게임 엔지니어링에 대한 새로운 태도: 변화를 받아들이고, 재사용하고, 즐겨라

Johannes Norneby, Tobias Olsson

가마수트라 등록일(2009. 8. 6)

(http://www.gamasutra.com/view/feature/4102/a_new_attitude_to_game_.php)

[Massive Entertainment(Ground Control 시리즈)의 베테랑인 Norneby 와 Olsson 은 이 심층 기술 아티클에서 소프트웨어 엔지니어링 프로세스를 게임 개발에 적용하여 비디오 게임을 계획하는 경험을 근본적으로 향상시키는 방법을 논의한다.]

소개

이 아티클에서는 게임 개발의 맥락에서 소프트웨어 엔지니어링의 네 가지 모범 관행을 설명하고 정리한다. 이 네 가지 모범 관행은 다음과 같다.

- 반복적 개발
- 요구사항 관리
- 변경 관리
- 품질 검증

우리는 게임 개발에서 흔히 나타나는 상당수의 소프트웨어 문제들은 전통적인 소프트웨어 개발 조직에서 이미 발생했고 성공적으로 해결되었다는 것을 게임 개발자들이 알기를 바란다. 이 아티클은 자신의 프로젝트에서 모범 관행들을 적용하기 위해 영감을 주는 시작점이 되도록 고안된 것이다.

소프트웨어 엔지니어링의 모범 관행

소프트웨어 개발에는 많은 심각한 문제가 따라 다닌다. 복잡도는 증가하고, 출시 기간은 줄어들며, 품질 요구사항은 치솟는다. 이 때문에 소프트웨어 개발 조직과 그 안에 있는 사람들은 큰 중압감을 느낀다. 더 이상 더 열심히 일하는 것은 불가능한 일이다. 우리는 더 현명하게 일해야 한다.

소프트웨어 엔지니어링의 모범 관행은 성공적인 소프트웨어 개발에 대한 흔히 관찰되는 상업적으로 입증된 접근법이다. 경쟁이 치열한 소프트웨어 개발의 최전선에서 경쟁하고자 하는 개발자는 자기 자신의 조직과 프로젝트의 세부사항에 이러한 관행을 적응시켜야 한다.

게임 개발

모든 컴퓨터 게임에서 가장 복잡한 부분은 그 뒤에 있는 소프트웨어이다. 소프트웨어는 게임플레이 기술, 인공 지능 알고리즘, 네트워크 프로토콜, 실시간 물리학 시뮬레이션, 음악의 압축 해제와 재생, 사운드의 필터링과 믹싱, 3D 시각화 등을 통합할 필요가 있다. 이 모든 것 외에도 복수의 플랫폼을 지원해야 하는 경우가 많다. 또한 소프트웨어는 무엇인가 재미있고 유쾌한 것을 만들기 위하여 텍스처, 모델, 애니메이션, 사운드, 스크립트 등의 형태로 막대한 양의 데이터로 패키지화되어야 한다. 즉, 기술과 예술이 복잡하게 혼합되어야 한다.

이 모든 것을 고려하면 많은 게임들이 품질이 낮은 상태에서 릴리스되고, 첫날부터 소프트웨어 및/또는 콘텐츠 패치의 형태로 수정을 요구하는 경우가 많은 것도 놀라운 일은 아니다. 이 모든 복잡한 상황으로 인하여 게임은 예산을 초과하게 되고 일의 단위가 아니라 월과 년의 단위로 출시할 시간을 놓치게 된다. 결과적으로 많은 게임 개발자들은 살아 남기 위하여 지는 싸움을 하고 있는 것이다.

우리는 이러한 문제들 중 상당수가 "전통적인" 소프트웨어 개발자들이 직면했고 자신의 조직의 모범 관행을 이행하여 해결했던 동일한 문제들이라고 주장한다.

게임 개발에서의 모범 관행

반복 개발

개념적으로 반복적으로 개발한다는 것은 소프트웨어의 개발을 반복이라고 하는 몇 개의 미니어처 프로젝트로 나누는 것으로 볼 수 있다. 이러한 스타일의 개발의 중심에 있는 것은 각 반복이 현재 프로젝트 품질 및 진행상황을 적절하게 평가하는 것을 허용하는 동작하는 실행 파일을

계속해서 전달한다는 것이다.

각 반복은 전체적으로는 어떤 추상화 수준에서 표현되는 최종 제품의 각 영역을 갖는 프로젝트의 마이크로 버전이며, 반복이 진행함에 따라 제품의 각 측면의 품질은 향상된다. 각 반복은 작동하는 소프트웨어가 각 반복의 산물이기 때문에 분석, 디자인, 구현, 테스트 및 제품의 최종 전달을 통하여 요구사항을 다루는 자연스러운 흐름을 통합한다.

이 흐름은 각 반복을 기준으로 해서 반복될 뿐만 아니라 프로젝트 기준에서도 반복된다. 가장 초기의 반복의 초점은 게임의 즐거움을 고정시키는 쪽에 더 가깝기 때문에 초점 그룹 및 요구사항과 함께 작업하는 경우가 많은 반면에 이후의 반복은 제품의 세부 기능에 더 많은 중점을 둔다. 이 작업 흐름은 일일 기준으로 나타날 수도 있는데, 개발자의 하루가 작업을 받는 것으로 시작하여, 어쩌면 어떤 작은 분석/디자인 스케치를 하면서 작업의 요구사항을 조사하고, 구현으로 이동하여, 마지막으로는 해당 기능을 시험하고 이를 제품에 통합하는 것으로 끝난다.

