4. 개념적 모델링 - 실습

#0.강의/2.데이터베이스로드맵/3.설계1

실전 요구 사항 분석

실습 시간은 지금까지 배운 내용을 기반으로 실제 프로젝트를 진행할 것이다.
각각의 모델링 과정마다 이전에 진행한 실습을 이어갈 것이다.

- 개념적 모델링 학습 ➜ 개념적 모델링 실습
- 논리적 모델링 ➜ 논리적 모델링 실습
- 물리적 모델링 ➜ 물리적 모델링 실습

시나리오: 당신은 성장 잠재력이 큰 스타트업에 합류했다. 이 스타트업은 오프라인에서 성공적으로 자신의 쇼핑몰을 운영하며 탄탄한 고객층을 확보했다. 이제 사업 확장을 위해 온라인 쇼핑몰을 구축하려 한다. 당신은 이 프로젝트의 핵심 개발자로 참여하여, 데이터베이스 설계는 물론이고 팀의 개발자들과 함께 서비스를 만들어야 한다.

이론은 충분히 배웠다. 이제 진짜 실전이다. 그런데 실전은 우리가 배운 이론처럼 순탄하게 흘러가지 않는다.

문제 상황: 폭풍처럼 쏟아지는 요구 사항

프로젝트의 시작과 함께 기획팀과 사업팀에서 온라인 쇼핑몰에 필요한 기능 명세서를 전달했다. 그런데 그 양이 어마어마하다.

[쇼핑몰 초기 기능 명세서]

1. **회원 기능**: 가입, 탈퇴, 여러 배송지 관리, 기본 배송지 설정 등
2. **판매자 기능**
    - 누구나 우리 쇼핑몰에 입점해서 제품을 판매할 수 있는 오픈마켓이다.
    - 외부 판매자 입점, 판매자별 상품 등록 및 관리, 판매자 정보 관리(상호명, 사업자번호 등), 판매자 정산 등
3. **상품 및 카테고리 기능**: 상품 등록, 카테고리 관리, 가격/재고 변경 등

4. **장바구니 기능**: 상품 담기, 수량 조절 등
5. **주문 기능**: 장바구니에 담긴 여러 상품을 한 번에 주문, 배송지 선택, 쿠폰 적용, 주문 상태 관리 등
6. **결제 기능**: 신용카드, 계좌이체 등 다양한 결제 수단 연동
7. **배송 기능**: 배송 상태 추적(준비 중, 배송 중, 완료), 운송장 번호 관리
8. **리뷰 기능**: 구매 완료 상품에 대한 별점 및 리뷰 작성
9. ... 등등

이 모든 기능을 보니 눈앞이 캄캄해진다. 당신은 이 많은 기능을 보고 직감한다. '이걸 한 번에 다 만드는 건 현실적으로 일정 안에 불가능하다.'
당신과 팀은 이 많은 기능을 짧은 일정 안에 모두 개발해서 서비스를 출시해야 한다. 여기서 개발자는 두 가지 선택의 기로에 놓인다.

1. 기술만 생각하는 개발자의 접근법
이 개발자는 주어진 모든 요구 사항을 그대로 받아들여 개발 일정을 산정하기 시작한다. 모든 기능을 구현하려니 할 일이 태산이다. 회원, 상품, 주문은 물론이고 외부 판매자 입점 시스템, 복잡한 쿠폰 정책, 리뷰 시스템까지... 데이터베이스 설계부터 만만치 않다.

꼼꼼하게 일정을 산정해보니, 약 **6개월**이라는 시간이 걸린다는 결론이 나왔다.

6개월 정도 걸린다는 결과를 사업팀에 전달하자 강한 반발이 있었다. 사업팀은 당장 **3개월 뒤**에는 서비스를 출시해서 온라인 시장에 진입할 것으로 기대하고 있었기 때문이다. "6개월이나 걸린다고? 우리 경쟁사는 이미 저만치 앞서가고 있는데!" 라는 반응이 돌아온다. 결국 이 소식은 사장님 귀에까지 들어가고, 개발팀은 신뢰를 잃게 된다. 프로젝트는 시작부터 삐걱거린다.

2. 전체를 보는 개발자의 접근법
훌륭한 개발자는 단순히 주어진 기능을 '어떻게' 구현할지에만 집중하지 않는다. 그보다 먼저 이 기능을 '왜' 만들어야 하는지 질문해야 한다.

