1 장 진화적 데이터베이스 개발

"폭포는 관광객들에게 훌륭한 볼거리다. 하지만 폭포수 모델(Waterfall Model)[1]은 소프트웨어 개발 프로젝트를 진행하는 데 있어 구경거리라 할 만한 형편없는 전략이다."

–스콧 엠블러

반복적이고 점진적으로 개발을 수행하는 데 필요한 현대의 소프트웨어 프로세스 또는 방법론이라 부르는 것은 사실상 모두 진화적이다. 예를 들어 RUP(Rational Unified Process)[2], XP(Extreme Programming)[3], 스크럼(Scrum), 동적 시스템 개발 방법론(DSDM), 크리스탈 패밀리, 팀 소프트웨어 프로세스(TSP), AUP(Agile Unified Process), EUP(Enterprise Unified Process), 기능 주도 개발(FDD), 그리고 RAD(Rapid Application Development)[4] 등과 같은 프로세스가 이에 해당한다. 반복적으로 수행한다는 것은 모델링이나 테스팅, 코딩, 또는 개발을 동시에 조금씩 진행하고 나서 다른 것을 조금, 그리고 또 다른 것을 조금씩 하는 식의 진행을 말한다. 이러한 프로세스는 구현해야 할 모든 요구사항을 먼저 정의하고 난 후에, 그에 따른 상세 설계를 만들고, 그리고 나서 해당 설계를 구현하여 테스트하고 최종적으로 시스템을 배포하는 순차적인 접근방법론과는 조금 다르다.

1. 폭포수 모델(Waterfall model): 전통적인 소프트웨어 개발 모델로, 각 단계를 확실히 매듭짓고 다음 단계로 넘어가는 개발 방법론. 그 모양이 한번 떨어지면 거슬러 올라갈 수 없는 폭포수와 같다고 해서 이름 붙여짐.
2. RUP(Rational Unified Process): 원래는 Rational 사(지금은 IBM으로 합병)의 객체지향 방법론.
3. XP(Extreme Programming): 켄트 벡, 론 제프리 등에 의해 제안된 애자일 방법론 중 하나.
4. RAD(Rapid Application Development): 반복적이고 점진적인 개발 방법론으로 폭포수 모델과 대조적임.

점진적인 접근방법에서는 한 번에 대규모로 배포하기보다는 일련의 작은 배포로써 시스템을 구축해간다. 더욱이, 오늘날 대부분의 프로세스들은 단순하면서도 사실상 진화적이고 고도의 협업성을 모두 가진 애자일 프로세스이다. 팀이 협업적인 접근법을 취하게 되면, 팀원들은 효과적으로 함께 작업할 방법을 열심히 찾을 것이다. 이때 비즈니스 고객 같은 프로젝트 이해 관계자들을 프로젝트의 실제 멤버가 될 수 있도록 반드시 확인해야 한다. 코번(Cockburn 2002)은 상황에 적절히 적용할 수 있는 최신의 의사소통 기법을 도입하는 데 노력을 아끼지 말라고 조언한다. 즉 전화통화보다는 화이트보드 주위에 모여 얼굴을 맞대고 대화하는 것이 좋고, 누구에게 전자우편을 보내기에 앞서 전화를 걸고, 상세한 문서를 전달할 때에는 이메일을 사용하는 것이 좋다고 말한다. 소프트웨어 개발팀 내에서 보다 나은 의사소통과 협업은 프로젝트를 성공적으로 이끄는 핵심 열쇠이다.

진화적인 것과 애자일한 방법 모두가 개발 분야에서는 손쉽게 도입되었지만 데이터 분야에서는 그렇다고 말할 수가 없다. 소프트웨어 구현을 시작하기에 앞서 상당히 상세한 수준의 데이터 모델이 필요한데, 이들 대부분의 데이터 지향적 기술은 사실상 순차적이다. 더 나쁜 것은, 이러한 모델이 종종 베이스라인이 되며, 변경을 최소화하기 위해 변경 관리 제어 시스템하에 둔다는 것이다(최종 결과를 고려해 본다면, 이것은 실제 변경 방지 프로세스라 불러야 한다). 문제는 여기에 있다.[5] 다시 말해, 일반적인 데이터베이스 개발 기술은 현대의 소프트웨어 개발 프로세스의 현실을 반영하지 않는다. 이런 식으로 할 필요가 없는 것이다.

