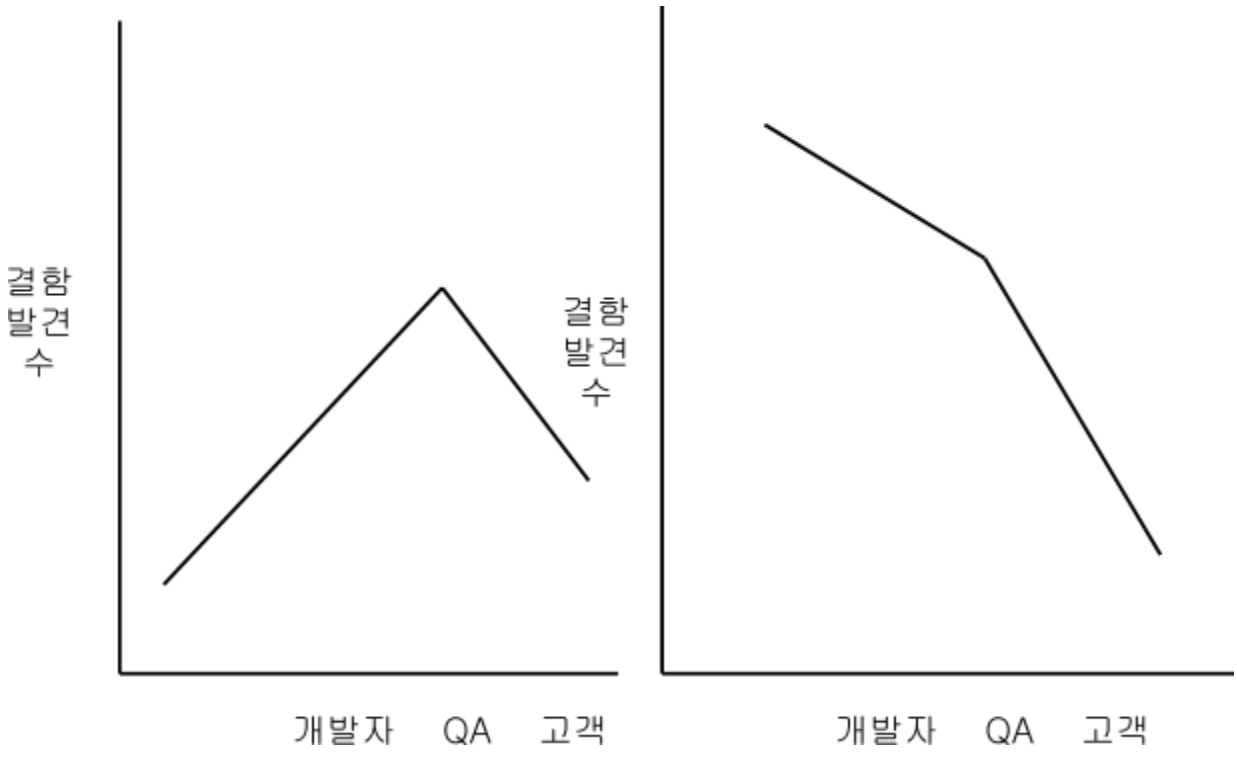
CH5. Defect Prevention

결함 발견 (Defect Detection) 보다 결함 예방 (Defect Prevention)에 초점을 두는 것은 지속 가능한 개발의 중요한 원리이다. 결함 발견은 오늘날 소프트웨어 개발에서 사용되는 가장 일반적인 방법이다. 이것은 코드를 짜고 고치는 (code-then-fix) 사고체계로서, 기능이 개발되고 나서 테스트가 이루어진 후 (유저든, 테스팅 그룹이든, QA 든) 결함이 수정되는 것이다. 이러한 접근방법에서는 결함이 유입되고 실제로 수정되기까지 상당한 시간 지연이 발생한다. 코드를 짜고 고치는 사고체계의 반대편에 있는 결함 예방은 저렴하면서도, 즉시 결함을 수정하기 위해 결함이 유입될 때 대부분의 결함을 잡아내는 자동화된 테스트를 사용해서 개발이 진행된다. 결함 예방 문화와 결함 발견 문화 사이의 차이점을 묘사하는 프로필은 그림 5-1 에 표현하였다.

그림 5-1. 결함 발견 (왼쪽) 문화에서는 결함 발견의 가장 많은 책임이 QA (또는 테스팅) 조직에 있으며, 소수의 결함이 고객에게까지 나간다. 하지만, 결함 예방 문화에서는 QA 에 제품이 도달하기 전에 상당수의 결함이 초기 단계에서 발견되어, 극소수의 결함만 고객에게 나간다.



값싸거나 손쉽거나?

몇몇 개발자가 내게 말했었다. “결함을 예방하는 것보다 발견될 때 수정 하는 게 더 값싼 방법이 아닌가요?” 이 개발자는 자기 자신을 속이는 중이고, 쉬운 것과 값싼 것을 혼동하고 있다. 확실히

기능을 내보내는데 집중하고, 수정에 대한 걱정은 나중에 하는 것이 쉬워 보인다. 하지만 이들이 놓치고 있는 점은 고객들이 수많은 결함을 찾아내고, 이것이 당신의 고객에게나 당신의 회사에 막대한 비용손실을 초래한다는 점이다. 하지만, 더욱 중요한 점은 나중에 결함을 수정하는 경우, 팀이 프로젝트의 제어할 능력을 잃게 되어, 이러한 제어 상실이 기민한 업무처리에 직접적인 영향을 준다는 점이다.

화학 제조 플랜트에서의 결함 예방은 예방적인 유지보수를 하기 위해 주기적으로 펌프의 작동을 끄는 것이다. 주기적으로 유지보수 된 펌프는 잘 고장 나지 않는 경향이 있다. 특히 자신들의 장비를 사전에 유지 보수함으로써, 플랜트 작업자들은 한 가지 위기에서 또 다른 위기로 전이되는 것을 막을 수 있다. 그들이 상황의 희생자가 되는 것이 아니라, 상황을 제어하는 것이다. 소프트웨어 개발에서의 결함 예방의 사고체계 또한 순전히 이벤트에 대응하기 보다는 좀더 제어되는 것이어야 한다.

