3. 개념적 모델링

#0.강의/2.데이터베이스로드맵/3.설계1

- /요구 사항 분석과 핵심 요소 식별
- /엔티티란?
- /엔티티 분류1
- /엔티티 분류2
- /속성과 식별자
- /카디널리티와 참여도
- /ERD 완성하기
- /연관 엔티티 - 다대다 관계 해결
- /용어 사전
- /정리

요구 사항 분석과 핵심 요소 식별

개념적 모델링 - 데이터 세상의 청사진 그리기

지난 시간에는 잘못된 설계가 불러오는 끔찍한 재앙들을 목격하고, 이를 막기 위한 3단계 설계 로드맵(개념적-논리적-물리적)을 확인했다. 이제 우리는 그 첫 번째 여정인 '개념적 모델링'을 시작할 차례다.

그런데 왜 우리는 곧바로 `CREATE TABLE`을 사용하지 않고, 그림을 그리는 것부터 시작해야 할까? 그것은 우리가 만들어야 할 데이터베이스가 결국 '현실 세계의 비즈니스'를 반영하는 거울이기 때문이다. 어떤 테이블과 컬럼이 필요한지는 전적으로 우리가 어떤 비즈니스를 하고 싶은지에 달려있다. **개념적 모델링은 개발자, 기획자, 현업 담당자 등 모두가 함께 모여 "우리가 만들려는 세상은 이런 모습입니다"라고 합의하는 과정**이다. **기술적인 용어 대신, 모두가 이해할 수 있는 그림과 용어**로 소통하며 생각의 차이를 줄이고, 우리가 만들어야 할 것의 본질을 명확히 하는 가장 중요한 단계다.

이제 우리의 '쇼핑몰' 프로젝트에 이 개념적 설계를 적용해보자.

요구 사항 분석

모든 설계는 요구 사항 분석에서 시작한다. 우리의 쇼핑몰 프로젝트에 주어진 요구 사항을 살펴보자.

[쇼핑몰 요구 사항]

1. **회원 기능**

    - 회원은 고유한 아이디, 비밀번호, 회원명, 주소(기본 배송지), 연락처 정보를 이용해 가입할 수 있다.
    - 회원은 자신의 정보를 언제든지 수정할 수 있다.
    - 회원은 시스템에서 탈퇴할 수 있다.
2. **상품 기능**
    - 관리자(또는 판매자)는 상품을 시스템에 등록할 수 있다.
    - 상품은 고유한 상품 코드, 상품명, 상품 가격, 재고 수량을 정보로 가진다.
    - 상품의 가격이나 재고 등은 언제든지 변경될 수 있다.
3. **주문 기능**
    - 회원은 여러 상품을 한 번에 주문할 수 있다.
    - 주문 시, 각 상품의 주문 수량을 지정할 수 있다.
    - 주문 시, 배송지를 변경할 수 있다. (기본 배송지와 다를 수 있음)
    - 주문이 완료되면, 주문은 '주문 완료' 상태가 되며, 고유한 주문 번호가 생성된다.
    - 주문에는 주문 일시(주문한 날짜), 최종 결제 금액, 배송지 정보가 기록되어야 한다.
    - 관리자는 주문 상태를 '배송 중', '배송 완료' 등으로 변경할 수 있다.
    - 회원은 자신의 주문 내역을 목록으로 조회할 수 있다.

요구 사항 속에 우리가 만들어야 할 데이터 세상의 모든 뼈대가 숨어있다. 이 뼈대를 찾아내는 아주 효과적인 방법이 있는데, 바로 **'명사'와 '동사'에 집중하는 것**이다.

- **명사 (Nouns)**: 우리가 관리해야 할 데이터의 대상, 즉 **엔티티(Entity)** 또는 그 엔티티가 가지는 **속성(Attribute)**이 될 확률이 높다.
- **동사 (Verbs)**: 데이터들 사이의 행위나 관계, 즉 **관계(Relationship)**가 될 확률이 높다.

요구 사항 문장에서 명사와 동사를 추출해보자.

- **명사**: 회원, 아이디, 비밀번호, 회원명, 주소, 연락처, 상품, 상품 코드, 상품명, 가격, 재고 수량, 주문, 주문 번호, 배송지, 주문 일시, 결제 금액, 주문 내역, 주문 수량 등
- **동사**: 가입하다, 수정하다, 탈퇴하다, 등록하다, 변경하다, 주문하다, 조회하다

핵심 데이터 덩어리, 엔티티(Entity) 도출하기

개념적 모델링의 가장 기본적인 구성 요소는 **'엔티티(Entity)'**다. 엔티티는 저장할 가치가 있는 중요 데이터를 가지고 있으면서, 다른 것과 명확히 구별되는 대상이다.

엔티티는 '네이트', '이철수'와 같은 개별 데이터의 '집합'을 의미한다. 예를 들어 '회원'이라는 엔티티는 우리 쇼핑몰에 가입한 모든 개별 회원들의 집합이다.

엔티티를 찾는 쉬운 방법은 먼저 명사에서 시작하는 것이다.

이제 추출한 명사를 보면서, 우리가 데이터로 관리해야 할 의미 있는 대상, 즉 **엔티티(Entity)**를 찾아내야 한다. 하나씩 검토해보자.

- `회원`: 우리 시스템이 관리해야 할 핵심 대상인가? 그렇다. 회원 개개인은 구별되며, 아이디, 회원명, 주소 등의 데이터를 가진다. **(엔티티 확정)**
- `상품`: 회원이 주문하는 대상이다. 상품명, 가격, 재고 등의 데이터를 가진다. 당연히 관리해야 한다. **(엔티티 확정)**
- `주문`: 회원이 상품을 구매하는 '사건'이다. 언제, 누가, 무엇을, 몇 개 주문했는지 등의 중요한 데이터를 담고 있다. 이것 역시 관리해야 할 매우 중요한 대상이다. **(엔티티 확정)**
- `아이디`, `회원명`, `가격`, `재고 수량`, `주문 일시` 등: 이들은 '회원', '상품', '주문'과 같은 독립적인 데이터 덩어리라기보다는, 해당 엔티티에 소속된 세부 정보, 즉 **속성(Attribute)**에 가깝다.

자, 이렇게 해서 우리는 '쇼핑몰'의 핵심 엔티티 3가지를 도출했다.

- **도출된 엔티티:** `회원`, `상품`, `주문`

주문 엔티티

주문은 동사처럼 보인다. 그런데 왜 명사로 분류한 것일까?

'명사를 찾으라'는 규칙은 초보자가 엔티티 후보를 쉽게 찾아내기 위한 가장 기본적인 가이드라인이다. 하지만 이 규칙의 진짜 의미는 '문법적인 명사'만 찾으라는 뜻이 아니라, **'시스템이 정보를 저장하고 관리해야 할 대상'**을 찾으라는 의미이다.

'행위(동사)'와 '행위의 기록(명사)'을 구분하기

'주문하다'는 분명 동사이다. 하지만 우리가 모델링하는 것은 '주문하는 행위 그 자체'가 아니라, **'주문이라는 행위가 발생한 결과로 남는 정보의 묶음, 즉 주문 기록'**이다. 쉽게 이야기하면 주문서라고 볼 수 있는 것이다.

- **동사 (순간적인 행위):** 고객이 "주문하기" 버튼을 누르는 그 순간의 액션.
- **명사 (행위의 결과물이자 기록):** 그 결과로 시스템에 생성되는 **'주문서(An Order)' 또는 '주문 내역'**. 이 주문서에

는 주문번호, 날짜, 금액, 상태 등의 정보가 담겨 있다.

쇼핑몰 사장님 입장에서 생각해보자. 사장님은 고객이 버튼을 누르는 행위(동사)에는 관심이 없다. 대신 어떤 고객이, 언제, 무엇을, 얼마에 샀는지에 대한 **기록(명사)**이 필요하다. 이 기록을 보고 물건을 포장하고 배송하기 때문이다.

'명사를 찾으라'는 규칙의 실질적인 의미

1. 우선 눈에 보이는 명사(고객, 상품 등)를 찾아 엔티티 후보로 삼는다.
2. 다음으로, 중요한 동사(주문하다, 예약하다, 수강신청하다 등)를 찾는다.
3. 그리고 그 동사가 **"관리해야 할 기록이나 증빙을 남기는가?"**라고 질문한다.
4. 만약 그렇다면, 그 동사의 명사형(주문, 예약, 수강)을 새로운 엔티티로 만든다.

