Suomenkielinen laitos Eric Steven Raymondin kirjasta The Cathedral and the Bazaar.

Sisältö

Luku 1

Tiivistelmä

Tässä esseessä analysoin yksityiskohtaisesti erästä menestyksekästä avoimen lähdekoodin projektia, Fetchmailia, jossa harkitusti kokeiltiin käytännössä yllättäviä, Linuxin historiaan pohjautuvia ohjelmistokehityksen teorioita. Tarkastelen näitä teorioita kahdesta perustavanlaatuisesti erilaisesta näkökulmasta. ”Katedraali”-malli on se, jota käytetään yleisesti kaupallisissa ohjelmistoissa, kun taas Linux-maailmassa toimii ”basaari”-malli. Tulen osoittamaan, kuinka nämä mallit juontavat juurensa vastakkaisiin käsityksiin ohjelmistovirheiden jäljittämisen (*debugging*) luonteesta. Sitten esitän Linux-kokemukseen pohjautuvan perustelun väitteelle, jonka mukaan ”riittävä määrä silmiä näkee kaikki virheet”, sekä esitän hedelmällisiä analogioita muiden itsenäisistä toimijoista koostuvien, itseään korjaavien järjestelmien kanssa. Lopuksi tarkastelen hieman, miten tämä kaikki vaikuttaa käsitykseemme tulevaisuuden ohjelmistoista.

Luku 2

Katedraali ja Basaari

Linux on vallankumouksellinen. Kuka olisi vuonna 1991 tullut ajatelleeksi, että maailmanluokan käyttöjärjestelmä voisi syntyä kuin taikaiskusta, osa-aikaisesta hakkeroinnista, johon osallistuu tuhansia ympäri maapalloa asuvia ohjelmistosuunnittelijoita ainoana yhteysmuotonaan Internetin hatarat rihmat?

En minä ainakaan. Kun Linux ui tutkaani alkuvuodesta 1993, olin jo ollut mukana Unixin ja avoimen lähdekoodin kehityksessä kymmenen vuotta. Olin ensimmäisiä GNU-projektin avustajia 1980-luvun puolivälissä ja olin julkaissut useita avoimen lähdekoodin ohjelmia netissä. Olin kehittänyt yksin tai muiden kanssa monia ohjelmia, joita käytetään laajalti vielä nykyäänkin, kuten Nethack, Emacsin VC- ja GUD-moodit ja Xlife. Luulin tietäväni kuinka nämä hommat tehdään.

Linux käänsi päälaelleen melkein kaiken luulemani. Olin jo vuosia saarnannut Unixin hyvää sanomaa pienistä työkaluista, nopeasta prototyyppien teosta ja jatkuvaan kehitykseen perustuvasta ohjelmoinnista. Uskoin myös että olisi olemassa tietty kriittinen kompleksisuustaso, jonka yläpuolella tarvittaisiin keskitetympää, *a priori* -lähestymistapaa. Uskoin, että tärkeimmät ohjelmat, käyttöjärjestelmät ja erittäin laajat työkalut, kuten Emacsohjelmointieditori, täytyisi rakentaa kuten katedraalit: huolellisesti yksittäisten gurujen tai pienten taituriryhmien työstämänä täydellisessä eristyksessä julkaisematta yhtäkään betaa ennenaikaisesti.

* Linus Torvaldsin kehitystyyli tuli yllätyksenä – julkaista aikaisin ja usein, jakaa muille kaikki mahdollinen ja olla umpimähkäisyyteen saakka avoin. Ei ollenkaan hiljaista, kunnioittavaa katedraalin rakentamista. Linux-yhteisö muistutti valtaisaa puheensorinan täyttämää basaaria erilaisine ohjelmajulistuksineen ja ajattelutapoineen, minkä vertauskuvana jokainen saattoi muokata Linuxin verkkosivuja ja osallistua. Tältä pohjalta yhtenäisen ja vakaan järjestelmän kehittäminen vaikuttaa mahdolliselta vain onnekkaiden sattumusten kautta.

Oli suoranainen järkytys, että basaari-kehitysmalli näytti toimivan ja vieläpä hyvin. Uutta opetellessani työskentelin yksittäisten projektien parissa ja yritin samalla selvittää miksi Linux-maailma ei hajonnut palasina ympäriinsä, vaan näytti voimistuvan voimistumistaan nopeudella, jota katedraalin rakentajat hädin tuskin pystyivät edes kuvittelemaan.

Vuoden 1996 puolivälin tietämillä luulin alkaneeni ymmärtää. Sattuma ojensi minulle täydellisen tavan laittaa teoriani kokeeseen: avoimen lähdekoodin projektin, jota voisin tietoisesti yrittää johtaa basaarityyliin. Tein niin, ja se oli huomattava menestys.

Tämä on kertomus tuosta projektista. Käytän sitä saadakseni ehdottaa joitakin ajatelmia,

jotka liittyvät tehokkaaseen avoimen lähdekoodin kehittämiseen. Ihan kaikki eivät ole sellaisia, jotka tulivat esiin juuri Linuxin parissa, mutta näemme kuinka Linux-maailmassa ne saavat aivan erityisen merkityksen. Mikäli olen oikeassa, ne auttavat käyttäjiä ymmärtämään täsmällisesti mikä tekee Linux-yhteisöstä hyvien ohjelmien runsaudensarven, ja kenties auttavat lukijaa kehittymään entistä tuottavammaksi.

Luku 3

Postin on kuljettava

Vuodesta 1993 olin johtamassa tekniikkaa pienessä Internet-palveluja tarjoavassa yrityksessä nimeltään Chester County InterLink (CCIL), Pennsylvaniassa. Olin mukana perustamassa CCIL:iä ja kirjoitin ilmoitustauluohjelmistomme, jonne useat käyttäjät saattoivat jättää viestejä. Työni mahdollisti minulle ympärisvuorokautisen pääsyn nettiin CCIL:n 56K moodemilinjan kautta – homma tosiaankin käytännössä vaati sitä.

Totuin käyttämään Internetin mahdollistamaa nopeaa sähköpostia. Huomasin, että on ärsyttävää vähän väliä kirjautua telnetillä postipalvelimelle tarkistamaan uusia posteja. Halusin postini suoraan kotona käyttämiini järjestelmiin, jolloin saisin ilmoituksen saapuneesta postista ja voisin käsitellä sitä kaikilla paikallisilla ohjelmillani.

Internetin alkuperäinen postinvälitysprotokolla SMTP (Simple Mail Transfer Protocol) ei soveltunut tähän tarkoitukseen, koska se toimi vain kun koneet olivat kytkettyinä toisiinsa kaiken aikaa. Oma koneeni ei ollut aina yhteydessä Internetiin eikä sillä ollut kiinteää IP-osoitetta. Tarvitsin ohjelman, joka hakee postini silloin kun ajoittainen nettiyhteyteni sattuu olemaan päällä. Tiesin, että tällaisia järjestelmiä oli olemassa, ja useat niistä käyttivät yksinkertaista POP-protokollaa (Post Office Protocol). Nykyään POP on laajasti tuettu useimmissa postinlukuohjelmissa, mutta silloin sitä ei ollut käyttämässäni lukijassa.

Tarvitsin POP3-asiakasohjelman. Löysinkin sellaisen Internetistä, tai oikeastaan löysin kolme tai neljä. Käytin erästä jonkin aikaa, mutta siitä puuttui ominaisuus jota tarvitsin: mahdollisuus muuttaa haettujen postien osoitteita, jotta vastaaminen posteihin olisi tapahtunut asianmukaisesti.

Ongelmani ilmeni, kun esimerkiksi ”joe@locke” lähetti minulle postin @locke-osoitteeseen. Kun hain postin ”snarkkiin” ja yritin vastata siihen, postiohjelmani hyväntahtoisesti yritti lähettää viestin ”joe@snark”ille, jota ei ole olemassa. Osoitteiden korjaaminen käsin muuttui nopeasti tuskaiseksi.

Tämä oli selvästi tehtävä, jonka tietokoneen pitäisi tehdä puolestani. Yksikään olemassa olevista POP-ohjelmista ei tätä kuitenkaan osannut. Tässä onkin ensimmäinen oppituntimme:

1. Kaikki hyvät ohjelmat syntyvät kehittäjän omasta tarpeesta

Kenties tämän pitäisi olla itsestään selvää (sananlaskunkin mukaan ”tarve on keksinnön äiti”), mutta liian usein ohjelman kehittäjät käyttävät aikaansa vääntämällä palkkansa eteen ohjelmia, joista he eivät välitä tai joita he eivät tarvitse. Näin ei ole Linux-maailmassa – tämä saattaa selittää, minkä takia Linux-yhteisön ohjelmien laatu on keskimäärin korkea.

Syöksyinkö siis samantien koodaamaan uutta ja hienoa POP3-asiakasohjelmaa kilpailemaan olemassa olevien kanssa? En. Katsoin tarkkaan POP-sovelluksia, joita jo oli käytettävissäni ja tutkin mikä niistä vastasi parhaiten tarpeitani. Sillä:

2. Hyvät ohjelmoijat osaavat ja tekevät. Parhaat tietävät mitä uusiokäyttävät.

En väitä, että olisin huippuohjelmoija, vaikka yritänkin jäljitellä sellaista. Eräs huippuohjelmoijan tärkeä ominaisuus on tuottava laiskuus. He tietävät, että huippuarvosanoja ei saa yritysten vaan tulosten perusteella ja että melkein aina on helpompaa aloittaa hyvästä osittaisesta ratkaisusta kuin täysin tyhjästä.

Esimerkiksi Linus Torvalds ei yrittänyt kirjoittaa Linuxia tyhjästä. Sen sijaan hän aloitti käyttämällä koodia ja ideoita Minixistä, joka on pieni Unixin kaltainen käyttöjärjestelmä PC-koneille. Lopulta kuitenkin kaikki Minixin koodi oli hävinnyt tai kirjoitettu täysin uudestaan – mutta sen hetken minkä se oli mukana, se tarjosi rakennustelineen alkuajan viritelmälle, josta oli tuleva Linux.

Samassa hengessä etsin olemassa olevista POP-sovelluksista parhaiten koodattua oman kehitystyöni lähtökohdaksi.

Koodin jakamisen perinne Unix-maailmassa on aina suosinut koodin uusiokäyttöä (siksi GNU projekti valitsi Unixin pohjakseen, vaikka Unixiin liittyikin tiettyjä varauksia). Linux-maailma on vienyt tämän perinteen melkeinpä teknisille rajoilleen saakka. Käytettävissä on teratavuittain avointa koodia. Ajan käyttäminen katsellen toisten tekemiä *melkein kyllin hyviä* ohjelmia antaakin parhaan tuloksen juuri Linux-maailmassa.

Näin kävi juuri minulle. Etsin vielä lisää ja yhteensä minulla oli nyt yhdeksän ehdokasta: fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail ja upop. Ensimmäisenä kävin toimeen Seung-Hong Oh'n "fetchpopin" kanssa. Lisäsin ohjelmaan otsaketietojen uudelleenkirjoitus -ominaisuuden ja tein joitakin muita parannuksia, jotka tekijä otti mukaan omaan 1.9-versioonsa.

Joitakin viikkoja myöhemmin tutustuin sattumalta Carl Harrisin "popclientin" koodiin ja tajusin, että minulla oli ongelma. Vaikka fetchpopissa oli joitakin hyviä alkuperäisiä ajatuksia (kuten tausta-ajo mahdollisuus), se pystyi käsittelemään vain POP3-protokollaa. Lisäksi se oli jokseenkin amatöörimäisesti koodattu (Seung-Hong oli tuolloin loistava mutta kokematon ohjelmoija, mikä myös näkyi koodissa). Carlin koodi oli parempaa, ammattimaisempaa ja tasaisempaa, mutta hänen ohjelmastaan puuttui muutamia tärkeitä ominaisuuksia, jotka fetchpopissa olivat (mukaan lukien ne jotka itse koodasin), ja joiden lisääminen olisi kinkkistä.

Pitää vai vaihtaa? Jos vaihtaisin, heittäisin pois koodini jonka olin jo tehnyt, mutta saisin paremman kehitysalustan.

Käytännön motiivi vaihtamiselle oli useamman protokollan tuki. POP3 on yleisin käytössä oleva postipalvelinten protokolla, mutta ei toki ainoa. Fetchpopista ja muista kilpailijoista puuttui tuki POP2-, RPOP- ja APOP-protokollille, ja minulla oli jo hämärä ajatus IMAPin (Internet Message Access Protocol, uusin ja tehokkain postipalvelinprotokolla) lisäämisestä ihan vain huvin vuoksi.

Mutta minulla oli enemmänkin teoreettinen syy olla sitä mieltä, että vaihtaminen olisi hyvä vaihtoehto. Tämän olin oppinut jo kauan ennen Linuxia.

3. "Valmistaudu heittämään jotakin pois. Heität kuitenkin" (Fred Brooks, The Mythical Man-Month*, Luku 11)*

Toisin sanoin: useinkaan ei oikeasti ymmärrä ongelmaa ennen kuin on ensimmäisen kerran toteuttanut sille ratkaisun. Toisella kertaa tietää ehkä riittävästi tehdäksesi sen oikein. Joten jos aikoo tehdä jotakin oikein, on oltava valmis tekemään se ainakin kerran uudestaan. [JB]

No, fetchpop oli vasta minun ensimmäinen yritykseni, joten vaihdoin.

Kun lähetin 25. kesäkuuta 1996 Carl Harrisille ensimmäiset koodini popclienttiin, tajusin että hän oli jo oikeastaan menettänyt mielenkiintonsa koko ohjelmaan. Ohjelmakoodi oli jo hiukan vanhahtavaa ja sisälsi useita pieniä vikoja. Minulla oli monia muutoksia tarjolla ja sovimme, että minä ottaisin koko ohjelman vastuulleni.

Aivan huomaamattani oli projekti paisunut. En enää hautonutkaan vähäisiä korjauksia johonkin olemassa olevaan POP-asiakasohjelmaan. Nyt otin vastuulleni koko popclientin ja päässäni pyöri paljon ideoita, joiden tiesin johtavan mittaviin muutoksiin.

Ohjelmistokulttuurissa, joka rohkaisee koodin jakamiseen tämä on projektin luonnollinen tapa uudistua. Noudatin tätä periaatetta:

4. Jos sinulla on oikeanlainen asenne, mielenkiintoiset ongelmat löytävät sinut.

Carl Harrisin asenne oli vieläkin tärkeämpi. Hän ymmärsi tämän:

5. Kun mielenkiintosi ohjelmaan loppuu, viimeinen tehtäväsi on antaa se pätevälle seuraajalle.

Emme edes keskustelleet asiasta Carlin kanssa, tiesimme että yhteinen tavoitteemme oli parhaan mahdollisen ratkaisun löytäminen. Ainut kysymys oli pystyisinkö osoittamaan, että en tyrisi koko hommaa. Kun onnistuin siinä, hän oli erittäin myötämielinen. Toivon että pystyn itse samaan kun tulee vuoroni luopua.

