# Катедралата и базарът

Ерик С. Реймънд <esr@thyrsus.com>, превод: Никола Колев <nikky@sdf.lonestar.org> и Калоян Доганов <kaloian@europe.com>

8 август 1999, версия на превода: 26 декември 2000

Разглеждам подробно един успешен проект с отворен код – fetchmail, който беше преднамерено стартиран като проверка на някои удивителни теории в софтуерното инженерство, подсказани от историята на Linux. Обсъждам тези теории в светлината на два различни стила на разработка – „катедралният“ модел, използван широко в комерсиалния свят, противопоставен на „базарния“ модел от Linux света. Посочвам, че тези модели произлизат от противоположни основни положения относно естеството на задачата по отстраняване грешките в софтуера. След това давам аргумент, подкрепен от Linux опита, че „Когато има достатъчно очи, всички грешки изплуват на повърхността“, предлагам продуктивни аналогии с други самокоригиращи се системи от самостойни агенти, и завършвам с малко изследване на следствията от това прозрение за бъдещето на софтуера.

## Съдържание

# 1. Катедралата и базарът

Linux е подмолен. Кой би си помислил само преди пет години (1991), че от хакерските занимания в свободното време на няколко хиляди разработчици, пръснати из планетата, и свързани единствено от тънките нишки на Интернет, като на магия ще се получи операционна система от световен клас?

Определено не и аз. По времето, когато Linux се появи на радара ми в началото на 1993, аз се занимавах с Unix и разработка на софтуер с отворен код вече 10 години. По средата на 80-те години на 20-ти век бях един от първите сътрудници на GNU. Бях пуснал доста свободен софтуер в мрежата, разработвайки или съразработвайки няколко програми (сред които nethack, режими VC и GUD на Emacs, xlife и други), които и днес все още са широко употребявани. Смятах, че зная как стават тези неща.

Linux преобърна голяма част от това, което си мислех, че зная. От години проповядвах Unix евангелието на малките инструменти, бързото създаване на прототипи и еволюционното програмиране. Вярвах обаче, че има определена критична сложност, отвъд която е необходим по-централизиран, априорен подход. Вярвах, че най-важният софтуер (операционните системи и наистина големите инструменти като Emacs) трябва да бъде съграждан като катедрали – внимателно майсторени от отделни вълшебници или малки групички магове, работещи в прекрасна изолация, като никоя бета версия не бива пусната преди да й дойде времето.

Стилът на разработка на Линус Торвалдс – „пускай рано и често; възлагай на други желаещи всичко, което можеш; бъди открит до степен на безразборност“, ми дойде изненадващо. Нямаше го тихото, благоговейно катедрално разработване – Linux обществото по-скоро ми приличаше огромен шумен базар с различни планове и подходи (доста точно уподобени на от местата с Linux архиви, които приемат софтуер от *всеки*), от който само по чудо би могла да се роди съгласувана и стабилна операционна система.

Фактът, че този базар като че ли работеше, при това добре, ми дойде като гръм от ясно небе. Докато работех усилено по индивидуални проекти, а освен това се опитвах да разбера защо Linux светът не само че не се разпада, но и като че ли става все по-силен и по-силен, при това със скорост, за която „катедралните строители“ едва ли биха могли да мечтаят.

Към средата на 1996 вече мислех, че започвам да разбирам. Случайно се откри идеална възможност да изпитам теорията си под формата на проект с отворен код, който съзнателно се опитах да управлявам в „базарен“ стил. Така и направих – имаше значителен успех.

Това е историята на този проект. Ще я използвам, за да предложа някои афоризми относно ефективната разработка на отворен код. Не всички неща научих от Linux света, но ще видите как той им придаде определен смисъл. Ако съм прав, те ще ви помогнат да разберете кои са нещата, които правят Linux обществото истински извор на добър софтуер, и вероятно ще ви помогнат сами да станете по-продуктивни.

# 2. Пощата трябва да пристигне

От 1993 година насам ръководя техническия отдел на един малък, свободно достъпен Интернет доставчик, наречен Chester County Interlink (CCIL), намиращ се в Уест Честър, Пенсилвания. (Аз съм един от съоснователите на CCIL и автор на нашия уникален многопотребителски софтуер за таблото за обяви – можете да му хвърлите едно око чрез telnet до *locke.ccil.org* <telnet://locke.ccil.org>. Днес той поддържа почти три хиляди потребители на тридесет линии.) Работата ми позволяваше денонощен достъп до мрежата през линията на CCIL с 56K, а на практика дори и го изискваше!

Постоянно изпитвах възпиюща нужда от Интернет поща. Поради различни сложни причини беше трудно да се подкара SLIP между домашната ми машина (snark.thyrsus.com) и CCIL. Когато най-накрая успях, открих че е доста досадно периодично да правя telnet до locke, за да си проверя пощата. Исках тя ми да се доставя на snark, така че да бъда уведомяван при пристигането й, и да мога да я обработвам с всичките си налични инструменти.

Простото препращане със sendmail не ми вършеше работа, тъй като личната ми машина не беше

винаги в мрежата и нямаше постоянен IP адрес. Имах нужда от програма, която да може да се свърже чрез моята SLIP връзка и през нея да изтегли пощата ми, която да бъде доставена на място. Знаех, че съществуват такива решения, и повечето от тях използват един прост приложен протокол, наречен POP (Post Office Protocol). Така или иначе в операционната система BSD/OS на locke вече беше включен готов POP3 сървър.

Нуждаех се от POP3 клиент. Така че влязох в мрежата и открих един. По-точно, открих три-четири. Използвах pop-perl за известно време, но му липсваше свойство, което изглеждаше очевидно: възможността да пренаписва адресите на източената поща, така че когато отговарям, всичко да сработва правилно.

Проблемът беше следният – да предположим, че някой с име 'joe' ми е изпратил поща от locke. Ако аз източа пощата и сетне се опитам да му отговоря, моят пощенски софтуер бодро би се опитал да я достави на несъществуващия 'joe' на snark. Ръчното редактиране на адреса при отговор, за да се добави '@ccil.org' бързо започна да се превръща в голяма мъка.

Това очевидно беше нещо, което компютърът трябва да прави вместо мен. Но никой от съществуващите POP клиенти не знаеше как! И това ни довежда до първата поука:

*1. Всеки качествен софтуер възниква, за да начеше крастата на разработчика.*

Може би това трябваше да е очевидно (от край време е известно, че „Необходимостта е майка на изобретателността"), но твърде често софтуерните разработчици срещу заплащане прекарват дните си в досадна работа върху програми, от които нито имат нужда, нито намират удоволствие. Но не и в света на Linux – което може да обясни защо е толкова високо общото ниво на качество на софтуера, създаден в Linux обществото.

И какво мислите – да не би веднага да се хвърлих в буйния вихър, за да кодирам чисто нов POP3 клиент, който да се конкурира със съществуващите? В никакъв случай! Внимателно огледах POP инструментите, които имах на разположение, задавайки си въпроса: „Кой от тях е най-близо до това, което искам?". Защото:

*2. Добрите програмисти знаят какво да пишат. Великите знаят какво да пренаписват (и преизползват).*

Не претендирам да съм велик програмист, но поне се опитвам да подражавам на такъв. Характерен белег на великите програмисти е градивният мързел. Те знаят как да получат нещо не с оглед на усилията, които трябва да се вложат, а с оглед на резултатите, и знаят че почти винаги е по-лесно да се започне от добро частично решение, вместо от нищо.

Например *Линус Торвалдс* <http://www.tuxedo.org/esr/faqs/linus> всъщност не се опитва да напише Linux от нищото. Вместо това, той започва да преизползва код и идеи от Minix, една малка подобна на Unix операционна система за PC клонинги. В края на краищата, целият код на Minix отпада или е пренаписан изцяло – но докато е там, той осигурява скеле, опора за пеленачето, което впоследствие ще се превърне в Linux.

В същия дух, аз се оглеждах за съществуващ POP инструмент, който да е сравнително добре кодиран, с цел да го използвам като основа за разработка.

Традицията на споделяне на код в Unix света винаги е била благосклонна към преизползването на кода (тъкмо затова проектът GNU избра Unix като основна операционна система, въпреки сериозните резерви спрямо самата нея). Linux светът доведе тази традиция почти до технологичните й граници – съществуват терабайти от общодостъпен отворен код. Така че отделянето на време за търсене на нечие друго „почти достатъчно" решение, е по-вероятно да ви донесе добри резултати в Linux света, отколкото където и да е другаде.

Поне така стана при мен. Заедно с тези, които намерих по-рано, моето второ претърсване даде общо девет кандидата – fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail и upop. Първия на който се спрях, беше 'fetchpop' от Seung-Hong Oh. Добавих моето свойство за пренаписване на заглавната част, и направих разни други усъвършенствания, които авторът прие в своето издание 1.9.

Няколко седмици по-късно, обаче, попаднах на кода на 'popclient' от Карл Харис, и открих, че имам проблем. Макар че fetchpop носеше някои добри оригинални идеи (например своя daemon режим), той

можеше да обработва само POP3 и беше кодиран твърде аматьорски (по това време Seung-Hong беше блестящ, но неопитен програмист, и двата белега на това си личаха). Кодът на Карл беше по-добър – напълно професионален и надежден. Но на неговата програма ѝ липсваха няколко важни и сложни за реализация свойства на fetchpop (включително тези, които бях кодирал самият аз).

Да остана или да се прехвърля? Ако се прехвърлех, щях да изоставя работата по кода, която вече бях свършил, в замяна на по-добра основа за разработка.

Практически мотив за прехвърлянето беше поддръжката на множество протоколи. POP3 е най-често използваният, но не единствен пощенски протокол. Fetchpop и другите конкуренти не разбираха нищо от POP2, RPOP, или APOP, а аз вече имах смътни помисли за възможността просто за кеф да добавя *IMAP* <http://www.imap.org> (Internet Message Access Protocol, най-модерно проектираният и най-мощен пощенски протокол).

Но аз имах и едно по-скоро теоретично основание да мисля, че прехвърлянето може да е много добра идея. Това е нещо което научих много преди Linux.

> *3. „Свиквай с мисълта, че един опит ще отиде на вятъра – така или иначе това ще се случи.“ (Фред Брукс, „Митичният човеко-месец“, Глава 11)*

Казано другояче, човек често не разбира проблема, докато не завърши с първата реализация на решението му. Вторият път може би ще знае как да го направи правилно. Така че ако иска да го схване, трябва да бъде готов да започне отначало *поне* веднъж (виж. глава 15 [JB]).

Е, добре – казах си – промените във fetchpop бяха моят първи опит. И тъй, аз се прехвърлих.

След като на 25 юни 1996 изпратих първия си набор от поправки до Карл Харис, открих, че всъщност той вече е загубил интерес към popclient. Кодът беше малко изоставен, с висящи незначителни грешки. Аз имах да прилагам много промени, и бързо се споразумяхме, че най-логичното, което мога да сторя, е да поема програмата.

Всъщност без да забележа, проектът дръпна напред. Вече не просто гласях малки промени за един съществуващ POP клиент. Аз поех поддръжката на целия клиент, и в главата ми имаше бълбукащи идеи, които знаех, че може би ще доведат до големи промени.

В една софтуерна култура, която насърчава споделянето на код, това е естественият начин за развитието на един проект. Действах съобразно това:

> *4. Ако имаш подходящата нагласа, интересните проблеми сами ще те намерят.*

Нагласата на Карл Харис обаче беше дори по-важна. Той разбра, че:

> *5. Когато изгубиш интерес към една програма, последният ти дълг към нея е да я оставиш в ръцете на способен наследник.*

Без дори да го обсъждаме, Карл и аз разбирахме, че имаме една обща цел – да намерим най-доброто решение. Единственият въпрос и за двама ни беше дали мога да докажа, че при мен програмата ще остане в добри ръце. След като го сторих, той постъпи достойно и бързо ми я предаде. Надявам се да постъпя по същия начин, когато дойде и моя ред.