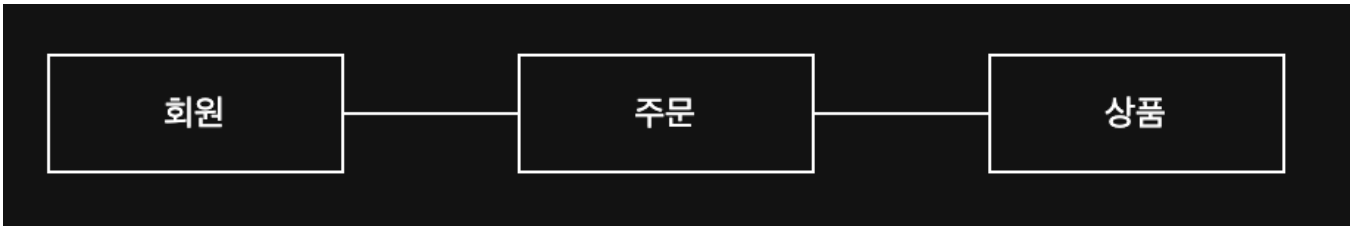
따라서 '주문'은 '주문하다'라는 동사에서 파생되었지만, **그 행위의 결과로 생성된 '데이터 덩어리'이자 '관리 대상'**이므로 개념적 모델링에서는 올바른 **명사형 엔티티**가 된다. 쉽게 이야기해서 '주문하다'라는 동사 행위의 결과로 생성된 '주문서'라는 명사를 관리한다고 생각하면 된다.

엔티티 간의 연결고리, 관계(Relationship) 파악하기

이번에는 동사를 살펴보며 엔티티들이 서로 어떻게 연결되는지, 즉 **관계(Relationship)**를 찾아보자.

- `회원`은 `상품`을 `주문`한다: 이 문장 하나에 여러 관계가 보인다.
  - '회원'과 '주문' 사이에 관계가 있다. (누가 주문했는가?)
  - '주문'과 '상품' 사이에 관계가 있다. (무엇을 주문했는가?)

이렇게 우리는 엔티티라는 기둥을 세우고, 관계라는 보를 연결해서 데이터 구조의 뼈대를 완성했다.



엔티티란?

이번에는 엔티티에 대해서 자세히 알아보자.

엔티티는 한마디로 **"우리가 데이터를 저장하고 관리해야 할 대상"**을 의미한다. 조금 더 구체적으로는 **'저장할 만한 가치가 있는 정보를 여러 개 가지고 있으면서, 다른 것과 명확히 구분되는 유무형의 모든 것'**을 말한다.

유형 엔티티

사람, 사물, 장소와 같이 물리적인 형태가 있는, 우리가 직접 만지거나 볼 수 있는 대상을 의미한다.

- **회원, 직원**: 특정 개인을 나타내는 엔티티
- **상품, 책**: 판매하거나 관리하는 사물을 나타내는 엔티티
- **건물**: 물리적인 장소를 나타내는 엔티티

무형 엔티티

물리적인 형태는 없지만, 개념적으로 존재하는 관리 대상을 의미한다. 주로 어떤 '사건'이나 '개념'을 데이터로 관리할 때 나타난다.

- **주문, 예약, 수강 신청**: '주문하다', '예약하다'와 같은 행위(동사)의 결과로 생성되는 '사건'을 기록하는 엔티티다. 이런 엔티티들은 '주문 번호', '예약 날짜'와 같은 속성을 통해 관리된다.
- **계좌, 학과**: 물리적 실체는 없지만, 은행 업무나 학사 관리에서 중요한 역할을 하는 개념적인 엔티티

아직도 조금 추상적으로 들릴 수 있다. 가장 쉽게 이해하는 방법은 **엑셀(Excel) 시트**를 떠올리는 것이다. 만약 우리가 엑셀로 쇼핑몰 데이터를 관리한다고 상상해보자. 어떤 시트들을 만들게 될까? 아마도 '회원 목록' 시트, '상품 목록' 시트, '주문 내역' 시트 등을 만들 것이다.

바로 이 **'엑셀 시트' 하나하나의 주제가 바로 '엔티티'**다.

- `회원 목록` 시트의 주제 ➔ `회원` **엔티티**
- `상품 목록` 시트의 주제 ➔ `상품` **엔티티**
- `주문 내역` 시트의 주제 ➔ `주문` **엔티티**

[회원 엔티티 예시]

회원id	로그인id	회원명	주소	가입일
1	nate123	네이트	서울시 강남구 테헤란로	2025-08-01 10:00:00
2	chulsoo	이철수	경기도 성남시 분당구	2025-08-03 15:30:00
3	younghee	김영희	서울시 마포구 연남동	2025-08-05 11:00:00

[상품 엔티티 예시]

상품id	상품명	상품 가격	재고 수량

101	기계식 키보드	120000	50
102	무소음 마우스	45000	120
103	4K 모니터	350000	30

[주문 엔티티 예시]

주문id	주문 일시	주문 상태	배송지
1001	2025-08-05 12:10:00	ORDERED	서울시 강남구 테헤란로
1002	2025-08-05 12:15:20	ORDERED	부산시 해운대구
1003	2025-08-05 12:20:05	CANCELED	서울시 강남구 테헤란로

- 개념적 모델링에서 관계는 선으로 표현한다. 따라서 이번 예시에는 외래 키(FK)가 없다.

> **주문 항목 엔티티**
> 주문 엔티티를 보면 어떤 상품을 구매했는지에 대한 정보가 없다. 이 부분은 뒤에서 발견할 주문 항목 엔티티를 통해서 해결한다.

개념적 모델링 단계에서 사용하는 용어들이 실제 데이터베이스의 물리적인 테이블과 어떻게 연결되는지, 엑셀에 비유하여 표로 정리하면 다음과 같다.

[용어 표]

개념적 모델링 용어	데이터베이스 물리 모델	엑셀(Excel) 비유	쇼핑몰 예시
엔티티 (Entity)	**테이블 (Table)**	**시트 (Sheet)**	`member` 테이블
속성 (Attribute)	**열, 컬럼 (Column)**	**열, 컬럼 (Column)**	`member` 테이블의 `member_name`, `address` 컬럼
인스턴스 (Instance)	**행 (Row)**	**행 (Row)**	`member` 테이블의 '네이트' 데이터 행

이처럼 엔티티는 우리가 만들 데이터베이스의 가장 핵심적인 뼈대가 된다. 좋은 엔티티는 다음과 같은 특징을 가진다.

1. **업무 관련성 (Business Relevance)**
   엔티티는 반드시 해당 업무에 필요하고 관리해야 하는 정보여야 하며, 업무 프로세스에 의해 이용되어야 한다.

2. **식별 가능해야 한다 (Uniquely Identifiable)**
   엔티티에 속하는 각각의 데이터(인스턴스, 행)는 서로 명확하게 구분될 수 있어야 한다. '회원' 시트에 회원명이 똑같은 '네이트'가 두 명 있을 수 있다. 이때 우리는 이 두 명을 구분할 수 있는 방법이 필요하다. 예를 들어, 각 회원에게 고유한 `회원번호`나 `아이디`를 부여하는 것이다. 이것이 나중에 '식별자(Identifier)' 또는 '기본 키(Primary Key)'가 된다.

3. **두 개 이상의 정보를 가진다 (Has Attributes)**
   엔티티는 관리할 만한 가치가 있는 여러 정보, 즉 **속성(Attribute)**들을 가진다. 엑셀 시트의 '열(Column)'에 해당한다. '회원'이라는 엔티티(시트)는 `회원명`, `주소`, `연락처`, `가입일` 등 여러 속성(열)을 가진다. 만약 어떤 대상이 단 하나의 정보만 가지고 있다면, 그것은 엔티티가 아니라 다른 엔티티의 일부인 '속성'일 가능성이 높다.

4. **인스턴스(Instance)의 집합이다**
   엔티티는 개념적인 '틀'(시트)이고, 그 틀에 따라 만들어진 실제 데이터 하나하나(행)를 **인스턴스(Instance)**라고 부른다.
   엔티티는 영속적으로 존재하는 두 개 이상의 인스턴스로 구성된 집합이어야 한다.
   - **엔티티 (틀)**: `회원` (엑셀의 '회원 목록' 시트 자체)
   - **인스턴스 (실제 데이터)**: 네이트, 이철수, 김영희 (시트 안의 한 줄, 한 줄의 데이터)

   '회원'이라는 엔티티는 '네이트', '이철수' 등 여러 인스턴스의 집합체인 것이다. 예를 들어, `회원` 엔티티(시트)에 들어있는 각 '행'이 하나의 인스턴스에 해당한다.

