**성당과 시장**

**Eric Steven Raymond**

**저작권**: 문서의 복사, 배포, 수정에 대한 권한은 Open Publication License 2.0에 따릅니다.

개요

소프트웨어 개발에서의 대립되는 두 스타일인 “성당” 모델과 “시장” 모델을 소개한다. 리눅스의 경험으로부터 “충분히 많은 사람이 있다면, 찾을 수 없는 버그는 없다”는 주장을 펴고, 자가수정 시스템과의 비유를 제시한 다음, 소프트웨어의 미래에 있어 이 통찰이 갖는 의미에 대해 탐구하며 마무리한다.

**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

**1. 성당과 시장**

리눅스의 탄생은 유닉스와 오픈 소스 도메인에서 10년간 경력을 쌓은 작성자의 관념을 완전히 깨버렸다. 많은 오픈 소스 프로젝트에 기여하면서 작성자에게 있어 큰 소프트웨어의 탄생은 마치 “성당”을 건축하듯이, 특정 몇 명의 도사들이나 작은 그룹의 뛰어난 사람들에 의해 조심스럽게 만들어져야 하며, 완성이 되기 전에는 절대로 공개되어서는 안 되는 것으로 굳어있었다.

허나 리눅스의 개발자 리누스 토발즈의 스타일은 일찍, 그리고 자주 발표하며, 다른 사람들에게 위임할 수 있는 것은 모두 위임하고, 엉망인 부분까지 공개하는 스타일, 즉 말하자면 “시장”의 스타일이었다. 작성자는 이러한 너저분한 개발 스타일의 그룹이 어떻게 공중분해 되기는커녕 도리어 상상하기 힘든 속도로 강해지는지에 대해 의문을 가졌다. 마침내 작성자는 자신에게 찾아온 기회를 이용해 “시장” 스타일을 시도해보았고 큰 성공을 거두었다.

**2. 메일은 배달되어야만 한다**

작성자가 Chester County Interlink(CCIL) 에 재직할 당시 locke라는 게시판 소프트웨어를 작성하였는데, 메일을 체크하기 위해 locke에 일일이 접속해야 하는 것에 큰 불편을 느끼고 있었다. 작성자는 메일이 자신의 개인컴퓨터로 바로 배달되어 그것을 알 수 있고, 컴퓨터의 도구들을 이용해 메일을 다룰 수 있게 되는 것이었다. 이와 관해서 작성자는 5가지 교훈을 얻었다고 한다.

1. 모든 좋은 소프트웨어는 개발자 개인의 필요로 인해 시작된다.

* 작성자는 리눅스 소프트웨어의 품질이 이러한 이유로 크게 상향 평준화 되어있다고 믿는다.

1. 좋은 프로그래머는 어떤 프로그램을 만들어야 할 지 알지만, 위대한 프로그래머는 어떤 프로그램을 “다시” 만들어야 할 지(그리고 재사용해야 할 지)안다.

* 실제로 리눅스 세계는 거의 기술적인 한계에 다다를 때까지 코드 재사용의 전통을 유지했다. 리누스 토발즈 또한 맨바닥이 아닌 Unix 비슷한 소형 OS인 Minix의 코드와 아이디어를 재사용 하였다.

1. 가지고 있는 것을 버릴 계획을 세우라. 언젠가는 버리게 될 것이다.

* 첫 번째 해결책을 구현할 때까지도 진짜 문제가 무엇인지 이해하지 못할 수가 있기 때문에, 만일 올바른 방법을 찾고 싶다면 최소한 한 번은 처음부터 다시 시작할 준비를 해두어야 한다.

1. 적절한 태도를 가지고 있으면 흥미로운 문제가 당신을 찾아갈 것이다.

* 코드 공유를 장려하는 소프트웨어 문화에서는 만약 본인이 적절한 실력과 새로운 프로젝트를 수용하려는 태도가 있다면, 비록 자신이 시작한 프로젝트가 아닐지라도 넘겨 받아 발전시켜나가는 기회를 얻을 수 있다.

1. 프로그램에 흥미를 잃었다면 프로그램에 대한 당신의 마지막 의무는 능력 있는 후임자에게 프로그램을 넘겨주는 것이다.

**3. 사용자가 있다는 것의 중요성**

개발자는 사용자가 있는 것만으로도 좋은(wonderful) 일이지만, 비단 당신이 누군가의 필요를 충족시켜 주고 있으며 주어진 업무를 잘 하는것 뿐만 아니라 사용자들을 잘 유도한다면 그들은 co-developers가 될 수 있다.

사용자들을 co-developers로 여기는 것은 least-hassle 루트들을 빠르게 향상시키며 효율적으로 debugging을 할 수 있게 한다.

이는 Linus Torvalds가 보여주기 전 까진 저평가되고 있었다. 하지만 그의 성공으로 인해 재평가 되었다.