## 3. Колко е важно да имаш потребители

И така, аз наследих popclient. по-важното е обаче, че наследих и потребителската му база. Прекрасно е да имаш потребители – при това не само защото те са олицетворението на нужда, която да задоволяваш, не само защото правиш нещо добро. Обработени по подходящ начин, те могат да се превърнат в съразработчици.

Друга от силните страни на Unix традицията, която Linux развива до крайност е, че много от самите потребители също са хакери. Понеже изходния код е достъпен, те могат да бъдат *ефективни хакери*, а това е много полезно за скъсяване времето за отстраняване на грешки. Ако се окуражат малко, потребителите ви ще откриват проблеми, ще предлагат поправки и ще ви помагат да подобрявате кода много по-бързо, отколкото бихте се справили сами.

> *6. Да се отнасяте към потребителите си като към съразработчици е най-безпроблемният начин бързо да усъвършенствате кода и да направите отстраняването на грешки по-ефективно.*

Лесно е да се подцени мощта на този ефект. Всъщност всички ние от обществото на отворения код драстично подценявахме колко добре би се мащабирал по-големият брой потребители относно сложността на системата, докато Линус Торвалдс не ни показа обратното.

Всъщност най-умното и значително нещо, което Линус направи беше не толкова конструирането на самото ядро на Linux, колкото изнамирането на модела за разработка на Linux. Веднъж, когато в негово присъствие изразих това мнение, той се усмихна, и тихо повтори нещо, което е казвал много пъти: „Всъщност аз съм един доста мързелив тип, и обичам да получавам заслуги за неща, които са свършили други хора“. Мързелив като лисица. Или както Робърт Хайнлайн беше описал един от своите герои – „прекалено мързелив, за да не сполучи“.

Погледнато в ретроспекция, в разработката на библиотеката GNU Emacs Lisp и на архивите с Lisp код може да се види прецедент за методите и успеха на Linux. Обратно на „катедралния“ стил на изграждане на C ядрото на Emacs и повечето други FSF инструменти, еволюцията на кодовата база на Lisp беше съвсем плавна и направлявана най-вече от потребителите. Идеите и прототипните предложения често бяха пренаписвани по три-четири пъти, преди да стигнат до стабилна крайна форма. А свободно формираните сътрудничества в стил Linux през Интернет бяха доста чести.

И наистина, моята собствена най-успешна разработка преди fetchmail вероятно е режимът VC на Emacs, разработка в стил Linux между мен и още трима души, само един от които (Ричард Столман, автор на Emacs и създател на *Free Software Foundation* <http://www.fsf.org>) бях срещал до този момент. Това беше фронт-енд на SCCS, RCS а по-късно и на CVS от Emacs, който предлагаше операции по контрол на версиите с „едно докосване“. Разви се от малкия и необмислен sccs.el режим, написан от някой друг. А разработката на VC имаше успех, защото за разлика от самия Emacs, Lisp кода му можеше много бързо да преминава през етапите на пускане/тестване/подобрения.

# 4. Пускай рано. Пускай често.

Ранните и чести издания са критична част от модела за разработка на Linux. По-рано, повечето разработчици (включително и аз) вярваха, че тази политика не върши работа при проекти по-големи от тривиални, защото ранните издания почти винаги са пълни с грешки, а човек не би искал да изчерпва търпението на потребителите си.

Тази вяра подсили общото отдаване на катедралния стил на разработка. Ако най-важното беше потребителите да виждат възможно най-малко грешки, то защо да не пускаш по едно ново издание на всеки шест месеца, а през останалото време да бачкаш като луд по отстраняването на грешките. С ядрото на Emacs беше разработено по този начин. Lisp библиотеката, по обратния – защото имаше активни Lisp архиви извън контрола на FSF, където човек би могъл да отиде, за да намери нови версии на кода независимо от цикъла на пускане на Emacs (виж. глава 15 [QR]).

Най-важният от тях – elisp архивът на щата Охайо, носеше духа и много от характерните черти на днешните големи Linux архиви. Малцина от нас обаче се замисляха какво правим, или какво точно предполагаше за проблемите в катедралния стил на разработка на FSF самото съществуване на този архив. Около 1992-а направих един сериозен опит формално да вкарам в официалната Emacs библиотека голяма част от кода на Охайо. Заради това обаче имах неприятности от политическо естество и общо взето нямах успех.

Една година по-късно обаче, когато Linux започна да се откроява все повече, ставаше ясно, че се случва нещо много по-различно и здравословно. Политиката за отворена разработка на Линус беше пълна противоположност на катедралния модел. Архивите Sunsite (след това *Metalab* <http://metalab.unc.edu>, а по-късно и *Ibiblio* <http://www.ibiblio.org>) и tsx-11 се разраствaха, можеха да се намерят множество дистрибуции. И всичко това направлявано от нечувана до този момент честота на пускане на основната система.

Линус се отнасяше към своите потребители като към съразработчици по възможно най-ефективния начин:

*7. Пускай рано. Пускай често. И слушай клиентите си.*

Откритието на Линус беше не толкова че правеше това (в Unix-света дълго време преди това имаше подобна традиция), колкото в усилването му до степен, еквивалентна на сложността на това, което разработваше. В онези ранни времена (около 1991-а) нерядко му се случваше да пуска ново ядро повече от веднъж на *ден*! И тъй като той култивираше базата си от съразработчици и използваше Интернет за съвместна работа по-здраво от всеки друг, това сработваше.

*Как* обаче? И дали беше нещо, което мога да повторя, или се осланяше на някаква уникална гениална черта на Линус Торвалдс?

Не мислех така. Линус наистина е дяволски добър хакер (колцина от нас биха могли да разработят цяло ядро на операционна система с продуктово качество?). Но Linux не представляваше някаква гигантска стъпка напред в концепцията. Линус не е (поне все още не е) изобретателен гений на дизайна каквито са, да кажем, Ричард Столман или Джеймс Гослинг (създали съответно NeWS и Java). Линус по-скоро ми изглеждаше като гений на инженерството с шесто чувство за избягване на грешките, задънените улици в разработката, и с истински талант да намира най-лесния път от точка А до точка Б. И наистина целият дизайн на Linux е пропитс това качество, и отразява общо взето консервативния и опростяващ подход на Линус.

И така, ако бързите издания и използването на Интернет-средата до максимум бяха не случайни, а неделими части от прозрението на инженерния гений на Линус към пътя, изискващ най-малко усилия, какво усилваше той? Какво изчовъркваше от машинарията?

Изказан по този начин, въпросът е реторичен. Линус постоянно стимулираше и възнаграждаваше своите хакери/потребители – стимулираше ги възможността да свършат някаква част от работата, задоволяваща егото им, а ги възнаграждаваше гледката на постоянното (дори *всекидневно*) подобряване на работата им.

Линус пряко целеше да увеличи човеко-часовете, хвърлени в отстраняването на грешки, дори и това да стане за сметка на нестабилност на кода и яд у потребителите, ако някоя грешка не може да се последи. Линус се държеше, като че вярваше в следното:

*8. Ако има достатъчно голяма база от бета-изпитатели и съразработчици, почти всеки проблем ще бъде характеризиран бързо, а поправянето му ще бъде очевидно за някой.*

Или казано не така формално – „Ако има достатъчно очи, всички грешки се виждат“. Аз го наричам „Законът на Линус“.

Моята първоначална формулировка беше, че всеки проблем ще е „очевиден за някого“. Линус обаче смяташе, че този, който разбере и оправи проблема, не е задължително, нито обикновено същият, който го характеризира. „Някой вижда проблема“, казва той, „а някой *друг* го разбира. Продължавам да твърдя, че да откриеш е по-голямото предизвикателство“. Важното е обаче, че и двете неща се случват бързо.

Тук според мен се намира основната разлика между катедралния и базарния стил. В „катедралните“ възгледи за програмирането грешките и проблемите при разработка са тайнствени, коварни, дълбоки феномени. Нужни са месеци усърдна работа от посветените малцина, докато се добие увереност, че всичко е чисто. Оттам идват и дългите интервали между изданията и неизбежните разочарования, когато дългоочакваните издания не са съвършени.

При базарния възглед за нещата, от друга страна, се приема, че грешките общо взето са просто явление, или поне се оказват прости, когато бъдат изложени пред хилядите нетърпеливи съразработчици, които се нахвърлят на всяко ново издание. Съответно се правят по-чести издания, за да има повече поправки, като една от добрите страни на това е, че имаш да губиш по-малко, ако някоя грешка случайно успее да се промъкне незабелязана.

Това е всичко. И е достатъчно. Ако Законът на Линус не е верен, то всяка друга система, сложна като ядрото на Linux, и по която „хакерстват“ толкова много ръце както при ядрото на Linux, би

трябвало в един момент да се срине под тежестта на непредвидени лоши взаимодействия и неоткрити „дълбоки“ грешки. Ако обаче е верен, то е достатъчно да обясни относителната липса на грешки в Linux и продължителното му непрекъснато време на работа, стигащо до месеци и дори години.

Всъщност това може би не трябва да ни изненадва. Още преди години социолозите са открили, че усредненото мнение на група от еднакво учени (или еднакво невежи) наблюдатели е доста по-благонадеждно, отколкото това на един случайно избран наблюдател.

Те наричат това „Делфийски ефект“. Явно Линус показва, че това може да се приложи дори и при разработката на операционна система – че Делфийският ефект може да „укроти“ сложността на разработката дори и при ниво на сложност като това на ядро на операционна система.

Една специфична характеристика на ситуацията с Linux, която много допринася освен „Делфийския ефект“, е фактът, че работещите по всеки даден проект се самоизбират. Преди си кореспондирах с един човек, който посочи, че контрибуции се получават не от случайно подбрани хора, а от такива, които са достатъчно заинтересувани да използват софтуера, да научат как работи, да се опитат да намерят решения на проблемите, които срещат, и всъщност да направят несъмнено разумна поправка. Всеки, който отговаря на тези условия, е много вероятно да има какво да допринесе.

Задължен съм на моя приятел Джеф Дътки <dutky@wam.umd.edu>, задето ми посочи, че Законът на Линус може да бъде перифразиран като „Отстраняването на грешки е паралелизируемо“. Джеф е забелязал, че въпреки необходимостта хората, отстраняващи грешки, да поддържат връзка с някой координиращ разработчик, не е необходима кой-знае каква координация между самите тях. Следователно отстраняването на грешки не изпада в същата квадратична сложност и увеличение на разходите, които правят прибавянето на разработчици проблематично.

На практика теоретичната загуба на производителност поради дублиране на работата почти никога не е проблем в Linux света. Един от ефектите, следващи политиката „пускай рано и често“ се състои в минимизирането на подобно дублиране чрез бързото разпространяване на върнатите поправки (виж. глава 15 [JH]).

Брукс дори е направил импровизирано наблюдение, свързано с това на Джеф: „Общата стойност на поддържането на широко използвана програма обикновено е 40% или повече от стойността на разработката й. Изненадващото е, че тази стойност се влияе силно от броя на потребителите. *Колкото повече потребители има, толкова повече грешки откриват те.*“ (курсивът е мой).