5. **다른 엔티티와 관계를 맺는다 (Has Relationships)**
   엔티티는 홀로 존재하지 않고, 다른 엔티티와 서로 관계를 맺는다. `회원`은 `주문`을 하고, `주문`은 여러 `상품`을 포함하는 것처럼 말이다. 이 관계를 파악하는 것이 데이터베이스 설계의 핵심이다. (가끔 관계를 맺지 않는 엔티티가 존재할 수 있다.)

엔티티와 속성의 분류

초보자들은 설계 시점에 엔티티와 속성을 가장 많이 혼동한다. "이게 엔티티인가? 속성인가?" 헷갈릴 때는 이렇게 자문해보자. **"이것이 우리 비즈니스에서 독립적으로 관리되어야 할 정보 덩어리인가?"** 예를 들어 '회원 주소'는 독립적인 정

보 덩어리가 아니라 '회원'이라는 더 큰 정보 덩어리에 속한 일부다. 따라서 '주소'는 속성이다. 반면 '주문'은 주문번호, 주문 일시, 배송지 등 자신만의 여러 정보를 가지는 독립적인 정보 덩어리이므로 엔티티가 된다.

> ✨ **실무 팁 - 과도한 설계**
> 요구 사항에 `관리자(또는 판매자)`라는 명사가 있지만, 엔티티로 도출하지 않았다. 왜일까? 현재 요구 사항에서는 상품 등록의 주체로만 언급될 뿐, 판매자별로 정산을 하거나 판매자 정보를 따로 관리하는 기능이 없다. 이런 경우, 초기 모델에서는 `회원` 엔티티에 '관리자' 역할을 부여하는 컬럼(`role` 등)을 하나 추가해서 구분하는 것이 더 단순하고 효율적이다. 프로젝트가 성장하여 여러 판매자가 입점하는 '오픈 마켓' 형태로 발전한다면, 그때 `판매자`를 별도의 엔티티로 분리하는 모델링 변경을 고려하면 된다. 처음부터 모든 가능성을 대비하여 복잡하게 설계하는 것은 오버 엔지니어링(over-engineering)일 수 있다.

엔티티 분류1

데이터베이스 설계는 '회원', '상품', '주문'과 같은 데이터 덩어리들을 식별하는 것에서 시작한다. 그런데 이 데이터들을 모두 똑같은 방식으로 취급해도 괜찮을까? 예를 들어, '회원' 데이터와 '주문' 데이터는 그 성격이 완전히 다르다. '회원' 데이터는 한 번 가입하면 잘 변하지 않고 꾸준히 유지되지만, '주문' 데이터는 매 순간 새롭게 쌓이는 사건에 가깝다. 결과적으로 시간이 지남에 따라 회원보다는 주문 데이터가 폭발적으로 증가할 것을 쉽게 예측할 수 있다.

이처럼 **엔티티(데이터 덩어리)의 성격과 역할을 제대로 파악하고 구분하는 과정이 바로 엔티티 분류**이다. 이 분류 과정을 통해 우리는 **데이터의 본질을 꿰뚫어 보고, 앞으로 만들 테이블의 구조, 키(Key) 설계, 관계 설정, 그리고 성능 최적화 전략까지 설계의 큰 그림을 그릴 수 있다.** 마치 건축가가 건물을 짓기 전에 벽돌, 철근, 유리 등 각 자재의 특성을 파악하여 적재적소에 사용하는 것과 같은 이치다.

개념적 모델링 단계에서 엔티티는 데이터의 성격, 역할, 그리고 다른 데이터와의 관계에 따라 여러 기준으로 분류할 수 있다. 이를 통해 데이터 구조를 더 명확하게 이해하고 효율적인 데이터베이스를 설계할 수 있다. 엔티티를 분류하는 주요 기준은 존재 형태, 생성 시점 및 역할, 그리고 존재 종속성(독립성)이다.

존재 형태에 따른 분류: 유형, 개념, 사건 엔티티

엔티티를 식별하는 가장 직관적인 방법 중 하나는 엔티티가 나타내는 대상의 존재 형태를 기준으로 분류하는 것이다. 엔티티는 크게 **유형 엔티티**, **개념 엔티티**, **사건 엔티티**의 세 가지로 분류할 수 있다.

1. 유형 엔티티

유형 엔티티는 물리적인 형태를 가지고 있어 눈으로 보거나 만질 수 있는 실체를 표현하는 엔티티다. 이들은 현실 세계에 구체적으로 존재하는 객체들이므로, 데이터 모델링 과정에서 가장 먼저, 그리고 가장 쉽게 식별되는 경향이 있다.

- **특징**:
  - 물리적 형태가 존재한다.
  - 업무로부터 식별이 용이하다.
  - 상대적으로 안정적이며 지속적으로 활용되는 정보를 담는다.
- **예시**:
  - 사원, 학생, 고객, 교수 (사람)
  - 상품, 자재, 물품, 차량, 건물 (사물)
  - 강의실, 창고, 지점 (장소)

2. 개념 엔티티

개념 엔티티는 물리적인 형태는 없지만, 업무적으로 관리해야 할 중요한 개념이나 아이디어를 표현하는 엔티티다. 이는 눈에 보이지 않기 때문에 유형 엔티티에 비해 식별하기가 다소 까다로울 수 있으며, 비즈니스 프로세스에 대한 깊이 있는 이해를 필요로 한다.

- **특징**:
  - 물리적 형태가 없는 추상적인 개념이다.
  - 업무 규칙이나 제도, 분류 기준 등을 표현한다.
  - 유형 엔티티와 마찬가지로 비교적 안정적인 정보를 관리한다.
- **예시**:
  - 부서, 조직, 팀 (조직 구조)
  - 계좌, 보험상품 (금융 상품 및 장부)
  - 과목, 학과 (교육 및 분류 체계)

3. 사건 엔티티 (이벤트 엔티티)

사건 엔티티는 업무 프로세스가 진행됨에 따라 발생하는 특정 행위나 사건을 표현하는 엔티티다. 이는 특정 시점에 발생하며, 비즈니스 활동의 결과를 기록하는 역할을 한다. 따라서 **사건 엔티티는 시간이 지남에 따라 데이터가 지속적으로 누적**되는 특징을 가지며, 각종 통계 및 분석 자료의 핵심 소스가 된다.

- **특징**:
  - 업무 수행에 따라 발생하는 행위를 기록한다.
  - 시간의 흐름에 따라 인스턴스가 계속해서 발생한다.
  - 발생량이 많고, 각종 통계 자료에 활용될 수 있다.
- **예시**:

- 주문, 계약, 청구, 매출 (판매 및 계약 관련 사건)
- 결제, 입금, 출금, 미납 (재무 관련 사건)
- 수강신청, 예약, 취소, 사고접수 (서비스 관련 사건)

실무 예시

이 세 가지 엔티티의 조합은 우리가 만들려는 시스템의 **핵심 목적이 무엇인지 알려주는 중요한 힌트**가 된다.
만약 데이터 모델에 '주문', '결제', '예약' 같은 **사건 엔티티**가 가득하다면, 그 시스템은 마치 **매일매일의 거래를 기록하는 '온라인 거래 장부'**와 같다. 쇼핑몰의 실시간 주문 처리 시스템처럼, 계속해서 발생하는 사건들을 빠르고 정확하게 기록하는 것이 주된 임무라는 뜻이다.

반대로 '고객', '상품', '직원' 같은 **유형/개념 엔티티**가 설계의 중심이라면, 이는 여러 곳에서 사용될 **'기준 정보'나 '고객 명단'을 관리하는 시스템**일 가능성이 높다. 이런 시스템의 목표는 거래 기록보다는, 회사의 중요한 자산이 되는 핵심 정보를 정확하고 일관되게 유지하는 것이다.

결국 엔티티를 어떻게 분류하고 구성하는지 살펴보면, 만들려는 시스템의 최종 목적과 정체성을 엿볼 수 있다. 이는 마치 건물의 골격을 보면 그 건물이 주택인지, 공장인지 알 수 있는 것과 같다.

이런 분류가 실무에서 어떤 도움이 되는지 구체적인 예를 들어보자.

- 우리 쇼핑몰의 `회원`이나 `상품`은 **유형/개념 엔티티**다. 다른 데이터가 존재하기 위한 근간이 된다. 반면 `주문`은 주문이라는 '사건'이 발생할 때마다 생성되는 **사건 엔티티**(이벤트 엔티티)다.

- **실무 관점:** 사건 엔티티는 시간이 지남에 따라 데이터가 폭발적으로 증가할 것을 쉽게 예측할 수 있다. 예를 들어, 우리 쇼핑몰에 10만 개의 상품이 있고 100만 명의 회원이 있다면, `상품` 테이블은 10만 개, `회원` 테이블은 100만 개의 행을 가질 뿐이다. 하지만 하루에 주문이 10,000건씩만 들어와도 `주문` 테이블은 1년이면 365만 건, 3년이면 1000만 건이 넘는 데이터가 쌓이게 된다.