반복 개발 위에 있는 핵심 동인은 가능한 빨리 가능한 많은 위험을 줄이는 것이다. 게임 개발에 있어서 주요 위험은 그저 게임플레이가 충분히 즐겁지 않다는 것이다. 반복 개발은 (추상화의 매우 높은 수준에서기는 하지만) 동작하는 실행 파일이 첫 번째 반복이 완료되자마자 이용할 수 있게 되면서 끝날 무렵이 아니라 프로젝트의 초기에 이러한 근본적인 문제를 개발자가 발견할 수 있게 한다. 따라서 개발자는 예산의 큰 부분을 사용하기 전에 디자인을 변경할 수 있게 되었으며 심지어는 프로젝트를 포기할 수도 있게 되었다.

앞에서 언급하였듯이 각 반복의 중심이 되는 것은 최종 사용자에게 전달되어야 하는 안정적이고, 기능적이고, 시험이 가능한 소프트웨어 시스템을 만들어야 한다는 것이다. 게임 개발에서 이것은 제품을 핵심 이해관계자와 초점 그룹 및/또는 (어쩌면 초점 그룹이 대표할 수도 있는) 실제 최종 사용자에게 전달하는 것을 의미할 수 있다. 한 반복의 출력은 다음 반복의 입력이 되며, 가장 중요한 결점 및/또는 없는 기능은 다음 반복의 목적이 된다. 이런 식으로 개발자는 실제 동작하는 시스템에서 나오는 실제 경험에 반응하면서 반복당 한번씩 재평가를 하고 가장 적절한 방향으로 프로젝트를 돌릴 수 있는 기회를 갖게 된다.

이 절차의 한 가지 주요 영향은 게임 디자인과 게임 디자인에 대한 이해가 프로젝트와 함께 발전한다는 것이다. 본질적으로 프로젝트를 시작하기 전에 더 이상 완전한 디자인 문서에 대한 필요성이 없다. 실제로 즐거움의 파악하기 어려운 속성은 동작하는 소프트웨어 제품을 사용하여 시험하고 미세 조정해야 하는 것이기 때문에 이러한 문서를 작성하는 것은 분명히 자원의 낭비가 될 것이다. 어떠한 게임 문서도 실제 구현과의 접촉에서 살아남지 않으며 이

사실을 받아들이고 그에 따라 적절히 행동하는 것이 중요하다는 것이 우리의 계속되는 경험이다.

또한 반복 개발은 언제 게임이 "충분히 좋다"는 것을 결정하게 한다. 본질적으로 이것은 각 반복에 소비된 자원이 충분한 보상을 반환하지 않는다고 느껴질 때 프로젝트를 중단할 수 있다는 것을 의미한다. 소프트웨어는 개발 중의 어떤 특정한 시간에도 항상 "완전하다". 반복 개발의 이 속성은 매우 흥미로운 것이며 프로젝트에서 알려지지 않은 것에 관해 걱정해도 소용이 없다는 중대한 사실을 깨닫게 만든다. 현재 상태를 기준으로 게임을 개선하기 위해서 할 수 있는 최선을 다하기만 하면 되는 것이다.

추론에 의해서 앞으로 나타날 수 있는 문제점과 논란거리를 걱정하기 시작하면 분석에 의해 프로젝트가 완전히 비생산적이 되고 마비되는 일은 매우 흔한 일이다. 사례의 90%는 실제로는 절대 발생하지 않는 상황에서 문제점들에 관해 걱정하는 것은 정말로 해야 할 일이 실제로 발생하는 10%를 해결하는 것일 때 자원의 낭비이다. 처리해야 하는 것은 실제로 존재하는 구체적인 문제점들을 다루는 것이지 발생할 수도 있는 추상적인 문제점들이 아니다.

반복적으로 작업하는 것의 또 다른 주요 이점은 팀에 동기를 불어넣는 것이다. 꾸준히 발전하고 올바른 방향으로 이동하는 제품을 항상 갖는다는 것은 매우 중요한 일이다. 그 반대의 상황, 즉 수 개월, 심지어는 수년을 작업했는데도 어떠한 진전도 보이지 않는 것은 프로젝트를 망치는 것이다.

또한 반복 개발은 다른 많은 모범 관행의 기본이 되기 때문에 반복 개발은 이해하고, 이행하고, 다듬어야 할 가장 중요한 관행일 것이다.

요구사항 관리

단지 시스템이 잘못된 문제를 해결하기 때문에 실패하는 프로젝트가 많으며, 이는 소프트웨어가 최종적으로 전달될 때 고객에게 어떠한 이익도 제공하지 못한다는 것이다. 요구사항을 수집하고, 분석하고, 문서화하고, 추적하고, 조직해야 한다. 이것은 예산 내에서 시간에 맞게 제품을 전달하고자 하는 모든 조직에게 기본적인 것이다. 좋은 요구사항은 프로젝트 뒤에 있는 주요 추진력이 되어야 한다. 요구사항은 조심스럽게 다루어야 한다. 이를 수행하지 않으면 다른 모든 관행을 준수하고 실행했다고 하더라도 거의 확실한 프로젝트 실패가 나온다는 것을 의미한다.

하지만 반복 프로젝트에서는 지정된 "요구사항 단계"가 없다는 것을 강조하는 것이 중요하다. 또한 요구사항 관리 업무를 처리하는 (개발자들과 분리된) 사람들도 없다. 이렇게 분리된 사람들이 실제로 게임을 만들 책임이 있는 개발자들이 결코 읽거나 이해하지 않는

요구사항을 만들어야 한다. 그 대신 요구사항 관리는 모든 팀 구성원이 수행하는 거의 일상적인 작업이다.

전통적인 요구사항 관리

전통적인 게임 개발에서 요구사항은 (선임) 게임 디자이너에서 나오며 "게임 디자인 문서"라고 하는 빅 바이블에서 설명된다. 이것은 구세대 소프트웨어 개발의 "업무 목록(grocery list)"와 동일하며, 거의 확실하게 재미있는 게임을 만들지 못할 것이다.

