**성당과 시장**

**Eric Steven Raymond**

**저작권:** 이 문서를 복사, 배포 및 수정하는 권한은 Open Publication License 버전 2.0에 따릅니다.

**개요**

성공적인 오픈 소스 프로젝트인 fetchmail을 분석한다. 이 프로젝트는 리눅스의 역사에 의해 제시된 놀라운 소프트웨어 엔지니어링 이론을 신중하게 테스트하기 위해 실행된 것이다. 이 이론들을 두 개의 근본적으로 다른 개발 스타일의 용어들로 논할 것이다. 두가지 스타일이란 상업용 소프트웨어의 ``성당'' 모델과 리눅스 세계의 ``시장'' 모델이다. 이 모델들은 소프트웨어 디버깅 작업의 본질에 대한 서로 대립되는 가설들로부터 파생되었다는 것을 보일 것이다. 그리고 나서 리눅스의 경험으로부터 ``충분히 많은 사람이 있다면, 찾을 수 없는 버그란 없다'' 는 일관된 주장을 펴고, 이기적인 에이전트의 자가수정 시스템과의 생산적인 비유를 제시한 다음, 소프트웨어의 미래를 위해 이 통찰이 가지는 의미에 대한 탐구로 마무리짓는다.

**목차**

* 성당과 시장
* 메일은 배달되어야만 한다.
* 사용자가 있다는 것의 중요성
* 일찍, 그리고 자주 릴리즈하라.
* 얼마나 많은 수의 눈이 복잡성을 주시하는가?
* 장미가 장미가 아닌 순간은 언제인가?
* Popclient가 Fetchmail이 되다.
* Fetchmail의 성장
* Fetchmail에서 배울 점 몇 가지 더.
* 시장 스타일의 개발에 필요한 선행조건들
* 오픈 소스 소프트웨어의 사회적 문맥

**성당과 시장**

리눅스는 파괴적이다. 파트타임으로 해킹을 하면서 인터넷이라는 가느다란 선만으로 연결되어 있는 전세계 수 천명의 개발자들에 의해 세계적인 수준의 운영체제가, 마치 마술처럼 만들어질 수 있었으리라고 누가 5년 전에 감히 상상이나 할 수 있었을까? 나는 분명 상상하지 못했다. 1993년 초, 리눅스가 내 레이다 화면에 잡혔을 때 나는 이미 유닉스와 오픈 소스 개발을 10년 동안 해오고 있었으며 1980년대 중반에 GNU 에 첫 번째로 공헌한 사람들 중 한 명이었다. 나는 네트워크 상에 꽤 많은 오픈 소스 소프트웨어를 발표했고, 지금도 널리 사용되고 있는 몇몇 프로그램을 개발중이거나 공동개발하고 있었다. (네트핵, Emacs VC 와 GUD 모드, xlife, 등등) 나는 프로그램이 어떻게 개발되어야 하는지 알고 있다고 생각했다.

리눅스는 내가 알고 있다고 생각한 많은 부분을 뒤집어 버렸다. 몇 년 동안이나 나는 작은 도구, 빠른 프로토타이핑, 그리고 진화적인 프로그래밍을 여러 해 동안 유닉스의 복음으로 설교해 오고 있었다. 하지만 나는 어떤 종류의 매우 중요한 복잡성이 있어서 거기에는 더 집중되고 선험적인 접근방법이 필요하다고 믿고 있었다. 가장 중요한 소프트웨어 (운영체제나 Emacs 같이 대단히 커다란 도구들)는 성당을 건축하듯이, 즉 찬란한 고독 속에서 일하는 몇 명의 도사 프로그래머나 작은 그룹의 뛰어난 프로그래머들에 의해 조심스럽게 만들어지고 때가 되기 전에 발표되는 베타버전도 없어야 한다고 생각했던 것이다.

리누스 토발즈의 개발 스타일은 - 일찍, 그리고 자주 발표하며 다른 사람들에게 위임할 수 있는 것은 모두 위임하고, 뒤범벅이 된 부분까지 공개하는 그런 스타일 - 나에게 놀라움으로 다가왔다. 고요하고 신성한 성당의 건축방식은 여기에서 찾아볼 수 없었다. 대신, 리눅스 공동체는 서로 다른 의견과 접근방법이 난무하는 매우 소란스러운 시장과 같았다. (리눅스 아카이브 사이트가 이것을 적절히 상징하고 있다. 이곳에는 누구나 파일을 올릴 수 있다) 이런 시장바닥에서 조리있고 안정적인 시스템이 나온다는 것은 거듭되는 기적에 의해서만 가능한 것처럼 보였다.