Повечето потребители откриват повече грешки, защото добавянето на още потребители добавя и множество различни начини за изпитване на програмата. Този ефект се усилва, когато потребителите са и съразработчици. Всеки един подхожда към задачата да открие грешки с малко по-различна настройка и аналитичен модел от останалите, с различен ъгъл на проблема. Изглежда „Делфийският ефект“ работи точно именно поради тази разлика. В специфичния контекст на отстраняването на грешки, разликата освен това има тенденцията да намалява дублирането на труда.

Затова добавянето на повече бета-изпитатели може и да не намали сложността на настоящата „най-дълбока“ грешка от гледна точка на *разработчиците*, но увеличава вероятността нечий инструментариум да пасне на проблема по такъв начин, че грешката да стане очевидна *за този човек*.

Линус също залага на това. В случай че *съдържа* сериозни грешки, версията на ядрото на Linux се определя по такъв начин, че потенциалните потребители могат да избират дали да използват последната версия, определена като „стабилна“, или да се движат по острието на бръснача и да рискуват с грешките за сметка на някои нови функции. Повечето Linux хакери все още не подражават формално на тази тактика, а може би трябва; фактът, че са достъпни и двата избора ги прави по-привлекателни.

## 5. Кога розата не е роза?

Докато изучавах поведението на Линус и съставях теория защо то биде успешно, съвсем съзнателно реших да проверя тази теория върху моя нов (несъмнено съвсем не толкова сложен и амбициозен) проект.

Но първото което сторих, бе значително да реорганизирам и опростя popclient. Реализацията на Карл Харис беше в много добро състояние, но й личеше един особен вид ненужна сложност, типична

за много програмисти на C. Карл третираше кода като най-важен, а структурите от данни бяха като поддръжка за кода. В резултат на това, кодът беше прекрасен, но замисълът на структурите от данни беше непоследователен и твърде грозен (поне за високите стандарти на такъв стар LISP хакер като мен).

Освен за да усъвършенствам кода и дизайна на структурите от данни, имах и друга причина за това пренаписване. Така щях да ги развия в нещо, което разбирам напълно. Никак не е весело да отговаряш за поправянето на грешки в програма, която не разбираш.

През първия месец и нещо аз просто изпълнявах докрай следствията от основния дизайн на Карл. Първата сериозна промяна, която направих, беше да добавя поддръжка на IMAP. Сторих това като реорганизирах машините на протоколи в един общ драйвер и три таблици на методите (за POP2, POP3, и IMAP). Тази промяна, както и предишните, илюстрира един общ принцип, който е хубаво програмистите да имат наум, особено при езици като C, които по природа не извършват динамично типизиране:

> *9. Интелигентните структури от данни и тъпият код работят много по-добре, отколкото в обратния случай.*

Брукс, Глава 9: „Покажи ми [кода] си и скрий [структурите от данни], и аз ще продължавам да съм объркан. Покажи ми [структурите си от данни], и обикновено няма да имам нужда от твоя [код]; ще е очевидно."

В действителност, той е казал „блок-схеми" и „таблици". Но отчитайки тридесетгодишно терминологично/културно изместване, смисълът е почти същия.

От този момент (в началото на септември 1996, някъде около шест седмици от самото начало) започнах да мисля, че е редно да се смени името – в края на краищата, това вече не беше просто POP клиент. Но се колебаех, защото досега в дизайна нямаше нищо истински ново. Моята версия на popclient все още трябваше да развие своя собствена идентичност.

Това коренно се промени, когато fetchmail се научи как да препраща взетата поща към SMTP порт. Ще се върна на това след малко. По-напред туй: по-горе казах, че реших да използвам този проект, за да изпробвам теорията си за това какво свястно нещо е направил Линус Торвалдс. Как – с право може би ще попитате – сторих това? По следния начин:

1. Пусках рано и често (почти никога по-рядко от всеки десет дни; веднъж дневно през периодите на интензивна разработка).
2. Разширявах моя бета списък, добавяйки в него всеки, който се е свързал с мен относно fetchmail.
3. Изпращах приятелски известия в бета списъка, когато пусках нова версия, окуражавайки хората да участват.
4. Вслушвах се в моите бета-изпитатели, допитвах се до тях за решения в дизайна и ги хвалех всеки път, когато изпращаха кръпки и изказваха мнения.

Отплатата за тези прости оценки беше незабавна. От началото на проекта получих толкова качествени доклади за грешки, за които повечето разработчици биха извършили убийство – често дори прикрепени с добри поправки. Получих смислени критики, получих поща от почитатели, получих интелигентни предложения за нови свойства на програмата. Което довежда до:

> *10. Ако се отнасяш към своите бета изпитатели като към свой най-ценен ресурс, те ще ти отвърнат, превръщайки се в най-ценния ти ресурс.*

Интересен критерий за успеха на fetchmail е самият размер на бета списъка на проекта – броят на приятелите на fetchmail. По времето, когато пиша това, той има 249 членове и расте с двама-трима седмично.

Всъщност, тъй като го ревизирах в края на май 1997, поради една интересна причина списъкът започна да губи членове след като достигна своя връх от около 300. Няколко души ме помолиха да ги отпиша, тъй като fetchmail им служел толкова добре, че те вече нямали нужда да следят трафика от списъка! Може би това е част от нормалния цикъл на живот на един зрял проект от базарен тип.

# 6. Popclient става fetchmail

Истинският повратен момент в проекта настъпи, когато Хари Хокхейзър ми изпрати своя нахвърлян код за препращане на поща към SMTP порта на клиентската машина. Почти веднага осъзнах, че една надеждна реализация на това свойство би превърнала в почти ненужни всички други режими на доставка.

От много седмици човърках fetchmail, най-вече за да добавям разни неща, продължавайки да чувствам, че дизайнът на интерфейса върши работа, но е мърляв, тромав, с много незначителни опции, висящи навсякъде. Особено ме дразнеха опциите за стоварване на изтеглената поща в пощенски файл или към стандатния изход, но не можех да разбера защо.

Това, което видях, когато се замислих за SMTP препращането, беше че popclient се опитваше да прави твърде много неща. Той е бил проектиран да бъде хем агент за пренасяне на поща (MTA - Mail Transport Agent), хем агент за доставяне на място (MDA - Mail Delivery Agent). Чрез SMTP препращането, той можеше да излезе от сферата на MDA и да стане чист MTA, оставяйки доставката на място да бъде свършена от други програми, точно както го прави sendmail.

Защо да се оплескваме с цялата сложнотия по настройването на агент за доставяне на поща или установяването на една пощенска кутия като заключена само за добавяне, щом е почти сигурно, че на първо място порт 25 го има на всяка платформа, която поддържа TCP/IP? Особено когато това означава, че получената поща ще изглежда като съвсем обикновена, пусната от изпращача SMTP поща, което впрочем е точно това, което искаме.

Тук има няколко поуки. Първо, идеята за SMTP-препращането е най-голямата отплата, която получих от съзнателния ми опит да подражавам на методите на Линус. Един потребител ми даде тази страшна идея – всичко, което трябваше да сторя, беше просто схвана следствията от нея.

*11. Щом нямаш добри идеи, тогава разпознавай добрите идеи на своите потребители. Понякога второто е за предпочитане.*

Любопитното е, че вие бързо ще откриете, че ако сте напълно и от все сърце честен за това колко много дължите на другите хора, светът като цяло ще се отнася към вас все едно самите вие сте създали всяка частица от изобретението и сега просто скромничите за своя вроден гений. Всички виждаме колко добре се получи това при Линус!

(Когато се изказвах на Perl конференцията през август, 1997, Лари Уол беше на първия ред. Когато стигнах до горния пасаж, той се обади в религиозно-възрожденски стил: „Кажи го, кажи го, брате!“. Цялата аудитория се засмя, защото знаеха, че това се получи също и при създателя на Perl.)

След като няколко седмици движех проекта в същия дух, започнах да получавам подобни похвали, и то не само от моите потребители, но и от други хора, които са го дочули. Скатах настрана малко от тази електронна поща. От време на време си я поглеждам, ако случайно започна да се чудя дали животът ми си е струвал. :-)

Но тук има две по-фундаментални, неполитически поуки, които са общи за всички видове дизайн.

*12. Често най-поразителните и новаторски решения идват след откритието, че представата ти за проблема е погрешна.*

Аз съм се опитвал да реша грешен проблем, като съм продължавал да разработвам popclient като комбиниран MTA/MDA с всички изчанчени видове доставяне на място. Дизайнът на fetchmail се нуждаеше от основно преосмисляне като чист MTA, като част от нормалния SMTP-говорящ път на Интернет пощата.

Когато удариш на камък при разработка, тогава откриваш, че е трудно да мислиш по-напред от следващата кръпка. Често това означава, че е време не да се запиташ дали разполагаш с правилния отговор, а дали задаваш правилния въпрос. Може би проблемът трябва да бъде преосмислен.

И тъй, аз преосмислих моя проблем. Накратко, правилното нещо, което трябваше да се свърши беше: (1) в общия драйвер да се добави поддръжка на SMTP препращане, (2) то да се установи като режим по подразбиране, и (3) накрая да се изхвърлят всички други режими на доставка, особено възможностите за доставяне във файл и доставяне към стандартния изход.

За известно време се колебаех относно стъпка 3, страхувайки се да не разстроя някои дългогодишни потребители на popclient, зависещи от другите механизми на доставка. На теория, те могат незабавно да превключат към .forward файлове или към техните не-sendmail еквиваленти, за да получат същите резултати. На практика преходът би могъл да бъде объркващ.

Но след като го сторих, ползата беше огромна. Изчезнаха най-претруфените части от кода за доставка. Настройката стана много по-проста – повече нямаше умилкване около системния MDA и потребителската пощенска кутия, нямаше повече грижи за това дали лежащата отдолу операционна система поддържа заключване на файлове.

Също така изчезна единственият начин да се загуби поща. Ако сте посочили доставка във файл и дискът се препълни, пощата ви се губи. Това не може да се случи със SMTP препращането, защото програмата, която слуша на SMTP порта няма да върне OK, докато съобщението не може да бъде доставено, или поне съхранено за доставка по-късно.

Подобри се и производителността (макар и да не се забелязва при едно единствено изпълнение). Друга незначителна полза от тази промяна беше, че страницата с ръководството стана много по-проста.

По-късно открих, че трябва да върна отново функцията за доставка чрез посочен от потребителя MDA, за да е възможно обработването на някои неясни ситуации, усложнени от динамичния SLIP. Но намерих много по-прост начин да го сторя.

Каква е поуката? Не се колебай да изхвърляш остарелите свойства, щом можеш да го направиш без загуба на ефективност. Антоан дьо Сент-Екзюпери (авиатор и проектант на летателни апарати, когато не е бил автор на класически книги за деца) е казал:

> *13. „Съвършенството (в дизайна) е постигнато не когато няма какво повече да се добави, а когато няма какво повече да се отнеме.“*

Когато кодът ви става хем по-добър, хем по-прост, тогава *знаете*, че е тъй. И в този процес, дизайнът на fetchmail придоби своя собствена идентичност, различна от наследения popclient.

Беше време за смяна на името. Новият дизайн изглеждаше много повече като двойник на sendmail, отколкото стария popclient. И двата са MTA, но докато sendmail избутва и сетне доставя, новият popclient издърпва и доставя. И тъй, след два месеца колебание, аз го преименувах на fetchmail.

В историята за това как SMTP доставката се появи във fetchmail се крие една по-обща поука. Успоредно може да бъде не само отстраняването на грешки. Разработката и (може би в удивителна степен) изследването на пространството на дизайна също могат. Когато начинът ви на разработка е рязко итеративен, разработката и подобряването на софтуера могат да се превърнат в частен случай на отстраняването на грешки, т.е. поправяне на „грешки поради недостиг“ в първоначалните възможности или в концепцията за софтуера.

