**얼마나 많은 복잡도 안목에 길들여졌나**

많은 것 중 하나는 시장 스타일은 디버깅과 코드 발전을 가속했다는 것이다. 이것은 작은 수준의 일상적인 개발자 및 테스터의 행동에서 어떻게 적용되는지 정확히 이해해야 합니다. 이번 섹션(원본을 작성한 지 3년이 지난 후, 이를 읽고 자신의 행동을 다시 돌아본 개발자의 통찰을 통하여)에서 우리는 현실적인 메커니즘에 대하여 자세히 살펴볼 것이다. 전문적이지 않은 독자는 이번 섹션을 넘겨도 된다.

이해를 위한 한가지 핵심은 non-source-aware(비 소스 인식) 사용자가 일반적으로 제출하는 버그 리포트는 그다지 유용하지 않은 경향이 있다는 것을 인식하는 것입니다. 비 소스 인식 사용자는 표면적인 조짐만을 제출하는 경향이 있습니다. 그들은 부여된 환경에서 다루므로 (a) 중요한 배경 테이터를 생략, (b) 버그 재현을 위한 방법을 포함하는 경우는 거의 없습니다.

근본적인 문제는 테스터와 개발자의 mental model(정신 모델)이 다르다는 것입니다. 테스터는 밖을 보고 개발자는 안을 살핍니다.

Closed-source 개발에서 그들은 둘 다 이러한 역할에 갇혀 있으며, 서로의 과거를 이야기하고 좌절하는 경향이 있습니다.

오픈 소스 개발은 이러한 틀을 깨버립니다. 테스터와 개발자가 소스 코드에 기반하여 개발하고 이에 대하여 효과적으로 소통할 수 있습니다. 실제로 외부에서 보이는 증상을 보고하는 버그 리포트와 개발자의 소스 코드 표현에 연결되는 종류 간에는 상당한 차이점이 있습니다.

대부분의 버그는 소스코드 수준에서 불안전한 암시적인 특징에 고정됩니다. 베타 테스터 중 누군가 “라인 nnn의 범위 문제가 있다거나 또는 조건 X,Y 그리고 Z 아래에서 변수가 벗어나는 문제가 있다고 지적할 수 있는 경우 문제가 되는 코드를 간략하게 살펴보는 것으로도 수정이 가능합니다. 정확한 실패 상황을 막고 수정 사항하십시오.

따라서 개발자와 테스터의 소스 코드 인식은 좋은 커뮤니케이션 효과를 향상시킵니다. 따라서 핵심 개발 시간은 다수의 공동 작업자 와도 잘 보존됩니다.

개발 시간을 절약하는 오픈 소스 방식의 또 다른 특징은 오픈 소스 프로젝트의 소통 구조입니다. 위에서 “핵심 개발자”라는 용어를 사용했다. 이는 프로젝트 코어(일반적으로 매우 작은, 단일 코어 개발자) 와 베타 테스터 및 지원자(종종 수백에 해당하는) 를 구분합니다.

전통적인 소프트웨어 개발 조직에서 제기한 근본적인 문제는 Brook의 법칙이다. : “오래된 프로젝트에 더 많은 개발자를 넣는 것은 더 늦춰지게 만든다” 일반적으로 Brook의 법칙은 복잡도와 소통간의 비용은 개발자 수의 제곱에 비례하고 작업은 증가한다고 예측합니다.

Brook의 법칙은 경험적으로 버그가 다른 사람들이 작성한 코드 사이의 인터페이스에 집중하는 경향이 강하고 프로젝트의 소통과 동등한 오버 헤드가 사람들간의 행동 사이의 인터페이스의 수와 함께 증가하는 경향이 있다는 것을 나타냅니다. 그러므로, 개발자 사이의 소통의 수에 따라 문제는 달라지며, 이는 개발자의 humber의 제곱에 비례합니다. (더 정확하게는 N\*(N-1) / 2 에 따라 N은 개발자의 수)

Brook의 법칙 분석(결과적으로 큰 개발 그룹의 어려움)은 숨겨진 결론에 달려 있다. 프로젝트의 소통 구조는 필연적으로 완벽한 그래프를 필요로 한다. 그러나 오픈 소스 프로젝트에서는 배경 개발자는 사실상 분리가 가능한 병렬 작업을 수행하므로 거의 상호 작용하지 않습니다. 코드 변경 및 버그 리포트는 핵심 그룹을 통하여 진행되고, 작은 그룹 내에서만 오버 헤드를 지불해야 합니다.