이렇게 데이터의 성격과 증가 추이를 미리 파악하면, `주문` 같은 테이블을 설계할 때부터 다음과 같은 전략을 세울 수 있다.

- **인덱스 전략**: 데이터가 수백만 건이 되면, 특정 회원의 주문 목록을 찾거나(`WHERE 회원id = ?`) 특정 날짜의 주문을 조회하는(`WHERE 주문 일시 BETWEEN ? AND ?`) `SELECT` 문은 **인덱스 없이는 재앙적인 성능 저하**를 일으킨다. 따라서 `회원id`나 `주문 일시` 같은 조회 조건으로 자주 사용될 컬럼에 인덱스를 생성하는 것은 선택이 아닌 필수다.
- **데이터 파티셔닝(Partitioning) 및 아카이빙(Archiving)**: 5년 전, 10년 전 주문 데이터까지 하나의 거대한 테이블에 모두 담아두는 것은 매우 비효율적이다. **파티셔닝**은 `주문` 테이블을 주문 일시(예: 연도별, 월별) 기준으로 물리적으로 분리하여 저장하는 기술이다. 이렇게 하면 '최근 한 달 주문 조회'와 같은 쿼리는 훨씬 작은 데이터 영

역만 탐색하므로 속도가 비약적으로 향상된다. **아카이빙**은 아예 사용 빈도가 극히 낮은 오래된 데이터를 별도의 백업 테이블이나 스토리지로 옮겨서, 현재 운영 중인 테이블의 크기를 작고 빠르게 유지하는 전략이다.

역할 및 발생 시점에 따른 분류: 기본, 중심, 행위 엔티티

이번에는 조금 다른 관점에서 엔티티를 더 깊이 있게 분류해보자.
엔티티는 비즈니스 프로세스 내에서 수행하는 역할과 데이터가 생성되는 시점에 따라 계층적으로 분류할 수 있다. 이 분류법은 데이터의 **발생 순서와 의존성**을 명확히 하여 모델의 논리적 흐름을 체계화한다. 일반적으로 **기본 엔티티**, **중심 엔티티**, **행위 엔티티**로 구분한다.

1. 기본 엔티티 (Fundamental/Key Entity)

기본 엔티티는 업무에 원래부터 존재하는 정보로서, 다른 엔티티에 의해 생성되지 않고 독립적으로 존재할 수 있는 핵심적인 엔티티다. '키 엔티티(Key Entity)'라고도 불리며, 일반적으로 다른 엔티티의 부모 역할을 수행한다. 기본 엔티티는 다른 엔티티로부터 주식별자를 상속받지 않고 자신만의 고유한 주식별자를 가진다는 특징이 있다.

- **특징**:
  - 독립적으로 생성이 가능하다.
  - 주로 다른 엔티티의 부모 역할을 한다.
  - 자신만의 고유한 주식별자를 가진다.
- **예시**:
  - 회원, 상품, 사원, 부서, 고객, 자재

2. 중심 엔티티 (Main/Center Entity)

중심 엔티티는 기본 엔티티로부터 파생되어 생성되며, 해당 업무에서 중심적인 역할을 수행하는 엔티티다. 이들은 독립적으로 존재하기보다는 기본 엔티티 간의 관계에서 발생하는 핵심적인 비즈니스 트랜잭션을 나타내는 경우가 많다. 따라서 데이터 발생량이 많고, 다른 엔티티와의 관계를 통해 수많은 **행위 엔티티를 생성하는 허브 역할**을 한다.

- **특징**:
  - 기본 엔티티로부터 발생한다.
  - 업무 프로세스의 중심적인 역할을 담당한다.
  - 데이터 발생량이 많고, 많은 행위 엔티티를 파생시킨다.
- **예시**:
  - 주문, 계약, 청구, 매출

3. 행위 엔티티 (Action/Behavioral Entity)

행위 엔티티는 두 개 이상의 부모 엔티티(주로 기본 엔티티와 중심 엔티티)로부터 발생하며, 업무가 흘러가면서 생성되

는 상세 정보를 기록하는 엔티티다. 이들은 내용이 자주 변경되거나 데이터양이 지속적으로 증가하는 특징을 가지며, 모델 내에서 데이터양이 가장 많은 엔티티가 되는 경우가 많다.

- **특징**:
  - 두 개 이상의 부모 엔티티로부터 발생한다.
  - 데이터 내용이 자주 변경되거나 데이터양이 빠르게 증가한다.
  - 상세 설계 단계에서 도출되는 경우가 많다.
- **예시**:
  - 주문 이력, 신청변경이력, 결제 내역, 로그, 주문 항목

실무 예시

이 세 가지 분류는 '기본 → 중심 → 행위'로 업무가 흘러가는 순서와 데이터의 '원인과 결과'를 명확하게 보여준다. 쉽게 비유를 하자면 이것은 마치 한 편의 이야기를 만드는 것과 같다. 이야기는 주인공인 '고객'과 '상품'(기본 엔티티)이 등장하면서 시작된다. 이 두 주인공이 만나 '주문'(중심 엔티티)이라는 핵심 사건을 만든다. 그리고 이 '주문'이라는 사건으로 인해 '주문 이력(행위 엔티티)'이나 '결제 내역'(행위 엔티티) 같은 구체적인 행동들이 뒤따라 기록된다.

이런 분류가 실제 개발할 때 어떤 도움이 될까?

1. **모델의 이해도와 커뮤니케이션 비용 감소**
   - 프로젝트는 혼자 하는 것이 아니다. 기획자, 개발자, DBA 등 다양한 직무의 사람들이 협업한다. 이때 '회원'은 기본 엔티티, '주문'은 중심 엔티티, '주문 상품'은 행위 엔티티'라고 정의하면, 모두가 데이터의 위계와 흐름을 동일한 관점에서 이해할 수 있다. "이번 기능은 주문(중심) 로직에 변경이 있고, 그에 따라 결제 이력(행위) 데이터가 추가로 쌓여야 합니다"처럼 명확하고 효율적인 소통이 가능해진다.

2. **체계적인 개발 순서와 일정 관리**
   - 엔티티 분류는 그대로 개발의 우선순위가 된다. 당연히 독립적으로 존재하는 **기본 엔티티**(`회원`, `상품`) 관련 기능부터 개발해야 한다. 그다음, 이들을 기반으로 동작하는 **중심 엔티티**(`주문`) 기능을, 마지막으로 상세 내역인 **행위 엔티티**(`주문 이력`, `결제 내역`) 기능을 구현하는 것이 자연스러운 순서다. 이렇게 하면 의존성 문제없이 안정적으로 시스템을 구축할 수 있다. 프로젝트 관리자(PM) 입장에서는 이를 기준으로 업무 순서를 정할 수 있다.

3. **성능과 데이터 관리 전략 수립의 기준**
   - 엔티티의 특성은 곧 데이터의 특성을 의미한다.
   - **기본 엔티티**(`회원`, `상품`): 데이터 변경이 잦지 않고, 주로 조회(READ) 작업이 많다. 따라서 조회 성능에 최적화된 인덱싱 전략이 중요하다.
   - **중심 엔티티**(`주문`): 조회의 중심축이자 트랜잭션의 핵심이다. 데이터는 꾸준히 증가하며, 조회(`SELECT`)뿐

만 아니라 주문 상태 변경(`UPDATE`)도 빈번하게 일어난다. 따라서 다양한 검색 조건에 대한 인덱스 전략이 매우 중요하다. 예를 들어, `주문` 테이블은 사용자가 '내 주문 목록'을 볼 때(`회원id` 기준), 관리자가 '오늘의 주문'을 확인할 때(`주문 일시` 기준), '배송 준비 중'인 주문을 찾을 때(`주문상태` 기준) 등 여러 방식으로 조회된다. 따라서 `회원id`, `주문 일시`, `주문상태` 같은 컬럼에는 **각각의 쓰임새에 맞는 인덱스를 생성**하는 것이 성능 유지의 관건이다.