Luku 4

Käyttäjien merkittävä asema

Olin siis perinyt POP-asiakasohjelman kehityksen kontolleni ja, mikä tärkeää, perin myös sen käyttäjäkunnan. On hienoa kun on käyttäjiä - ei vain siksi, että he antavat merkin siitä, että ohjelmalle on tarve, että edes jotakin on tehty oikein. Oikein vaalittuina ja koulittuina heistä voi myöhemmin tulla kanssakehittäjiä.

Toinen Unix-perinteen vahvuus, jota Linux hyödyntää äärimmilleen, on se tosiseikka että suuri osa käyttäjistä on myös innokkaita hakkereita. Koska lähdekoodi on saatavilla, he voivat olla tehokkaita hakkereita. Tämä voi olla äärimmäisen hyödyllistä virheiden korjaukseen (debuggaukseen) kuluneen ajan lyhentämiseksi. Kun heitä rohkaisee, käyttäjäsi diagnosoivat ongelmat, ehdottavat korjauksia, sekä auttavat koodin kehittämisessä nopeammin kuin mitä saisit aikaan yksin ilman apua.

6. Käyttäjiesi kohteleminen kanssakehittäjinä on sinulle helpoin tie nopeaan ohjelmistokehitykseen sekä tehokkaaseen ohjelmavirheiden löytämiseen ja karsintaan.

Tämän vaikutuksen tehoa on helppo aliarvioida. Tosiasiassa hyvin monet meistä open-source maailmassa ratkaisevasti aliarvioivat kuinka hyvin se toimisi myös käyttäjien määrän kanssa merkittävästi kasvaessa, samalla kun järjestelmä monimutkaistuu. Sitten Linus Torvalds todisti toisin.

Itseasiassa, luulen että Linusin fiksuin ja kaikein merkittävin keksintö ei ollut niinkään itse Linux-ytimen rakentaminen, vaan Linuxin kehitysmallin luominen. Kun ilmaisin tämän näkemyksen kerran hänen läsnäollessaan, hän hymyili ja vaimeasti toisti jotain mitä hän on usein sanonut: "Olen periaatteessa todella laiska ihminen, joka tykkää ottaa kunnian muiden ihmisten tekemästä työstä."Laiska kuin kettu. Tai, kuten Robert Heinlein kuuluisasti kirjoitti eräästä hahmostaan, "liian laiska epäonnistumaan".

Jälkeenpäin tarkasteltuna, eräs ennakkotapaus Linuxin kehitystyylin menestykselle voidaan havaita GNU Emacsin Lisp-kirjaston ja -koodiarkistojen kehityksessä. Vastakohtana Emacsin C-ytimen sekä useimpien muiden GNU-työkalujen katedraali-tyyppiselle rakennustyylille, Lispin kehitys oli aaltomainen ja erittäin käyttäjävetoinen. Ideat prototyypitettiin usein kolmeen tai neljään kertaan ennen vakaan ja lopullisen muodon saavuttamista. Internetin mahdollistamat väljälti yhdistetyt yhteistyöt, Linuxin tapaan, olivat yleisiä.

Todellakin, oma kaikkein menestyksekkäin tuotokseni fetchmailin kehityksen jälkeen oli luultavasti Emacsin VC (versionhallinta)tila, jonka tein yhteistyössä sähköpostin välityksellä kolmen muun ihmisen kanssa. Heistä vain yhden (Richard Stallman, Emacsin luoja sekä Free Software Foundationin perustaja) olen tavannut tähän mennessä. Kyseinen ohjelma toimi

edustana SCCS:lle, RCS:lle sekä CVS:lle, joihin Emacs tämän ohjelman avulla tarjosi yksinkertaisen käyttöliittymän. Se kehittyi pienestä, karkeasta sccs.el-ohjelmasta jonka joku toinen oli kirjoittanut. VC:n kehitys edistyi koska- toisin kuin Emacs itse- Emacs Lisp-koodi voi edetä kehitys/testaus/julkaisu -vaiheiden läpi todella nopeasti.

Emacsin tarina ei ole ainutlaatuinen. On muitakin ohjelmistotuotteita kaksikerrosarkkitehtuurilla ja kaksiportaisella käyttäjäyhteisöllä jotka yhdistivät katedraali-tilan ja basaari-tilan työkalupakit. Eräänä esimerkkinä tällaisesta on MATLAB, joka on kaupallinen ohjelmisto mm. datan analysointia ja visualisointia varten. MATLABin käyttäjäkunnan mielestä kaikkein innovatiivisimmat ideat syntyvät avoimesti toisten käyttäjien taholta, kun laaja ja monipuolinen käyttäjäkunta voi itse keksiä uusia käyttätapoja ohjelmistolle.

Luku 5

Julkaise aikaisin, julkaise usein

Aikaisin ja usein tapahtuva julkaiseminen on oleellinen ja tärkeä osa Linuxin kehitystapaa. Useimmat meistä ohjelmistokehittäjistä elivät ennen siinä uskossa, että tällainen periaate voi toimia korkeintaan triviaalitason projekteissa, koska ohjelmien alkuversiot ovat lähes poikkeuksetta täynnä virheitä, eikä käyttäjien kärsivällisyyttä ole syytä koetella loputtomiin.

Tämä käsitys vahvisti entisestään sitoutumistani katedraalimalliseen ohjelmistokehitykseen. Jos päämääränä on mahdollisuuksien rajoissa säästää käyttäjät bugeilta, silloin olisi syytä julkaista vain kerran puolessa vuodessa (tai harvemmin), ja orjallisesti raataa ohjelmistovirheiden poistamiseksi julkaisujen välissä. Emacsin C-kielellä kirjoitettu ydin kehitettiin näin. Lisp-kirjaston kehityksessä ei näin toimittu, lähinnä koska olemassa oli aktiivisia Lisp-arkistoja, joita FSF ei hallinnut. Näistä arkistoista pystyi löytämään uusia ja kehityksen alla olevia versioita, joiden julkaisu ei ollut riippuvainen Emacsin julkaisuaikataulusta.

Näistä merkittävin, Ohion osavaltion Emacs Lisp -arkisto, enteili nykypäivän suurten Linux-arkistojen henkeä ja niiden monia ominaisuuksia. Vain harva meistä todella kuitenkaan ajatteli sen vakavammin mitä olimme tekemässä, tai mitä arkisto pelkällä olemassaolollaan vihjasi FSF:n katedraalityylisen kehitysmallin ongelmista. Minä yritin tosissani vuoden 1992 tienoilla saada valtaosan Ohion koodista muodollisesti yhdistettyä viralliseen Emacs Lisp-kirjastoon. Siitä seurasi poliittisia ongelmia, ja yritykseni epäonnistui lähes kaikessa.

Vain vuotta myöhemmin Linux sai laajaa näkyvyyttä, ja oli selvää, että jotakin uudenlaista ja paljon terveempää oli tapahtumassa. Linusin avoin kehitystapa oli täysi vastakohta katedraalin rakentamiselle. Linuxin Internet-arkistot kehittyivät harppauksin, ja useita jakeluversioita laskettiin liikkeelle. Kaikkea tätä vei eteenpäin ennenkuulumattoman ripeä ydinjärjestelmän julkaisutaajuus.

Linus kohteli käyttäjiään ohjelmistokehittäjinä tehokkaimmalla mahdollisella tavalla:

7. Julkaise aikaisin. Julkaise usein. Kuuntele mitä käyttäjilläsi on sanottavaa.

Linusin perusoivallus ei niinkään ollut vain nopeissa käyttäjien palautteeseen perustuneissa suunnanmuutoksissa versioiden välillä, sillä jotain vastaavaa oli tehty Unix-maailmassa perinteisesti jo kauan, vaan pikemminkin jo olemassaolevan prosessin intensiteetin lisäyksessä vastaamaan hänen kehittämänsä järjestelmän monimutkaisuuden tarpeita. Pioneerikaudella vuoden 1991 paikkeilla ei ollut mitenkään ainutkertaista häneltä julkaista uusi ydin useita kertoja päivässä! Koska hän keräsi ympärilleen suuren joukon muita kehittäjiä ja käytti Internetin mahdollisuuksia yhteistyöhön kovemmin kuin kukaan muu, tämä toimi.

Miten se toimi? Olisiko saavutus minun toistettavissani, vai oliko kaikki seurausta jostakin

Linus Torvaldsille ominaisesta ainutlaatuisesta nerokkuudesta?

En ollut sitä mieltä. Taatusti Linus on pahuksen loistava hakkeri. Kuinka moni meistä pystyy rakentamaan kokonaisen tehdasvalmiin käyttöjärjestelmän ytimen tyhjästä? Linux ei kuitenkaan edustanut mitään vaikuttavaa konseptitason loikkaa eteenpäin. Linus ei ole (ainakaan vielä) sellainen näkemyksellinen suunnittelijanero, kuten Richard Stallman tai James Gosling (NeWS ja Java). Sitä vastoin Linus näyttää, ainakin minusta, olevan insinööritaidon ja soveltamisen mestari. Hänellä tuntuu olevan jonkinlainen kuudes aisti ohjelmointivirheiden ja sudenkuoppien välttämiseen, minkä lisäksi hänellä on siunattu kyky löytää pienimmän vastuksen reitti pisteestä A pisteeseen B. Todellakin, koko Linuxin kehitys huokuu tätä laatua ja heijastaa Linusin pohjimmiltaan perinteistä, yksinkertaisuuteen pyrkivää suunnittelutapaa.

Joten, jos pikajulkaisut ja Internetin nosteen täysimääräinen käyttö eivät olleet vahinkoja, vaan kaikkein keskeisintä sisältöä siinä tavassa, jolla Linus insinööritaidon nerokkuudessaan paneutui minimaalisen vastuksen polkuunsa, mitä hän sitten venyttikään äärimmilleen? Kuinka hän sai huippukierrokset koneestaan, ja mitä hän sillä sai aikaan?

Näin aseteltuna kysymys sisältää vastauksen. Linus innosti ja palkitsi hakkeri-käyttäjiensä yhteisön jäseniä jatkuvasti. Palkitsevat tulevaisuudennäkymät, osallistuminen egoa hivelevään toimintaan ja onnistuneesta työstä melkein päivittäin saatu kiitos kantoivat pitkälle.

Linus oli suoraan tähtäämässä henkilötyötuntien, joita käytettiin debugaamiseen ja kehitykseen, maksimointiin jopa siten, että se saattoi tuottaa epävakaata koodia ja käyttäjien stressaantumista, jos joku vakava ohjelmointivirhe osoittautui erityisen hankalaksi. Linus käyttäytyi kuten olisi uskonut johonkin tällaiseen:

8. Kun käytössä on riittävän suuri määrää beta-testaajia ja ohjelmistokehittäjiä, miltei kaikki ongelmat jäsentyvät nopeasti ja ratkaisu tulee ilmeiseksi jollekulle.

Tai hieman vähemmän muodollisesti sanottuna, "Kunhan tarpeellinen määrä silmiä valvoo, ei yksikään virhe jää kääntämättä". Minä kutsun tätä "Linusin laiksi".

Alkuperäinen muotoiluni mukaan mikä tahansa ongelma on "läpinäkyvä jollekin". Linusin mielestä henkilö joka ymmärtää ja korjaa ongelman ei välttämättä tai useinkaan ole se kuka ensiksi paikallistaa ongelman. "Joku löytää ongelman."hän sanoo, "ja joku toinen ymmärtää sen. Niiden löytäminen on suurempi haaste."Korjaaminen on tärkeää ja tulemme seuraavassa kappaleessa näkemään syyn tähän, kun tutkimme debuggausta tarkemmin. Tärkeä havainto on se, että molemmilla prosessin osilla, löytämisellä ja korjaamisella, on taipumus tapahtua nopeasti.

Linusin laissa, luulen, on perustavanlaatuinen ero katedraalinrakentamistyylin ja basaarityylin välillä. Kathedraalin ohjelmoija näkee bugit ja kehitysongelmat hankalina, salakavalina ja ymmärrystä vaativina ilmiöinä. Asialle omistautuneelta ryhmältä kestää kuukausia päästä varmuuteen siitä, että kaikki virheet on poistettu. Tämä johtaa pitkiin julkaisuvälehin ja väistämättömään pettymykseen, kun pitkään odotettu julkaisu ei olekaan täydellinen.

Basaarinäkemyksen mukaan toisaalta oletetaan bugit yleisesti pinnallisiksi ongelmiksi tai vähintään ne muuttuvat hyvin pian pinnallisiksi kun ne on löydetty kun tuhannet innokkaat kehittäjät pauhaavat jokaisen uuden julkaisun kanssa. Vastaavasti usein julkaisemmalla saa enemmän korjauksia ja suotuisana sivuvaikutuksena on vähemmän menetettävää, jos joitakin suoranaisia möhläyksiä tuleekin julkaistua.

Siinäpä se. Siinä onkin kylliksi. Jos "Linusin Laki"on väärä, silloin mikä tahansa monimutkainen järjestelmä, kuten Linuxin ydin jota ovat rakentaneet monet hakkerit, tulee jossakin pisteessä romahtamaan löytymättömien ja vakavien virheiden yhteisvaikutusten painosta.

Jos tämä pitää paikkansa, on toisaalta huomattava että se silti riittää selittämään Linuxin suhteellisen vähäisen virhemäärän ja kuukausiin tai jopa vuosiin venyviä yhtäjaksoisia käynnissäoloaikoja Linux-järjestelmille.

Ehkä sen ei edes olisi pitänyt olla yllätys. Jo vuosia sitten sosiologit havaitsivat Delphin efektinä tunnetun ilmiön, jonka mukaan suuren yhtä asiantuntevista henkilöistä kootun joukon yhteinen mielipide on selvästi luotettavampi kuin yksittäisen tämän joukon henkilön. Vaikuttaa siltä että Linus on näyttänyt että tämä ilmiö toimii myös käyttöjärjestelmän virheiden korjaamisessa. Eli että Delphin efekti mahdollistaa niinkin monimutkaisen järjestelmän kuin käyttöjärjestelmän ytimen kehityksen hallinnan.

Yksi merkittävä ominaisuus (yhdessä Delphi-efektin kanssa) Linuxin tapauksessa on se tosiasia, että osallistujat mihin tahansa projektiin ovat hakeutuneet projektiin itsenäisesti. Osallistumiset projekteihin eivät tule satunnaisilta henkilöiltä, vaan henkilöiltä jotka ovat kiinnostuneita ohjelman käyttämisestä, joita kiinnostaa se kuinka ohjelma toimii, jotka yrittävät löytää ratkaisuja ongelmiinsa joihin he törmäävät ja joilla on kykyä tuottaa näihin ongelmiin toimivia ratkaisuja. Jokaiselta, joka täyttää kaikki esitellyt ehdot, on todennäköisesti jotakin hyödyllistä annettavaa projektille.