당신은 기술의 구현에만 집중하는 대신, "우리 사업의 성공을 위해, 이 모든 기능이 정말 지금 당장 필요한가? 지금 시점에 가장 중요한 것은 무엇이고, 덜 중요한 것은 무엇일까?"라는 근본적인 질문을 던져야한다. 이것이 바로 성공적인 프로젝트의 시작이다.

단순히 기술 구현에만 매몰되는 것이 아니라, 비즈니스의 성공이라는 더 큰 그림을 보는 것이다. 이것이 바로 우리가 지향해야 할 개발자의 모습이다.

당신은 기획자를 찾아가 정말 이 모든 기능이 서비스 첫 출시에 필수적인지 묻는다. 기획자는 "사업팀에서 모두 필요하

다고 했다"라고 답한다. 당신은 기획자, 사업 담당자를 모두 불러 함께 커피를 마시며 편안한 분위기에서 이야기를 시작한다.

> **당신**: "요구 사항을 보니 기능이 정말 탄탄하네요. (먼저 그들의 노력을 인정하자)
> 제가 알기로는 우선 우리 쇼핑몰의 자체 상품만 판매하는 것으로 알고 있었는데, '외부 판매자 입점' 기능이 있어서 조금 놀랐습니다. 이 기능이 지금 당장 필요한가요?"

사업팀 담당자와 자세히 이야기를 나누어보니, '외부 판매자 입점' 기능은 사업팀에서도 지금 당장 진행하기는 어렵다고 한다.
외부 판매자를 모집하려면 사업 전략 수정, 홍보, 영업 등 상당한 준비 기간이 필요하기 때문이다.
사업팀은 이 기능을 쇼핑몰이 성공적으로 자리 잡은 이후, 빠르면 1~2년 뒤에 시작하고 싶어 한다.

또한, 사업팀 담당자는 미래에 필요할 것 같은 기능까지 모두 처음부터 정리해서 전달해야 하는 줄 알고 있었다.

당신은 솔직하게 현실을 이야기한다.

> **당신**: "솔직하게 말씀드리면, 이 모든 기능을 지금 개발하려면 최소 6개월이 걸립니다. 하지만 우리가 정말 살아남기 위해 필요한 핵심 기능에만 집중한다면, **2개월 안에 출시**할 수 있습니다. 먼저 시장에 우리 제품을 선보이고, 고객 반응을 보면서 다음 기능을 하나씩 붙여 나가는 것이 어떨까요?"

이 제안에 사업팀과 기획팀의 눈이 빛난다. 그들도 빠른 출시를 간절히 원했기 때문이다. 이제 모두가 한 팀이 되어 '함께 핵심 기능을 찾으며 무엇을 뺄 것인가'를 논의하기 시작한다.

실전 개념적 모델링 - 시작

1단계: 핵심 요구 사항 다시 정의하기 (MVP)

우리는 이 과정을 **MVP(Minimum Viable Product, 최소 기능 제품)**를 정의하는 과정이라고 부른다. 말 그대로, 시장에서 살아남기 위해 필요한 최소한의 기능을 정의하는 것이다.

논의 끝에, 우리는 다음과 같이 1차 오픈 스펙을 정했다.

- **외부 판매자 기능**: 당연히 제외한다. 1차 목표는 우리 회사 상품을 온라인으로 판매하는 것이다.
- **쿠폰, 리뷰, 복잡한 카테고리**: 일단 없어도 주문은 가능하다. 2차 오픈 기능으로 미룬다.

- **장바구니 기능**: 매우 중요하지만, 굳이 데이터베이스에 저장할 필요는 없다. 우선은 애플리케이션이나 클라이언트(웹 브라우저) 측에 임시 보관하도록 구현할 수 있다. 데이터베이스 설계에서는 일단 제외한다.

그 결과, 정말 꼭 필요한 핵심 기능은 다음 기능으로 압축되었다.

[쇼핑몰 MVP 기능 명세서]

1. **회원**: 고객이 가입하고 자신의 정보를 관리할 수 있어야 한다.
2. **상품**: 우리가 판매할 상품을 등록하고 관리할 수 있어야 한다.
3. **주문**: 회원이 상품을 구매할 수 있어야 한다.
4. **결제**: 주문에 대한 결제 정보를 기록하고 관리할 수 있어야 한다.
5. **배송**: '결제가 완료된' 주문의 배송 상태를 관리할 수 있어야 한다.

이제 훨씬 마음이 편안해졌다. 이 핵심 기능을 중심으로 데이터베이스 설계를 시작해보자.

2단계: 핵심 엔티티 도출

이제 단순해진 요구 사항에서 데이터의 뼈대, 즉 엔티티를 찾아보자.