**4. 일찍, 자주 발표하라**

일찍, 그리고 자주 발표하는 것은 리눅스 개발 모델의 중요한 부분이다. 초기 버전들은 버그가 많으며 당신 역시 사용자들의 인내심을 시험하고 싶진 않을 것이다. 그래서 cathedral-building 스타일의 개발 방식을 더 선호하게 되었다. 하지만 리눅스는 이와 정 반대의 방식을 지녔다.

리누스는 가장 효과적인 방식으로 그의 사용자들을 co-developers로 여긴 것 이였다.

일찍, 자주 발표하라. 그리고 고객의 소리를 들어라.

리누스의 혁신은 그가 개발하는 것의 복잡성에 비견될 것 만한 레벨로 만들었다는 점에 있다. 이는 리누스 토발즈가 engineering and implementation의 천재인 것에 기반한다. 리눅스의 전반적인 설계는 그의 버그와 개발의 dead-ends를 피하는 육감과 A to point B의 최소 노력 경로를 찾아내는 특성을 바탕으로 본질적이고 보수적이며 단순한 설계 방식이다. 그렇다면 그가 최대화 한 것은 무엇일까? 리누스는 해커/사용자들에게 지속적인 자극과 보상을 주었다. 자극은 그들의 자기만족의 전망이며, 보상은 하는 작업의 지속적인 향상이었다.

충분히 많은 beta-tester와 co-developer 가 주어진다면, 거의 모든 문제가 빨리 파악될 것이며, 누군가 분명히 고칠 것이다. 덜 형식적으로는 “충분히 많은 eyeballs가 주어진다면, 모든 버그들은 사라질 것이다.”로 표현되며 “Linus의 법칙”으로 부른다. 여기서 cathedral-builder와 bazaar의 핵심적인 차이가 있다. 프로그래밍에 대한 cathedral-builder의 관점으로 본다면, 버그는 어렵고 까다로우며 심오한 현상이다. 반면 bazaar의 관점으로는, 버그는 보통 쉽게 해결되는 현상으로 본다.

수 년 전, 사회학자들은 다수의 관찰자들의 의견은 그 중 무작위로 선택된 한 명의 관찰자의 의견보다 더 신뢰가 있다는 결과를 발표했다. 이는 사회학자들이 Delphi-effect라고 부르는 현상이다. 리누스가 보여준 이것은 심지어 운영체제를 디버깅하는 곳에도 사용할 수 있다는 것이다.

**5. 복잡도에 길들어진 눈은 얼마나 되는가**

시장의 방식은 디버깅과 코드 발전을 가속시켰다. 이해를 위한 한가지 핵심은 소스코드를 모르는 사용자가 제출하는 버그 리포트는 그다지 유용하지 못하다는 것이다. 근본적인 문제는 테스터와 개발자 사이의 생각이 다르다는 것이다. 그러나 오픈 소스 개발은 이러한 틀을 부셔버렸다. 테스터와 개발자가 소스 코드에 기반하여 효과적으로 소통할 수 있게 되었다. 이로 인하여 개발 시간을 절약할 수 있게 되었다.

또한 오픈 소스 프로젝트는 소통의 구조를 변화시켰다. 기존의 소프트웨어 개발에서는 Brook의 법칙이 적용되었다. (“오래된 프로젝트에 더 많은 개발자를 넣는 것은 개발 시간을 더 늦춰지게 만든다”) 즉, 개발자 사이의 소통의 수에 따라 문제가 달라진다는 것이다. 그러나 오픈 소스 프로젝트에서는 서로 분리가 가능한 병렬 작업을 수행하므로 상호 작용하지 않는다.

소스 코드에 기반한 버그 리포트가 더 효율적인 이유는 더 있다. 오류는 사용자의 사용 환경에 따라 다르게 나타날 수 있다는 사실이다. 이는 재현이 어렵고 복잡하다. 이러한 문제에서 테스터가 소스 코드에 기반하여 중요한 단서를 제공할 수 있다. 이를 통하여 오류를 빠르게 알 수 있으며 기존의 버그가 수정되었는지 여부를 빠르게 알 수 있다.

또한 복잡한 여러 증상의 오류의 경우 증상을 찾기 위하여 랜덤한 세트를 셈플링하게 되는데 여러 사람들이 동시에 셈플링을 시도한다면 누군가 가장 쉬운 해결방법을 찾아내고 관리자는 이를 릴리즈하여 같은 오류에 소비되는 시간을 아낄 수 있다.

**6. 장미가 장미 다우려면**

리누스의 행동을 연구하고 그것이 왜 성공적이었는지 이론을 만든 후, 이를 새로운 프로젝트에 적용해 보기로 했다.