우리의 전제조건은 데이터 전문가들이 이러한 개발자들과 마찬가지로 진화적인 기술을 도입할 필요가 있다는 것이다. 설령 개발자들도 데이터 분야 내에서는 유효성이 입증된 전통적인 접근방법론으로 돌아가야 한다고 주장하더라도, 전통적인 방법이 늘 제대로 수행되고 있지는 않다는 사실이 점점 더 분명하게 나타나고 있다. 5장 애자일과 반복적인 개발에서 크래이그 라만(Craig Larman, 2004)은 정보 기술(IT) 분야를 선도하는 리더들 사이에서 압도적으로 지지받는 진화적 접근 방법과 그 연구 결과를 제시하고 있다. 결론은 개발분야에서 널리 사용되고 있는 진화적이고 애자일한 기술이 데이터 분야에서 일반적으로 사용되고 있는 전통적인 기술보다 더 좋다는 것이다.

데이터 전문가들이 그렇게 하겠다고 선택만 한다면 그들의 작업 전 영역에 걸쳐 이러한 진화적인 접근방법론을 도입하는 것이 가능하다. 그 첫 번째 단계는 IT 프로젝트 팀의 요

5. Therein lies the rub: 세익스피어의 햄릿에서 인용한 문구

구사항을 반영할 수 있도록 여러분의 IT 조직의 '데이터 문화'를 다시 생각하는 것이다. '애자일 데이터(AD) 방법론'(The Agile Data method, 엠블러 2003)에서는 현대의 데이터 지향 활동에 대한 일련의 원칙과 역할을 정확하게 설명하고 있다. 그 원칙은 데이터를 비즈니스 소프트웨어의 많은 중요한 관점 중 하나로서 어떻게 반영할 것인가 하는 것이며, 개발자는 데이터 기술에 더욱 적응해야 할 필요가 있다는 것과 데이터 전문가 또한 현대의 개발 테크닉과 기법을 배워야 할 필요성이 있음을 내포한다. AD 방법론을 사용함으로써, 각 프로젝트 팀은 그 프로젝트 상황에 맞는 고유한 프로세스에 따라야 할 필요가 있다는 것을 인식하게 된다. 현재의 프로젝트가 다루는 엔터프라이즈 이슈 그 이상을 바라보는 중요성 또한 강조되며 이와 같은 것은 애자일 방법론을 수행하여 프로젝트 팀과 함께 작업하기에 충분한 유연성을 가지려는 데이터베이스 운영 관리자와 데이터 아키텍트 같은 엔터프라이즈 전문가에게 필요하다.

두 번째 단계는 진화적인 방법으로 작업을 가능케 하는 새로운 기술을 도입하려는 데이터 전문가를 위한 것으로, 특히 데이터베이스 관리자가 이에 해당한다. 1장에서는 이들 핵심 기술을 간단히 소개할 것이지만, 이 책에서 초점을 두고 있는 가장 중요한 기술은 데이터베이스 리팩토링이다. 진화적인 데이터베이스 개발 기술은 다음과 같다:

1. **데이터베이스 리팩토링**. 기존의 데이터베이스 스키마를 한 번에 조금씩 작게 발전시켜 나가는 것으로 스키마 자체의 의미는 변경하지 않고 그 디자인의 질을 개선한다.
2. **진화적 데이터 모델링**. 시스템의 데이터 관점을 반복적이고 점진적으로 모델링하는 것으로 다른 모든 시스템 관점과 마찬가지로 데이터베이스 스키마와 애플리케이션 코드를 함께 서서히 발전시켜 나간다.
3. **데이터베이스 회귀**(regression) **테스트**. 데이터베이스 스키마가 실제로 수행되는지 확인한다.
4. **데이터베이스 가공품**(artifacts)**의 형상관리**. 데이터 모델, 데이터베이스 테스트, 테스트 데이터 등이 중요한 프로젝트 가공품으로서 다른 어느 가공품과 마찬가지로 관리되어야 한다.
5. **개발자 샌드박스**. 개발자는 그들 자체적으로 수행할 테스트 환경이 필요하다. 이 환경에서 구축한 시스템의 일부를 변경하기도 하고, 같은 팀사람들과 통합 작업을 하기 전에 테스트를 수행하기도 한다.

각각의 진화적 데이터베이스 기술을 자세하게 살펴보기로 하자

1.1 데이터베이스 리팩토링

리팩토링(마틴 파울러 1999)은 소스코드를 조금씩 변경하는 것으로 그 디자인을 개선하고 그렇게 함으로써 기존보다 쉽게 작업할 수 있도록 하는 훈련된 방법이다. 리팩토링의 핵심적인 관점은 소스코드의 행위적 의미를 유지하는 것이다[6]–리팩토링 시 소스코드의 어떠한 추가나 삭제도 없다. 다만 그 질을 향상시킨다. 리팩토링 예는 getPersons() 오퍼레이션을 getPeople()로 이름을 변경하는 것이다. 이런 리팩토링을 구현하려면 오퍼레이션의 정의를 수정하고 나서, 애플리케이션 코드 전체에 걸쳐 이 오퍼레이션의 모든 호출을 변경하여야 한다. 리팩토링은 코드가 수정 전처럼 아무 문제 없이 다시 실행될 때까지는 끝난 것이 아니다.