- `회원(Member)` : 서비스를 사용하는 고객.
- `상품(Product)` : 판매의 대상이 되는 물건.
- `주문(Order)` : 회원의 구매 활동 결과.
- `결제(Payment)` : 주문에 대한 지불 정보.
- `배송(Delivery)` : 주문된 상품의 물리적 이동에 대한 정보.

도출된 핵심 엔티티: `회원`, `상품`, `주문`, `결제`, `배송`

여기서 `주문`과 `배송`을 별도의 엔티티로 분리한 것에 주목하자. 주문은 '결제'까지 포함하는 비즈니스 트랜잭션의 단위이고, 배송은 '물류'라는 별개의 프로세스다. 이렇게 역할을 분리하면 나중에 "주문은 완료되었지만 배송은 시작 전"과 같은 다양한 상태를 명확하게 관리할 수 있고, 각자의 책임이 분명해져 시스템을 유지보수하기 쉬워진다.

3단계: 속성 정의 및 관계 설정

이제 각 엔티티의 세부 정보(속성)를 정의하고, 엔티티 간의 관계를 설정할 차례다.

- `회원(Member)` : 회원id, 로그인id, 비밀번호, 회원명, 이메일, 주소

- `상품(Product)` : 상품id, 상품명, 상품 가격, 재고 수량
- `주문(Order)` : 주문id, 주문 상태, 배송지 주소, 주문 일시
- `결제(Payment)` : 결제id, 결제 수단, 결제 금액, 결제 상태, 결제 일시
- `배송(Delivery)` : 배송id, 배송 상태, 운송장 번호

> **식별자(기본 키) 전략**
> 이번 설계에서 식별자는 단순화와 일관성을 위해 엔티티명 + id를 사용하겠다.
>
> 그런데 여기서 하나의 의문점이 생긴다. 회원의 경우 `회원id`와 `로그인id`가 별도로 존재한다. `회원id`는 회원을 저장할 때 마다 데이터베이스나 시스템이 절대 중복되지 않도록 임의로 만들어주는 식별자 값이다. 예를 들면 값이 하나씩 증가하는 자동 증가(Auto Increment) 같은 값을 사용하는 것이다. `로그인id`는 사용자가 회원가입시에 입력한 로그인 하는데 필요한 id이다.
> 물론 여기서 `로그인id`, `이메일`은 절대로 중복되지 않기 때문에 `회원id`를 제거하고, 대신에 `로그인id`나 `이메일` 속성을 식별자로 사용해도 된다.
>
> 그럼에도 불구하고 왜 `회원id` 같은 임의의 id 값을 만들어서 식별자로 사용하는 것일까?
> 식별자(기본 키)를 선택하는 전략은 데이터베이스 설계에서 가장 중요한 부분 중 하나이다.
> 이 부분은 너무 중요하기 때문에 뒤의 논리적 모델링 단계에서 매우 자세히 설명하겠다.

이제 관계를 살펴보자.

- `회원`과 `주문`의 관계는? ➜ 한 명의 `회원`은 여러 번 `주문`할 수 있다. **(1:N 관계)**
- `주문`과 `결제`의 관계는? ➜ 하나의 `주문`에 대해 하나의 `결제`가 이루어진다. **(1:1 관계)**
- `주문`과 `배송`의 관계는? ➜ 하나의 `주문`에 대해 하나의 `배송`이 이루어진다. **(1:1 관계)**
- `주문`과 `상품`의 관계는? ➜ 하나의 `주문`에는 여러 `상품`이 포함될 수 있다. 반대로 하나의 `상품`도 여러 `주문`에 포함될 수 있다. **(M:N 관계)**

4단계: M:N 관계 해소와 '연관 엔티티'

여기서 가장 중요한 M:N 관계, 즉 `주문`과 `상품`의 관계를 찾았다.
앞서 연관 엔티티에서 배웠듯이 이런 다대다 관계는 다음과 같은 문제가 있다.

- **문제 1**: M:N 관계는 물리적으로 구현할 수 없다
- **문제 2**: 관계에 속한 데이터를 저장할 장소가 없다

이런 문제를 해결하기 위해 연관 엔티티를 도입해야 한다. 그리고 이런 다대다 관계를 만나면 반드시 다음 질문을 해야 한다.

"두 엔티티의 관계 속에서만 의미를 가지는 속성이 있는가?"

어떤 회원이 'A상품'과 'B상품'을 하나의 주문서로 주문했다고 생각해보자.
이때, 'A상품을 몇 개 주문했는가?'(`주문 수량`), 'A상품을 얼마에 주문했는가?'(`주문 당시 가격`)라는 정보가 필요하다.

