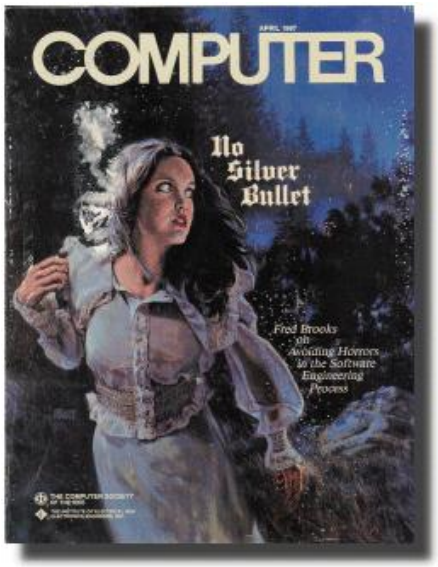
Domain-Driven Design Essence

Eternity(http://aeternum.egloos.com/)

은 총알은 없다



1986 년 소프트웨어 공학의 선구자인 Frederick Brooks 는 그의 기념비적인 논문 "은 총알은 없다 No Silver Bullet"에서 소프트웨어가 태생적으로 안고 있는 본질적인 문제(essence)로 인해 앞으로 10 년 내에 생산성, 신뢰성, 단순성을 단 열 배라도 향상시켜줄 기술이나 기법은 출현하지 않을 것이라고 예견했다. Frederick Brooks 는 소프트웨어와 관련된 본질적인 어려움을 복잡성, 순응성, 변경 가능성, 비가시성으로 보고, 이러한 본질적인 문제가 해결되지 않는 한 소프트웨어를 프로그래밍 언어로 표현하기 위한 고수준 언어, 통합 환경과 같은 부차적인 작업(accidents)에 많은 노력을 쏟는 것만으로는 소프트웨어 개발에 있어 비약적인 생산성 향상을 기대할 수는 없다고 주장했다. 그리고 20 년 이상이 지난 현재까지도 이성을 잃고 폭주하는 늑대 인간을 잠재울만한 은 총알이 발명될 확률은 극히 낮아 보인다.

정보 은닉과 소프트웨어 아키텍처의 기틀을 닦은 David L. Parnas 역시 1985 년 발표한 논문에서 이와 유사한 주장을 펼치고 있다.

> 흔히 새 언어나 새 도구가 소프트웨어 생산성을 극적으로 향상시키리라는 예측을 들어보았을 것이다. 이에 대해 Parnas 는 아니라고 말한다. "새 프로그래밍 언어가 극적인 향상을 가져오리라 기대해서는 안되며", "프로그래밍 환경 문제가 우리 분야의 진짜 장애물은 아니다."
>
> \- Robert L. Glass, 소프트웨어 컨플릭트 2.0

그 동안 은 총알은 아니더라도 납 총알 정도는 되기를 희망하는 다양한 기법과 도구들이 소프트웨어 개발 현장으로 쏟아져 들어 왔다. 그 중 일부는 생산성 향상에 어느 정도 기여하기도 했지만 대부분은 과대선전의 불명예를 안은 채 쓸쓸히 무대의 뒤편으로 퇴장할 수 밖에 없었다. 실패한 대부분의 기법이 안고 있는 가장 큰 문제점은 소프트웨어의 본질적인 문제가 아닌 부차적인 문제를 해결하려고 했다는 점이다.

최근 소프트웨어의 본질적인 문제를 해결하기 위한 방법으로 Domain-Driven Design – 이후로는 DDD 로 표기한다 - 가 주목을 받고 있다. DDD 는 Eric Evans 가 2003 년에 출간한 책의

제목이지만 현재는 특정한 소프트웨어 개발 방법 및 기법을 의미하는 고유명사처럼 사용되고 있다.

DDD 의 슬로건은 단순하다. 유용한 소프트웨어를 개발하고 싶다면 도메인에 귀를 기울여라. 도메인에 대한 모델을 만들고 이에 대한 이해를 모든 이해관계자들 간에 공유하라. DDD 에 처음 입문하는 대부분의 사람들은 DDD 를 ENTITY, VALUE OBJECT, AGGREGATE 와 같은 패턴 집합으로 바라보거나 Spring ROO 와 같은 프레임워크 레벨에서 지원하는 구현 기법으로 인식한다. 그러나 이것은 DDD 에 대한 단편적인 시각일 뿐 전반적인 개념과는 거리가 멀다. DDD 의 핵심은 그것이 소프트웨어 개발과 관련된 본질적인 측면으로 우리들을 인도한다는 것이다. DDD 는 소프트웨어의 본질적인 문제를 해결하기 위한 분석/설계 기법일 뿐만 아니라 구성원 간의 사회학적인 관계 또한 포괄하고 있다.

DDD 가 납 총알 정도는 될 수 있을까? 글쎄, 그에 대한 대답을 하기 전에 우선 DDD 가 출현하게 된 배경을 알아보기 위해 소프트웨어 개발 분야가 걸어온 험난한 여정을 뒤돌아 보자.

분석/설계 미신

소프트웨어 개발 분야는 다른 성숙한 분야에 비해 상대적으로 그 역사가 짧기 때문에 다양한 분야로부터 메타포를 받아 들였다. 가장 대표적인 것이 제조 manufacturing 와 건축 architecture 이며 두 분야는 소프트웨어 개발에 필요한 어휘와 개념을 제공하기도 했지만 개발 프로세스와 사람이라는 관점에서 부정적인 영향을 끼치기도 했다.

> 제조 manufacturing 는 소프트웨어 개발에 대한 메타포로 자주 사용되곤 한다. 이 메타포로부터 얻게 되는 한 가지 결론은 고도로 숙련된 엔지니어는 설계를, 덜 숙련된 노동자는 제품을 조립한다는 것이다. 이 은유는 소프트웨어 개발은 모든 것이 설계라는 한 가지 단순한 이유로 많은 프로젝트를 엉망으로 만들어 왔다.
>
> \- Eric Evans, Domain-Driven Design

제조 메타포에서 숙련된 엔지니어는 UML 과 같은 표기법을 사용해서 구현에 필요한 설계 도면을 작성하고 덜 숙련된 노동자는 기계적으로 다이어그램에 명시된 요소를 프로그램 명령문으로 변환한다. 프로그래밍이란 설계 문서를 프로그램으로 변환시키는 것이므로 이 과정에서 오류를 줄이기 위해 자동으로 코드를 생성해주는 기법을 적용시키는 것이 효과적이다. 과연 그럴까?

제조 메타포를 기반으로 한 역할 분리는 근본적으로 두 가지 오류를 범하고 있다. 첫 번째 오류는 설계자와 구현자 간의 차이를 고려하지 않고 기계적으로 설계를 구현으로 변환할 수 있다는 믿음을 전제로 한다는 점이다. 유사한 경험과 경력을 가진 사람들 간에도 커뮤니케이션이 불완전해지는 경우를 자주 보게 된다. 하물며 서로 다른 역할과 경험을 기반으로 하는 두 계층 간에 다이어그램과 문서 중심의 커뮤니케이션이 가능하다는 근거 없는 믿음은 어디에서 기인한 것일까?

바로 여기에 문제가 있다. 만약 설계자가 코더 보다 높은 수준의 기본단위를 사용한다면, 결과로 나오는 설계는 코더가 작업을 시작하는데 부적절할 것이다. 따라서 코더는 코딩을 하기 전에 적절한 수준의 설계를 추가하기 위해 시간을 소모해야 할 것이다. 이런 경우 코더는 설계자가 기대한 완전한 설계 솔루션과는 다르게 작업할 수 있으므로, 설계에서 코딩으로의 전환은 최상의 경우라도 불편할 테고, 보통은 많은 문제를 야기할 것이다.
반대의 경우도 동일한 문제가 발생한다. 만약 설계자의 경험이 부족하다면 설계자는 매우 상세한 수준의 설계를 만들 것이다. 그러나 이 설계자보다 경험이 많은 코더는 지나치게 상세한 수준의 설계를 받아들이지 않고 자신의 설계 아이디어로 대체할 것이다. 이는 설계자가 코더보다 똑똑하거나 숙련된 사람이어야 하는가에 대한 문제가 아니라, 그들이 같은 경험과 기본 단위를 가지고 있는가에 대한 문제이다.

