mens omkostningerne ved dobbelt arbejde ikke desto mindre er et særligt tilfælde, der vokser langsommere. Det er ikke svært at udvikle plausible grunde til dette, startende med det utvivlsomme faktum, at det er meget lettere at blive enige om funktionelle grænser mellem forskellige udvikleres kode, som vil forhindre dobbelt indsats, end det er at forhindre de forskellige uplanlagte, dårlige samspil over hele systemet, der ligger til grund for de fleste fejl.

Kombinationen af Linus' lov og Haslers lov antyder, at der faktisk er tre kritiske størrelsesforhold i software-projekter. I mindre projekter (med mellem en og højst tre udviklere) behøver ledelsesstrukturen ikke at være mere kompliceret end at udnævne en hovedprogrammør. Og der er en mellemgruppe derover, hvor omkostningerne ved traditionel ledelse er relativt lave, så fordelene ved at undgå dobbelt arbejde, fejlfinding og overvågning, så detaljer ikke overses, er en netto fordel.

Men derover betyder kombinationen af Linus' lov og Haslers lov, at omkostningerne og problemerne i traditionelt ledet store projektgrupper stiger meget hurtigere end den forventede omkostning ved dobbelt indsats. Ikke mindst består en del af omkostningen af en strukturelt bestemt manglende evne til at udnytte effekten af mange øjne, der (som vi har set) meget bedre end traditionel ledelse sikrer, at fejl og detaljer ikke overses. I store projekter er det således kombinationen af disse love, der effektivt får nettogevinsten ved traditionel ledelse til at gå mod nul.

[HBS] Opsplitningen i Linux' eksperimentelle og stabile versioner har en anden funktion, som er beslægtet med, men alligevel forskellig fra sikring mod risici. Opsplitningen bekæmper et andet problem: de farlige tidsfrister. Når programmører skal opfylde en uforanderlig facilitets-liste og nå en fastsat definitiv deadline, går kvaliteten tabt, og der er sandsynligvis et stort rod på vej. Jeg er taknemmelig overfor Marco Iansiti og Allan MacCormack fra Harvard Business School, som gjorde mig opmærksom på, at planlægningen kan lykkes, hvis der slækkes på et af disse krav.

En måde at gøre dette på er at fastholde fristen men lade facilitets-listen være fleksibel og fjerne faciliteter, hvis de ikke er færdige inden fristen udløber. Det er i grunden strategien for den 'stabile' udviklingsgren af kernen; Alan Cox (der vedligeholder den stabile kerne) sender frigivelser ud med ret jævnlige intervaller men garanterer ikke, hvornår specifikke fejl vil være rettet, eller hvornår faciliteter fra den eksperimentelle kerne implementeres i den stabile.

Den anden måde at gøre dette er at fastsætte en ønskelig facilitets-liste og først levere, når det er lavet. Det er i grunden strategien for den 'eksperimentelle' udviklingsgren. De Marco og Lister citerede forskning, der viste, at denne planlægningspolitik ('væk mig, når arbejdet er udført') ikke

* Se `http://www.cdrom.com/pub/infozip/`.

bare giver den bedste kvalitet men i gennemsnit kortere afleveringstider end enten 'realistisk' eller 'aggressiv' planlægning.

Jeg er begyndt at tro (i starten af 2000), at jeg i tidligere udgaver af dette dokument alvorligt undervurderede vigtigheden af 'væk mig, når arbejdet er udført' anti frist-politikken for open source-miljøets produktivitet og kvalitet. Den generelle erfaring fra den forcerede GNOME 1.0 i 1999 antyder, at pres for en for tidlig frigivelse kan neutralisere mange af kvalitetsfordelene, som open source normalt giver.

Det kan meget vel vise sig, at den gennemskuelige proces inden for open source er en af tre ligeværdige drivkræfter bag kvaliteten sammen med 'væk mig, når arbejdet er udført'-planlægning og selvudvælgelse blandt udviklere.

[IN] Det er et emne beslægtet med spørgsmålet, om man kan starte projekter fra bar bund efter basar-metoden, hvorvidt basar-metoden er i stand til at understøtte virkeligt innovativt arbejde. Nogle hævder, at da basaren mangler stærkt lederskab, kan den kun håndtere kloningen og forbedringen af ideer, der allerede findes i produktion, men ikke er i stand til at være markedsførende. Dette argument blev måske mest berygtet fremsat i the Halloween Documents, to pinlige, interne Microsoft-notater skrevet om open source-fænomenet. Forfatterne sammenlignede Linux' udvikling af et Unix-lignende operativsystem med at 'løbe efter biler' og hævdede, at '(når et projekt har opnået 'paritet' med de markedsførende) bliver det nødvendige behov for ledelse massivt for blive i stand til at bryde igennem nye grænser'.