- `주문 수량`: 이 정보는 `주문` 엔티티에 속할까? 아니다. 주문 하나에는 상품이 여러 개일 수 있으니, 특정 상품의 수량을 저장할 수 없다. `상품` 엔티티는 더더욱 아니다.
- `주문 당시 가격`: `상품` 엔티티에는 현재 `상품 가격` 정보가 있지만, 상품 가격은 언제든 바뀔 수 있다. 3일 전에 10,000원에 주문한 내역이 오늘 가격이 12,000원으로 올랐다고 해서 바뀌면 안 된다. 즉, 주문이 일어난 '시점'의 가격을 어딘가에 저장해야 한다.

바로 이 `주문 수량`과 `주문 당시 가격`은 `주문`과 `상품`의 **관계 속에서만** 의미를 갖는 데이터다.

따라서 우리는 이 M:N 관계를 해소하고, 이 추가적인 데이터를 담을 새로운 테이블, 즉 **연관 엔티티**가 필요하다. 우리는 이것을 `주문 항목(order_item)` 이라고 부르겠다.

이렇게 주문 항목 연관 엔티티를 도입하면, 복잡했던 M:N 관계는 두 개의 단순한 1:N 관계로 해결된다.

- `주문`과 `주문 항목`의 관계 (1:N)
  - 하나의 `주문`은 여러 개의 `주문 항목`을 가질 수 있다.
  - 하나의 `주문 항목`은 반드시 하나의 `주문`에 속한다.
- `상품`과 `주문 항목`의 관계 (1:N)
  - 하나의 `상품`은 여러 `주문 항목`에 포함될 수 있다.
  - 하나의 `주문 항목`은 반드시 하나의 `상품`을 가진다.

정리하면 다음과 같다.

- `주문` (1) - (N) `주문 항목` (N) - (1) `상품`

이렇게 주문 항목(`order_item`) 연관 엔티티를 도입하면, 복잡했던 M:N 관계는 단순한 일대다(1:N), 다대일(N:1) 관계로 해결된다.

참고: 연관 엔티티 추천 이름 및 특징

주문 항목(`order_item`)은 주문(`order`)과 상품(`product`)의 관계에서 나온 연관 엔티티이다. 연관 엔티티의 이름을 지을 때는 크게 2가지 방법이 있다.

1. 연결 강조

- 주문 상품(**order_product**)
  - '주문'과 '상품' 테이블을 직접적으로 연결한다는 관계를 이름에 명시적으로 보여준다.
  - 매우 직관적이고 단순하여 관계를 파악하기 쉽다는 장점이 있다.

2. 의미 있는 이름

- 주문 항목(**order_item**)
  - '주문'에 포함된 '항목'이라는 의미를 가장 명확하게 전달한다.
  - 실제 쇼핑몰 비즈니스 로직(예: 장바구니 항목, 주문 항목)과 용어가 일치하여 이해하기 쉽다.

- 주문 상세(**order_detail**)
  - '주문의 상세 내역'이라는 의미로, `order_item`과 거의 동일한 의미로 널리 사용된다.
  - 주문에 속한 각 상품 정보를 상세히 나타낸다는 점을 강조한다.

실무에서는 의미 있는 이름을 사용하는 것이 좋다. 데이터베이스는 단순히 데이터를 저장하는 창고가 아니라 비즈니스 모델을 반영하는 설계도이기 때문이다. 만약 관계가 단순하고 의미 있는 이름을 짓기 어렵다면 양쪽 엔티티의 이름을 사용하는 연결을 강조하는 방식을 선택할 수 있다.