- Robert L. Glass, 소프트웨어 공학의 사실과 오해

작업이 더 작은 단계로 세분화될수록, 한 사람에서 또 다른 사람으로 정보를 전달하는 데에 더 많은 시간이 걸린다. 생산라인 접근 방식은 수작업 노동에는 잘 맞을 수 있다. 그러나 지적인 작업에는 형편없이 실패한다.
소프트웨어 개발은 팀원들의 머리에서 일어난다. 사람들을 특정 활동에 전문화시킴으로써 간단한 프로젝트의 딜리버리도 여러 경로를 통해야 한다. 각 경로는 실수와 결함에의 잠재성을 갖고 있는 값비싼 과정이다.

- Pete McBreen, 소프트웨어 장인 정신

제조 메타포의 두 번째 오류는 구현 전에 설계를 완성시키는 것이 가능하다고 가정하는 점이다. 그러나 실제로는 개략적인 설계를 구현으로 옮기는 도중에 전체 구조에 대한 가장 중요한 통찰을 얻게 된다. 구현 전에 대략적인 설계를 동결시키는 방식은 구현으로부터 얻게 되는 피드백을 원천적으로 차단하는 역효과를 가져온다. 특히 CASE 툴을 사용해서 설계 문서를 다이어그램으로 표현하고 이를 별도의 산출물로 유지할 경우 문제가 더욱 커지게 되는데, 구현 변경이 곧 설계 문서의 변경을 의미하므로 이를 기피하게 되기 때문이다. 따라서 UML 과 같은 표기법을 사용해서 설계를 동결시키고 이를 구현자에게 전달해서 프로그램을 구현하도록 하거나 코드를 생성하는 아이디어는 소프트웨어 개발에는 적합하지 않다.

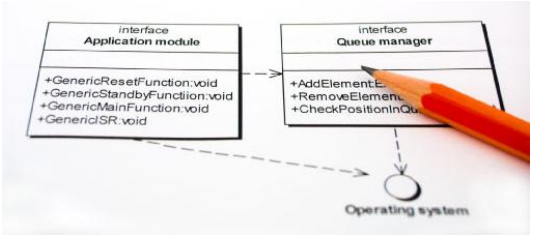


그림 1 구현과 분리된 설계는 의미가 없다

우리는 분석 모델, 설계 모델, 구현 모델이 서로 달라야 한다는 오랜 미신 속에 살아 왔다. 이론적으로 분석 모델은 해결 방법에 대한 언급 없이 문제 도메인을 설명하는 모델이다. 분석 모델은 순수하게 문제 도메인에 초점을 맞추어야 하며 기술적인 해결 방법을 언급해서는 안된다.

분석 모델이 완성되면 이를 바탕으로 기술적인 관점에서 솔루션을 서술하는 설계 모델이 만들어진다. 프로그래머는 이렇게 만들어진 청사진을 프로그래밍 언어를 사용하여 컴퓨터가 이해할 수 있는 명령어로 변환한다. 그러나 분석 모델, 설계 모델, 구현 모델을 명확하게 구분하는 것은 가능하지도 않을 뿐만 아니라 오히려 소프트웨어의 품질에 악영향을 미친다. 진실로 우리가 원하는 것은 소프트웨어의 영혼에 영광을 안겨줄 분석과 설계와 구현의 삼위일체론이다.

> 필자의 개인적인 견해로는 분석 모델이 사용할 구현 기술을 지향하는 것은 매우 합리적이며 그 결과 모델은 작업해야할 문제를 이해하는데 더 유용해 진다는 것이다. 분석과 설계의 근본적인 차이는 분석이란 도메인을 이해하는 것인 반면, 설계는 도메인을 지원하는 소프트웨어를 이해하는 것이라는 점이다. 명확하게 이 두 가지는 밀접하게 연결되어 있으며 둘 사이의 경계가 매우 모호해질 때가 많다. 그러나 경계가 뚜렷할 필요는 없다. 유용성보다 순수성을 강조해서는 안 된다. 이론적으로는 분석인 동시에 설계의 특성을 가지는 하이브리드 모델이 절대로 좋은 것이 아니지만 이런 하이브리드한 특징이 가장 좋은 모델을 만든다고 믿는다.
>
> - Martin Fowler, Is There Such a Thing as Object-Oriented Analysis?

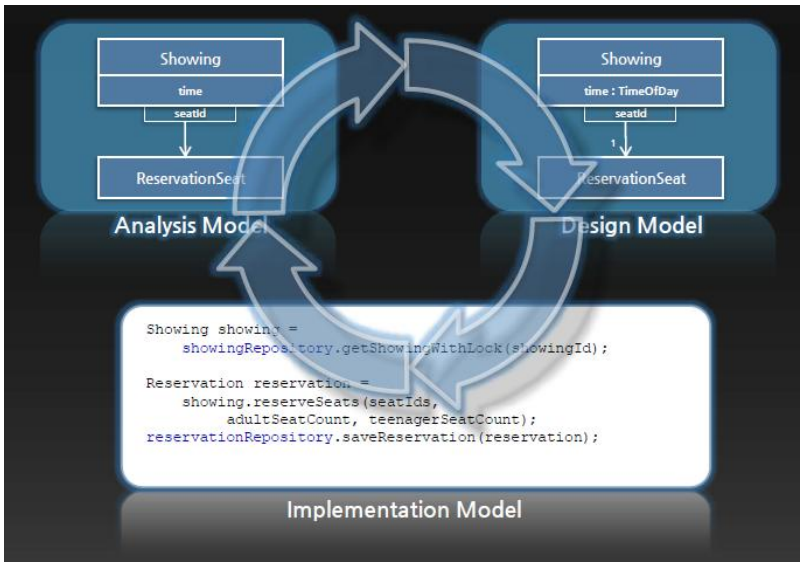


그림 2 분석/설계/구현의 삼위일체

분석 모델과 설계 모델의 하이브리드한 특징이 가장 좋은 모델을 낳는 토양이라면 프로그래밍 언어를 사용하여 구현한 코드가 이 모델을 최대한 반영하는 것이 가장 이상적일 것이다. 만약 설계 모델의 일부가 적용 기술 내에서 구현 불가능하다면 설계 모델을 변경해야 한다. 프로그래밍 과정 동안 설계의 실현 가능성과 정확성이 검증되고 테스트되며 그 결과 잘못된 설계가 수정되거나 새로운 설계로 대체된다. 따라서 프로그래밍은 설계의 한 과정이며 설계는 프로그래밍을 통해 개선된다.

많은 방법론에서 분석, 설계, 구현 단계를 세분화하는 이유는 대부분의 사람들이 그것을 프로그래밍 세계의 이데아라고 생각하기 때문이다. 그러나 실제 소프트웨어 설계 과정은 아름답지도, 깔끔하지도, 올곧지도 않다. 소프트웨어 설계는 지저분하고 변덕스러우며 지우고 새로 그리는 과정을 무한히 반복해야 하는 '스케치' 작업이다. 프로그래밍은 설계를 명령어로 옮기는 작업이 아니라 설계 그 자체를 만들기 위한 과정이다.

> 초등학교 시절에 선생님이 가르쳐준 대로 연필을 쥐지 못해서 괴로워했던 것처럼, 나는 오랫동안 이런 프로그래밍 방식에 대해서 남몰래 부끄러워했다. 하지만 내가 그 당시에 화가나 건축가 같은 다른 창조자들이 일하는 방식을 알았더라면, 내가 프로그래밍하는 방식을 지칭하는 특별한 이름이 있다는 것을 알 수 있었을 것이다. 그 이름은 바로 '스케치'다. 적어도 내가 보기에 대학 시절에 배운 프로그래밍 방식은 전부 잘못 되었다. 소설가, 화가, 그리고 건축가의 작업이 그런 것처럼 프로그램이란 전체 모습을 미리 알 수 있는 것이 아니라 작성해 나가면서 이해하게 되는 존재다.
>
> 이런 깨달음은 소프트웨어 디자인에 대한 실질적인 의미를 갖는다. 그것은 프로그래밍이라는 것이 부드럽고 말랑말랑한 존재라는 엄연한 사실에 대한 재확인이다. 프로그래밍 언어는 당신이 이미 머릿속으로 생각한 프로그램을 표현하는 도구가 아니라, 아직 존재하지 않는 프로그램을 생각해 내기 위한 도구다.
>
> - Paul Graham, 해커와 화가

분석/설계/구현이 별개의 것이 아닌 단일 행위라는 사상은 DDD 를 지탱하는 두 가지 축 중 하나인 MODLE-DRIVEN DESIGN 을 지지하는 핵심 사상이다. 이러한 분석/설계/구현에 대한 시각의 변화는 새로운 주류의 방법론이 출현할 수 있는 토양을 만들어 주었다. 애자일 진영이 바로 그것이다.

