유효정_3장~4장

3장 모델 주도 설계

모델을 코드로 어떻게 변환할 것인가?

- 좋은 모델을 코드로 적절하게 바꾸는 작업은 중요하다
- 소프트웨어 분석가와 비즈니스 도메인 전문가가 아무리 좋은 모델을 개발자에게 전달 하였더라도 개발이 계속 되면서 모델의 가치가 떨어질 수 있음을 주의하라.
- 분석적으로 정확한 모델이더라도 소프트웨어 설계 원칙을 깨뜨려야 하는 모델이라면 좋지 않다.
- 쉽고 정확하게 코드로 변환할 수 있는 모델을 선택해야 한다.

분석 모델과 코드 설계를 분리하는 방식 → 👎

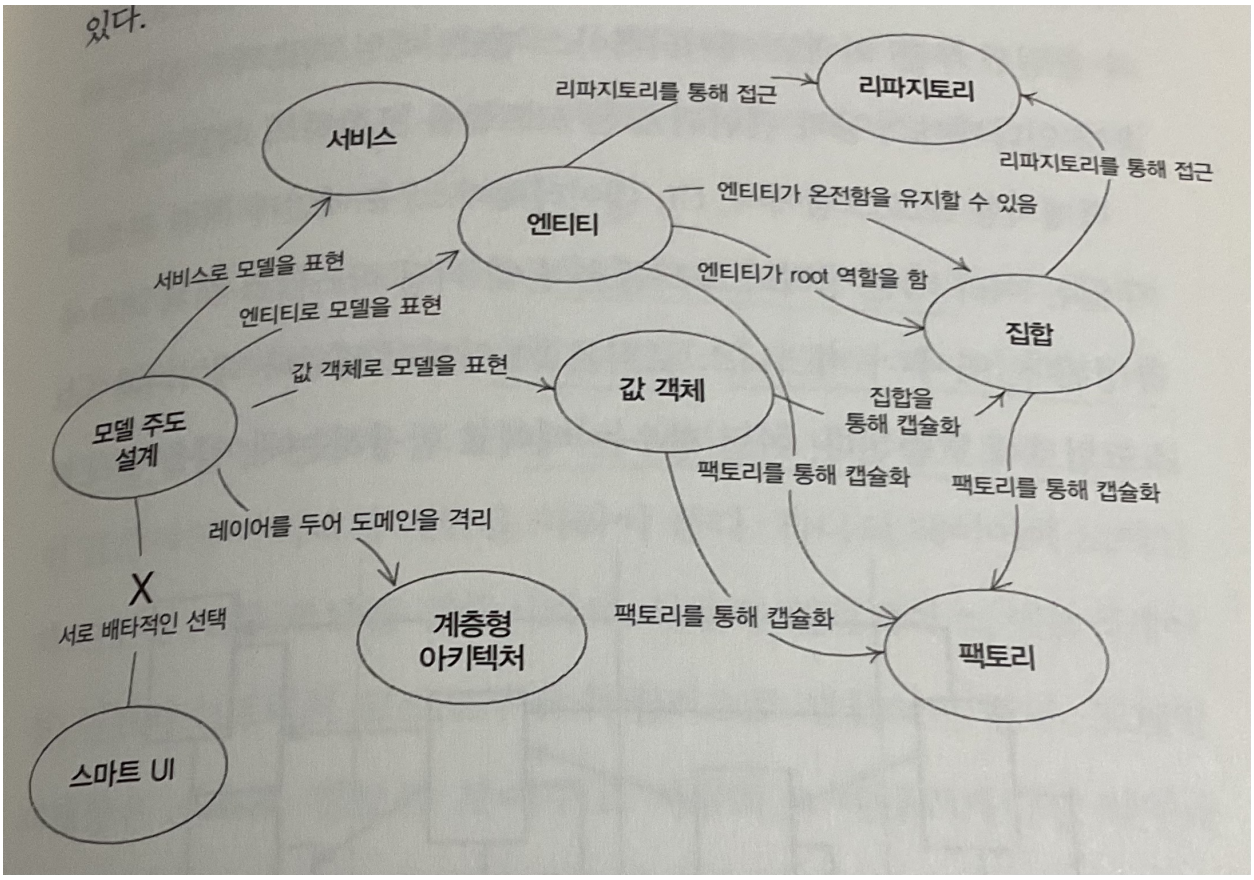
- 분석과 코드 설계를 서로 다른 사람이 작업하도록 함으로써 설계와 코드 구현을 분리함.
- 분석 모델은 비즈니스 도메인 분석의 결과물일 뿐, 소프트웨어 구현은 염두하지 않는 방식.
- 개발 과정 속에서 분석 모델이 점차 변질될 가능성이 커서 좋은 방식은 아니다.
    - 개발자들은 분석 모델에 적응하거나 아니면 모델과 별개의 설계를 하게 된다.
    - 모델과 코드 사이 매핑 관계가 존재하지 않게 된다.