실전 개념적 모델링 - ERD 작성

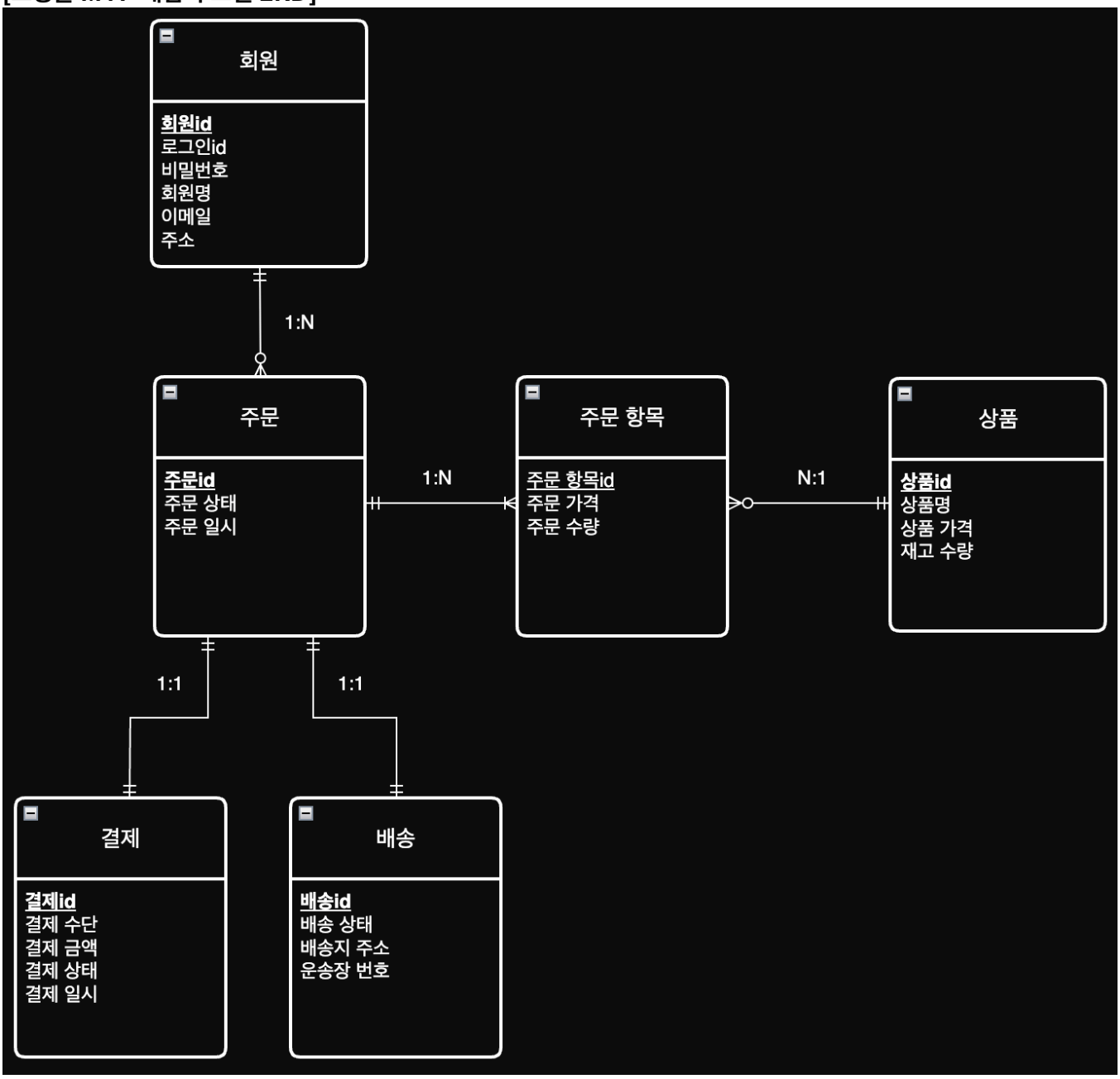
5단계: 개념적 모델 ERD 작성

지금까지 분석한 MVP 요구 사항을 바탕으로 개념적 모델 ERD를 그려보자.

여기서는 draw.io라는 무료 툴을 사용하겠다. 참고로 ER 다이어그램을 그릴 수 있는 어떤 툴을 사용해도 괜찮다.

> **draw.io**
> draw.io는 웹에서 무료로 다양한 다이어그램을 그릴 수 있는 툴이다.
> 사이트 경로: https://draw.io

[쇼핑몰 MVP 개념적 모델 ERD]



1. **회원과 주문 - 1:N 관계**: `회원`과 `주문`은 1:N 관계다. 한 명의 회원은 여러 주문을 할 수 있다.
    - 회원 ➜ 주문 (선택적 관계), 주문하지 않은 회원도 존재할 수 있다.
    - 주문 ➜ 회원 (필수적 관계), 주문할 때 반드시 회원이 존재해야 한다.
2. **주문과 결제 - 1:1 관계**: `주문`과 `결제`는 1:1 관계다. 하나의 주문은 반드시 하나의 결제를 가진다.
    - 주문 ➜ 결제 (필수적 관계), 주문시 결제가 필수다. 결제가 없는 임시 주문은 없다.
    - 결제 ➜ 주문 (필수적 관계)
3. **주문과 배송 - 1:1 관계**: `주문`과 `배송` 역시 1:1 관계다. 주문, 결제, 배송의 책임을 분리함으로써 모델이 훨씬 명확해졌다.
    - 주문 ➜ 배송 (필수적 관계)