방법론 엑소더스, 그리고 기민함의 시대로

"테러리스트와 방법론자의 차이점은 테러리스트는 협상이 가능하다는 점이다"라는 우스개 소리가 있을 만큼 90 년대는 방법론 전쟁의 시대였다. 무수한 방법론이 소프트웨어 생산성을 향상시켜줄 은 총알임을 자처하며 조직의 곳곳으로 침투해 들어갔다.

방법론이란 소프트웨어를 만들어 내기 위해 행하는 모든 것을 의미하며 우리가 속한 그룹이 동의한 약속이다. 따라서 적절한 방법론을 가지고 있다면 체계적인 절차와 적합한 도구 및 기법을 사용하여 효율적으로 소프트웨어를 개발할 수 있게 된다.

그러나 방법론이 모든 문제를 해결하려고 시도하는 순간 자기 모순에 빠지게 된다. 구체적인 문제를 해결하기 위해서는 구체적인 해법이 필요하다. 일반적인 해법은 특수한 상황이나 예외적인 상황에서는 그 힘을 쉽게 잃고 만다. 결국 대부분의 방법론이 목표로 하는 모든 조직의

모든 문제를 해결할 수 있는 범용적인 방법론은 결국 어떤 문제도 해결할 수 없다는 의미와 동일하다.

> 문제가 무엇인지 명확히 밝히지 못함으로써 많은 프로젝트들이 위기에 봉착하게 된다. 그런데 개발 '방법론'이 발전해감에 따라 더더욱 위기를 맞게 되었다. 왜냐하면 문제에 대해 얘기하지 않았기 때문에, 문제를 분석하지도 문제를 분류하지도 못했다. 모든 개발 문제를 해결해 줄 수 있는 보편적인 개발 방법론이 존재할 수 있으리라는 어리석은 생각에 빠지고 말았다. 방법론을 모든 질병을 치료해줄 만병 통치약으로 착각했던 것이다. 애석하게도 이것은 불가능하다. 다시 말해 방법론이 일반화되면 될수록 그 가치는 더욱 하락한다는 뜻이다. 모든 문제를 해결하기 위한 방법론이 만약 있다면, 역설적이게도 그 방법론은 구체적이고 특수한 문제를 해결하는 데에는 거의 도움을 주지 못할 것이다.
>
> \- Michael Jackson, Software Requirements & Specification

방법론을 적용하는 조직이 저지르는 일반적인 실수는 이를 적용할 개별 팀 단위, 또는 프로젝트 단위의 상황이나 구성원에는 신경을 쓰지 않은 채 일괄적으로 단일 규모의 방법론을 적용하려는데 있다. 각 팀이 담당하고 있는 도메인의 복잡성과 무관하게 동일한 절차, 동일한 산출물, 동일한 아키텍처, 동일한 설계 기법과 동일한 품질 기준을 모든 상황에 적용하려고 하는 시도는 구성원들의 시선을 소프트웨어의 본질적인 문제에서 조직의 규칙을 지키기 위한 형식주의로 돌리게 만든다.

모든 경우에 적용할 수 있는 일반적인 방법론을 만들기 위해서는 방법론의 무게가 점점 더 무거워질 수 밖에 없다. 실패를 겪은 조직은 실패에 대한 두려움으로 인해 방법론을 보완하게 된다. 모든 경우를 포괄하는 방법론이라는 이데아를 좇기 위해 더 많은 제약과 더 많은 산출물을 방법론에 추가하게 되고, 결국 팀은 방법론의 제약과 무게에 눌려 한발자국을 내딛는 것조차 버거워 지게 되었다. 따라서 소프트웨어 개발이라는 본연의 목표보다는 계획과 관리에 초점이 맞추어지면서 적응적인 프로세스보다는 예측적인 프로세스가 우선하는 유연하지 못한 개발 환경으로 변해가고 있었다. '계속되는 악순환'을 끊기 위해 무언가 변화가 필요했다.

방법론이라는 빅 브라더에 짓눌려 있던 개발 커뮤니티는 마침내 2001 년 '애자일 동맹 선언 Manifesto of the Agile Alliance'을 발표한다. '애자일 동맹 선언'의 핵심은 다음의 4 가지 문장으로 압축된다.

- **개인과 상호작용**이 프로세스와 툴보다 우선이다.
- **동작하는 소프트웨어**가 포괄적인 문서보다 우선이다.
- **고객 협력**이 계약 협력보다 우선이다.
- **변화에 대한 반응**이 계획을 따르는 것보다 우선이다.

'애자일 동맹 선언'은 소프트웨어 팀으로 하여금 빠르게 일하고 변화에 반응할 수 있도록 하는 가치와 원칙을 기반으로 한다. 그리고 방법론과 산출물에 밀려 그 동안 잊혀져 있던 존재인 개인과 팀, 그리고 이해 당사자 간의 의사소통을 소프트웨어 개발의 중심으로 되돌려 놓았다.

그렇다면 이러한 변화가 소프트웨어 개발 방식에 미친 영향에는 어떤 것이 있을까? 소프트웨어 개발은 그 과정에서 발생하는 다양한 잡음으로 인해 태생적으로 예측이 불가능한 활동이다. 따라서 명시적인 제어 모델보다는 경험적인 제어 모델을 따르는 것이 효과적이다. 경험적인 제어 모델은 변화를 환영하고 즉각적이고 지속적인 피드백 feedback 을 기반으로 기민하게 계획을 수정하고 변화에 적응한다. 기존의 무겁고 예측적인 방법론은 변화를 적으로 생각하고 모든 것을 고정시킴으로써 피드백으로 인한 개선의 여지를 부정했다. 애자일은 용어 그 자체에서 알 수 있는 것처럼 소프트웨어 개발과 관련된 모든 활동을 기민하게 만들었다.

변화의 수용과 즉각적인 피드백이 애자일의 모토로 자리잡으면서 개발 방법 역시 많은 변화를 겪게 되었다. 통합 시 실패에 대한 피드백을 받을 수 있는 방법은 무엇인가? 지속적인 통합 Continuous Integration 을 적용하라. 고객에게 원하는 가치를 제공하고 있는지에 대한 피드백을 받을 수 있는 방법은 무엇인가? 최대한 빨리 배포하고 반복 iteration 주기를 통해 주기적으로 배포하라.

이제 원래의 주제로 돌아 가자. 설계자와 구현자 간의 분할이 의미가 없고 설계와 구현이 동일한 활동이라면 결국 이것은 짜고 고치기 Code-and-Fix 방법으로의 회귀를 의미하는 것은 아닌가? 그렇지는 않다. 오히려 애자일의 권고는 '설계하지 말라'가 아니라 '언제나 설계하라'다. 피드백 주기가 길어지는 사전 설계 Big Up-front Design 대신 피드백 주기가 짧은 TDDTest-Driven Development 를 적용하라. 그리고 리팩토링 Refactoring 을 통해 설계와 코드를 지속적으로 개선함으로써 코드를 항상 최상의 상태로 유지하라. 점진적인 설계 사이클을 도입함으로써 설계와 구현을 독립적인 활동이 아닌 상호 영향을 주고 받는 유기적인 절차로 바라볼 수 있게 되었다. 따라서 경험을 기반으로 한 지속적인 리팩토링을 통해 최적의 소프트웨어의 설계를 얻을 수 있다.