시장 스타일이 매우 효과적이라는 사실은 분명 충격이었다. 리눅스 공동체에 익숙해져 가면서 나는 개개의 프로젝트에 열심이었을 뿐만 아니라 왜 리눅스 세계가 공중분해 되지도 않고 성당건축가들이 상상하기도 힘든 속도로 계속해서 강해지는지 이해하려고 애썼다.

1996년 중반에야 이해가 되기 시작했다. 내 이론을 시험해 볼 수 있는 완벽한 기회가 오픈 소스 프로젝트의 형태로 찾아왔다. 여기에서 나는 의식적으로 시장 스타일을 시도해 볼 수 있었고, 큰 성공을 거두었다.

이 글의 나머지 부분에서는 그 프로젝트에 대해 이야기하고 효과적인 오픈 소스 개발에 대한 격언들을 제시할 것이다. 내가 이 모든 것을 리눅스 세계에서 처음 배운 것은 아니지만 리눅스 세계는 이 격언들이 특별한 의미를 가질 수 있게 해주었다. 만일 내가 옳다면, 독자들은 이 격언들로부터 리눅스 공동체가 훌륭한 소프트웨어를 만들어내는 원천이 될 수 있었던 이유를 이해할 수 있을 것이며, 독자들 자신도 더 생산적으로 되는 데 도움을 받을 수 있을 것이다.

**메일은 배달되어야 한다**

1993년에 나는 펜실베니아 주, 서 체스터(West Chester) 시의 자그마한 무료 ISP인 체스터 카운티 인터링크 (Chester County InterLink : CCIL) 에서 기술적인 측면을 담당하고 있었다. (나는 CCIL 의 공동설립자였으며 우리만의 멀티유저 게시판 소프트웨어를 작성했다 - locke.ccil.org에 telnet 으로 접속하면 볼 수 있으며 지금은 19회선으로 3000 여명의 사용자를 지원한다) 이 일 덕분에 나는 하루 24시간 내내 CCIL의 56K 회선을 통해 네트워크에 접속해 있을 수 있었다 -- 사실, 그렇게 해야만 하는 상황이었다.

그래서 나는 바로바로 배달되는 인터넷 이메일에 매우 익숙해져 있었는데 몇 가지 복잡한 이유들로 인해 내 집의 컴퓨터 (snark.thyrsus.com) 과 CCIL 사이에 SLIP 연결을 하기가 꽤 힘들었다. 마침내 성공하고 나자, 주기적으로 locke 에 접속해 메일이 왔는지 체크해 보는 것이 매우 귀찮은 일이라는 것을 알게 되었다. 내가 원하는 것은 내 메일이 snark 로 배달되어 도착하는 즉시 내가 그것을 알 수 있고, 내 컴퓨터의 도구들을 이용해 메일을 다룰 수 있게 되는 것이었다.

sendmail을 이용해 단순히 포워드 시키는 것은 소용이 없었다. 내 개인 컴퓨터가 항상 네트워크에 연결되어 있는 것도 아니고 고정적인 IP 어드레스를 가지고 있지도 않았다. SLIP 연결이 되면 내 메일을 가져와 내 컴퓨터 안에서 배달해주는 프로그램이 필요했다. 그런 프로그램이 몇 개 있었고, 대부분은 프로토콜로 POP(Post Office Protocol)을 사용했다. 물론, locke 의 BSD/OS 운영체제에는 POP3 서버가 포함되어 있었다.

하지만 내게 필요한 것은 POP3 클라이언트였다. 그래서 네트워크를 뒤져 하나를 찾아냈다. 사실 3~4개를 찾아내긴 했다. 잠시 동안은 pop-perl을 사용했지만 기본적인 기능이 빠져 있었다. 가져온 메일에서 발신인의 주소를 제대로 처리하지 못해 답장을 보낼 수가 없었던 것이다.

문제는 이런 것이었다. locke 의 사용자 중에 `Joe' 라는 사람이 나에게 메일을 보냈다고 해보자. snark 로 메일을 가져와서 그 메일에 답장을 하려고 하면 메일 프로그램은 snark 에는 있지도 않은 `Joe' 에게 답장을 보내려고 시도한다. 그래서 손으로 `@ccil.org'를 답장 받는 사람의 주소 뒤에 붙여주어야 했는데, 이것은 곧 매우 피곤한 일이 되어버렸다.