결함 예방은 실제적인 투자 대비 효과가 있다. 결함 발견과 결함 예방의 각각의 문화에서 동일한 수의 디자인, 코딩 에러가 만들어 졌다고 가정해 보자. 결함 발견 문화에서 결함의 대다수는 코드가 제품에 통합된 후 수동 테스팅을 통해 발견된다. 반면에 결함 예방 문화에서는 코드가 제품에 통합되기 전에 결함을 발견하는 것이 강조된다. 결함 발견 문화에서 낮은 가치의 수동 테스팅이 주로 활용되고, 결함 추적 시스템 사용에 들이는 많은 공수는 아주 실제적인 비용이 든다. 왜냐하면, 결함이 유입되고 수정되는 기간 동안 시간 지연이 있다는 점이 사실이기 때문이다. 반대로, 결함 예방의 문화에서는 자동화된 테스팅과 높은 가치의 수동 테스팅을 통해, 소수의 결함만 고객에게 전달하게 되어, 팀 협업과 자동화 테스트 작성에 들어가는 추가적인 시간 비용이 상쇄된다. 정교한 재정 모델을 개발할 수도 있겠지만, 빠른 사고 전환이 이루어지는 것이 좋을 것 같다.

불행히도, 대다수의 소프트웨어 조직들은 결함 발견을 강조하고, 결함 예방 활동은 아주 조금만 수행한다. 결함 발견의 문화에서 보이는 확실한 징후들은 다음과 같다.

- 개발자들이 제품에 있는 결함을 찾아내는데 테스터/QA/사용자에게 의존한다. 개발자들이 소프트웨어를 가지고 잠시 동안 작업을 하고, 이후에 “벽 너머로 던져 버리듯이" 테스터/QA 에게 넘겨주고, 결함이 발견되면 다시 개발자에게로 넘겨진다.
- 확연히 구분될 정도의 회귀버그들을 사람들 (사용자나 QA)이 찾아낸다. 새로운 변경사항이 추가되면, 기존에 동작하던 기능들이 기대와 다르게 동작한다. 회귀버그는 충분히 테스팅이 이루어지지 않았고, 제품이 예상할 수 없는 상호의존성 때문에 리팩토링이 이루어져야 되는 신호일 수도 있다. 이것은 기술적인 빚 (technical debt) 의 또 다른 신호이다.
- 사람들은 테스팅을 포함한 많은 작업들이 많은 급여를 받는 프로그래머들이 수행하기에는 “적절치 않으며", 낮은 급여를 받는 사람의 도움을 받아야 최상의 결과를 낳는 것으로 믿는다. 따라서, 개발자들은 기능에만 (예를 들면, 코드 작성) 매진할 수 있다. 기능들이 짧은 기간 안에 완성되지만, 이 제품을 개발했던 사람들은 지속할 수 없는 개발 곡선을 보이며, 시간이 지남에 따라 개선과 유지보수에 비싼 비용이 들어간다.

- 제품의 릴리즈와 릴리즈 사이에 결함 백로그가 옮겨다 닌다. 결함 백로그는 제품에 기술적인 빚이 누적되어 있다는 증거로, 어떠한 희생을 치르더라도 이것은 막아야 한다. 한 팀이 백로그를 만난 이후, 지속적으로 더 많은 기능을 개발하라고 지속적인 압력을 받게 되면, 백로그가 정체되거나 거의 항상 계속 늘어날 것이다.
- 프로젝트 팀 멤버들은 결함과 관련된 정보의 우선순위 조정, 정렬, 확인하는데 과도한 시간을 허비한다. 결함 백로그는 거대해져서, 프로젝트 팀의 시간을 잡아먹는다. 결함 추적과 관련한 관리적인 허드렛일을 하는데 시간이 적게 들수록, 더 나아질 것이다.
- 전략적인 도구로서 버그 추정 도구의 중요성이 강조된다. 버그 추적 도구는 중요하다. 하지만, 그것들이 당신의 성공을 위한 전략이 될 수는 없다. 완전히 적합한 시스템들은 널려 있다.

어떤 문화인지 알아보는 또 다른 방법은 당신의 팀에게 아래의 질문을 던져보는 것이다.

당신은 심각한 결함을 알아내는데 선호하는 방법이 있나요?

a. 고객으로 부터
b. QA 부서로 부터
c. 자신으로 부터

결함 예방의 문화에서 당신의 대답은 c 가 될 것이다. 반면에 결함 발견의 문화에서는 아마도 b 가 될 것이다. 또 다른 문화에서는 a 가 ...

결함 발견의 문화에서 개발자는 테스팅과 제품 품질에 대한 책임을 다른 그룹, 대개는 QA 에 미룬다. 이것은 잘못이다. 왜냐하면, 개발자들이 제품을 생산하는데 사용된 프로세스와 활동을 제어하고 있으며, 제어라는 것은 기민함에 필수적인 요소이기 때문이다.

결함 발견과 결함 예방 문화 사이의 큰 차이점은 개발과 관리자가 테스팅을 대하는 태도 (다른 말로 하면, 사고체계)이다. 개발자들은 자신의 모든 능력을 동원하여 소프트웨어가 QA 나 고객에게 도달할 때까지 결함을 예방하고 잡아내야 한다. QA 에서 발견된 결함은 가능한 한 빨리 수정되어 고객이 결함을 경험하지 않도록 하는 것이 자부심이 되어야 한다. 이것이 개발자가 자신의 시간을 수동 테스팅 하는데 모두 쏟아 부어야 함을 의미하진 않는다. 오히려, 개발자가 소프트웨어 업무를 코드를 작성하는 것보다 더 큰 어떤 것으로 바라봐야 한다는 걸 의미한다. 이 장에서의 활동들은 소프트웨어 개발의 전문적인 태도를 개발하는 중요한 부분으로써 결함 예방 문화를 수립하고자 하는 의도가 있다.