- **행위 엔티티**(`주문 이력`, `결제 내역`): 하나의 주문을 할 때 결제 방식을 신용카드, 포인트, 쿠폰 3가지를 사용해서 결제했다고 가정하자. 그러면 `주문`(중심 엔티티)은 하나의 행이 만들어지지만, `결제 내역`(행위 엔티티)은 3개의 행이 만들어진다. 따라서 데이터가 폭발적으로 증가(Heavy `INSERT`)하며, 가장 많은 저장 공간을 차지하게 될 테이블이다. 따라서 설계 초기부터 **데이터 파티셔닝(Partitioning)**이나 **주기적인 아카이빙(Archiving)** 전략을 고민해야 한다. 이런 테이블을 주문일 기준으로 월별 또는 분기별로 분할해두면, 특정 기간의 데이터 조회나 삭제 시 성능 저하를 막을 수 있다. 이런 고민 없이 테이블을 하나로만 운영하면, 몇 년 뒤 데이터가 수억 건 쌓였을 때 시스템 전체가 느려지는 재앙을 맞게 된다. 물론 중심 엔티티도 데이터가 많다면 이런 고민이 함께 필요하다.

엔티티를 역할과 시점에 따라 분류하는 것은 단순히 이론적인 활동이 아니다. 복잡한 비즈니스를 명확한 데이터 구조로 풀어내고, 프로젝트를 안정적으로 이끌어가기 위한 **실무 데이터베이스 설계의 첫걸음**이라 할 수 있다.

실무 이야기: 분류 용어를 외우기보다 본질을 이해하는 것

실무에서 동료 개발자나 DBA와 회의하면서 "이건 중심 엔티티니까..." 라거나 "저 테이블은 사건 엔티티의 특징을 가지므로..." 와 같은 학술적인 용어는 거의 사용하지 않는다. 그렇다면 왜 엔티티를 분류하는 방법을 배우는 것일까?

이 분류법은 데이터베이스를 설계하는 **'사고의 틀(Framework)'**을 제공한다. 용어 자체를 암기해서 사용하는 것이 목적이 아니라, 각 테이블이 가지는 데이터의 **'본질'과 '성격'을 빠르고 정확하게 간파하는 훈련**을 하는 것이다. 숙련된 개발자는 이런 용어를 사용하지 않아도, 테이블 설계를 보면 본능적으로 그 성격을 파악하고 그에 맞는 전략을 구사한다. 우리가 이 개념을 배우는 것은, 바로 그 '본능적인' 전문가의 사고방식을 체계적으로 따라가기 위함이다.

정리하자면, 엔티티 분류는 실무에서 사용하는 '용어'라기보다는, 복잡한 요구 사항 속에서 데이터의 구조와 성격을 꿰뚫어 보고, 미래에 발생할 문제를 예측하며, 최적의 해결책을 찾아가는 과정에서 머릿속에 그려지는 **'설계 지도'** 와 같다. 이 지도를 그리는 훈련을 통해 우리는 더 견고한 데이터베이스 시스템을 만들 수 있는 것이다.

엔티티 분류2

강한 엔티티와 약한 엔티티

엔티티는 다른 엔티티와의 관계 속에서 자신의 존재 여부가 결정되는지에 따라 **강한 엔티티**와 **약한 엔티티**로 분류할 수 있다.

1. 강한 엔티티 (Strong Entity)

강한 엔티티는 다른 어떤 엔티티의 존재 여부와 관계없이 독립적으로 존재할 수 있는 엔티티를 의미한다. 즉, 자신의 존재를 위해 다른 엔티티에 의존하지 않는다. 강한 엔티티는 자신을 유일하게 식별할 수 있는 고유한 속성 또는 속성들의 집합, 즉 **기본 키(Primary Key)**를 가지고 있다.

- **특징**:
  - 독립적인 존재가 가능하다.
  - 자신만의 고유한 주식별자를 가진다.
  - 다른 엔티티의 존재에 의존하지 않는다.
- **예시**:
  - 사원, 고객, 상품, 부서 등과 같이 독립적으로 관리될 수 있는 대부분의 엔티티.

2. 약한 엔티티 (Weak Entity)

약한 엔티티는 다른 엔티티(이를 '소유 엔티티' 또는 '식별 엔티티'라 부르는 강한 엔티티)가 존재하지 않으면 독립적으로 존재할 수 없는, 존재 종속적인 엔티티다.

- **특징**: 약한 엔티티는 소유 엔티티가 있어야만 자신의 존재가 의미를 가진다.
- **식별자 구성**: 전통적인 방식에서 약한 엔티티는 자신을 식별하기 위해 소유 엔티티의 주식별자를 빌려와 자신의 속성(부분키)과 결합하여 주식별자로 삼는다.
  - 예를 들어 부모의 기본 키(`employee_id`)를 가져와 자식의 속성(`name`)과 합쳐 **복합 기본 키 (Composite Primary Key)** (`employee_id, name`)를 만든다.
- **예시**:
  - 부양가족: '부양가족' 정보는 특정 '사원'에게 소속될 때만 의미가 있다. 어떤 사원에도 속하지 않은 '홍길동(자녀)'이라는 데이터는 존재할 수 없다. 즉, 부양가족은 사원 없이는 존재 의미가 없다.

실무 예시

왜 강한 엔티티와 약한 엔티티를 구분해야 할까?

예를 들어보자. 회사 인사 시스템 데이터베이스에 '네이트(자식)'라는 부양가족 정보는 있는데, 이 사람이 대체 **누구의** 부양가족인지 알 수 없다면 어떻게 될까?

- 이 부양가족에게 연말정산 혜택을 적용해야 하는가? 누구의 연말정산에?

- 가족수당을 지급해야 한다면, 어느 사원에게 지급해야 하는가?
- 비상 연락망에 이 가족의 정보를 포함해야 하는가? 누구의 비상 연락망에?

이처럼 부모 엔티티(사원) 정보가 없는 자식 데이터(부양가족), 즉 **'고아 데이터(Orphaned Data)'**가 생겨나면 데이터베이스의 **정합성(Integrity)**이 깨지고, 인사 관리 시스템 전체가 신뢰를 잃게 된다. 급여, 복지, 세금 등 모든 핵심 기능에 심각한 오류를 초래할 수 있다.
약한 엔티티는 저장할 때 부모의 식별자를 반드시 함께 포함해서 저장해야 한다. 따라서 부모가 존재하지 않는 문제를 설계 단계에서 부터 원천 차단할 수 있다.
강한 엔티티와 약한 엔티티의 구분은 바로 이런 데이터 재앙을 **설계 단계에서부터 원천적으로 차단**하기 위해 존재한다

엔티티의 독립성을 기준으로 강한 엔티티와 약한 엔티티를 구분하는 것은 **테이블의 키와 외래 키 제약조건을 설계하는데 중요한 역할**을 한다.

> **실무 이야기 - 식별 관계, 비식별 관계**
> 전통적인 설계 방식에서 약한 엔티티는 소유 엔티티의 주식별자를 빌려와 자신의 속성(부분키)과 결합해서 복합 기본 키를 만드는 방식을 사용한다. 이것을 논리적 모델링 단계에서 **식별 관계**라 한다.
> 이렇게 되면 자식을 만들 때 반드시 부모의 PK 값을 입력해야 한다. 결과적으로 논리적 제약 조건이 테이블의 기본 키 구조에 그대로 반영되어, 누가 봐도 관계를 명확하게 이해할 수 있고 데이터 무결성을 PK 레벨에서 보장한다. 즉 부모 없는 자식이 존재할 가능성을 원천 차단한다.
> 이것이 바로 데이터 모델링의 원칙을 가장 충실하게 따른 '전통적'이고 '교과서적인' 방법이다.
> 하지만 이런 전통적인 방식은 최근에는 잘 사용하지 않고, 대신에 더 유연하고 실용적인 **비식별 관계**라는 방법을 주로 사용한다.
> 식별 관계와 비식별 관계에 대한 자세한 내용은 논리적 모델링에서 설명한다.

구조적 관계 표현을 위한 특수 엔티티

엔티티 간의 **관계**가 단순한 연결선을 넘어, 그 자체가 **하나의 독립된 의미를 갖는 개념**일 때가 있다.
이러한 **사건(Event)이나 분류(Classification)**와 같은 복잡한 관계를 효과적으로 모델에 반영하기 위해 사용하는 것이 바로 **연관 엔티티**와 **슈퍼타입/서브타입** 구조이다.

> **연관 엔티티, 슈퍼타입/서브타입 엔티티는 별도로 다룬다.**
> 두 내용은 모두 별도로 자세히 다루기 때문에 여기서는 이런 것이 있구나 참고만 하고 넘어가자
>
> - 연관 엔티티는 개념적 모델링에서 뒤에서 별도로 설명한다.
> - 슈퍼타입/서브타입은 설계2편에서 다룬다.

1. 연관 엔티티 (Associative Entity)

연관 엔티티는 두 개 이상의 엔티티 간에 발생하는 특정 **사건, 행위, 계약** 등을 표현하기 위해 도출되는 엔티티이다. 즉, 관계(Relationship) 자체가 중요한 속성을 가져 하나의 독립된 실체(Entity)로 다뤄져야 할 때 사용된다.