- 배송 ➜ 주문 (필수적 관계)

4. **주문/상품과 주문 항목 - M:N 관계 해소**: `주문`과 `상품`은 `주문 항목`이라는 연관 엔티티를 통해 연결된다. `주문 항목`은 주문 당시의 `주문 가격`과 `주문 수량`처럼 관계 속에서만 발생하는 중요한 정보를 저장하는 엔티티이다.
   - 주문 ➜ 주문 항목 (필수적 관계)
   - 주문 항목 ➜ 주문 (필수적 관계)
   - 상품 ➜ 주문 항목 (선택적 관계), 주문되지 않은 상품도 존재한다.
   - 주문 항목 ➜ 상품 (필수적 관계)

> **카디널리티 표기**
> 까마귀 발 표기법을 통해 이미 카디널리티(1:N 등)를 표기했기 때문에 ERD에 1:N과 같은 카디널리티를 따로 표기하지는 않는다. 여기서는 학습의 이해를 돕기 위해 1:N과 같은 내용을 직접 표기했다.

> **개념적 모델링과 생략 - ERD**
> 개념적 모델링의 목표는 모든 것을 다 나타내는 것이 아니라, 기획, 개발등 모든 사람들의 이해를 돕는 것이 목적이다. 따라서 ERD라는 지도에 모든 내용을 표현하기 보다는 중요한 내용 위주로 표현하는 것이 좋다. 앞으로 모든 담당자와 이 지도로 이야기해야 한다. 지도가 너무 복잡하면 정말 중요한 핵심을 놓칠 수 있다.
>
> 예를 들어 `등록일` 속성은 거의 대부분의 엔티티에 필요하므로 개념적 모델링 단계에서는 단순화를 위해 생략하겠다.
>
> 물론 개념적 모델의 ERD에 표시하지는 않았지만, 문서에는 자세히 남겨두어야 한다.
> 꼭 하고 싶은 이야기는 개념적 모델링 단계에서는 ERD라는 형식에 너무 얽매이기 보다는 모두의 입장에서 쉽게 비즈니스를 이해할 수 있는게 중요하다.

개념적 모델링에서 외래 키를 생략한 이유

모델링 결과를 보면 외래 키(Foreign Key)가 생략되어 있다. 예를 들어 실제 테이블을 구현하려면 `주문` 테이블에 `회원 id`가 외래 키로 포함될 것이다. 그래야 어떤 회원이 주문했는지 알 수 있다.

개념적 모델링에서는 외래 키를 표시하지 않는 것이 원칙이다. 하지만 실무에서는 종종 처음부터 외래 키를 포함하기도 한다.

원칙: 특정 기술과 독립적인 설계
개념적 모델은 데이터의 구조와 관계를 큰 그림에서 이해하기 위한 설계도이다. 이 단계에서는 어떤 데이터베이스 기술

(예: 관계형 데이터베이스, NoSQL 등)을 사용할지 아직 결정하지 않았다고 가정한다.

- **외래 키의 정체**: 외래 키는 테이블 간의 관계를 연결하기 위해 사용하는 관계형 데이터베이스(RDBMS) 전용 기술이다.
- **개념적 모델의 목표**: 특정 기술에 얽매이지 않는, 순수한 데이터의 논리적 구조를 표현하는 것이다.
- **관계 표현 방식**: 따라서 개념적 모델에서는 외래 키 대신 **선(Line)**을 사용해 회원과 주문 같은 데이터(엔터티) 간의 관계를 표현한다.
- **예시**: 주문 엔티티에 `회원id`라는 외래 키를 넣는 대신, `회원`과 `주문`이라는 두 개의 상자를 그리고 그 사이를 선으로 연결하여 "회원이 주문을 한다"는 관계를 보여주는 식이다.

현실: 실용성을 고려한 접근
이론적으로는 위와 같지만, 실제 업무 환경은 조금 다르다.

- **예상된 기술**: 대부분의 프로젝트는 처음부터 관계형 데이터베이스 사용을 염두에 두고 모델링을 시작한다.
- **효율성**: 어차피 나중에 외래 키를 추가할 것이므로, 설계 초기 단계부터 외래 키를 포함하면 전체 개발 과정이 더 효율적일 수 있다.

따라서 실무에서는 개념적 모델링 단계임에도 불구하고 실용적인 관점에서 외래 키를 미리 포함하는 경우가 훨씬 많다. 더 나아가서 처음부터 개념적 모델링 + 논리적 모델링을 함께 진행하기도 한다.