이와 비슷하게, 데이터베이스 리팩토링은 데이터베이스 스키마를 간단히 변경하여 그 디자인을 개선하되, 그것의 행위와 정보의 의미는 모두 원래대로 유지하는 것이다. 테이블과 뷰 정의 같은 데이터베이스 스키마의 구조적 관점이나, 저장 프로시저와 트리거 같은 함수적 관점 중 하나를 리팩토링할 수 있다. 데이터베이스 스키마를 리팩토링할 때는 스키마 자체를 뜯어 고쳐야 할 뿐만 아니라, 해당 스키마에 종속성을 가진(커플링된) 비지니스 애플리케이션이나 데이터 추출 같은 애플리케이션 또한 재작성되어야 한다. 데이터베이스 리팩토링은 코드 리팩토링보다 그 구현이 훨씬 더 어렵기에 매우 신중해야 한다. 데이터베이스 리팩토링은 2장에서 자세하게 설명하며, 3장에서는 데이터베이스 리팩토링을 수행하는 프로세스에 대해 설명한다.

1.2 진화적 데이터 모델링

이전에 뭐라고 들었든 상관없이, 진화적이고 애자일한 기술은 단순한 "일단 짜보고 고치기(code-and-fix)[7]"의 새로운 이름이 아니다. 여전히 시스템을 구축하기 이전에 요구사항을 찾아내야 하고 아키텍처와 디자인을 처음부터 끝까지 깊이 생각해야 한다. 그리고 이렇게 하

6. 다시 말하면, 겉으로 보이는 동작의 변화없이 내부 구조를 변경한다

7. code-and-fix: 이러한 개발 방식은 계획 수립과 분석, 설계의 상위 과정을 무시하고 오로지 코딩(프로그래밍)만을 강조하는, 빨리 일정을 끝내려는 변형된 무리한 속성 과정

는 한 가지 좋은 방법은 코딩하기 전에 먼저 모델링을 하는 것이다. 그림 1.1은 애자일 모델 주도 개발(AMDD) (엠블러 2004; 엠블러 2002)에 대한 생명주기를 리뷰한 것이다. AMDD를 사용하면 프로젝트 시작 시점에서 상위 레벨의 초기 모델링을 만든다. 이 모델을 사용하여 해결하고자 하는 문제 도메인의 전체적인 범위 개요를 나타내고, 구축할 잠재적인 아키텍처를 모델링한다. 전형적으로 생성하는 모델 중 하나는 '슬림(slim)' 개념적/도메인 모델로 주 비즈니스 엔터티와 엔터티 사이의 관계(relationship)를 그린 것이다.(파울러와 세달라지 2003). 그림 1.2는 금융기관의 간단한 예를 그린 것이다. 이 예제에 프로젝트의 시작 시점에 필요한 모든 것을 상세하게 선보였다. 목표는 프로젝트 초기에 전체의 주요 이슈를 생각하는 것이다. 당장은 상세화할 필요성이 없는 것에 시간을 쓰지 말고 JIT(Just-in-time) 기반으로 나중에 필요할 때 상세화한다.