- **개념적 본질**: 연관 엔티티의 가장 중요한 존재 이유는 **'관계'에 종속되는 속성을 저장**하기 위함이다. 예를 들어, '학생'과 '과목'의 관계에서는 '성적'이나 '수강신청일' 같은 데이터가 발생한다. 이 데이터는 '학생'의 것도, '과목'의 것도 아닌, '수강'이라는 **행위** 자체에 속한다. 이 '수강'을 표현하는 것이 바로 연관 엔티티다.
- **구조적 의미**: 연관 엔티티는 두 개 이상의 다른 엔티티 간에 존재하는 다대다(M:N) 관계를 해소한다. 관계형 데이터베이스는 다대다 관계를 직접 구현할 수 없기 때문에, 이 중간 엔티티를 통해 하나의 다대다 관계를 일대다(1:N), 다대일(N:1) 관계로 변환한다.

2. 슈퍼타입/서브타입 엔티티 (Supertype/Subtype Entity)

슈퍼타입/서브타입 모델은 논리적으로 동일한 개념 그룹에 속하지만 일부 속성이나 관계에서 차이가 있는 엔티티들을 효과적으로 표현하기 위한 기법이다. 이는 **'IS-A' 관계**(예: '관리자는 사원의 한 종류이다')를 모델링하며, 객체지향 프로그래밍의 상속(Inheritance) 개념과 매우 유사하다.

- **슈퍼타입 (Supertype)**: 그룹 내 모든 엔티티(서브타입)들이 공통적으로 가지는 속성과 관계를 정의하는 상위 엔티티다.
- **서브타입 (Subtype)**: 슈퍼타입으로부터 공통 속성을 상속받고, 자신만의 고유한 속성이나 관계를 추가로 가지는 하위 엔티티다.
- **예시**: '사원' 엔티티를 슈퍼타입으로, '정규직 사원'과 '계약직 사원'을 서브타입으로 모델링할 수 있다.

엔티티 분류는 설계자가 다양한 관점에서 데이터의 본질을 이해하는데 도움을 준다. 다음 시간에는 엔티티의 내용을 채우는 속성과 식별자에 대해 알아보자.

속성과 식별자

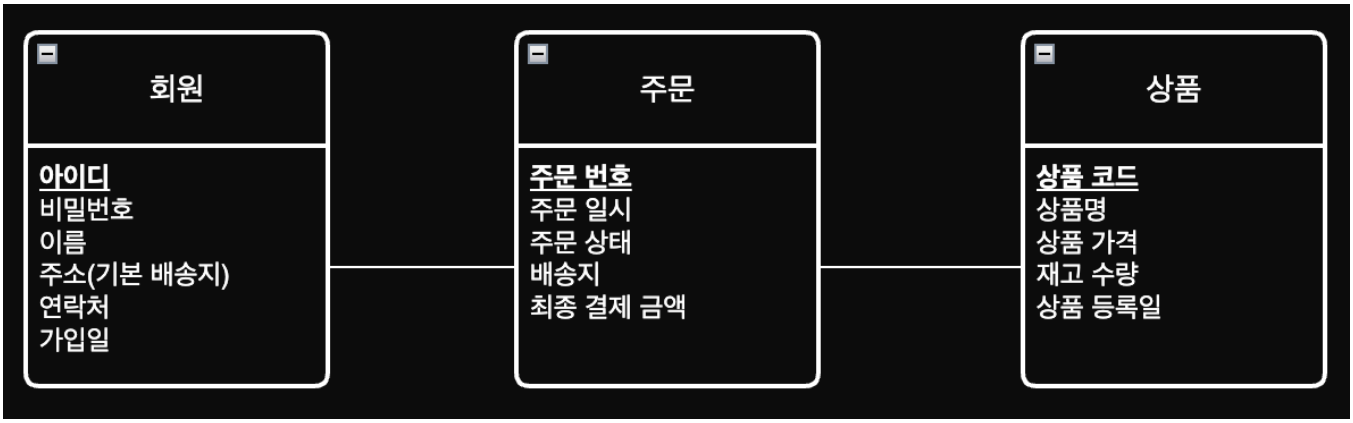
뼈대를 세웠으니 이제 살을 붙일 차례다. 각 엔티티가 어떤 구체적인 데이터 항목들을 가져야 하는지 요구 사항을 기반으로 정의해야 한다.

엔티티의 세부 정보, 속성 정의하기

속성(Attribute)이란 엔티티가 가지는 구체적인 특성이나 정보다. '회원' 엔티티라면 회원명, 이메일, 주소 등이 속성이 된다. 우리가 실제로 테이블에 만들게 될 '컬럼(열)'이 바로 이 속성이다.

'쇼핑몰'의 각 엔티티에 어떤 속성들이 필요할지 요구 사항을 다시 꼼꼼히 살펴보며 정의하자.

- **회원**: `아이디`, `비밀번호`, `회원명`, `주소(기본 배송지)`, `연락처`, `가입일`
- **상품**: `상품 코드`, `상품명`, `상품 가격`, `재고 수량`, `상품 등록일`
- **주문**: `주문 번호`, `주문 일시`, `주문 상태`, `배송지`, `최종 결제 금액`



- 이런 그림을 ERD(Entity-Relationship Diagram)라 한다. ERD는 뒤에서 설명한다.

데이터의 유일한 이름표, 식별자 지정하기

정의된 속성들 중에서 아주 특별한 역할을 하는 속성을 골라야 한다. 바로 **식별자(Identifier)**다.

왜 식별자가 필요한가?

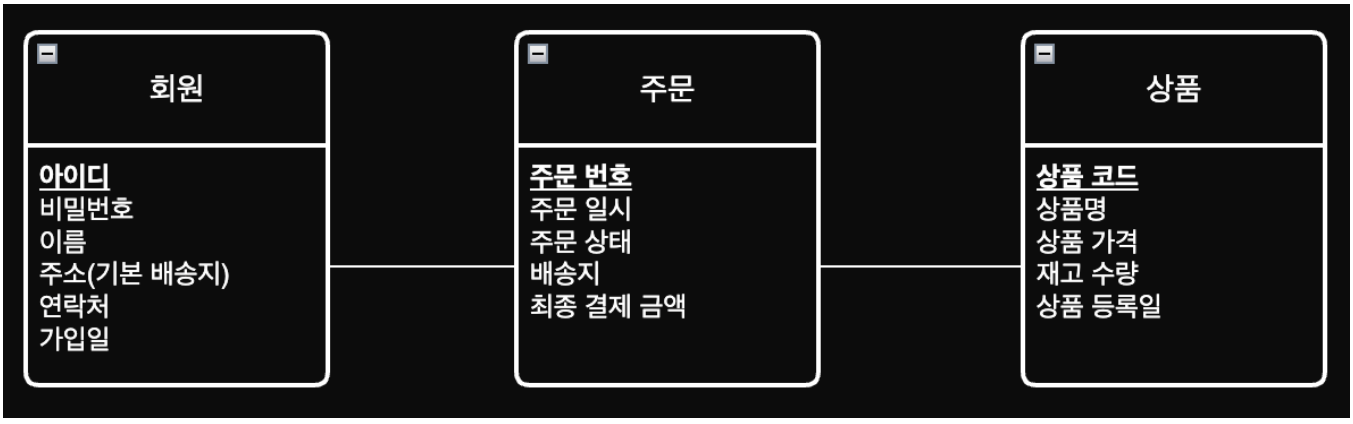
세상에 동명이인은 많다. 우리 쇼핑몰에 '네이트'라는 회원명을 가진 고객이 여러 명일 수 있다. 이때 우리가 '네이트' 고객의 주소를 변경하려면, 여러 '네이트' 중에 정확히 어떤 '네이트'을 말하는지 구분할 수 있어야 한다. 식별자는 이처럼 각 데이터(엔티티의 인스턴스)를 다른 데이터와 유일하게 구별해주는 이름표 역할을 한다. 나중에 데이터베이스 테이블에서는 **기본 키(Primary Key)**가 된다.

각 엔티티의 식별자로 무엇이 좋을지 요구 사항을 보며 정해보자.

- `회원` 엔티티 ➜ `고유한 아이디`: 회원은 가입 시 중복되지 않는 고유한 아이디를 사용한다. 식별자로 적합하다.
- `상품` 엔티티 ➜ `고유한 상품 코드`: 상품명은 겹칠 수 있지만, 시스템에서 부여하는 고유한 코드는 겹치지 않는다.
- `주문` 엔티티 ➜ `고유한 주문 번호`: 주문이 생성될 때마다 시스템에서 부여하는 고유한 번호다.