정리
처음 기획서의 모든 기능을 담으려 했을 때보다 훨씬 단순하고 명확해졌다. 이처럼 데이터베이스 설계는 단순히 기술적인 작업이 아니라, 비즈니스 목표를 이해하고, 현실적인 제약 조건(시간, 인력)을 파악하며, 이해관계자들과 소통하고, 논리적 허점을 찾아 보완하는 **전략적인 과정**이다. **복잡한 문제일수록 작게 나누어 가장 핵심적인 것부터 해결하는 것이 성공하는 프로젝트의 비결**이다.

실전 개념적 모델링 - 용어 사전 작성

앞서 우리는 MVP 요구 사항을 분석하고, 핵심 엔티티를 도출하여 ERD로 개념적 모델을 완성했다. 이 과정에서 '회원', '주문', '상품'과 같은 여러 비즈니스 용어를 사용했다.

이제 이 개념적 모델을 기반으로, 앞으로의 논리적, 물리적 모델링과 실제 개발에서 사용할 용어를 명확하게 정의하는 **'용어 사전'**을 작성할 차례다. 이는 프로젝트의 일관성을 유지하고 모든 팀원이 같은 언어로 소통하기 위한 매우 중요한 과정이다.

우리가 만든 개념적 모델의 엔티티와 속성들을 하나씩 용어 사전에 등록해보자. 이 사전은 앞으로 우리가 만들 모든 테이블과 컬럼 이름의 기준이 될 것이다.

쇼핑몰 MVP 프로젝트 용어 사전

분류	명칭	전체 영문명	축약어	설명	관련 시스템 요소
엔티티	회원	`member`		서비스를 이용하는 고객. `member`, `customer` 등 유사 용어 대신 `member`로 통일한	`member` 테이블
	상품	`product`		판매하는 물건 또는 서비스.	`product` 테이블
	주문	`order`		회원의 상품 구매 요청 행위. `order`는 SQL 예약어이므로 테이블명은 `orders`를 사용한다.	`orders` 테이블
	주문 항목	`order_item`		하나의 주문에 포함된 개별 상품 정보. 주문과 상품의 M:N 관계를 해소하는 연관 엔티티.	`order_item` 테이블
	배송	`delivery`		주문된 상품의 물리적 이동 정보.	`delivery` 테이블
	결제	`payment`	`pay`	주문에 대한 지불 정보.	`payment` 테이블

주요 속성	식별자	`identifier`	`id`	데이터를 고유하게 식별하는 번호. `[엔티티명]_id` 형식 (예: `member_id`) 으로 사용한다.	`member_id`, `product_id`
	이름, 명	`name`		사람, 상품 등 대상을 지칭하는 명칭.	`member_name`, `product_name`
	가격	`price`		상품의 현재 판매가 또는 주문 시점의 가격. 문맥에 따라 명확히 구분한다.	`order_price`
	금액	`amount`		금액 예): 총 주문 금액	
	재고	`stock`		상품의 재고	재고 수량 (`stock_quantity`)
	수량	`quantity`		개수나 양. 재고 `quantity`	재고 수량 (`stock_quantity`) 주문 수량 (`order_quantity`)
	개수, 횟수	`count`		개별 항목을 하나씩 세는 행위, 이벤트 발생 횟수, 데이터의 총 개수, 인원수 등	방문 횟수(visit count) 클릭 수(click count) 직원 수(employee count)
	상태	`status`		주문, 배송 등 엔티티의 현재 상태를 나타내는 값.	`order_status`, `delivery_status`

	주소	`address`	`addr`	위치 정보. 회원의 기본 주소(`addr`), 주문의 배송지(`ship_addr`) 등으로 사용한다.	`member.addr`, `orders.ship_addr`
	비밀번호	`password`		로그인 시 사용하는 비밀번호.	`member.password`
	배송	`shipping`	`ship`	배송과 관련된 속성을 나타낼 때 접두사로 사용. (예: `ship_addr`)	`ship_addr`
	로그인	`login`		시스템에 접속하는 행위. (예: `login_id`)	`member.login_id`
	수단	`method`		예) 결제 수단	`pay_method`
	번호	`number`	`no`	번호, 예) 운송장 번호	`tracking_no`
	운송장 번호	`tracking_no`		배송 추적 번호	
행위/수식어	생성	`create`		데이터가 만들어진 시점을 나타낼 때 사용. (예: `created_at`)	`created_at`
	수정	`update`		데이터가 변경된 시점을 나타낼 때 사용. (예: `updated_at`)	`updated_at`

이제 우리는 이 용어 사전을 기준으로 논리적, 물리적 설계를 진행할 것이다. 만약 개발 중에 배송 수량이라는 새로운 컬럼이 필요하다면 바로 배송(`ship`)과 수량(`quantity`)을 조합해서 모든 팀원이 예측 가능한 `ship_quantity`라는 이름을 일관되게 사용할 수 있다.