이런 일은 분명히 컴퓨터가 해주어야 하는 일이었다. 하지만 이미 있는 POP 클라이언트들 중에서는 어느 것도 이 일을 해주지 못했다. 여기에서 첫 번째 교훈을 얻을 수 있다.

1. 모든 좋은 소프트웨어는 개발자 개인의 가려운 곳을 긁는 것으로부터 시작된다. (Every good work of software starts by scratching a developer's personal itch)

명확해 보이는 교훈이긴 하지만 (``필요는 발명의 어머니'' 라는 오래된 속담이 있지 않은가) 소프트웨어 개발자들은 너무나 자주, 단지 돈 때문에 그들이 필요로 하지도 않고 좋아하지도 않는 프로그램을 만들어 내는데 시간을 쓰고 있다. 하지만 리눅스 세계에서는 그렇지 않다 - 아마도 이것이 왜 리눅스 공동체에서 만들어진 소프트웨어들의 평균적인 품질이 그렇게나 좋은지를 설명해줄 것이다.

그래서 내가 이미 있는 POP3 클라이언트들과 경쟁하는 새로운 프로그램을 곧바로 코딩 하기 시작했을까? 천만에. 나는 이미 가지고 있는 POP 유틸리티들을 조심스럽게 살피면서 스스로에게 물었다. ``내가 원하는 것과 가장 가까운 프로그램이 어느 것일까?'' 그 이유는

2. 좋은 프로그래머는 어떤 프로그램을 만들어야 할 지 안다. 위대한 프로그래머는 어떤 프로그램을 다시 만들어야 할 지 (그리고 재사용해야 할 지) 안다. (Good programmers know what to write. Great ones know what to rewrite(and reuse))

내가 위대한 프로그래머라는 말은 아니지만 흉내 내려고는 했다. 위대한 프로그래머의 중요한 특징 중 하나는 건설적인 게으름이다. 그들은 들인 노력으로가 아니라 결과로 평가 받는다는 것을 알고 있으며 완전한 무에서 시작하는 것보다는 부분적으로나마 좋은 해결책에서 시작하는 것이 거의 항상 더 쉽다는 것을 알고 있다.

리누스 토발즈 를 예로 들자면 그는 맨바닥에서 Linux를 만들어 내려고 하지 않았다. 대신 그는 386 기계를 위한 Unix 비슷한 소형 OS, Minix 의 코드와 아이디어를 재사용하는 것으로부터 시작했다. 결국 모든 Minix 코드는 사라지거나 새로 쓰여졌다 -- 하지만 Minix 의 코드가 남아있을 동안 그 코드는 나중에 Linux 가 될 어린 아기의 발판 역할을 했다.

똑같은 생각으로 나는 이미 있는 POP 유틸리티 중 코딩이 잘 되어있는 것을 찾아 개발의 기초로 사용하려 했다.

Unix 세계의 소스를 공유하는 전통은 언제나 코드 재사용에 대해 호의적이었다. (GNU 프로젝트가 Unix 자체에 대한 심각한 의혹에도 불구하고 Unix 를 기본 OS 로 선택한 것도 바로 이런 이유에서였다) 리눅스 세계는 거의 기술적인 한계에 다다를 때까지 이 전통을 받아들였다. 일반적으로 찾아볼 수 있는 오픈된 소스가 수 테라바이트에 달하는 것이다. 그래서 리눅스 세계에서는 다른 어느 곳에서보다 누군가의 거의 완성된 소스를 찾아보는데 시간을 들이는 것이 좋은 결과를 가져다 줄 가능성이 높다.

나에게도 역시 그랬다. 예전에 찾아놓은 것에다가 두 번째 검색결과를 더하니 모두 아홉 개의 후보가 생겼다. fetchpop, PopTart, get-amil, gwpop, pimp, pop-perl, popc, popmail, 그리고 upop 이었다. 내가 제일 먼저 정착한 프로그램은 오승홍 씨의 fetchpop 이었다. 헤더 재작성 기능과 더불어 몇몇 개선사항을 추가했고, 저자가 릴리즈 1.9 에 그것을 수용했다.