게임의 "재미"라는 것은 한 장의 종이 위에 완전하게 기술한다는 것이 매우 어려운 것이다. 필요한 것은 요구사항을 평가하고, 반복하고, 다듬을 수 있는 동작하는 게임이다. 이를 깨달은 결과는 빅 바이블 게임 디자인 문서가 시간과 자원의 낭비이며 더욱 가볍고 경쾌한 접근법이 필요하다는 것이다.

반복 요구사항 관리

대개 게임 디자이너는 초점 그룹과 함께 최신의 반복을 평가한다. 그 결과로 다수의 새로운 요구사항, 더 이상 필요하지 않은 요구사항, 그리고 변경해야 하는 요구사항이 나오게 된다. 이론적으로 이러한 요구사항을 도구에 넣은 다음 반복의 초기에 우선 순위를 정하고, 분석하는 작업 등을 하게 된다.

또한 이것은 생산이 실제로 시작되기 전에 완료되는 것과 반대로 프로젝트가 끝나고 게임이 선적될 때 게임 디자인이 완료된다는 것을 의미한다. 본질적으로 우리는 게임 디자이너가 워드 프로세서나 스프레드시트를 사용하여 "게임 작성자"의 역할을 갖는 것에서 벗어나서 올바른 방향으로 이동할 수 있도록 하기 위하여 프로젝트가 반복으로부터 충분한 피드백을 얻는 것을 확실히 하는 사람이 되는 것을 제안한다.

"요구사항 백로그"의 우선 순위 정하기

전형적인 프로젝트가 하나의 반복 중에 구현될 수 있는 것보다 더 많은 요구사항과 기능 요청을 가지고 시작되면서 첫 번째 반복이 끝나자마자 요구사항의 "백로그"가 존재한다. 또한 각 반복을 평가하면 항상 새로운 요구사항이 생성되는 것과 함께 기존의 요구사항이 무효가 되거나 변경될 것이다.

우리의 경험에 따르면 이 항상 증가하는 요구사항의 "더미"의 우선 순위를 잘 정하는 것은 게임의 품질에 매우 중요하다. 이것은 전체 프로젝트 시간 프레임과 예산이 주어졌을 때 대부분의 경우 구현할 시간보다도 요구사항이 훨씬 더 많은 상황에 있기 때문이다.

결과적으로 가장 중요한 요구사항을 먼저 처리하는 것이 중요하며, 그렇기 때문에 우선 순위를 정하는 일이 중요한 것이다.

(모든 중요한 "할 일" 목록이 그렇지만) 이런 종류의 우선 순위 지정 작업은 좋은 도구의 지원을 받지 못하는 겁나는 작업이지만 프로젝트 리더와 게임 디자이너들이 직면하는 가장 중요한 작업들 중의 하나이다. 품질 좋은 게임을 만들려고 한다면 고객의 기대와 위험에 바탕을 두고 요구사항의 우선 순위를 지속적이고 현명하게 정하는 일은 반드시 해야 하는 일이다.

비젼

반복 요구사항 관리의 기반에는 게임 비젼이 있다. 이 비젼은 게임의 무엇, 왜, 그리고 누구에게 중점을 두는 짧지만 정보로 가득 찬 인위적 구조이다. 이 비젼은 게임이 무엇에 관한 것인지, 대부분의 시간 동안 플레이어가 실제로 하는 것은 무엇인지, 왜 플레이어가 이러한 것들을 하고 있는지, 주위를 둘러싸고 있는 게임 세계는 무엇인지, 게임이 플레이어에게 전달해야 하는 느낌은 무엇인지 등에 대한 답변을 한다. 이러한 내용은 최상위 수준의 기능으로 설명한다. 다른 모든 요구사항은 이러한 기능 중 하나 이상을 지원하기 위한 것이며, 이것은 논리적인 계층 구조를 구성한다.

비젼은 매우 짧게 구성된 것이기 때문에 팀의 모든 사람들이 비젼을 쉽게 읽고 이해할 수 있다. 프로젝트의 공유된 비젼을 수립하는 것이 초기 반복의 주요 중점 사항이 되어야 한다. 비즈니스에서 개발에 이르는 모든 사람이 프로젝트의 명확하고, 공유되며, 모호하지 않은 비젼을 가져야 한다. 간결한 비젼을 수립하지 못하는 것은 초점이 너무나 흐리고, 이에 따라서 프로젝트를 계속하기에 위험하기 때문에 프로젝트를 재평가하고 어쩌면 취소해야 할 수도 있는 확실한 징조이다.

품질 요구사항

품질 요구사항은 비젼에서 특별한 무게를 지니고 있다. 게임은 특히 흥미로운 품질 요구사항이 가득하다. 성능 및 플랫폼 독립과 같은 기술적인 품질 요구사항뿐만 아니라 특히 흥미 요소, 다시 플레이할 가능성, 그리고 플레이어의 느낌과 같은 더 부드럽고 불명확한 품질 요구사항이 중요한 역할을 수행해야 한다. 'A Theory of Fun for Game Design'이라는 책에서 Raph Koster 는 고려해야 할 다수의 흥미 요소를 설명하고 있다. 최근의 Gamasutra 특집에서 Tom Hammersley 역시 게임의 맥락에서 품질 요구사항을 논의하고 있다.

비즈니스와 조직의 목적

게임이 갖고 있는 비즈니스 목적에 대한 명확한 그림을 조직이 갖는 것 또한 중요한 일이다. 이것은 주어진 기간 동안 판매된 단위, 순수익, 그리고 그 밖의 경제적 측정값으로부터 다양하게 설명할 수 있다. 비즈니스 발전 쪽으로도 초점을 맞출 수 있다. 예를 들어 새 개발자는 특정 퍼블리셔들에 의해 게임이 퍼블리시되게 하거나 특정한 양의 미디어 취재를 받거나, 어떤 최소한도의 리뷰 점수를 받는다는 비즈니스 목적을 가질 수 있다. 셋째, 새로운 기술이나 프로젝트 관행을 도입하거나 진정으로 반복적으로 일을 하는 방법을 배우는 것과 같이 조직의 목적을 가질 수도 있다.