Linusin laki voidaan muotoilla myös seuraavasti: *Debuggaus voidaan rinnakkaistaa* Vaikka debuggaus vaatii debuggaajia kommunikoimaan jonkun debuggausta koordinoivan kehittäjkän kanssa, se ei ei kuitenkaan vaadi merkittävää koordinointia debuggaajien kesken. Täten se ei kärsi samasta eksponentiaaliesti lisääntyvästä monimutkaisuudesta ja hallintokulujen kasvusta, joka tekee kehittäjien lisäämisen ongelmalliseksi.

Useiden henkilöiden samanaikaisesti tekemän debuggauksen aiheuttama hukkatyö ei näytä olevan merkittävä ongelma Linux-maailmassa. "Julkaise aikaisin, julkaise usein-kehitysmalli onkin omiaan minimoimaan kyseiset päällekkäisyydet tarjoamalla päivitykset löydettyihin ongelmiin nopeasti.

Brooks (The Mythical Man-Monthin kirjoittaja) teki jopa toisen käden huomion tähän liittyen: "Laajasti käytetyn ohjelman ylläpitokustannukset ovat tyypillisesti 40 prosenttia tai enemmän ohjelman kehityskustannuksista. Käyttäjien lukumäärä vaikuttaa yllättävän paljon ylläpitokustannuksiin. Useammat käyttäjät löytävät enemmän bugeja."[korostus lisätty]

Useammat käyttäjät löytävät enemmän bugeja, koska lisättäessä käyttäjiä lisätään samalla erilaisia tapoja koetella ohjelmaa. Tämä efekti vahvistuu silloin kun käyttäjät ovat apukehittäjiä. Jokainen lähestyy bugiongelman karakterisointia hieman erilaisten aistien ja analyyttisten työkalujen kanssa, hieman eri kulmasta siis. Delphi-efekti näyttää toimivan täsmälleen juuri tämän vaihtelun takia. Erityisessä debuggaustapauksessa tällä vaihtelulla on usein tapana kaksinkertaistaa panostus.

Jotenka betatestaajien lisääminen ei välttämättä vähennä tämänhetkisen *hankalimman* bugin monimutkaisuutta kehittäjän näkökulmasta, mutta lisää todennäköisyyttä sille, että jonkun työkalupaketti täsmää ongelmaan siten, että bugi on hänelle ratkaistavissa.

Linus asettaa panoksensa kuten muutkin. Vakavien ohjelmavirheiden varalta Linux ytimen versionumerot mahdollistavat, että käyttäjät voivat valita käyttävätkö vakaaksi todettua/nimitettyä versiota, vai ottavatko riskin ja käyttävät viimeisintä mahdollista versiota saadakseen uusimmat ominaisuudet käyttöönsä. Kaikki Linux-kehittäjät eivät ole ottaneet samaa taktiikkaa systemaattisesti käyttöönsä, vaikka heidän ehkä pitäisi. Sillä pelkästään se tosiasia, että on olemassa toinenkin vaihtoehto, tekee molemmista kiinnostavampia.

Luku 6

Kuinka monta silmäparia tarvitaan kesyttämään monimutkaisuus

Basaarityylin mahdollistama nopeampi ohjelmavirheiden korjaus ja koodin kehittyminen ansaitsee oman tarkastelunsa, mutta aivan toista on todella ymmärtää mikrotasolla, kehittäjien ja testaajien päivittäisessä työssä, miten ja miksi niin käy. Tässä luvussa (jonka kirjoitin kolme vuotta alkuperäisen paperin jälkeen hyödyntämällä paperin lukeneiden kehittäjien itsetutkiskelussaan saamia oivalluksia) keskitytään perinpohjaisesti käytännön mekanismeihin. Vähemmän tekniset lukijat voivat huoletta hypätä seuraavaan lukuun.

Yksi ymmärryksen avaimista on tajuta tarkalleen miksi lähdekoodista tietämättömien käyttäjien bugiraportit eivät yleensä ole kovin hyödyllisiä. Lähdekoodista tietämättömillä käyttäjillä on tapana raportoida vain pinnallisia oireita; he ottavat ympäristönsä itsestään selvyytenä, joten he (a) jättävät antamatta kriittistä taustatietoa ja (b) harvoin sisällyttävät mukaan luotettavia ohjeita virheen toistamiseksi.

Taustalla vaikuttava ongelma on epäsuhde testaajien ja kehittäjien ohjelmaa koskevissa ajatusmalleissa: testaaja katsoo ohjelmaa ulkoa sisäänpäin ja kehittäjä sisältä ulospäin. Suljetun lähdekoodin kehityksessä molemmat ovat jumiutuneet rooleihinsa, heillä on taipumus puhua toistensa ohi ja turhautua toisiinsa tavattomasti.

Avoimen lähdekoodin kehitysmalli rikkoo nämä rajat. Se tekee yhteisen näkemyksen luomisen testaajien ja kehittäjien välille helpoksi, sillä se voidaan perustaa ohjelman lähdekoodiin ja ongelmasta voidaan keskustella tehokkaasti lähdekoodi huomioiden. Ohjelmoijan näkökulmasta virheraportti, joka perustuu lähdekoodiin on huomattavasti antoisampi kuin pelkästään käyttäjälle näkyvään toiminnallisuuteen perustuva virheraportti.

Useimmat ohjelmavirheet on helppo ratkaista, jos virheraportissa on edes suuntaa antavat tiedot siitä, missä kohtaa lähdekoodia ongelma esiintyy. Kun joku beta-testaajista voi osoittaa "rivillä nnn on rajaongelma"tai kertoa edes, että "olosuhteissa X, Y ja Z tämä muuttuja vuotaa yli"nopea katsaus huomautettuun koodiin riittää useimmiten yksilöimään virheen ja tekemään korjauksen.

Näin ollen sekä kehittäjien että testaajien pääsy lähdekoodiin mahdollistaa toimivan kommunikoinnin ja yhteistyön osapuolten välillä. Tämä puolestaan merkitsee sitä, että varsinaisten kehittäjien aika tulee käytettyä tehokkaasti vaikka osapuolia olisi montakin.

Toinen kehittäjän aikaa kuluttava tyypillinen piirre avoimen lähdekoodin menetelmässä on avoimen lähdekoodin projektien tavanomainen kommunikointirakenne. Yllä käytin termiä

"varsinainen kehittäjä"; tämä heijastaa sitä eroa projektin kehittäjien ydinryhmän (tyypillisesti melko pieni, usein yhdestä kolmeen henkilöä) ja projektin ympärillä olevan betatestaajien ja muiden työpanoksensa antavien ryhmän (koostuu yleensä sadoista henkilöistä) välillä.

Keskeinen ongelma, jonka perinteinen ohjelmistoa kehittävä organisaatio kohtaa on Brooken Laki: *Ohjelmoijien lisääminen myöhästyneeseen projektiin myöhästyttää sitä entisestään.* Yleistäen Brooken Laki ennustaa, että projektin kompleksisuus ja yhteydenpidon kustannukset kasvavat suhteessa kehittäjien lukumäärän neliöön, kun samanaikaisesti työsaavutus kasvaa vain lineaarisesti.

Brooken Laki perustuu havaintoon, että bugeilla on voimakas taipumus kasaantua eri ihmisten kirjoittamien koodien rajapinnoille ja projektin yhteydenpidon/koordinaation määrän on tapana nousta ihmisten välisten rajapintojen määrän nousun myötä. Täten ongelma skaalautuu kehittäjien välisten yhteydenpitoreittien määrän mukaan ja skaalaus tapahtuu kehittäjien määrän neliössä (tarkemmin, kaavan N*(N - 1)/2 mukaan, missä N kehittäjien määrä).

Brooken Lain analysointi (ja sen tuloksena pelko suuria kehittäjäryhmiä kohtaan) perustuu piilossa olevaan oletukseen: projektin yhteydenpitorakenteen on oltava täydellinen graafi, jossa jokainen puhuu jokaisen kanssa. Mutta avoimen lähdekoodin projekteissa ydinryhmän ulkopuoliset kehittäjät työskentelevät käytännössä erillisten rinnakkaisten alitehtävien parissa ja kommunikoivat keskenään hyvin vähän. Koodin muutokset ja bugiraportit virtaavat ydinryhmän kautta, ja vain ydinryhmän osalta aiheutuu kustannuksia Brooken Lain kuvaamalla tavalla.

On vielä lisää syitä sille, miksi lähdekooditason virheraportoinnilla on tapana olla hyvin tehokasta. Syyt keskittyvät sen tosiasian ympärille, että yksittäisellä virheella voi olla useita oireita, jotka ilmaantuvat eri tavoilla riippuen käyttäjien toimintatavoista ja ympäristöstä. Kuvatuilla virheillä on tapana olla monimutkaisia ja salakavalia (kuten dynaamiseen muistinhallintaan liittyvät virheet ja ennakoimattomat keskeytysikkunat) ja siten virhetilanne on hankalaa ja työlästä toistaa haluttaessa tai löytää tilastollisesti. Kaiken kukkuraksi salakavalat bugit ovat pitkäaikaisia ongelmia ohjelmissa.

Testaaja, joka lähettää alustavan lähdekooditasoisen kuvauksen monioireisesta bugista (kuten "Näyttää siltä, että signaalinkäsittelyssä on ikkuna lähellä riviä 1250"tai "Missä nollaat sen puskurin?") voi antaa muutoin liian lähellä koodia olevalle kehittäjälle ratkaisevan vihjeen puolesta tusinasta erillisestä oireesta. Tällaisissa tapauksissa voi olla hankalaa tai jopa mahdotonta tietää mikä selvästi havaittava virheellinen käyttäytyminen oli minkäkin bugin tekemä. Usein julkaistaessa sen tietäminen onkin tarpeetonta. Muut mukana työskentelevät ottavat todennäköisesti nopeasti selvää siitä onko heidän buginsa korjattu vai ei. Monissa tapauksissa lähdekooditason virheraportti johtaa virheen poistumiseen, ilman että sitä varten on koskaan tehty mitään tiettyä korjausta.

Monimutkaisilla ja monioireisilla virheillä on myös usein monia jäljityspolkuja pinnallisista oireista itse todelliseen virheeseen. Mitä reittiä myöten kehittäjä tai testaaja voi virhejahdissaan kulkea, voi riippua henkilön käyttöympäristön hienon hienoista yksityiskohdista. Polku voi myös ajan mukaan vaihtua täysin ennakoimattomasti. Käytännössä kukin kehittäjä ja testaaja kokeilee puolisatunnaista joukkoa ohjelman eri tiloja oireiden syitä etsiessään. Mitä ovelampi ja monimutkaisempi virhe on, sitä epätodennäköisemmin tällä keinolla tulee valituksi oikea joukko.

Yksinkertaisten ja helposti toistettavien virheiden osalta paino on enemmän yhdyssanan

osassa "puoli"kuin "satunnaisesti". Virheidenkorjaustaito ja tuttuus koodin ja sen arkkitehtuurin kanssa merkitsevät paljon. Monimutkaisten virheiden osalta paino on kuitenkin osassa "satunnaisesti". Tällaisessa tapauksessa monta ihmistä ajelemassa eri reittejä on paljon tehokkaampaa kuin muutama ihminen ajamassa eri reittejä yksi toisensa perästä - jopa silloin, kun jälkimmäisten osaaminen on keskimäärin muita korkeampaa tasoa.

Aikaansaatu vaikutus on paljon suurempi, jos jäljityspolun seurantavaikeus eri pintatason oireista ohjelmavirheeseen vaihtelee tavalla, jota ei voi ennustaa ohjelman käyttäytymisestä itsestään. Satunnainen ohjelmistokehittäjä, joka haarukoi näitä polkuja perättäisissä sarjoissa, valinnee ensi yrittämällä yhtä todennäköisesti vaikean kuin helpon jäljityspolun. Toisaalta, oletetaan että monet yrittävät löytää rinnakkaispolkuja samalla, kun ohjelmistosta julkaistaan useita pikaversioita. Silloin olisi luultavaa, että joku heistä löytääkin sen helpoimman polun välittömästi, ja onnistuu liiskaamaan bugin paljon lyhyemmässä ajassa. Projektin ylläpitäjä havaitsee, että uuden ohjelmaversion toimituksen myötä muut projektissa mukana olevat ihmiset, jotka haarukoivat samaa ohjelmavirhettä, älyävät lopettaa haaskaamatta liikaa aikaa vaikeampiin polkuihin.

Luku 7

Milloin ruusu ei ole ruusu

Kun olin aikani tutkinut Linusin toimintatapaa ja muodostanut teorian siitä, miksi se oli menestyksellinen, päätin kokeilla tätä teoriaa uudessa, tosin paljon vähemmän monimutkaisessa ja kunnianhimoisessa projektissa.

Ensimmäinen asia, jonka tein oli kuitenkin pop-asiakasohjelman uudelleenjärjestäminen ja yksinkertaistaminen. Carl Harrisin toteutus oli erittäin hyvä, mutta vaikutti tarpeettoman monimutkaiselta, mikä on yleistä monille C-ohjelmoijille. Hän käsitteli koodia keskipisteenä ja tietorakenteita koodin tukena. Tällöin koodi oli kaunista, mutta tietorakenteet oli suunniteltu vain tätä koodia varten, ja lopputulos oli melko ruma, ainakin tämän kaiken kokeneen LISP-hakkerin korkeiden laatuvaatimusten mukaan.

Minulla oli kuitenkin uudelleen kirjoittamiselle muukin tarkoitus kuin koodin ja tietorakenteiden parantaminen. Tarkoitus oli kehittää siitä jotakin, mitä ymmärtäisin täysin. Ei ole hauskaa vastata virheiden korjaamisesta ohjelmassa, jota ei ymmärrä.

Arvioilta ensimmäisen kuukauden ajan seurasin yksinkertaisesti Carlin perussuunnitelmaa. Ensimmäisenä vakavasti otettavana muutoksena lisäsin IMAP-tuen. Tein tämän uudelleenorganisoimalla protokollakoneita geneeriseksi laitteeksi ja kolmeksi menetelmätauluksi (POP2, POP3 ja IMAP). Tämä ja edelliset muutokset kuvaavat erästä yleistä periaatetta, joka ohjelmoijien on hyvä pitää mielessä erityisesti C:n kaltaisissa kielissä, jotka eivät luonnostaan tee dynaamista tyypitystä:

9. Viisaat tietorakenteet ja tyhmä koodi toimii paljon paremmin kuin toisinpäin.

Brooks, Kappale 9: *Näytä minulle vuokaaviosi ja piilota taulukkosi ja olen edelleen ymmälläni. Näytä taulukkosi niin en yleensä edes tarvitse vuokaaviotasi: se on ilmiselvä.* Kolmekymmentä vuotta terminologista ja kulttuurista muutosta ei ole muuttanut tätä periaatetta miksikään.