그러나 이에 앞서 Popclient를 단순화할 필요가 있었다. 칼 해리스(Carl Harris)의 구현방식은 불필요한 복잡성을 보여준다. 그는 코드를 중심으로, 자료구조는 코드를 뒷받침 해주는 것으로 취급했다. 그 결과 코드는 간결하나 좋은 방법이 아니었다. 칼의 기본적인 설계가 어떤 의미를 가지는지 고민하다 C 와 같이 즉흥적으로 프로그래밍하기 힘든 언어에서는

**9. 자료구조를 휼륭하게 만드는 것이 그 반대의 경우보다 더 잘 작동한다.** 는 사실을 알게 되었다.

리누스 토발즈가 옳은 방법으로 일했다는 이론을 시험하기 위하여 다래의 방법을 사용했다.

1. 일찍, 자주 발표한다.
2. 나에게 연락해 오는 사람은 누구든지 테스터 목록에 등록한다.
3. 새로 발표할 때마다 테스터에서 이를 알려 참여를 장려한다.
4. 테스터의 이야기를 경청한다.

이러한 방법은 즉각적으로 효과를 나타냈다. 이를 통하여

**10. 테스터는 가장 중요한 자원으로 여긴다면 정말 가장 중요한 자원이 되어준다.** 라는 결론을 이끌어 낼 수 있었다.

**7. Popclient가 Fetchmail이 되다.**

처음의 Popclient는 너무 많은 기능을 가지려고 했다. MTA(Mail Transport Agent)와 MDA (Mail Delivery Agent)의 기능을 모두 가지도록 설계되었다. 하지만 Harry Hochheiser가 보내준 SMTP 코드를 봤을 때 생각을 달리 해야 했다. 이 기능을 구현할 수 있다면 MDA기능이 필요 없는 순수한 MTA가 될 수 있었다. **좋은 아이디어를 생각하는 것 다음으로 중요한 것은 사용자들의 아이디어를 깨닫는 것이다. 가끔은 더 나을 수도 있다.**

Fetchmail 에서 MDA기능을 제거하자 많은 이점들이 나타났다. 드라이버 코드 중 가장 힘든 부분이 사라졌으며 OS가 파일잠금을 지원하는지 걱정할 필요도 없어졌다. 낡아서 뒤떨어진 기능이라면 효율적으로 제거할 수 있을 때 제거해버려야 한다. **설계에 있어서 완벽함이란 더 이상 추가할 것이 없을 때가 아니라 더 이상 버릴 것이 없을 때다.**

Fetchmail은 Popclient와 다르게 순수한 MTA가 되었다. Fetchmail은 자신만의 정체성을 얻었다.

**8. Fetchmail의 성장**

Fetchmail이 성공함에 따라서 Fetchmail을 좀 더 완벽하게 만들어야 했다. 나 자신의 필요성만이 아니라 나와 상관없는 다른 사람들에게 필수적인 기능을 지원하고 추가해야 했다. 이것을 깨닫고 추가한 첫번째 기능은 멀티드롭이였다. 멀티드롭 기능을 추가하기로 한 데에는 몇몇 사용자들이 이 기능을 원하는 것도 있었지만 가장 큰 이유는 멀티드롭 어드레싱을 구현함으로써 싱글드롭의 버그를 잡아낼 수 있을 거라고 생각했기 때문이다. 그리고 그렇게 되었다. **어떤 도구여도 기대하는 방법으로 쓸모가 있어야 하지만 정말 위대한 도구는 사용자가 전혀 기대하지 않았던 용도에 알맞게 된다.**

**9. Fetchmail에서 배울점**

Fetchmail의 rc파일의 선언들이 명령형 소언어를 얼마나 많이 닮아가고 있었다. 명령형 소언어를 더 영어처럼 만들면 사용하기 쉬울 것이라고 생각할 수 있다. 하지만 프로그래머들은 정확하고 짧으며 중복을 허용하지 않는 제어구문을 선호하는 경향이 있다. 하지만 영어는 50%정도의 중복을 허용하므로 대단히 부적절한 모델로 보인다. Fetchmail 제어구문은 이런 문제를 피하려고 했다. 언어의 영역이 매우 제한되어 있기 때문이다. **언어가 튜링완전하지 않다면 구문상의 유연함이 필요하다**.

또 하나의 교훈은 불투명함에 의한 보안에 대해서이다. Fetchmail 사용자들 중에는 rc파일에 있는 패스워드를 암호화하자고 이야기하는 사람들이 있었다. 하지만 그렇게 한다고 해서 보안이 강화되지 않기 때문에 그 이야기는 받아들여지지 않았다. rc파일의 읽기 퍼미션을 얻은 사람이라면 사용자와 마찬가지로 Fetchmail을 실행시킬 수도 있고 디버깅을 통해서 코드를 뽑아낼 수도 있기 때문이다. **보안시스템은 그것이 보호하려고 하는 비밀만큼만 안전하다. 가짜비밀에 주의할 것**