    - 이는 분석가들이 분석 모델에 내재된 결함이나 도메인 자체의 복잡성을 미리 알 수 없기에 코드 구현에 문제가 생기는 경우 스스로의 의지에 따라 어떤 결정을 내려야 할 수 있다. 이 과정에서 모델과 점점 맞지 않는 구현으로 변질될 가능성이 있다.

도메인 모델링과 설계를 밀접하게 관련시키는 방식 → 👍

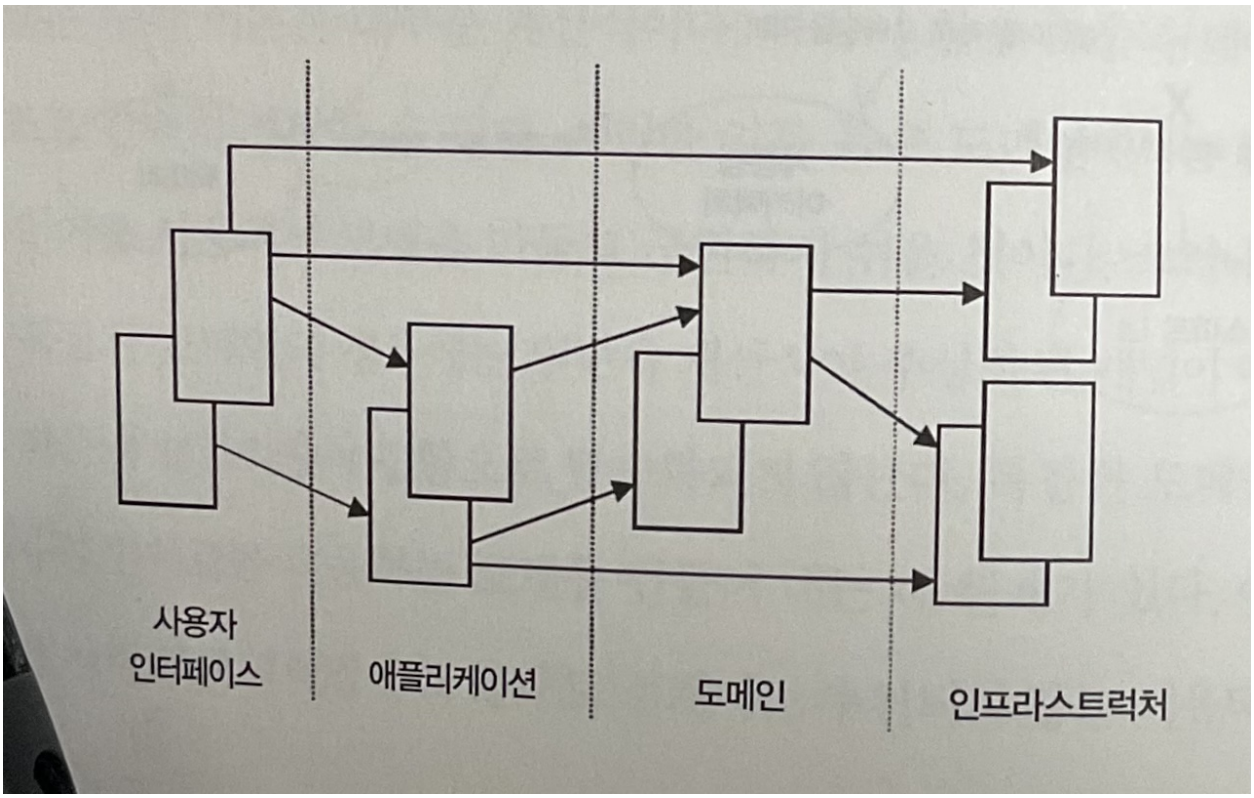
- 모델이란 소프트웨어 그 자체와 설계 고려 사항에 초점을 맞추어 만들어져야 한다.
- 설계 작업이 직관적이면서 모델을 기반으로 수행되도록 소프트웨어로 정확하게 표현할 수 있는 모델을 선택하자.
- 개발자들도 모델링 프로세스에 함께 참여해야 한다. 개발자들도 피드백 제공자로 동참시킨다.
    - 모델이 소프트웨어로 구현될 수 있음을 보장하도록 하자.
    - 개발자들은 모델을 아주 잘 알고 있어야 하고, 모델의 무결성에 대해 책임감을 느껴야 한다.
- 분석가들도 구현 프로세스에 함께 참여해야 한다.
    - 분석가들이 구현 프로세스에서 분리된다면 모델은 비현실적인 것이 된다.
- 코드의 변경 ↔ 모델의 변경
- 객체지향 프로그래밍은 구현과 모델이 같은 패러다임에 기초하고 있기 때문에 모델을 구현하기 적합하다.

