A katedrális és a bazár

 Magyar terminológiai kérdésekkel kapcsolatban értékes segítséget nyújtott a HWSW.hu fórumából lacos és Supposed Former Infatuation Junkie. A fordítás verzióadatai: $Revision: 0.16 $, $Date: 2003/11/07 10:15:29 $

Tartalomjegyzék

- Kivonat
- A katedrális és a bazár
- A levélnek át kell jutnia
- Felhasználóink jelentősége
- Korai, gyakori kiadások
- A sok szem csökkenti a komplexitást
- Nem fenékig tejföl
- A popclientből fetchmail lesz
- A fetchmail felnő
- További fetchmail-tapasztalatok
- A bazár szükséges előfeltételei
- A nyílt forráskódú szoftver szociális kontextusa
- A vezetésről és a Maginot-vonalról
- Epilógus: a Netscape és a bazár
- Jegyzetek
- Bibliográfia
- Köszönetnyilvánítás

Kivonat

A fetchmail nevű, sikeres nyílt forráskódú projektet boncolgatom, amely tudatos kísérlet volt a Linux történetéből leszűrhető meglepő szoftvertervezési elméletek tesztelésére. Ezeket az elméleteket két különböző fejlesztési stílus mentén fejtem ki, a kereskedelmi világ katedrális modellje, illetve a vele szemben álló linuxos világ bazár modellje mentén. Bemutatom, hogy ezek a modellek a szoftverhiba-keresési folyamat természetére vonatkozó ellentétes feltételezésekből származnak. A linuxos tapasztalatok alapján amellett érvelek, hogy „elég sok szem mellett minden hiba jelentéktelenné válik”; termékeny analógiákat javaslok más, önző ágensekből álló önjavító rendszerekkel, majd ezen meglátások következményeinek a szoftverek jövőjére gyakorolt hatásával fejezem be.

A katedrális és a bazár

A Linux felforgató. Ki gondolta volna még csak öt évvel ezelőtt is (1991), hogy a bolygón szétszórt, csupán az internet finom fonalával összekötött sok ezer fejlesztő részidős bütykölése, mintegy varázsütésre, világszínvonalú operációs rendszerré egyesül?

Én biztosan nem. Mire feltűnt az érzékelőimen a Linux 1993-ban, már benne voltam a Unix-fejlesztésekben és a nyílt forráskódú fejlesztésekben tíz éve. Az első GNU-közreműködők

egyike voltam a nyolcvanas évek közepén. Jelentős mennyiségű nyílt forráskódú szoftvert tettem közzé a hálón, fejlesztettem és társfejlesztettem számos programot (nethack, az Emacs VC és GUD üzemmódjai, xlife és egyebek), amelyek még ma is széles körben használtak. Azt hittem, hogy tudtam, hogyan készültek.

A Linux felborított sok dolgot, amelyről úgy gondoltam, ismerem. Hirdettem a kis eszközök, a gyors modellalkotás és a lépésenkénti fejlődést elősegítő programozás unixos igéjét évekig. Ugyanakkor abban is hittem, hogy létezik egy bizonyos összetettség, amely fölött egy centralizáltabb, a priori megközelítés szükséges. Hittem benne, hogy a legfontosabb szoftverek (az operációs rendszerek és az olyan igazán nagy eszközök, mint az Emacs programozói szerkesztő) szükségszerűen a katedrálisokhoz hasonlóan épülnek, egyéni varázslók által, óvatosan ügyeskedve, vagy mágusokból álló, elszigetelt kis csoportok által, idő előtti bétaverziók nélkül.

Linus Torvalds fejlesztői stílusa – adj ki korán és gyakran, adj ki mindent, amit csak tudsz, a kuszaságig légy nyílt – meglepetésként ért. Nem volt itt semmiféle csöndes, tiszteletteljes katedrálisépítés, a Linux-közösség a különféle tennivalók és megközelítések nagy, fecsegő bazárjára hasonlított (ezt leginkább azok a Linux-archívumok jelképezik, amelyek *bárkitől* elfogadják a beküldött dolgokat), amelyből egy koherens és stabil rendszer látszólag csak valami csoda folytán születhetne.

Az a tény, hogy a bazár stílus működni látszott, és nem is rosszul, határozottan sokkoló volt. Ahogy jártam az utamat, nemcsak egyéni projekteken dolgoztam keményen, hanem annak a megértésén is, hogy a linuxos világ miért veszi olyan jól az akadályokat a katedrálisépítők számára aligha elképzelhető sebességgel, ahelyett, hogy egyszerűen darabjaira esne.

1996 közepére úgy gondoltam, hogy kezdem érteni. Tökéletes lehetőségem nyílt az elméletem tesztelésére egy nyílt forráskódú projekt keretében, amit megpróbálhattam a bazár módszerével fenntartani. Meg is tettem, és hatalmas siker lett belőle.

Mindez annak a bizonyos projektnek a története. Felhasználom, hogy néhány fontos dolgot állíthassak a hatékony nyílt forráskódú fejlesztésről. Ezek közül némelyikkel nem a linuxos világban találkoztam először, de majd meglátjuk, hogyan ad nekik a linuxos világ különös jelentőséget. Ha nem tévedek, segíteni fognak annak a pontos megértésében, hogy mi teszi a Linux-közösséget olyan termékennyé a jó szoftverek terén, és talán abban is segít, hogy te magad termékenyebb legyél.

A levélnek át kell jutnia

1993 óta felelős vagyok egy kicsi, ingyenes hozzáférést biztosító, Chester County InterLink (CCIL) nevű internetszolgáltató műszaki üzemeltetésért a pennsylvaniai West Chesterben. A CCIL társalapítójaként megírtam az egyedi, többfelhasználós hirdetőtábla-szoftverünket, ami a locke.ccil.org-ra betelnetelve megnézhető. Mára már csaknem háromezer felhasználóval működik, harminc vonalon. A munka lehetővé tette számomra a napi 24 órás hálózati hozzáférést a CCIL 56K-s vonalán, sőt egyenesen megkövetelte.

Igencsak hozzászoktam az azonnali internetes e-mailezéshez. Bosszantónak találtam telnettel periodikusan bejelentkezni a locke-ra, hogy megnézzem a leveleim. Azt szerettem volna, hogy a levelek a snarkra (az otthoni rendszeremre) érkezzenek, így értesülhetnék kézbesítésükről, és a saját eszközeimmel kezelhetném őket.

Az interneten használt levéltovábbító protokoll, az SMTP (Simple Mail Transfer Protocol) azért nem felelt meg, mert az akkor működik optimálisan, ha a gépek folyamatosan a hálózatra vannak kapcsolva, miközben az én számítógépem nem volt állandóan az interneten, és nem volt statikus IP-címe sem. Szükségem volt egy programra, amely az időszakosan felépített betárcsázós kapcsolaton keresztül behúzza a leveleket helyi kézbesítés céljából. Tudtam, hogy ilyen dolgok léteznek, és hogy legtöbbjük a POP (Post Office Protocol) nevű egyszerű alkalmazásprotokollt használja. A POP-ot ma a legtöbb e-mail kliens támogatja, de abban az időben nem volt beépítve a levélolvasó programba, amit használtam.

Szükségem volt egy POP3 kliensre. Szétnéztem az interneten, és találtam egyet. Valójában hármat vagy négyet találtam, használtam őket egy ideig, de hiányoltam a bejövő levelekben szereplő címek megpiszkálásának egyébként nyilvánvaló lehetőségét, amellyel az elhozott levélre adott válasz funkciója is jól működött volna.

A probléma a következő volt: ha valaki a locke-ról „joe” néven küldött nekem levelet, akkor a snarkra átmásolt levélre adandó választ a levelezőprogramom boldogan elküldte volna a snarkon nem is létező „joe” számára. A válaszcím kézzel való szerkesztése, és átirányítása a @ccil.org-ra hamarosan gyötrelmessé vált.

Ez egyértelműen olyan dolog volt, amit a számítógépnek kellett volna megtennie, de egyetlen létező POP kliens sem tudta hogyan. Ez vezet minket az első tanulsághoz:

1. Minden jó szoftver egy fejlesztő személyes vágyainak kielégítésével kezdődik.

Ennek bizonyára nyilvánvalónak kellett volna lennie (ahogy a mondás tartja: a szükség találékonnyá tesz), ám a szoftverfejlesztők túl gyakran töltik az idejüket olyan lélekölő, de pénzt hozó programokkal, amelyekre nincs szükségük és amelyek nem is érdeklik őket. Másképp van ez a Linux világában, ami megmagyarázhatja, hogy miért olyan jó minőségűek a Linux-közösséghez visszavezethető szoftverek.

Vajon őrült pörgésbe kezdtem-e azonnal, hogy lekódoljak egy teljesen új POP3 klienst, hogy versengjek a már létezőkkel? Semmi esetre sem. Alaposan megvizsgáltam a kezem ügyébe kerülő POP segédprogramokat és megkérdeztem magamtól, hogy melyik áll legközelebb ahhoz, amit szeretnék. Mert:

2. A jó programozók tudják mit írjanak. A nagyok azt is tudják, mit írjanak (és használjanak) újra.

Bár nem tartom magam nagy programozónak, azért megpróbálok úgy tenni. A nagyok fontos jellemzője a konstruktív lustaság. Tudják, hogy a legjobb jegyet nem a szándékra, hanem az eredményre adják, és majdnem mindig egyszerűbb továbblépni egy jó részmegoldásról, mint a semmiről kezdeni.

Linus Torvalds például nem az alapoktól kezdte el a Linux írását, hanem a Minix, egy kicsi, PC-re írt Unix-szerű operációs rendszer kódját és ötleteit használta fel újra. Végül minden Minix-kód eltűnt vagy teljesen át lett írva, de amíg a helyükön voltak, segítették azt a kezdeményt, ami később Linuxszá vált.

Ennek szellemében egy létező POP segédprogramot kerestem, amit elég jól kódoltak ahhoz, hogy a fejlesztés alapjául használhassam fel.

A unixos világ forráskód-megosztással kapcsolatos hagyományai mindig kedveztek a kódújrahasznosításnak (ezért választotta a GNU Projekt is a Unixot alap OS-nek, az OS súlyos megszorításai ellenére is). A linuxos világ ezt a hagyományt majdnem a technológiailag lehetséges határokig elvitte, terabájtnyi nyílt forráskódot téve elérhetővé. Ezért mások majdnem jó megoldásainak keresése valószínűleg eredményesebb lehet a linuxos világban, mint bárhol máshol.

Eredményes is volt. A korábbiakkal együtt a második keresés után kilenc jelöltem volt: a fetchpop, a PopTart, a get-mail, a gwpop, a pimp, a pop-perl, a popc, a popmail és az upop. Elsőként a Seung-Hong Oh által írt fetchpop mellett tettem le a voksomat. Kibővítettem a fejlécátíró funkcióval és egyéb javításokkal, amelyeket a szerző az 1.9-es kiadásban át is vett.

Néhány hét múlva azonban belebotlottam Carl Harris popclientjének kódjába, és rájöttem, hogy ez így nem lesz jó. Bár a fetchpopban volt néhány igazán eredeti ötlet (például a háttérben, démonként való futás), csak a POP3-at tudta kezelni, és eléggé amatőr módon volt kódolva (Seung-Hong akkoriban egy okos, de tapasztalatlan programozó volt, mindkettő érezhető volt). Carl kódja jobb, eléggé professzionális és megbízható volt, de a programból számos fontos és nehezen implementálható fetchpop-tulajdonság hiányzott, beleértve azokat is, amelyeket már én kódoltam.

Maradjak vagy váltsak? Ha váltanék, eldobnám az addigi munkámat egy jobb fejlesztési alapért.

A több protokoll támogatásának megléte a váltást motiválta. A POP3 a leggyakrabban használt postafiók-protokoll, de nem az egyetlen. A fetchpop és a többi vetélytárs nem tudta a POP2-t az RPOP-ot vagy az APOP-ot, és időnként voltak olyan gondolataim, hogy talán szórakozásból hozzáadhatnám az IMAP-ot is (Internet Message Access Protocol, a legfrissebb tervezésű és legerőteljesebb postafiók-protokoll).

De voltak elméleti okai is annak, hogy a váltás jó ötlet lehet, ezt még jóval a Linux előtt tanultam.

3. „Tervezd be, hogy egyet el fogsz dobni, úgyis el fogod” (Fred Brooks, *The Mythical Man-Month*, 11. fejezet)

Avagy másképp fogalmazva: gyakran nem is érted a problémát egészen addig, amíg először nem készítesz rá megoldást. A második alkalommal talán már eleget tudsz ahhoz, hogy jól csináld. Ha tehát jól akarod csinálni, készülj fel arra, hogy *legalább egyszer* újra kell kezdened [JB].