Tässä vaiheessa (syyskuun alussa 1996, noin kuusi viikkoa aloittamisesta) aloin ajatella, että nimen vaihtaminen saattaisi sittenkin olla paikallaan, sillä ohjelma ei enää ollut pelkkä POP-asiakasohjelma. Epäröin tässä kuitenkin, sillä ohjelmassa ei ollut vielä mitään nerokasta ja uutta. Minun POP-asiakasohjelmani täytyi vielä kehittää oma identiteettinsä.

Tämä muuttui radikaalisti, kun pop-asiakasohjelma oppi lähettämään haetun postin edelleen SMTP porttiin. Palaan tähän kohta. Mutta ensin: sanoin aikaisemmin, että olin päättänyt käyttää tätä projektia testatakseni teaoriaani siitä, mitä Linus Torvalds oli tehnyt oikein. Voit kysyä kuinka tein sen. Näillä tavoilla:

- Julkaisin aikaisin ja usein, yleensä useammin kuin kymmenen päivän välein ja intensiivisen kehityksen aikana päivittäin.
- Kasvatin beta-listaani lisäämällä siihen jokaisen, joka otti minuun yhteyttä Fetchmailiin liittyvissä kysymyksissä.
- Lähetin vuolaita viestejä beta-listalle aina kun julkaisin rohkaisten ihmisiä osallistumaan.
- Kuuntelin beta-testaajiani, pidin äänestyksiä suunnittelupäätöksistä ja imartelin heitä aina kun he lähettivät korjauksia ja palautetta.

Näiden pienten juttujen hyöty näkyi välittömästi. Projektin alusta lähtien sain hyvin laadittuja virheilmoituksia - usein korjaukset liitteinään, joiden edestä moni kehittäjä olisi valmis murhaamaan. Sain tarkoin harkittua kritiikkiä, ihailijapostia, fiksuja ehdotuksia uusista ominaisuuksista. Mistä seuraa:

10. Jos kohtelet beta-testaajia kuin he olisivat tärkein voimavarasi, he reagoivat muuttumalla sellaiseksi.

Yksi mielenkiintoinen Fetchmailin menestyksen mittari on pelkästään betatestaajien listan koko, Fetchmail-kavereiden määrä. Tämän esseen viimeisimmän korjatun version aikoihin (marraskuu 2000) listalla on 287 jäsentä, ja lisää tulee kahdesta kolmeen viikossa.

Kun päivitin uuteen versioon toukokuun lopussa 1997 huomasin, että lista oli oikeastaan alkanut menettää jäseniään huippulukemista, joka oli lähellä kolmeasataa. Syy oli mielenkiintoinen: useat ihmiset pyysivät minua poistamaan itsensä listalta, koska heidän mielestään Fetchmail toimi jo niin hyvin, ettei heillä ollut enää tarvetta nähdä listan viestiliikennettä! Ehkäpä tämä onkin tavallista valmistumassa olevan basaariprojektin elinkaaressa.

Luku 8

Popclientista Fetchmailiksi

Varsinainen käännekohta projektissa oli kun Harry Hochheiser lähetti minulle luonnoskoodinsa sähköpostin edelleenohjaamiseen asiakaskoneen SMTP-porttiin. Ymmärsin lähes välittömästi että tämän ominaisuuden luotettava toteuttaminen tekisi kaikki muut sähköpostin toimitustavat käytännössä tarpeettomiksi.

Monien viikkojen ajan olen kehittänyt fetchmailia jokseenkin pienillä askelilla, mutta samaan aikaan minulla on ollut tunne siitä, että käyttäjärajapinta on toimiva, mutta köpö - epäelegentti ja liian monia pikkuoptioita roikkumassa mikä missäkin. Toimintanappi haettujen mailien pakkaamiseksi mailbox-tiedostoksi tai standardi ulostuloksi erityisesti häiritsi minua, mutten hahmottanut miksi.

(Jos et välitä Internet-postin teknisestä puolesta, voit huoletta hypätä kahden seuraavan kappaleen yli.)

Kun ajattelin SMTP-edelleenohjausta, näin että popclient oli yrittänyt tehdä liian montaa asiaa. Se oli suunniteltu olemaan sekä sähköpostin kuljetusohjelma (Mail Transport Agent, MTA) ja paikallisjakeluohjelma (local delivery agent, MDA). SMTP-edelleenohjauksella se voisi päästä eroon MDA-toimesta ja olla puhdas MTA, antaen postia muille ohjelmille paikallista jakelua varten sendmailin tapaan.

Miksi taistella monimutkaisten sähköpostiohjelmien asetusten kanssa, tai yrittäen ottaa käyttöön lukitse-ja-liitä toimintoa sähköpostissa, kun portti 25 on lähes varmasti tuettuna joka alustalla TPC/IP:n kanssa? Erityisesti kun tämä tarkoittaa että noudettu posti näyttää taatusti samalta, kuin tavallinen lähettäjä-lähtöinen SMTP posti, joka juuri se mitä tahdoimmekin.

(Takaisin korkeammalle tasolle...)

Jopa jos et lukenut tai ymmärtänyt edeltävää teknistä jargonia, tässä on useita tärkeitä opetuksia. Ensinnäkin, tämä SMTP-edelleenohjauskonsepti oli suurin yksittäinen palkkio jonka sain yrittämällä tietoisesti jäljitellä Linusin toimintatapoja. Eräs käyttäjä antoi minulle tämän loistavan idean – minun tarvitsi vain ymmärtää seuraukset.

11. Seuraavaksi paras asia omien hyvien ideoiden omaamisen jälkeen on tunnistaa käyttäjiesi hyvät ideat. Joskus jälkimmäinen on parempi.

Kiinnostavaa kyllä, sitä huomaa hyvin pian, että kunhan vain rehellisesti itseään vähätellen kertoo olevansa velkaa kaikesta muille, kaikki yleisesti ottaen kohtelevat kuin olisi tehnyt kaiken ominpäin, ja onkin vain mukavan vaatimaton omasta neroudestaan puhuttaessa. Kuinka hyvin se onkaan toiminut Linusin kohdalla!

Pitäessäni puhettani ensimmäisessä Perl-konferenssissa elokuussa 1997 hakkeri-guru vailla vertaa Larry Wall oli eturivissä. Ehdittyäni puheen viimeiseen virkkeeseen hän päästi suustaan uskonnollisen herätysliikkeen tyyliin "Anna tulla, anna tulla, veli!". Koko yleisö nauroi, koska he tiesivät että sama jekku oli toiminut Pearlin keksijänkin kohdalla.

Pidettyäni projektia päällä vasta muutaman hätäisen viikon samaisessa hengessä aloin saada kehuja, en vain käyttäjiltäni vaan myös muilta ihmisiltä, joille sana oli kiirinyt. Arkistoin osan sähköposteista, ja aion ottaa ne uudelleen esille joskus taas, kun päähäni iskee epäilys onko elämäni ollut elämisen arvoista :-).

Mutta tässä on kaksi muuta perustavanlaatuista, ei-poliittista opetusta jotka ovat samoja kaikille suunnittelutavoille.

12. Usein kaikkein iskevimmät ja kekseliäimmät ratkaisut tulevat ymmärryksestä, että käsityksesi ongelmasta oli väärä.

Olin yrittänyt ratkaista väärää ongelmaa jatkamalla popclientin kehittämistä yhdistettynä MTA/MDA:na kaikenlaisilla vängillä paikallisilla toimitustavoilla. Fetchmailin suunnittelu täytyi miettiä uudelleen läpikotaisin puhtaana MTA:na, osana normaalia SMTP:tä puhuvaa Internet-postin polkua.

Kun törmää kehityksessä seinään — kun huomaa vaikeaksi ajatella seuraavaa päivitystä pidemmälle — on usein aika kysyä itseltään; ei sitä onko saavuttanut oikean vastauksen, vaan kysyykö oikeita kysymyksiä. Ehkäpä ongelma pitääkin muotoilla uudestaan.

No, olin uudelleenmuotoillut ongelmani. Selvästikin oikea asia tehtäväksi oli (1) virittää SMTP-edelleenohjaustuki yleiseen ajuriin, (2) tehdä siitä oletustila, ja (3) lopulta heittää pois kaikki muut toimitustilat, etenkin ”toimita tiedostoon” ja ”toimita standarditulosteeseen (STD-OUT)”-vaihtoehdot.

Epäröin kohdan kolme kanssa jonkin aikaa, pelästen hermostuttavani pitkäaikaisia popclientin käyttäjiä jotka olivat riippuvaisia vaihtoehtoisista toimitustavoista. Teoriassa he voisivat välittömästi vaihtaa .forward-tiedostoihin tai niiden ei-sendmail-tyylisiin vastineisiin tuottaakseen saman tuloksen. Käytännössä siirtymisvaihe olisi voinut olla sotkuinen.

Mutta kun tein sen, edut osoittautuivat huikeiksi. Purkkaisimmat osat ajurikoodista katosivat. Asetusten muuttamisesta tuli radikaalisesti helpompaa – ei enää ryömimistä ympäriinsä etsien järjestelmän MDA:ta ja käyttäjän postilaatikkoa, ei enää huolia siitä, tukeeko allaoleva käyttöjärjestelmä tiedostojen lukitsemista.

Lisäksi ainoa tapa hävittää postia katosi. Jos asetit jakelun tiedostoon ja levy täyttyi, postiasi hävisi. Tätä ei voi tapahtua SMTP-edelleenohjauksen kanssa koska SMTP-ohjelmasi ei palauta OK:ta jos viestiä ei voida jakaa tai vähintään varastoida myöhempää jakelua varten.

Myös suorituskyky parani (joskaan ei niin merkittävästi että sitä huomaisi yksittäisessä ajossa). Toinen ei-merkityksetön seikka muutoksesta oli että ohjesivusta tuli paljon yksinkertaisempi.

Myöhemmin minun täytyi lisätä toimittaminen käyttäjän määrittelemällä paikallisella MDA-taustaohjelmistolla jotta eräiden epämääräisten dynaamiseen SLIP:iin liittyvien tilanteiden käsittely olisi mahdollista. Mutta löysin paljon yksinkertaisemman tavan tehdä sen.

Opetus? Älä epäröi heittää pois vanhentuneita ominaisuuksia kun voit tehdä sen menettämättä tehokkuutta. Antoine de Saint-Exupéry (joka oli lentäjä ja lentokonesuunnittelija silloin, kun ei kirjoittanut klassisia lastenkirjoja) sanoi:

13. ”Täydellisyys (suunnittelussa) saavutetaan ei silloin kun ei ole mitään lisättävää, mutta ennemminkin silloin kun ei ole mitään pois otettavaa.”

Kun koodisi tulee sekä paremmaksi että yksinkertaisemmaksi, tiedät että se on oikein. Ja tässä prosessissa fetchmailin suunnittelu sai oman identiteetin, erillään antiikkisesta popclientista.

Oli aika nimenvaihdokselle. Uusi suunnittelu näytti paljon enemmän sendmailin kaksoiskappaleelta kuin vanha popclient; molemmat ovat MTA:ita, mutta siinä missä sendmail työntää ja toimittaa, uusi popclient vetää ja toimittaa. Joten, kaksi kuukautta lähtöviivalta uudelleennimesin sen fetchmailiksi.

On olemassa yleisempikin opetus siitä, miten SMTP:stä tuli fetchmail. Ei se ole vain virheiden etsintä, joka on rinnakkaistettavissa; kehitys ja (ehkäpä yllättävässäkin määrin) suunnitteluavaruuden tutkimus on myös. Kun kehityksesi malli on nopeasti toistuva, kehitys ja parannukset voivat tulla virheiden etsinnän erityistapauksiksi — ‘laiminlyöntivirheiden’ korjaus alkuperäisessä toiminnallisuudessa tai koko ohjelmiston määrittelyssä.

Jopa korkeammilla suunnittelutasoilla, voi olla hyödyllistä, että on useita kanssakehittäjiä käymässä satunnaisesti läpi suunnitteluavaruutta tuotteen lähettyvillä. Ajattele miten vesinoro löytää kanavan, tai paremminkin miten muurahaiset löytävät ruokaa: tutkimus etenkin hajaantumisen avulla, kun hyödynnetään mukautuvia viestintä järjestelmiä. Tämä toimii todella hyvin; Kuten Harry Hochheiserilla ja minulla, yksi sinunkin esitaistelijoista voi hyvinkin tehdä suuren läpimurron, jota sinä olit liian lähikatseinen nähdäksesi.

Luku 9

Fetchmail kasvaa isoksi

Ja siinä minä olin siistin ja kekseliäästi suunnitellun koodini kanssa. Koodin, jonka tiesin toimivan hyvin, sillä käytin sitä joka päivä, vierellään nupullaan oleva beta-lista. Asteittain minulle valkeni, että en enää tehnytkään omaa yksinkertaista viritelmääni, josta saattaisi mahdollisesti olla hyötyä muutamalle muulle henkilölle. Minulla oli ulottuvillani ohjelma, joka oli vastaus jokaisen Unix-konetta ja SLIP/PPP-sähköpostiyhteyttä käyttävän hakkerin tarpeisiin.

SMTP-uudelleenohjausominaisuuden ansiosta se oli päässyt sen verran kilpailijoiden edelle tullakseen varteenotettavaksi ”luokkatappajaksi”, yhdeksi niistä klassisista ohjelmistoista, jotka täyttävät kolonsa niin pätevästi, että muita vaihtoehtoja ei ainoastaan hylätä, vaan lähestulkoon unohdetaan.

Mielestäni tämänkaltainen tuotos ei ole suunniteltavissa. Koodarin täytyy joutua ideoiden virran viemäksi, joka on tarpeeksi voimakas ja rikas, että jälkipyykissä tulos vaikuttaa väistämättömältä, kohtalon sanelemalta, jopa ennalta määrätyltä. Ainoa keino osua oikeaan on käyttää ja hylätä koko joukko ideoita tai sitten soveltaa kehittäjän näkemystä viedäkseen muiden ajatukset pidemmälle kuin keksijä arvasikaan.

Andy Tanenbaumilla oli alkuperäinen idea rakentaa yksinkertainen natiivi Unix IBM PC:ille opetuskäyttöön (hän antoi sille nimen Minix). Linus Torvalds vei Minix konseptin pidemmälle kuin Andrew oli ajatellut ja Minix kasvoi joksikin mahtavaksi. Samalla tavalla (tosin pienemmässä mittakaavassa), otin joitakin ideoita Carl Harrikselta ja Herry Hochheiseriltä ja vein niitä kovasti eteenpäin. Kukaan meistä ei ollut alkuperäinen sillä romanttisella tavalla, jota ihmiset pitävät nerokkuutena. Tosin suurinta osaa tieteestä, insinöörisuunnittelusta ja ohjelmistokehityksestä eivät ole tehneet alkuperäiset nerot, vaikkakin hakkertimytologia päinvastaisesta kertookin.