Дори и при едно по-високо ниво на дизайн, може да е много ценно да имаш мисълта за много съразработчици, разхождащи се напосоки из пространството на дизайна около твоя продукт. Нещо като локва с вода, която намира пролука, от където да се процеди, или по-скоро както мравките намират храна – изследване чрез дифузия, последвано от изследване, опосредявано от мащабируем комуникационен механизъм. Това работи много добре. Подобно на случая с Хари Хокхейзър и мен, някой от вашите съратници може лесно да се доберe до огромен пробив наблизо, докато вие сте фокусирали на твърде близък кадър, за да го забележите.

# 7. Fetchmail пораства

Така се оказах със спретнат и иновативен дизайн; с код, за който знаех че работи добре, понеже го използвах всекидневно, и с набъбващ списък от бета потребители. Постепенно ми стана ясно, че вече не съм ангажиран с незначителна лична джунджурийка, която евентуално може да бъде полезна другиму. В ръцете си държах програма, от която действително се нуждаеше всеки хакер с Unix система и пощенска връзка през SLIP/PPP.

С възможността за SMTP препращане, програмата дръпна много пред конкуренцията, с изгледи да

се превърне в „убиец в категорията си“ – една от онези класически програми, които толкова плътно запълват своята ниша, че алтернативите не само биват изоставяни, но и почти забравяни.

Мисля, че не можете действително да целите или планирате подобен резултат. До него трябва да ви докарат такива силни дизайнерски идеи, че впоследствие резултатите просто изглеждат неизбежни, естествени, дори предопределени. Единственият начин да се домогнете до такива идеи е да разполагате с много идеи. Или да разполагате с инженерна преценка, за да схващате чуждите добри идеи отвъд границите, до които авторите им са предполагали, че е възможно да се отиде.

Анди Таненбаум е имал първоначалната идея да построи една проста нативна Unix система за IBM PC, която да се използва като учебно пособие. Линус Торвалдс е тласнал концепцията за Minix по-напред, отколкото Андрю изобщо е предполагал, че е възможно да отиде. И тя се превърна в нещо удивително. По същият начин (макар и в по-малък мащаб), аз взех някои идеи от Карл Харис и Хари Хоккейзър и ги тласнах силно. Никой от нас не беше „оригинален“ в романтичния смисъл, според който хората мислят за гения. Напук на хакерската митология, повечето научни, инженерни и софтуерни разработки не са дело на някой оригинален гений.

Резултатите водеха до едно и също шеметно опиянение – всъщност, точно това е успехът, заради който живее всеки хакер! И това означаваше, че трябваше да вдигна летвата още по-високо. За да направя fetchmail толкова добър, колкото тогава видях че може да бъде, трябваше да пиша не само за собствените си нужди, но също така да включвам и поддържам възможности извън моята орбита, но необходими за други хора. И да го правя тъй, че да държа програмата проста и ясна.

Първата и изумително важна възможност, която написах след като осъзнах горното, беше поддръжката на множествено пускане – възможността да се изтегля поща от пощенски кутии, които са насъбрали всичката поща за група потребители, и после всяко съобщение да се маршрутизира до индивидуалните му получатели.

Режих да добавя поддръжката на множествено пускане отчасти защото някои потребители настояваха за нея, но най-вече защото мислех, че това ще изтръска грешките от кода за еднично пускане, принуждавайки ме да работя с адресирането в пълната му всеобщност. Така и стана. Сколасването на правилно граматично анализиране според *RFC 822* <http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt> ми отне забележително много време, и то не защото всяка отделна негова част е трудна, а понеже е заплетена цяла камара от взаимнозависими и мъгляви подробности.

Но адресирането с множествено пускане също се оказа отлично дизайнерско решение. Ето как разбрах това:

> *14. Всеки инструмент трябва да бъде полезен по очаквания начин, но един истински велик инструмент се употребява по начини, които никога не си очаквал.*

Неочакваната употреба на поддържащия множествено пускане fetchmail беше изпълнението на пощенски списъци, където списъкът се пазеше на и псевдонимите се обработваха от *клиентската* страна на SLIP/PPP връзката. Това означава, че дори някой с личната си машина през ISP акаунт може да управлява пощенски списък, без да разчита на достъп до alias-файловете на Интернет доставчика си.

Друга важна промяна, изисквана от моите бета изпитатели, беше поддръжката на работа с 8-битово MIME. Това беше много лесно за изпълнение, тъй като аз бях внимателен да пазя кода хигиеничен към 8-те бита. Не защото предугаждах изискването на това свойство, а по-скоро заради спазването на едно друго правило:

> *15. Когато пишеш какъвто и да било шлюзов софтуер, направи си труда колкото се може по-малко да смущаваш потока от данни. И *никога* не изхвърляй информация, освен ако получателят не те принуди да го сториш.*

Ако не бях спазил това правило, подръжката на 8-битово MIME щеше да бъде мъчна и пълна с грешки. Но в случая, всичко което трябваше да сторя беше да прочета *RFC 1652* <http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt> и да добавя малко незначителна логика за генериране на заглавия.

Някои европейски потребители ме врънкаха да добавя възможност за избор на броя на съобщенията, обработвани за една сесия (така че да могат да контролират разходите за техните скъпи телефонни

мрежи). Дълго време бях непреклонен за това, и то все още не ме радва напълно. Но ако пишете за света, трябва да се вслушвате в своите клиенти – това не се променя само защото те не ви плащат пари.

# 8. Още няколко поуки от fetchmail

Преди да се върнем към общите въпроси на софтуерното инженерство, трябва да премислим няколко по-конкретни поуки от опита с fetchmail.

Синтаксисът на rc файла включва допълнителни „шумови“ ключови думи, които напълно се пренебрегват от граматичния анализатор. Англо-подобният синтаксис, който те позволяват, е значително по-четим от традиционните сбити двойки „ключ-стойност“, които получавате, ако напълно премахнете шумовите думи.

Те започнаха като среднощен експеримент, когато забелязах как декларациите от rc файла много замязват на императивен миниезик. (Точно заради това и промених оригиналната ключова дума на popclinet от „server“ на „poll“).

Струваше ми се, че този императивен миниезик може да стане по-лесен за използване, ако се доближи до английския. Сега, макар да съм убеден привърженик на школата за дизайн „направи го език“, илюстрирана от Emacs, HTML и много езици за бази от данни, обикновено не съм голям почитател на „англо-подобните“ синтаксиси.

По традиция програмистите са склонни да предпочитат контролните синтаксиси, които са много точни и компактни, и нямат никакъв излишък на информация. Това е културно наследство от времето, когато компютърните ресурси бяха скъпи, така че степените на граматичен анализ трябваше да бъдат колкото се може по-евтини и по-прости. Тогава английският език, с около 50% излишък на информация, е изглеждал като много неподходящ модел.

Но това не е моето основание обикновено да избягвам англо-подобните синтаксиси; споменах го тук само за да го срина. С евтини цикли и ядро, сбитостта не бива да бъде самоцел. За един език в днешно време е по-важно да бъде удобен за хората, отколкото да бъде евтин за компютъра.

Все още има добри основания да бъдем предпазливи. Едно от тях е цената на сложността във фазата на граматичен анализ – вие не искате да я повишите до нивото, където сама по себе си ще бъде значителен източник на грешки и объркване за потребителя. Друго основание е, че при опит да се направи англо-подобен синтаксис на език често се оказва, че „английският“, който той ще говори, е толкова объркващ, колкото би бил традиционният синтаксис. (Това може да се види при много от т.нар. „езици от четвърто поколение“ и комерсиалните езици за заявки към бази от данни.)

Контролният синтаксис на fetchmail изглежда заобикаля тези проблеми, понеже приложението на езика е силно ограничено. Той съвсем не е език с общо предназначение. Нещата които той казва, просто не са толкова сложни, така че съществува малка вероятност от объркване при мисленото движение между дребно подмножество на английския език и действителен контролен език. Мисля, че тук има една по-обща поука:

*16. Когато езикът ти съвсем не е Тюринг-издържан, синтактичният подсладител може да бъде твой приятел.*

Друга поука се отнася за сигурността чрез мъглявост. Някои потребители на fetchmail ме помолиха да променя софтуера, за да съхранява паролите в шифриран вид в rc файла, така че любопитковците да не могат случайно да ги видят.

Не го сторих, понеже това всъщност не увеличава защитата. И без туй ако някой е придобил права за да прочете вашия rc файл, тогава ще е в състояние да изпълнява fetchmail – и ако те искат да се докопат до паролата ви, ще са в състояние да измъкнат необходимия декодер от самия код на fetchmail, за да я получат.

Всичко, което ще постигне шифрирането на паролите във .fetchmailrc, е да даде фалшиво чувство за сигурност на хората, които не мислят много усилено. Общото правило тук е:

*17. Една система за сигурност е толкова сигурна, колкото сигурна е нейната тайна.*

*Пазете се от псевдо-тайни.*

# 9. Необходими предварителни условия за базарния стил

Хората, които първи прегледаха и изпитаха този документ постоянно повдигаха въпроси относно предварителните условия за успешна разработка в базарен стил, включително както по отношение на квалификациите на лидера на проекта, така и състоянието на кода, когато стане публично достъпен, и лидерът започне да се опитва да изгражда общност от съразработчици.

Съвсем ясно е, че човек не може да започне да кодира още от самото начало в базарен стил (виж. глава 15 [IN]). Така може да се тества, да се отстраняват грешки и да се подобрява кода, но би било много трудно са се *начене* проект в базарен стил. Линус не се и опита го стори. Нито пък аз. Вашата зараждаща се общност от разработчици има нужда от нещо, което може да се стартира и да се изпитва; нещо, с което да може да си играе.

Когато наченете изграждането на една общност, това, което трябва да можете да предложите, е *правдиво обещание*. Няма нужда програмата ви да работи особено добре. Може да е незряла, пълна с грешки, недовършена, да е зле документирана. Това, в което тя не трябва да се проваля е а) да тръгне, и б) да убеди потенциалните разработчици, че може да се развие до нещо наистина хубаво в едно обозримо бъдеще.

Както Linux, така и fetchmail станаха обществено достояние, когато вече имаха силен и привлекателен основен дизайн. Мнозина, мислещи за базарния модел, както го представих тук, съвсем вярно определят това за критично, а след това скачат на заключението, че е необходимо лидерът на проекта да проявява интуиция в дизайна и ум.

Но Линус се сдоби с дизайна си от Unix. Аз взех моя от предшественика му – popclient (въпреки че по-късно той се промени много, дори повече от Linux, ако говорим пропорционално). И така, трябва ли наистина лидерът/координаторът на едно начинание в базарен стил да има изключителен дизайнерски талант, или може да мине, възползвайки се от таланта на други?

Според мен не е чак толкова важно координаторът да може да измисля изключителни, блестящи дизайни; напротив – от критична важност е той да може да *разпознава добрите идеи в дизайна от други хора*.

Проектите на Linux и fetchmail са доказателство за това. Линус, въпреки че не е (както казахме по-горе) изключително оригинален дизайнер, показва невероятно умение да разпознава добрия дизайн и да го интегрира в ядрото на Linux. А и аз вече описах как единствената най-мощна идея в дизайна на fetchmail (SMTP пренасочването) дойде от друг човек.

Първите читатели на този документ ми направиха комплимент, като предполагаха, че съм склонен да подценявам оригиналността на дизайна в базарните проекти, защото самият аз го умея, и го приемам за даденост. Може и да има нещо вярно в това – дизайнът (за разлика от кодирането или отстраняването на грешки), определено е най-силната ми черта.