Nos, azt mondtam magamnak, hogy a fetchpop megváltoztatása volt az első próbálkozásom, ezért váltottam.

Miután elküldtem a popclienthez készült első patchgyűjteményemet Carl Harrisnek 1996 június 25-én, rájöttem, hogy ő valamivel előtte már elvesztette a popclient iránti érdeklődését. A kód már porosodott, kisebb hibái is voltak. Sok változtatást szerettem volna végrehajtani benne, így gyorsan megállapodtunk, hogy a logikus lépés az lesz, ha átveszem a programot.

Mielőtt észrevettem volna, a projekt már be is indult. Már nem a létező POP kliensek kisebb

javításaival foglalkoztam, hanem átvettem egynek a karbantartását, gondolatokat forgattam a fejemben, amelyekről tudtam, hogy fontos változásokat fognak eredményezni.

Egy szoftverkultúrában, amely bátorítja a forráskód-megosztást, ez a projektfejlődés természetes útja. A következő alapelvet vittem át a gyakorlatba:

4. Megfelelő attitűd mellett az érdekes problémák megtalálnak.

De Carl Harris hozzáállása ennél is fontosabb volt. Ő megértette azt, hogy:

5. Ha már nem érdekel egy program, az utolsó kötelességed átadni azt egy kompetens utód számára.

Anélkül, hogy valaha is meg kellett volna beszélnünk, Carl és én tudtuk, hogy közös célunk a létező legjobb megoldás létrehozása. Az egyetlen kérdés az volt, hogy vajon be tudom-e bizonyítani, én vagyok az alkalmas személy. Amint ez megtörtént, ő jóindulatúan és gyorsan cselekedett. Remélem, hogy én is ezt fogom tenni, ha egyszer majd rajtam lesz a sor.

Felhasználóink jelentősége

Így történt, hogy megörököltem a popclientet, és legalább ennyire fontos, hogy örököltem a felhasználói bázisát is. Nagyszerű dolog, ha felhasználóid vannak, nemcsak azért, mert mutatják, hogy szükségletet elégítesz ki, vagy mert valami jót teszel. Helyesen nevelve őket, társfejlesztőkké is válhatnak.

A Unix tradíció másik erőssége, ami a Linuxnál ismét szélsőség, hogy sok felhasználó egyben hacker is, és mivel a forráskód hozzáférhető, ezek a hackerek *hatékonyak* lehetnek, ami leírhatatlanul hasznos lehet a hibakeresésre fordított idő csökkentésében. Egy kis bátorítással a felhasználók diagnosztizálják a problémákat, javaslatokat tesznek a javításra, és segítenek a kód annál jelentősen gyorsabb továbbvitelében, mint amire a segítségük nélkül lennél képes.

6. A gyors kódfejlesztés és a hatékony hibakeresés felé vezető legkönnyebb út a felhasználók társfejlesztőként való kezelésén át vezet.

Ennek a hatásait könnyű alábecsülni. Valójában a nyílt forráskódú világban jóformán mindannyian drasztikusan alábecsültük, hogy milyen viszonyban áll ez a felhasználók számával és a rendszerek bonyolultságával, amíg Linus Torvalds be nem mutatta az ellenkezőjét.

Úgy gondolom, Linus legokosabb és legkövetkezetesebb tette nem a Linux kernel felépítése volt, hanem a linuxos fejlesztési modellre való rátalálás. Amikor kifejtettem ezen véleményem a jelenlétében, elmosolyodott, és csöndben megismételte azt, amit gyakran elmond: „alapvetően nagyon lusta vagyok, aki szereti mások helyett learatni a babérokat”. Okos. Vagy ahogy Robert Heinlein írja az egyik karakteréről: túl lusta ahhoz, hogy kudarcot valljon.

Visszatekintve a Linux módszereinek és sikerességének egy példáját találjuk a GNU Emacs Lisp programkönyvtár és a Lisp kódarchívum fejlesztésében. Az Emacs C nyelven írt magjának, illetve az egyéb GNU eszközöknek a katedrálisépítő stílusával ellentétben a Lisp

kódkészlet rugalmas fejlődését a felhasználók irányították. Az ötleteket és az üzemmódok prototípusait gyakran háromszor-négyszer is újraírták, mielőtt elnyerték végső, megbízható formájukat. Az internet segítségével létrejövő laza együttműködés, ahogy a Linuxnál, gyakori volt.

A fetchmail előtti legsikeresebb alkotásom talán az Emacs VC (verziókövetés, version control) módja volt, egy Linux-szerű, e-mailen keresztüli együttműködés három másik emberrel, mind a mai napig csak az egyikükkel (Richard Stallmannel, az Emacs szerzőjével és a Free Software Foundation alapítójával) találkoztam. A VC mód egy előtét (frontend) volt az SCCS-hez, az RCS-hez, majd később a CVS-hez, amelyben az Emacs egyszerűen tudott végezni verziókövetési műveleteket. Egy apró, durva, más által írt sccs.el nevű üzemmódból fejlődött ki. A VC fejlesztése azért volt sikeres, mert az Emacs-től eltérően az Emacs Lisp kódja nagyon gyorsan volt képes átmenni a kiadás-tesztelés-fejlesztés ciklusokon.

Az Emacs története nem egyedi, más szoftvertermékek is léteznek kétszintű architektúrával és kétfelé kötődő felhasználói közösséggel, amely egyesíti a katedrálisként épülő magot és a bazárként fejlődő kiegészítőket. Az egyik ilyen a MATLAB, egy kereskedelmi adatelemző és -megjelenítő programrendszer. A MATLAB és egyéb, hasonló szerkezetű termékek felhasználói egyöntetűen arról számolnak be, hogy a mozgás, forgolódás, innováció leginkább az eszközök nyílt részében történik, ahol a nagy és változatos közösség kedvére barkácsolhat.

Korai, gyakori kiadások

A Linux fejlesztési modelljének kritikus részei a korai és gyakori kiadások. A legtöbb fejlesztő (köztük én is) úgy gondolta, hogy ez hibás vezérelv a triviális feladatoknál nagyobb projektek esetén, mert a korai verziók majdnem definíció szerint hibásak, és senki nem akarja ilyesmivel próbára tenni a felhasználók türelmét.

Ez a hit erősítette meg a katedrálisépítés-szerű modell melletti elkötelezettséget. Ha a kiemelt cél az, hogy a felhasználók minél kevesebb hibát lássanak, miért adnál ki félévenként új verziót, és dolgoznád magad halálra a kiadások közti hibakeresés során? Inkább legyen hosszabb a kiadási ciklus. Az Emacs C magját így fejlesztették. A Lisp könyvtárat gyakorlatilag nem, ugyanis az FSF (Free Software Foundation) hatókörén kívül is léteztek Lisp archívumok, ahol az Emacs kiadásaitól független új, illetve fejlesztői kódváltozatokat lehetett találni [QR].

A legjelentősebb az Ohio State Emacs Lisp archívum volt, amely a mai nagy Linux-archívumok hangulatát és tulajdonságait vetítette előre. Néhányunk azonban elgondolkodott azon, mit is csinálunk, illetve azon, hogy ennek az archívumnak a léte az FSF katedrálisépítő modelljében lévő problémákra mutat rá. 1992 körül volt egy komoly kísérletem, hogy az ohioi kódot formálisan is beolvasszam a hivatalos Emacs Lisp könyvtárba, de problémákba ütköztem, rendkívül sikertelen kísérlet volt.

Egy évvel később, amikor a Linux szélesebb körben is láthatóvá vált, nyilvánvaló volt, hogy valami más, valami sokkal egészségesebb történik arrafelé. Linus nyílt fejlesztési irányelve a katedrálisépítés szöges ellentéte volt. Gombamód szaporodtak a linuxos internetes archívumok, többféle disztribúció keringett, amelyek mögött egy soha nem tapasztalt gyakoriságú alaprendszer-kiadás állt.

Linus a lehető leghatékonyabb módon kezelte társfejlesztőként a felhasználóit:

7. Adj ki korán. Adj ki gyakran. És figyelj a fogyasztóidra.

Linus innovációja nem igazán a gyors ciklusokról szólt, amelyekbe a felhasználói visszajelzések is beépültek (valami ilyesminek a Unix világában már hosszú hagyománya volt), hanem arról, hogy a fejlesztés bonyolultságával arányban álló intenzitással csinálta. Azokban a korai időkben (1991 körül) nem volt szokatlan, hogy *egyetlen nap* alatt több új kernelt is kiadott. Mivel kitermelte a társfejlesztőit, és mindenki másnál erőteljesebben használta az internetet az együttműködésre, a dolog működött.

De *hogyan* működött? Valami olyasmi volt, amit én is lemásolhatnék? Vagy mindez Linus Torvalds kivételes zsenijére támaszkodott?

Nem hinném. Elismerem, Linus átkozottul tehetséges hacker. Közülünk vajon hányan lennének képesek egy teljes, minőségi operációs rendszer magjának megtervezésére az alapoktól? A Linux azonban nem jelentett semmilyen nagyobb horderejű elméleti előrelépést. Linus nem (vagy legalábbis még nem) olyan innovatív tervezőzseni, mint mondjuk Richard Stallman vagy James Gosling (NeWS és Java). Linus számomra inkább a tervezés és a megvalósítás géniuszának tűnik, egyfajta hatodik érzékkel a hibák és fejlesztési zsákutcák elkerülésére, és rendkívüli ügyességgel az A-ból B pontba vezető könnyű utak megtalálására. A teljes Linux konstrukció ezt sugallja, híven tükrözi Linus mértéktartó és egyszerűsítő tervezési megközelítéseit.

Ha tehát a gyors kiadások és az internetes közeg megragadása nem véletlen volt, hanem Linus tervezőtehetségéből fakadó meglátás, amellyel felismerte a legkisebb erőkifejtést igénylő utat, akkor mit maximalizált? Mit hozott ki a szerkezetből?

Linus folyamatosan ösztönözte és jutalmazta a hackereit/felhasználóit. Ösztönözte őket az önbecsülésüket kielégítő cselekvések távlataival, és jutalmazta őket azzal, hogy látták, állandóan (akár *naponta*) fejlődik munkájuk.

Linus közvetlenül megcélozta, hogy minél több munkaóra jusson a hibakeresésre és a fejlesztésre, még a kódban lévő instabilitás és a felhasználói bázis esetleges konok hibákon való kiégése árán is. Úgy viselkedett, mintha valami ilyesmiben hinne:

8. Elegendően sok bétateszter és társfejlesztő mellett majdnem minden probléma gyorsan felismerhető, és a javítás is nyilvánvaló valaki számára.

Ugyanez lazábban: elegendő mennyiségű szem mellett minden hiba jelentéktelenné válik. Ezt nevezem Linus törvényének.

Eredetileg úgy fogalmaztam, hogy minden probléma „világos lesz valaki számára”. Linus közbevetette, hogy általában nem, vagy nem feltétlenül az az ember érti a problémát, és ad rá megoldást, aki először felismeri: „Valaki megtalálja a problémát és valaki *más* pedig megérti. Fel fogják jegyezni rólam, hogy azt mondtam: ráakadni a nagyobb kihívás”. Ez a kiigazítás fontos, a következő fejezetben, ahol a hibakeresés gyakorlatát vizsgáljuk részletesebben, majd kiderül, hogy miért. A lényeg, hogy a folyamat mindkét része (a felfedezés és a javítás) gyorsan megtörténhet.

Szerintem Linus törvényében testesül meg a katedrálisépítő és a bazári stílus mögött lévő

legfontosabb különbség. A programozás katedrálisépítő megközelítésében a hibák és fejlesztési problémák ravasz, alattomos, mély jelenségek. A kiválasztott kevesek hónapokig tartó alapos vizsgálatába kerül eljutni a megbízhatóságnak arra fokára, amikor úgy érzik, kigyomlálták őket. Ezért hosszúak a kiadási ciklusok, és ezért az elkerülhetetlen csalódottság, ha a régóta várt kiadás nem tökéletes.

A bazárban viszont feltételezed, hogy a hibák általában felszíni jelenségek, vagy legalábbis elég gyorsan felszíni jelenséggé válnak az egyes kiadásokhoz kapcsolódó ezernyi lelkes társfejlesztő zsivaja mellett. Éppen ezért gyakran készítesz kiadásokat, hogy még több javítást kapj, és mindez azzal a remek mellékhatással jár, hogy sokkal kevesebbet veszítesz, ha alkalmasint valami tákolmány is kikerül az utcára.