QA 의 역할

결함 예방을 강조한다고 해서, QA 조직의 필요성이 없어지는 것은 아니다. 오히려, 결함 발견 문화에서 주로 요구되는 QA 의 역할과는 완전히 달라진다. 훌륭한 QA 는 제품이 고객에게 전달되기 전에 제품의 품질을 확보하기 위해 전전긍긍하며, 자신들의 직무를 충실히 수행한다. QA 의 역할은

고객이 기대하는 방식대로 새로운 기능이 동작하는지, 하나의 서비스로서 제품이 고객의 요구를 만족하는지 확인하는 것이어야 한다.

결함 발견을 중시하는 문화에서는 본인의 역할을 충족하기 위한 실제적인 능력은 제품 내에 존재하는 기술적인 빚에 의해 상당부분 영향을 받는다. 기술적인 빚이 누적되어 감에 따라, QA 는 끊임없이 회귀 테스팅에 소모되는 시간을 늘려갈 것이다. 왜냐하면, 새로운 기능이 추가되어감에 따라 이전에 존재했던 기능들이 깨질 것이며, 제품 전체는 예상치 못한 방식으로 동작하기 시작할 것이기 때문이다. 종종 회귀 테스팅은 이미 테스트된 기능, 버튼, 메뉴를 가진 제품을 아무 생각 없이 쓸어 내리는 것들로 구성된다. 회귀 테스팅 그 자체로는 고객이 제품을 어떻게 사용할지를 대변해 주지 못하기 때문에 비현실적이다. 다시 말해, 그것은 너무나 로우 레벨이다. 그러므로, 실제 유저의 문제는 아무 생각 없는 회귀 테스팅에서 거의 발견할 수 없다. 추가로 QA 팀은 다량의 결함들을 관리해야 하는 임무가 주어져 있기 때문에, 팀 구성원들은 그들의 시간을 결함을 분류하고 우선순위를 매기는 일에 점점 더 투입하게 된다. 이렇게 되면, 그들이 테스팅에 사용해야 할 시간들을 빼앗게 된다.

결국 QA 는 로우 레벨 테스팅에 더 많은 시간을 투입하게 되고, 결함 발견에 더욱 많이 의존하게 되어 새로운 기능에 결함이 존재하고 심각한 작업흐름 문제가 그대로 출시되어 고객에 의해 발견된다. 그림 5-2 에서 알 수 있듯이, 이것은 결함이 유입되고 수정되는 시간상의 지연 때문에 문제를 수정하는데 더 비싼 비용이 들어가는 상황이다. 불행히도, 결함 비용은 소프트웨어 조직 내부에서는 숨겨진 비용이다. 왜냐하면, 결함이 프로세스의 일부분인 것으로 받아들여지기 때문이다. 따라서, 사실상 일부 계산하려는 노력에도 불구하고 [Rothman 2000], 진정한 결함 비용을 계산하기는 불가능하다. 하지만, 다음에 대해서는 쉽게 수긍이 갈 것이다.

- 한 개의 결함을 수정하는 비용은 감소될 수가 없다. 소수의 개발자들만이 코드와 문제 유형을 다시 이해하는데 드는 추가적인 비용 없이 해당 코드 영역으로 되돌아갈 수 있다.
- 고객이 결함을 발견하면 가장 비용이 많이 드는데, 지원, 문제 리포팅, 추적에 추가적인 비용이 들고 무엇보다도 패치를 만들고 테스트하는데 비용이 들어가기 때문이다.

그림 5-2

(생략)

결함 발견 문화에서는 QA 팀이 요구 받는 입장이기 때문에, 새로운 기능과 작업흐름을 능동적으로 테스트하기가 어렵거나, 업무 부하를 감당하기 위해서 항상 QA 팀의 크기를 늘려야 하는 부담을 가지게 된다. 불행히도, 이 둘 중 어느 것도 지속할 수가 없다. (Unfortunately, neither solution is sustainable.)

결함 예방을 중시하는 문화에서 QA 는 중요한 것에 집중할 수 있다. 새로운 기능과 제품 전체가 적합한지가 그것이다. 이런 경우 소수의 결함만 제품에 남겨져서 고객에게 전달된다. QA 조직이 중요한 것에 집중하게 하는 유일한 방법은 개발자들이 능동적으로 결함 예방 활동들을 수용하는

것이다. (The only way to enable a QA organization to focus on what is important is for developers to proactively embrace defect prevention practices.)

프랙티스 1. 무자비한 테스팅

무자비한 테스팅은 자신이 만든 소프트웨어가 사람들 특히, QA 나 사용자들에게 전달되기 전에 가능한 한 강도 높게 테스트되는데 개발자 자신의 모든 능력을 쏟아 부어야 하는 것이다.

무자비한 테스팅은 지속 가능한 개발에서 가장 중요한 활동 중 하나로, 만일 하나의 활동에만 전념해야 한다면, 이것을 해야 한다. 무자비한 테스팅은 주로 자동화된 테스팅에 대한 것이다. 개발자들은 다음 중 어느 것도 해서는 안 된다.

- 자신들의 시간 대부분을 QA 나 사용자가 하는 방식으로 기능 테스팅을 하는 것.
- 문제를 발견할지도 모른다는 희망을 갖고, 아무 생각 없이 버튼을 누르는 것.
- 그들 자신이나 누군가가 테스트 하는데 사용할 테스팅 플랜이나 테스팅 스크립트 (예를 들면, 버튼을 누르거나 그 다음에 무엇을 하는)를 작성하는데 시간을 투자하는 것. 이것은 독자들에게 최소한의 문서화나, 높은 가치를 지닌 활동에만 전력하는 것 등을 상기시킬 것이다. 테스트 스크립트는 테스트를 문서화하기 위해 QA 조직에서 사용할 때 유용할 것이다. 하지만, 개발자들에게는 유용하지 않다.
- 테스팅 작업을 수행하기 위해 인턴이나 보조자 같은 낮은 급여의 작업자를 고용해야 한다고 대변하는 것.

위의 모든 것은 시간, 공수, 비용을 낭비하는 것이다. (All of the above would be a waste of time, effort, and money.)