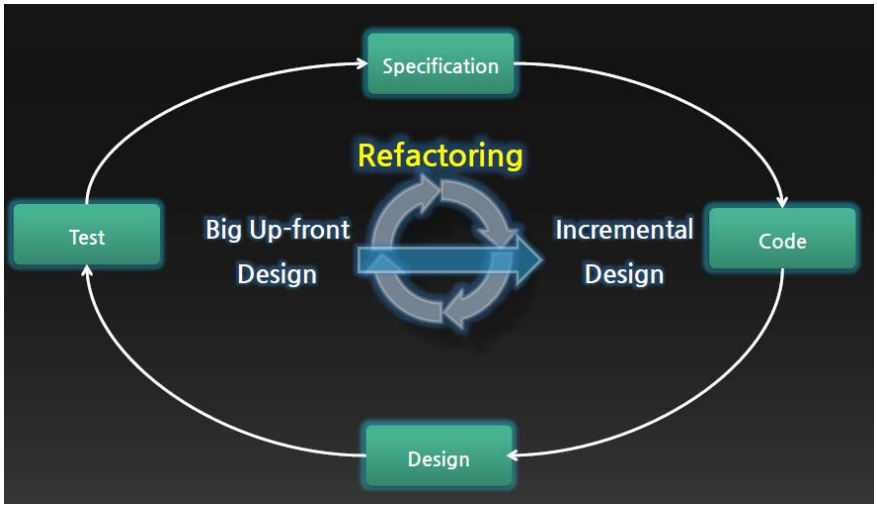


그림 3 TDD 사이클을 통한 구현과 설계의 통합

> 구현하기 전 설계하기의 대안은 구현한 후 설계 하기다. 어느 정도의 초기 설계가 필요하긴 하지만, 최초 구현을 시작할 수 있는 정도만 설계하면 된다. 그 이상의 설계는 구현이 자리를 잡고 설계의 진짜 제약들이 분명하게 보일 때 하도록 한다. '아무것도 설계하지 말라'와 정반대로, XP 의 전략은 '언제나 설계하라'다.
>
> - Kent Beck, 익스트림 프로그래밍 2nd Edition

애자일에서 강조하는 또 하나의 핵심 주제는 문서를 통한 의사소통을 가능하면 배제하고 이해관계자 간의 직접적인 대면과 대화를 통해 의사소통의 효율성을 촉진시키라는 것이다. 이것은 DDD 를 지탱하는 두 가지 축 중 하나인 UBIQUITOUS LANGUAGE 를 구축하기 위한 핵심 조건이다.

방법론적인 측면에서의 변화는 소프트웨어 개발 활동을 역동적이고 기민하게 만들었다. 좀 더 유연하고 자유롭게 소프트웨어를 개발하고자 하는 사람들의 욕망은 또 다른 방향에서 자신의 어깨를 짓누르고 있던 무거운 개발 기술을 벗어 던지게 만들었다.

기술 엑소더스, 그리고 가벼움의 시대로

객체 지향 프로그래밍을 처음 배웠던 시절 가장 중요한 원칙은 '객체의 속성을 사용하는 메소드는 해당 객체에 두어라'였다. 객체 지향 분석/설계를 처음 배웠던 시절 가장 중요한 원칙은 '문제 도메인에 존재하는 개념과 관계를 찾아 이를 반영하는 소프트웨어를 설계하라'였다. 최소한 위의 두 가지 원칙을 준수함으로써 표현적 차이 Representation Gap - 또는 개념적 차이 Semantic Gap –를 최소화하는 시스템을 개발할 수 있다. 소프트웨어 모델, 즉 코드가 도메인 모델을 떠오르게 하고 도메인 모델을 바탕으로 예측 가능한 방식으로 작동할 경우 두 모델 간의 "표현적 차이가 적다"라고 한다.

시간이 흘러 엔터프라이즈 애플리케이션을 구축하기 위해 새로운 기술을 습득하던 중 기존에 알고 있던 방식과는 다른 방식으로 소프트웨어를 개발하고 있다는 사실을 알게 되었다. 해당 기술을 사용하는 프로젝트의 표준적인 설계 방법은 객체의 속성을 사용하는 메소드를 객체를 사용하는 다른 객체에 위치시키는 것이었다. 도메인의 개념을 표현한 소프트웨어 코드는 더 이상 도메인의 개념적인 모습을 찾아볼 수 없을 정도로 인프라스트럭처 코드로 어지럽혀져 있었다. EJB 의 시대가 도래한 것이다.

1990 년대 말에 등장한 EJBEnterprise Java Beans 는 Java 의 'Write Once, Run Anywhere' 모토를 엔터프라이즈 어플리케이션 환경으로 확장하기 위한 Sun 의 야심작이다. EJB 의 중심 전략은 분산, 트랜잭션, 퍼시스턴스, 보안 등의 인프라스트럭쳐 서비스는 WASWeb Application Server 에서 제공하고 어플리케이션 개발자는 비즈니스 로직에만 집중하도록 하자는 것이다. 개발자들은 Sun 에서

제시하는 장미빛 미래에 열광적으로 환호했고, WAS 라는 프론티어를 향한 벤더들의 골드 러시는 EJB 가 21 세기의 은총알일 것이라는 신기루를 만들어 냈다.

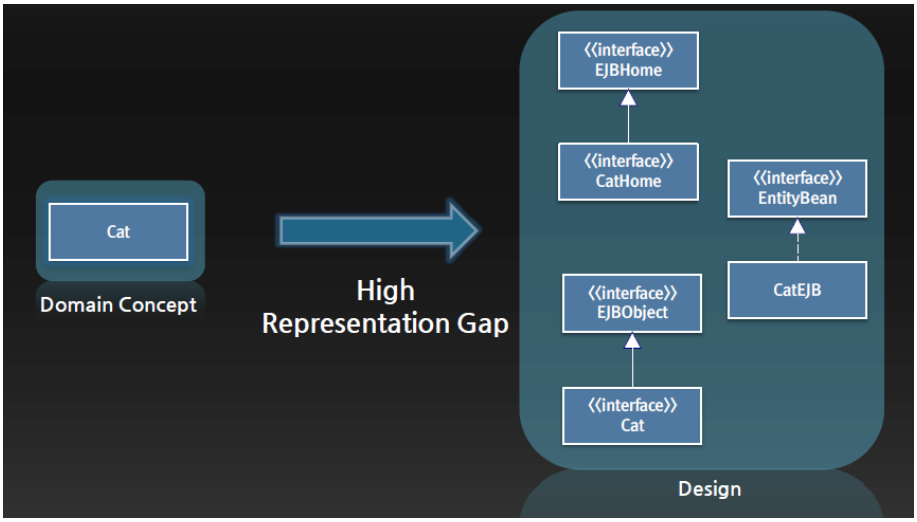


그림 4 표현적 차이가 큰 EJB

그러나 Sun 의 호언장담과 달리 EJB 는 개발자들에게는 악몽과도 같은 존재였다. 인프라스트럭처에 대한 과도한 의존성은 컨테이너에 독립적인 개발 및 테스트를 불가능하게 만들었으며, 컴파일, 빌드, 배포에 의한 피드백의 현저한 감소는 개발자들의 개발 사기를 떨어트리기에 충분했다. 그렇게 몇 년 동안 EJB 라는 빙하기가 Java 진영을 덮쳐 왔다. 순수한 객체 지향 프로그래밍과 분석/설계 방법이 두터운 빙하 속에서 얼어 붙어 가고 있었다.

사람들은 문제가 과도한 인프라스트럭처에 대한 의존성이라는 사실을 깨닫고 EJB 이전의 순수한 객체 시대로 회귀할 것을 꿈꿨다. Martin Fowler 는 원래의 순수한 Java 오브젝트에 POJO[Plain Old Java Object] 라는 명칭을 부여하고 인프라스트럭처에 대한 의존성이 없는 순수한 객체 지향의 시대로 돌아가자고 외쳤다.

그리고 기술의 발전과 함께 무거운 EJB 없이도 유사한 수준의 컨테이너 서비스를 구축할 수 있는 오픈 소스 프레임워크들이 속속 선을 보이게 되었다. 그 중 대표적인 것이 Rod Johnson 의 Spring 프레임워크와 Gavin King 의 Hibernate 였다. 두 프레임워크는 인프라스트럭처에 대한 의존성을 가지지 않는 순수한 객체인 POJO 에 분산, 트랜잭션, 퍼시스턴스, 보안 등과 같은 인프라스트럭처 서비스를 제공할 수 있도록 한다. 즉, 비침투적인 프레임워크를 제공함으로써 기술적인 수준에서 표현적 차이를 줄일 수 있는 기반을 마련했다.