Ennyi. Ez már elég. Ha Linus törvénye hamis, akkor bármilyen rendszer, amely a kernelhez hasonlóan bonyolult, és amelyen annyi kéz dolgozott, mint a kernelen, egy ponton össze kellene, hogy omoljon az előre nem látott hibás interakciók és a felfedezetlen mély hibák súlya alatt. Ha viszont a törvény igaz, akkor megfelelő magyarázat lehet a Linux relatív hibátlanságára és a hónapoktól egészen az évekig ívelő folyamatos üzemelési idejére.

Talán ez nem is kell, hogy meglepetést okozzon. Évekkel ezelőtt szociológusok arra jöttek rá, hogy egyformán képzett (vagy egyformán tudatlan) megfigyelőkből álló csoportok véleményének középértéke sokkal megbízhatóbb előrejelzést ad, mint a véletlenszerűen kiválasztott megfigyelők vélemények. A jelenséget *Delphoi-effektusnak* nevezték el. Úgy tűnik, hogy Linus Torvalds megmutatta, hogy ez még az operációs rendszerek hibakeresésére is érvényes, hogy a Delphoi-effektus elnyomhatja a fejlesztés bonyolultságát, és még egy operációs rendszer magjának komplexitását is megszelídítheti [CV].

A Linux helyzetének egy speciális sajátossága, amely egyértelműen segíti a Delphoi-effektust, az, hogy a projektek munkatársai önkéntesek. Egy korai olvasóm mutatott rá arra, hogy a munkatársak nem véletlen mintából származnak, hanem olyan emberek, akiket érdekel annyira a szoftver használata, hogy megtanulják hogyan működik, és megpróbáljanak megoldást keresni a felmerülő problémákra, sőt nyilvánvalóan elfogadható javításokat is létrehoznak. Aki átjut ezeken a szűrőkön, annak feltehetően vannak hasznos ötletei, amelyekkel hozzájárulhat a munkához.

Linus törvénye úgy is újrafogalmazható, hogy „a hibakeresés párhuzamosítható”. Noha a hibakeresés megköveteli a hibakeresőktől, hogy valamilyen koordináló fejlesztővel kommunikáljanak, nem követeli meg a jelentős koordinációt a hibakeresők között. Ezért nem is esik áldozatául annak a bonyolultságnak és irányítási költségnek, amely az újabb fejlesztők felvételekor jelentkezik [a zárt kódú projektek esetében].

A hibakeresők által duplán elvégzett munkából származó elméleti veszteség a gyakorlatban szinte soha nem gond a Linux világában. A korai és gyakori kiadás irányelvének egyik hatása, hogy a visszajelzésekből származó javításokkal minimalizálja az ilyen jellegű duplikációt [JH].

Brooksnak (a *The Mythical Man-Month* szerzőjének) ezzel kapcsolatban még egy spontán megfigyelése is van: „Egy széles körben használt program karbantartásának teljes költsége általában a kifejlesztés költségeinek 40 százaléka vagy még több. Meglepő módon ez a költség erősen függ a felhasználók számától. *Több felhasználó több hibát talál*.”

Több felhasználó több hibát talál, mert több felhasználó többféle módon veszi igénybe a

programot. Ezt a hatást erősíti, ha a felhasználók társfejlesztők is. A hibafelismerést mindegyikük másként közelíti meg, némileg különböző módon érzékelve, más eszközökkel elemezve, más szögből. A Delphoi-effektus úgy tűnik, hogy pontosan ezen változatosság miatt működik. A hibakeresés kontextusában a variáció segít az azonos törekvések többszöri előfordulásának csökkentésében is.

Tehát több bétateszterrel nem csökken az éppen legkomolyabbnak tartott hiba mélysége a *fejlesztő* szemében, de megnő annak a valószínűsége, hogy lesz olyan személy, akinek a tarsolyában benne van *a számára* viszont könnyű probléma megoldása.

Linus pedig a legjobbat hozhatja ki a helyzetből: ha *valóban* komoly hibák vannak benne, akkor a kernelt olyan módon számozzák, hogy a potenciális felhasználók eldönthessék, az utolsó stabil változatnál maradnak, vagy vállalják a hibák kockázatát is az új lehetőségek miatt. Ezt a taktikát még nem minden fejlesztő vette át, de talán így a jó, a tény, hogy mindkét választás elérhető, mindkettőt vonzóbbá teszi [HBS].

A sok szem csökkenti a komplexitást

Más dolog nagy vonalakban megfigyelni, hogy a bazár módszere felgyorsítja a hibakeresést és kód fejlődését, és ugyancsak más pontosan megérteni, hogyan és miért működik így a mindennapi fejlesztői és tesztelői magatartás mikroszintjén. Ebben a fejezetben (amely három évvel az eredeti szöveg után íródik, és felhasználja olyan fejlesztők nézeteit, akik olvasták az eredetit, és megfigyelték saját viselkedésüket) alaposan megvizsgáljuk a tényleges mechanizmusokat. A nem műszaki érdeklődésű olvasók nyugodtan átugorhatnak a következő fejezetre.

A megértés egyik kulcsa annak a pontos felismerése, hogy a forráskódhoz nem jutó felhasználók hibabejelentései általában miért bizonyulnak kevésbé hasznosnak. A forráskódot nem ismerők csak a külső tüneteket szokták jelenteni, természetesnek veszik a saját környezetüket, így (a) lényeges háttérinformációkat hagynak ki, és (b) ritkán adnak megbízható receptet a hiba reprodukciójához.

Az alapprobléma itt a fejlesztő és a tesztelő programról alkotott mentális modelljének különbségéből adódik, a tesztelő kívülről néz be, míg a fejlesztő belülről néz ki. A zárt kódú fejlesztések során mindketten megragadnak ebben a szerepkörben, elbeszélnek egymás mellett, és mélységesen frusztrálónak találják egymást.

A nyílt forráskód feloldja ezt a kötöttséget, és sokkal könnyebbé teszi a tesztelő és a fejlesztő számára egy közös, a forráskódra építő ábrázolásmód kialakítását, illetve az arról való hatékony kommunikációt. A gyakorlatban óriási a különbség a fejlesztőre gyakorolt hatás tekintetében a csak külsőleg látható tünetek felsorolása, valamint az olyan hibabejelentés között, amely közvetlenül a fejlesztőben, a forráskód alapján kialakult mentális ábrázolásmódhoz kapcsolható.

A legtöbb hiba az esetek többségében könnyen megcsíphető egy részleges, de a hiba körülményeiről a forráskód szintjén sokatmondó jellemzéssel. Ha valaki a bétatesztelők közül rámutat arra, hogy egy elhatárolási (boundary) probléma van az nnn sorban, vagy csak annyit mond, hogy „X, Y és Z feltételek esetén ez a változó átfordul”, egy gyors pillantás a gyanús kódra gyakran elegendő a hiba pontos megállapításához, illetve a javítás elkészítéséhez.

A forráskód hozzáférhetősége mindkét oldalról nagymértékben segíti a kommunikációt, és kapcsolatot teremt bétatesztelő jelentései, valamint a vezető fejlesztők tudása között. Ráadásul ez azt is jelenti, hogy a vezető fejlesztők ideje gyakran még sok együttműködő esetén is megmarad.

A fejlesztői időt megőrző nyílt forráskódú módszer másik jellemzője az ilyen projektek kommunikációs struktúrája. Az imént a vezető fejlesztő kifejezést használtam, ez a projekt központját alkotók (rendszerint elég kevés, gyakran csak egy vagy egy-három fejlesztő), illetve a projekt holdudvarát alkotó munkatársak, bétatesztelők (több százan is lehetnek) megkülönböztetésére utal.

Az alapvető probléma, amivel a tradicionális szoftverfejlesztő szervezet küzd, Brooks törvénye: „Egy késésben lévő projekt bővítése újabb programozókkal további késést eredményez". Általánosabban megfogalmazva Brooks törvénye szerint egy projekt bonyolultsága és kommunikációs költsége a fejlesztők számával négyzetesen emelkedik, míg az elvégzett munka mennyisége csupán lineárisan nő.

Brooks törvényének alapjait az a tapasztalat adja, amely szerint a hibák a különböző emberek által írt az interfészeknél hajlamosak összegyűlni, valamint az, hogy egy projekt kommunikációs/koordinációs költsége az embereket elválasztó határfelületek számával szokott emelkedni. Ilyenformán a problémák a fejlesztők közötti kommunikációs útvonalak számával állnak arányban, amelyek viszont a fejlesztők számának négyzetével állnak arányban (pontosabban az N*(N-1)/2 képlet szerint alakulnak, ahol az N a fejlesztők száma).

Brooks törvényének elemzése (és az ebből következő félelem a nagy létszámú fejlesztőcsoportoktól) azon a burkolt feltételezésen nyugszik, hogy a projekt kommunikációs struktúrája szükségszerűen teljes gráf, és mindenki mindenki mással beszél. A nyílt forráskódú projektekben azonban a holdudvarhoz tartozó fejlesztők gyakorlatilag elkülöníthető, párhuzamos részfeladatokon dolgoznak, alig van köztük interakció, a kód változásai és a hibabejelentések a vezető fejlesztőkön keresztül folynak, és csak *ebben* a kis csoportban kell megfizetni a teljes Brooks-féle költséget [SU].

Vannak egyéb okai is a forráskód szintű hibabejelentés hatékonyságának, amelyek aköré szerveződnek, hogy egy hibának gyakran több lehetséges tünete van, és ezek a felhasználó által gyakorolt kezelési mintázat és a környezet függvényében változnak. Az ilyen hibák pontosan azok a komplex, körmönfont dolgok (például a dinamikusmemória-kezelési hibák vagy a nem determinisztikus megszakításablakokkal kapcsolatos leletek), amelyeket nehéz szándékosan reprodukálni vagy elemzéssel kiszúrni, és amelyek a szoftverek hosszú távú problémáiért leginkább felelősek.

Az a tesztelő, aki próbaképpen beküldi egy hiba forráskód szintű jellemzését (például: „Úgy tűnik, mintha lefedetlenség lenne a szignálkezelésben az 1250. sor körül" vagy „Hol nullázzátok ezt a puffert?"), a fejlesztőt – aki közvetlenül a kóddal dolgozik, és e közelség miatt a problémát egyébként észre sem venné – egy féltucat különböző tünet okára ébresztheti rá. Ilyen esetekben nehéz vagy egyenesen lehetetlen megtudni, hogy az egyes, külsőleg is látható rendellenességeket pontosan melyik hiba okozza, de a gyakori kiadások mellett ez nem is szükséges. Az együttműködők feltehetően gyorsan feltérképezik, hogy a hibájukat kijavították-e vagy sem. A forráskód szintű hibabejelentésekkel sokszor a tudatos javítások nélkül is kiszóródnak a rendellenességek.

A bonyolult, több tünetet okozó hibáknál gyakori, hogy több visszakövetési útvonal vezet a

tényleges hibáig. Egy adott fejlesztő vagy tesztelő csak néhány ilyen úton képes elindulni a környezetének finomságaitól függően, amely egyébként idővel jócskán változhat, és nem is feltétlenül szükségszerű módon. Gyakorlatilag minden fejlesztő és tesztelő egy félig véletlen mintát vesz a program állapotaiból a tünetek okainak kutatása közben. Minél ravaszabb, bonyolultabb egy hiba, annál kisebb az esélye annak, hogy az adott mintába fontos nyom kerül.

Az egyszerű, könnyen reprodukálható hibáknál a hangsúly nem a véletlenen van, sokat számít a hibakeresésben való jártasság és kód felépítésének ismerete. A komplex hibák estében viszont a véletlen a hangsúlyos. Ilyen feltételek mellett a sok ember által bejárt hibakeresési utak jóval hatékonyabbak, mint a kevés ember által egymás után végigkövetett nyomok – még akkor is, ha ezen keveseknek sokkal nagyobb a jártasságuk.

Ezt a hatást nagymértékben erősíti, ha a különböző felszíni jelenségektől a hibáig vezető nyomkövetési útvonalak, a felszíni tünetekből nem következő módon, jelentősen változnak. Egyetlen fejlesztő az ilyen utakból vett mintákat egymás után bejárva legalább olyan valószínűséggel akad rá egy nehéz útvonalra elsőként, mint egy könnyűre. Ha viszont feltételezzük a gyors kiadási ciklusokat, és azt, hogy sok ember végzi a nyomkövetést, akkor feltehetően az egyikük azonnal megtalálja a hibához vezető legrövidebb utat, és sokkal gyorsabban azonosítja a hibát. Amikor a projekt vezetője értesül a dologról, új kiadást készít, és az ugyanezen a hibán dolgozó egyéb emberek leállhatnak a kereséssel, még mielőtt túl sok időt töltenének el a bonyolultabb útvonalakon [RJ].