몇 주 후에 나는 Carl Harris 가 만든 popclient 의 코드를 들여다 보다가 문제점을 발견했다. fetchpop 에는 훌륭한 독창적인 아이디어가 들어 있었지만 (daemon 모드 같은 것) POP3 만을 처리할 수 있었고, 아마추어 티가 나는 코딩이었다. (오승홍 씨는 똑똑하기는 하지만 경험이 부족한 프로그래머였으며 그 두 가지 특징 모두를 코딩에서 볼 수 있었다) Carl 의 코드는 전문가가 만든 탄탄하면서 더 나은 코드였으나 몇가지 중요하면서도 구현하기 위해서는 약간의 잔머리가 필요한 fetchpop 의 기능들이 (내가 추가한 기능들을 포함해서) 빠져 있었다.

머물러 있을 것인가, 옮겨갈 것인가? 옮겨간다면 더 나은 개발기반을 위해 이미 해놓은 코딩을 포기해야만 했다.

옮겨가는데 실질적인 동기가 되었던 것은 다중 프로토콜 지원 여부였다. POP3 가 우체국 서버 프로토콜 중에서 가장 널리 쓰이는 것이긴 했지만 유일한 프로토콜은 아니었다. fetchpop 을 비롯하여 다른 경쟁자들은 POP2, RPOP, 또는 APOP 를 지원하지 않았고, 나는 당시에 재미 삼아서 IMAP(Internet Message Access Protocol, 가장 최근에 고안되었으며 가장 강력한 우체국 프로토콜) 을 지원해 볼까 하는 생각을 가지고 있었다.

하지만 옮겨가는 것이 좋은 생각이라는 좀 더 이론적인 이유도 가지고 있었다. 리눅스를 알기 오래 전에 배운 교훈이었다.

3. ``가지고 있는 것을 버릴 계획을 세우라 ; 언젠가는 버리게 될 것이다 (Plan to throw one away; youu will anyhow)'' (Fred Brooks, ``The Mythical Man-Month'', Chapter 11)

다른 말로 하자면, 첫 번째 해결책을 구현할 때까지도 진짜 문제가 무엇인지 이해하지 못하는 경우가 종종 있다는 것이다. 두 번째가 되어서야 어떻게 하는 것이 옳은 것인지 충분히 알게 될 수 있다. 따라서 만일 올바른 방법을 찾고 싶다면 최소한 한 번은 처음부터 다시 시작할 준비를 해 두어야 한다. 그래, fetchpop을 고친 것은 내 첫 번째 시도였어, 하고 스스로에게 말하고 나서 나는 popclient 로 옮겨갔다.

1996년 6월 25일에 Carl Harris 에게 내 첫 번째 popclient 패치를 보낸 후, 나는 그가 popclient 에 대한 흥미를 이미 잃었다는 것을 알게 되었다. 코딩이 좀 지저분했고, 자잘한 버그들이 널려있었다. 내가 수정해야 할 것이 많았고, Carl 과 나는 곧 내가 프로그램을 넘겨받는 것이 합리적이라는 데에 동의하게 되었다. 내가 알아차리지 못하는 새에 프로젝트가 차츰 궤도에 오르기 시작했다. 나는 이미 존재하고 있는 POP 클라이언트의 마이너 패치를 생각하는 것이 아니었다. 클라이언트 하나를 통째로 관리하고 있었으며 내 머리에서는 커다란 변화가 될 아이디어들이 솟아나고 있었다.

코드 공유를 장려하는 소프트웨어 문화에서는 이런 방식으로 프로젝트가 진화하기 마련이다. 이렇게 말할 수 있다.

4. 적절한 태도를 가지고 있으면 흥미로운 문제가 당신을 찾아갈 것이다. (If you have the right attitude, interesting problems will find you)

하지만 Carl Harris 의 태도가 훨씬 더 중요했다. 그는 이것을 이해하고 있었다.

5. 프로그램에 흥미를 잃었다면 프로그램에 대한 당신의 마지막 의무는 능력있는 후임자에게 프로그램을 넘겨주는 것이다. (When you lose interest in a program, your last duty to it is to hand it off to a competent successor)

토론할 필요도 없이 Carl 과 나는 우리가 가장 좋은 해결책을 찾고 있다는 것을 알고 있었다. 우리에게 남아있는 한가지 문제는 내가 적임자라는 것을 입증할 수 있느냐 하는 것이었다. 내가 그것을 증명하자 그는 기꺼이, 그리고 신속하게 행동했다. 내가 그렇게 행동할 차례가 되었을 때 나도 그만큼 잘 할 수 있기를 바란다.