모델 주도 설계를 위한 블록



도메인 주도 설계 관점에서의, 객체 모델링 및 SW 설계의 핵심 요소들

계층형 아키텍쳐



- 응용 시스템은 도메인 관련 부분 이외에 DB 접근, 파일/네트워크 접근, UI 등의 많은 코드를 포함하고 있다.
- 레이어가 분리되어 있지 않다면
  - 응집성 있는 구현이 힘들다
  - 모델 기반 객체가 현실성을 잃는다.
  - 자동화된 테스트가 어려워진다.
  - 코드를 이해하기 어렵다.
- 레이어 분리를 잘 해야 한다.
  - 응집도를 높여야 하며
  - 상위 레이어에 대한 결합도를 낮추고, 자기 하위 레이어에만 의존해야 한다.
  - 예를 들어 도메인 객체들은 스스로 정보를 보여주고 저장하고 애플리케이션 작업을 관리하는 일을 하면 안되고, 도메인 모델 자체를 표현하는데 집중해야 한다.

도메인 주도 설계의 설계 레이어

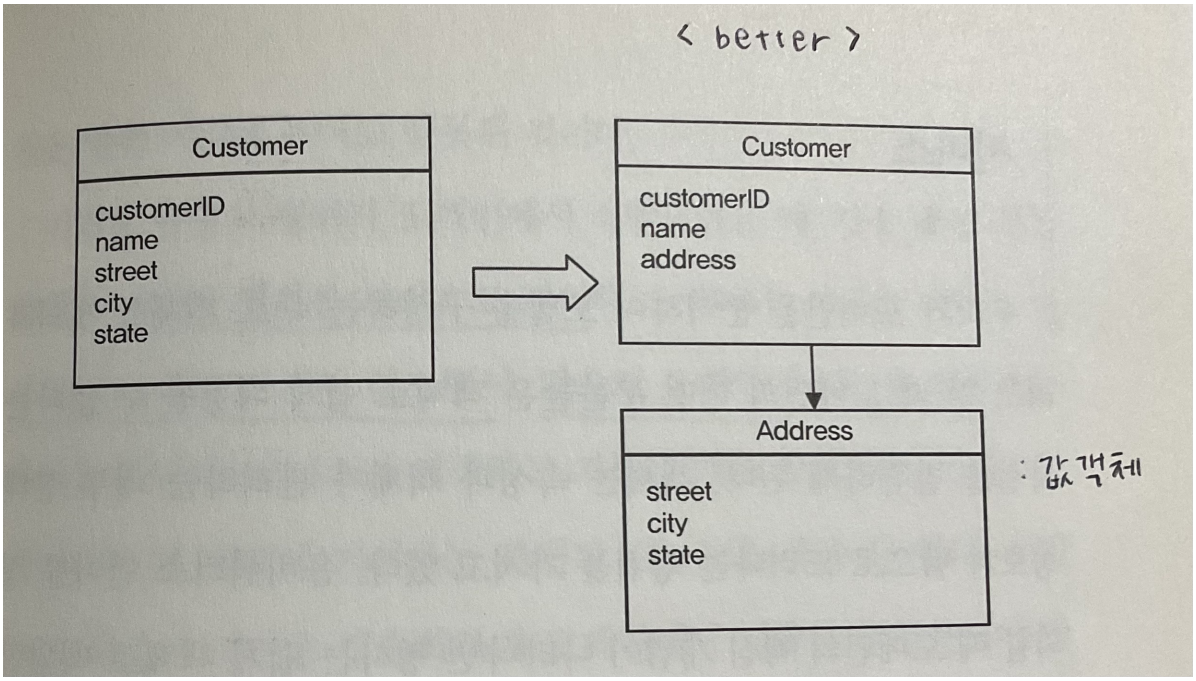
Aa Name	Tags	Column
사용자 인터페이스 (Presentation Layer)	사용자에게 정보를 보여주고 사용자의 명령을 해석하는 책임을 진다.	1. 비행기 편을 예약하고자 하는 사용자의 요청
애플리케이션 레이어	애플리케이션 활동을 조율한다. 업무 로직은 포함하지 않는다. 객체의 상태는 보관하지 않지만, 애플리케이션 작업의 처리 상태는 보관한다.	2. 사용자의 요청을 받는다 6. 도메인 객체들의 정보를 인프라 스트럭쳐에 반영한다.
