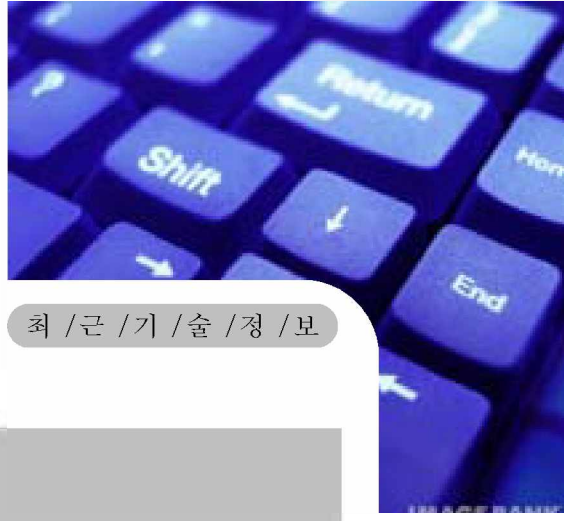


소프트웨어를 이용한 버그 해결

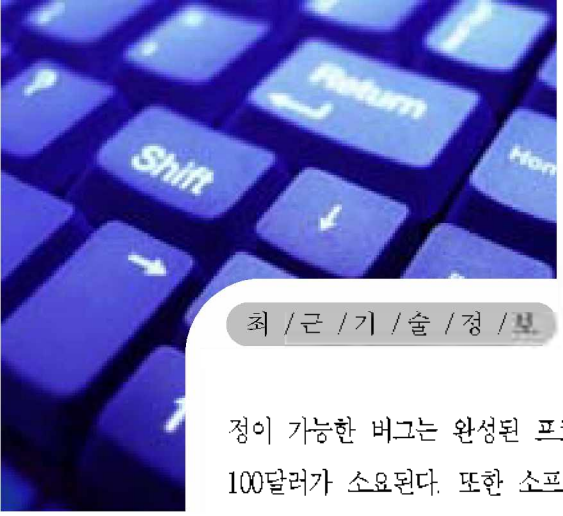
버그를 퇴치하기 위한 노력이 경주되어 왔으나 프로그래머들은 여전히 많은 실수를 한다. 소프트웨어는 소프트웨어를 더 잘 작성하는데 도움이 될 수 있을 것인가?

우리의 문명은 소프트웨어로 운영된다고 프로그래밍 분야의 대가인 Bjarne Stroustrup는 관측했다. 소프트웨어는 컴퓨터뿐만 아니라 가전제품, 자동차, 항공기, 엘리베이터, 전화기, 장남감과 다른 많은 기계류 내부에 저장되어 있다. 소프트웨어에 의존하는 사회에서, 프로그래밍 에러("버그")의 영향은 점차 의미가 증대되었다. 버그는 로케트의 충돌, 전화 네트워크의 붕괴, 또는 관제 시스템의 정지 등을 야기하였다. 2002년에 미국 국립기술표준원((NIST)이 발간한 연구자료에 따르면 소프트웨어 버그가 너무 일반적이어서 미국 경제에서 차지하는 비용이 일년에 600억 달러 또는 GDP의 0.6%에 달한다.

더 우려되는 바는 소프트웨어 기반 시스템이 널리 보급되고 상호 연결성이 높아짐에 따라 이들의 행태는 더욱 복잡해지고 있다. 코드 작성 후, 컴퓨터에서 운영하여 목적한 바대로 작동되는지 여부를 확인하고, 발생되는 문제를 수정하게 되는 구식으로 버그를 추적하는 것은 점점 비효율적이 되어 가고 있다. 프로그래머들은 신규 코드 작성보다 기존 코드의 버그를 수정하는데 훨씬 더 많은 시간을 소비한다. NIST에 따르면, 일반적인 프로젝트에서 소프트웨어 개발비용의 80%가 결함을 파악하고 교정하는데 소비된다.

이에 따라 코드 작성 시에 코드 분석이 가능하고, 테스팅 및 품질인증 절차를 자동화하는 소프트웨어 툴에 대한 관심이 증가하고 있다. 마이크로소프트 연구소의 Amitabh Srivastava 엔지니어는 "자동화 툴의 목적은 자동차 제작과 마찬가지로 소프트웨어 제작 시에 예측 가능한 품질을 획득하는 것이며, 과정이 자동화될수록 신뢰성은 높아질 것이다"고 말한다. 즉, 더 나은 소프트웨어의 제작을 위하여 소프트웨어를 사용하라는 것이다.