개발자들은 반복적으로 아무 생각 없이 멍청한 작업들을 수행하는데 최적인 테스팅 리소스를 가지고 있다. 바로 컴퓨터이다! 테스트들은 코드 형태로 작성될 수 있다. 즉, 기능을 구현하는 것보다 더 많은 코드를 생성하는 코딩 활동을 기본적인 개발 업무로 유지하는 것이다. 하지만, 대개 기능을 구현하는 코드 보다는 테스팅하는 코드들이 더 많다. 개발자들은 여전히 다소간 현재처럼 유저 레벨의 테스팅을 해야 한다. 개발자들이 기능을 추가하거나 변경하는 과정과 결함을 고치는 과정이 다른 점은, 자동화 테스트의 형태로 안전망을 구축한다는 점이다. 이러한 자동화된 테스트는 이후로 제품이 예상한대로 동작하는지 시간이 지나면서 기대결과를 유지하는지 확인하기 위해 가능한 한 자주 실행된다.

그림 5-3 에서 무자비한 테스팅에서 활용할 수 있는 (활용해야 하는) 여러 단계와 유형의 테스팅을 나타내고 있다. 각각의 유형들은 다음 절에서의 설명처럼 다양한 효율성을 가지고 있다.

그림 5-3. 무자비한 테스팅에서 활용되어야 하는 다양한 유형, 단계의 테스트들. 테스트의 유형은 그들이 유저를 염두에 두느냐 (user-awareness), 코드 구현을 염두에 두느냐 (code-awareness) 에 의해 분류된다. 유저를 염두에 둔다는 것은 유저가 수행하는 것을 그 테스트가 얼마나 가깝게

재현해내느냐를 의미한다. 코드를 염두에 둔다는 것은 테스트에 사용되는 지식이 제품 실제 구현에 사용되는 지식과 유사한 정도를 의미한다.

테스트 자동화는 무자비한 테스팅의 중요한 부분이다. 코드를 염두에 두는 테스트는 자동화의 가능성이 높고, 최소한의 공수로 최대의 효과를 제공한다. 사용편의성, 유저 확인 (user verification), 시각적인 검증 (visual validation) 같은 유저를 염두에 두는 테스트는 거의 자동화되기 어려운데, 각각의 테스트들이 테스트를 디자인하거나 주의 깊게 관찰하거나 테스트 결과를 분석하는 등의 사람의 공수가 필요하기 때문이다. 하지만, 코드를 염두에 둔 테스트가 코드가 의도한 대로 동작하는지 판단하는 반면에, 사용자가 의도한 대로 제품이 동작하는지 항상 고민해야 하는 것은 아니다. 그러므로, 무자비한 테스팅은 코드 레벨과 유저 레벨의 테스트 양쪽을 혼합한 전략이 필요하다.

유닛 테스트: 테스트 주도 개발

유닛 테스트는 소프트웨어에서 작성할 수 있는 가장 낮은 단계의 테스트이다. 왜냐하면, 이들은 소프트웨어의 클래스, 메쏘드, 함수에 대한 명시적인 지식을 가지고 작성되기 때문이다. 이들은 본질적으로 결과가 의도대로 나오는지 확인 하기 위해 메쏘드나 함수를 호출하는 것으로 구성된다. 테스트 주도 개발은 코드를 작성하기 전에 새로운 코드에 대한 유닛 테스트를 작성하고, 새로운 변경사항이 발생할 때마다 그 테스트가 항상 통과되는지 확인하는 것이다. 테스트 주도 개발에서 테스트는 개발자가 소프트웨어를 빌드 할 때마다 수행된다. 빌드의 결과는 녹색등 (모든 테스트 통과)이나 빨간등 (테스트 하나 실패) 중 하나가 된다.

테스트 주도 개발은 소프트웨어를 개발하는 강력한 방법이다. 이것을 옵션으로 생각해서는 안 된다. 이것은 지속성을 달성하는 중요한 방법 중 하나이다. 테스트 주도 개발의 몇 가지 장점들은 다음과 같다.

- 테스트를 포함하고 있는 코드는 상당한 안정감을 가지고 수정되거나, 리팩토링 될 수 있다. 왜냐하면, 에러가 그 테스트를 통해 즉시 발견되기 때문이다. 이것은 그 코드를 최초로 작성한 개발자뿐만 아니라, 그 코드에 익숙하지 않은 개발자에게도 중요하다.
- 해당 테스트는 소프트웨어 동작에 관해 대단한 가치를 가진 문서로서 기능한다. 한 영역의 코드가 변경되어 또 다른 코드의 동작이 변경된다면, 그 테스트는 실패할 것이다. 이것은 기존에 구현된 기능에 존재하는 결함을 예방하는데 필수적이다.
- 테스트 주도 개발을 잘 활용하는 개발자는 자신의 소프트웨어를 디버깅하는 시간이 극도로 줄어들었다고 보고한다. 심지어 몇몇은 디버깅 시간이 아예 없다고도 말한다!

팁: 목 오브젝트 (Mock Objects) 를 사용하라!

당신이 데이터베이스, 네트워크 인터페이스, 유저 인터페이스, 외부 애플리케이션을 다루어야 한다면, 목 오브젝트에 대해서 배워야 한다. 목 오브젝트는 당신의 코드가 도달할 수 있는 에러나 여러 유형의

이벤트를 조작하여, 작성된 코드를 독립적으로 테스트 하기 위해, 인터페이스를 시뮬레이트할 수 있는 필수적인 디자인, 테스팅 기법이다.

팁: 자동화된 테스팅을 구현하는데 창의력을 발휘하라

자동화 테스트라고 해서 JUnit 에서 하는 것처럼 항상 호출과 기대값, 에러 조건들을 수동으로 구현하는 것을 의미하진 않는다. 많은 경우에 당신을 위해 테스트를 생성해 주는 프로그램을 작성하는 것이 가능하다. 예를 들어, 당신이 일반적인 유닉스 유틸리티 (예를 들면, grep, awk, sed) 같이 플래그 스트링들을 입력 받는 소프트웨어 조각 (메쏘드, 클래스, 컴포넌트) 을 가지고 있다고 가정해 보자. 당신은 가능한 모든 순서로 플래그들을 가지고 코드를 호출하는 프로그램을 작성할 수 있다. 이런 유형의 테스트는 극도로 꼼꼼하며, 효과를 보게 되어 있다. 나는 아무런 테스트 없이 수년간 사용해왔던 라이브러리가 이런 식으로 테스트를 추가하고, 심각한 문제가 발견되었던 케이스를 본 적이 있다. 일부 케이스는 아주 오랫동안 존재했었던 문제를 수정했으며, 이후에는 다시 발생하지 않을 것으로 생각되었다!