소스 코드 수준의 버그 리포트가 더 효율적인 이규는 더 있습니다. 단일 오류는 종종 사용자의 사용 패턴과 환경에 따라 다르게 나타날 수 있다는 사실을 중점에 두어야 합니다. 이러한 오류는 재현이 어렵고 복잡하며(동적 메모리 관리 또는 인터럽트) 소프트웨어에 장기적인 문제를 만듭니다.

이러한 증상의 소스 코드 레벨의 증상을 보내는 테스터(예: 1250 근처의 신호 창에 있는 것이 맞나요? 또는 어디에서 버퍼를 초기화 하나요?) 는 다른 증상에 대한 중요한 단서를 제공할 수 있습니다. 이처럼 외부 버그가 어떤 이유로 발생했는지 알기 어려울 수 있습니다. 다른 공동 작업자는 버그가 수정되었는지 여부를 빠르게 알 수 있습니다. 다양한 경우에서 소스 레벨의 버그 리포트는 특정 수정 사항에 관계없는 행동을 유발합니다.

복잡한 여러 증상의 오류는 또한 표면적인 증상에서 실제 버그로 여러 경로를 추적하는 경향이 있습니다. 개발자나 테스터가 추적할 수 있는 추적 경로는 해당 사용자의 환경에 따라 달라질 수 있으며 시간이 지남에 따라 변경될 수 있습니다. 결과적으로 개발자와 테스터는 증상을 찾을 때 까지 랜덤한 세트를 셈플링합니다. 버그가 더 복잡할수록 해당 기술이 샘플과의 관련성을 보장할 가능성이 낮아집니다.

간단하게 재현할 수 있는 버그의 경우 랜덤이 아닌 “Semi”를 이용합니다. 디버깅과 코드 및 아키텍쳐와의 친밀성은 많은 문제가 됩니다. 그러나 복잡한 버그는 랜덤에 기반합니다. 이러한 상황에서 추적을 실행하는 것은 순차적으로 추적을 실행하는 것보다 더 효과적입니다.

이는 다른 표면 증상에서 버그로 돌아가는 것이 어려울 때 예측할 수 없는 방식으로 크게 달라지는 경우 효과는 늘어납니다. 순차적으로 샘플링하는 개발자는 첫 시도에서 어려운 경로를 택하기 쉽습니다. 반면 많은 사람들이 동시에 추적 경로를 시도한다 가정해봅시다. 그러면 그 중 하나가 가장 쉬운 경로를 즉시 찾아내고 버그를 더 짧은 시간 안에 찾아낼 것이다. 프로젝트 관리자는 새로운 릴리즈를 보고 다른 사람들은 트레이스를 실행합니다. 같은 버그에 대해 너무 많은 시간을 소비하기 전에 이를 멈출 수 있습니다.

**장미가 장미 다우려면**

리누스의 행동을 연구하고 그것이 왜 성공적이었는지에 대한 이론을 만든 후, 나는 이 이론을 내 새로운 프로젝트 (물론 훨씬 덜 복잡하고 덜 야심적인 프로젝트) 에 적용해 보기로 했다.

그러나 내가 가장 먼저 한 일은 popclient 를 더 재조직화하고 단순화한 것이었다. 칼 해리스 (Carl Harris) 의 구현방식은 매우 건강한 것이었지만 많은 C 프로그래머들에게서 볼 수 있었던 것처럼 일종의 불필요한 복잡성을 보여주고 있었다. 그는 코드를 중심적인 것으로, 자료구조는 코드를 받쳐주는 것으로 취급했다. 그 결과 코드는 아름답지만 자료구조는 임시변통(ad-hoc)으로 설계되었고, 보기에 좋지 않았다. (최소한 옛 LISP 해커의 높은 기준에서 보자면 말이다)

그리고 코드와 자료구조를 개선하는 것 말고도 나는 또다른 목적을 가지고 있었다. 그것은 popclient를 내가 완전히 이해하는 무엇인가로 진화시키는 것이었다. 이해하지 못하는 프로그램의 버그를 수정하는 책임을 맡는 것은 괴로운 일이다.