이 버그 퇴치 소프트웨어의 최적의 위치는 가능한 프로그래머 선에서 처리하도록 하는 것이다. 개발 과정에서 버그를 더 빨리 발견할수록 교정 비용이 더 저렴해지기 때문이다. 캐나다의 신생 기업인 클락웍(Klocwork)의 Djenana Campara 수석 기술자의 경험에 의하면, 프로그래머의 데스크탑에서 1달러에 교



정이 가능한 버그는 완성된 프로그램으로 통합되면 100달러가 소요된다. 또한 소프트웨어가 그 분야에 유포된 이후에 버그가 발견되는 경우에 수천 달러가 든다. 어떤 경우에 이 비용은 훨씬 더 높아질 수 있다. 예를 들어 통신 장비의 버그나 항공 관제 시스템의 버그로 장비가 고장나는 경우에는 수리비용으로 수백만 달러가 소요될 수 있다.

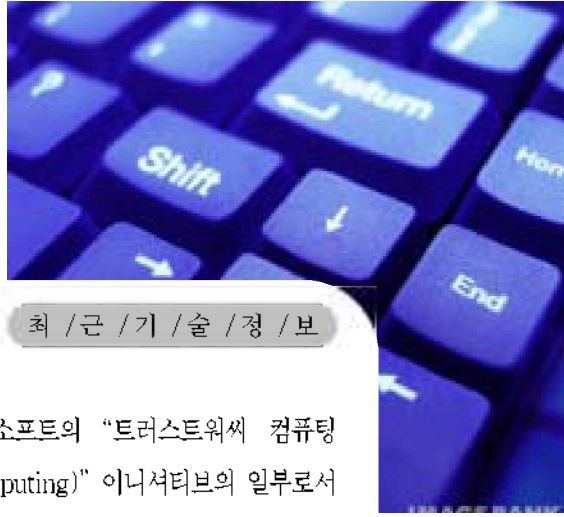
다른 소프트웨어를 조사하여 실수를 발견하기 위한 목적으로 소프트웨어를 이용하는 것은 실행이 더욱 어렵다. 컴퓨터 학자들이 소프트웨어를 분석하여 본래 임무를 잘 수행하고 있는지를 확인하기 위한 "정형화 기법"을 고안하는데 수십년이 걸렸다. 그러나 여기에는 두 가지의 큰 문제점이 있다. 첫 번째는 정형화 기법이 비례 증대되지 않는다는 사실이다. 예를 들어 3줄로 작성된 프로그램의 적절한 작동 여부를 확인하기 위해서는 한 페이지의 대수학이 필요하다. 그러나 현재 많은 프로그램들은 수백만 줄의 코드로 운영된다. 두 번째로 정형화 기법은 자동화하기 어려우므로 적절한 작동의 확인은 여전히 수작업으로 진행되는 노동 집약 공정이다.

이 분야의 많은 사람들이 정형화 기법이 대전제를 달성하는데 실패한 것으로 인식하지만, 정형화 기법이 전혀 유용하지 않은 것은 아니다. "미션 크리티컬(mission-critical)" 애플리케이션에서, 정형화 기법의 적용 비용은 가치가 있는 것으로 간주된다. 예를 들어, 이 기술은 자동차의 "임베디드" 시스템뿐만 아니라 통신 장비, 항공우주 및 군사 시스템 프로그램의 중요 부분들을 확인하기 위해 사용된다. 현재 많은 기업들은 정형화 기법에 숙달되지 않은 프로그래머들이 이러한 기술들을 널리 응용할 수 있는 소프트웨어 툴을 개발하고 있다.

해결 방법은 이들 소프트웨어 툴을 코드 작성 및 관리에 사용되는 "통합 개발환경" 소프트웨어 프로그램으로 통합하는 것이다. 기대하는 바는 이러한 신규 테스팅 및 분석 툴이 프로그래머 툴킷의 표준 부분이 되어, 더 널리 사용됨으로써 버그의 수를 꾸준히 감소시키고 소프트웨어 품질을 점차적으로 향상시키는 것이다.