통합 테스트

통합 테스트는 라이브러리/모듈/컴포넌트로 묶이거나 연결되는 클래스 집합에 의해 노출되는 인터페이스를 테스트한다. 유닛 테스트와는 달리, 이들은 라이브러리의 구현에 대한 정보가 없어도 된다. 따라서, 통합 테스트는 라이브러리의 기대 동작을 테스트하고 문서화하는 유용한 메커니즘이 될 수 있다.

그림 5-4 는 유닛과 통합 테스트 간의 차이점을 나타낸다. 이 예에서 라이브러리 안에 있는 각 클래스에 대한 유닛 테스트는 클래스의 퍼블릭과 프라이빗 인터페이스를 테스트한다. 반면에, 통합 테스트는 옵션으로 간주되는데, 이들은 집합적인 유닛 테스트의 하위집합이기 때문이다. 하지만, 통합 테스트는 여전히 유용할 수 있는데, 이들이 유닛 테스트로부터 생성되었고, 별개로 관리되기 때문이다. 특히, 라이브러리에 의존하는 프로그래머라면 자신들의 코드가 라이브러리의 동작을 어떻게 예상해야 하는지 문서화하기 위해 통합 테스트를 사용할 수 있다. 통합테스트가 가장 유용한 두 가지 상황은 라이브러리가 다수의 애플리케이션 (또는 프로그램)에서 사용되거나, 써드파티 라이브러리가 사용될 때이다.

그림 5-4. 통합 테스트는 유닛 테스트와는 다르다. 이 예에서 Omega 라 불리는 라이브러리는 세 개의 클래스로 구성된다. 각각의 클래스는 자신만의 유닛 테스트를 가지고 있으며, 이 라이브러리는 별도의 통합 테스트 집합을 가진다.

통합 테스트를 사용해야 하는 시점의 한 가지 예는, 당신이 써드파티 라이브러리를 다루거나, 제품에 프리컴파일된 형태의 라이브러리를 포함시킬 때이다. 당신에게 필요한 것은 테스트 하니스 (test harness) 와 당신이 어떻게 라이브러리와 메쏘드를 호출해야 하는지 또 그러한 메쏘드를 호출할 때 예상되는 동작을 현실적으로 나타내는 일련의 테스트들이다. 때로는 라이브러리에서 제공하는 모든

인터페이스를 호출할 필요는 없는데, 이는 당신의 상황에 달렸다. 새 버전의 라이브러리가 당신에게 전달되었다면, 당신이 해야 되는 일은 당신의 테스트 하니스에 넣고, 테스트를 돌리는 것이다. 당신의 소프트웨어 아키텍쳐를 써드파티 라이브러리로 생각해서 이런 식으로 점점 더 많이 테스트 할수록, 당신의 아키텍쳐는 더욱 단단한 인터페이스를 가지게 될 것이다. 잘 구성된 인터페이스를 가지게 되면, 전체 시스템의 깨짐 없이 이러한 인터페이스 중 하나를 언제든지 교체해야 할 때, 대단한 유연성을 가지게 된다. 이것은 대단한 이득이며, 쉽게 테스트 될 수 있는 잘 디자인된 인터페이스를 가졌다고 말할 수 있다.

자동화된 테스팅의 중요성

나는 네트워킹 소프트웨어를 만드는 팀에서 처음으로 자동화된 테스팅의 중요성을 배웠다. 우리 팀은 자체적인 라이브 테스트 네트워크를 가지고 있었는데, 소프트웨어에 변경이 있을 때마다 테스트 네트워크에 최신의 소프트웨어를 넣고 여러 테스트를 수행했다. 몇몇 테스트는 수동으로 하는 것이지만, 가장 효과적인 테스트는 자동화된 통합 테스트였다. 우리의 자동화된 테스트는 하루 24 시간, 매주 7 일동안 동작했다. 왜냐하면, 종종 며칠씩, 상당한 시간이 지나지 않으면 나타나지 않는 문제를 발견했기 때문이다. 나는 리뷰에서 문제가 없었던 것으로 판단되었던 변경사항이 실제로는 아주 찾기 어려운 결점을 가지고 있어서, 반복적인 스트레스 테스팅이나 수시간의 동작 이후에만 결함이 발생하는 많은 상황을 기억한다.

이름을 밝힐 수 없는 아주 규모가 큰 소프트웨어 애플리케이션 회사와 작업했던, 자동화된 테스트의 효과를 알 수 있었던 특별한 에피소드를 기억한다. 그들은 우리 네트워크 프로토콜 스택의 결함이라고 생각하는 문제를 발견했고, 그것 때문이 출시할 수 없다고 판단했다. 우리가 리포트를 받았을 때 우리들은 우리 내부의 자동화된 테스트가 해당 케이스를 커버하고 있었기 때문에 의아하게 생각했다. 이야기 끝에, 우리는 우리 경쟁사의 프로토콜 스택에서는 그들의 코드가 작동하기 때문에, 그 문제가 우리의 문제라고 생각한다는 걸 알게 되었다. 그들은 아주 완고했고, 문제가 그들의 코드에 있음을 인정하지 않았다. 따라서, 그들의 소프트웨어 바이너리를 입수해서 우리 테스트 네트워크에 설치한 후, 우리가 찾으려고 하는 에러 조건이 발생할 때 멈추도록 특별한 디버거를 설치해 두고 주말 내내 동작시켰다. 월요일 아침에 사무실에 도착했을 때, 어셈블리 코드가 발생한 것이 수집되었다. 우리는 그들을 불러 “우리는 당신들의 소스 코드를 가지고 있지 않지만, 당신들은 수집된 어드레스를 통해 소스 라인을 알 수 있으니, 우리의 테스트 수트 (test suite) 를 사용해서 재현해 보세요.” 라고 말했다. 우리는 그 테스트 수트를 그들에게 보냈고, 그들은 그 문제가 실제로는 다른 운영체제에서 발생했었지만 증상은 다른 문제였다는 걸 알아냈다. 그런 다음, 그들은 사용자가 그 문제를 그들에게 보냈다는 사실을 알게 되었지만, 더 이상 문제를 재현해내지는 못했다!