도메인 레이어	도메인 정보를 포함한다. 비즈니스 객체의 상태를 포함한다. 비즈니스 객체와 이 객체의 상태 정보 중 가능한 부분의 영속성에 대한 책임은 인프라 스트럭처 레이어로 위임된다.	4. 이미 예약된 자리 이외의 여유 좌석이 있는지 등을 알아보는 메서드를 호출하여 검증을 거친다. 5. 상태를 확약으로 변경한다.
인프라 스트럭쳐 레이어	다른 레이어 모두를 지원하는 라이브러리 역할. 레이어 간 통신을 제공, 비즈니스 객체의 영속성을 구현, UI 레이어의 라이브러리를 포함한다.	3. 관련 도메인 객체를 찾아낸다. 7. 도메인 객체들의 정보를 영구히 저장한다.
Untitled		

엔티티와 값 객체

- 엔티티
  - 소프트웨어가 여러 상태를 거치는 동안 동일한 값을 유지하는 식별자를 지닌 유형의 객체
  - 속성 값을 이용해서 두 객체의 일치 여부를 판단하는 요구사항을 경계해야 한다.

- 값 객체
  - 모든 객체를 엔티티로 만들지 않아도 된다.
  - 추적 가능한 객체(영속 되어야 하는 객체)는 엔티티여야 한다.
  - 하나의 객체가 도메인의 어떤 측면을 표현하는데 사용되지만 식별자가 없는 경우 값 객체이다.
  - 값 객체는 쉽게 생성되고 폐기 가능하다.

- 값 객체는 공유가 가능하기에 수정할 수 없게 만들어야 한다.



Address 의 street, city, state 속성은 개념적으로 각각 분리된 정보가 아니며, 함께 존재해야 완전한 의미를 가지는 값 객체이다.