비즈니스와 조직의 목적은 프로젝트에 대한 높은 수준의 평가 기준을 구성한다.

사용자

실제로 게임을 플레이하는 사람이 누구인지를 명확히 파악하는 것은 매우 중요한 일이다. 하나 또는 여러 사용자를 구성해야 하며 게임을 플레이한 시기, 게임을 플레이한 장소, 게임을 플레이한 기간, 게임을 친구, 가족, 아이들, 배우자 등과 함께 플레이했는지의 여부, 사용자가 플레이한 다른 게임들, 사용자가 게임을 플레이한 이유 등의 질문을 조사해야 한다. 가능한 깊게 파고 들어야 하며 성, 연령 및 교육 수준과 같은 기본적인 질문에만 답해서는 안 된다.

게이머의 머릿속을 파고 들어가서 명확한 비젼을 갖는다면 단지 높은 수준의 변경사항과 요구사항이 사용자의 비젼과 목적을 지원하는지 여부를 결정하여 이러한 변경사항 및 요구사항을 평가하는 작업이 더 쉬워질 것이다. 사용자를 주의 깊게 조사하면 현재 제품에서 다루고 있지 않은 틈새를 발견할 수 있는 비즈니스 기회도 나올 수 있다. 예를 들면 다음과 같다.

*"점심 시간에 **10** 분의 세션으로 사무실에서 플레이한 실시간 전략 게임…"*

변경 관리

다수의 요인 때문에 변경은 불가피하다. 정의하는 코드의 세부사항 대 요구사항의 모호함 때문에 첫 번째 시도(반복)에서 고객의 기대에 완벽하게 맞는 것을 전달하는 것은 무척 어려운 일이다. 이 때문에 더욱 간결하고 구체적이 되기를 바라면서 요구사항의 변화가 나오게 되고 그 결과로 소프트웨어도 변화해야 된다.

"...아나킨, 너는 변화를 멈출 수 없어..."

하지만 변화는 자연스러운 것이며 두려움을 갖고 보아서는 안 된다. 소프트웨어의 핵심은 (하드웨어의 "딱딱함"과 대조적으로) "부드럽게" 남아 있는 것이며 그 때문에 수정을

가능하게 하는 것이다. 요구사항의 변경에 따라 소프트웨어를 변경할 수 있는 게임 개발자가 된다는 것은 좋은 징조이다. 당신은 변화에 탄력성이 있는 소프트웨어를 갖고, 고객의 말에 귀를 기울이며 고객은 프로젝트에 참여하는 것이다. 여러 가지 면에서 변화는 받아들여야 하는 것이다.

소프트웨어와 변화

1990 년대에 블레킹게 기술대(Blekinge Institute of Technology)에서 소프트웨어 엔지니어링을 공부하면서 우리는 소프트웨어가 빌딩을 계획하고 건설하는 것과 흡사한 엔지니어링 분야라고 배웠다. 소프트웨어 엔지니어링의 상당 부분은 요구사항을 발견하고, 문제를 분석 및 이해하고, 최적의 해결책을 디자인하고, 마지막으로는 실제로 구현하고 시험하는 것이었다.

Massive Entertainment 와 함께 일하게 된 후에 우리는 학교에서 배웠던 것의 상당 부분이 응용성이 제한되어 있었다는 것을 알게 되었다. 이것은 빠르거나 고통 없이 깨닫게 된 것이 아니었다. 실제로 우리는 학교에서 배웠던 모든 멋들어진 "아키텍처 및 디자인 패턴 관련 사항"을 우리가 만들고 있었던 게임(첫 번째 *Ground Control*)에 적용하기 위해 정말로 열심히 노력했다.

수년이 흘러가면서 소프트웨어 시스템을 계획하는 것의 중심 개념은 우리가 매일 작업 중에 겪는 현실의 면전에서 매우 많이 흘러갔으며, 이는 주로 끊임없이 변하는 요구사항의 현실을 고려할 방법이 없었다는 사실 때문이었다.

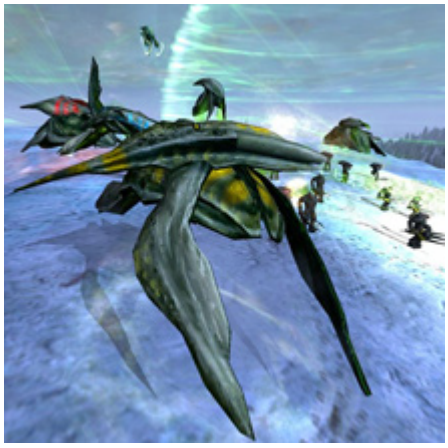
오랫동안 우리 프로그래머들은 잘못은 게임 디자이너들에게 있다고 생각했다. 우리는 "그들은 자신의 일을 하지 않았어! 일을 철저하게 생각하지 않고 있어"라고 말하곤 했다. 때때로 이것은 사실이었지만 대체로 우리는 우리 프로그래머들이 우리의 소프트웨어에 인위적인 한계를 만들고 있었다는 것을 서서히 깨닫게 되었다.

이것은 충격적이었는데, 그 이유는 우리가 우리의 우수한 스웨덴 소프트웨어 엔지니어링 기술 덕분에 우리가 문제를 해결하고 진실로 적응이 가능하면서 본질적으로는 "미래 안정적인" 시스템을 만들 수 있다고 생각했었기 때문이다. 어떤 의미로는 우리는 이것을 "객체 지향"이 되는 것의 전부로 생각하게 되었다. 우리는 "시스템"에 열중하고 있었고 "엔진"에 열중하고 있었다.

