4장. 도메인의 격리

복잡한 작업을 처리하는 소프트웨어를 만들수록 관심사의 분리가 필요하며, 이로써 격리된 상태에 있는 각 설계 요소에 집중 할 수 있다. 그와 동시에 시스템 내의 정교한 상호작용은 그러한 분리와는 상관없이 유지 되어야한다.

소프트웨어 시스템을 분리하는 방법

분리 방법은 다양하지만, 몇 개의 일반화된 계층이 널리 받아들여 지고있다. 계층화의 핵심 원칙은 한 계층의 모든 요소는 오직 같은 계층에 존재하는 다른 요소나 계층상 아래에 위치한 요소에만 의존한다는 것이다. 위로 올라가는 의사소통은 반드시 간접적인 메커니즘을 거쳐야 한다.

4가지 개념적 계층

- 사용자 인터페이스
- 응용 계층
- 도메인 계층
- 인프라스트럭처 계층

사용자 인터페이스 계층

사용자에게 정보를 보여주고 사용자의 명령을 해석하는 일을 책임진다.

- Controller

응용 계층

소프트웨어가 수행할 작업을 정의하고 표현력 있는 도메인 객체가 문제를 해결하게 한다.
여기에는 업무 규칙이나 지식이 포함되지 않으며, 오직 작업을 조정하고 아래

에 위치한 계층에 포함된 도메인 객체의 협력자에게 작업을 위임한다.

- Service

도메인 계층

이 계층에서는 업무 상황을 반영하는 상태를 제어하고 사용하고, 그 와 같은 상태 저장과 관련된 기술적인 세부사항은 인프라스트럭처에 위임한다. 이 계층은 업무용 소프트웨어의 핵심이다.

- DAO,VO

인프라스트럭처 계층

상위 계층을 지원하는 일반화된 기술적 기능을 제공한다, 애플리케이션에 대한 메시지 전송, 도메인 영속화, UI에 위젯을 그리는 것 등이 있다.

- Database

계층 간 관계 설정

각 계층은 설계 의존성을 오직 한 방향으로만 둬서 느슨하게 결합된다. 상위 계층은 하위 계층의 공개 인터페이스를 호출하고 하위 계층에 대한 참조를 가지며, 그리고 일반적으로 관례적인 상호작용 수단을 이용해 하위 계층의 구성요소를 직접적으로 사용하거나 조작할 수 있다. 그러나 하위 수준의 객체가 상위 수준이 객체와 소통해야 할 경우에는 또 다른 메커니즘이 필요한데, 이 경우 callback이나 observer 패턴 처럼 계층간에 관계를 맺어주는 아키텍처 패턴을 활용할 수 있다.

아키텍처 프레임워크

일부 기술적인 문제에는 더욱 침습적인 형태의 인프라스트럭처가 필요하다. 종종 다른 계층이 매우 특수한 방식으로 구현되기를 요구하는데, 이를테면 프레임워크 클래스의 하위 클래스가 되야 한다거나 일정한 메서드 서명을 지정해야 한다는 것이 여기에 해당한다. 가장 바람직한 아키텍처 프레임워크라면 도메인 개발자가 모델을 표현하는 것에만 집중하게 해서 복잡한 기술적 난제를 해결한다.

- Spring?

- Spring?

도메인 계층은 모델이 살가는곳

도메인 모델은 일련의 개념을 모아놓은 것이다. 도메인 계층은 그러한 모델과 설계 요소에 직접적으로 관계돼 있는 모든 것들을 명시한 것이다. 도메인 계층은 업무 로직에 대한 설계와 구현으로 구성된다. MODEL-DRIVEN DESIGN에서는 도메인 계층의 소프트웨어 구성물이 모델 개념을 반영한다.
도메인 로직이 프로그램상의 다른 관심사와 섞여 있다면 그와 같은 대응을 달성하기가 수월하지 않다. 따라서 도메인 주도 설계의 전제 조건은 도메인 구현을 격리하는 것이다.

SMART UI(지능형 UI) 안티패턴

수많은 소프트웨어 프로젝트에서는 START UI라고 하는 설계 접근법을 취하는데,
도메인 주도 설계 접근법과는 서로 양립할 수 없는 상호배타적인 길에 놓인 접근법이다.
장점 :

- 애플리케이션이 단순한 경우 생산성이 높고 효과가 즉각적으로 나타난다.
- 다소 능력이 부족한 개발자도 약간의 교육으로 이러한 방식으로 업무를 진행할 수 있다.
- 요구사항 분석 단계에서 결함이 발생하더라도 사용자에게 프로토타입을 배포한 후 요구에 맞게 제품을 변경해서 문제를 해결할 수 있다.
- 애플리케이션이 서로 분리되므로 규모가 작은 모듈의 납기 일정을 비교적 정확하게 계획할 수 있다. 부가적이고 간단한 작업만으로도 시스템을 확장하기가 수월할 수 있다.
- 관계형 데이터베이스와 잘 어울리고 데이터 수준의 통합이 가능하다.
- 4세대 언어 도구와 잘 어울린다.
- 애플리케이션을 인도했을 때 유지보수 프로그래머가 이해하지 못하는 부분을 신속하게 재작업 할 수 있다.

단점

난점 :

- 데이터베이스를 이용하는 방식 말고는 여러 애플리케이션을 통합하기가 수월하지 않다.
- 행위를 재사용하지 않으며 업무 문제에 대한 추상화가 이뤄지지 않는다. 업무 규칙이 적용되는 연산마다 업무 규칙이 중복된다.
- 신속한 프로토타입 작성과 반복주기가 SMART UI가 지닌 태생적인 한계에 도달하게 된다.
- 애플리케이션의 성장 경로가 단순 응용으로만 향한다. 우아한 방법으로 풍부한 행위를 갖출 수 없다.