Проблемът да си умен и оригинален в софтуерния дизайн обаче се състои в това, че се превръща в навик – започваш да правиш нещата готини и сложни, когато трябва да са прости и смислени. Провалял съм проекти по тази причина, но все пак се предпазих от това с fetchmail.

И така, смятам, че проектът fetchmail имаше успех, отчасти защото ограничих склонността си да се правя на умен; това е довод (поне) срещу оригиналността в дизайна при успешните базарни проекти. Вземете например Linux. Да предположим, че Линус се беше опитвал да изтегли фундаментални нововъведения в дизайна на операционната система по време на разработката; дали въобще получилото се в резултат ядро щеше да бъде така стабилно и успешно като това, което имаме?

Разбира се, необходимо е едно основно ниво умения за дизайн и кодиране, но смятам, че почти всеки, който сериозно се замисля да стартира такова базарно начинание, ще да е над този минимум. Вътрешният пазар на репутацията в обществото на отворения код упражнява едва доловим натиск върху хората да не начеват разработка, която не са компетентни да следват. Засега това като че ли работи доста добре.

Има и едно друго умение, което обикновено не се свързва с разработката на софтуер, а аз мисля, че е също толкова важно за базарните проекти, колкото и разумът в дизайна, а дори и по-важно. Един координатор или лидер на базарен проект трябва да умее да общува с хората.

Това би трябвало да е очевидно. За да се създаде общност от разработчици, трябва да привличаш хора, да ги заинтригуваш с това, което правиш, и да бъдат доволни от количеството работа, която вършат. Техническото бръщолевене има още много хляб да яде, докато го постигне, но това далеч не е всичко. Има значение и личността, която проявяваш.

Неслучайно Линус е готин пич, който кара хората да го харесват и да искат да му помогнат. Неслучайно аз съм енергичен екстроверт, който обича да работи с много хора, и притежава някои от характерните черти и инстинктите на неизявен комик. За да сработи базарният модел, от огромна помощ би било ако можете поне мъничко да очаровате хората.

# 10. Социалният контекст на софтуера с отворен код

Истина е, че най-добрите програми започват като лични разрешения на всекидневните проблеми на автора, а се разпространяват, понеже се оказва, че проблемът е типичен за широк кръг потребители. Това ни връща към смисъла на правило 1, може би по-добре преформулирано:

> *18. Ако искаш да разрешиш интересен проблем, първо започни да търсиш проблем, който е интересен за самия теб.*

Тъй беше с Карл Харис и наследения popclient, тъй беше с мен и fetchmail. Но това е разбрано много отдавна. Интересният въпрос, въпросът, върху който историите на Linux и fetchmail ни принуждават да се концентрираме, е следващият етап – развитието на софтуера при наличието на голяма и активна общност от потребители и съразработчици.

В „Митичният човеко-месец“ Фред Брукс е забелязал, че програмисткото време не нараства лесно и гъвкаво – добавянето на програмисти в напреднал софтуерен проект води до забавянето му. Той твърди, че цената на сложността и на комуникацията в един проект се покачва квадратично с броя на разработчиците, докато количеството свършена работа се покачва едва линейно. От тогава насетне това твърдение става известно като „Закон на Брукс“ и се възприема като очевидна истина. Но ако всичко се изчерпва със Закона на Брукс, Linux би бил невъзможен.

В класическата си книга „Психология на компютърното програмиране“, Джералд Уейнбърг показва нещо, което (със закъснение) можем да привидим като живо опровержение на Брукс. При своето разглеждане на „програмирането без его“, Уейнбърг е наблюдавал, че в предприятията, където програмистите не са ревниви спрямо кода си и окуражават другите хора да търсят вътре грешки и възможности за подобрение, подобрението става драматично по-бързо, отколкото където и да било другаде.

Анализът на Уейнбърг не бива приет по подобаващ начин, може би заради избраната от него терминология – някой би се усмихнал на идеята хакерите в Интернет да се характеризират като „без его“. Но аз мисля, че днес неговият аргумент изглежда по-силен от всякога.

Историята на Unix би трябвало да ни е подготвила за това, което научихме от Linux (и което аз експериментално проверих в по-малък мащаб, съзнателно копирайки методите на Линус (виж. глава 15 [EGCS])). А то е, че докато кодирането остава дейност, която се извършва главно в уединение, наистина великите разработки идват от впрягането на умствените способности и вниманието на цели общности. Разработчикът, който използва само своя собствен акъл в един затворен проект, ще изостане зад разработчика, който умее да създаде отворен, еволюционен контекст, в който чрез обратна връзка се изследва пространството на дизайна, дарява се код, посочват се грешки, и други подобрения идват от стотици (може би дори хиляди) хора.

Но няколко фактора попречиха на традиционния Unix свят да развие този подход докрай. Един от тях беше наличието на юридически ограничения на разните лицензи, търговски тайни и комерсиални интереси. Друг (както се оказа в последствие) се състоеше в това, че Интернет все още не беше достатъчно развит.

Във времето преди евтиния Интернет съществуваха няколко географски компактни общности, където културата окуражаваше програмирането „без его“, в смисъла на Уейнбърг. Там един разработчик можеше лесно да привлече голям брой кадърни кибици и съразработчици. Лабораториите „Бел“, Лабораторията по изкуствен интелект към Масачузетския Технологичен Институт, Калифорнийският Университет в Бъркли – всички те се превърнаха в дом на иновациите, който днес е легендарен и все още плодотворен.

Linux беше първият проект за съзнателно и успешно използване на целия *свят* като запас от таланти. Не мисля, че е случайност това, че периодът на бременност на Linux съвпадна с раждането на World Wide Web. И че Linux излезе от детството си през същия период на 1993-1994, който бе свидетел на летящия старт на ISP индустрията и бума на всеобщия интерес към Интернет. Линус беше първият човек, който се научи как да играе по новите правила, които донесе широко разпространеният Интернет.

Докато евтиният Интернет беше необходимо условие за създаването на модела на Linux, той не беше сам по себе си достатъчно условие. Друг жизненоважен фактор беше развитието на стил на водачество и множество обичаи за сътрудничество, които могат да позволят на разработчиците да привличат съразработчици и да извличат максималното от възможностите на средата.

Но какъв е този стил на водачество и какво представляват тези обичаи? Те не могат да бъдат основани на властови отношения – а дори и да можеха, водачеството чрез принуда нямаше да доведе до резултатите, които наблюдаваме. Уейнбърг цитира автобиографията на руския анархист от 19-ти век Пьотр Алексеевич Кропоткин „Спомените на един революционер“, която е в сила за нашия въпрос:

> „Като отгледан в семейство на владетел на крепостни селяни, аз влязох в действителния живот подобно на всички млади хора от моето време, твърде убеден в необходимостта от командване, заповядване, хокане, наказване и прочие. Но когато в началото трябваше да управлявам сериозни предприятия и да си имам работа със [свободни] хора, когато всяка грешка можеше изведнъж да доведе до тежки последици, тогава започнах да разбирам разликата между действането на принципа на командването и дисциплината и действането на принципа на общото съгласие. Първото работи чудесно при един военен парад, но не струва нищо там, където е замесен истинският живот, където целта може да бъде постигната само чрез усилието на голям брой събиращи се воли.“

„Усилието на голям брой събиращи се воли“ е точно това, което изисква проект като Linux. И „принципът на командването“ действително е неприложим между доброволците в този анархистки рай, който наричаме Интернет. Хакерите, желаещи да ръководят сътруднически проекти, ако искат ефикасно да работят и да се съревновават, трябва да се научат да вербуват и стимулират заинтересовани общности в стила, неясно формулиран в „принципа на съгласието“ на Кропоткин. Трябва да се научат да използват Закона на Линус. (виж. глава 15 [SP])

По-рано аз се позовах на „Делфийския ефект“ като възможно обяснение на Закона на Линус. Но неизбежно ми идват наум и по-силни аналогии с адаптивните системи в биологията и икономиката. В много отношения, поведението на Linux света е подобно на свободен пазар или екологична система, където сбор от себелюбиви агенти се опитват да увеличат ползата си до най-голямата възможна степен, в хода на което се получава самокоригиращ се спонтанен ред с по-голяма сложност и ефикасност, отколкото би могло да бъде постигнато от каквото и да било централно планиране. Тъкмо тук трябва да се търси „принципа на съгласието“.

„Полезната функция“, увеличена до най-голямата възможна степен от Linux хакерите, не е класически икономическа, а е невещественото удовлетворение на собственото им его и репутация сред останалите хакери. (Някой може да нарече мотивацията им „алтруистична“, но така не се обръща внимание на факта, че алтруизмът сам по себе си е форма на задоволяване на егото на алтруиста). В действителност, доброволческите култури, които действат по този начин, не са необикновени. Друга такава, в която дълго време съм участвал, е общността на феновете на научната фантастика, където, за разлика от Хакерландия, „гъделът“ (увеличаването на нечия репутация всред другите фенове) се признава като основна движеща сила на доброволческата дейност.

Линус показа едно проницателно схващане на „принципа на споделено съгласие“ на Кропоткин, успешно поставяйки се като портиер на проект, в който разработката се извършва главно от други хора, под-

хранвайки интереса към проекта, докато той придобие своя самостоятелност. Този квази-икономически поглед върху Linux света ни позволява да разберем как се прилага това съгласие.

Можем да разглеждаме метода на Линус като начин за създаване на ефикасен пазар в „гъдела“ – себичността на отделните хакери да се свърже колкото се може по-здраво в реализацията на трудни задачи, които могат да бъдат осъществени само при постоянно сътрудничество. С проекта fetchmail аз показах (макар и в по-малък мащаб), че неговите методи могат да бъдат повтаряни, при което се получават добри резултати. Аз дори може би го сторих малко по-съзнателно и систематично от него.

Много хора (особено онези, които политически се съмняват в свободните пазари) биха очаквали, че една култура от самоорганизиращи се егоисти ще е разпокъсана, разделена, прахоснически, потайна и враждебна. Но това очакване е напълно опровергано (да дадем само един пример) от смайващото разнообразие, качество и задълбоченост на документацията за Linux. Свещена истина е, че програмистите *мразят* писането на документация. Но как тъй Linux хакерите създават толкова много такава? Очевидно, свободният пазар на гъдел в Linux работи по-добре, за да доведе до добродетелно, насочено в друга посока поведение, отколкото при здраво финансираните отдели за документация на комерсиалните производители на софтуер.

И двата проекта – fetchmail и ядрото Linux – показват, че чрез прилично възнаграждаване на егото на много хакери, един силен разработчик/координатор може да използва Интернет, за да улови ползите от притежаването на много съразработчици, без проектът да се сгромоляса в пълен миш-маш. И тъй, срещу Закона на Брукс предлагам следното:

> *19. Когато на координатора на разработката е предоставена среда, поне толкова добра колкото Интернет, и той знае как да ръководи без принуда, тогава повече глави са неизбежно по-добри от една.*

Мисля, че бъдещето на софтуера с отворен код все повече ще принадлежи на хора, които знаят как да играят играта на Линус, хора, които изоставят катедралата и се присъединяват към базара. Това не означава, че индивидуалната проницателност и надареност вече няма да са от значение. Напротив, мисля, че острието на софтуера с отворен код ще принадлежи на хора, които започват от индивидуална проницателност и надареност, а после я усилват чрез сполучливото изграждане на заинтересувани доброволчески общности.

И може би не само бъдещето на софтуера с *отворен код*. Няма разработчик на затворен код, който да се сравни със запаса от таланти, който общността на Linux може да пусне в действие върху дадена задача. Колцина изобщо биха си позволили да наемат повече от двеста (1999 – шестотин, 2000 – осемстотин) души, които са допринесли с нещо към fetchmail.