Tuloksena oli varsin laadukasta koodia. Juuri sellaista, jonka vuoksi jokainen hakkeri elää! Ja se tarkoitti että minun olisi pitänyt asettaa tavoitteeni korkeammalle. Tehdäkseni fetchmailista niin hyvän kuin halusin, minun olisi pitänyt tehdä se vastaamaan, ei pelkästään omiani vaan myös ulkopuolisten käyttäjien tarpeita. Ja tehdä se siten, että ohjelma on yksinkertainen ja vakaa.

Ensimmäinen ja ylivoimaisesti tärkein ominaisuus jonka kirjoitin sen jälkeen kun tajusin tämän oli tuki monipudotukselle - ominaisuudelle, jolla postit voi noutaa postilaatikosta, johon koko käyttäjäryhmän postit on kasattu, ja välittää jokainen yksittäinen viesti sen vastaanottajalle.

Päätin lisätä monipudotuksen tuen osittain joidenkin käyttäjien esittämien toiveiden takia. Mutta tärkein syy oli se, että uskoin sen toteuttamisen auttavan löytämään koodissa olevia virheitä, sillä ominaisuuden toteuttaminen pakottaisi minut käsittelemään viestien välitystä yleisemmin. Niin kuin tapahtuikin. RFC 822 -standardin mukaisen osoitteiden käsittelyn toteuttaminen kesti huomattavan pitkään, ei siksi että joku yksittäinen osa olisi ollut hankala toteuttaa, vaan koska se käsitti lukuisia yksittäisiä ja sekavia yksityiskohtia.

Mutta monipudotuksen vaatima viestien osoittaminen osoittautui erityisen hyväksi suunnittelupäätökseksi, sillä.

14. Jokaisen työkalun tulisi olla käyttökelpoinen odotetulla tavalla, mutta todella mahtava työkalut ovat hyödyllisiä odottamattomilla tavoilla.

Odottamaton käyttö monipudotusta tukevalle fetchmailille on postituslistojen hallinta siten, että listanimet muutetaan vastaanottajiksi Internet-yhteyden asiakaspuolella. Tämä mahdollistaa sen, että henkilökohtaisella tietokoneella ja Internet-yhteydellä voi hallinnoida postituslistoja ilman jatkuvaa pääsyä palveluntarjoajan osoitetiedostoihin.

Toinen tärkeä beta-testaajien vaatima muutos oli tuki 8-bittiselle MIMElle (Multipurpose Internet Mail Extensions). Tämä oli melko helppo tehdä, koska olin pitänyt koodin huolellisesti siistinä 8-bitistä (ASCII-merkistössä käyttämätöntä kahdeksatta bittiä ei käytetty siirtämään tietoa ohjelman sisällä). En siksi että olisin odottanut tämän ominaisuuden lisäämisen vaatimista, vaan lähinnä noudattaakseni toista sääntöä:

15. Mitä tahansa yhdyskäytäväohjelmaa kirjoitettaessasi pyri häiritsemään datavirtaa mahdollisimman vähän, äläkä koskaan heitä pois mitään tietoa, ellei vastaanottaja pakota tekemään niin!

Jos en olisi noudattanut tätä sääntöä, 8-bittisen MIME-tuen toteuttaminen olisi ollut vaikeaa ja siitä olisi tullu buginen. Mutta nyt minun tarvitsi vain lukea MIME-standardi (RFC 1652) ja lisätä yksinkertainen pala logiikkaa otsakkeen muodostamiseksi.

Jotkut eurooppalaisista käyttäjistä kinusivat minua lisäämään mahdollisuuden rajoittaa istunnon aikana noudettavien viestien määrää jotta he voivat hallita kalliiden puhelinyhteyksiensä kuluja. Vastustin tätä pitkään, enkä ole vieläkään täysin tyytyväinen siihen. Mutta jos kirjoitat ohjelmia maailmalle, on kuunneltava asiakkaita - tämä ei muutu, vaikka he eivät maksakaan sinulle rahaa.

Luku 10

Vielä pari oppituntia Fetchmailista

Ennen kuin palaamme ohjelmistokehitystä yleisesti käsitteleviin aiheisiin, on syytä käydä läpi vielä muutama Fetchmailista opittu asia. Vähemmän tekniset lukijat voivat huoletta hypätä tämän luvun yli.

Asetustiedoston kielioppi sisältää vapaaehtoisia "häly-avainsanoja, jotka jäsentäjä jättää huomioimatta. Tämä mahdollistaa englannin kielen kaltaisen muotoilun, joka on huomattavasti selkeämpää lukea kuin perinteiset avain-arvo-parit, jotka jäävät jäljelle kun hälysanat jätetään pois.

Tämä järjestelmä sai alkunsa eräänä iltana kun huomasin, kuinka suurelta osin asetustiedoston määritykset muistuttivat yksinkertaista käskykieltä. Tässä on myös yksi syy siihen, miksi vaihdoin Popclient-ohjelman avainsanan "server"(palvelin) muotoon "poll"(kysely).

Näytti siltä, että yksinkertaisen käskykielen muuttaminen enemmän englannin kielen kaltaiseksi voisi tehdä siitä helpomman käyttää. Vaikka olenkin yksinkertaisten mm. Emacsissa, HTML:ssä ja monissa tietokantamoottoreissa käytettävien kielten vannoutunut kannattaja, en ole yleensä erityisemmin pitänyt englannin kieltä muistuttavista lauseopeista.

Ohjelmoijat ovat perinteisesti pitäneet asetustiedostoista, jotka ovat hyvin tarkkoja ja yksinkertaisia ilman mitään ylimääräistä. Tämä on perua ajalta, jolloin tietokoneresurssit olivat kalliita ja tiedostojen jäsentelyn tuli olla mahdollisimman halpaa ja yksinkertaista. Englannin kieli, jossa jopa puolet on sisällön kannalta ylimääräistä, oli siinä valossa täysin mahdoton hyväksyä.

Oma syyni englannin kieltä muistuttavien kielioppien välttelemiseen ei kuitenkaan ollut tämä. Mainitsin sen vain esimerkkinä. Kun tietojenkäsittelyresurssit ovat halpoja, kielen ytimekkyyden ei pitäisi olla itseisarvo. Nykyään kielen on tärkeämpää olla helposti luettavaa ihmiselle kuin halpaa koneelle.

On kuitenkin olemassa muitakin asioita jotka on otettava huomioon. Yksi on tällaisen luonnollisen kielen kaltaisen kielen jäsentämisen monimutkaisuus. En halunnut päätyä tilanteeseen, jossa tämä seikka olisi keskeinen syy ongelmiin ja samalla vaikeasti ymmärrettävä käyttäjille. Toinen asia on se, että englannin kielen kaltaisen kieliopin luominen yleensä vaatii, että käytettävä "englanti"eroaakin huomattavasti oikeasta englannista, jolloin sen käyttäminen on yhtä monimutkaista kuin perinteisten asetustiedostojen kirjoittaminen. Tämä näkyy selkeästi niin kutsutuissa "neljännen sukupolven"ja kaupallisten tietokantojen kielissä.

Fetchmailin asetustiedoston kielioppi näyttää välttävän nämä ongelmat, sillä siinä käytetty kieli on tiukasti rajoitettu. Se on kaukana yleiskäyttöisestä kielestä. Sillä ilmaistavat asiat

eivät ole kovinkaan monimutkaisia, joten ei ole kovinkaan todennäköistä, että käyttäjälle tulisi ongelmia asetustiedoston kielen ja varsinaisesta englannin kielestä otetun pienen osan välilä. Mielestäni tarinan opetus on:

16. Kun kielesi on todella yksinkertainen, voi olla hyvä, että se muistuttaa luonnollista kieltä.

Toinen oppitunti koskee "tietoturvaa salailemalla-periaatetta. Jotkut Fetchmailin käyttäjät pyysivät minua muuttamaan ohjelmaa siten, että salasanat tallennettaisiin asetustiedostoon salattuna, jotta urkkijat eivät saisi niitä haltuunsa.

En tehnyt muutosta, sillä se ei olisi oikeasti parantanut tietoturvaa. Kuka tahansa, jolla on oikeudet lukea käyttäjän asetustiedostoa, voi myös ajaa Fetchmailia käyttäjän oikeuksilla. Ja jos tällainen henkilö haluaa saada selville käyttäjän salasanan, hän voi hakea ohjelman lähdekoodista salasanan purkamiseen tarvittavan osan.

Asetustiedoston (.fetchmailrc) salasanojen kryptaus olisikin tuonut vain valheellisen turvallisuuden tunteen käyttäjille, jotka eivät ajattele asiaa loppuun asti. Tästä saadaan yleinen sääntö:

17. Turvajärjestelmä on yhtä turvallinen kuin se on salainen. Vältä näennäisesti salattuja ominaisuuksia.

Luku 11

Välttämättömät ennakkoehdot basaarityylille

Tämän aiheen alkuaikojen arvostelijat ja koeyleisö esittivät johdonmukaisesti samoja kysymyksiä menestyksekkään basaarityylisen kehitysprojektin alkuehdoista. Kysymykset alkuehdoista liittyivät projektipäällikön kelpuutuksen, koodin kehitystilanteeseen julkaisuhetkellä ja hetkeen jolloin kannattaa aloittaa apukehittäjien joukkoa keräämään.

On melkoisen selvää, ettei projektia voi luoda tyhjästä suoraan basaarityyliin. Projektia voidaan testata, debugata ja kehittää basaarityylillä, mutta uuden projektin kehittäminen tyhjästä on todella vaikeaa basaarityylillä. Linus ei yrittänyt sitä, kuten en minäkään. Orastava kehittäjäyhteisösi tarvitsee jotain ajokelpoista, jota se voi testata ja jolla se voi leikkiä.

Kun aloitat yhteisön rakentamisen, tulee sinun pystyä antamaan toteutettavissa oleva lupaus. Ohjelmasi ei tarvitse toimia erityisen hyvin, se voi olla karu, buginen, keskeneräinen ja huonosti dokumentoitu. Sen tulee kuitenkin olla ajettavissa ja vakuuttaa mahdolliset tulevat kehittäjät siitä, että se todella voi kehittyä joksikin todella hienoksi ohjelmaksi tulevaisuudessa.

Linux ja Fetchmail olivat julkaisuvaiheessaan jo vahvalla pohjalla ja niiden perusrakenne viehätti. Monet kuvaamani kaltaista basaarimallia harkinneet ihmiset ovat pitäneet tätä syystäkin ensiarvoisen tärkeänä, mistä ovat päätyneet lopputulemaan, että suunnittelun sisäinen näkemys ja projektinvetäjän nokkeluus ovat välttämättömiä ominaisuuksia.

Linus otti rakennemallinsa Unixista. Minä taas omani varhaiselta pop-asiakasohjelmalta, joka kuitenkin muuttui myöhemmin suhteellisesti ottaen Linuxia huomattavasti enemmän. Onko basaarityylisellä ryhmänvetäjä-koordinaattorilla siis oltava ehdottomasti poikkeuksellisia kykyjä etukäteissuunnitteluun, vai voiko hän pärjätä käyttämällä toisten kykyä luoda?

Mielestäni koordinaattorilta ei välttämättä vaadita kykyä tuottaa poikkeuksellisen hienoja suunnitelmia, mutta on ehdottoman tärkeää, että koordinaattori osaa erottaa hyvät ideat huonoista.

Sekä Linux- että Fetchmail-projektit ovat todiste tästä. Linuksella, joka ei ole huikean omintakeinen suunnittelija, on armoitettu kyky ja ymmärrys tunnistaa hyvä suunnittelu, jonka on sitten ottanut mukaan Linuxin ytimeen. Olenkin jo aiemmin kuvannut, kuinka kaikkein tärkein yksilöitävissä oleva idea Fetchmailiin (SMTP-edelleenlähetys) tuli muualta.

Esseeni alkuvaiheen yleisön edustajat ovat maininneet, että minulla on taipumus aliarvioida suunnittelun omintakeisuutta basaarihankkeissa, koska minulla on sitä kohtuullisesti itsel-

länikin, mistä syystä pidän sitä itsestäänselvyytenä. Väitteessä voi olla perää: suunnittelu ohjelmoimisen ja virheenkorjauksen vastakohtana on varmasti vahvimpia puoliani.

Nokkeluus ja omintakeisuus ohjelmistosuunnittelussa kantavat mukanaan ongelmaa: siitä tulee tapa, ohjelmoija alkaa tehdä asioista fiksuja ja monimutkaisia, kun niiden pitäisi antaa olla karuja ja helppotajuisia. Minulla on ollut projekteja, jotka ovat epäonnistuneet siksi, että olen syyllistynyt tähän syntiin, mutta Fetchmailin tapauksessa onnistuin välttämään sudenkuopan.

Uskon täten, että Fetchmail-projekti onnistui osin koska vastustin itsessäni taipumusta olla fiksu. Tämä sotii vähintäinkin sitä vastaan, että suunnittelun omaperäisyys olisi menestyksekkään basaarihankkeen edellytys. Otetaanpa Linux: oletetaan, että Linus Torvalds olisi yrittänyt kehitystyön kuluessa riisua käyttöjärjestelmäsuunnittelun perustavaa laatua olevat innovaatiot pois. Olisiko tämän jälkeen enää luultavaa, että saavutettu ydin olisi ollenkaan niin vakaa ja onnistunut kuin nyt?

Tehtävään vaaditaan tiettyä pohjatason suunnittelu- ja koodaustaitoa, totta kai, mutta oletan lähes jokaisen basaarihankkeeseen lähtevän jo ylittävän nämä vähimmäisvaatimukset. Avoimen lähdekoodin yhteisön sisäiset maineen ja kunnian markkinat hillitsevät ihmisiä aloittamasta kehitystyötä, jota eivät ole valmiita seuraamaan loppuun saakka. Toistaiseksi tämä näyttää toimineen hyvin.

On toinenkin taito, jota ei tavallisesti liitetä ohjelmistosuunnitteluun, joka mielestäni on basaariprojekteille yhtä tärkeä kuin suunnittelun nokkeluus, ehkä jopa tärkeämpikin. Basaariprojektin koordinaattorilla tai vetäjällä on oltava hyvät ihmissuhde- ja viestintätaidot.