► 실수에서 얻은 교훈

정형화 기법 중에서 인기 있는 기술은 특정 프로그램의 예상 작동을 수리적으로 기술하여, 이를 코드의 실제 작동과 비교하는 것이다. 문제는 수리적 기술도 종종 코드와 마찬가지로 잘못 기술될 수 있다는 점이다. 예외적으로, 잘 정의된 기능을 갖춘 소형 프로그램들은 명백하게 불규칙한 일반 목적의 프로그램보다는 분석하기가 더 쉽다. 예를 들어 마이크로소프트는 장치 드라이버의 버그 검색을 위해 "정적 드라이버 검색기"라 부르는 시스템을 사용한다. 단지 10만 줄의 코드로 이루어진 이 프로그램들은 컴퓨터와 저장 장치 또는 비디오 카드와의 인터페이스를 제공하는 등의 특정적인 제한된 업무를 실행한다. 그러나 운영 시스템 또는 웹브라우저의 예상 작동을 수리적으로 기술하는 것은 실제적으로 불가능하다.



이를 극복하는 한가지 방법은 특별한 올바른 내용을 기술하는 대신, 일반적인 틀린 내용을 기술하는 것이 더 쉽다. 좋은 예로서 너무 작은 메모리 저장부분(버퍼)에 정보를 저장할 때 야기되는 에러의 일반 형태인 "버퍼과잉"이다. 버퍼가 과잉되면 다른 데이터가 과다하게 작성되게 하고 이로 인해 프로그램이 잘못 기능하게 된다. 즉 버그가 생성된다.

버퍼과잉 에러를 방지하는 한가지 방법은 버퍼에 정보가 작성되는 상황을 파악하여, 프로그램이 버퍼 크기가 충분한지를 체크하는 코드검색 프로그램을 사용하는 것이다.

► 소스로의 회귀

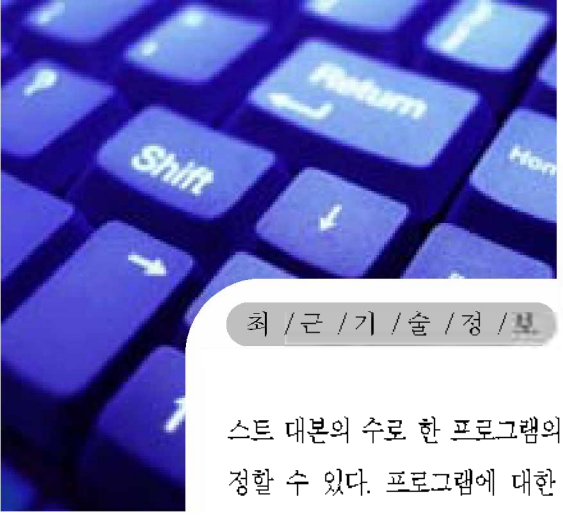
코드는 사실상 운영중인 상태는 아니며 단지 "소스 코드"로 알려진 거대한 양의 텍스트이므로, 이러한 방식으로 코드를 조사하여 버그를 검색하는 것은 "정적 분석"이라 불린다. 정적분석 툴은 보안 결함의 검색, 비효율적으로 작성된 코드의 파악, 더 이상 사용되지 않는 코드 부분의 파악 등에 사용될 수 있다. 이들은 모두 해당 코드의 특정기능에 대한 지식이 없이도 일반 목적의 정적분석 툴을 사용하여 실행할 수 있다.

마이크로소프트는 프로그래머의 데스크탑에서 운영되며 새로 작성된 코드로 정적 분석을 수행하는 PREfast라 불리는 툴을 개발하였다. 이는 코드를 더 진행하기 전에 신속하게 에러를 발견할 수 있다. PREfast는 자사 소프트웨어를 좀 더 안정적으로 제작하려는 마이크로소프트의 "트러스트워씨 컴퓨팅(Trustworthy Computing)" 이니셔티브의 일부로서 개발되었다. 원래는 Windows XP 운영체제의 개발 중에 사용되었으며, 회사 전체적으로 채용되고 있다. 마이크로소프트는 머지 않아 이 툴을 외부 프로그래머들에게 개방할 계획이다.

버그는 또한 특정 프로그램의 업데이트 버전을 효과적인 것으로 알려진 구버전과 비교함으로써 발견될 수 있다. 마이크로소프트 워드의 스펠링 체크 기능을 갱신하는 마이크로소프트 프로그래머를 상상해보자. 일단 수정을 완료하면, 프로그램의 신규 버전을 설치한다. 이를 테스트하기 위해, 이 프로그램 작동(이 경우에는 스펠링 체크)의 다른 측면의 테스트를 위해 설계되어 미리 준비된 "대본" 세트를 검색할 것이다. 이들 대본은 프로그램의 신규 버전이 구 버전과 마찬가지로 작동하는지를 확인하기 위해 실행된다.