시스템 테스트

시스템 테스트는 완성되고, 통합된 시스템이나 제품의 기능성을 커버하는 자동화된 테스트이다. 시스템 테스트는 소프트웨어의 구조에 대해 알지 못하며, 단지 유저의 관점에서 제품이 동작해야

한다고만 알고 있다. 시스템 테스트는 마우스 움직임, 키보드 입력 같은 유저의 액션을 흉내 내는 의도가 있다. 왜냐하면, 이러한 입력들은 의도된 순서나 의도된 시점에 애플리케이션에 입력될 수 있기 때문이다.

시스템 테스트는 잘 구현하기 가장 어려운 테스트 유형이다. 왜냐하면, 시간이 지나고 나서도 유용해지려면, 그 시스템 테스트가 소프트웨어 내부에 디자인되어야 하기 때문이다. 또한, 이미 존재하는 소프트웨어에 시스템 테스트 능력을 추가하는 것은 극도로 어렵다. 새로 개발되는 소프트웨어의 경우, 잘 고려된 시스템 테스트는 소프트웨어를 수동으로 테스트할 때 드는 사람에 대한 의존성을 감소시키고, 사람에게 전달될 때 (예를 들면, QA 나 사용자) 결함을 예방하는 추가적인 메커니즘을 제공해서, 개발팀의 생산성에 큰 기여를 하므로, 추가적인 디자인 공수는 들일만한 가치가 있다.

테스트 용이성 (testability)를 제품에 고려해 넣어라. 너무나 많은 팀들이 외부적인 테스트 하니스나 사람에 의한 수동 테스팅에 의존하고, 추가적인 고려를 하지 않았음을 후회한다. 현대의 컴퓨터 애플리케이션이 복잡하기 때문에 자동화된 시스템 테스트를 작성하기 어렵다. 현대의 애플리케이션들은 멀티 쓰레드 실행, 비동기적으로 데이터를 교환하는 클라이언트와 서버, 다양한 입력 장치, 수 백 가지 인터페이스 요소를 가진 유저 인터페이스 같은 복잡성을 가지고 있다. 결과는 높은 비정형성 동작 (nondeterministic behavior)을 하게 되어 테스트 용이성을 내장하지 않으면, 재현 불가능한 문제를 만들어 내게 된다.

기록하고 재생하기 (Record and Playback)

자동화된 시스템 레벨 테스팅의 복잡성 때문에, 대부분의 소프트웨어조직들은 시스템 테스팅을 자체 QA 조직 (예를 들면, 사람)에게 넘긴다. 하지만, 이것은 대부분의 소프트웨어 특히, 신규로 개발되는 소프트웨어가 기록하고 재생하는 아키텍쳐의 도움을 크게 받을 수 있기 때문에 기회를 날려버리는 것이다.

많은 애플리케이션은 로깅 (logging) 기능을 내장하고 있다. 애플리케이션이 동작하면, 액션을 기록하고 로그 파일에 담는다. 로그 파일은 문제를 디버깅하는데 유용하다. 하지만 대부분의 경우 로깅은 프로그래머를 위한 의도가 있다. 로그 파일을 읽어서 애플리케이션에 다시 넣을 수 있고, 애플리케이션이 레코딩이 된 시점의 결과와 동일한 결과를 만들기 위해 동일 시점에 동일한 순서로 로그 파일에 저장된 모든 액션들을 애플리케이션이 실행할 수 있다고 생각해 보자. 이것이 기록하고 재생하기 이다. 이러한 아키텍쳐에서는 문제 재현과 문제 디버깅이 사소한 문제가 되어 버린다.

기록하고 재생하기는 정형성 (deterministic)과 비정형성 (nondeterministic) 애플리케이션 모두에서 효과적이다. (1) 기록하고 재생하기가 널리 적용되는 높은 비정형성을 가진 애플리케이션의 예는 컴퓨터 게임 산업에 존재한다. 컴퓨터 게임에서 게임 회사는 특정한 목적을 가진 게임 엔진을 만드는 데 많은 노력을 기울인다. 이 엔진들은 게임 지형, 고저차 (levels), 캐릭터들, 기타 아이템들을 취해서, 다수 유저의 입력에 대한 반응으로 실시간으로 그들을 렌더링한다. 게임 엔진은 게임 내의 캐릭터들이

상호작용하고, 여러 장면들이 있으며, 이러한 상호작용들이 시간, 공간, 순서의 개념을 가지는 다중입력 (멀티플레이어 게임처럼)을 고려해야 하는 복잡한 소프트웨어이다. 즉, 상호작용은 언제든지 한 캐릭터가 다른 캐릭터와 어떤 관계를 가지느냐에 따라 의존성이 제 각각이다. 이러한 복잡성 때문에 많은 게임 회사는 게임 엔진 내부에 기록하고 재생하는 기능을 포함한다. 그래서, 기록하고 재생하기 기능이 없다면 사실상 재현하고 수정하기 거의 불가능한 문제들을 빠르게 진단한다. 몇몇 3D 게임 엔진에서는 재생하기가 완전히 새로운 카메라 각도에서 실행되도록 할 수 있으며, 이것은 흡사 게임 플레이를 중간자적인 3 자 시점으로 관찰하는 것과 같다. www.gamasutra.com 같은 게임 개발자 웹 사이트에서 게임 엔진에 대한 많은 유용한 기사들을 찾아볼 수 있다.

(1) 정형성을 가진 프로그램은 입력들이 제공될 때, 단일한 실행 경로를 가져서 항상 동일한 결과를 나타내는 것을 의미한다. 비정형성 프로그램은 시간차가 있는 입력에 의존하는 복잡한 방식으로 서로간에 상호작용하는 다중 실행 경로를 가진다.