Nem fenékig tejföl

Linus magatartását tanulmányozva, és fogalmat alkotva arról, miért volt sikeres, úgy döntöttem, hogy tesztelem az elméletet új, kétségkívül egyszerűbb és kevésbé ambiciózus projektemen.

De előtte újraszerveztem és egyszerűsítettem a popclientet. Carl Harris megvalósítása becsülettel volt megírva, de volt benne egyfajta, a C programozókra jellemző, szükségtelen összetettség. A központban maga a kód állt, míg az adatszerkezeteket a kód kiegészítőiként kezelte. Ennek az eredménye volt, hogy a kód gyönyörű volt ugyan, de az adatszerkezeti elgondolás bizony csak alkalmi és csúnyácska (legalábbis egy veterán Lisp-hacker magas mércéje szerint).

Egy másik szándékom is volt az újraírással, a kód és az adatszerkezet tökéletesítésén túl szerettem volna olyasmivé fejleszteni, amit teljesen megértek. Nem élvezetes felelősséget vállalni egy olyan program hibáinak javításáért, aminek nem érted a működését.

Az első hónapban egyszerűen csak Carl alapvető szándékait követtem. Az első komoly változtatás amit végrehajtottam, az IMAP támogatás hozzáadása volt. Úgy oldottam meg, hogy a protokollokért felelős részt újraszerveztem, létrehozva egy általános motort és három eljárástáblázatot (a POP2, a POP3 és az IMAP részére). Ez, és az előző változtatások azt az általános elvet követik, amit a programozóknak érdemes szem előtt tartaniuk, különösen az olyan erősen típusos nyelveknél, mint a C:

9. A buta kód ügyes adatszerkezetekkel jobban működik, mint a fordítottja.

Brooks, kilencedik fejezet: „Mutasd a folyamatábrád és rejtsd el táblázataid, továbbra is zavarban leszek. Mutasd meg a táblázatokat, és nem lesz szükségem a folyamatábrára,

minden nyilvánvalóvá válik.” Eltekintve a harmincéves technológiai és kulturális különbségtől, a lényeg ugyanaz.

Akkoriban (1996 szeptemberének elején, a kezdés után hat héttel) azt kezdtem el fontolgatni, hogy helyénvaló lenne a névváltoztatás. Végül is már nem csak egy POP kliensről volt szó. Azért bizonytalankodtam, mert semmi eredeti nem volt a szerkezetben, popclientemnek még ki kellett fejlesztenie a saját egyéniségét.

Mindez radikálisan megváltozott, amikor a popclient megtanulta, hogy továbbíthatja az áthúzott leveleket az SMTP portra. Egy pillanat, és erre is rátérek, de előtte: azt mondtam korábban, hogy eldöntöttem, felhasználom a projektet Linus Torvalds igazáról alkotott elméletem tesztelésére. Megkérdezhetnéd, hogyan? Így:

- Korán és gyakran adtam ki (általában tíznaponként, intenzív fejlesztések idején minden nap).
- A béta listához hozzáadtam mindenkit, aki a fetchmail ügyében kapcsolatba lépett velem.
- Bő lére eresztett bejelentéseket küldtem a béta listára minden kiadás alkalmával, így bátorítottam a részvételt.
- Figyeltem bétatesztelőimre, tervezési kérdéseket tettem fel nekik, hízelegtem, amikor javításokat és visszajelzéseket küldtek.

Egyszerű lépéseimnek azonnali hatása volt. A projekt kezdeteitől olyan minőségű hibajelentéseket kaptam, gyakran használható javításokkal együtt, amilyenekért a legtöbb fejlesztő ölni tudna. Kaptam elgondolkodtató kritikát, rajongói levelet, intelligens javaslatokat. Ebből következően:

10. Ha bétatesztelőidet a legértékesebb erőforrásodként kezeled, a legértékesebb erőforrásoddá válva reagálnak.

A fetchmail sikerének egy érdekes mércéje a projekt bétatesztelői listájára, a fetchmail-friendsre feliratkozott tagok száma. Amikor utoljára átnéztem ezt a szöveget (2000 novemberében), 287 tag volt, és heti két-három új taggal bővült.

Egyébként amikor 1997 májusának végén néztem át, a lista érdekes okból kezdett el tagokat veszteni a 300 körüli maximumából. Többen is jelezték, hogy szeretnének leiratkozni, mert a fetchmaillel olyan mértékben elégedettek, hogy nincs szükségük már a listára. Talán ez is része a egy érett, bazár stílusú projekt életciklusának.

A popclientből fetchmail lesz

A projekt valódi fordulópontja az volt, amikor Harry Hochheiser elküldte nekem azt a rögtönzött kódot, amely a leveleket a kliens számítógép SMTP portjára továbbította. Majdnem azonnal ráébredtem, hogy ennek a lehetőségnek a megbízható megvalósítása az összes egyéb levélkézbesítési üzemmódot elavulttá teszi.

Heteken át trükköztem a fetchmaillel, miközben úgy éreztem, hogy az interfész felépítése használható, de fésületlen, nem elegáns, túl sok benne a jelentéktelen opció. A leszedett levelek postafiókfájlba vagy a szabványos kimenetre való kiírása különösen sok bosszúságot okozott, de nem tudtam, hogy miért.

(Ha nem érdekel az internetes levelezés technikája, a következő két bekezdést nyugodtan átugorhatod.)

Az SMTP-továbbításról gondolkodva azt láttam, hogy a popclient túl sok dolgot próbál elvégezni. Egyszerre alkalmas levéltovábbításra (MTA, Mail Transport Agent) és helyi kézbesítésre (MDA, Mail Delivery Agent). Az SMTP-továbbítással kiszállhattam volna az MDA-s ügyekből, és megmaradtam volna tisztán MTA-nak, átadva a leveleket más programoknak helyi továbbításra, pont úgy, ahogy azt a sendmail is teszi.

Miért maszatoljak a bonyolult kézbesítési beállítással, vagy a postaláda zárolásával és a hozzáfűzéssel, ha akármilyen TCP/IP támogatással rendelkező platformon csaknem garantáltan ott vár már a 25-ös port? Különösen, ha ez azt jelenti, hogy az elhozott levelek biztosan úgy fognak kinézni, mint egy rendes üzenetküldő által elindított SMTP üzenet, hiszen ez volt egyébként is a cél.

(Vissza magasabb szintre…)

Még ha nem is követted a műszaki zsargont, fontos tanulságok várnak itt ránk. Elsőként az, hogy az SMTP-továbbító koncepció volt a legnagyobb eredménye a Linus által használt módszer követésének. Egy felhasználó iszonyatos ötlettel állt elő, és csak annyit kellett tennem, hogy megértem ennek a következményeit.

11. A saját jó ötletek utáni legjobb dolog a felhasználóidtól származó jó ötletek felismerése. Időnként ez utóbbi jobb.

Érdekes módon hamar ráébredsz: ha teljesen, önelnyomó módon őszinte vagy abban, hogy mennyivel tartozol másoknak, a világ általában úgy fog kezelni, mintha az ötlet utolsó bitje is a tiéd lenne, csupán szerényen nyilatkoztál a tehetségeddel kapcsolatban. Láthatjuk, hogy ez milyen jól működött Linus esetében is.

(Amikor előadtam 1997 augusztusában az első Perl Konferencián, Larry Wall, a nagyszerű hacker az első sorban ült. Amint elértem a fenti mondathoz, mintha újjászületett volna, felkiáltott: „Még ilyet, testvérem!”. Az egész közönség hahotázott, tudták jól, hogy ugyanez működött a Perl atyjának esetében is.)

A projektet hasonló szellemben futtatva néhány héten belül dicséreteket kezdtem kapni nemcsak a felhasználóimtól, de más emberektől is, akikhez eljutott az ige. Néhány ilyen e-mailt biztonságba is helyeztem, és elő fogom szedni őket, ha valaha is azon kezdek el tűnődni, hogy ért-e valamit az életem :-).

De van még két alapvető tanulságunk, amelyek mindenféle tervre vonatkoztatva általánosan érvényesek.

12. A leginkább megdöbbentő és innovatív megoldások gyakran annak a felismeréséből származnak, hogy hibás volt a probléma felfogása.

Rossz problémát próbáltam megoldani a popclient kombinált MTA/MDA-ként való fejlesztésének folytatásával és az összes trükkös helyi továbbítási móddal. A Fetchmail szerkezetét az alapoktól tisztán MTA-ként kellett újragondolni, az internetes levelezés SMTP-t beszélő világának normális részeként.

Ha falba ütközöl a fejlesztés során, ha úgy érzed, hogy nem vagy képes túljutni a következő javításon, akkor gyakran nem azt a kérdést kell feltenned, hogy vajon van-e jó válaszod, hanem azt, hogy egyáltalán jó-e a kérdés. Talán új keretbe kell helyezni a problémát.

Nos, én megtettem. Világos, hogy az első dolog (1) az SMTP-továbbítás támogatásának az általános motorba való való beillesztése volt, majd (2) ennek az alapértelmezett móddá való tétele, és (3) minden más kézbesítési üzemmód kidobása, különösen a fájlba és szabványos kimenetre való kézbesítés kidobása.

A harmadik lépés előtt hezitáltam egy ideig, attól féltem, hogy a régi popclient-felhasználókat, akik számítottak az ilyen alternatív kézbesítési mechanizmusokra, ki fogom borítani. Elméletileg azonnal válthattak volna, és `.forward` fájlokat, vagy annak a nem sendmailes megfelelőit létrehozva, ugyanazt a hatást érhették volna el. A gyakorlatban azonban az átmenet rázósabb is lehet.

Végül amikor léptem, hatalmas előnyök származtak belőle. A motor kódjának túlbonyolított részei eltűntek. A beállítások radikálisan leegyszerűsödtek, nem volt több nyűglődés az MDA-val és a felhasználói postafiókkal, nem kellett amiatt sem aggódnom, hogy program kezére dolgozó operációs rendszer támogatja-e fájlok zárolását.

A levelek elvesztésének egyetlen lehetősége is eltűnt. Ha ugyanis a kézbesítés fájlba történt, a lemez pedig megtelt, a levelek elvesztek. Ez nem történhet meg az SMTP-továbbítás esetében, mert az SMTP-listener nem tér vissza az OK-val, csak abban az esetben, ha az üzenet kézbesíthető, vagy legalábbis bekerült a későbbi kézbesítésre várakozó sorba.

Növekedett továbbá a teljesítmény (ha nem is túl jelentősen). A változtatás további jelentős előnye a kézikönyvoldal sokkal egyszerűbbé válása volt.

Később aztán a felhasználó által definiált helyi MDA-n keresztüli kézbesítést vissza kellett hoznom, hogy lehetővé tegyem bizonyos különös, dinamikus SLIP-pel kapcsolatos szituációk kezelését. De sokkal egyszerűbb megoldást találtam rá.

Kételyek? Ne tétovázz, dobd ki a túlhaladott lehetőségeket, ha meg tudod tenni a hatékonyság csökkenése nélkül. Antoine de Saint-Exupéry (aki pilóta és repülőgép-tervező volt, nem pedig klasszikus gyermekkönyvek szerzője) mondta:

13. A tökéletességet (a tervezésben) nem akkor érjük el, amikor már nincs mit hozzáadni, hanem amikor már nincs mit elvenni.

Ha a kódod jobbá és egyszerűbbé is válik, *tudod*, hogy az helyes. A folyamat közben a fetchmail felépítése eredetivé, a régi popclienttől különbözővé vált.

Elérkezett a névváltoztatás ideje. Az új felépítés miatt a fetchmail a régi popclientnél sokkal inkább volt a sendmail párja. Mindkettő MTA, de míg a sendmail kiküld és kézbesít, addig a megújult popclient elhoz és kézbesít. Két hónappal az indulás után megtörtént a fechmaillé való átnevezés.

Az SMTP-továbbítás fetchmailbe kerülésén túl általánosabb tanulsággal is szolgál a történet. Nemcsak a hibakeresés párhuzamosítható, hanem a fejlesztés (talán egészen meglepő mértékig) és a tervezési tér felfedezése is. Ha a fejlesztési módszered gyorsan ismétlődő kiadásokat alkalmaz, a fejlesztés és a bővítés a hibakeresés speciális eseteivé, a szoftver

eredeti képességeiben és koncepciójában maradt „mulasztási hibák” javításaivá válhatnak.