서비스

- 서비스에 의해 수행되는 오퍼레이션은 일반적으로 엔티티 또는 값 객체에 속할 수 없는 도메인의 개념을 나타낸다.
- 수행되는 오퍼레이션은 도메인의 다른 객체를 참조한다.
- 오퍼레이션은 상태를 저장하지 않는다.
- Ex) 한 계좌에서 다른 계좌로 돈을 보내는 경우

모듈

- 모듈화? 관련된 개념과 작업을 조직화하여 복잡도를 감소시키는 기법
- 대규모 모델을 이해할 때, 모델에 속해있는 모듈들 간의 상호작용을 먼저 이해하고, 모듈 하나하나의 내부를 파악하는 것이 효과적이다.

- 응집도를 높이고 결합도는 낮추는 방향으로 모듈을 사용해야 한다.
  - 모듈은 다른 모듈들이 접근할 수 있는 잘 정의된 인터페이스를 가져야 한다.
  - 여러개의 객체에 접근하는 것보다 하나의 인터페이스에 접근하는 것이 더 좋다.
- 모듈을 최종 확정시키지 말고 프로젝트와 함께 진화시켜 나가는 방식을 권장한다.

도메인 객체의 생명주기를 관리하는 패턴

- 집합

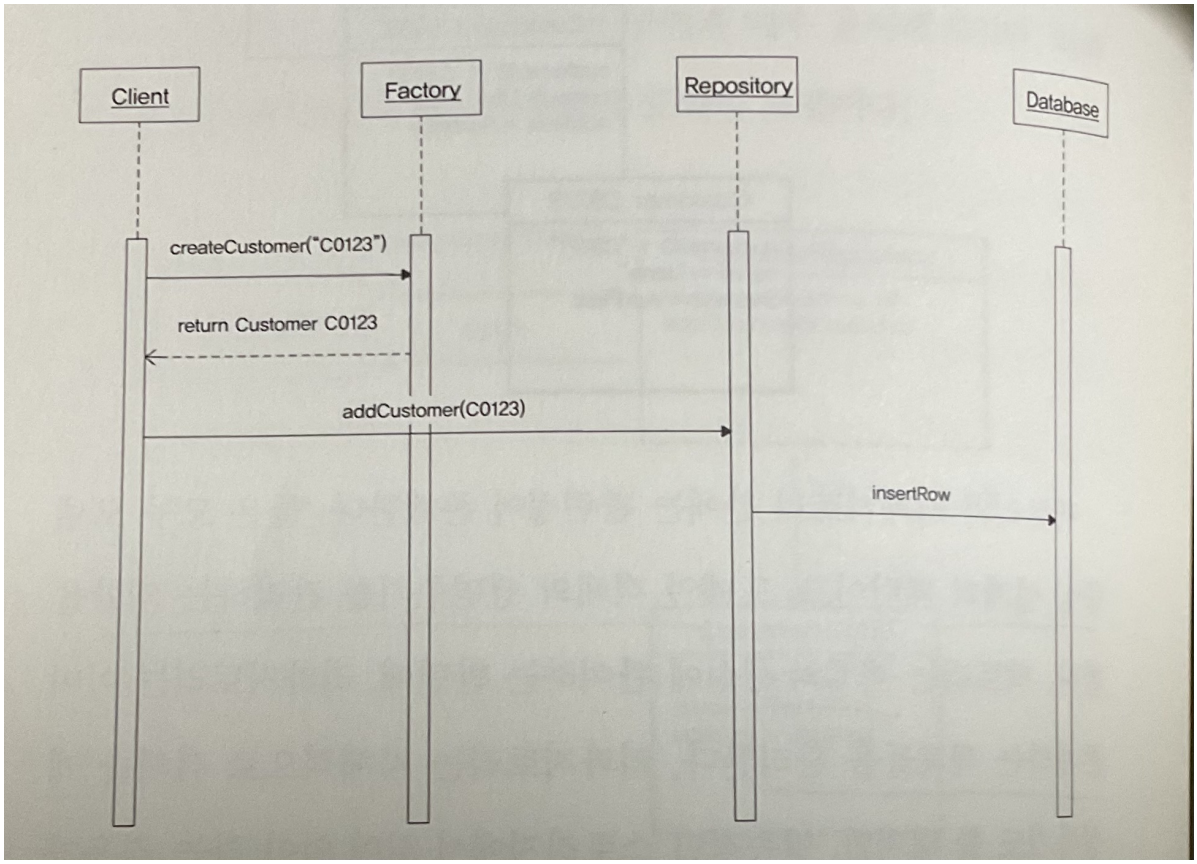
  객체의 소유권과 경계를 정의하는데 사용되는 패턴
- 팩토리, 리파지토리