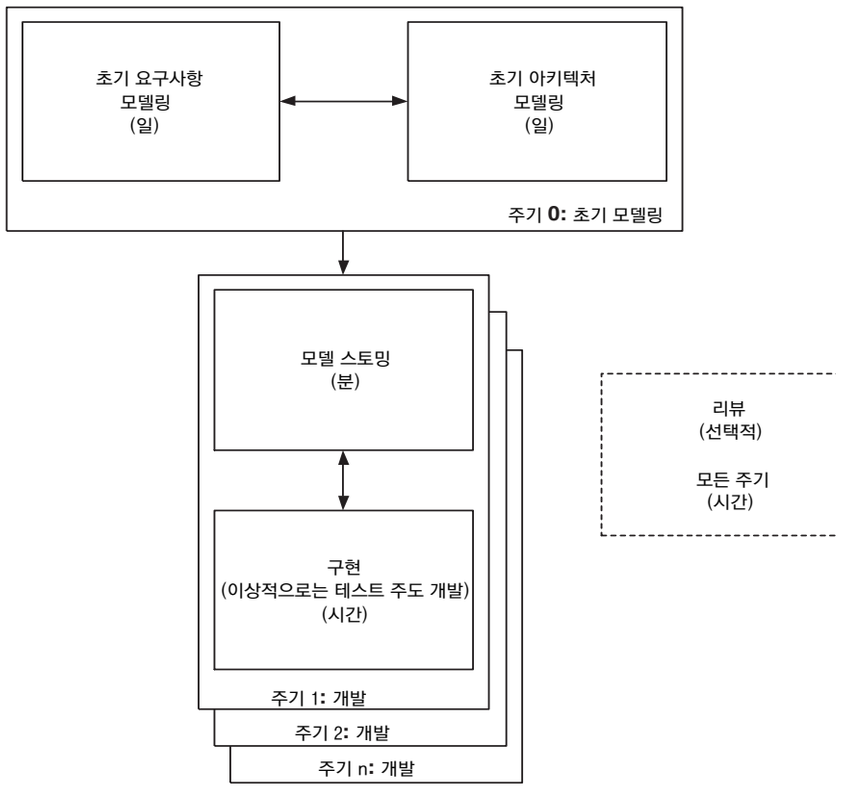

그림1.1 애자일 모델 주도 개발(AMDD) 생명주기

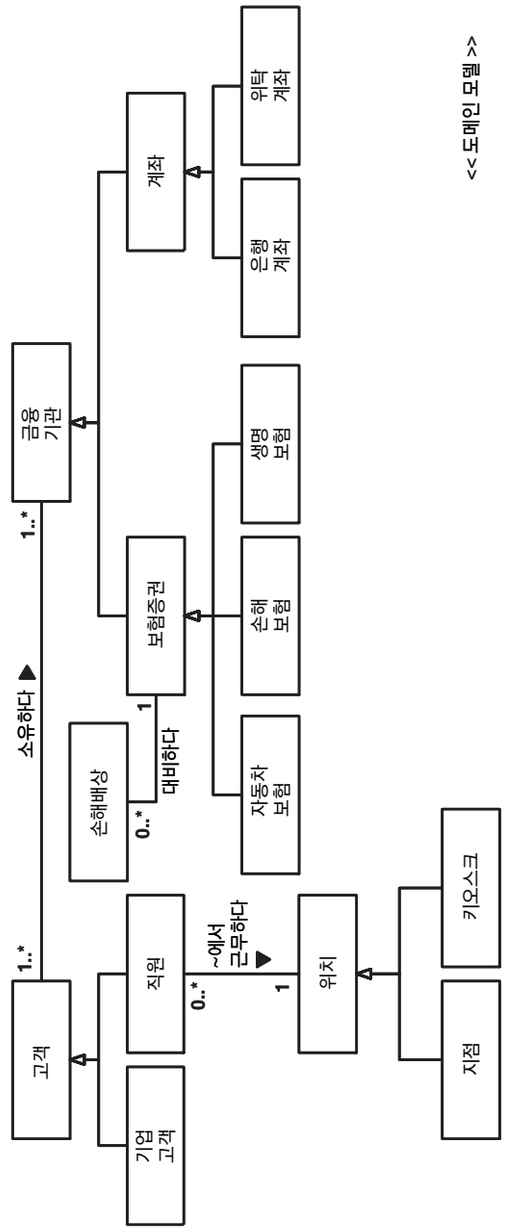

그림1.2 UML을 사용한 가상 금융기관의 개념적/도메인 모델

개념적 모델은 도메인이 점차 커지는 것을 이해함으로써 자연스럽게 점점 발전되지만, 상세 레벨은 동일하게 남겨둔다. 상세화는 객체모델(소스코드가 된다)과 물리적 데이터 모델 내에서 파악된다. 이러한 모델은 개념적 도메인 모델에 의해 이끌리며, 그리고 일관성을 보장하는 다른 아티팩트[8]와 함께 동시에 개발된다. 그림 1.3은 세 번째 개발 주기의 끝 단계에서 모델의 범위를 표현한 상세 물리적 데이터 모델(PDM)을 그린 것이다. 만약 '주기 0' 단계가 1 주일이 걸렸다면, 전형적인 프로젝트 기간은 1년 이하이고 개발 주기는 2주 정도다. 이 PDM은 프로젝트 7주 후에 작성한 것이다. PDM은 이 지점에 오기까지의 프로젝트의 데이터 요구사항과 어떠한 레거시 제약조건이라도 반영한다. 미래의 개발 주기에 대한 데이터 요구사항은 JIT에 근거하여 이 주기동안 모델된다. 진화적 데이터베이스 모델링은 쉽지 않다. 레거시 데이터 제약조건을 고려해야 할 필요가 있으며, 모두 알고 있듯이 부주의하고 쓸모없는 소프트웨어 개발 프로젝트가 만든 레거시 데이터 소스는 종종 매우 지저분하다. 다행히도 훌륭한 데이터 전문가가 원래의 데이터 소스의 뉘앙스를 이해해서 순차적 접근방법론에 기초를 둔 것처럼 쉽게 JIT에 근거하여 적용할 수 있도록 하였다. 애자일 모델링의 모델링 표준 적용 수행 지침처럼 여전히 인텔리전트 데이터 모델링 규칙을 적용할 필요가 있다. 진화적/애자일 데이터 모델링의 상세한 예제는 www.agiledata.org/essays/agileDataModeling.html에 있다.