Tämän pitäisi olla itsestäänselvyys. Voidaksesi rakentaa kehittäjäyhteisön sinun on houkuteltava ihmisiä, saatava heidät kiinnostumaan siitä mitä olet tekemässä, ja pidettävä heidät tyytyväisinä tekemänsä työn määrän suhteen. Tekninen läppä on huomattava osa tätä, mutta paketissa on paljon muutakin. Antamasi vaikutelma ihmisenä on tärkeä sekin.

Ei ole mikään sattuma, että Linus on mukava heppu, joka saa muut pitämään henkilöstään ja auttamaan itseään. Ei sekään ole sattumaa, että minä olen ulospäin suuntautuva kaveri, joka nauttii olla ihmisten kanssa tekemisissä, ja jolla on joitakin stand-up -koomikolta vaadittavia vaistoja ja ulosannin piirteitä. Jotta saa basaarimallin toimimaan, on erittäin suuri etu, jos osaa edes vähäisessäkään määrin hurmata muita.

Luku 12

Avoimen lähdekoodin sosiaalinen puoli

Aivan totuudenmukaisesti on kirjoitettu, että parhaat hakkeriniksit ovat saaneet alkunsa vastauksena ohjelmoijansa käytännön tarpeisiin, ja ne ovat levinneet laajemmalle siksi, että ongelma on ollut yhteinen suurelle joukolle käyttäjiä. Tämä vie meidät takaisin ensimmäisen säännön äärelle, joka voidaan muotoilla ehkä hieman käytännöllisemmällä tavalla toisin:

18. "Ratkaistaksesi mielenkiintoisen ongelman, aloita etsimällä ongelma, joka on sydäntäsi lähellä."

Näin oli Carl Harriksen esikuvallisen pop-asiakasohjelman kanssa, ja samoin oli minun Fetchmailini laita. Mutta tämä on kuitenkin ymmärretty jo pitkään. Kiinnostavin huomio, johon Linuxin ja Fetchmailin versiohistoria näyttää vaativan meitä keskittymään, on seuraava taso - ohjelman evoluutio suuressa, aktiivisessa apulaiskehittäjien yhteisössä.

Esseessään *The Mythical Man-Month* Fred Boorks huomioi, että ohjelmoijan aika ei ole jaettavissa. Lisättäessä kehittäjiä myöhästyneeseen projektiin saa sen vain myöhästymään enemmän. Kuten olemme nähneet aiemmin, hän väitti, että projektin kommunikoinnin monimutkaisuus kasvaa neliöllisesti kehittäjien lukumäärän mukaan, kun taas työteho kasvaa vain lineaarisesti. Brookin lakia on laajasti kohdelut selviönä. Mutta me tutkimme tätä lakia monesta suunnasta ja avoimen lähdekoodin kehittäminen osoittaa taustalla olevan oletuksen empiirisesti vääräksi. Jos Brooken laki olisi koko totuus, Linuxin ei olisi pitänyt olla mahdollinen.

Gerald Weinbergin klassikkoteos *The Psychology of Computer Programming* tuki sitä, minkä näin jälkiviisaana voimme nähdä elintärkeänä korjauksena Brooksin teorioihin. Käsitellessään minäkeskitöntä "egotonta ohjelmointia"Weinberg havainnoi, että niissä yrityksissä, joissa kehittäjät eivät ole omistushaluisia omien koodiensa suhteen ja rohkaisevat muita ihmisiä etsimään ohjelmointivirheitä ja kertomaan mahdollisista kehityskohteista, parannukset tapahtuivat dramaattisesti nopeammin kuin muualla. Kent Becksin "äärimmäinen ohjelmointi-tekniikka, jossa ohjelmoijat asetetaan pareittain kurkkimaan toistensa olkien yli, voidaan nähdä yrityksenä aikaansaada sama vaikutus.

Weinbergin terminologian valinta on kenties estänyt hänen analyysiänsä saamasta sellaista hyväksyntää kuin se olisi ansainnut- kukapa ei olisi hymyillyt ajatellessaan Internet hakkereita kuvailtavan *egottomiksi*. Mutta minusta näyttää siltä, että hänen argumenttinsa nyt paljon vakuuttavampi kuin koskaan.

Valjastaessaan "egottoman ohjelmoinnin"vaikutuksen täyteen voimaansa, basaarimenetelmä voimakkaasti lieventää Brooken lain vaikutusta. Periaatetta Brooken lain takana ei ole ku-

mottu, mutta sen vaikutus voidaan vesittää käyttämällä suurta kehittäjien joukkoa joidenka keskinäinen kommunikaatio on vähäistä ja näin epälineaarisuudet, jotka muuten olisivat ilmeisiä, saadaan hallintaan. Tämä muistuttaa Newtonin ja Einsteinin fysiikkojen suhdettavanhempi järjestelmä on yhä voimassa matalilla energioilla, mutta jos massa ja nopeudet ovat riittävän suuria voidaan saada yllätyksiä kuten ydinräjähdys tai Linux.

Unixin historian olisi pitänyt valmistaa meitä siihen, mitä kaiken aikaa opimme Linuxista, ja minkä minä olen Linuksen menetelmiä tarkoituksellisesti kopioiden todentanut kokeellisesti pienemmässä mittakaavassa [EGCS]). Silloin kun koodaaminen pysyy oleellisesti yksinäisenä harrasteena, todella mahtavat hakit tulevat valjastamalla muun yhteisön huomio ja aivovoima. Kehittäjä, joka käyttää vain omia aivojaan omassa suljetussa projektissaan on kaukana niiden kehittäjien perässä, jotka tietävät miten luodaan avoin ja kehittyvä tausta, jossa palaute suunnitteluavaruuden tutkimisesta, ohjelmoinnista, virheiden etsimisestä ja muista parannuksista tulevat sadoilta, jopa tuhansilta ihmisiltä.

Mutta perinteinen Unix-maailma oli estynyt saamaan tätä lähestymistapaa täyteen laajuuteensa useiden tekijöiden seurauksena. Eräs näistä tekijöistä oli lupaehtojen, liikesalaisuuksien ja kaupallisten etunäkökohtien mukanaan tuomat lakirajoitukset. Toinen jälkiviisaasti sanottuna oli se, että Internet ei ollut vielä riittävän hyvä.

Ennen halpaa Internetiä oli olemassa joukko maantieteellisesti tiiviitä yhteisöjä, joissa vallitseva kulttuuri rohkaisi Weinbergin epäitsekeskeiseen ("egottomaan") ohjelmointiin, missä kehittäjä pystyi helposti vetämään puoleensa monia taitavia touhuajia ja apukehittäjiä. Bellin Laboratoriot, MIT AI:n tekoälylaboratorio ja LCS Labs, sekä Kalifornian yliopisto Berkeleyssä muodostuivat uusien keksintöjen hautomoiksi, joiden innovaatioista monet ovat vieläkin tarunhohtoisia ja päteviä.

Linux oli ensimmäinen projekti, jossa tietoisesti ja onnistuneesti käytettiin koko maailman mittaista aivoriihtä. En usko, että Linuxin hautomisajan osuminen yhteen World Wide Webin syntymän kanssa olisi ollut vain sattumaa, ja että Linux jätti kapalonsa juuri samalla ajanjaksolla 1993-1994, jolloin internetyhteydet yleistyivät, ja valtavirta kiinnostui Internetistä. Linus oli ensimmäinen henkilö, joka oppi kuinka pelataan uusilla säännöillä, jotka kaikkialle tunkevat Internetyhteydet tekivät mahdollisiksi.

Vaikka halpa internet olikin oleellinen edellytys Linuxin mallille kehittyä, en usko sen yksinään olleen riittävä vaatimus. Toinen tärkeä tekijä oli kehittämisen johtamismalli, sekä yhteistyöhön pohjautuvat tavat, jotka mahdollistivat uusien kehittäjien huokuttelemisen ja parhaan mahdollisen tehon saamisen annetusta mediasta.

Mutta mikä oli tämä johtamistyyli ja mitä olivat nämä tavat? Ne eivät voi perustua voimasuhteisiin - ja vaikka voisivatkin, pakolla johtaminen ei tuota niitä tuloksia, joita näemme. Weinberg lainaa osuvasti 1800-luvun venäläisen anarkistin Pjotr Kropotkinin omaelämäkertaa *Vallankumouksellisen muistelmat*, joka sattuvasti sivuaa aihettamme:

"Kasvettuani perheessä, jolla oli maaorjia, astuin työelämään, kuten kaikki sukupolveni nuoret miehet, täysin vakuuttuneena siitä, että oli tarpeen komentaa, määrätä, nuhdella, rangaista ja sen sellaista. Mutta kun jo varhaisessa vaiheessa minun oli otettava vastuu vakavista työhankkeista, joissa kohtasin vapaita ihmisiä, ja jossa maailmassa jokaisella virheelläni olisi välittömät vakavat seuraukset, aloin ymmärtää ja arvostaa eroa komentokulttuurin ja yhteisymmärrykseen perustuvan johtamismallin välillä. Edellinen toimii moitteetta armeijan sulkeisissa, mutta siitä ei ole mitään hyötyä tosielämässä, jossa tavoitteet ovat saavutettavissa vain monien yhteen hiileen puhaltavien tahtojen ponnistuksin."

Monen *yhteenpuhaltavan tahdon yhteiset ponnistukset* on juuri se, mitä Linuxin kaltainen

hanke tarvitsee, ja *komentamisen periaatetta* on käytännöllisesti katsoen mahdotonta ajatella niiden vapaaehtoisten anarkistien keskuudessa, jotka muodostavat Internet-yhteisön. Toimiakseen ja pärjätäkseen niiden hakkereiden, jotka ovat halukkaita johtamaan yhteisöhankkeita, on opittava värväämään ja puhaltamaan energiaa eri sidosryhmiin vähän Kropotkinin *ymmärryksen periaatteen* hengen mukaisesti. Heidän täytyy oppia soveltamaan Linuksen lakia.

Aiemmin viittasin *Delphi-vaikutukseen* mahdollisena Linusin lain selityksenä. Mutta paljon paremmat esimerkit adaptoivista järjestelmistä löytyvät biologiasta ja taloustieteestä. Linux-maailma käyttäytyy monessa mielessä kuten vapaat markkinat tai ekosysteemi, joukko itsekkäitä toimijoita pyrkii maksimoimaan hyötynsä ja jossa prosessi pyrkii itse ohjaamaan itseään spontaanista. Tämän kaltainen ohjaus toimii paljon tehokkaammin ja tarkemmin kuin mihin mikään keskitetty ohjaus pystyy. Tässä onkin paikka tarkastella *ymmärtämisen periaatetta*.

Se *hyötyfunktio*, jota Linux hakkerit ovat maksimoimassa, ei ole taloustieteen klassinen hyötyfunktio, vaan se on aineetonta, heidän oman henkisen tyydytyksen ja maineen muiden hakkerien keskuudessa hyötyfunktio. (Joku voisi kutsua heidän motivaatiotansa jopa *altruistiseksi*, mutta tämä jättää huomiotta sen tosiasian, että altruisimi itsessään on eräst henkisen tyydytksen muoto altruistille). Vapaaehtoisen työn kulttuurit eivät ole itse asiassa epätavallisia; eräs niistä johon olen pitkään kuulunut on scifi-fanien ryhmä, joka toisin kuin hakkeriporukka, on pitkään ja selvästi tunnustanut *egobuustauksen* (lumoutuminen omasta maineesta muiden fanien keskuudessa) olennaiseksi ajavaksi voimaksi vapaaehtoiseen aktiivisuuteen.

Asettamalla itsensä onnistuneesti portinvartijaksi projektissa, jossa kehitystyö on pääosin muiden työtä, ja pitämällä yllä mielenkiintoa hankkeeseen kunnes siitä tuli itseäänkannattava, Linus osoitti Kropotkinin *jaetun ymmärryksen periaatteen* terävää hallintaa. Tämä näennäistaloudellinen näkökanta Linux-maailmaan auttaa meitä ymmärtämään, kuinka Kropotkinin periaatetta on sovellettava.

Me voimme käsittää Linusin menetelmän tapana luoda tehokkaat *egobuustauksen* markkinat - ja siten yhdistää yksittäisten hakkerien itsekkyys niin tiukasti kuin mahdollista hankaliin ratkaisuihin, jotka voidaan saavuttaa vain pysyvällä yhteistyöllä. Fetchmail projektilla olen näyttänyt (vaikkakin pienemmässä mittakaavassa), että tätä menetelmää voidaan kopioida hyvin tuloksin. Ehkäpä tein sen jopa hieman tietoisemmin ja systemaattisemmin kuin hän.

Monet (erityisesti he, joilla on poliittisia epäluuloja vapaita markkinoita kohtaan) voisivat epäillä, että itseensä suuntautuneiden egoistien kulttuuri on pirstaleinen, alueellinen, salaperäinen ja vihamielinen. Mutta tämä epäilys on selvästi todettavissa vääräksi (annan vain yhden esimerkin) vain katsomalla Linux dokumentaation tyrmäävää moninaisuutta, laatua ja syvyyttä. On taatusti kaikkien tiedossa, että ohjelmoijat inhoavat dokumentointia; kuinkapa siis Linux-hakkerit ovat luoneet niin paljon dokumentaatiota? Ilmeisesti Linuxin egobuustauksen vapaat markkinat toimivat paremmin luodessaan siistiä ja toisia huomioon ottavaa käyttäytymiskulttuuria kuin massiivisesti rahoitetut kaupallisten dokumentaatioiden tuottajat.

Sekä fetchmail että Linuxin kerneliprojekti näyttävät, että palkitsemalla monien hakkereiden egot asianmukaisesti, vahva kehittäjä/koordinoija voi käyttää Internettiä kaapatakseen suuren määrn kanssakehittäjiä ilman projektin muuttumista kaaottiseksi sotkuksi. Joten esitän seuraavan vastaehdotuksen Brooken laille:

Edellyttäen että kehitystyöstä vastaavalla henkilöllä on käytössään vähintäin Internetin ta-

soinen viestintäväline, ja henkilö tietää kuinka hallita ilman pakkoa, on usean pään yhteen lyöminen väistämättä parempi kuin yksin toimiminen.

Luulenpa, että avoimen lähdekoodin tulevaisuus on sellaisten käsissä, jotka osaavat pelata Linusin säännöillä, ihmisten jotka astuvat ulos katedraalista ja omaksuvat basaarin toimintamallikseen. Tämä ei kuitenkaan tarkoita sitä, että henkilökohtaisella näkemyksellä ja älyn voimalla ei olisi enää merkitystä. Pikemminkin olen sitä mieltä, että vapaan lähdekoodin ohjelmistojen etulyöntiasema kuuluu niille, jotka pitävät lähtökohtanaan sielunsa silmien näkyjä ja älyn kirkkautta, jotka yhdistettynä vapaaehtoisten eturyhmien rakentamiseen antavat lisäpotkua koko yhteisölle.