  객체의 생성과 저장을 도와주기 위한 설계 패턴

1. 집합
   - 모델에서 추적 가능한 엔티티들끼리 관계가 있을 때 이를 구현하기 위한 소프트웨어적 메커니즘이 존재해야 한다.
   - 모델은 단순하고 이해하기 쉽게 설계 되어야 한다.
   - 양방향 다대다 관계가 특히 복잡하다
     - 모델의 핵심 사항이 아닌 관계가 있다면 제거한다.
     - 다수성의 숫자는 제약사항을 추가하여 감소시킨다.
     - 단방향 관계로 대체 가능한지 고려해 봐라.
   - 객체들끼리 관계가 복잡하면
     - 무결성을 유지해야 하며, 이 과정에서 성능 저하를 초래할 수 있다.
     - 불변식을 따르도록 강제해야 하는데, 수 많은 객체가 수정된 데이터 객체를 참조하고 있을 때 이 불변식을 따르고 있는지 잘 검증하는 것은 매우 어렵다.
     - 객체가 일관성을 유지하는 것을 보장하는 것도 어렵다.
   - 따라서 하나의 객체의 외부와 내부를 가르는 경계를 구분하는 집합을 사용해라!
     - 엔티티와 값 객체를 집합 내부에 포함시키고

    - 집합 간 경계를 설정한다.
    - 엔티티를 하나 선정하여 각 집합의 root 로 삼고
    - 경계 내부의 객체들은 root 를 통해서만 접근 하게끔 한다.
    - 외부 객체들은 root 만을 참조해야 한다.
- 이러한 방식의 제어권 할당은 집합 내부 객체들 및 전체 집합 자체가 어떠한 상태 변화에서도 전체적으로 불변식을 보장할 수 있다.

2. 팩토리와 리파지토리



- 둘 다
    - 도메인 객체의 생명주기를 관리한다.
- 팩토리는
    - 객체의 생성에 관여한다.

        - 생성자로 생성되기 복잡할 때는 팩토리가 객체를 생성한다.
        - 순수 도메인
    - 리파지토리는
        - 아무것도 없는 상태에서 객체를 생성하는 것이 아니라 이미 존재하는 객체들을 복원하여 관리한다.
        - 캐싱 or 영속 스토리지 이용
        - 인프라 스트럭쳐와 이어지는 연결을 포함할 수 있다. (예를 들면 데이터베이스)

4장 깊은 통찰을 향한 리팩터링

지속적인 리팩터링

- 애플리케이션의 기능에 변화를 주지 않고 코드를 더 좋게 만들기 위해 재설계하는 절차
- 주의할 점
    - 작은 규모로
    - 제어 가능한 절차를 적용하면서
    - 기존에 정상 작동 하던 기능을 손상시키지 않고
    - 새로운 사이드 이펙트 버그를 만들지 않아야 한다.
- 자동화 테스트를 사용하여 정상적으로 리팩토링이 되고 있음을 보장해라.
- 설계가 유연해야만 리팩토링이 가능하다.
- DDD 관점에서 리팩토링이란, 도메인에 대한 통찰이나 모델이나 코드에 드러나는 표현을 상응하게 정련하기 우해 수행할 수도 있다.