Der er alvorlige faktuelle fejl indeholdt i dette argument. En viser sig, da Halloween-forfatterne senere selv bemærker, at 'ofte [...] er nye forskningsideer først implementeret og tilgængelige på Linux, før de er tilgængelige/indarbejdede på andre platforme'.

Hvis vi læser 'open source' i stedet for 'Linux', ser vi, at dette langt fra er et nyt fænomen. Historisk set opfandt open source-miljøet ikke Emacs, World Wide Web eller Internet ved at løbe efter biler eller ved at blive massivt ledet — og i nutiden foregår der så meget innovativt arbejde inden for open source, at man er forkælet i sit valg. GNOME-projektet (for at vælge et af mange) flytter grænser for GUI og objektorienteret teknologi så voldsomt, at det har tiltrukket sig opmærksomhed i såvel computerbladene som uden for Linux-miljøet. Der er talrige andre eksempler, som et besøg når som helst på Freshmeat hurtigt vil vise.

Men der er mere fundamentale fejl i den implicitte antagelse, at *katedral-modellen* (eller basar-modellen eller en hvilken som helst anden ledelsesstruktur) på en eller anden måde kan gøre innovativ skabelse pålidelig. Det er nonsens. Grupper får ikke gennembrydende indsigt — selv frivillige grupper af basar-anarkister er sædvanligvis ikke i stand til at være ægte originale, endnu mindre er folk i arbejdsgrupper, der har overlevelsesinteresse i et status quo ante. *Indsigt kommer fra individer.* Det eneste, det sociale maskineri omkring dem kan håbe på at gøre, er at være *lydhøre* over for gennembrydende indsigter — at nære, belønne og grundigt teste dem i stedet for at undertrykke dem.

Nogle vil karakterisere dette som et romantisk synspunkt, en tilbagevenden til gammeldags stereotyper med ensomme opfindere. Det er det ikke; jeg påstår ikke, at grupper er ude af stand til at *udvikle* gennembrydende indsigter, når de er udklækket; vi har faktisk lært af processen med gennemgang blandt ligemænd, at sådanne udviklingsgrupper er essentielle for at kunne producere et resultat af høj kvalitet. Snarere påpeger jeg, at alle sådanne udviklingsgrupper starter med — er nødvendigvis sat i gang af — en god ide i en persons hoved. Katedraler, basarer og andre sociale strukturer kan opfange indsigten og raffinere den, men de kan ikke frembringe den påkommando.

Derfor er grundproblemet med innovation (inden for software og alle andre steder) især hvordan man undgår at undertrykke det — men endnu mere fundamentalt hvordan man *producerer masser af folk, der overhovedet kan få indsigten i første omgang.*

Det vil være absurd at antage, at udvikling efter katedral-metoden kan udføre dette trick, mens basarens lave adgangskrav og flydende processer ikke kan. Hvis det kræver en person med en god ide, så vil et socialt miljø, hvor en person lynhurtigt kan tiltrække hundreder eller tusinder af andre i et samarbejde om den gode ide, uundgåeligt være mere innovativt end et miljø, hvor personen skal igennem et politisk salgsarbejde i hierakiet, inden han kan arbejde på sin ide uden at risikere at blive fyret.

Og især, hvis vi ser historisk på innovation inden for software i organisationer, der anvender katedral-modellen, opdager vi hurtigt, at det er sjældent. Store virksomheder binder an på universitetsforskning for at få nye ideer (derfor er forfatterne af Halloween Documents utrygge ved Linux' evne til at supplere sig selv med forskning hurtigere). Eller de overtager små firmaer opstået omkring en innovators hjerne. I ingen af tilfældene sker innovationen kun under katedral-kulturen; faktisk ender mange innovationer, der er importeret på den måde, med at blive kvalt af 'det massive ledelsesniveau', som forfatterne til Halloween Document lovpriser.

Det er dog en negativ pointe. Læseren vil være bedre tjent med en positiv pointe. Jeg foreslår følgende som et eksperiment:

(a) Vælg et kriterie for originalitet, som du mener, du kan bruge konsekvent. Hvis din definition er 'jeg ved det, når jeg ser det', erdet ikke noget problem for hensigten med denne test.

(b) Vælg et hvilket som helst operativsystem med lukket kildekode, der konkurrerer med Linux, og vælg den bedste kilde til information om igangværende udviklingsarbejde.