Még a tervezés magasabb szintjein is nagyon értékes lehet, ha több társfejlesztő próbálgatja a lehetőségeket. Ahogy a víz megtalálja a lefolyót, vagy még ennél is szemléletesebben, ahogy a hangyák rátalálnak az élelemre: szétszóródásból adódó felfedezés, amit majd egy rugalmas kommunikációs mechanizmus segítségével használnak ki. Ez nagyon jól működik. Ahogy Harry Hochheiserrel és velem is megtörtént, a kísérőid egyike olyan előnyökre bukkanhat, amelyeket a szemellenződ miatt észre sem veszel.

A fetchmail felnő

Ott voltam egy csinos, megújult felépítménnyel, olyan kóddal, amelyről tudtam, hogy jól működik, mert minden nap használtam, és egy beinduló béta listával. Apránként kezdtem megérezni, hogy többé már nem az egyszerű, személyes bütyköléssel kell törődnöm, amely történetesen hasznosnak bizonyulhat mások számára is. Olyan program volt a kezemben, amelyre minden SLIP/PPP levelezési kapcsolattal és Unixszal rendelkező hackernek szüksége van.

Az SMTP-továbbító funkcióval a program elhúzott a vetélytársak mellett, és megvolt az esélye, hogy a kategória legjobbjává váljon, egy olyan klasszikus programmá, amely kitölti a számára fenntartott helyet, így az alternatívák nemcsak feleslegessé válnak, de szinte el is felejtődnek.

Szerintem az ilyen eredményeket nem lehet igazán előre megtervezni. Azok az erőteljes tervezési ötletek visznek bele, amelyek későbbi eredményei egyszerűen kikerülhetetlennek, természetesnek, sőt eleve elrendeltnek tűnnek. Az ilyen ötleteket csak úgy lehet kipróbálni, ha sok ötlet van – vagy tervezési döntésekkel kell elvinni mások jó ötleteit oda, ahová az ötletgazdák sem gondolták volna, hogy eljuthatnak.

Eredetileg Andy Tanenbaum ötlete volt egy egyszerű, natív Unix építése az IBM PC-kre, amit oktatási segédeszközként használhat (Minixnek hívta). Linus Torvalds tovább vitte a Minix-koncepciót, mint amire Andrew számíthatott, és valami remek dolog lett belőle. Ugyanígy (kisebb arányokban persze) én átvettem Carl Harris és Harry Hochheiser ötleteit, és adtam nekik egy lökést. Egyikünk sem volt eredeti a romantikus értelemben. A hackermítoszokkal szemben a természettudomány, a mérnöki munka és a szoftverfejlesztés nagyobb része sem eredeti tehetségek műve.

Az eredmény ugyanakkor mámorító, pontosan az a típusú siker, ami minden hacker célja. Ez azt jelentette, hogy még magasabbra kell helyeznem a lécet, olyan mértékben kell hasznossá tennem a fetchmailt, amennyire csak most látom, hogy lehetséges volt. Nem csupán a saját igényeim szerint kell megírnom, hanem támogatnom kell másokat is. Mindezt úgy, hogy a program továbbra is egyszerű és erőteljes marad.

Az első, nyomasztóan fontos lehetőség amit ezek után megírtam, a multidrop támogatása volt, annak a megvalósítása, hogy az olyan postaládákból, amelyekben egy csoport összes közös levele gyűlt, a program kiszedje a leveleket és minden egyes darabot a megfelelő címzetthez irányítson.

A multidrop hozzáadását részben azért határoztam el, mert néhány felhasználó már követelte, de főként azért, mert úgy gondoltam, hogy ez ki fogja hozni a hibákat az egyszerű kézbesítésért felelős (single-drop) kódból azzal, hogy rákényszerít a címzés általános

kezelésére. Ez be is bizonyosodott. Az RFC 822 által leírt címelemzés létrehozása különösen sok időmbe került, nem azért, mert bármelyik része nehéz volt, hanem mert egy halomnyi független és vesződséges részletet is bekerült a képbe.

De a mutidrop címzés ismét csak nagyszerű tervezési döntésnek bizonyult. Ezt abból tudtam, hogy:

14. Akármilyen eszköznek az elvárt módon kell hasznosnak lennie, de az igazán jó eszköz ott is felhasználható, ahol soha nem számítottál volna rá.

A multidropos fetchmail váratlan felhasználása olyan levelezési listák üzemeltetésében jelentkezett, ahol a lista és az álnév-feloldás (alias expansion) az internetkapcsolat *kliensoldalán* történik. Ez azt jelenti, hogy egy internetszolgáltatónál fiókkal rendelkező gép képes egy levelezési lista kezelésére a szolgáltatónál lévő alias fájlokhoz való folyamatos hozzáférés nélkül.

A bétatesztelőim által igényelt másik változtatás a 8 bites MIME (Multipurpose Internet Mail Extensions) működés volt. Ezt rendkívül könnyű volt teljesíteni, mert elővigyázatosan úgy kódoltam, hogy az nem volt érzékeny a nyolcadik bitre (ez azt jelenti, hogy nem használtam a nyolcadik bitet, amely az ASCII karakterkészletben egyébként is üres, a programon belüli információátvitelre). Nem azért, mert előre láttam ezt az igényt, hanem mert a következő szabálynak engedelmeskedtem:

15. Bármilyen információközvetítő szoftver írása esetén törekedj arra, hogy a lehető legkisebb mértékben avatkozz csak be az adatáramlásba, és soha ne dobj el semmilyen információt, hacsak a fogadó fél nem kényszerít erre.

Ha nem tartottam volna be ezt a szabályt, a 8 bites MIME-támogatás bonyolult lett volna és hibás. Nekem azonban csak a MIME szabványt (RFC 1652) kellett elolvasnom, és egy kis fejlécgeneráló tudással ellátnom a programot.

Néhány európai felhasználó addig nyaggatott, hogy bővítsem a programot egy olyan opcióval, amellyel szabályozható az egy menetben letölthető üzenetek száma (így kézben tarthatják a drága telefonos hálózati költségeiket), amíg meg nem tettem. Sokáig ellenálltam, és még most sem igazán tetszik. De ha egy világ számára programozol, hallgatnod kell a fogyasztóidra, ez nem változik meg attól, hogy nem pénzben fizetnek neked.

További fetchmail-tapasztalatok

Mielőtt visszatérnénk az általános szoftvertervezési kérdésekhez, van néhány mérlegelendő fetchmail-es tapasztalat. A nem műszaki érdeklődésű olvasók bátran továbbléphetnek.

A program működését szabályozó rc fájl szintaxisa olyan opcionális, „zajként” jelenlévő szavakat is magában foglal, amelyekkel az elemző egyáltalán nem törődik. Ezek az angol nyelvhez hasonló szintaxist tesznek lehetővé, így a hagyományos, tömör kulcsszó-érték pároknál, amelyeket akkor kapunk, ha kiszűrjük ezt a zajt, jelentősen könnyebben olvasható.

Az egész egy késő éjszakai kísérletként kezdődött, észrevettem, hogy mennyire kezdenek hasonlítani az rc fájl deklarációi egy imperatív mininyelvre. (Ezért változtattam meg a popclient „server” kulcsszavát „poll”-ra.)

Úgy tűnt számomra, hogy ezt az imperatív mininyelvet talán könnyebb lesz használni, ha az angolhoz hasonlóvá teszem. Bár én az olyan példákat felvonultató „csinálj belőle nyelvet” tervezési iskola meggyőződéses partizánja vagyok, mint az Emacs, a HTML és számos adatbázismotor, rendszerint mégsem lelkesedem az angolhoz hasonló szintaxisért.

A programozók hagyományosan a precíz, kompakt és egyáltalán nem redundáns vezérlő szintaxist szokták kedveli. Ez még azokból az időkből származó kulturális örökség, amikor a számítási erőforrások drágák voltak, így az elemzéssel eltöltött lépéseknek minél olcsóbbaknak és egyszerűbbeknek kellett lenniük. Az angol az 50% százalék körüli redundanciájával akkor rendkívül rossz modellnek látszott.

Nem is ez az oka, hogy általában kerülöm az angolra hasonlító szintaxisokat, ezt az érvet csak azért említettem, hogy egyben le is romboljam. Az olcsó ciklusok idejében nem a tömörségre kellene törekedni. Manapság sokkal fontosabb egy nyelv kényelmessége az emberek számára a gép erőforrásaival való takarékoskodásnál.

De marad még okunk az óvatosságra. Az egyik az elemzési lépcsőfok összetettségének költsége, ezt nem akarhatod odáig emelni, ahol már önmagában is a hibák és felhasználók zavarodottságának jelentős forrásává válik. Egy másik ok, hogy egy nyelv szintaxisát az angoléhoz hasonlóvá tenni azzal jár, hogy a nyelv által létrehozott „angol” ugyancsak kitekert lesz, így a természetes nyelvhez való felszíni hasonlóság legalább olyan zavaró lesz, mint amilyen a hagyományos szintaxis lett volna. (Ezzel a beteges jelenséggel találkozhatunk több úgynevezett „negyedik generációs” kereskedelmi adatbázis-lekérdező nyelvben.)

A fetchmail vezérlő szintaxisa úgy tűnik, hogy elkerülte ezeket a problémákat, mert a nyelvi tartománya rendkívül korlátozott. A nyomába sem ér egy általános célú nyelvnek, a rajta kifejezett dolgok nem túl komplikáltak, így kevés az esélye annak, hogy összetévesztjük az angol egy picike részhalmazát magával a vezérlőnyelvvel. Szerintem ebből a következő tanulságot szűrhetjük le:

16. Ha a nyelved közel sem Turing-teljes, akkor a szintaxissal kifejezőbbé teheted.

Van itt még egy tanulság a bizonytalansággal elért biztonságról. Néhány fetchmail-felhasználó kérte, hogy olyan módosításokat végezzek, amelyek lehetővé teszik a jelszavak titkosított tárolását is az rc fájlban, így az utánuk szaglászók véletlenül sem láthatják meg azokat.

Nem teljesítettem a kérést, mert ez nem jelentene tényleges védelmet. Bárki, aki megszerezte a jogosultságokat ahhoz, hogy beleolvasson az rc fájlba, képes arra is, hogy annak tulajdonosához hasonlóan futtassa a fetchmailt. Ha pedig a jelszó kell nekik, akkor képesek kiszedni az ehhez szükséges dekódert a fetchmail forráskódjából.

A `.fetchmailrc`-ben lévő jelszavak titkosítása a biztonság hamis érzetét keltené bizonyos emberekben. Az általános szabály tehát:

17. Egy biztonsági rendszer csak annyira biztonságos, amennyire a titkai azok. Óvakodj az általtitkoktól.

A bazár szükséges előfeltételei

Az esszé korai bírálói és tesztközönsége folyamatosan kérdéseket tett fel a sikeres bazár stílusú fejlesztés előfeltételeiről, továbbá a projektvezető képzettségéről, valamint a kód állapotáról a nyilvánosságra hozatalkor, illetve a társfejlesztői közösség kiépítésének kezdeteiről.

Az egészen világos, hogy senki nem kezdhet neki a bazár stílusú kódolásnak az alapoknál [IN]. Tesztelni, hibát keresni és továbbfejleszteni lehet így, de igen nehéz egy projektet *létrehozni* a bazár módszerével. Linus sem próbálta. Én sem. A kialakulóban lévő fejlesztői közösségednek szüksége van valami futtatható, tesztelhető játékszerre.

Amikor közösségépítésbe kezdesz, fel kell tudnod mutatni egy *tetszetős ígéretet*. Nem kell a programodnak különösebben jól működnie, lehet durva, hibás, befejezetlen és gyengén dokumentált. De (a) futnia kell, és (b) meg kell győznie a lehetséges társfejlesztőket arról, hogy valami igazán klassz dologgá fejlődhet belátható időn belül.

A Linux és a fetchmail is erős, megnyerő alapszerkezettel került a nyilvánosság elé. Sokan azok közül, akik úgy gondolkodnak a bazár modellről, ahogy azt bemutattam, helyesen tekintik ezt lényegesnek, és ebből arra következtetésre jutnak, hogy jó tervezési előérzetek és az intelligencia a projektvezető nélkülözhetetlen tulajdonságai.