В крайна сметка, може би културата на отворения код ще триумфира не защото сътрудничеството е морално правилно или защото „прикриването“ на софтуера е морално неправилно (да приемем, че вярвате в последното – нещо, с което нито Линус, нито аз сме съгласни), а просто защото светът на затворения код не може да спечели революционната надпревара във въоръжаването с общностите на отворения код, които могат да съсредоточат значително повече квалифицирано време върху даден проблем.

## 11. За управлението и линията Мажино

Първоначалният вариант на „Катедралата и базарът“ завършваше с мечтата, описана по-горе – за щастливите орди програмисти/анархисти, съревноваващи се и надвиващи йерархичния свят на традиционния затворен софтуер.

Мнозина скептици обаче не бяха убедени; и въпросите, които те повдигнаха, заслужават съответния ангажимент. Повечето от възраженията срещу базарната теза се свеждат до твърдението, че неговите защитници подценяват ефекта от производителността и мащабируемостта на конвенционалното управление.

Традиционно-настроените координатори на софтуерни проекти често възразяват, че случайността, с която се формират, променят и разпускат групите на проектите в света на отворения код, значително

обезсилва очевидното преимущество, което общността на отворения код има като брой хора в сравнение с който и да е разработчик на затворен код. Според техни наблюдения в разработката на софтуер единствените неща, които имат значение, са постоянната работа, и степента, до която клиентите могат да очакват да продължат вложенията си, а не колко души са хвърлили по едно кокалче в гърнето, и са го оставили да къкри.

В този аргумент безспорно има нещо вярно; всъщност във *„Вълшебният котел“* <http://www.tuxedo.org/esr/writings/magic-cauldron/> аз развих идеята, че ключът към икономиката на софтуерното производство се корени в очакваната стойност на бъдещите услуги.

Тук обаче има и един голям скрит проблем – приема се по подразбиране, че общността на отворения код не може да осигури такава продължителна работа. Всъщност има проекти с отворен код, които доста дълго продължават да следват съгласувано целите си, и да имат ефективни общности от разработчици без стимулиращите структури или институционални авторитети, които конвенционалното управление счита за основни. Най-добър и назидателен пример за това е разработката на GNU Emacs; той е концентрирал усилията на стотици сътрудници за повече от 15 години в единно архитектурно виждане, въпреки голямото текучество и фактът, че само един човек (авторът му) продължава да е активен през цялото това време. Никой (текстов) редактор не е достигал този рекорд по дълголетие.

Това ни дава причина да се запитаме за предимствата на конвенционално управляваната разработка, които са независими от останалите аргументи в спора на катедралния модел срещу базарния. Ако за GNU Emacs е възможно да изразява съдържателно архитектурно виждане в един период от 15 години, или за една операционна система като Linux да направи същото за осем години, в които технологиите на хардуера и платформите се променят изключително бързо; и ако (какъвто наистина е случаят) има множество добре обмислени проекти с отворен код, които траят повече от 5 години, то ние можем да се почудим какво, ако въобще има такова, ни носи невероятното предимство на конвенционално управляваната разработка.

Каквото и да ни носят, то определено не е нито сигурно пускане в срок, нито побиране в рамките на бюджета, или съблюдаване на всички точки от спецификацията; рядко някой „управляван“ проект изпълнява дори едно от тези условия, а да не говорим за всичките три накуп. А очевидно не е и способността за приспособяване към промените в технологичния и икономически контекст по време на „битието“ на проекта; общността на отворения код се оказва *много* по-ефективна в това отношение (както всеки може бързо да провери, например като сравни 30-годишната история на Интернет с късичкия живот на фирмените мрежови технологии; или цената на мигрирането от 16-те към 32-та бита в Microsoft Windows с извършената почти без никакво усилие миграция при Linux през същия период, не само при Intel линията, но и при над дузина други хардуерни платформи, включително 64-битовата Alpha).

Едно от нещата, които мнозина смятат, че традиционният модел ни носи е някой да бъде отговорен пред закона и по възможност да възстанови щетите, ако проектът се обърка. Това обаче е илюзия – повечето софтуерни лицензи са написани така, че отхвърлят дори гаранцията за надеждност, а да не говорим за производителността; случаите на успешно възстановяване на щети поради непроизводителност на софтуера стават все по-редки. Дори и ако бяха по-често срещани, не би имало смисъл да се чувстваме по-добре, ако имахме кого да съдим. Хората искат не да се съдят, а да имат работещ софтуер.

И тъй, какво ни носи това предимство в управлението?

За разберем това, трябва да проумеем какво смятат, че правят софтуерните мениджъри. Една моя позната, която изглежда е много добра в тази работа, казва, че управлението на софтуерни проекти има пет функции:

1. Да *определя целите* и да направлява всички в една и съща посока.
2. Да *наблюдава* и да внимава да не бъдат пропускани критични подробности.
3. Да *мотивира* хората да вършат досадна, но необходима черна работа.
4. Да *организира* разпределението на хората с цел най-добра производителност.
5. Да *организира ресурсите*, необходими за поддържането на проекта.

Очевидно всички тези цели са важни, но при модела на софтуера с отворен код и в неговия окръ-

жаващ социален контекст, те могат да ни се сторят странно неуместни. Ще ги разгледаме в обратен ред.

Един мой приятел ми каза, че по-голямата част от *организирането на ресурсите* се състои в защитата – след като имаш хора, машини и офис пространство, трябва да ги пазиш от съседните мениджъри, които се състезават за същите ресурси, и от по-горните в йерархията, които се опитват по най-ефективен начин да използват ограничените средства.

Разработчиците на софтуер с отворен код обаче са доброволци, и се самоподбират както по интерес, така и заради възможността да допринесат за проектите, по които работят (това остава в сила дори и когато им се плаща, за да хакват отворен код). Доброволческият дух има тенденцията автоматично да се грижи за „атакуващата“ страна на организирането на ресурсите – хората слагат на масата своите собствени ресурси. Така съществува малка или никаква необходимост един мениджър да „играе в защита“ в конвенционалния смисъл.

Както и да е, в един свят на евтини персонални компютри и бърз Интернет, почти постоянно виждаме, че единственият наистина ограничаващ ресурс е квалифицираното внимание. Когато проектите с отворен код се провалят, това не става поради желанието за машини или офис-пространство – те умират само когато разработчиците загубят интерес.

След като случаят е такъв, несъмнено е важно хакерите на отворения код да *се организират* за максимум производителност чрез самоподбор – а социалната среда безжалостно подбира по компетентност. Моята позната, запозната както със света на отворения код, така и с големи затворени проекти, смята, че успехът на отворения код отчасти се дължи на това, че неговата култура приема само петте или колкото са там процента най- талантливи от програмистката популация. А моята позната прекарва повечето от времето си в организирането на останалите 95%, така че от първо лице е наблюдавала една известна вариация при фактор 100 в производителността между най-способните програмисти и почти компетентните.

Размерът на тази вариация винаги повдига един неудобен въпрос: дали индивидуалните проекти, а и областта като цяло, биха били по-добре без 50% от по-неспособните? Умните мениджъри отдавна са разбрали, че ако единствената функция на конвенционалното управление беше да превърне най-малко способните от нетна загуба в чиста победа, то значи играта просто не би си струвала.

Успехът на общността на отворения код значително изостря този въпрос, като предоставя доказателства, че често е по-евтино и по-ефективно да се подберат доброволци от Интернет, отколкото да се управляват цели сгради, пълни с хора, които биха искали да се занимават с нещо друго.

Това ни води към въпроса за *мотивацията*. Равностоен и често срещан начин за излагане гледната точка на моята позната е, че традиционното управление на разработката е необходима компенсация за зле мотивираните програмисти, които иначе не биха свършили добра работа.

Този отговор обикновено се движи с твърдението, че можем да се осланяме на общността на отворения код само ако работата, която трябва да се върши е „секси“ или технически готина – всичко друго остава несвършено (или зле направено), освен ако не е механично изработено от парично мотивирани, затворени в кутийки ратаи с управители, които размахват камшици наоколо им. В моята студия „Да обитаваш ноосферата“ говоря за психологическите и социални причини за моя скептицизъм по отношение на това твърдение. Засега обаче смятам, че е по-интересно да посоча изводите от приемането му за вярно.

Ако конвенционалният, със затворен код, тежък за управление стил на софтуерна разработка наистина се защитава само чрез нещо като линия Мажино от проблеми, благоприятстващи скуката, то това ще остане в сила за всяка отделна приложна област единствено до този момент, в който никой не смята тези проблеми за интересни и никой не намира начин да ги заобиколи. Понеже сега има надпревара в общността на отворения код за „отегчителни“ програми, клиентите ще знаят, че най-после някой се е захванал да разреши този проблем заради очарованието на самия проблем – което в програмирането, както и при други видове творческа дейност, е много по-мотивиращо, отколкото парите.

Следователно да имаш конвенционална структура на управление само за мотивация вероятно е добра тактика, но лоша стратегия; краткосрочна победа, но в перспектива почти сигурна загуба.

До тук конвенционалното управление на разработката изглежда лош избор в сравнение с отворения код поради две причини (подреждането на ресурсите, организацията), и като че ли е на доизживяване поради трета (мотивацията). А горкият обсаден конвенционален мениджър няма да получи никакви подкрепления от *наблюдаването*; най-силният аргумент в полза на отворения код се състои в това, че децентрализираният поглед върху нещата се оказва по-добър от всички конвенционални методи за старание да се осигури да не се изпуснат някакви детайли.

Можем ли да сметнем *определянето на целите* като оправдание за недостатъците на конвенционалното управление на софтуерни проекти? Може би; но за да направим това, ще ни трябва много важна причина да повярваме, че управителните комитети и корпоративните планове имат по- голям успех при определянето на подходящи и широко споделяни цели, отколкото водачите на проекти и племенните старейшини, които изпълняват аналогична роля в света на отворения код.

Изправени пред него, това е доста труден случай за разрешаване. Но това не е толкова по някаква причина от страна на отворения код (дълголетието на Emacs, или способността на Линус Торвалдс да мобилизира орди от разработчици с разговори за „световно господство“). Това по-скоро се дължи на демонстрираната невъзможност на конвенционалните механизми да дефинират целите на софтуерните проекти.

Една от най-известните фолколрни теореми в софтуерното инженерство гласи, че 60% до 75% от конвенционалните софтуерни проекти или никога не се завършват, или се отхвърлят от потребителите си. Ако тези проценти са близо до верните (а аз никога не съм срещал мениджър с какъвто и да било опит, който да ги оспори), то повечето проекти имат цели, които са или а) реалистично недостижими, или б) съвсем грешни.

Това, повече от всеки друг проблем, е причина в днешния свят на разработка на софтуер самият израз „управителен съвет“ да кара чулите го да потръпват – дори (а може най-вече) ако чулият го е мениджър. Дните, в които единствено програмистите говореха така, отдавна са отминали – днес над бюрата на *изпълнителните директори* са окачени комиксите „Дилбърт“ [1].

Следователно нашият отговор на мениджърът на традиционна софтуерна разработка е прост – ако общността на отворения код наистина подценяваше важността на конвенционалното управление, *защо толкова много от вас демонстрират презрение към собствените си дела?*

Още веднъж успехът на общността на отворения код значително изостря този въпрос – защото се *забавляваме* с това, което правим. Нашата творческа игра носи технически, пазарни и интелектуални успехи с невероятно темпо. Ние доказваме не само, че можем да правим по-добър софтуер, но и че *удоволствието е предимство*.

Две години и половина след първата версия на това есе, най-радикалните думи, с които мога да завърша, вече не са мечтата за доминиран от софтуера с отворен код свят; това, все пак, в наши дни изглежда правдоподобно на мнозина трезви хора с костюми.