(c) Hold øje med den kilde og Freshmeat i en måned. Tæl hver dag antallet af meddelelser om frigivelser, som du betragter som 'originalt' arbejde. Anvend den samme definition på meddelelser om det andet operativsystem og tæl også dem.

(d) Opsummer og sammenlign tredive dage senere.

Den dag, jeg skrev dette, var der toogtyve meddelelser om frigivelser på Freshmeat, hvoraf tre virker som om, de vil være banebrydende påen eller anden måde. Det var en sløv dag på Freshmeat, men jeg vil blive forbavset, hvis nogen læser rapporterer om tre potentielle innovationer på en måned fra en kilde til lukket kildekode.

[EGCS] Vi kender nu et projekt, som på mange måder kan give os en bedre test af basar-præmissen end Fetchmail; EGCS*, det Eksperimentelle GNU Oversættersystem.

Dette projekt blev lanceret i midten af august 1997 som et bevidst forsøg at anvende ideerne fra de tidlige offentlige versioner af 'Katedralen og basaren'. Stifterne af projektet mente, at udviklingen af GCC, GNUs C-oversætter, var stagneret. I de efterfølgende tyve måneder fortsatte GCC og EGCS som parallelle projekter — begge trak på den samme gruppe af udviklere på Internet, begge startede med den samme GCC kodebasis, begge brugte stort set de samme Unix-værktøjer og udviklingsmiljøer. Projekterne adskilte sig kun derved, at EGCS bevidst prøvede at anvende basar-taktikkerne, jeg tidligere havde beskrevet, mens GCC beholdte en mere katedral-lignende organisation med en lukket udviklergruppe og sjældne frigivelser.

Det var omtrent så tæt på et kontrolleret eksperiment, som man kunne håbe på, og resultaterne var dramatiske. I løbet af få måneder var EGCS kommet væsentligt foran med hensyn til faciliteter; bedre optimering, bedre understøttelse af FORTRAN og C++. Mange mennesker synes, at EGCS' udviklingsudgave var mere pålidelig end den seneste stabile version af GCC, og store Linux-distributioner begyndte at skifte til EGCS.

I april 1999 opløste Free Software Foundation (den officielle sponsor for GCC) den originale GCC-udviklingsgruppe og overrakte officielt kontrollen med projektet til EGCS-styringsgruppen.

[SP] Selvfølgelig rejser Kropotkins kritik og Linus' lov nogle bredere spørgsmål om den social organisations kybernetik. Et andet folkeligt teorem om softwareudvikling antyder en af dem; Conways lov — der normalt lyder 'Hvis du har fire grupper, der hver arbejder på en oversætter, vil du få en oversætter med fire gennemløb'. Den originale fremstilling var mere generel: 'Organisationer, som designer systemer, er begrænsede til at lave designs, der er kopier af kommunikationsstukturen i disse organisationer'. Vi kan udtrykke det mere koncist som: 'Midlerne afgører resultaterne' eller endda 'Proces bliver til produkt'.

Det er derfor værd at bemærke, at organisationsform og funktion i open source-miljøet matcher på mange niveauer. Netværket er alt og over alt: ikke bare Internet, men menneskene, der udfører arbejdet, skaber et distribueret, løst forbundet netværk af ligemænd, som bidrager med en mangfoldighed af redundans, og som opløses uden problemer. I begge netværk er hver enkelt enhed kun vigtig i det omfang, at andre enheder vil samarbejde med den.

Gennemgang blandt ligemænd er essentiel for miljøets forbavsende produktivitet. Kropotkins pointe om magtforhold er videreudviklet i SNAFU-princippet: 'Ægte kommunikation er kun

* Se `http://egcs.cygnus.com/`.

mulig mellem ligemænd, fordi underordnede mere konsekvent bliver belønnet for at fortælle deres overordnede behagelige løgne end for at fortælle sandheden'. Kreativt teamwork er fuldstændigt afhængigt af ægte kommunikation og er således alvorligt hæmmet af tilstedeværelsen af magtforhold. Open source-miljøet, der effektivt er fri for sådanne magtforhold, lærer os på den anden side, hvor forfærdelig meget de koster i fejl, nedsat produktivitet og mistede muligheder.

Desuden forudsiger SNAFU-princippet en progressiv adskillelse i autoritative organisationer mellem beslutningstagere og virkeligheden efterhånden som flere og flere input til dem, der beslutter, har tendens til at blive behagelige løgne. Det er let at se hvordan, dette foregår på inden for konventionel udvikling af software; der er stærke tilskyndelser for de underordnede til at skjule, ignorere og minimere problemerne. Når denne proces bliver produkt, er software en katastrofe.