De Linus az alapokat a Unixból vette, én pedig a popclientből (bár ez később jelentősen megváltozott, sokkal nagyobb arányban, mint amiről a Linux esetén beszélhetnénk). Tehát a bazár stílben való fejlesztés vezetőjének vagy koordinátorának valóban kivételes tervezési tehetséggel kell rendelkeznie, vagy elegendő felkarolnia mások tehetségét?

Szerintem az nem fontos, hogy a koordinátor képes legyen kivételesen briliáns tervek létrehozására, az viszont abszolúte fontos, hogy a koordinátor képes legyen *más jó tervezési ötleteinek a felismerésére*.

Ezt bizonyítja a Linux és a fetchmail projekt is. Linus, ahogy ezt korábban már tárgyaltam, nem egy látványosan eredeti tervező, de rendkívül jártas a jó tervezési ötletek felismerésében, és azok Linux kernelbe integrálásában. Azt is bemutattam, hogy a fetchmail legjobb ötlete (az SMTP-továbbítás) valaki mástól származik.

Az esszé korai közönsége azzal járt a kedvemben, hogy azt sugallta, szerényen alábecsülöm a bazár projektekben a tervezés eredetiségét, mert ebből bennem sok van, és ezért természetesnek veszem. Lehet, hogy ebben van némi igazuk, a tervezés, a kódolással és a hibakereséssel ellentétben, a legerősebb oldalam.

A szoftvertervezésben okosnak és eredetinek lenni azért problémás, mert szokássá válik, önkéntelenül is agyafúrt és összetett dolgokat kezdesz készíteni, akkor is, amikor egyszerűnek és szilárdnak kellene lenniük. Voltak olyan projektjeim, amelyek lefulladtak, mert elkövettem ezt a hibát, a fetchmaillel azonban el tudtam kerülni.

Úgy gondolom, hogy a fetchmail projekt sikere részben azon is múlt, hogy visszafogtam magam; ez egy érv a bazár projektekben nélkülözhetetlen tervezési eredetiség ellen. Vagy gondoljunk csak a Linuxra. Tegyük fel, hogy Linus Torvalds az operációs rendszerek tervezésének alapvető újításait próbálgatta volna fejlesztés közben. Van annak egyáltalán

valószínűsége, hogy az eredményül kapott kernel olyan megbízható és sikeres lett volna, mint amilyen most?

A tervezésben és kódolásban való jártasság egy bizonyos szintje persze elengedhetetlen, de úgy gondolom, hogy jóformán akárki, aki komolyan gondolkodik egy bazár projekt elindításán, már túllépett ezen a szinten. A nyílt forráskódú közösség belső tekintélyviszonyai finom nyomást gyakorolnak azokra az emberekre, akik nem képesek fejlesztési törekvéseik megvalósítására, és ez eddig egészen jól működött.

Van egy másik tulajdonság is, amit viszont általában nem kapcsolnak össze a szoftverfejlesztéssel, pedig szerintem legalább olyan fontos a bazár projektek esetében, mint a tervezési intelligencia, sőt talán fontosabb is annál. A projektvezetőnek vagy koordinátornak jól kell tudnia kezelni az embereket, és jó kommunikációs készséggel kell bírnia.

Ennek nyilvánvalónak kellene lennie. Ahhoz, hogy egy fejlesztői közösséget építs, fel kell keltened az emberek figyelmét, érdekessé kell tenned számukra a munkád, és elégedettségben kell őket tartanod, hogy az általuk végzett munka megfelelő. A műszaki érdekességek sokat segítenek ennek elérésében, de ez messze nem elegendő. Az általad kivetített személyiség is fontos.

Nem véletlen, hogy Linus remek fickó, akit szeretnek, és akinek segíteni akarnak az emberek. Az sem véletlen, hogy erélyes, kifelé forduló ember vagyok, aki élvezi a társas munkát, van humorérzéke, és elő tudja magát adni. A bazár modell működését segíti, ha legalább egy kis jártasságod van az emberek elbűvölésében.

A nyílt forráskódú szoftver szociális kontextusa

Jól írtam, a legjobb kódok a szerző mindennapi problémáinak személyes megoldásaiként indulnak, és elterjednek, mert a problémáról kiderül, hogy egy nagyobb felhasználói csoport számára sem ismeretlen. Ez az első számú szabályunkhoz vezet vissza, talán egy kissé gyakorlatiasabban újrafogalmazva:

18. Egy érdekes probléma megoldásához találj először egy olyan problémát, amely számodra érdekes.

Így volt ez Carl Harris és a régi popclient esetében, illetve velem és fetchmaillel is. De ezt már régóta tudjuk. Az érdekes dolog, az, amire a Linux és a fetchmail története is megkívánja, hogy figyeljünk, a következő lépcsőfok: a szoftver fejlődése egy nagy és aktív felhasználói, illetve társfejlesztői közösség jelenlétében.

A *The Mythical Man-Month*-ban Fred Brooks azt a megfigyelést tette, hogy a programozók ideje nem váltható ki, egy elcsúszott szoftverprojektbe újabb fejlesztők bevonása további csúszást eredményez. Ahogy láttuk korábban, ő amellett érvelt, hogy egy projekt összetettsége és kommunikációs költsége a fejlesztők számával négyzetesen emelkedik, míg az elvégzett munka csupán lineárisan növekszik. Brooks törvényére általában elcsépelt igazságként gondolnak. De ezen esszében megvizsgáltunk számos olyan nyílt forráskódú fejlesztési eljárást, amely rácáfol ezekre a feltételezésekre, és a gyakorlatban, ha Brooks törvénye teljesen lefedné a valóságot, a Linux nem létezhetne.

Gerald Weinberg klasszikus műve, a *The Psychology of Computer Programming* tartalmazta azt, amit utólagosan Brooks alapvető kiigazításaként értékelhetünk. Az „énnélküli programozás" tárgyalása során Weinberg megfigyelte, hogy azokban a műhelyekben, ahol a fejlesztők nem gyakorolnak hatalmat a kódjuk felett, és az azon végzett hibakeresésre, továbbfejlesztésre bátorítanak másokat, a fejlődés lényegesen gyorsabb, mint máshol. (Újabban Kent Beck „extrém programozási" technikáját – amikor a kódolók párokban figyelik egymást – foghatjuk fel úgy, mint egy kísérletet ezen hatás kikényszerítésére.)

A Weinberg által használt terminológia talán meggátolta elemzésének olyan mértékű elterjedését, amilyet megérdemelt volna – megmosolyogtató az internetes hackerek „énnélküli"-nek nevezése –, úgy gondolom, hogy az érvei ma vonzóbbak, mint eddig bármikor.

A bazár módszer az „énnélküli programozás" teljes erejét felhasználva erősen csillapítja Brooks törvényének hatásait. Brooks törvényének alapelvei jók, de nagy fejlesztőközösség és olcsó kommunikáció mellett a hatásait elnyomják az egyébként láthatatlan, versengő nemlinearitások. Ez a newtoni és az einsteini fizika viszonyára emlékeztet, a régebbi rendszer még érvényes alacsony energiák mellett, de elég nagy tömeg és a sebesség esetén olyan meglepetések érhetnek, mint a nukleáris robbanás vagy a Linux.

A Unix történetének fel kellett volna készítenie minket arra, amit a Linuxtól tanulunk (és amit kísérletileg ellenőriztem kisebb méretekben Linus módszereinek tudatos másolásával [EGCS]). Vagyis míg a kódolás alapvetően magányos tevékenység marad, az igazán jó megoldások teljes közösségek intelligenciáját és figyelmét igénylik. Az a fejlesztő, aki csak a saját elméjét használja egy elzárt projektben, elmarad amögött, aki tudja, hogyan hozzon létre nyílt evolúciós környezetet, amelyben a tervezési lehetőségekre vonatkozó visszajelzések, a hozzájárulás a kódhoz, a hibák kiszúrása és az egyéb fejlesztés több száz (esetleg több ezer) ember segítségével történik.

De a hagyományos unixos világot számos tényező akadályozta ennek a megközelítésnek a felkarolásában. Az egyik a különféle licencek, üzleti titkok és kereskedelmi érdekek szorítása volt, a másik pedig az (így utólag), hogy az internet még nem volt elég jó.

Az olcsó internet előtt voltak földrajzilag behatárolt közösségek, amelyek kultúrája segítette Weinberg „énnélküli" programozását, és egy fejlesztő könnyen találhatott képzett segítséget és társfejlesztőket. A Bell Labs, az MIT AI és LCS laborjai, az UC Berkeley – ezek a helyek legendás, máig ható újítások otthonaivá váltak.

A Linux volt az első projekt, amely számára egy öntudatos és sikeres törekvés a *világból* bárhonnan származó tehetséget elérhetővé tette. Nem hiszem, hogy az véletlen, hogy a Linux korai fejlődésének időszaka egybeesett a World Wide Web születésével, és hogy a Linux túllépett a gyermekkorán ugyanezen időszakban, 1993-94-ben, amikor beindult az internetszolgáltatási ipar, és robbant az internet iránti érdeklődés. Linus volt az első, aki megtanulta, hogyan kell az általánossá váló interneteléréssel együtt érkező új szabályok szerint játszani.

Noha az olcsó internet a Linux modellben szükséges feltétel volt a fejlődéshez, szerintem önmagában nem volt elegendő feltétel. Fontos tényező volt egy bizonyos vezetési stílus, illetve azon együttműködési szokások kifejlődése, amelyek által a fejlesztők megragadhatták a társfejlesztőket, és a legtöbbet hozhatták ki az internetes közegből.

De mi ez a vezetési stílus, milyenek ezek a szokások? Nem lehet őket hatalmi viszonyokra építeni – még ha lehetne is, a vezetők kényszerrel nem érhetnék el az általunk látott eredményt. Weinberg a témával kapcsolatban idézi Pjotr Alekszejevics Kropotkin, 19. századi orosz anarchista önéletrajzát, az *Egy forradalmár feljegyzéseit*:

> Jobbágytartó családban nevelkedve léptem a tevékeny életbe, és mint minden fiatalember az én időmben, erősen bíztam az irányítás, az utasítás, a szidás, büntetés és ehhez hasonlók szükségességében. De amikor, egy korai szakaszban, fontos vállalkozásokat kellett igazgatnom, és [szabad] emberekkel bánnom, amikor minden egyes hiba azonnali komoly következményekkel járt volna, elkezdtem becsülni a parancselven, fegyelem alapján végrehajtott tettek, illetve az általános megállapodás elvén végrehajtott tettek közti különbséget. Az első nagyszerűen működik egy katonai bemutatón, de semmit nem ér ott, ahol valódi életről van szó, és a cél csak sok egyező akarat komoly törekvésével érhető el.

A „sok egyező akarat komoly törekvésével” pontosan az, amire a Linuxhoz hasonló projekteknek szüksége van, míg a „parancselvet” gyakorlatilag lehetetlen alkalmazni az internetnek hívott anarchista paradicsom önkénteseinél. A hatékony működés és verseny érdekében azoknak a hackereknek, akik együttműködő projektet szeretnének vezetni, meg kell tanulniuk, hogyan toborozzanak és mozgassanak meg hatékony, érdeklődő közösséget Kropotkin „megállapodáselve” alapján. Meg kell tanulniuk használni Linus törvényét [SP].

Korábban említettem a Delphoi-effektust, mint lehetséges magyarázatot Linus törvényére. De ennél jobb analógiák kínálkoznak az adaptív rendszerekkel a biológiában és a közgazdaságtanban. A linuxos világ sok tekintetben a szabadpiachoz hasonlóan működik, vagy élőlények és környezetük, a hasznuk maximalizálására törekvő önző ágensek csoportjához hasonlóan, amelyek a folyamatba egy spontán, a központi tervezéssel elérhetőnél kimunkáltabb és hatékonyabb önjavító rendet visznek. Egy ilyen helyen kereshetjük a „megállapodás elvét”.

A Linux-hackerek által végsőkig fokozott, nem klasszikus gazdasági értelemben vett „hasznos működés” inkább önmaguk kielégítését és a hackertársak közötti megbecsülést célozza. (Ezt a motivációt önzetlennek is nevezhetnénk, de akkor nem vennénk tudomást arról a tényről, hogy az önzetlenség önmagában is az én kielégítésének egy formája.) Az ilyen módon működő önkéntes kultúrák nem ritkák, egy másik – amelynek magam is részese vagyok – a science-fiction rajongók tábora, ebben a hackerekétől eltérően már régóta világosan kirajzolódott az „egoboo” (az én növelése, önmagunk hírnevének fokozása ez egyéb rajongók között), mint az önkéntes tevékenységek alapvető mozgatórugója.