처음 한달 정도가 지날 동안 나는 그저 칼의 기본적인 설계가 어떤 의미를 가지고 있는지 따라다니기만 했다. 내가 처음으로 중요한 수정을 가한 것은 IMAP 지원이었다. 프로토콜 머신을 일반적인 드라이버와 세가지 메소드 테이블 (POP2, POP3, IMAP을 지원하는)로 재조직했다. 이것과 그 이전의 변경들은 프로그래머들이 기억해 둘만한 일반적인 원리를 보여준다. 특히 C 와 같이 즉흥적으로 프로그램하기 힘든 언어에서는.

**9. 자료구조를 훌륭하게 만들고 코드를 멍청하게 만드는 것이 그 반대의 경우보다 훨씬 잘 작동한다. (Smart data structures and dumb code works a lot better than the other way around)**

브룩스의 책 9장(Chapter 9) 에 이렇게 쓰여있다. ``내게 [코드]를 보여주고 [자료구조]를 숨긴다면 나는 계속 어리둥절할 것이다. 자료구조를 보여준다면 코드는 볼 필요도 없이 뻔한 것이다.'' 사실 브룩스는 ``흐름도'' 와 ``테이블''이라고 이야기했다. 하지만 30년간 변해온 용어들과 문화를 고려한다면 거의 똑같은 말이라고 할 수 있다.

이 시점에서 (1996년 9월 초, 일을 시작하고 6 주가 지난 후) 나는 이름을 바꿀 때가 되었다고 생각하기 시작했다. 이 프로그램은 더 이상 POP 클라이언트만이 아니었다. 하지만 설계상에 정말 새로운 것이 들어가 있지 않았기 때문에 머뭇거리고 있었다. 내가 만든 popclient 는 아직 스스로의 정체성을 확립하지 못하고 있었다.

fetchmail 이 어떻게 SMTP 포트로 가져온 메일을 포워드 시켜야 하는지 알고 난 후에는 상황이 급변했다. 그에 대해서는 잠시 후에 이야기할 것이다. 하지만 그보다 먼저, 앞서 나는 리누스 토발즈가 옳은 방법으로 일을 해냈다는 내 이론을 시험하기 위해 이 프로젝트를 수행하기로 했다고 말했다. 어떻게 시험을 했을까? 다음과 같은 방법을 사용했다.

1. 일찍, 자주 발표했다. (발표간격이 10일을 넘는 경우는 거의 없었으며 개발에 몰두해 있을 때는 하루에 한번씩 발표했다)
2. fetchmail 에 대한 일로 나에게 연락해 오는 사람은 누구든지 베타테스터 목록에 올렸다.
3. 새로 발표할 때마다 베타테스터들에게 떠들썩하게 발표를 알리며 사람들이 참여하도록 격려했다.
4. 그리고 그들의 이야기를 들었다. 설계 결정에 대해 투표를 하기도 했고 패치나 피드백을 보내올 때마다 베타테스터들을 구슬렀다.

이 단순한 방법들은 즉각 효력을 나타냈다. 프로젝트를 시작할 때부터 개발자들이라면 학수고대할 만한 버그 리포트를, 때로는 훌륭하게 수정된 코드를 받을 수 있었다. 사려깊은 비판과 팬 메일, 기능제안들을 받았다. 여기서 다음과 같은 결론을 이끌어 낼 수 있다.

**10. 베타테스터들을 가장 중요한 자원으로 여긴다면 그들은 정말 가장 중요한 자원이 되어준다. (If you treat your beta-testers as if the're your most valuable resource, they will respond by becoming your most valuable resource)**

fetchmail 의 성공을 재는 재미있는 척도 중 하나는 프로젝트 베타테스터 메일링리스트인 fetchmail-friends 의 크기이다. 이 글을 쓰고있을 때 목록에는 249 명이 있었고 1주일에 2~3명이 추가되었다.

1997 년 5월말 경에 글을 수정하면서 보니까 목록은 300명 가까이 되었고, 멤버들이 조금씩 줄기 시작했는데 그 이유가 흥미로왔다. 몇몇 사람들이 구독을 중단하면서 fetchmail 이 잘 작동하기 때문에 더 이상 메일링리스트를 보고 있을 이유가 없다고 말했다. 아마 이것이 성숙한 시장 스타일의 프로젝트가 가지는 정상적인 라이프사이클 중 하나일 것이다.