По-скоро бих искал да предложа една по-обща поука относно софтуера (а вероятно и за всеки друг вид творческа и професионална работа). Човешките същества общо взето изпитват удоволствие от едно задание, когато то попада в областта на оптималното предизвикателство – не прекалено лесно, за да е отегчаващо, но и не прекалено трудно за постигане. Щастлив програмист е този, който хем не стои без работа, хем не е претоварен от зле формулираните задания и стресиращото търкане на процесите. *Удоволствието следва ефикасността.*

Следователно ако се отнасяте към вашия собствен работен процес със страх и неохота (дори и по неуместния, ироничен начин с картинките на Дилбърт), то това трябва да се приеме само по себе си за знак, че процесът се е провалил. Удоволствието, хуморът и веселбата наистина са предимства; съвсем не за благозвучие използвах израза „щастливи орди“, нито пък е обикновена шегичка, че талисман на Linux е мекичкият, сладък пингвин.

Може да се окаже, че едно от най-важните последствия от успеха на отворения код ще е да ни научи, че играта е икономически най-ефективния начин за творческа работа.

[1] *[T-DILBERT]* Повече информация за комиксите с Дилбърт има *този адрес* `<http://www.dilbert.com>`. *(бел. прев.)*

## 12. Благодарности

Тази студия беше подобрена чрез разговори с голям брой хора, които помогнаха за отстраняването на най-различни грешки. Особени благодарности дължа на Джеф Дътки <dutki@wam.umd.edu>, който предложи формулировката „отстраняването на грешки е паралелизируемо“ и помогна при развиването на последвалия анализ. Също на Нанси Лебовиц <nancyl@universe.digex.net>, за подсказването ѝ, че цитирайки Кропоткин, аз подражавам на Уейнбърг. Приемлива критика дойде също от Джоан Еслингър <wombat@kilimanjaro.engr.sgi.com> и Марти Франц <marty@net-link.net> от списъка General Technics. Глен Ванденбърг <glv@vanderburg.org> отбеляза важността на естествения подбор в популациите от сътрудници и предложи плодотворната идея, че по-голямата част от разработката поправя „грешки поради недостиг“. Даниъл Ъпър <upper@peak.org> предложи естествени аналогии за това. Признателен съм на членовете на PLUG (Philadelphia Linux User's Group – Група на потребителите на Linux във Филаделфия), че станаха първите пробни читатели на първата публична версия на тази студия. Пола Матусчек <matusp00@mh.us.sbphrd.com> ме светна за практиката на софтуерен мениджмънт. Фил Хъдсън <phil.hudson@iname.com> ми припомни, че социалната организация на хакерската култура се отразява в организацията на нейния софтуер, и обратно. И накрая, коментарите на Линус Торвалдс бяха полезни и своевременната му подкрепа беше окуражаваща.

## 13. Допълнителна литература

Цитирах малко от класическата книга на Фредерик П. Брукс *„Митичният човеко-месец“* (Frederick P. Brooks *„The Mythical Man-Month“*) защото, в много отношения, от неговите прозрения все още могат да се извлекат поуки. Горещо ви препоръчвам 25-тото Юбилейно издание от Addison-Wesley (ISBN 0-201-83595-9), в което е добавена студията *„No Silver Bullet“*, написана през 1986 година.

Новото издание е обгърнато от една безценна 20-годишна ретроспекция, в която Брукс откровено признава за няколко възгледа в оригиналния текст, които не издържат проверката на времето. За първи път прочетох ретроспекцията след като първата публична версия на настоящата студия беше вече в основни линии завършена. С изненада открих, че Брукс приписва на Microsoft практики, подобни на базарните! (Все пак, това приписване всъщност се оказа погрешно. През 1998 научихме от *Halloween Documents* `<http://www.opensource.org/halloween/>`, че вътрешната общност от разработчици на Microsoft е силно разделена на враждуващи помежду си лагери, където дори не е съвсем възможен някакъв вид общ достъп до изходния код, така необходим за да се поддържа един базар.)

*„Психология на компютърното програмиране“* на Джералд М. Уейнбърг (Gerald M. Weinberg, *„The Psychology Of Computer Programming“*, New York, Van Nostrand Reinhold, 1971) ни запознава с една концепция, която по-скоро неуместно наречена „програмиране без его“. Докато той далеч не е бил първият човек, който е усетил безсилието на „принципа на командването“, той вероятно е бил първият, който е оценил и показал, че този въпрос има определена връзка с разработването на софтуер.

В своята студия *„Lisp: добри новини, лоши новини, и как да печелим първото място“* от 1989 година, Ричард П. Гейбриъл (Richard P. Gabriel, *„Lisp: Good News, Bad News, and How To Win Big“*), размишлявайки върху Unix културата от епохата преди Linux, неохотно е привел доводи в полза на първичния базароподобен модел. Въпреки че е остаряло в доста отношения, това есе все още, и с право, е широко известно сред феновете на Lisp (включително и мен). Един кореспондент ми припомни, че главата, наречена „Worse Is Better“ може да се тълкува едва ли не като очакване на Linux. Студията е достъпна в World Wide Web на адрес `<http://www.naggum.no/worse-is-better.html>`.

*„Работа с хора: творчески проекти и екипи“* на Де Марко и Листър (De Marco and Lister, *„Peopleware: Productive Projects and Teams“*, New York; Dorset House, 1987; ISBN 0-932633-05-6) е един недооценен бисер, за съществуването на който ме светна един цитат в ретроспекцията на Фред Брукс. Докато много малко от авторовите думи са пряко приложими към общността на Linux или общностите с отворен код като цяло, авторовите прозрения за необходимите условия за творческа работа са важни и ценни за всеки, който се опитва да внедри някои от силните страни на базарния модел в

комерсиална среда.

Най-накрая трябва да призная, че за малко щях да нарека тази студия „Катедралата и агората“. Последната дума е гръцка и означава отворен пазар или място за публични срещи. Плодотворните студии за „агорическите системи“ на Марк Милър и Ерик Дрекслър описват неочакваните характеристики на подобните на пазар изчислителни екологични системи. Това ми помогна да се подготвя да мисля ясно за сходните феномени в културата на отворения код, когато преди пет години Linux ми натри носа в тях. Тези студии са достъпни в Web на адрес: <http://www.agorics.com/agorpapers.html>.

# 14. Епилог: Netscape се присъединява към базара

Колко е странно да осъзнаеш, че участваш в изковаването на историята...

На 22 януари 1998, около седем месеца след като за първи път публикувах „Катедралата и базарът“, Netscape Communications, Inc. обяви плановете си да *подари изходни код на Netscape Communicator* <http://www.netscape.com/newsref/pr/newsrelease558.html>. Нямах и представа, че това ще стане, докато не излезе обявлението.

Ерик Хан, изпълнителен вицепрезидент и старши технолог в Netscape, малко по-късно ми изпрати следната електронна поща: „От името на всички в Netscape, искам на първо място да Ви благодаря, задето ни доведохте дотук. Решението ни беше съществено повлияно от Вашите размишления и съчинения.“

През следващата седмица отлетях за Силиконовата Долина, откликвайки на поканта на Netscape за еднодневна стратегическа конференция (на 4 февруари 1998) с някои от висшите им администратори и технолози. Заедно проектирахме стратегията за пускане на изходния код и лиценза на Netscape.

Няколко дни по-късно, написах следното:

„Netscape се кани да ни даде едно едромащабно, сериозно изпитание на базарния модел в комерсиалния свят. Културата на отворения код е изправена пред опастност. Ако изпълнението на Netscape не сполучи, идеята за отворения код може да се компрометира дотолкова, че комерсиалният свят да я избягва още едно десетилетие.

От друга страна, това е и една грандиозна възможност. На „Уол стрийт“ и други места, първоначалната реакция на хода беше сдържано положителна. Даден ни е шанс да се докажем. Ако чрез този ход Netscape си възвърне значителен пазарен дял, това може просто да възпламени една отдавна закъсняла революция в софтуерната индустрия.

Следващата година ще да бъде едно много поучително и интересно време.“

И наистина беше. Както писах в средата на 1999 година, разработката на това, което по-късно беше наречено „Mozilla“, беше само частичен успех. То постигна първоначалната цел на Netscape, която беше да не се позволи на Microsoft да стегне в монопол пазара на web-четци. То постигна и отделен главозамайващ успех (особено пускането на Gecko – рендираща машина от следващо поколение).

Но тъй или иначе, извън Netscape то все още не е концентрирало такива масови усилия за разработка, за каквито първоначално са се надявали основателите на Mozilla. Изглежда, проблемът тук е, че за дълъг период от време дистрибуцията на Mozilla в действителност нарушаваше едно от основните правила на базарния модел. Те не предоставиха нещо, което потенциалните сътрудници да могат лесно да стартират и да видят работещо. (Повече от година след пускането, компилирането на Mozilla от изходния код изискваше лиценз за патентованата библиотека Motif.)

Най-лошото бе (от гледната точка на външния свят), че групата Mozilla все още трябваше да достави web-четец с производствено качество. А един от ръководителите на проекта предизвика малка сензация като напусна, оплаквайки се от слабото управление и пропуснатите възможности. „Отвореният код“, правилно отбеляза той, „не е вълшебна пръчица.“

И наистина не е. Дългосрочните прогнози за Mozilla сега изглеждат доста по-добре (през август 1999), околкото бяха по време на оставката на Джейми Завински. Но той имаше право, като показа, че отвярянето няма задължително да спаси един съществуващ проект, който страда от лошо дефинирани цели, подобен на спагети код или която и да било друга хронична болест на софтуерното инженерство.

Mozilla успя да ни даде пример едновременно как отворения код може да успее и как би могъл да се провали.

Междувременно, идеята на отворения код набра точки и намери поддръжници другаде. 1998 и краят на 1999 станаха свидетели на страхотен бум на интереса към модела на разработка с отворен код. Една тенденция, хем задвижвана от, хем движеща растящия успех на операционната система Linux. Тенденцията, до която се докосна Mozilla, расте с ускоряващо се темпо.

# 15. Бележки

*[JB]* В „Програмистки бисери“, бележитият афорист на компютърните науки Джон Бентли коментира наблюдението на Брукс така: „Ако планираш да пропилееш едно, ще пропилееш две“. Той е почти напълно прав. Със своето наблюдение, Брукс, както впрочем и Бентли, не иска просто да каже, че трябва да очакваш, че първият ти опит ще бъде погрешен, а че обикновено е по-сполучливо да започнеш отначало с правилната идея, отколкото да се опитваш да оправиш един бълвоч.

*[QR]* Съществуват примери за успешна разработка с отворен код в базарен стил, предхождащи Интернет експлозията и несвързани с традиците на Unix и Интернет. Един такъв през 1990-1992 е разработката на инструмента за компресиране (предимно върху DOS машини) *info-Zip* `<http://www.cdrom.com/pub/infozip/>`. Друг е системата за обмен на съобщения RBBS (отново за DOS), която започна през 1983 и разви достатъчно силна общност, за да има сравнително редовни издания до момента (средата на 1999), въпреки огромните технически предимства на пощата и обмена на файлове по Интернет спрямо местните BBS-и. Докато info-Zip общността се осланяше до известна степен на Интернет пощата, разработчиците на RBBS успяха да установят една он-лайн общност върху RBBS, която е напълно независима от инфраструктурата на TCP/IP.

*[JH]* Джон Хеслър предложи интересно обяснение на факта, че дублирането на усилията не изглежда да е нетна загуба при разработката на отворен код. Той предложи нещо, което ще нарека „Закон на Хеслър“: цената на дублираната работа има тенденцията да нараства по-малко от квадратично спрямо големината на екипа – т.е. по-бавно от допълнителното планиране и управление, необходимо за отстраняването им.