Linus azzal, hogy sikeresen pozicionálta magát egy olyan projekt őrzőjeként, amelyben a fejlesztést jórészt mások végezték, és fenntartotta a projekt iránti érdeklődést, amíg az önfenntartóvá nem vált, jól megragadta Kropotkin „megosztott megértés elvét”. A linuxos világ e félig gazdasági szemlélete lehetővé teszi, hogy lássuk, hogyan alkalmazható ez a megértés.

Linus módszerére tekinthetünk úgy, mint egy hatékony „egoboo”-piacot létrehozó módszerre, amely a lehető legfinomabban kapcsolja össze a hackerek egyéni önzését olyan bonyolult célokért, amelyek csak a kooperáció fenntartásával érhetőek el. A fetchmail projekttel megmutattam (bár csak kicsiben), hogy ezek az eljárások jó eredménnyel másolhatóak. Talán némileg még nála is öntudatosabban és rendszeresen csináltam.

Sokan (különösen akik nem bíznak a szabadpiacokban) arra számítanának, hogy az önmagukat irányító egoisták kultúrája töredezett, felségterületekre osztott, pazarló, titkolózó és ellenséges lesz. Erre a várakozásra azonban, hogy csak egyetlen példát említsek, rácáfol a linuxos dokumentációk minősége, meglepő sokszínűsége és mélysége. Megbocsátható tény, hogy a programozók *gyűlölik* a dokumentálást, hogyan fordulhat akkor elő, hogy a linuxos hackerek ennyi dokumentációt állítanak elő? A Linux törekvő énekből álló szabadpiaca nyilvánvalóan könnyebben hoz létre értékes, mások által kívánatosnak tartott viselkedést, mint a kereskedelmi szoftvergyártók jól megalapozott dokumentációs műhelyei.

A fetchmail és a Linux kernel projekt is azt mutatja, hogy más hackerek egójának megfelelő elismerésével egy erős fejlesztő/koordinátor kihasználhatja a társfejlesztők előnyeit az interneten keresztül anélkül, hogy a projekt kaotikus összevisszaságba esne. Ezért a következőt javaslom Brooks törvényével szemben:

19. Több ember kétségtelenül jobb egynél, ha a fejlesztés koordinátorának legalább olyan jó kommunikációs közeg áll a rendelkezésére mint az internet, és képes a kényszer nélküli vezetésre.

A nyílt forráskódú szoftverek jövője szerintem egyre inkább azokon az embereken múlik, akik tudják, hogyan játsszák Linus játékát, azokon, akik a bazárért otthagyják a katedrálist. Ez nem azt jelenti, hogy az egyedi értékek többé nem fognak számítani. Sokkal inkább jelenti azt, hogy a vezető nyílt forráskódú szoftverek mögött olyan emberek állnak majd, akik az egyedi látásmódjukat, értékeiket önkéntes közösségek létrehozásával erősítik.

Talán ez nem csak a *nyílt forráskódú* szoftverek jövője. A zárt kódú világban nem képesek egy problémára annyi tehetséget áldozni, amennyit a Linux-közösség tud ugyanarra fordítani. Rendkívül kevesen engedhetik meg maguknak már azt a 200-nál (1999-ben 600, 2000-ben 800) több embert is, akik a fetchmail fejlesztésében közreműködtek.

Végül talán a nyílt forráskódú kultúra győzedelmeskedik, nem azért, mert a kooperáció morálisan helyes, míg a szoftver működésének elrejtése elítélendő, hanem egyszerűen azért, mert a zárt kódú szoftverek képtelenek győzni egy evolúciós fegyverkezési versenyben a problémákra nagyságrendekkel több szakértői időt áldozni képes nyílt kódú közösségek ellenében.

A vezetésről és a Maginot-vonalról

A katedrális és a bazár 1997-es eredeti szövegváltozata az előbbi vízióval végződött – a programozók/anarchisták hálózatra kapcsolt elégedett seregei a versenyben legyőzik és elnyomják a konvencionális, zárt szoftverek hierarchikus világát.

Ugyanakkor ez rengeteg szkeptikust nem győzött meg, és az általuk felvetett kérdésekkel érdemes foglalkozni. A bazár melletti érvek ellenérveinek legtöbbje szerint alábecsüljük a konvencionális vezetés termelékenységet többszöröző hatását.

A hagyományos beállítottságú szoftverfejlesztési vezetők gyakran hangsúlyozzák, hogy azt a lazaság, amellyel a nyílt forráskódú világban a projekteket alkotó csoportok szerveződnek, változnak és felbomlanak, jelentősen ellensúlyozza a látszólagos számbeli előnyt, amellyel a nyílt forráskódú közösség rendelkezik a zárt kódú szoftvereket fejlesztőkkel szemben. Azt is megjegyezhetik, hogy a szoftverfejlesztés területén valóban állandó a törekvés a fogyasztók által elvárt időtartamú és mértékű folyamatos befektetésre a fontos termékek esetében, és

nem csupán arról van szól, hogy hány ember dobott be valamit a közös fazékba, és hagyta megfőni.

Kétségkívül van ebben az érvelésben valami. Arra jutottam a *The Magic Cauldron* című esszémben, hogy az elvárható jövőbeli szolgáltatások kulcsfontosságúak a szoftvertermelés gazdasági rendszerében.

Ugyanakkor az érvelés fontos rejtett problémát tartalmaz, azt az implicit feltételezést, hogy a nyílt forráskódú fejlesztési modell nem képes ilyen törekvések fenntartására. Valójában azonban léteznek nyílt forráskódú projektek, amelyek következetes irányban haladnak, és hosszú időn át fenntartják hatékony karbantartó közösségüket a konvencionális vezetés által nélkülözhetetlennek tartott ösztönző struktúrák és szervezeti kontroll nélkül is. A GNU Emacs szerkesztőprogram fejlesztése szélsőséges és tanulságos példája ennek, az Emacs több száz közreműködő 15 éven át tartó hozzájárulásait olvasztotta egységes szerkezetbe. Tette ezt a nagy fejlesztői forgalom, és azon tény ellenére, hogy csak egyetlen ember (a program eredeti szerzője) maradt folyamatosan aktív mind a 15 éven át. Nincs olyan zárt kódú szerkesztő, amelyik ilyen hosszú életű lett volna.

Ez azt sugallja, hogy okunk van megkérdőjelezni a hagyományosan irányított szoftverfejlesztés olyan előnyeit, amelyek nem függenek a katedrális és a bazár módszer egyéb kérdéseitől. Ha a GNU Emacs 15 éven át képes volt egységes szerkezetet felmutatni, vagy például egy olyan operációs rendszer, mint a Linux képes volt ugyanerre 8 éven át, a gyorsan változó hardverplatformok és technológiák ellenére, és ha számos jó felépítésű nyílt forráskódú szoftver jött létre az elmúlt több mint öt évben, akkor okkal töprenghetünk el azon, hogy mi haszna van a hagyományosan irányított fejlesztés fenntartásának.

Bármiről is legyen szó, nagyon ritka az olyan irányított projekt, amely eléri a megbízható programműködést a határidőre, belefér a költségvetésébe, és minden, az előírásnak megfelelő funkciót tartalmaz, sőt az is ritkaság, hogy ezekből egyetlen feltétel teljesül. Az ilyen projektek az élettartamuk alatt általában nem képesek jól alkalmazkodni a technológiai és a gazdasági környezet változásaihoz sem. A nyílt forráskódú közösség ebben *lényegesen* hatékonyabb. (Ezt bárki könnyedén ellenőrizheti az internet harmincéves történetének, illetve a védjegyzett hálózati technológiák gyors felezési idejének összevetésével – vagy a Microsoft Windows alatti 16-ról a 32 bitre való átállás költségeinek, és az ugyanabban az időszakban különösebb megerőltetés nélkül zajló, ám hasonló jelentőségű Linux alatti változások [költségeinek] összevetésével, amelyek egyébként nemcsak az Intel, hanem több mint egy tucat hardverplatformon történtek, köztük a 64 bites Alphán is.)

Sokan gondolják azt, hogy a hagyományos módszer előnye, hogy a törvény előtt is felelőssé tesz valakit, akitől lehetőség van kárpótlást követelni, ha a projekt elvéti az irányt. Ez azonban illúzió, a legtöbb szoftverlicenc úgy van megírva, hogy kizárja még az eladhatósági garanciát is, a teljesítményről már nem is szólva, és a szoftverek hibás működése miatti sikeres kárpótlási ügyek száma elenyészően csekély. De ha gyakoriak is lennének, az érzés, hogy valaki perelhető, nem oldana meg semmit. Működő szoftvert szeretnénk, nem pereskedést.

Akkor tehát mi haszna a hagyományos vezetési rendszernek?

Hogy ezt megértsük, meg kell ismernünk, hogy mit gondolnak a szoftverfejlesztési vezetők a munkájukról. Egy munkájában sikeresnek tűnő ismerős hölgy elmondta, hogy a szoftverprojekt-menedzsmentnek öt funkciója van:

- *Célok kitűzése*, és a munkatársak azonos cél felé irányítása.
- *Megfigyelés*, illetve annak biztosítása, hogy nem maradnak ki fontos részletek.
- Az emberek *motiválása* az unalmas, de szükséges egyhangú feladatokra.
- A produktivitást legjobban elősegítő felépítés *megszervezése*.
- A projekt fenntartásához szükséges *erőforrások megszerzése*.

Nyilvánvalóan kiváló cél mindegyik, de a nyílt forráskódú modellben és az azt körülvevő szociális környezetben különösen jelentéktelennek tűnhetnek. Lássuk őket fordított sorrendben.

Az ismerősöm szerint az *erőforrások megszerzése* gyakran védekezés. Ha egyszer rendelkezésre állnak az emberek, a gépek és az irodahelyiség, akkor ezeket meg kell védeni az ugyanezekért az erőforrásokért versengő társvezetőktől és a magas beosztású vezetőktől, akik a korlátozott erőforrásokat lehető leghatékonyabban próbálják elosztani.

A nyílt forráskódú világ fejlesztői önkéntesek, érdeklődésük és képességeik alapján járulnak hozzá projektjeik munkájához (és ez általában akkor is igaz marad, ha fizetést kapnak a nyílt forráskódú programozásért). Az önkéntes szellemiség feleslegessé teszi az erőforrásszerzést, az emberek a saját erőforrásaikat teszik az asztalra. Így nincs szükség védekező vezetőre a hagyományos értelemben.

Egyébként az olcsó PC-k és a gyors internetkapcsolatok világában az egyetlen korlátozott erőforrás a szakértői figyelem. A nyílt forráskódú projektek felbomlása alapvetően sosem számítógépek, hálózati kapcsolatok vagy irodahelyiségek miatt történik, csak akkor múlnak ki, amikor a fejlesztők maguk vesztik el az érdeklődésüket.

Ha így áll a helyzet, akkor duplán fontos, hogy a nyílt forráskódú hackerek az önkiválasztás által a maximális teljesítmény elérésére *szerveződjenek*. Az ilyen közösségek szociális környezete a kompetencia alapján kíméletlenül válogat. Az ismerősöm szerint, aki egyaránt ismeri a nyílt forráskódú világot és a nagy, zárt projekteket, a nyílt forráskód részben azért sikeres, mert kultúrája csupán a programozással foglalkozó populáció legtehetségesebb 5 százalékát fogadja el. Ő az idejének nagyobb részében a maradék 95 százalékból valókat szervezi, így első kézből tapasztalta meg a legjobb programozók és a pusztán kompetensek termelékenysége közötti százszoros eltérést.

Az ilyen mértékű eltérés mindig fölveti a kínos kérdést, hogy a zárt kódú projektek a hátterükkel együtt, nem lennének-e jobb helyzetben a kevésbé tehetségesek több mint 50 százaléka nélkül? A komoly vezetők régóta tudják, hogy ha a konvencionális szoftver-menedzsment egyetlen funkciója a kevésbé rátermettekből származó nettó veszteség csekély nyereséggé alakítása lenne, az egész játék nem érné meg a fáradságot.

A nyílt forráskódú közösség sikere meglehetősen súlyossá teszi a kérdést, azt bizonyítva, hogy gyakran olcsóbb és hatékonyabb önjelölt önkéntesek toborzása az internetről, mint olyan felépítmények irányítása, amelyek tele vannak valami mással szívesebben foglalkozó emberekkel.