한계는 소프트웨어가 기능하게 되어 있었던 방식에 관한 우리의 가정에 바탕을 두고 있었다. 다시 이것은 게임 디자인의 초기 초안과 우리가 같은 것에서 추출한 요구사항에 바탕을 두고 있었다. 하지만 (우리가 나타날 것을 알고 있었던) 변화에 대하여 우리의 빛나는

소프트웨어 "시스템"을 얼마나 많이 준비하든, 게임 디자인의 "구멍"을 다룰 수 있도록 하기 위하여 얼마나 많은 추상화를 도입하든, 우리는 항상 부족했다는 것이 분명해졌다.

전면에서 상황이 "변경된" 날 (pun intended)



Ground Control II: *Operation Exodus* 의 후반 단계에 있던 어느 날 수석 프로그래머(이 글의 저자 Johannes)는 옛날에 배웠던 것에 정이 떨어지고 말았다. 사무실에서 매일 벌어졌던 상황은 요구사항은 너무나 빨리 변할 수 있고 다음에 들여야 할 커다란 노력의 초점은 거의 매일 변할 수 있는 것이어서 우리는 옛 계획을 고수하고 소프트웨어가 어디로 갈 것인지를 예상하려고 노력해도 아무것도 얻을 수 없는 것이었다.

나는 소규모에 있어서는 어떤 것이 좋은 코드인지에 대한 나의 생각을 버리지 않았고 소프트웨어의 상위 수준 아키텍처를 완전히 버리지도 않았지만 일반적으로는 초기의 소프트웨어 디자인을 변화하는 요구사항에 맞도록 노력하는 것을 멈추었다는 것을 알고 있다. 코드를 버리는 것이 괜찮아졌다. 오랫동안 고정되었던 인터페이스를 변경시키는 것이 괜찮아졌다.

시간의 문제였지만 얼마 안 있어 뭔가 근사한 일이 일어났다. 심각한 위기 상태에도 불구하고 나의 일을 더 즐기기 시작했다. 게임 디자이너들이 나의 "소프트웨어 시스템"을 거칠게 다루도록 하는 “고통”이 곧 "유용하고" "요점에 맞는" 것을 코딩하는 즐거운 느낌으로 바뀌었다. 프로그램을 작성하는 순수한 즐거움이 다시 거대하게 나타나는 것을 경험했으며 이것은 대부분 내가 소프트웨어의 디자인을 제어할 필요를 의식적으로 놓아주었기 때문이라고 생각한다.

"미래 안정적"이 되도록 노력하는 대신에 단지 내가 모든 것을 미리 알지 못했고 알 수도 없었다는 것을 받아들이고 내가 소프트웨어를 코딩했던 그 순간에 소프트웨어에 "되기를 원했던" 방식을 스스로가 발견하게 놔둘 수 있었다. 더욱 놀랍고 보람이 있었던 것은 더 나은 코드를 작성하기 시작했던 것이다. 코드는 크기가 작아지고, 요점에 더 가까워졌으며, 훨씬 더 최적화되었다.

그 이후로 소프트웨어 엔지니어링과 게임 개발에 대한 나의 접근법은 근본적으로 바뀌었다. 내가 작성하고 있는 소프트웨어의 근본적인 가정에 대한 변화를 요구하는 무엇인가가 나타나면 내 능력으로 해당 가정을 변경시킬 수 있다고 확신할 수 있다는 것을 배웠기 때문에 미래를 예상하기 위해 그렇게 노력하는 것을 기본적으로 멈추었다. 그 결과는 내가

해결하려고 노력하고 있는 실제 문제와 더 일치하는 코드이다. 그리고 우리는 모두 게임 개발의 현장에서는 문제가 항상 변하기 쉽다는 것을 알고 있다.

나중에 생각해 보면 이것은 그리 놀랄 일이 아니다. 소프트웨어의 주요 사항 중의 하나를 기억할 준비가 되어 있다면 그렇지 않다. 즉, 소프트웨어는 **부드러워야** 하는 것이다. 하드웨어와는 대조적으로 소프트웨어의 핵심 요소를 이용해야 했을 때 우리는 초기 기술 아키텍처를 고정된 것(즉, 딱딱한 것)으로 처리하고 있었던 것이다. 즉, 쉽게 **변경**할 수 있어야 하는 것이다.

재사용과 객체 지향 프로그래밍의 함정

재사용이라는 객체 지향 소프트웨어의 성배는 개념적으로는 정말로 대단한 것으로 보이는 것이지만, 실제로는 곤란하게 되는 경향이 있다. 시기 상조의 재사용은 종종 "모든 (소프트웨어) 악의 근원"이라고 인용되는 시간 상조의 최적화보다 훨씬 더 나쁘다.

재사용 가능성에 지나치게 마음을 빼앗긴다면 특정적이고, 적절하며, 구체적인 것인 실제 값과 기능을 프로그램에 만드는 것과 대조적으로 일반적이고, 유연하고, 플러그 가능하며, 통칭적인 것인 시스템 및 프레임워크에 관해 생각하게 된다.

그 외에도 게임 디자이너와 게이머는 특별한 경우를 원하는데, 이것은 당신이 예상하지 않았던 것이며, 이러한 특별한 경우는 당신의 웅대한 소프트웨어 디자인에 맞지 않기 때문에 당신을 화나게 만드는 것이다.