유저 인터페이스 테스팅의 자동화

시스템 테스팅에서 가장 빈번하게 인용되는 복잡성 중 하나는 제품의 유저 인터페이스를 테스팅하는 것이다. 어떤 종류의 유저 인터페이스를 가진 애플리케이션을 테스트하는 한 가지 방법은 레코딩을 통해 모든 유저 입력을 저장하고, 재생할 수 있는 기록하고 재생하기 아키텍쳐를 통하는 것이다 (또 다른 하나는 위에서 언급했던 목 오브젝트를 사용하는 것이다). 잘 구조화된 애플리케이션에서는 항상 있어야 하는 것은 아니지만, 유저 인터페이스 없이 배치 모드에서 테스트를 수행할 수 있도록 해준다. 하지만, 당신이 기록하고 재생하기 아키텍쳐를 가지지 않았더라도, 여전히 몇몇 유용한 테스트들이 자동화될 수 있다.

유저 인터페이스 “아래"에 있는 애플리케이션 로직의 유닛/통합 테스트를 사용할 수 있다. 이런 경우에 테스트를 하는 가장 효과적인 방법은 유저 인터페이스의 구현과 애플리케이션 로직을 분리해서 시작하는 것이다. 예를 들면, 모델-뷰-컨트롤러 패턴의 사용이 해당된다. 테스팅의 관점에서는 이렇게 로직을 분리하면, 소프트웨어의 UI 이외의 부분을 위한 유닛, 통합 테스트를 구현하기가 아주 쉬워진다. UI 에 대한 수동 테스팅은 여전히 필요하다. 하지만, 이것은 시각 검증 테스팅의 영역이다. 좀더 확장해서 기본적인 애플리케이션 로직이 테스트되면 될수록, 시각 검증에서는 UI 요소가 올바르게 표시되는지 확인하는 등의 유저 인터페이스의 고유한 측면에 더욱더 집중할 수 있다.

시각 검증 테스팅

시각적인 애플리케이션 (이미지, 영화를 생성하거나, 실시간으로 유저가 그러한 데이터와 상호작용하게 할 수 있는)은 자동화 테스트를 생성하기 더 복잡하다. 애플리케이션이 실제로 생성해야 하는 결과를 만들었는지 알기 위해 여러 가지 방식으로 (컴퓨터 게임 같은 경우) 프로그램과 상호작용하거나 시각적인 결과물을 사람이 봐야 하는 수많은 경우가 있다. 예를 들어, 2D 그래픽

애플리케이션에서 프로그램이 생성한 원을 확인할 수 있는 유일한 방법은 특히 안티앨리어징이 적용된 경우에는 정말로 원인지 눈으로 확인하는 수밖에 없다.

시각적인 확인이 필요한 부분이 최소한의 수준인지 명확히 하는 것이 중요하다. 이것은 검증에 집중하기 위해 가능한 한 많이 코드를 염두에 두는 테스트 (code-aware tests)를 자동화할 때만 가능하다. 이것을 회귀 테스트라고 부를 수는 없다. 다시 말하면, **사람에 의한 테스트를 현명하게 사용하고, 컴퓨터가 할 수 있는 테스트를 사람이 하지 않도록 하라.**

시각 검증의 일부 측면은 자동화가 가능하다는 점을 알아야 한다. 만일 시각적인 결과가 이미지로 추출될 수 있고, 올바른 이미지를 가진 데이터베이스 (올바른 것은 사람이 결정)가 존재한다면, 새로운 버전의 소프트웨어에서 생성된 이미지가 올바른 이미지와 동일한지 또는 원래 그래야 하기 때문에 다른지 확인하는데 사용할 수 있을 것이다.

자동화된 테스팅의 경제성을 잊지 말 것

자동화된 테스트를 작성하는데 시간 (과 비용)이 든다는 것을 기억해라. 어떤 사람들은 최초 자동화된 테스트를 만드는 비용이 수동으로 테스트를 수행하는 것에 비해서 3 배에서 30 배까지 든다고 주장한다. 그러므로, 제품을 구성하는 실제 코드의 바깥에 사용자가 사용하는 거대한 인프라를 구축하는 등 모든 것을 자동화하려고 시도하지 마라. 높은 가치를 가진 테스트나, 수동으로 수행할 때 높은 비용이 드는 테스트에 집중하라. 할 수만 있다면, 소프트웨어에 테스트용이성을 구축하라.

성능 테스팅

모든 소프트웨어 조각은 어느 정도의 성능 목표를 가져야 한다. 성능 테스트는 당신의 애플리케이션이 달성하거나 초과 달성하기를 기대하는 성능 특성을 가지고 있는지 확인한다. 성능은 한 개 또는 모든 유닛 (예, 자주 호출되는 메쏘드나 함수의 성능을 테스트 하는 등)에 대해 측정될 수 있다. 성능 테스트를 수행하는 가장 일반적인 방법은 테스트에서 시스템 타이머를 사용하고, 결과를 기록해 두고, 이전 테스트 결과와 비교하거나, 기대 범위를 못 미치는 성능이 발생하면, 테스트 실패라고 정의하는 것이다. 또 다른 방법으로 몇 가지 상용이나 오픈 소스 성능 테스트 애플리케이션이 존재한다. 내가 선호하는 것은 Shark 로, 이것은 OS X 상에서 개발자를 위해 애플사가 제공하는 것이다.

실행시간 모니터링

소프트웨어는 테스트용이성을 염두에 두고 만들어져야 한다. 유용하지만 잘 쓰이지 않는 아이디어 중 한가지는 실행시간 모니터를 만드는 것이다. 실행시간 모니터는 중요 이벤트들 (예, 메시지, 입출력, 메모리 사용량)을 활성화하고, 비활성화하고, 표시하고 로그에 저장하는 코드 조각이다. 실행시간

모니터는 자동화된 테스트에 도움이 된다. 왜냐하면, 애플리케이션이 실행하는 동안 (심지어 고객사 사이트에서 조차) 라이브 데이터를 얻는데 유용하기 때문이다.