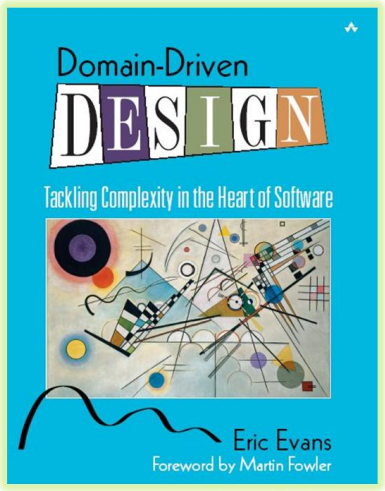
객체-지향 설계가 어떤 특별한 구현 기술(J2EE 나 심지어는 Java)보다 더 중요하다.

- Rod Johnson, Expert One-on-One J2EE Design and Development

비슷한 시점에 Martin Fowler 는 "엔터프라이즈 애플리케이션 아키텍처 패턴"이라는 책을 통해 순수한 객체 지향 원칙에 따라 속성과 메소드를 하나의 객체에 두도록 도메인 레이어를 구축하는 DOMAIN MODE 패턴을 소개했다. EJB 의 컨테이너 지원 서비스를 받기 위해 어쩔 수 없이 도메인 개념과 거리가 먼 설계, 구현 모델을 구축해야 했던 EJB 의 암흑기를 지나 원래의 순수했던 시절의 객체 모델로의 회귀를 선언한 것이다. POJO 와 DOMAIN MODEL 패턴의 목표는 동일하다. 표현적 차이를 줄이는 것이다.

빙하기는 끝났다. 얼음이 녹자 그 밑에서 잠자던 수 많은 생명들이 눈을 뜨고 민첩하게 움직이기 시작했다. 방법론적인 측면과 기술적인 측면 모두에서 어느덧 소프트웨어 진영은 따뜻한 봄의 햇살을 맞으며 과거의 순수했던 시절로 돌아갈 준비를 마친 상태였다. 그리고 사람들의 관심이 서서히 도메인으로 옮겨져 가기 시작한다.

순수의 시대



제조, 건축 메타포로부터 파생된 분석/설계/구현의 명확한 분리가 소프트웨어 개발에 적용하기에는 부적합하다는 깨달음과, 애자일 방법론의 대두로 인한 프로세스와 문서화로부터 개인과 팀, 소프트웨어로의 무게 중심 이동, 리팩토링과 피드백을 통한 점진적인 설계 기법의 적용을 통한 분석/설계/구현 사이클 통합, 그리고 엔터프라이즈 애플리케이션 환경에서 표현적 차이를 줄일 수 있는 POJO 중심의 경량 프레임워크의 대두로 소프트웨어 업계는 기민함과 가벼움의 시대로 접어 들었다. 애자일 동맹은 소프트웨어 개발이 기계적인 작업이 아닌 사람의 창조성과 협업에 의한 작업이라는 사실을 일깨워 주었다. POJO 와 경량 프레임워크는 소프트웨어가 기술이 아닌 도메인에 의해 주도되어야 한다는 사실을 일깨워 주었다. 기술을 향한 골드러시에 동참했던 사람들의 열기가 서서히 식기 시작했다. 사람들은 발전된 기술에 열광하면서도 과거에 소프트웨어를 개발하던 순수한 방식으로 회귀하기를 갈망했다. 그리고 이러한 방법론과 기술에 대한 기대치가 점점 높아져가고 있던 2003 년 Eric Evans 는 "Domain-Driven Design"이라는 책을 출판한다.

"Domain-Driven Design"은 새로운 개념이 아니다. 책의 부제인 "Tackling Complexity in the Heart of Software"에서 알 수 있듯이 DDD 는 소프트웨어의 본질적인 문제인 복잡성을 제어하기 위해 기존에 존재해 왔던 다양한 방법과 기법들을 패턴 언어 Pattern Language 형식으로 정리해 놓은 것이다. DDD 가 전혀 새로운 기법을 제시하지 않고 있음에도 불구하고 많은 사람들이 그것에 열광하는 이유는 DDD 의 접근 방식에 있다. DDD 는 소프트웨어 개발의 베스트 프랙티스를 도메인이라는 컨텍스트 안에서 체계적으로 조합하고 연결시킨다. DDD 는 "모든 소프트웨어 복잡성은 도메인에 기인한다"는 매우 일반적이고 직관적인 명제로부터 출발한다.

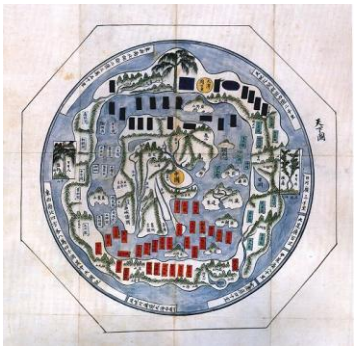
애자일 연맹이 소프트웨어를 개발하는 본질이 사람이라는 사실을, POJO 가 소프트웨어 구성 단위의 본질이 순수한 객체라는 사실을 묻혀있던 과거의 기억 속에서 끄집어 냈다면, DDD 는

우리가 기술의 홍수 속에서 잊고 있었던 소프트웨어가 딛고 있는 땅의 본질이 도메인이라는 사실을 깨닫게 해준다. DDD 의 가치는 과거로부터 중요하고 본질적이라고 믿어 왔던 가치들을 현재의 복잡한 엔터프라이즈 애플리케이션 환경에 적용 가능한 수준으로 확장했다는 점에 있다. 즉, 눈부시게 빠른 속도로 발전하는 기술의 변화에 유연하게 대처하면서도 소프트웨어의 본질인 도메인의 표현을 가능하게 해주는 패턴과 기법의 복합체가 바로 DDD 인 것이다.

DDD 는 도메인 모델과 소프트웨어 모델, 즉 코드 간의 표현적 차이를 최소화하기 위한 접근 방법이다. 이를 위해 EJB 와 같은 구현 기술이 소프트웨어 개발을 주도함으로써 발생되는 여러 가지 문제점을 해결하기 위해 기술 주도적인 방식이 아닌 도메인 주도적인 방식으로 소프트웨어를 개발할 것을 주장한다.

DDD 를 성공적으로 적용하기 위해서는 기본적으로 두 가지 요소가 갖추어져야 한다. MODEL-DRIVEN DESIGN 과 UBIQUITOUS LANGUAGE 가 바로 그것이다. 두 가지 원리를 이해하기 위해 모델과 소프트웨어의 관계에 관해 간략히 살펴 보기로 하자.

MODEL 과 DOMAIN-DRIVEN DESIGN



모델 Model 은 대상을 단순화한 것이다. 모델은 추상화다. 즉, 모델은 얽히고 설킨 복잡한 사실에 대한 하나의 해석이며, 당면한 문제를 해결하기 위해 필요한 측면을 강조하고 문제 해결에 무관하거나 불필요한 세부사항에는 의도적으로 주의를 기울이지 않는다. 사실과 완전히 동일한 모델을 만드는 것은 바람직하지 않다. 모델은 현실이라는 기반 위에 해결하고자 하는 문제에 적합한 새로운 추상화 계층을 창조하는 과정이다. '천하도'는 과거 사람들이 생각하던 세계의 모델이다. 천하도가 세계의 모습과 완전히 동일하지 않다고 해도 과거 사람들의 세계관을 적절히 반영하고 있기 때문에 적합한 모델이라고 할 수 있다.

모든 소프트웨어는 특정한 사용자의 문제를 해결하는 것을 목적으로 한다. 이 때 소프트웨어를 사용할 사용자의 활동이나 관심사의 대상이 되는 영역을 소프트웨어의 도메인 Domain 이라고 한다. 따라서 도메인 모델 Domain Model 이란 소프트웨어가 문제를 해결해야 하는 대상 영역을 단순화시키고 추상화시킨 것이다. 적절한 도메인 모델을 선택함으로써 소프트웨어 개발과 관련된 다양하고 복잡한 측면들을 이해할 수 있고 해결하려는 문제에 집중할 수 있다.

도메인 모델은 어떤 특정한 다이어그램이나 문서가 아니라 이를 통해 전달하고자 하는 개념의 집합이다. 도메인 모델은 단지 도메인 전문가의 머리 속에만 존재하는 지식이 아니라 소프트웨어 개발에 적합하도록 지식을 정밀하게 조직하고 선택적으로 추상화한 것이다.