그렇다면 어떻게 하면 소프트웨어 재사용을 달성할 수 있을까? 재사용을 위하여 계획하고 디자인하는 대신에 재사용을 위한 기회를 찾아야 한다! 이것은 자신이 이미 했던 일을 반복하게 되었을 때 적용할 수 있는 경우가 많으며 그렇게 한다면 실제 공통점을 추출하여 이것을 재사용 가능한 어딘가에 두어야 한다(즉, 공통 라이브러리). 실제로 재사용 가능한 소프트웨어 요소는 발견되는 것이지 디자인되는 것이 아니다.

재사용과 리팩토링

재사용을 위한 디자인은 "괜찮음"이라는 요인에 바탕을 두는 경우가 많다. 즉, 재사용 가능 시스템을 디자인하고 구현하는 것은 "괜찮은" 것이다. 하지만 문제가 되는 사실은 필요할 수 있거나 자신이 만들고 있는 시스템으로 할 수 있는 것을 예상하기 때문에 이러한 디자인은 항상 추론적이라는 것이다. 이것은 가까이에 있는 애플리케이션의 요구사항의 범위를 벗어나기 때문에 가장 나쁜 종류의 추론이다. 이것은 소프트웨어를 작성하는 요점이 아니다. 소프트웨어를 작성하는 요점은 최종 고객에게 비즈니스 가치(또는 흥행 가치 또는 양쪽 가치)를 전달하는 유용한 시스템을 전달하는 것이다.

다시 말하지만 재사용을 위해 디자인해서는 안 된다. 이것은 추론적이며 나중에 필요할 수도 있다고 생각되는 코드보다는 지금 당장 필요하다고 알고 있는 코드를 작성하는 것이 훨씬 더 낫다. 옛 소프트웨어를 유지 보수하거나 새 소프트웨어를 작성하면서 전에 했던 무엇인가를 하고 있다고 깨달을 경우에만 어떤 코드를 재사용하는 것이다.

그 때조차도 옵션이 있다. 코드를 재사용하는 근원이 되는 코드 기반을 계속 유지 관리하고 있다면 공통점을 라이브러리로 뽑아내고 과거의 소프트웨어와 새 소프트웨어가 모두 이 공통 라이브러리에 의존하게 만든다. 옛 소프트웨어를 유지 관리하고 있지 않다면 단지 소스를 복사하면 된다.

이 견해는 몇 개의 요인에 바탕을 두고 있다. 우리는 모두 성공적이고 유용한 소프트웨어를 작성하려고 한다. 그렇지 않다면 무엇보다도 소프트웨어를 작성할 이유가 무엇이겠는가? 이러한 소프트웨어의 자연스러운 상태는 그 유용성과 부드러움 때문에 지속적인 유지 보수의 상태이다. 공통점을 추출하는 이유는 유지 관리해야 할 총 코드 수를 줄이는 것이며, 그렇게 되면 유지 관리가 쉬워진다. 그 때문에 공통 코드의 재사용이 나온다.

코드를 재사용하는 다른 이유는 단지 (소스에 접근할 수 있을 경우) 소프트웨어를 복사하기 쉽다는 것이다. 그렇다면 단순히 공짜로 복사할 수 있는 노력을 왜 중복하는가? 이것은 여전히 재사용이다. 이것은 특히 전체 애플리케이션을 복사하거나 재사용하게 될 때 매우 흔한 형태의 재사용이다. 최소한 이것은 시작할 때 아무것도 이용할 것이 없는 것에 비해서 새로운 애플리케이션에 대한 훨씬 더 나은 시작점을 제공할 것이다. 새로운 코드를 작성하는 것보다 더 예전의 코드를 재작성하거나 조정하는 것으로 끝나기를 원하지는 않기 때문에 이러한 종류의 애플리케이션 차원의 재사용을 수행하는 결정은 새 애플리케이션이 옛 애플리케이션과 얼마나 비슷한가에 많은 기준을 둔다.

재사용은 동일한 애플리케이션 내에서도 작은 규모로 나타날 수도 있다. 이것은 새로운 소프트웨어를 대상으로 막 시작할 때 전형적으로 나타나는 일이다. 고객 가치를 이행하는 자연스러운 과정에서 코드 기반의 전체 크기를 최소화하기 위하여 어떤 공통점을 뽑아내거나 어떤 방법을 매개변수화 시킬 수 있다는 것을 깨닫게 된다.

위의 모든 사항은 리팩토링과 관련이 있는데, 이것은 소프트웨어의 성장과 조정의 매우 필수적이고 자연스러운 부분인 기능을 변경시키지 않고 유지 관리성과 확장성을 강화시키기 위하여 코드를 재구성하는 것이다. 분명히 재사용은 리팩토링과 정확히 동일한 사고 방식의

일부이지만 규모와 범위에서의 차이(오직 인지된 차이일 경우가 많음) 때문에 사람들에게 혼동을 준다. 리팩토링을 통하여 소프트웨어 디자인 작업을 하는 것은 정말로 가능한 가볍고 변경이 가능한 코드 기반을 만든다는 목적을 가진 모든 프로그래머의 일상 작업의 일부가 되어야 한다.

변경 제어

하지만 변경을 제어하지 않으면 프로젝트는 쉽게 혼란에 빠질 수 있다. 변경이 제어되지 않은 상태로 소프트웨어에 들어가게 해서는 안 된다. 요구사항과 흡사하게 변경은 수집하고, 분석하고, 문서화하고, 추적하고, 조직해야 한다. 변경 요청이 비교적 세부적이지 못하고 변경에 대한 동기를 더 많이 강조하는 경우가 있기는 하지만 변경 요청은 요구사항과 비슷한 요소를 많이 갖고 있다. 변경 요청의 주요 목적은 평가에 대한 기준에 대한 것이다. 프로젝트에 가장 긍정적인 영향을 주는 것으로 생각되는 변화에 빠르고 효율적으로 중점을 두기 위하여 대개 빠르고 선별 작업 같은 평가를 하기를 원한다("건네주고", "기다리고", "처리한다").