Ezzel akár át is térhetünk a *motiváció* kérdésére. A ismerősöm által mondott dolgok gyakran hallható újrafogalmazása, hogy a hagyományos szoftverfejlesztés-menedzsment a gyengén motivált és másként jó munkára képtelen programozók kompenzálásához szükséges.

Ez az állítás általában azzal jár együtt, hogy a nyílt forráskódú közösségre csak szexi,

műszakilag érdekes munka esetén lehet számítani, bármi más befejezetlenül marad (vagy gyenge minőségben készül el), hacsak nem készítik el a vezetők által korbáccsal hajtott melósok pusztán a pénzért. *Homesteading the Noosphere* című esszémben pszichológiai és társadalmi érveket sorakoztatok fel ezen állítással szemben, most azonban úgy gondolom, hogy sokkal érdekesebb lesz, ha rámutatunk az állítás elfogadásának következményeire.

Amennyiben a hagyományos, zárt kódú, erősen irányított szoftverfejlesztési módszert valóban csak egyfajta Maginot-vonal védi az unalmas problémáktól, az fenntartható marad minden alkalmazási területen egészen addig, amíg valaki nem találja ezeket a problémákat valóban érdekesnek, és más sem akad olyan utakra, amelyeken megkerülhetőek lennének. Mivel ebben a pillanatban megjelenik egy „unalmas" szoftver nyílt forráskódú vetélytársa, a fogyasztók tudni fogják, hogy végre olyasvalaki oldotta meg a dolgot, akit már maga a probléma is elbűvölt – ez a szoftverek területén éppúgy, mint bármilyen egyéb, kreativitást igénylő területen sokkal hatékonyabb ösztönző, mint a pénz önmagában.

Ilyenkor a hagyományos vezetési szerkezet fenntartása pusztán a motiváció céljából talán jó taktika, de rossz stratégia, rövid távon nyereség, de hosszú távon biztosabb veszteség.

Eddig tehát a konvencionális fejlesztés-menedzsment két szempontból is rossz választásnak tűnik a nyílt forráskóddal szemben (erőforrásszerzés, szervezés), egy harmadikból vizsgálva pedig csak rövidtávú előnyei vannak (motiváció). A szegény, sarokba szorított konvencionális vezetőt nem segíti ki a *megfigyelés* témája sem, a nyílt forráskódú közösség legerősebb érve, hogy a decentralizált, mindenki által gyakorolható vizsgálat üti a részletek kihagyásának elkerülésére irányuló megszokott eljárásokat.

Talán a *célok kitűzése* igazolhatja a konvencionális szoftverprojekt-menedzsment létezését. Talán. De ehhez jó indokot kell szolgáltatniuk arra, hogy vezetőségi bizottságok és vállalati útitervek sikeresebben határozzák meg a közös, érdemes célokat, mint a nyílt forráskódú világban hasonló szerepet betöltő projektvezetők vagy a „törzs öregjei".

Ez első pillantásra kemény döntésnek tűnik. Nem is annyira a nyílt forráskódú oldala (az Emacs hosszú élete, Linus Torvalds fejlesztők seregeit megmozgató képessége) teszi nehézzé, inkább a konvencionális mechanizmusok itt bemutatott borzalmassága a szoftverprojektek céljainak meghatározásában.

A szoftvertervezés jól ismert általános bölcsessége, hogy a konvencionális szoftverprojektek 60-75 százaléka nem éri el a célját, vagy a megcélzott felhasználók visszautasítják az eredményt. Ha ez a tartomány az igazság közelében van (soha nem találkoztam olyan tapasztalt vezetővel, aki ezt vitatta volna), akkor több projekt elé tűznek ki (a) reálisan megvalósíthatatlan (b) vagy egyszerűen csak rossz célokat, mint ahány elé nem.

Minden más problémánál erősebb indok ez arra, hogy a mai szoftvertervezői világban a „vezetői bizottság" kifejezéstől borsóddzék a közönség háta, még – vagy talán különösen akkor – ha vezetők is vannak köztük. Azok az idők, amikor mindezt csak a programozók értették, elmúltak, a Dilbert képregények („Dilbert a technológiát a technológia miatt szereti. Valójában jobban szereti a technológiát, mint az embereket, és egy egéralátét szociális képességeivel rendelkezik" – www.dilbert.com, *a ford*.) ma már a *vezetők* asztalai fölött lógnak.

Válaszunk tehát a hagyományos szoftverfejlesztési vezető számára egyszerű: ha a nyílt forráskódú közösség valóban alábecsülte a konvencionális vezetés értékeit, akkor miért vetik

meg olyan sokan önök közül is a saját eljárásaikat?

Csak ismételni tudom, hogy a nyílt forráskódú közösség példája tovább súlyosbítja ezt a kérdést – mert mi *örömet* lelünk abban, amit csinálunk. Kreatív játékunk műszaki, piaci és szakmai sikerei elképesztőek. Nemcsak azt bizonyítjuk, hogy jobb szoftverek írására vagyunk képesek, hanem azt is, hogy az öröm tőke.

Az esszé első változata után két és fél évvel a legradikálisabb gondolat, amivel zárhatok már nem a nyílt forráskód által uralt szoftvervilág víziója, az ma már sok megfontolt, nyakkendős ember számára kézenfekvőnek tűnik.

Arra emlékeztetnék inkább, ami általánosan érvényes lehet a szoftverekre (és talán mindenfajta kreatív vagy szakmai munkára). Az emberi lények általában akkor élveznek egy feladatot, ha az egyfajta optimális kihívási zónába esik, nem elég egyszerű ahhoz, hogy unalmas legyen, de nem is túl nehéz megvalósítani. Az elégedett programozót nem alkalmazzák képességei alatt, és nem is terhelik rosszul megfogalmazott célokkal és stresszes munkafolyamatokkal. *Az élvezet megelőlegezi a hatékonyságot*.

A saját feladatainkkal kapcsolatos félelmeinkre és ódzkodásunkra (még ha olyan ironikus módon is történik, mint Dilbert képregények kiaggatása) úgy kellene tekintenünk, mint a feladat csődjének a jelére. Az élvezet, a humor, a játékosság kétségtelenül tőke. Nem véletlenül említettem a fejezet elején az „elégedett seregeket”, és az sem vicc csupán, hogy a Linux jelképe, kabalája egy ennivaló kis pingvin.

Az is kiderülhet, hogy a nyílt forráskód sikerének legfontosabb hatása az lesz, hogy megtanít mindnyájunkat a kreatív munka gazdaságilag leginkább hatékony módjára.

Epilógus: a Netscape és a bazár

Különös érzés arra ébredni, hogy segítesz történelmet csinálni. 1998. január 22-én, körülbelül hét hónappal *A katedrális és a bazár* első publikálása után a Netscape Communications Inc. bejelentette a Netscape Communicator forráskódjának kiadásáról szóló terveit. Azt végképp nem gondoltam volna, hogy ez már a bejelentés előtti napon meg is történik.

Eric Hahn, a Netscape alelnöke és műszaki vezérigazgatója nem sokkal ezután a következő levelet küldte nekem: „Mindenki nevében itt a Netscape-nél szeretném megköszönni önnek, hogy segített nekünk idáig eljutni. Döntésünket az ön gondolkodásmódja és írásai alapvetően befolyásolták”.

A következő héten a Szilícium-völgybe repültem a Netscape meghívására, egynapos stratégiai konferenciára (1998. február 4-én) a felsővezetőikkel és műszaki szakembereikkel. Együtt terveztük meg a Netcape forráskódkiadási stratégiáját és licencét.

Néhány nappal később a következőket írtam:

> A Netscape a bazár modell nagy és valóságos kísérletét fogja végrehajtani a kereskedelmi világban. A nyílt forráskódú kultúrának szembe kell néznie azzal a veszéllyel, hogy amennyiben a Netscape kísérlete nem válik be, a nyílt forráskódú koncepció értékvesztetté válhat, így a kereskedelmi világ nem nyúl

hozzá egy újabb évtizedig.

Ugyanakkor ez egy látványos lehetőség, a lépést a Wall Streeten és máshol is óvatosan, de pozitívan fogadták. Kaptunk egy esélyt a bizonyításra. Ha a Netscape jelentős piaci részesedést szerez vissza ezzel a lépéssel, akkor az egy régen megkésett forradalmat indíthat el a szoftveriparban.

A következő év bizonyára tanulságos és érdekes lesz.

Az is volt. 2000 nyarán annak a fejlesztése, amit később Mozillának neveztek, csak mérsékelt siker volt. Elérte ugyan a Netscape az eredeti célját, amely a Microsoft megakadályozása volt a böngészőpiac monopollá alakításában, és bizonyos területeken komoly sikereket is elért (különösen az új generációs Gecko megjelenítőmotor kiadásával).

Ugyanakkor nem vonzott jelentős fejlesztői erőt a Netscape-en kívülről, pedig a Mozilla alapítói eredetileg ebben is reménykedtek. A probléma talán az volt, hogy a Mozilla közzététele a bazár modell egyik alapvető szabályát megszegte, nem volt ugyanis benne semmi olyasmi, amit a közreműködők könnyen futtathattak, és működni láthattak. (Több mint egy évvel a kód kiadása utánig a Mozilla forráskódból való felépítéséhez szükség volt a védjegyzett Motif programkönyvtár licencére).

A legnagyobb hiba (kívülről szemlélve) az volt, hogy a Mozilla munkacsoport két és fél évvel a projekt indulása után még mindig nem adott ki használható böngészőt, illetve 1999-ben a projekt egyik vezetője keltett egy kis szenzációt azzal, hogy visszalépett, és a gyenge vezetésre, illetve az elszalasztott lehetőségekre panaszkodott. „A nyílt forráskódnak” – figyelte meg helyesen – „nincs varázsereje”.

Tényleg nincs. A Mozilla hosszú távú kilátásai azonban lényegesen jobbnak tűnnek most (2000 novemberében), mint Jamie Zawinski felmondólevele idejében – az utóbbi néhány hétben az automatikus éjszakai kiadások elérték a használhatóság határát. De Jamie-nek abban igaza volt, hogy a nyílttá válás nem feltétlenül ment meg egy rossz célokkal, rossz kóddal, vagy egyéb szoftvertervezési betegségekkel terhes projektet. A Mozilla egyszerre példája annak, hogy a nyílt forráskód hogyan lehet sikeres és sikertelen.

Időközben a nyílt forráskód egyéb sikereket ért el, és máshol is támogatókra talált. A Netscape kiadása óta hatalmas robbanás történt a nyílt forráskódú modell iránti érdeklődésben, a trendet a Linux operációs rendszer folyamatos sikerei táplálják. A Mozilla által elindított áramlat egyre jelentősebbé válik.

Jegyzetek

Bibliográfia

Köszönetnyilvánítás

Esszém számos emberrel való beszélgetés során fejlődött, akik segítettek hibáinak kijavításában. Különös köszönettel tartozom Jeff Dutkynak <dutky@wam.umd.edu>, aki „hibakeresés párhuzamosíthatósága” megfogalmazást javasolta, és segített a jelenség elemzésében. Köszönöm Nancy Lebovitznak <nancyl@universe.digex.net> a javaslatot, hogy Weinberget kövessem Kropotkin idézésével. Figyelmes kritikát kaptam Joan Eslingertől <wombat@kilimanjaro.engr.sgi.com> és Marty Franztól <marty@net-link.net> az általános technológiai listáról. Glen Vandenburg <glv@vanderburg.org> az önkiválasztás fontosságára mutatott rá a közreműködő populációkban és gyümölcsöző javaslatot tett arra, hogy a sok fejlesztés helyrehozza a „mulasztási hibákat”, míg Daniel Upper <upper@peak.org> pedig természetes analógiákkal szolgált ugyanerre. Hálás vagyok a PLUG-nak, a Philadelphiai Linux-felhasználók Csoportjának, ők voltak esszém első nyilvános változatának tesztközönsége. Paula Matuszek <matusp00@mh.us.sbphrd.com> a szoftvermenedzsment gyakorlati oldaláról világosított fel. Phil Hudson <phil.hudson@iname.com> arra emlékeztetett, hogy a hacker-kultúra társas szerveződése az általuk készített szoftverek felépítését tükrözi és fordítva. John Buck <johnbuck@sea.ece.umassd.edu> rámutatott, hogy a MATLAB az Emacs-hez hasonló példa. Russell Johnston <russjj@mail.com> segített tudatosítani azokat a mechanizmusokat, amelyekről „A sok szem csökkenti a komplexitást” fejezet szól. Végül Linus Torvalds megjegyzései és korai hozzájárulása rendkívül bátorító volt.