Това твърдение всъщност не противоречи на закона на Брукс. Може би общата допълнителна сложност и уязвимостта от грешки расте на квадрат от големината на екипа, но въпреки това цената на *дублираната* работа е особен случай и нараства по-бавно. Не е трудно да се намерят правдоподобни причини за това, като се започне с несъмнения факт, че е много по-лесно да се постигне съгласие по функционалните граници между кода на отделните разработчици, което ще предотврати дублиране на работата, отколкото да се предотвратят различните видове лошо непланирано взаимодействие в цялата система, които са в основата на повечето грешки.

Комбинацията от законите на Линус и Хеслър подсказва, че всъщност софтуерните проекти биват три вида според размера си. При малките проекти (бих казал такива с не повече от трима разработчици) не е необходима по-сложна структура на управление от тази да се избере водещ програмист. Над тях съществува някакъв среден диапазон, при който разходите на традиционното управление са относително ниски, така че предимствата му от избягването на дублирането на работа, проследяването на грешки и опитите да не се пропускат детайли всъщност се оказват положителни.

Над това обаче комбинацията от законите на Линус и Хеслър подсказва, че има едно ниво на големите проекти, в което разходите и проблемите на традиционното управление се увеличават много по-бързо, отколкото очакваните разходи от дублирането на работа. Не на последно място сред тези разходи е структурната неспособност да се използва „ефекта на многото очи“, който (както вече видяхме) изглежда върши много по-добра работа от традиционното управление при подсигуряването да не се пропускат грешки и подробности. Така при големите проекти комбинацията от тези закони всъщност прави нетната полза от традиционното управление равна на нула.

*[IN]* Един проблем, свързан с това дали могат да се наченат проекти от нулата в базарен стил, се състои в това дали базарния стил може да поддържа наистина творческа работа. Ня-

кои твърдят, че тъй като липсва силно водачество, базарът може да се справи само с клонирането и усъвършенстването на идеи вече съществуващи в последната дума на инженерството, но не може да разработва нови. Това твърдение е може би най-известно от позорните *документи Halloween* <http://www.opensource.org/halloween>, два обезпокоителни вътрешни меморандума на Microsoft, в които се говори за феномена на отворения код. В него авторите наричат Линусовата разработка на Unix-подобна операционна система „да гониш влака по перона“, и изказват мнение, че „(веднъж щом проектът е достигнал последната дума на технологията) необходимото ниво на управление за преодоляване на нови бариери става огромно“.

В този аргумент се съдържат сериозни фактологични грешки. Едната си проличава по-късно, когато авторите на Halloween сами забелязват, че „често [...] нови изследователски идеи се прилагат и са достъпни първо в Linux, преди да се включат в други платформи“.

Ако четем „отворен код“ вместо „Linux“, ще видим, че този феномен съвсем не е нов. В исторически план общността на отворения код не е открила Emacs или World Wide Web или Интернет сама като гони влаковете по перона или като бъде масивно управлявана – и понастоящем няма много откривателска работа, протичаща в общността на отворения код, която да подлежи на избор. Проектът GNOME (ако изберем един от многото) достатъчно усилено се движи по ръба на най-новите разработки в областта на графичните интерфейси и обектната технология, за да привлече вниманието на компютърната търговска преса, която е доста далеч от Linux обществото. Има хиляди други примери, както веднага би показало едно посещение във *Freshmeat* <http://freshmeat.net> в който и да е ден.

Но има и една още по-фундаментална грешка в имплицитното допускане, че *катедралният модел* (или базарният, или който и да е друг вид управленческа структура) може благонадеждно да прави нововъведения. Това е безсмислица. Бандите не получават внезапни прозрения – дори на доброволческите групи от базарни анархисти обикновено им липсва истинска оригиналност, а да не говорим за корпоративните комитети от хора със залог за оцеляване в някакво положение quo ante. *Прозрението идва от личностите.* Най-многото, което може да се надява някога да направи тяхната окръжаваща социална машинария, е да бъде *отзивчива* на внезапни прозрения – да ги поддържа, възнаграждава и сурово да ги изпитва, вместо да ги мачка.

Някои ще определят това като романтично схващане, като връщане към остарелите стереотипи за самотния изобретател. Не е така – аз не твърдя, че групите не могат *да доразвиват* внезапни прозрения, щом веднъж са се излюпили; и наистина, ние се учим от гледания отстрани процес, че такива групи разработчици са жизнено важни за получаването на висококачествени резултати. По-скоро посочвам, че разработката на всяка една такава група започва – дори задължително се начева – от една добра идея в нечия глава. Катедралите, базарите и другите социални структури могат да впрегнат тази мълния и да я усъвършенстват, но това не става по поръчка.

Следователно наистина основният проблем на иновацията (в софтуера, както и навсякъде другаде) е как да не я мачкаш; но дори още повече *как да развиваш много хора, които на първо място да имат прозрения*.

Би било абсурдно да предположим, че катедралният стил на разработка владее този трик, а ниските входни точки и плавността на процесите на базарния – не. Ако беше необходим само един човек с добра идея, то тогава социалната среда, в която един човек може бързо да привлече за сътрудници стотици или хиляди други с тази си добра идея, ще трябва неминуемо да „от-изобретява“ всяка от тях, в която човекът ще трябва да се занимава с политически продажби в йерархия, преди да може да работи по идеята си без да има опасност да го уволнят.

И наистина, ако погледнем историята на софтуерните нововъведения в организации, използващи катедралния модел, бързо ще открием, че това става твърде рядко. За нови идеи големите корпорации разчитат на университетски изследвания (оттук и опасенията на авторите на документите Halloween от способността на Linux по-бързо да развиват тези изследвания). Или пък купуват малки компании, изградени около нечий изобретателски мозък. В нито един от случаите изобретението не е присъщо на катедралната култура; наистина, много от вложените по този начин изобретения свършват тихо задушени под „масивното ниво на управлението“, които авторите на документите Halloween така възхваляват.

Това обаче е негативна позиция. Читателят би ме разбрал по-добре от положителна такава. Като експеримент аз предлагам следното:

1. Изберете критерий за оригиналност, който смятате, че можете последователно да прилагате. Ако определението ви е „Разбирам го, като го видя“, то не е проблем за целта на този тест.
2. Изберете коя да е операционна система със затворен код, конкурираща Linux, и най-добрият код, за да прецените текущата разработка по нея.
3. Наблюдавайте този код и Freshmeat в продължение на един месец. Всеки ден преброявайте обявите за нови издания във Freshmeat, които смятате за „оригинална“ работа. Прилагайте същото определение за „оригиналност“ на обявите за другата ОС и ги пребройте.
4. 30 дни по-късно сумирайте и сравнете двете числа.

В деня, в който написах това, във Freshmeat имаше 22 обяви за нови издания, от които три изглеждаха, като че могат в някакво отношение да подтикнат напред технологията. За Freshmeat това беше слаб ден, но бих се учудил, ако някой читател съобщи за повече от три изобретения *месечно* в който и да е канал за затворен код.

*[EGCS]* Днес разполагаме с историята на проект, който в много аспекти може да ни послужи като по-показателна проверка на базарната предпоставка, отколкото fetchmail. Това е *EGCS (Experimental GNU Compiler System)* `<http://egcs.cygnus.com>`, Експерименталната система от компилатори на GNU.

Този проект бе обявен в средата на август 1997, като съзнателен опит да се приложат идеите от ранната публична версия на „Катедралата и базарът“. Основателите на проекта почувстваха, че разработката на GCC (Gnu C Compiler), C компилатора на GNU, е в застой. За около 20 месеца след това, GCC и EGCS продължиха като паралелни продукти – и двата добивани от едно и също население от Интернет разработчици, и двата започнати от един и същ основен изходен код на GCC, и двата използващи съвсем едни и същи Unix инструменти и среда за разработка. Проектите се различаваха само по това, че EGCS съзнателно се опитваше да приложи базарната тактика, която описах по-горе. Докато GCC остана като една подобна на катедрала организация със затворена група от разработчици, която рядко пускаше нови версии.

Това дотолкова наподобяваше един контролиран експеримент, че едва ли някой би могъл изобщо да желае повече. И резултатите бяха вълнуващи. За няколко месеца, версиите на EGCS дръпнаха съществено напред във възможностите си – по-добра оптимизация, по-добра поддръжка за FORTRAN и C++. Много хора откриха, че работните „снимки“ на EGCS бяха по- надеждни от последната стабилна версия на GCC. Така основните Linux дистрибуции започнаха да се прехвърлят към EGCS.

През април 1999, Free Software Foundation (Фондацията за свободен софтуер, официалните спонсори на GCC) разпуснаха първоначалната работна група на GCC и официално връчиха контрола върху проекта на екипа, направляващ EGCS.

*[SP]* Разбира се, критиката на Кропоткин и Законът на Линус повдигат някои по-общи въпроси относно кибернетиката на социалните организации. Друга фолклорна теорема на софтуерното инженерство подсказва един от тях: Законът на Конуей, най-често гласящ: „Ако разполагате с четири групи, работещи върху един компилатор, ще получите четирипасов компилатор“. Автентичното твърдение е било по-общо: „Организациите, чиито системи за дизайн са обречени да произвеждат дизайни, които са копия на структурите на общуване в тези организации.“ Можем да го дадем малко по-сбито като: „Намеренията определят резултатите“, или дори като „Процесът става продукт“.

Значи на всеки е ясно, че в общността на отворения код организационната форма и функция съвпадат на много нива. Мрежата е всичко и навсякъде – не само Интернет, но самите работещи хора образуват разпределена, свободно свързана и равноправна мрежа, която дава сложно изобилие и се снижава много елегантно. И в двете мрежи, всеки възел е важен само доколкото останалите възли желаят да си сътрудничат с него.

Равноправният аспект е съществен за удивителната продуктивност на общността. Въпросът за отношенията на сила, който Кропоткин се опитва да посочи, е развит по-нататък от „Принципа SNAFU“

[2]: „Истинска комуникация е възможна само между равни, защото подчинените по-систематично биват награждавани за това, че съобщават приятни лъжи на висшестоящите, вместо за това, че съобщават истината.“ Творческата работа в екип съвършено зависи от истинската комуникация и затуй е много сериозно спъвана от наличието на отношения на сила. Бивайки действително свободна от подобни отношения на сила, общността на отворения код ни учи на обратното – колко ужасно много ни струват те във формата на грешки, в понижена продуктивност, и в пропуснати възможности.

По-нататък, принципът SNAFU предрича на авторитарните организации едно все по-задълбочаващо се прекъсване на връзката между онези, които взимат решения и реалността, като все повече и повече от входната информация за тези, които решават, клони към превръщане в приятна лъжа. Всеки може да види как се разиграва това в традиционното разработване на софтуер; за подчинените съществуват силни стимули да крият, пренебрегват, или омаловажават проблемите. Когато този процес стане продукт, софтуерът е бедствие.

[2] *[T-SNAFU]* По време на Втората световна война в армията на Съединените щати е използвана тази абревиатура, образувана от: „Situation Normal, All Fucked Up“, което горе-долу значи: „Положението е нормално, всичко е преебано.“ или по-свободно звучи близко до „Спокойно, моряци, корабът потъва – вода има за всички.“ За повече информация за SNAFU Principle, виж в жаргонния *справочник* <http://www.tuxedo.org/esr/jargon/html/entry/SNAFU-principle.html>. Уви, авторът не е гледал българския филм „Кит“, където сюжетът е изцяло построен върху подобен принцип. *(бел. прев.)*