게임 개발자는 변화의 끝없는 원천에 직면하고 각 반복이 시작될 때 처리해야 할 "최상의" 변화를 매우 주의 깊게 선택해야 한다. 그렇지 않다면 곧 예산이 초과될 것이며 제품에 추가되는 것은 거의 없을 것이다. 변화를 받아들일 수 있다는 것은 가볍게 이동해야 한다는 것, 즉 가능한 민첩해야 한다는 것을 의미한다. 게임에 대한 거대한 추론적인 디자인 문서를 갖거나 완전한 선행 소프트웨어 디자인 청사진을 갖는다면 분명히 너무 거추장스럽게 될 것이며 게임을 발전시킬 중대한 변화를 일으키는 데 실패할 것이다.

게임 개발자들은 이동하는 목표를 노리기 때문에 "재미있는" 게임은 전통적인 소프트웨어 제품보다 더 급격한 방식으로 변하는 경향이 있다. 따라서 변경 사항, 변경에 대한 이유, 그리고 변경의 결과를 추적하는 것이 매우 중요하다.

이 변경 및 이러한 변경의 각 결과의 데이터베이스는 프로젝트와 프로젝트 사이에서 언급할 흥미로운 것이 될 수도 있다. 본질적으로 이것은 게임 디자인의 해야 할 일과 하지 말아야 할 것의 데이터베이스가 될 것이다. 이와 관련하여 변경 요청에서 시작해서 요구사항을 통하여 소프트웨어의 작동하는 카피에 이르기까지 추적할 수 있다는 것이 매우 중요해진다. 게임의 특정 빌드에서 어떤 기능이 구현되는지를 알기만 하면 되고, 그렇지 않다면 당신이 만드는 변화의 영향을 정확하게 평가하는 것이 매우 어려워질 것이다.

지역적 최적 상태

진정으로 반복적인 프로젝트에서 소프트웨어의 자연스러운 상태는 유지 보수가 되는 것이며 어떤 "개발 중" 상태에서 최종 사용자로부터 감추어지지 않는 것이다. 요구사항의 추가와

변경은 끊임없이 유입되며 게임은 천천히, 하지만 확실하게 자신의 최적의 목적을 향하여 흔들리며 나아간다. 이 모든 변화에서 게임 비젼을 명심하는 것 또한 중요한 일이다.

일부 변경에서는 처음에는 자신의 앞선 변경보다 낮게 평가되는 게임이 나올 수 있다. 큰 도전과제는 이러한 변경이 진정으로 유리한 것인지 여부 및 유리한 시기를 아는 것 또는 변경을 도입하는 것이 실제로는 잘못된 것이었는지를 아는 것이다. 지역적인 최적 시나리오에 고착되지 않기 위해서는 용기를 내는 동시에 겸손해야 하며 좋은 추적성과 도구 지원을 가져야 한다.

자신의 이력을 모른다면 반복해야 한다

우리는 인간으로서 매우 빨리 잊는 경향이 있으며 구현된 요구사항의 치환은 빠른 속도로 매우 커진다. 이 때문에 프로젝트는 어떤 반복만큼 떨어져 있는 상태에서 거의 똑같은 우울한 결과를 낳으면서 동일한 변경 사항과 요구 사항을 몇 번이고 시도 및 재시도하는 상황에 처하게 된다. 즉, 앞으로 전진하는 대신에 원을 그리며 도는 게임 디자인을 갖게 된다.

따라서 특정 변경 뒤에는 추론을 위한 변경 백로그를 갖는 것이 매우 중요하다. 이 백로그는 어떤 변경 사항을 지원하고 어떤 변경 사항을 버려야 할 때 매우 귀중한 정보를 제공한다. 무엇인가가 시도되었는지 여부와 어떤 결과에서 시도되었는지를 알 수 있다는 것은 매우 강력하지만 간과되는 경우가 많다. 특정 변경이 게임의 일부였던 정확한 빌드에 대한 접근을 갖고 이 특정 버전을 다시 플레이할 수 있다는 것도 귀중한 정보를 제공한다.

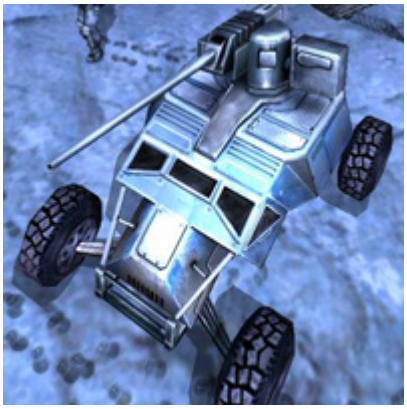
품질 검증

문제를 빨리 찾을수록 좋다. 따라서 품질을 가능한 빨리 검증하는 것이 매우 중요하다. 소프트웨어는 전체 프로젝트에 걸쳐서 기능 및 품질 문제를 확인하기 위한 시험을 해야 한다. 개별적인 시험 단계로서 품질을 추가할 수는 없다.

품질은 매일 프로젝트에 관여하는 모든 사람에 대한 주요 관심사가 되어야 한다. 최상 품질의 소프트웨어를 제공한다는 것도 팀의 동기 제공에 주요하게 기여하는 것이며, 품질이 결여되면 조직에서 변화하려는 헌신과 용기가 빠져나갈 것이 분명하다.

게임을 시험하는 것은 특히 겁나는 작업이다. 서로 다른 하드웨어, 운영 체제, 그리고 API 버전의 모든 가능한 조합에 대한 기능 시험을 고려해 보자! 여기에 시험 주제에 대한 감정적 반응을 측정하는 것과 같이 게임에 고유한 특히 파악하기 어려운 품질 요구사항과 시험 또는 품질 보장 활동이 더 높다고 간주되는 위치로 향하는 경력에 있어서의 디딤돌로 간주된다는 사실(즉, 이것은 단계가 낮으며 그다지 중요하지 않은 작업이다)을 더한다.