1.3 데이터베이스 회귀 테스트

기존의 소프트웨어를 안전하게 수정하기 위해 리팩토링을 하거나 새로운 기능을 추가하는 경우, 어떤 부분을 수정한 후에도 잘못된 점이 없다는 사실을 확인할 필요가 있다. 다시 말하자면, 시스템상에서 회귀 테스트를 수행할 수 있어야 한다는 것이다. 수정후 뭔가 잘못된 점을 발견했다면, 그것을 고치든지 아니면 변경사항을 다시 되돌려야 한다. 개발자들 사이에서 회귀테스트는 점점 일반적인 일이 되어 가고 있다. 그 이유는 프로그래머들이 코드와 단위 테스트 슈트를 동시에 개발하기 위해서이기도 하고, 사실상 애자일리스트들은 실제 코드를 작성하기 전에 테스트 코드를 먼저 사용하는 것을 선호하기 때문이다. 단지 애플리케이션 소스 코드를 테스트하는 것처럼 데이터베이스를 테스트하면 안 되는 것인가?

8. Artifact: 시스템을 개발, 운영또는 지원하는 동안 만들어지고, 변경되는 문서나, 모델, 파일, 다이어그램(diagram) 또는 관련된 다양한 산출물들

중요한 비즈니스 로직은 저장 프로시저 형태나, 데이터 검증규칙, 참조무결성(RI) 규칙을 따라 데이터베이스 내에서 구현되고, 명확한 테스트를 거쳐야 한다