자원사용 테스팅

자원사용 테스팅은 당신의 애플리케이션이 의도된 메모리, CPU, 시간, 디스크 사용량, 네트워크 사용량 같은 것을 사용하는지 확인하는 것이다. 리소스 사용 테스트는 사람이 리소스를 엄격하게 측정하지 못할 때 초기에 문제를 잡아내는데 도움이 된다. 각 프로젝트에서는 다른 측정값보다 더 중요하게 측정되어야 하는 고유한 리소스들이 있다. 고려해 볼 요인들은 다음과 같다.

- 애플리케이션의 유형. 웹 기반의 애플리케이션은 데스크톱 애플리케이션과 전혀 다른 리소스를 사용한다. 둘 다 데이터베이스를 사용하기는 하지만, 데스크톱 애플리케이션은 네이티브 운영 시스템의 리소스를 사용한다. 반면에, 웹 기반의 애플리케이션은 웹 서버의 리소스와 클라이언트의 네트워크 대역폭을 사용한다.
- 운영 시스템. 각 운영 시스템에는 애플리케이션이 이용 가능한 고유한 리소스들이 있다. 예를 들어, 마이크로소프트 윈도우에서 그래픽 컨텍스트에 주의를 기울이는 것이 중요한데, 애플리케이션에서 누출이 되면, 더 이상 그래픽 컨텍스트가 사용 가능하지 않는 상황에서 새로운 그래픽 컨텍스트를 생성하려고 할 때 애플리케이션이 크래쉬되기 때문이다.
- 프로그래밍 언어. C 나 C++ 같은 언어는 프로그래머가 메모리 할당을 신경 써서 관리해야 하는 측면에서 악명이 높다. 프로그래머가 신경을 쓰지 않는 경우, 메모리 누출이 발생한다 (메모리 누출은 뒤에서 상세히 설명한다).

회귀 테스팅

회귀 테스트는 새로운 코드가 만들어진 경우 기존 기능이나 버그 수정이 깨어지는 것을 예방하도록 작성된 테스트이다. 이들은 새로운 유형의 테스트가 아니다. 왜냐하면, 이 테스트들은 유닛, 통합, 시스템 테스트에 들어있어야 하는 것이기 때문이다. 하지만, 그 외의 것들도 추가된다. 결함을 수정하고 나면, 테스트를 추가했는지 확인하라. 결함 수정을 한 번 이상 하는 것보다 더 힘 빠지는 상황은 없다!

사용편의성 테스팅

사용편의성 테스팅은 앞에서 논의한 모든 다른 유형의 테스팅과는 다르다. 왜냐하면, 자동화될 수 없기 때문이다. 하지만, 유저 인터페이스가 적용된 어떤 애플리케이션이라도 사용편의성 테스팅은 지속 가능한 개발의 중요한 부분이다. 또한, 인터페이스가 무식하게 간단하지 않다면, 옵션으로 고려되어서는 안 된다. 그러므로, 나는 이것을 수행하는 것을 추천한다.

사용편의성 테스팅은...
(생략)

무자비한 테스팅을 하는데 필요한 리소스

무자비한 테스팅은 테스트를 수행하고, 결과를 얻기에 쉽고, 분석도 빠르게 할 수 있는 상황 일 때만 효과적이다. 무자비한 테스팅은 추가적인 컴퓨팅 리소스와 공수를 필요로 하며, 이는 제품의 복잡도와 크기에 비례한다. 팀이 빠르고 쉽게 수행하고, 스스로가 테스트를 분석할 수 없다면 추가적인 비용이 필요하다. 또한 분산적인 테스팅에 대한 투자는 정당하지만, 중앙식이나 공유식의 빠른 컴퓨터에 다소 비용이 들어갈 수도 있다. 특히, 밤에 테스트를 수행하는데 여분의 CPU 싸이클이 소모된다. 많은 테스트가 수행되고 있을 때, 일부 맞춤 프로그램이 결과를 간단히 보여주고 분석하는데 도움이 될 수 있다.

당신이 추천되는 모든 유형의 테스트를 할 수 없다면?

당신이 어디에 테스팅 비용을 들일 것인지 정해야 할 때, 기억해야 할 것은 어떤 자동화라도 없는 것보다는 낫다는 것이다. 어떤 것을 하든 테스트 주도 개발부터 시작해야 한다. 왜냐하면, 이것은 조직 내에 테스팅의 훈련을 서서히 주입하기 때문이다. 그런 다음, 할 수 있는 한 최대로 시스템 레벨의 테스트를 자동화한다. 가급적 기록하고/재생하기의 형태를 추천한다. 왜냐하면, 이것으로 테스팅이 소프트웨어 아키텍쳐 내에 고려되었다고 확신할 수 있기 때문이다. 당신이 사용편의성과 유저 검증 테스팅을 통해서 이것을 더 개선할 수 있다면, 당신의 프로젝트는 아주 좋은 상태에 놓이게 될 것이다.

당신이 아무 테스트도 존재하지 않는 제품을 가지고 작업해야 한다면?

테스트를 작성하지 않는 이유에 대한 변명이 현재 아무런 테스트가 없다는 사실이 되어서는 안 된다. 테스팅에 대한 어떠한 투자도 없는 것보다는 낫다. 당신이 만드는 모든 새로운 코드에 대한 테스트부터 추가해 가면서, 당신 코드에 문제 있는 영역에 대해서 테스트를 추가하고, 기존 코드를 점차 변경해 나가라.

당신이 만든 테스트가 수행하는데 오래 걸린다면?

빠른 결과 확인이 필요한 유닛 테스트 같은 테스트는 테스트 수를 줄여야 할 것이다. 당신이 이런 문제에 봉착했다면, 우선먼저 축배를 드는 일을 멈추어야 한다. 이것은 **커다란** 문제다! 테스트 수를 줄이기 위한 시금석 테스트 (cornerstone test)를 판단하기 위해 테스트 커버리지 분석을 고려해야 한다. 시금석 테스트는 다른 테스트와 겹치지 않고 최대한의 이익 (예, 커버리지)을 내는 테스트이다. 당신은 중복 커버리지의 테스트를 제거해서 테스트 수를 줄이기 위해 시금석 테스트를 사용해야 한다.

Fin.