하지만 재미는 있는가?



게임 개발의 가장 큰 위험은 수익을 낼 만큼 재미있지 않은 게임을 부주의하게 개발할 가능성이다. 게임 디자이너들은 게임 디자인 문서에서 게임의 흥행 요소를 설명하려고 노력하지만 "재미"라는 것은 게임의 돌발적인 요소라서 실제로 게임이나 게임의 불완전한 버전을 실제로 플레이하는 방법으로만 검증할 수 있다.

많은 화려한 콘텐츠와 특별한 기능을 이용하지 않고도 게임이 재미있다고 보여줄 수 있다면 실제로 배치된 후에는 훨씬 더 많은 재미와 보상을 줄 것이 틀림없다. 다른 한편으로 핵심 게임플레이가 지루하거나 좌절감을 준다면 어떠한 추가적인 화려한 콘텐츠나 기능도 게임을 구할 수 없다. 게임 개발자들은 자신들의 게임이 재미있는지를 가능한 빨리 알아내야 한다.

게임의 "재미"를 검증하는 주요 수단의 측면에서 볼 때 시험은 전문적으로 주의 깊게 처리해야 하는 것이다. 시험은 개발에서 이토록 중요한 부분이기 때문에 모든 개발자가 시험에 관여해야 하며 품질 보장 활동은 소규모 전문 시험 팀의 지도에 따라야 한다.

시험 팀은 테스트 품질을 모니터링하고 여러 가지 시험 기법을 적용하는 방법에 관하여 개발자들에게 조언을 할 책임을 가진다. 이것은 품질 보장을 모든 개발자에 대한 더욱 통합된 일상 작업으로 만들 것이다. 또한 QA 부서에 사람을 고용할 때는 사람들이 QA 업무를 다른 어떤 경력을 위한 디딤돌로 보고 있다는 것을 알고 있어야 한다.

소프트웨어가 변하고 발전함에 따라 시험 절차도 이를 따라잡아야 한다. 시험은 모든 반복의 일부이며 프로젝트에서 모든 개발자에 대한 일상 작업이라는 것을 기억하는 것이 중요하다. 소프트웨어를 따라잡기 위하여 수 많은 시험 문서를 업데이트하는 것은 큰 일이 될 수 있으며 이러한 시험을 반복해서 수행하는 것은 수작업으로 할 경우 많은 시간을 소모하게 된다.

여기에는 자동화된 시험 지원 및 추적을 위한 훌륭한 도구가 필요하며 궁극적으로 전체 소프트웨어를 시험하기 위해 단추를 누르기만 하면 되어야 한다. 이것은 게임의 실제 구현이 시험에 친화적이어야 하며 시험 가능성은 그 자체로 품질 요구사항이 될 수 있다는 것을 의미한다. 이것의 범위는 단위 시험 프레임워크 및 시험 친화적인 소프트웨어 아키텍처를 사용하는 것에서부터 사용자 시나리오의 자동 생성을 가능하게 하기 위하여 사용자 입력의 기록 및 재생을 가능하게 하는 것에 이를 수 있다.

소프트웨어의 품질을 평가하는 방법은 시험이 유일한 것이 아니다. 코드 품질 검사, 워크스루, 그리고 그 밖의 선제적인 기법들도 이행해야 하며 조직에 맞도록 개량해야 한다. 이러한 기법들은 프로젝트 팀에서 지식을 전파하는 경우에도 좋다.

시험 및 요구사항

요구사항들이 실행될 여러 가지 테스트 케이스의 근본을 이루며 요구사항을 작성할 때에는 테스트 케이스를 병행하여 작성해야 한다. 이것은 요구사항의 품질을 높이는 한편 시험 활동을 개발 과정에서 더욱 통합된 부분으로 만들 것이다. 또한 시험과 요구사항들 사이의 정보를 중복하지 않도록 주의를 기울여야 한다.

게임은 개방형인 경우가 많으며 입력 옵션의 양은 다소 끝이 없다. 요구사항에만 기준을 두고 좋은 테스트 케이스를 찾는 것은 무척 어려운 일이며 탐험적 시험과 같은 기법들도 사용하여 고려해야 할 테스트 케이스와 중요한 입력 자료를 찾아야 한다.

결론

소프트웨어 엔지니어링의 모범 관행은 성공적으로 전달하는 소프트웨어 제품에 대하여 관찰된 접근법들이다. 따라서 게임 개발을 위한 소프트웨어 프로세스는 일반적인 소프트웨어 엔지니어링을 기반으로 가져야 하며, 그 다음에는 이 기반을 특별히 게임 개발 커뮤니티를 겨냥한 개발 프로세스에 맞게 고쳐야 한다. 이 프로세스는 기업의 수준과 심지어는 프로젝트 사이의 수준에서도 추가적인 개량을 지원해야 한다.

참고 자료

Alan MacCormack "Product-Development Practices That Work: How Internet Companies Build Software", MIT Sloan ManagementReview, Winter 2001, Volume 42, Number 2

Raph Koster, "A Theory of Fun for Game Design", Paraglyph Press, ISBN: 1932111972, www.theoryoffun.com

Gamma et al. "Design Patterns Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN: 0-201-63361-2

Tom Hammersley, "Planning For Fun In Game Programming",www.gamasutra.com/view/feature/4031/planning_for_fun_in_game.php

Extreme Programming: A gentle introduction. www.extremeprogramming.org

Manifesto for Agile Software Development, www.agilemanifesto.org

Martin Fowler, www.martinfowler.com

Yoopeedoo, www.yoopeedoo.org

UML Resource Page, www.uml.org