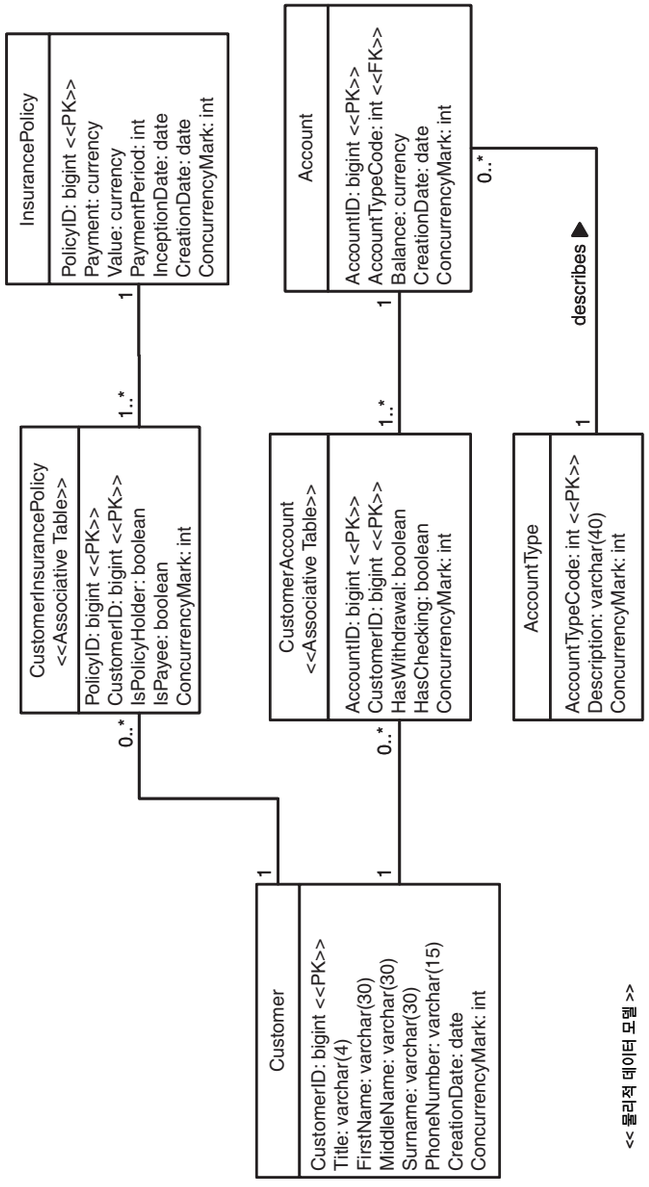

그림1.3 UML을 사용한 상세 물리적 데이터 모델링

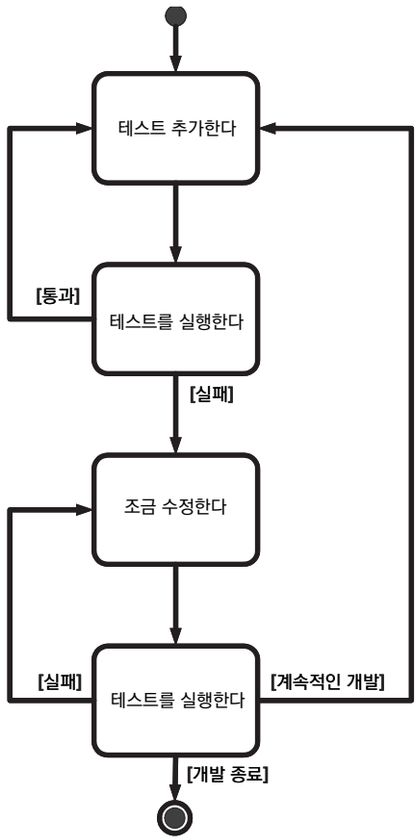

그림1.4 개발을 위한 테스트 우선 접근 방법

흔히 테스트 우선 프로그래밍(Test-First Programming)이라고도 하는 테스트 우선 개발(TFD)은 개발 방법에 있어 진화적인 접근방법이다. 새로운 코드를 작성하기 전에 먼저 실패할 테스트 코드를 작성해야만 한다. 그림 1.4의 UML 액티비티 다이어그램에 나타난 바와 같이, TFD의 단계는 다음과 같다.

1. 빠르게 테스트를 추가하라. 지금은 테스트가 실패할 것이므로 기본적으로 테스트하기에 충분한 코드면 된다.

2. 테스트를 실행하라. 단지 빠른 속도를 위해 종종 완전한 테스트 슈트가 단지 일부분만 실행된다 하더라도 새로운 테스트가 사실상 실패한다는 것을 확인하기 위해서 테스트를 실행하라.
3. 새로운 테스트를 통과하기 위해 코드를 업데이트하라.
4. 테스트를 다시 실행하라. 만약 테스트가 실패했을 경우, 단계3으로 되돌아간다. 그렇지 않으면, 처음부터 다시 시작한다.

TFD의 가장 좋은 점은 기능을 구현하기 전에 그 기능에 대해 충분히 생각할 수 있게 해준다는 것이다(여러분은 효과적으로 상세 설계를 하고 있다). 또한 여러분의 작업물을 언제라도 검증할 수 있는 테스트 코드가 존재한다는 것을 보증하며 그것이 시스템을 변화시킬 수 있는 용기를 준다. 왜냐하면 누구나 자신이 변경한 결과로 인해 무언가 '잘못되었다'면 이를 강하게 거부할 수 있기 때문이다. 단지 애플리케이션 소스 코드의 완전한 회귀 테스트를 수행할 때 코드 리팩토링이 가능한 것과 마찬가지로, 데이터베이스를 위한 완전한 회귀 테스트는 데이터베이스 리팩토링을 원활하게 한다(메스자로스 2006).

테스트 주도 개발(TDD) (아스텔스 2003; 벡 2003)은 TFD와 리팩토링이 결합된 것이다. 먼저 TFD 접근법에 따라 코드를 작성하고, 그 후에 실행한다. 필요에 따라 리팩토링에 의한 높은 수준의 설계가 가능하다는 것을 확인할 수 있다. 리팩토링을 할 때, 하나라도 잘못된 것이 없는지 확인하기 위해 회귀테스트를 반복해서 진행해야 한다.

이것은 여러 가지의 단위 테스팅 툴을 필요로 하게 될 것이라는 사실을 의미한다. 데이터베이스를 위한 적어도 하나 이상의 툴과, 각각의 프로그래밍 언어를 위해 외부 프로그램에서 사용되는 하나 이상의 툴이 필요하게 될 것이다. XUnit 툴은(자바의 Junit, 비주얼베이직의 VBUnit, 닷넷을 위한 Nunit, 그리고 오라클의 Ounit등) 다행히도 무료이며, 서로 간에 유사성을 가지고 있다.