소프트웨어의 본질적인 복잡성은 도메인의 복잡성에 기인한다. Frederick Brooks 의 말을 재인용하면 소프트웨어의 본질적인 문제를 외면한 상태에서 비약적인 생산성의 향상을 기대할

수는 없다. 따라서 소프트웨어의 본질적인 문제를 해결하기 위해서는 도메인을 이해하고 끊임없는 탐구를 통해 적절한 도메인 모델을 선택하는 것이 중요하다.

그림 5 도메인 모델을 중심으로 소프트웨어 개발을 진행하라

훌륭한 도메인 모델은 다음의 3 가지 요구사항을 충족시킨다.

- 모델과 핵심 설계는 상호 영향을 주고 받으며 구체화된다.

  모델과 구현 간의 긴밀한 연결과 피드백을 통해 모델이 의미를 가지도록 하고 최종 산출물인 동작하는 프로그램이 사용자의 요구사항을 만족시킬 수 있도록 보장한다. 모델과 구현을 연결 시키는 것은 분석/설계/구현을 동일한 사이클로 묶음으로써 달성된다. 애자일의 언제나 설계하기 사상, 즉 리팩토링을 통한 점진적인 설계는 모델과 구현의 불일치를 예방하기 위한 프로세스적인 장치다. 모델과 구현이 일치하기 위해서는 모델과 코드의 표현적 차이가 적어야 한다. 코드가 인프라스트럭처에 대해 강하게 의존할 경우 모델과 코드 간의 표현적 차이가 커진다. POJO 기반의 경량 프레임워크 기술이 인프라스트럭처의 늪에서 코드를 구원한다. 모델을 기반으로 분석/설계/구현을 통합하고 피드백을 통해 개선시키는 과정을 MODEL-DRIVEN DESIGN 이라고 한다.

- 모델은 모든 팀 구성원들이 사용하는 언어의 근간을 이룬다.

  모델과 구현이 긴밀하게 연관되어 있으므로 개발자들은 모델을 사용해서 대화할 수 있다. 모델은 도메인 전문가가 도메인을 바라보는 관점을 담고 있기 때문에 도메인 전문가 간의 대화 시에 사용할 수 있다. 개발자와 도메인 전문가 간에 공통의 모델을 공유하기 때문에 별도의 번역 과정 없이 개발자와 도메인 전문가 사이에에 모델을 기반으로 의사소통 할 수 있다. 애자일 동맹에서 강조하는 문서에 의한 지식 전달이 아닌 직접 대면과 대화를 통한 의사소통 방식은 언어의 중요성을 강조한다. 모델은 소프트웨어 지식의 집합체이며 정보 흐름의 통로다. 이해관계자들 간의 의사소통에 사용되는 언어에 변화가 생기면 모델의 용어가 변경되고, 모델의 용어가 변경되면 코드가 수정된다. 반대로 코드가 변경될 경우

모델과 의사소통에 사용되는 용어가 변경된다. 모델은 이해관계자들의 공통 언어인 UBIQUITOUS LANGUAGE 를 구성하는 기반이다.

- 모델은 불순물을 걸러낸 핵심 지식만을 포함한다.

  효과적인 도메인 모델은 끊임없는 지식 탐구 활동과 개선을 통해 얻어진다. 수많은 모델을 시도하고, 버리고, 변형한 끝에 도메인의 모든 세부 사항에 적절한 일련의 추상적 개념들을 발견함으로써 적절한 도메인 모델을 얻게 된다. 지식 탐구 활동은 개발자와 도메인 전문가 간의 긴밀한 협력이 전제되어야 한다. UBIQUITOUS LANGUAGE 가 빛을 발휘하는 때다. 지식 탐구 활동은 끊임없이 모델과 코드를 개선하는 정제 과정이다. 모델과 코드를 긴밀히 연결함으로써 새로운 통찰이 모델과 코드에 반영되도록 할 수 있다. MODEL-DRIVEN DESIGN 이 우리를 정제의 길로 인도한다. 끊임없는 정제가 가능하기 위해서는 소스 코드를 항상 깨끗하고 안정적인 상태로 유지해야 한다. 애자일의 언제나 설계하기 사상과 분석/설계/구현 사이클의 통합, 리팩토링을 통한 점진적인 설계 방식은 이를 가능하게 한다.

DDD 는 훌륭한 도메인 모델을 만들고 이를 소프트웨어 반영하기 위한 모든 것이다.

MODEL-DRIVEN DESIGN

MODEL-DRIVEN DESIGN 의 개념은 간단하다. 도메인 모델을 반영하는 코드를 작성하라. 코드에 나타나는 구조와 용어는 도메인 모델에 기반해야 한다. 도메인에 대한 이해가 바뀐다면 이를 도메인 모델과 코드 양쪽 모두에 반영하라. 만약 모델이 코드를 작성하기에 적합하지 않다면 구현에 적합하도록 모델을 수정하라. 모델과 코드의 개념적 거리가 멀어지면 멀어질수록 소프트웨어는 도메인과 멀어지고, 도메인 전문가와 개발자는 서로 간의 개념을 일치시키기 위해 번역 과정을 거쳐야 하므로 의사소통이 어려워지고 유지보수가 어려운 코드를 얻게 된다. 모델을 기반으로 코드를 작성하되 모델과 코드를 동기화하기 위해 모든 노력을 쏟아라.

MODEL-DRIVEN DESIGN 을 적용하기 위해서는 분석/설계/구현이 개별적인 활동이라는 환상에서 벗어나야 한다. 이를 하나의 사이클로 묶음으로써 모델과 코드 간의 거리를 좁히도록 노력해야 한다. 둘 사이의 개념적 거리가 멀어지면 지속적인 리팩토링을 통해 간격을 좁히도록 최선을 다해야 한다.

모델링과 구현이 개별적인 사람들에 의해 이루어져야 한다는 착각 역시 MODEL-DRIVEN DESIGN 을 방해하는 요소다. 구현에 관여하는 사람들은 모델링 작업에 직접 참여해야 한다. 두 역할을 분리할 경우 모델링 작업에서 얻어지는 지식이 코드에 반영되지 못한 채 소리 없이 사라져 버리게 된다. 또한 구현 과정에서 얻어지는 통찰을 모델에 반영할 수 없게 되므로

결과적으로 코드와 모델 간의 연결 고리가 점점 약해지게 된다. 코드는 모델이며, 모델은 코드다. 극단적인 흑백논리는 소프트웨어 개발에 있어 최대의 적이다.

모델과 코드 간의 표현적 차이를 줄이기 위해서는 도메인 전문가와 개발자 간에 동일한 언어를 기반으로 의사소통하는 것이 중요하다. 역동적이고 활동적인 UBIQUITOUS LANGUAGE 가 MODEL-DRIVEN DESIGN 의 기반을 이룬다.

> 도메인 모델과 소프트웨어 시스템 간의 맵핑이 명확해지도록 도메인 모델을 충실하게 반영하는 소프트웨어 시스템을 설계하라. 도메인에 대한 더 깊은 식견을 반영할 방법을 찾는 순간 소프트웨어 내에서 모델을 더 자연스럽게 구현할 수 있도록 모델을 재검토하고 수정하라. 견고한 UBIQUITOUS LANGUAGE 를 지원함과 동시에 두 가지 목적 모두에 잘 부합하는 하나의 모델을 추구하라.
>
> 설계에서 사용되는 용어와 기본 책임 할당을 사용해서 모델을 작성하라. 코드는 모델의 표현이 되고 코드에 대한 수정은 모델에 대한 수정이 된다. 효과는 나머지 프로젝트 활동 내내 적절히 파급되어야 한다.
>
> - Eric Evans, Domain-Driven Design

UBIQUITOUS LANGUAGE

바벨탑을 쌓아 신의 권위에 도전하려던 인간의 오만은 결국 신의 분노로 인해 실패하고 만다. 인류 역사상 가장 거대한 공학 프로젝트인 동시에 가장 큰 프로젝트 실패 사례로 꼽히는 바벨탑의 가장 큰 실패 요인은 언어의 분할로 인해 이해관계자 간에 의사소통이 불가능해졌다는 것이다. 신의 권위에 도전하지 않는 현대의 소프트웨어 프로젝트에 있어 불가능한 의사소통이 프로젝트를 실패하게 만들지는 않을 지라도 그 길을 더디고 험난하게 만들 수는 있다.