이는 마이크로소프트에서 사용된 테스팅 방법이며, 동일 방법이 다른 곳에서도 널리 사용된다. 그러나 문제는 운영중인 테스트 대본을 알고 있는 프로그래머에게 의존한다는 사실이다. 이는 스펠링 체크의 경우에는 매우 명확하지만, 일반적으로 특정 프로그램 작동의 어느 측면이 일부의 변경에 의해 영향을 받을 것인지는 덜 명확하다. 이에 따라 마이크로소프트는 "바이너리(binary) 매칭" 기술을 이용하여 프로그램의 구 버전과 신규 버전을 비교하는 스카우트(Scout) 시스템을 고안하였다. 이는 변경된 비트를 결정하면, 적용될 테스트 대본을 파악 해낸다.

Scout는 다른 용도로도 사용될 수 있다. 필요한 테



스트 대본의 수로 한 프로그램의 수정내용 규모를 추정할 수 있다. 프로그램에 대한 작은 변경으로 수천 개의 테스트가 요구된다면, 이는 변경이 상당히 위험하다는 사실을 나타낸다. 이 때는 반드시 필요한 경우가 아니라면, 변경하지 않는 것이 상책이 될 수 있다.

캘리포니아의 신생기업인 Agitar사는 한 단계 더 발전하는 것도 가능하다고 믿고 있다. 이 기업은 프로그램을 검사하여 테스트 대본을 자동적으로 고안해내는 Agitator라고 불리는 테스트 시스템을 개발하였다. 이로써 프로그래머들은 테스트 대본을 고안하고, 수동으로 유지하는 작업으로부터 자유로워졌다.

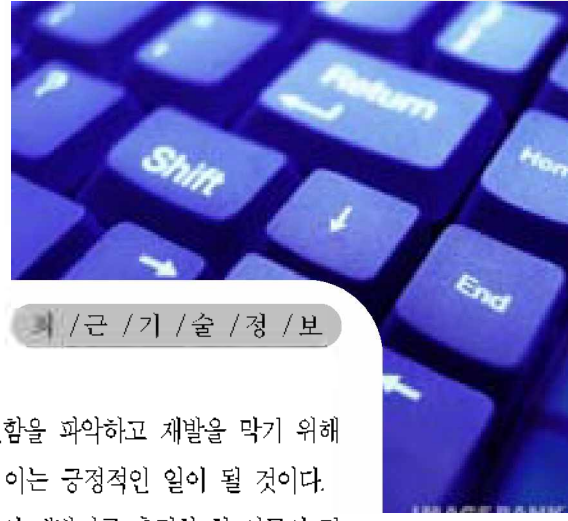
► 모델 행태

버그를 파악하는 또 다른 방법은 특정 "상위레벨" 모델 형태에서의 구조를 파악하기 위하여 프로그램 코드를 분석하는 것이다. 이 모델을 수정된 프로그램 버전에서 파생된 유사 모델과 비교함으로써 이들이 일치하는지를 확인하고, 프로그램의 갱신 및 신규 장점들이 추가되는 경우에 새로운 버그들이 유입되지 않은 사실을 보증할 수 있다. 이 방법은 신규 버그의 발견뿐만 아니라, 모델의 불일치로 나타나는 기존 버그도 일부 파악할 수 있다.

코드에서 모델이 파생될 뿐만 아니라, 최소한 어느 정도는 모델에서 코드가 파생될 수도 있다. Rational Rose와 같은 모델링 시스템은 예를 들어, 소프트웨어가 실제 코드를 작성하지 않고 "아키텍쳐 레벨"로 설계될 수 있도록 한다. 이는 프로그래머들이 코드를 삽입할 수 있는 프레임워크를 생산한다. 이에 대한 가장 대중적인 방식으로 소프트웨어 개발 기술의 개척자인 Grady Booch가 공동 발명한 "합동 모델링 언어"(UML)이라 불리는 방식이 있다. "UML은 소프트웨어에 대한 청사진 언어"라고 그는 말한다. 이는 프로그래밍 언어, 운영 시스템 및 다른 기술적 세부사항을 능가하며, 코드가 쓰여지기 전에 설계요건에 대한 프로그램 테스트를 가능하게 하는 언어이다. 정형화 기법은 점차 특히 임베디드 시스템과 같이, UML로 작성된 모델에 이용되고 있다.