이제 우리는 각 엔티티와 그 엔티티를 구성하는 속성, 그리고 각 엔티티를 대표하는 식별자까지 모두 정의했다. 개념적 설계의 절반 이상이 끝난 셈이다. 이 내용을 정리하면 다음과 같다.

- **회원(member)**
  - 아이디 (식별자)
  - 비밀번호
  - 회원명
  - 주소
  - 연락처
  - 가입일
- **상품(product)**
  - 상품 코드 (식별자)
  - 상품명
  - 상품 가격
  - 재고 수량
  - 등록일
- **주문(order)**
  - 주문 번호 (식별자)
  - 주문 일시
  - 주문 상태
  - 배송지
  - 최종 결제 금액



- 밑줄과 강조를 통해 식별자를 구분했다.

카디널리티와 참여도

우리는 엔티티라는 기둥을 세우고, 그 기둥들을 관계라는 선으로 연결했다. 하지만 이 연결이 얼마나 튼튼하고, 어떤 규칙으로 연결되어야 하는지는 아직 정하지 않았다. 가령 '회원'과 '주문' 사이에 선을 그었지만, "한 명의 회원이 여러 주문을 할 수 있는가?", "주문 없이 회원만 존재할 수 있는가?"와 같은 구체적인 규칙을 정립해야 한다. 이 규칙을 정의하는 것이 바로 관계를 상세화하는 과정이다.

관계의 규칙을 정의하는 데에는 두 가지 핵심 요소가 있다. 바로 **카디널리티(Cardinality)**와 **참여도(Optionality)**다.

관계의 수량, 카디널리티 분석하기

카디널리티는 한 엔티티의 인스턴스(개별 데이터)가 다른 엔티티의 인스턴스와 몇 개나 관계를 맺을 수 있는지를 나타내는 수량 제약이다. 간단히 말해 **1:1, 1:N, N:1, M:N** 관계를 따지는 것이다.

카디널리티 관계의 4가지 종류

- 일대일(1:1)
- 일대다(1:N)
- 다대일(N:1)
- 다대다(M:N)

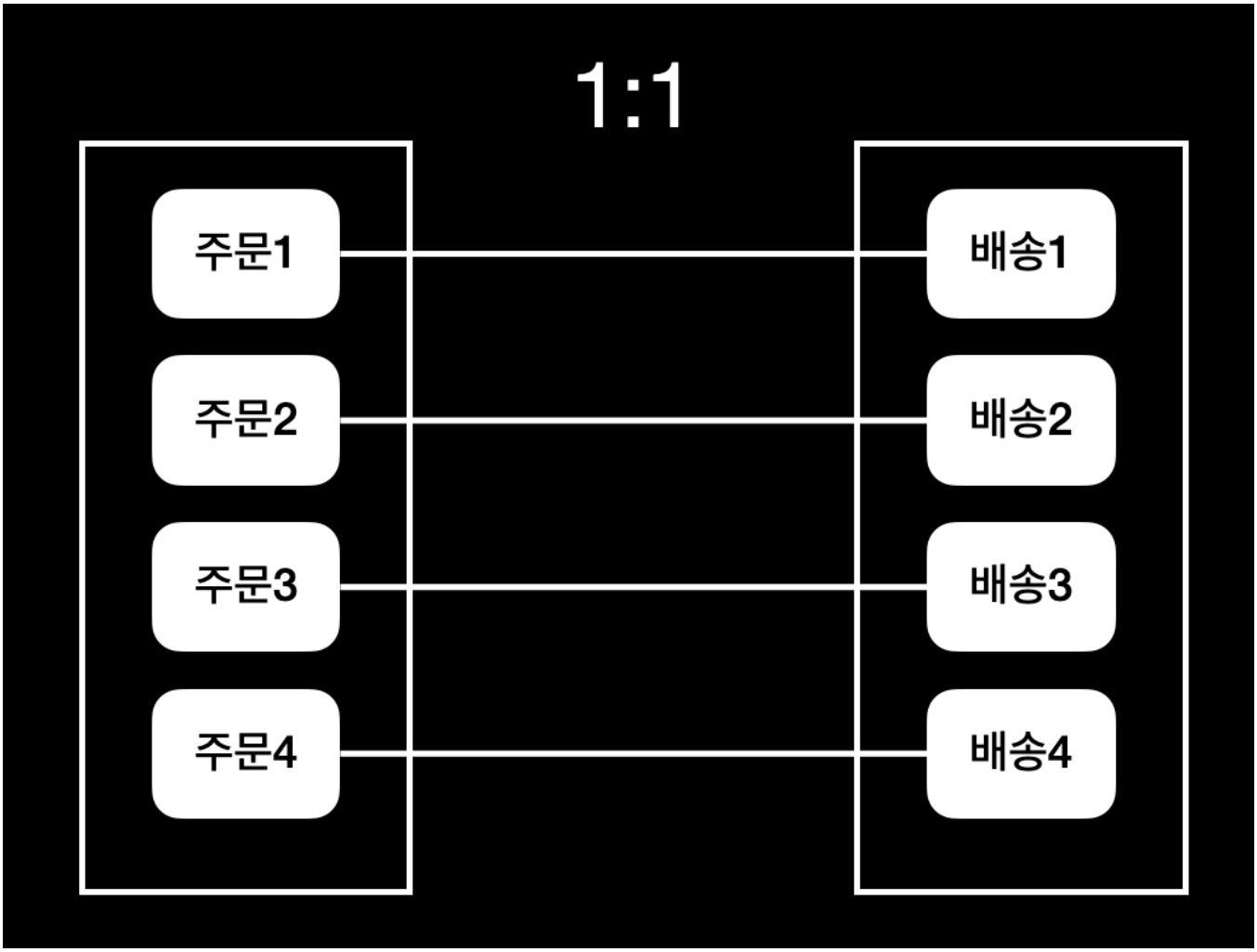
카디널리티 표기 기준 정리

카디널리티는 엔티티 **인스턴스(행)** 하나가 다른 엔티티의 인스턴스와 맺을 수 있는 **최대 숫자**를 기준으로 표기한다.

- **1 (하나):** 한 엔티티의 인스턴스가 다른 엔티티의 인스턴스와 **최대 1개**의 관계를 가질 수 있을 때 사용한다.
- **N (하나 이상):** 한 엔티티의 인스턴스가 다른 엔티티의 인스턴스와 **1개 이상**, 즉 여러 개의 관계를 가질 수 있을 때 사용한다.

카디널리티의 종류를 하나씩 분석해보자.

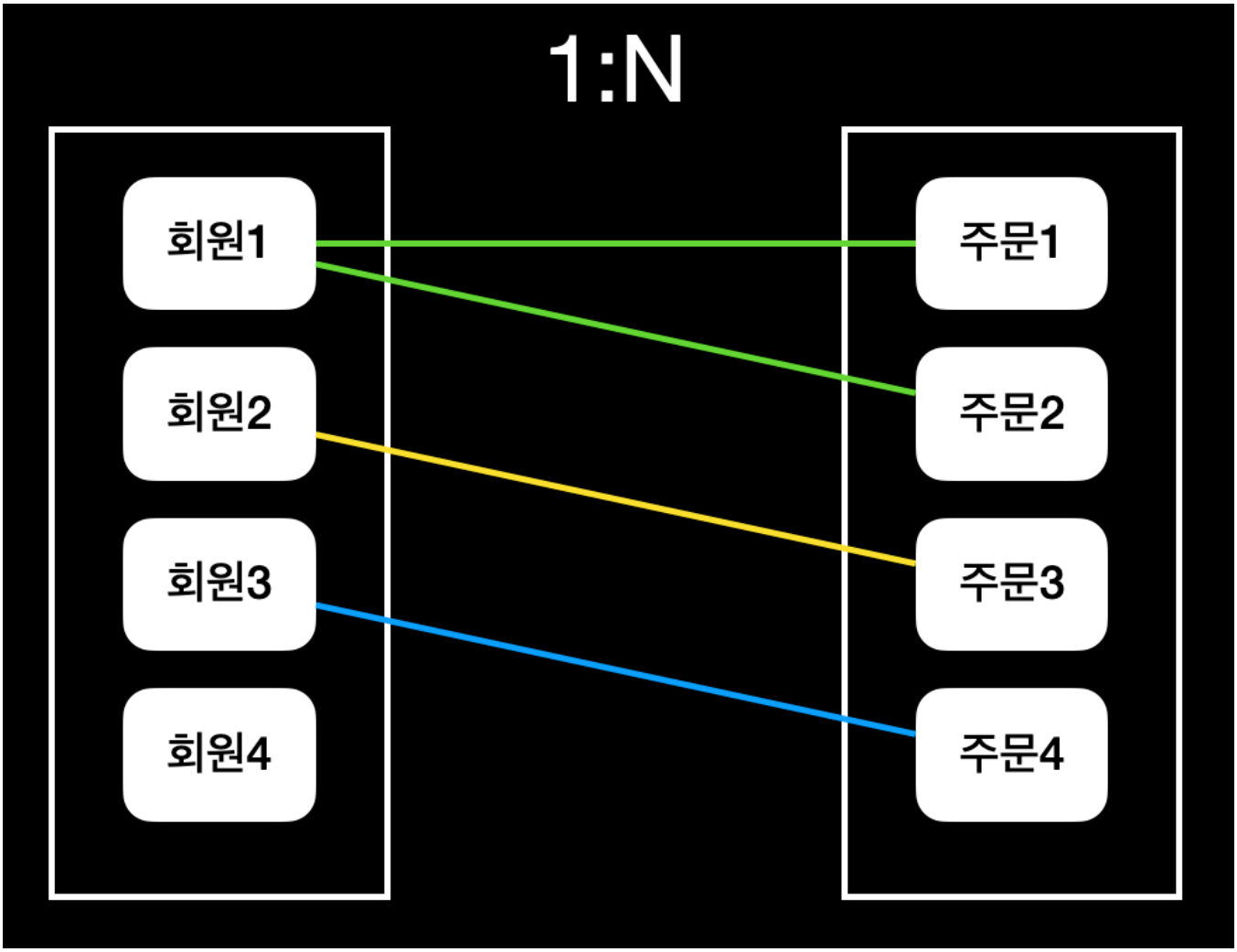
일대일(1:1)