그림 6 의사소통의 단절로 인해 실패한 바벨탑 프로젝트

도메인 전문가와 개발자가 서로 다른 용어를 사용할 경우 두 역할 간의 의사소통을 위해 번역 작업이 수반되어야 하며, 이는 곧 의사소통의 효율성 저하, 두 역할 간의 불협화음으로 인한 프로젝트 능률 저하, 도메인과 격리된 코드라는 결과를 낳게 된다. 따라서 프로젝트에 참여하는 모든 이해관계자들이 의사소통 시에 사용할 수 있는 공통 언어가 필요하다.

도메인 모델은 도메인에 대해 이해관계자들이 가지고 있는 관점과 지식의 축적이며 추상화의 산물이다. 도메인 모델 내에 표현된 모든 개념과 관계는 도메인을 바라보는 이해관계자들의 지식 탐구 활동을 통해 얻어진 통찰이 반영되어 있다. 따라서 도메인 모델은 이해관계자들의 의사소통을 위한 공통 언어 구축의 핵심 요소로 사용할 수 있다. 도메인 모델을 기반으로 한 공통 언어는 소프트웨어 개발과 관련된 전반적인 활동의 중심에 모델을 위치시키며, 결과적으로 모델과 코드를 연결시키기 위한 MODEL-DRIVEN DESIGN 에 있어 매우 중요한 연결고리를 제공한다.

> 모델을 언어의 기반으로 삼아라. 팀에서 이루어지는 모든 의사소통과 코드에 적극적으로 공통의 언어를 적용하라. 다이어그램과, 문서화, 특히 대화에 동일한 언어를 사용하라.
>
> 여러 가지 모델을 반영하는 다른 표현을 실험해봄으로써 공통 언어 선택에 따르는 어려움을 해소하라. 그 후 새로운 모델에 적합하도록 클래스, 메서드, 모듈의 이름을 변경하여 코드를 리팩토링하라. 일상에서 사용하는 용어를 다르게 사용할 경우 의미에 대한 공감대를 형성하는 것과 동일한 방식으로 대화에 사용하는 용어 상의 혼란 역시 해결하라.
>
> UBIQUITOUS LANGUAGE 의 변경은 곧 모델의 변경이라는 사실을 인식하자.
>
> - Eric Evans, Domain-Driven Design

애자일 진영은 문서나 다이어그램을 통한 의사소통보다는 직접 대면과 대화를 통한 의사소통 방식을 선호한다. UBIQUITOUS LANGUAGE 를 확립하기 위해서는 대화와 같은 직접적인 의사소통 수단을 적극 활용하여 언어가 팀 전체에 골고루 퍼지게 해야 한다. 언어가 널리 사용되면 사용될수록 이해관계자들 간의 원활한 의사소통이 가능해진다. 형식적인 문서화도 중요하지만 가능하면 언어가 지닌 장점을 최대한 활용하라.

UBIQUITOUS LANGUAGE 의 용어와 개념이 도메인에서 사용되는 개념이라고 해서 반드시 도메인 전문가들이 사용하는 용어만으로 구성되는 것은 아니다. 도메인 전문가의 용어 중 실제 소프트웨어가 해결하려는 컨텍스트와 관련된 용어들만이 UBIQUITOUS LANGUAGE 에 포함된다. 또한 개발에 필요한 용어 중 도메인 전문가와 개발자 간의 의사소통을 위해 필요한 용어들 역시 UBIQUITOUS LANGUAGE 에 포함된다. 따라서 UBIQUITOUS LANGUAGE 는 도메인 전문가들이 사용하는 용어 일부와 개발자들이 사용하는 용어 일부로 구성된다.

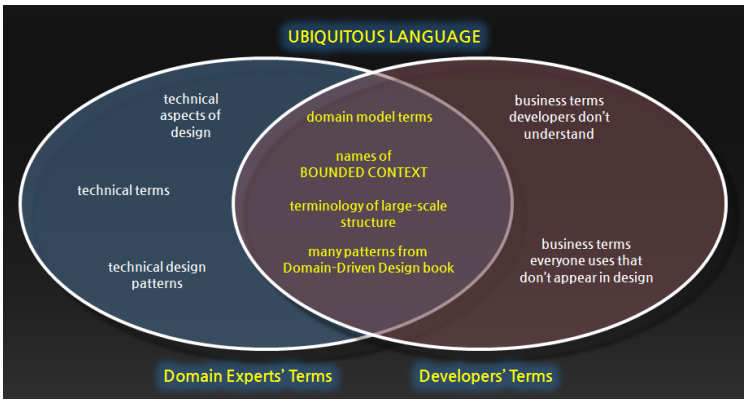


그림 7 도메인 전문가와 개발자 용어의 교집합으로 구성된 UBIQUITOUS LANGUAGE

UBIQUITOUS LANGUAGE 상의 용어에 변경이 발생하면 곧 모델, 코드에 대한 변경으로 파급되어야 한다. 역으로 코드에 사용된 어떤 용어가 수정될 경우 변경 사항은 모델과 UBIQUITOUS LANGUAGE 로 파급되어야 한다. 가장 중요한 것은 변경 전의 용어를 폐기처분하고 일상적인 의사소통, 특히 대화 시에 새로운 용어를 사용하도록 노력하는 것이다.

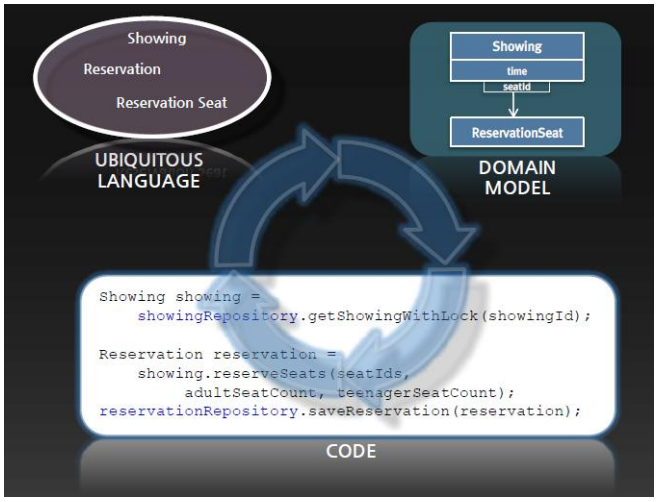


그림 8 UBIQUITOUS LANGUAGE와 코드 간의 순환 관계

패턴에 대해서

DDD 를 접하는 대부분의 사람들이 가장 먼저 접하게 되는 것이 ENTITY, VALUE OBJECT, AGGREGATE, SERVICE, REPOSITORY 와 같은 패턴이다. 지면상의 제약으로 인해 본 글에서 DDD 의 패턴에 대해 포괄적으로 설명하기는 어려울 것으로 판단되어 여기에서는 DDD 의 패턴을 바라보는 개인적인 견해를 피력하는 정도로 마무리하고자 한다.

패턴에 대한 자세한 내용이 궁금한 사람들은 Eric Evans 가 저술한 "Domain-Driven Design"을 읽어 보기 바란다. 부족하나마 필자의 블로그(http://aeternum.egloos.com/)에 올려져 있는 글을 읽어 보는 것도 도움이 되리라고 생각한다. 웹에서 얻을 수 있는 Domain Driven Design

Quickly 와 Domain-Driven Design Step-By-Step 역시 패턴을 이해할 수 있는 유용한 정보를 얻을 수 있다.

개인적으로 도메인 모델을 반영한 코드를 작성하고, 도메인 모델과 코드가 상호 연결된 관계를 유지할 수 있도록 해 주는 두 가지 핵심 원리인 UBIQUITOUS LANGUAGE 와 MODEL-DRIVEN DESIGN 이 DDD 의 핵심을 이루는 개념이라고 생각한다. 따라서 위 두 가지 원리를 이해하는 것이 DDD 의 개별 패턴을 이해하는 것 보다 더 중요하다.