Tämä ei ehkä ole ainoastaan avoimien ohjelmistojen tulevaisuus. Suljetun ohjelmiston kehittäjä ei voi vastata siihen lahjakkuuden varantoon, jonka Linux yhteisö voi tuoda vastaamaan ongelmasta. Hyvin harvalla on varaa palkata enemmän kuin 200 (1999: 600, 2000: 800) ihmistä, jotka ovat avustaneet fetchmailia

Avoimen lähdekoodin kulttuuri saattaa lopultakin viedä voiton, mutta ei siksi, että talkootyö olisi moraalisesti oikein tai rajoitetut ohjelmistot olisivat samassa mielessä väärin. Tämä siis edellyttäen, että uskomme jälkimmäiseen, mitä minä tai Linus emme tee, vaan yksinkertaisesti siitä syystä, että suljetun koodin maailma ei voi voittaa evoluution varustelukilpailua avoimen lähdekoodin yhteisön kanssa, jolla on suunnattomasti enemmän laadukkaita miestyövuosia käytettävissään ongelmanratkaisussa.

Luku 13

Johtaminen ja Maginot

Alkuperäinen vuonna 1997 julkaistu Katedraali ja basaari päättyi ylläolevaan näkemykseen tyytyväisten verkottuneiden ohjelmoitsija-anarkistien uurastajalaumoista, jotka selättävät ja ottavat haltuunsa tavanomaisten suljettujen ohjelmistojen hierarkisen maailman.

Monet epäilevät tuomaat eivät kuitenkaan olleet tästä vakuuttuneita, ja heidän esiin nostamansa kysymykset ansaitsevat tulla reilusti käsitellyksi. Monet vasta-argumentit perustuvat siihen, että basaarimallin kannattajat ovat aliarvioineet tavanomaisen johtamisen kautta saatavan tuottavuuden lisäyksen.

Perinteisesti ajattelevat ohjelmistokehitysjohtajat kommentoivat usein paheksuen, että epämuodollinen tapa, jolla avoimen lähdekoodin maailman projektiryhmät muodostuvat, muuttuvat ja hajoavat, kumoaa merkittävän osan eduista, joita avoimen lähdekoodin yhteisöllä on mihin tahansa suljetun koodin kehittäjään verrattuna. Heidän mielestään ohjelmistokehityksessä vain pitkän aikajanan uhraukset ja varmuus investoinneista tuotteeseen tulevaisuudessa ovat tärkeitä, ei se kuinka monta eri ihmisyksilöä on pistänyt lusikkansa soppaan.

Väittämässä on varmasti hitunen tottakin. Itse asiassa esseessäni *The Magic Cauldron* ('taikapata') olenkin kehittänyt ajatusmallin, jonka mukaan odotettavissa oleva tulevaisuuden palveluarvo on avain ohjelmistotuotannon kassalippaaseen.

Väitteeseen on piilotettu silti iso ongelma: siihen sisältyy olettamus, että avoimen lähdekoodin kehitystyö ei pysty pitkäjänteiseen suoritukseen. Todellisuudessa on ollut vapaita projekteja, joiden kehitystyö pysyi kurssissaan ja ylläpitäjäyhteisö tehokkaasti koossa hyvinkin pitkään ilman tavanomaisen liikkeenjohdon välttämättömäksi katsomia institutionaalisia valvontarakenteita. GNU Emacs -editori on äärimmäinen ja havainnollinen esimerkki tästä. Sen tiukka näkemyksellinen arkkitehtuuri on aikaansaannosta satojen osanottajien työstä viidentoista vuoden aikana. Tämä siitäkin huolimatta, että vaihtuvuus on ollut suurta, ja vain yksi kehittäjistä (hankkeen alullepanija) on ollut aktiivisesti mukana alusta saakka. Mikään suljetun ohjelmiston editori ei ole päässyt lähellekään samaa kestävyysennätystä.

Tästä saadaan syy kyseenalaistaa tavanomaisesti johdetun ohjelmistokehityksen edut - syy joka on riippumaton muista katedraali- vastaan basaarimalli -argumenteista. Jos GNU Emacs pystyi pitämään yllä määrätietoista projektia viidentoista vuoden ajan, ja mikäli Linux ylsi samaan kahdeksassa vuodessa, vaikka harware ja laitealusta muuttuivat nopeasti, tai jos on ollut monia hyvin suunniteltuja, yli viisi vuotta kestäneitä avoimen lähdekoodin projekteja (kuten asianlaita on), niin silloin voimme hyvällä syyllä ihmetellä mitä, jos mitään, tavanomaiseen kehitystyöhon sijoitetut valtavat rahasummat ovat meille poikineet.

Oli saatu vastine mikä hyvänsä, se ei ainakaan takaa deadlinen takuuvarmaa täyttymistä, budjetissa pysymistä tai kaikkien luvattujen ominaisuuksien toteutumista. Vain harva "manageroitu"projekti saavuttaa edes yhden näistä tavoitteista, saatikka kaikkia. Kyky mukautua teknologian tai talouden muutoksiin tuotteen elinkaaren kuluessa ei sekään näyttäisi olevan listalla. Näissä suhteissa avoimen lähdekoodin liike on osoittautunut paljon tehokkaammaksi, mistä todisteena esimerkiksi vertailu 30-vuotisen Internetin ja suljettujen verkkoratkaisuiden elinkaarten lyhyiden puolittumisaikojen välillä tai Microsoftin kallis siirtyminen 16-bittisestä 32-bittiseen teknologiaan verrattuna Linuxin lähestulkoon vaivattomaan muuttumiskykyyn samana aikana - ei vain Intelin kehityslinjassa, vaan myös yli kymmenellä muulla alustalla mukaan lukien 64-bittinen Alpha.

Monien mielestä yksi syy käyttää tavanomaista tuotantotapaa on sen tuoma laillinen vastuunkantaja ja mahdollisuus saada korvausta, jos projekti menee metsään. Mutta tämä on pelkkää harhaa: useimmat ohjelmistolisenssit suovat valmistajilleen vastuuvapauden kaikista kaupallisista ja tuotannollisista korvausvelvoitteista, eikä ohjelmiston toimimattomuudesta saaduista palautuksista ole juurikaan näyttöä. Vaikka asia olisi päinvastoinkin, on suhtautumistapa väärä: et osta ohjelmaa päästäksesi oikeussaliin, vaan halusit toimivan ohjelmistotuotteen.

Mitä näillä liikkeenjohdollisilla kulungeilla sitten oikein saa?

Sen ymmärtämiseksi meidän täytyy muodostaa kuva siitä, mitä ohjelmistokehitysmanagerit luulevat tekevänsä. Eräs tuntemani nainen, arvatenkin erittäin pätevä työssään, sanoo, että ohjelmistoprojektien hallinnalla on viisi tehtävää:

- Määritellä tavoitteet ja pitää kurssi kohti näitä yhteisiä tavoitteita
- Valvoa ja varmistaa, että elintärkeitä seikkoja ei sivuuteta
- Valaa työntekijöihin uskoa ja intoa ikävystyttävän, mutta välttämättömän arkityön suorittamiseksi
- Hyödyntää työvoimaa parhaan tuottavuuden saavuttamiseksi
- Pitää huolta tarvittavista resursseista projektin toteuttamiseksi

Kaikki vallan perusteltuja tavoitteita, mutta avoimen lähdekoodin mallissa ja sitä ympäröivässä sosiaalisessa viitekehyksessä ne saattavat alkaa näyttää oudon toissijaisilta. Laitetaanpa ne käänteiseen järjestykseen.

Ystäväni mukaan valtaosa resurssien vaalimisesta on perusteiltaan puolustusta: kun sinulla kerran on ihmiset, koneet ja työtilat, sinun on pakko puolustaa niitä kollegoiltasi, jotka kilpailevat samoista resursseista, ja ylemmiltäsi, jotka mielivät jakaa rajatun työryhmän resurssit uudelleen.

Avoimen lähdekoodin liikkeen kehittäjät ovat vapaaehtoisina, itsensä valitsemina kiinnostustensa ja kykyjensä mukaan auttamassa projektia, jonka piirissä työskentelevät. Tämä asetelma pätee, vaikka heille maksettaisiin koodaamisesta. Vapaaehtoisuuden eetos on omiaan huolehtimaan resurssipihtareista automaattisesti, sillä he tuovat resurssit mukanaan työpisteeseensä. Eikä ole juurikaan mitään syytä "leikkiä puolustavaa esimiestä"tavanomaisessa mielessä.

Joka tapauksessa halpojen tietokoneiden ja nopean Internetin maailmassa havaitsemme johdonmukaisesti, että ainoa resurssirajoitus on osaava ja asiantunteva huomioaika. Avoimen

lähdekoodin projektit tuskin koskaan kuivuvat kokoon välineiden tai toimistotilan puutteessa, vaan ne kuolevat vasta kun kehittäjät itse menettävät mielenkiintonsa.

Näin ollen on kaksinverroin tärkeämpää, että avoimen lähdekoodin hakkerit järjestyvät ryhmiin itsenäisesti mahdollisimman suuren tuottavuuden nimissä. Sosiaalinen ympäristö toimii säälimättömänä luonnonvalitsijana ja kilpailuttajana. Molempien maailmojen projekteissa mukana ollut tuttavani uskoo, että avoin lähdekoodi on menestynyt osin, koska sen kulttuurissa mukaan hyväksytään vain lahjakkain viitisen prosenttia ohjelmoivasta väestöstä. Hän käyttää suuren osan ajastaan kaitsemaan jäljelle jäänyttä 95 prosenttia, ja siten on kokenut omakohtaisesti hyvin tunnetun tuottavuuden sadasosatekijän varianssin ohjelmoijista kyvykkäimpien ja vain kelvollisten välillä.

Varianssin koko on aina nostanut esiin kiusallisen kysymyksen: olisiko projektien kannalta parempi, jos enemmän kuin puolet vähiten pätevistä jätettäisiin sen ulkopuolelle? Ajatteluun taipuvaiset managerit ovat selvittäneet itselleen jo kauan sitten, että jos tavanomaisen ohjelmistotuotannon hallinnon ainoa tehtävä olisi muuntaa kädettömimmät nettotappiosta marginaaliseksi voitoksi, ei koko hanke maksa vaivaa.

Avoimen lähdekoodin liikkeen suosio selventää tätä kysymystä huomattavasti näyttämällä toteen sen, että on usein halvempaa ja tehokkaampaa värvätä itsensä valituttaneita vapaaehtoisia Internetistä kuin hallinnoida fyysisiä rakennuksia täynnänsä ihmisiä, jotka tekisivät mieluiten kaikkea muuta.

Tästä pääsemmekin sopivasti kysymykseen motivoinnista. Vastaava ja usein kuultu tapa tukea ystäväni näkökantaa on se, että perinteinen kehitystyön johtaminen on välttämätön edellytys puristamaan huonosti motivoituneista ohjelmoijista kunnon tuotos.

Tämänkaltainen vastaus yleensä kantaa mukanaan lisäväitettä, jonka mukaan avoimen lähdekoodin yhteisö on luotettava vain niin kauan kuin työ on "seksikästä"tai "teknisesti miellyttävää". Kaikki muu jää tekemättä joko kokonaan tai tehdään vasemmalla kädellä, kunnes suunnan muuttavat kellokorttiin sidotut palkkatyöläiset, joiden yllä heiluu esimiehen ruoska. Käsittelen psykologisia ja sosiaalisia syitä suhtautua epäillen tähän väitteeseen esseessäni *Homesteading the Noosphere*.

Jos ainoa puolustus tavanomaisen suljetun koodin visusti ylhäältä ohjatulle tavalle tehdä softaa on eräänlainen ongelmien barrikadi, Maginot-linja, joka johtaa väistämättä epämielekkyyteen, niin se voi säilyä kilpailukykyisenä kussakin käyttötarkoituksessaan ainoastaan niin kauan kuin kukaan ei pidä näitä ongelmia mielekkäinä tarttua tai löydä niille kiertotietä. Heti kun avoimessa lähdekoodissa syntyy kilpailutilanne "ikävystyttävästä"ohjelmanpätkästä, loppukäyttäjät saavat välittömästi kokea, kuinka ongelmaan tartutaan sen sisältämän ongelmanratkaisuhaasteen vuoksi. Haasteellisuus on aina rahaa parempi motivaattori niin ohjelmistosuunnittelussa kuin kaikessa muussakin luovassa työssä.

Perinteisen työnjohdon ylläpitäminen vain motivoinnin vuoksi voi olla taktisesti oikea veto, mutta silti strategisesti huono ratkaisu: lyhytnäköinen voitto, mutta pitkän tähtäimen varma tappio.

Tähän mennessä olen havainnollistanut, kuinka tavanomainen kehitystyön johtaminen on huono arpa avoimeen lähdekoodiin rinnastettuna kahdesta syystä (resurssien hallinnoiminen, organisaatio), ja kuinka se näyttää elävän laina-ajalla mitä tulee kolmanteen kohtaan (motivaatioon). Nurkkaan ajettu manageri-rukka ei löydä apua edes työnvalvontaoikeudestaan: avoimen lähdekoodin yhteisön vahvin argumentti on se, että hajautettu vertaisarviointi kaataa kaikki perinteiset tavat varmistaa, että työn jälki on hyvää yksityiskohtia myöten.

Voimmeko sitten pitää päämäärien määrittelyä riittävänä oikeutuksena tavanomaisen johtamismallin aiheuttamille kuluille? Ehkä, mutta silloin meille on annettava hyvä syy uskoa, että työnjohtoryhmät ja yrityksen pitkän tähtäimen agendat ovat avoimen lähdekoodin yhteisön vastineitaan, projektinvetäjiä ja tribaalivanhimpia, parempia määrittelemään nämä tavoitteet

Ratkaisu voi olla näin perstuntumalta vaikea tehdä. Vaakakupin avoimen lähdekoodin puoli - esimerkkeinään Emacsin pitkäikäisyys tai Linus Torvaldsin kyky saattaa liikkeelle kehittäjien laumat, jotka julistavat "maailmanherruutta", ei sekään ole syypää päätöksen vaikeuteen. Pikemminkin syynä on tavanomaisten mekanismien ilmentämä kehnous, kun määritellään ohjelmistoprojektien tavoitteet ja päämäärät.

Eräs hyvin tunnettu ohjelmistosuunnittelun kansanteoreema kuuluu, että 60-75 prosenttia perinteisistä softaprojekteista eivät joko valmistu koskaan tai aiotut loppukäyttäjät eivät tuotetta huoli. Jos prosenttiluku on lähelläkään oikeata - enkä ole koskaan tavannut ketään vähänkään kokeneempaa ohjelmistomanageria, joka kiistäisi luvun - niin valtaosa projekteista on synnytetty tarpeisiin, jotka (a) eivät ole realistisesti saavutettavissa tai (b) ovat pelkkää potaskaa.