1.4 데이터베이스 아티팩트 형상관리

때때로 시스템 변경이 나쁜 아이디어였음이 입증된다면, 변경하기 이전의 상태로 되돌릴 필요가 있다. 예로 Customer.Fname 컬럼을 Customer.Firstname으로 이름을 변경하는 것이 50개의 외부 프로그램을 중단시킬지도 모른다면 이 프로그램의 업데이트 비용은 지

금으로선 매우 높은 것이다. 데이터베이스 리팩토링을 가능케 하려면 형상관리 제어 시스템 하에 다음의 항목들을 저장할 필요가 있다:

- 데이터베이스 스키마를 생성할 데이터 정의 언어(DDL) 스크립트
- 데이터 적재/ 추출/ 마이그레이션 스크립트
- 데이터 모델 파일
- ORM(Object/relational mapping) 메타 데이터
- 참조 데이터
- 저장프로시저 및 트리거 정의서
- 뷰 정의서
- 참조무결성 제약조건
- 시퀀스, 인덱스등과 같은 기타 데이터베이스 개체
- 테스트 데이터
- 테스트 데이터 생성 스크립트
- 테스트 스크립트

1.5 개발자 샌드박스

'샌드박스'는 시스템이 구축, 테스트 및(또는) 실행되는 완전한 개발환경이다. 안전성의 이유로 다양한 샌드박스를 분리해서 유지해야 한다. 즉 개발자가 다른 사람의 노력의 결실에 손실을 줄 수 있다는 두려움을 가지지 않도록 자신만의 샌드박스에서 작업하는 것이 가능해야만 한다. 품질보증 및 테스트 그룹도 개발자가 그들의 소스 데이터와(또는) 시스템 기능에 오류를 줄 수 있다는 것에 대한 걱정없이 안전하게 시스템 통합 테스트를 수행할 수 있어야 한다. 그림 1.5는 샌드박스의 논리적 구성을 그린 것이다. 이는 논리적인 것이며 실제로 규모가 크고 복잡한 환경은 아마도 7~8개의 물리적 샌드박스를 가질 것이고, 반면 규모가 작고/단순한 환경에서는 2~3개의 물리적 샌드박스를 가질 것이다. 데이터베이스 스키마를 성공적으로 리팩토링하기 위해서 개발자는 소스 코드를 복사하고 작업할 데이터

베이스를 복사하여 점진적으로 발전시켜 나갈 자신만의 물리적 샌드박스를 가질 필요가 있다. 자신만의 개발 환경을 가짐으로써 안전하게 변경을 수행할 수 있고, 테스트하고 테스트한 것 중 하나를 취하거나, 버릴수 있다. 데이터베이스 리팩토링이 눈에 보일만큼 만족스럽다면 공유 프로젝트 환경으로 승급시켜 테스트하고, 형상관리 제어 시스템하에 저장함으로써 나머지 팀이 전달받게 된다. 결국, 팀은 모든 데이터베이스 리팩토링을 포함한 그들의 작업을 어떤 데모 및(또는) 프로덕션 테스팅 환경으로 승급시킨다. 이러한 승급은 보통 개발 주기에서 한 번 발생한다. 하지만 환경에 따라서는 종종 일어나기도 한다.(더 자주 시스템을 승급시키는 것은 가치 있는 피드백을 얻을 수 있는 좋은 기회이다.) 마지막으로, 시스템의 수용과 시스템 테스트가 통과한 후에는 프로덕션에 배포가 된다. 4장 '프로덕션으로 배포하기'에서는 좀 더 자세하게 이러한 승급/배포 프로세스에 대해서 다룬다.