그렇다고 해서 패턴이 중요하지 않다는 것은 아니다. 원리를 이해하는 것과 이를 실천하는 것은 별개의 문제다. DDD 에서 제시하는 패턴은 DDD 의 기본 원리를 실행할 수 있는 기반을 제공한다는 점에서 그 가치가 매우 높다. 그러나 기본적인 원리의 이해 없이 성급하게 개별 패턴을 적용하려는 시도는 위험하다. DDD 의 패턴에서 차용된 Annotation 을 코드에 사용하고 DDD 를 지원한다고 선전하는 프레임워크를 적용한다고 해서 DDD 를 적용하는 것은 아니다.

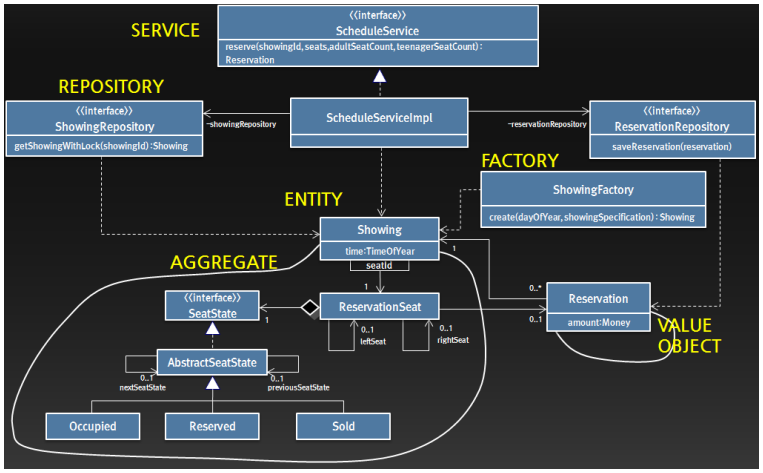


그림 9 패턴은 중요하다. 그러나 원리가 그보다 더 중요하다.

패턴의 적용 유무와 무관하게 다음 질문에 대해 모두 YES 라고 대답할 수 있다면 DDD 를 실천하고 있다고 봐도 무방하다.

- 모든 이해당사자들이 이해하는 도메인 모델이 존재하는가?

- 모든 이해당사자들이 의사소통에 사용하는 UBIQUITOUS LANGUAGE 가 존재하는가?

- 소프트웨어가 MODEL-DRIVEN DESIGN 방식으로 개발되고 있는가?

그래도 은 총알은 없다

DDD 는 끊임없는 지식 탐구 활동을 통해 적절한 도메인 모델을 선택하고, 선택된 도메인 모델을 기반으로 분석, 설계, 구현에 걸친 소프트웨어 개발 전 과정을 주도하고, 모델을 의사소통의 핵심 수단으로 사용하기 위해 적용할 수 있는 기법과 패턴의 집합이다. 그렇다면 DDD 가 과연 우리가

찾고 있던 은 총알인가? 그렇지는 않다고 생각한다. DDD 를 소프트웨어 개발과 관련된 모든 문제를 해결해 줄 수 있는 만병 통치약으로 생각해서는 안 된다.

DDD 는 장점이 많은 만큼 적용에 따르는 리스크와 제약 역시 크다. 그렇다면 DDD 를 적용해야 하는 때는 언제인가? 이에 대한 Casey Charlton 의 견해를 들어 보자.

> DDD 는 다음과 같은 경우 적용하기 적절한 소프트웨어 방법이다 – 매우 복잡하고 잘 정의된 비즈니스 모델에 초점을 맞추어야 하는 소프트웨어.
>
> 아마 모든 소프트웨어 애플리케이션의 95%는 "DDD 를 적용하기에 적절하지 않은" 범주에 속할 것이다. 대부분의 소프트웨어는 근본적으로 데이터 중심적이다 – 대부분의 웹 사이트가 그렇고, 대부분의 데스크톱 애플리케이션이 그렇다. 사실 데이터를 수정하고 보고하는 대부분의 애플리케이션은 데이터 중심적이라고 할 수 있다. 나머지 부류의 애플리케이션이라고 하더라도 매우 복잡한 도메인을 가진 경우는 많지 않다. 대부분은 단순하거나, 소프트웨어의 규모가 크더라도 그렇게 복잡하지는 않다.
>
> DDD 를 적용하기에 적절한 나머지 5%의 애플리케이션에 대해서는 DDD 를 적용하기에 매우 적합하다고 할 수 있다. 그런 상황에서 DDD 를 적용하는 것은 어렵고 복잡한 문제를 해결하는데 있어 매우 큰 도움이 될 것이다. 이런 상황이라면 DDD 는 늑대를 물리치기 위한 은 총알이 될 지도 모른다.
>
> 비록 애플리케이션이 DDD 에 적합한 5% 범주에 포함되지 않는다고 하더라도 DDD 에는 대량의 지혜와 경험이 녹아 있다 – 자신의 상황에 적용할 수 있다고 생각되는 기법을 적용하면 소프트웨어가 좀 더 유연해지고, 사용자에게 더 잘 반응하며, 이해하기 더 쉬워짐을 알게 될 것이다.
>
> \- Casey Charlton, Domain Driven Design Step by Step Guide

DDD 는 복잡한 도메인을 공략하기 위한 최상의 설계 지침을 제공한다. DDD 의 가치는 소프트웨어의 부차적인 문제가 아니라 본질적인 문제를 해결하기 위해 모든 사람들의 역량을 모을 수 있도록 해준다는데 있다.

비록 DDD 의 지침을 현재의 프로젝트에 적용할 수 없는 상황에 있다고 하더라도 DDD 의 세계에 발을 들여 놓는 것이 큰 도움이 될 것이라 생각된다. Eric Evans 의 충고에 귀를 기울여 보자.

> 개발자들이 [도메인에 대한] 통찰을 얻기 위해 적용할 수 있는 체계적인 사고 방법이 존재한다. 무질서하게 뻗어 나가는 소프트웨어 애플리케이션에 질서를 부여할 수 있는 설계 기법 역시 존재한다. 이런 기술을 연마한다면 익숙하지 않은 도메인을 접하게 될 경우에도 더 가치 있는 개발자로 발전할 수 있게 될 것이다.
>
> \- Eric Evans, Domain-Driven Design

참고 자료

Robert L. Glass, 소프트웨어 컨플릭트 2.0, 위키북스, 2007

Robert L. Glass, 소프트웨어 공학의 사실과 오해, 인사이트, 2004

Pete McBreen, 소프트웨어 장인 정신, 피어슨에듀케이션코리아, 2002

Paul Graham, 해커와 화가, 한빛미디어, 2005

Manifesto for Agile Software Development, http://agilemanifesto.org/

Kent Beck, 익스트림 프로그래밍, 인사이트, 2006

Frederick Brooks, 맨먼스 미신 – 소프트웨어 공학에 관한 에세이, 케이앤피북스, 2007

Alastair Cockburn, Agile 소프트웨어 개발, 피어슨 에듀케이션 코리아, 2002

Eric Evans, Domain-Driven Design, Addison-Wesley Professional, 2003

Abel Avram, Domain-Driven Design Quickly, http://www.infoq.com/minibooks/domain-driven-design-quickly

Casey Charlton, Domain-Driven Design Step-By-Step, http://dddstepbystep.com/cfs-filesystemfile.ashx/__key/CommunityServer.Components.SiteFiles/Domain-Driven-Design-_2D00_-Step-by-Step.pdf

Martin Fowler, Is There Such a Thing as Object-Oriented Analysis?, http://martinfowler.com/distributedComputing/analysis.pdf

Martin Fowler, POJO, http://www.martinfowler.com/bliki/POJO.html

Martin Fowler, 엔터프라이즈 애플리케이션 아키텍처 패턴, 피어슨 에듀케이션 코리아, 2003

Michael Jackson, Software Requirements & Specification – a lexicon of practice, principles and prejudices, Addison-Wesley, 1995

Rod Johnson, Expert One-on-One J2EE Design and Development, Wrox, 2002

Craig Larman, UML 과 패턴의 적용 2/e, 홍릉과학출판사, 2003

Robert C. Martin, 소프트웨어 개발의 지혜, 야스미디어, 2004

Ken Schwaber, Mike Beedle, 스크럼: 팀의 생산성을 극대화시키는 애자일 방법론, 인사이트, 2008