Tämä ongelma enemmän kuin mikään muu on syynä siihen, että nykypäivän ohjelmistosuunnittelumaailmassa jo pelkkä maininta "työnjohtoryhmästä"saa puistatuksen väreet kulkemaan pitkin kuulijan selkäpiitä, jopa (tai eritoten) jos kuulija on itse johtoasemassa. Ajat jolloin vain ohjelmoitsijat nurisivat kuviosta ovat jo kaukana takana. Nykyään työnteon prosessiin sarkastisesti suhtautuvat sarjakuvaleikkeet, kuten B. Virtanen, riippuvat ylimmänkin johdon työhuoneiden seinillä.

Vastauksemme tavanomaiselle ohjelmistomanagerille onkin yksinkertainen. Jos avoimen lähdekoodin yhteisö on todellakin aliarvioinut perinteisen johdon arvon, miksi niin moni teistä suhtautuu työprosessiinsa halveksunnalla?

Taaskin avoimen lähdekoodin yhteisön esimerkki valaisee tätä kysymystä oleellisesti, sillä meillähän on hauskaa työssämme. Luovassa leikissämme olemme saavuttaneet merkittäviä teknisiä, liiketaloudellisia ja psykologisia voittoja hämmästyttävän lyhyessä ajassa. Emme ole ainoastaan todistaneet, että osaamme tehdä parempia ohjelmia, vaan sen lisäksi, että työnilo on vahvuus.

Kaksi ja puoli vuotta tämän esseen ensiversion julkaisusta kaikkein mullistavinta, mitä voin lopuksi sanoa, ei olekaan enää ennustus avoimen lähdekoodin maailmanherruudesta. Loppujen lopuksihan se vaikuttaa jo täysin mahdolliselta monien selväpäisten pukuherrojenkin mielestä.

Sen sijaan lausunkin lopuksi opetuksen sanan, joka saattaa koskettaa ohjelmistojen ohella laajempaa ryhmää ja kaikkea luovaa tai ammattimaista työtä. Ihmiset yleensä pitävät miellyttävänä tehtäviä ja tavoitteita, jotka ovat haasteina optimialueella: eivät liian triviaaleja ja tylsiä, eivät liian vaikeita saavuttaa. Onnellinen ohjelmoija asuu ihmisessä, joka ei alisuoriudu tai jota ei ylikuormiteta väärin asetetuilla tavoitteilla ja stressiä aiheuttavilla työilmapiirin jännitteillä.

Jos suhtaudut työntekoon pelon- ja vihansekaisin tuntein tai edes ripustelet ulkopuolisen sarkastisesti B. Virtanen -sarjakuvaleikkeitä työpisteesi seinille, voit pitää sitä merkkinä siitä, että työn tekemisen prosessi on väärä ja epäonnistunut. Ilo, huumori ja leikinlasku ovat todellisia vahvuuksia, enkä suinkaan aiemmin kirjoittanut huvikseni "onnellisista uurastajista", eikä ole pelkkä vitsi, että Linuxin maskottieläin on pehmoileva ja lapsekas pingviini.

Avoimen lähdekoodin liikkeen saavutuksista suurimpia saattaa olla sen tosiasian havaitseminen, että leikki on taloudellisesti kaikkein tehokkain tapa tehdä luovaa työtä.

Luku 14

Jälkisanat: Netscape liittyy basaariin

On omituinen tunne tajuta auttavansa historian tekemisessä...

22. tammikuuta 1998, noin seitsemän kuukautta sen jälkeen kun julkaisin The Cathedral and the Bazaar -esseen, Netscape Communications julkaisi suunnitelmansa luovuttaa Netscape Communicator -selaimen lähdekoodi. Minulla ei ollut ollut mitään ennakkotietoa asiasta ennen julkaisupäivää.

Netscapen teknologia- ja varapääjohtaja Eric Hahn lähetti minulle vähän julkistuksen jälkeen seuraavansisältöisen sähköpostin: *Kaikkien puolesta täällä Netscapella haluan kiittää sinua siitä, että ylipäänsä pääsimme tähän pisteeseen. Ajatuksesi ja kirjoituksesi olivat keskeisiä innoituksenlähteitä päätöksellemme.*

Seuraavalla viikolla 4. helmikuuta 1998 lensin Netscapen kutsusta Piilaaksoon päivän kestäneeseen strategiapalaveriin joidenkin heidän ylimpien johtajien ja teknisten työntekijöidensä kanssa. Suunnittelimme Netscapen lähdekoodin julkistamisstrategian ja lisenssin yhdessä.

Muutamia päiviä myöhemmin kirjoitin seuraavasti:

Netscape tarjoaa meille suuren mittakaavan tilaisuuden kokeilla basaarimallia käytännössä kaupallisessa maailmassa. Avoimen lähdekoodin kulttuuria kohtaa nyt vaara: jos Netscape ei onnistu, avoimen lähdekoodin konsepti saattaa joutua niin huonoon valoon, ettei kaupallinen maailma koske siihen uudelleen toiseen vuosikymmeneen.

Toisaalta, tämä on myös mahtava tilaisuus. Ensireaktio siirtoon on ollut Wall Streetillä ja muualla varovaisen myönteinen. Meille on myös annettu tilaisuus punnita samalla oma merkityksemme. Jos Netscape säilyttää huomattavan markkinaosuuden tämän siirron yli, se voi saada aikaan kauan odotetun mullistuksen ohjelmistoteollisuudessa.

Seuraavan vuoden pitäisi olla erittäin opettavaista ja mielenkiintoista aikaa.

Ja se myös oli. Kuten kirjoitin vuoden 2000 keskivaiheilla, sittemmin Mozillaksi nimetty kokeilu on ollut odotukset täyttänyt menestys. Se saavutti Netscapen alkuperäisen päämäärän, joka oli estää Microsoftia luomasta monopolia selainmarkkinoille. Se on myös yltänyt muihinkin dramaattisiin saavutuksiin, joista huomattavimpana seuraavan sukupolven Gecko-kuvantamismoottorin julkaiseminen.

Se ei kuitenkaan ole vielä haalinut sellaista massiivista kehityspanostusta Netscapen ulkopuolelta jota Mozillan perustajat olivat alunperin toivoneet. Ongelma tässä näyttäisi olevan se, että Mozillan jakelu itse asiassa rikkoi pitkän aikaa yhtä basaarimallin perussäännöistä: Siinä ei tullut mukana sellaista, mitä mahdolliset avustajat voisivat helposti ajaa ja nähdä

toiminnassa. (Yli vuoden julkaisemisen jälkeen Mozillan kääntäminen lähdekoodista vaati lisenssin kaupalliseen Motif-kirjastoon.)

Kaikkein kielteisimpänä (ulkopuolisen maailman näkökulmasta), Mozilla-ryhmä ei julkaissut tuotantokäyttöön laadultaan sopivaa selainta kahden ja puolen vuoden aikana projektin aloittamisen jälkeen. Vuonna 1999 yksi projektin päätekijöistä aiheutti kohua eroamalla, valittaen huonoa johtamista ja ohitettuja tilaisuuksia. *Avoin lähdekoodi*, hän huomautti osuvasti, *ei ole mitään taikapölyä*.

Eikä se tosiaan ole. Mozillan pitkän aikavälin ennuste näyttää merkittävästi paremmalta nyt (marraskuussa 2000) kuin mitä se näytti Jamie Zawinskin eroamiskirjeen aikoihin. Viime viikkoina yölliset julkaisut ovat vihdoin läpäisseet kriittisen rajan tuotantokäyttöön soveltuvuuteen. Mutta Jamie oli oikeassa osoittaessaan että avoimuuteen meneminen ei välttämättä pelasta olemassaolevaa projektia joka kärsii väärin määritellyistä tavoitteista tai spagettikoodista tai muista ohjelmistotekniikan kroonisista vaivoista. Mozilla on onnistunut tarjoamaan samanaikaisesti esimerkin sekä siitä, miten avoin lähdekoodi voi onnistua ja miten se voisi epäonnistua.

Tällä välin avoimen lähdekoodin idea on kuitenkin ollut menestys ja löytänyt tukijoita muualta. Netscapen julkaisun jälkeen olemme nähneet suunnattoman, räjähdymäisen kasvun kiinnostuksessa avoimen lähdekoodin kehitysmallia kohtaan – trendiä, jota Linuxkäyttöjärjestelmä ajaa ja joka ajaa Linux-käyttöjärjestelmän jatkuvaa menestystä. Mozillan käynnistämä trendi jatkaa kiihtyvällä vauhdilla.

Luku 15

Huomautukset

Luku 16

Kirjallisuutta

Suomenkielisen laitoksen lisäys: *Eric Steven Raymond The Cathedral and the Bazaar* -verkkoessee.

Esseessäni on useita lainauksia Frederick B. Brookin hyvin tunnetusta kirjasta *The Mythical Man-Month*, koska monessa suhteessa hänen ajatuksensa eivät ole vieläkään kohdanneet voittajaansa. Suosittelen sydämellisesti Addison-Wesleyn julkaisemaa 25-vuotisjuhlalaitosta (ISBN 0-201-83595-9), jossa on liitteenä Brookin 1986 lisäämä *No Silver Bullet*-osio.

Uudessa laitoksessa on korvaamaton 20-vuotta myöhemmin -ajantasakatsaus, jossa Brooks avoimesti myöntää, että aivan kaikki hänen alun perin kirjoittamansa ei ole kestänyt aikaa. Luin katsauksen vasta kun tästä esseestä oli julkaistu ensimmäinen julkinen versio, ja hämmästyin, että Brooks oli pannut basaarityyliset kokeilut Microsoftin nimiin! Itse asiassa tämä sidos osoittautui virheeksi. 1998 saimme kuulla Halloween-asiakirjavuodosta, että Microsoftin sisäinen kehittäjäyhteisö on pahasti eripurainen ja balkanisoitunut, jonka sisällä basaarimalli ei olisi edes teoriassa mahdollinen.

Gerald M. Weinbergin *The Psychology Of Computer Programming* (New York, Van Nostrand Reinhold 1971) esitteli melko onnettomasti nimetyn käsitteen "egoton ohjelmointi". Vaikka hän ei ole niiden joukossakaan, jotka ensimmäisinä huomasivat komentojohtamisen mielettömyyden, oli hän luultavasti ensimmäinen, joka keksi yhdistää sen ohjelmistosuunnitteluun ja otti asian puheeksi tässä yhteydessä.

Richard P. Gabriel, joka pohdiskeli Unix-maailman kulttuuria ennen Linuxia, vastahakoisesti joutui puolustelemaan alkukantaista basaarityyliä muistuttavaa toimintamallia vuonna 1989 kirjoittamassaan esseessä *LISP: Good News, Bad News, and How To Win Big*. Vaikka teos on nykyään osin jo vanhentunut, esseellä on silti edelleen oma kunniapaikkansa LISP-fanien kirjahyllyssä (kuten minun). Eräs minulle kirjoittanut henkilö mainitsi minulle, kuinka luku nimeltä *Worse is Better* voidaan tulkita Linuxin tulemisen ennuskirjoitukseksi. Essee on saatavilla osoitteessa http://www.naggum.no/worse-is-better.html.

De Marcon ja Lister's Peoplewaren *Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) on aliarvostettu helmi, jonka näin ilahtuneena ansaitusti ottavan paikkansa Fred Brooksin katsauksessa. Vaikka hyvin vähän siitä on suoraan sovellettavissa Linuxiin tai avoimen lähdekoodin yhteisöjen tarpeisiin, on tekijän näkemys luovan työn ehdoista edelleen ajankohtainen ja lukemisen väärti kaikille, jotka haluavat ammentaa basaarimallista ja sen hyveistä kaupallisessa ympäristössä.

Lopuksi minun on pakko myöntää, että oli sillä hilkulla, etten antanut esseelleni nimeä

Katedraali ja agora. Agorahan on kreikkalainen kattamaton tori ja kohtaamispaikka. *Agoric Systems* -essee, joka on Mark Millerin ja Eric Drexlerin tärkeä teos, kuvaa nousevien markkinahenkisten tietokoneyhteisöjen piirteitä tavalla, joka sai minut harkitsemaan uudelleen vastaavia ilmiöitä avoimen lähdekoodin ympäristössä, kun Linux heilutteli niitä nenäni edessä viisi vuotta myöhemmin. Esseekokoelma on saatavilla osoitteesta http://www.agorics.com/agorpapers.html (sivusto ei enää toimi, *suomentajan huomautus*).

Luku 17

Kiitokset

Tätä esseetä paransivat keskustelut monien sellaisten henkilöiden kanssa, jotka auttoivat sen virheenkorjauksessa. Erityiskiitokset Jeff Dutkylle, joka ehdotti "virheiden etsintä on rinnakkaistettavissa-muotoilua ja auttoi kehittämään siitä kehittyvää analyysiä. Myös Nancy Lebovitzille hänen ehdotuksestaan, että matkin Weinbergiä lainaamalla Kropotkinia. Tarkkanäköistä kritiikkiä tuli myös Joan Eslingeriltä ja Marty Franzilta General Technics -listalta. Glen Vandenburg osoitti itsevalinnan tärkeyden kehittäjäyhteisöissä ja ehdotti hedelmällistä ideaa että suuri osa ohjelmistokehityksestä korjaa "laiminlyöntien bugeja"; Daniel Upper ehdotti luontoaiheisia analogioita tähän. Olen kiitollinen PLUG:in, Philadelphia Linux User's Groupin, jäsenille siitä, että he tarjosivat ensimmäisen testiyleisön ensimmäiselle tämän esseen julkiselle versiolle. Paula Matuszek valaisi minua ohjelmistohallinnan käytännöistä. Phil Hudson muistutti minua että hakkerikulttuurin sosiaalinen organisaatio peilaa sen ohjelmiston järjestymistä ja päin vastoin. John Buck huomautti että MATLAB tekee instruktiivin vastaavasti kuin Emacs. Russell Johnston sai minut tietoiseksi joistain mekanismeista, joita käsitellään "Kuinka monta silmäparia tarvitaan kesyttämään monimutkaisuus-kohdassa. Ja lopuksi, Linus Torvaldsin kommentit olivat hyödyllisiä ja hänen aikainen hyväksyntänsä erittäin rohkaisevaa.

Luku 18

Suomentajat

- Aapo Rantalainen
- Toni Alenius
- Artti Jaakkola
- Tarmo Toikkanen
- Jyri-Petteri Paloposki
- Petri Salo
- Jarkko Iso-Heiko
- Joonas Järnstedt
- Ari Takalo
- Tuomas Rantalainen
- Mauri Latvala
- Heikki Mäkinen
- Heikki Mäntysaari
- Mikko Piippo
- Hannu Makarainen
- Jyrki Palttala

18.1 Kiitokset

- Eric S. Raymond - hyvästä kirjasta