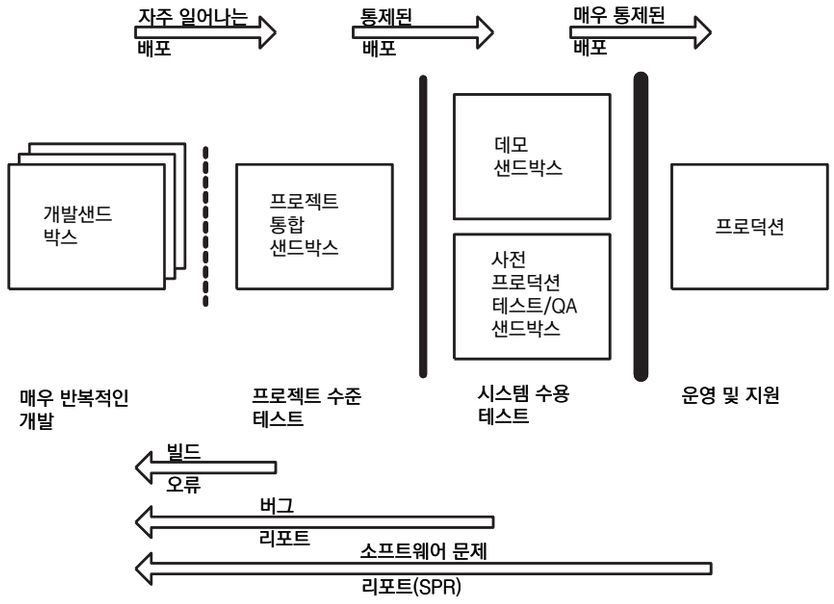

그림1.5 개발자에게 안전성을 제공하는 논리적 샌드박스

1.6 진화적 데이터베이스 개발 기술의 장애물

이 책에 언급한 기술들을 도입하는 데 따르는 일반적인 장애물을 논하지 않는다면 태만이나 마찬가지다. 그 첫 번째 장애물로 가장 극복하기 어려운 것이 바로 문화다. 오늘날의 데이터 전문가 대부분은 '일단 짜보고 고치기(code-and-fix)' 접근론을 개발에 일반적으로 사용했던 때인 1970년대와 1980년대 초에 경력을 쌓기 시작했다. IT 분야는 이 접근론이 질이 낮고, 코드를 유지보수하기 어렵게 하고 무겁고 구조적 개발 기술을 도입하는 결과를 가져다준다는 것을 인식하게 되었는데, 이들 대부분이 오늘날까지 여전히 쓰이고 있다. 이러한 경험 때문에 대다수의 데이터 전문가는 1990년대의 『오브젝트 테크놀로지 레볼루션[9]』에 의해 소개된 진화적 기술이 단지 1970년대의 '일단 짜보고 고치기'와 다를바 없는 재탕이라고 믿게 되었다. 솔직히 말해서, 많은 객체 실천가들이 사실상 그런 방법으로 작업을 했다. 그들은 진화적 접근론과 질이 낮은 것을 동등하게 생각했다. 하지만 애자일 진영이 나타남으로써 그와 같은 경우는 되풀이 되지 않을 것이다. 결국 대다수의 데이터 지향적 방법론은 과거의 프로세스를 통한 순차적이고, 전통적인 방법에 의해 궁지에 몰리는 양상을 띄게 되었고, 이는 대게 애자일 접근법의 잘못된 사용이 가져다 준 결과이다. 데이터 분야는 따라잡아야 할 것들이 많고, 이들 대부분은 많은 시간이 소요된다.

두 번째 장애물은 부족한 도구이다. 오픈 소스의 노력(적어도 자바진영 내에서는)이 빠르게 그 틈새를 메워주고는 있지만 여전히 부족하다. 설령 많은 노력이 ORM(object/relational mapping) 툴과 몇몇 데이터베이스 툴 개발에 들어가고 있다 하더라도 여전히 해야 할 일들은 많이 남아 있다. 툴 벤더가 그들의 툴 안에 리팩토링 기능을 구현하기 위해 수년동안 프로그래밍한 것처럼—사실, 지금의 통합 개발 환경(IDE)을 찾아보더라도 이 같은 기능은 제공하지 않는다 – 데이터베이스 툴 벤더도 동일한 기능을 구현하려면 수년이 걸릴 것이다. 분명하게 말해서, 데이터베이스 스키마의 진화적인 개발이 가능하려면 사용 가능한, 유연한 툴이 필요하다 – 오픈소스 진영은 이런 틈새를 채우기 위해 분명히 시작하고 있고 아마 상업 툴 벤더도 결국 동일하게 시작할 것이라고 생각한다.

9. Object Technology Revolution: 마이클 K.거트만, 존 R.매튜가 공통 집필한 저서로 비즈니스 데이터 처리에 있어 분산 객체 기술의 필요성을 다룬 책.

1.7 무엇을 배웠나

개발에 있어 반복적이고 점진적인 진화적 접근법은 현대 소프트웨어 개발을 위한 디팩토 표준[10]이라 할 수 있다. 프로젝트 팀이 이러한 접근방법을 사용하기로 결정했다면 데이터 전문가를 포함한 팀의 모두가 진화적인 방법으로 작업을 수행해야 한다. 다행히도 진화적인 기술은 데이터 전문가가 진화적인 방법으로 작업이 가능하도록 해 준다. 이러한 기술에는 데이터베이스 리팩토링, 진화적 데이터 모델링, 데이터베이스 회귀 테스트, 데이터 지향적 아티팩트의 형상관리와 개발자 샌드박스 분리 등이 포함된다.

10. 디팩토 표준(Defecto standard): 프랑스어로 사실상의 표준이란 의미로 개발분야에서 자주 사용되는 용어