> ✨ **실무 팁 - 용어 사전은 살아있는 문서다 (2번 강조!)**
> 용어 사전은 한 번 만들고 끝나는 문서가 아니다. 프로젝트가 진행되면서 새로운 용어가 추가되고 새로운 비즈니스 용어가 생겨날 때마다 **지속적으로 업데이트**되어야 한다. Confluence, Notion, Google Docs 등 팀원 모두가 쉽게 접근하고 편집할 수 있는 도구를 사용하여 '살아있는 문서'로 관리하는 것이 핵심이다. 잘 관리된 용어 사전 하나가 수십 장의 복잡한 설계 문서보다 더 큰 힘을 발휘할 때가 많다.
> 반드시! 용어 사전은 모두가 쉽게 접근 가능하고, 모두가 실시간으로 편하게 편집할 수 있는 툴을 사용해야 한다! 관리가 어려운 엑셀 파일로 관리하는 것은 추천하지 않는다. 이것은 용어 사전을 부패하게 만들 가능성이 높다.

정리

실전 요구 사항 분석

- 훌륭한 개발자는 기술 구현뿐만 아니라 '왜' 이 기능이 필요한지 질문하며 비즈니스의 큰 그림을 본다.
- 이해관계자(기획, 사업팀)와 소통하여 서비스 출시에 정말 필요한 핵심 기능을 선별하는 과정이 필수적이다.

실전 개념적 모델링 - 시작

- **MVP(최소 기능 제품) 정의**: 시장에서 살아남기 위해 꼭 필요한 최소한의 기능(회원, 상품, 주문, 결제, 배송)을 정의한다.
- **핵심 엔티티 도출**: MVP 요구 사항을 기반으로 데이터의 뼈대가 되는 엔티티(`회원`, `상품`, `주문`, `결제`, `배송`)를 도출한다.
- **속성 정의 및 관계 설정**: 각 엔티티의 속성을 정의하고 엔티티 간의 관계(1:1, 1:N, M:N)를 설정한다.
- **M:N 관계 해소**: `주문`과 `상품` 사이의 다대다(M:N) 관계는 물리적으로 구현할 수 없고, 관계에 종속된 데이터(`주문 수량`, `주문 당시 가격`)를 저장할 수 없는 문제가 있다.
- **연관 엔티티 도입**: `주문 항목(order_item)`이라는 연관 엔티티를 만들어 M:N 관계를 두 개의 1:N 관계로 해소한다.

실전 개념적 모델링 - ERD 작성

- **개념적 모델 ERD**: 분석한 엔티티와 관계를 바탕으로 ERD(개체-관계 다이어그램)를 작성하여 전체 데이터 구조를 시각화한다.
- **ERD의 목적**: 개념적 모델 ERD는 모든 담당자가 소통하기 위한 지도로, 너무 복잡하지 않게 핵심적인 내용 위주로 표현하는 것이 좋다.
- **외래 키(Foreign Key) 생략**: 원칙적으로 개념적 모델은 특정 기술(RDBMS)에 종속되지 않아야 하므로 외래 키

대신 '선'으로 관계를 표현한다.

- **실무적 접근**: 하지만 실무에서는 대부분 관계형 데이터베이스 사용을 전제하므로, 효율성을 위해 개념적 모델링 단계부터 외래 키를 포함하는 경우가 많다.

실전 개념적 모델링 - 용어 사전 작성

- **용어 사전의 중요성**: 프로젝트에서 사용될 엔티티와 속성 등의 용어를 명확히 정의하여 모든 팀원이 일관된 언어로 소통하게 한다.
- **작성 기준**: 개념적 모델의 엔티티와 속성을 기반으로 명칭, 전체 영문명, 축약어, 설명 등을 정리한다.
- **일관성 확보**: 용어 사전을 기준으로 테이블과 컬럼명을 정하면, `ship_quantity`처럼 누가 만들어도 예측 가능한 이름을 일관되게 사용할 수 있다.
- **살아있는 문서**: 용어 사전은 프로젝트가 진행됨에 따라 지속적으로 업데이트되어야 하며, 팀원 모두가 쉽게 접근하고 편집할 수 있는 도구(Confluence, Notion 등)로 관리해야 한다.