그러나 이는 모델이 신성불가침으로 남아있어야 한다는 사실을 의미하지는 않는다. 코드와 모델은 동일한 내용을 보는 두 가지 방법으로 "모델을 작성한 후에 코드를 작성하는 것은 전혀 효과가 없다"고 Grady Booch는 말한다. 그는 모델에서 코드로 상달하향식 구축보다는, 증대되며 반복되는 프로세스를 지지한다. 모델과 함께 시작하여, 작동 프로그램을 제작하고, 지속적으로 모델을 갱신하며, 점차적으로 프로그램에 장점을 추가해 나가는 것이다.

모델의 사용으로 자동화의 범주는 더 확대된다. 예를 들어 코드와 모델을 동시에 유지하며, 코드의 변경 내용을 모델에 반영하는 것이다. 자동화의 최종 형태는 코딩을 모두 제거하는 것이다. 수년동안 정형화 기법 연구자들은 적절한 모델로부터 프로그램의 전체 코드를 자동적으로 파생해내는 시스템을 구축하려고 노력해왔다. 그러나 이는 희망사항일 뿐이며, 실제로 대부분의 프로그래머들은



적절하게 모델화된 적이 없는 기존 코드에 대하여 작업하므로, 모델링 툴은 이러한 레거시 코드와 함께 작동할 수 있어야 한다.

► 코딩 문화

소프트웨어 품질 개선을 위하여 클레버 툴을 사용하려는 노력은 프로그래머들간의 관계를 소원하게 한다. 잠재적 에러를 약화시키고 비효율적 코드를 파악하는 자동화된 테스팅 툴은 소프트웨어 개발을 감독할 수 있는 "메트릭스"를 제공한다. 프로젝트 매니저는 다음 내용들을 파악할 수 있다. 프로그램의 일부가 특히 높은 "결함 정도"(프로그램의 추정 크기로 나눈 버그의 수)를 나타내는지 여부. 프로그램 일부의 규모 급증 여부(비효율적 코딩 가능성의 표시), 또는 규모가 전혀 증대되지 않는지 여부 (프로그래머가 문제에 봉착함을 나타냄) 등.

이는 개발 프로세스를 한층 더 예측 가능하도록 만든다. 즉, 예상치 못한 문제나 지연이 신속하게 발견될 수 있음을 의미한다. 그러나 또한 이는 한 프로그래머와 다른 프로그래머들을 냉정하게 비교할 수도 있다. 매니저는 어느 코드가 가장 많은 버그를 포함하는지 또는 가장 비효율적으로 작성된 코드를 질문할 수 있다. 이는 일부 사람들에게 두려움의 대상이 될 수도 있지만, 각 개발자의 생산성과 품질을 감독할 수 있는 장점이 있다.

프로그래머들이 이러한 툴을 환영하거나 거부할 것인지에 대해서는 의견이 나누어진다. 중요한 사실은 처벌보다 교육 지도를 위하여 메트릭스가 적절하게 사용되는지 여부이다. 결함을 파악하고 재발을 막기 위해 교육을 이용한다면, 이는 긍정적인 일이 될 것이다.

24시간 동안 50명의 개발자를 추적한 한 연구의 결과, 이 시간의 30%만 코딩에 사용되고 있음이 발견되었다. 나머지는 팀의 다른 멤버들과 대화하는데 사용되었다. 소프트웨어 개발은 궁극적으로 팀워크 문제이다. 프로그래머들이 작업 시에, 프로그래머의 코드를 조사하는 툴은 유행중인 정보를 수집하고, 프로그램의 중 가장 빈번하게 재작성되는 부분을 파악하여, 팀이 더욱 효율적으로 작업할 수 있도록 도울 수 있다.

코딩 메트릭스는 많은 통찰력을 제공하지만, 사람들이 통찰력을 훌륭한 결정을 내리는데 사용하는 것은 별도의 문제이다. 대신 누가 무엇을 변경하는지를 추적하고, 신규 장점에 대한 요청, 다른 지역 프로그래머들과의 토론 등 프로그래밍의 사회 및 문화적 측면의 일부를 자동화하기 위해 새로운 툴 종류가 자동화된 테스팅 툴과 함께 사용될 것이다.

이러한 커뮤니케이션 툴은 테스팅 및 모델링 툴에서 제거될 것으로 보인다. 그러나 실수를 만드는 경향과 마찬가지로 통신하고자 하는 욕망을 가지는 것은 인간뿐이다. 소프트웨어를 개선하기 위해 소프트웨어를 사용하는 것은 소프트웨어의 작성자가 인간이 먼저이고, 그 다음이 프로그래머라는 사실을 인식하는 것에 달려 있다.

> 위 글은 2003년 6월 19일자 Economist지에서 발췌 번역·분석한 것임

♣