- 예를 들어 쇼핑몰에 `배송`이라는 엔티티가 있다고 가정하자. 여기서 `배송`은 쇼핑몰 회사가 배송의 상태를 추적하기 위해 사용한다고 가정하자.
- 주문 인스턴스(행)가 하나 생성되면 해당 주문의 배송을 처리하는 배송 인스턴스(행)도 하나만 생성된다고 가정하겠다.
- 하나의 주문 인스턴스는 반드시 하나의 배송 인스턴스와 관계를 맺을 수 있다.
  - 예를 들어 `주문1`은 `배송1`이라는 하나의 인스턴스와 관계를 맺는다.
- 하나의 배송 인스턴스는 반드시 하나의 주문 인스턴스와 관계를 맺을 수 있다.
  - 예를 들어 `배송1`은 `주문1`이라는 하나의 인스턴스와 관계를 맺는다.

일대일 관계에서는 양쪽 방향 모두 하나의 인스턴스가 반드시 최대 하나의 인스턴스와만 관계를 맺을 수 있다. 만약 하나의 주문 인스턴스가 둘 이상의 배송 인스턴스와 관계를 맺을 수 있다면 이것은 일대일(1:1) 관계가 아니다.

일대다(1:N)



- 한 명의 `회원`은 여러 번 `주문`할 수 있는가? 그렇다. (다, N)
  - 예를 들어 `회원1`은 쇼핑몰에서 여러 번 주문할 수 있다. `회원1`은 총 2번 주문해서 `주문1`, `주문2`와 관계를 맺는다.
- 하나의 `주문`은 여러 `회원`에게 속할 수 있는가? 아니다. 하나의 주문은 반드시 한 명의 주인만 가진다. (일, 1)
  - 예를 들어 `주문1`은 `회원1`에만 속한다. 하나의 주문을 여러 회원이 함께 진행할 수는 없다.
- **결론**: `회원`과 `주문`은 **일대다(1:N)** 관계다.

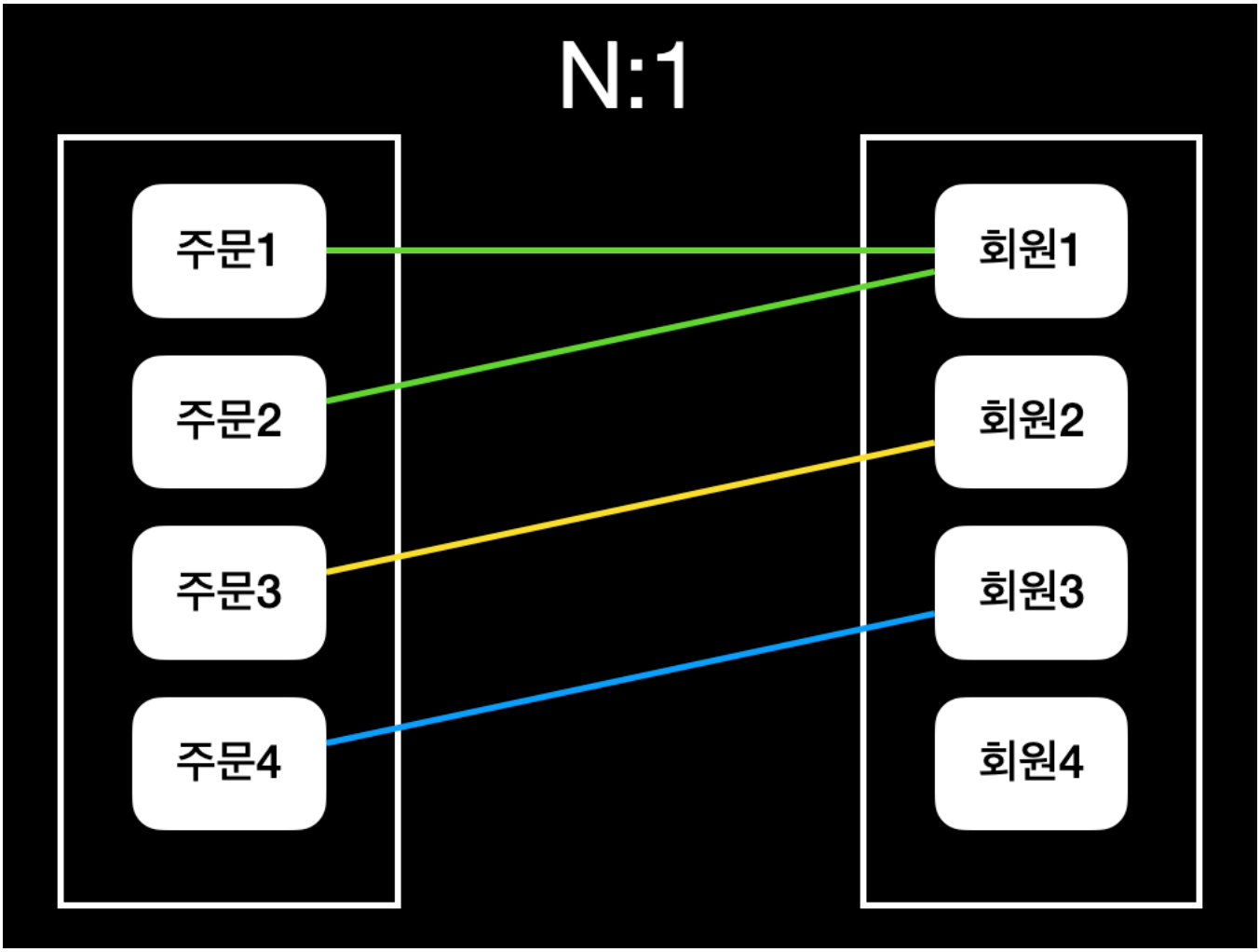
카디널리티 관계 쉽게 이해하기 - 일대다

카디널리티를 쉽게 이해하는 핵심은 인스턴스 하나의 입장으로 각각 나누어 생각해보면 된다.

- 먼저 `회원1`의 입장에서 보자. `회원1`은 여러개의 주문을 가질 수 있다. 그래서 `?(회원) -> N(주문)` 관계가 된다. 여기서 핵심은 인스턴스 하나의 입장에서 상대편과의 관계를 봐야한다는 점이다.
- 반대로 `주문1`의 입장에서 보면 `주문1`은 하나의 회원에만 속한다. 그래서 `1(회원) <- ?(주문)` 관계가 된다. 여기서 핵심은 인스턴스 하나의 입장에서 상대편과의 관계를 봐야한다는 점이다.
- 이 둘을 합치면 `1(회원) - N(주문)` 관계를 알 수 있다.
- 물론 `회원3`과 같이 하나만 주문해서 `1:1` 처럼 보이는 관계도 있고 `회원4`와 같이 아직 주문하지 않은 회원이어서, 주문과 관계가 없는 인스턴스도 있다. 카디널리티는 엔티티 간의 관계에서 한 인스턴스가 가질 수 있는 최대의

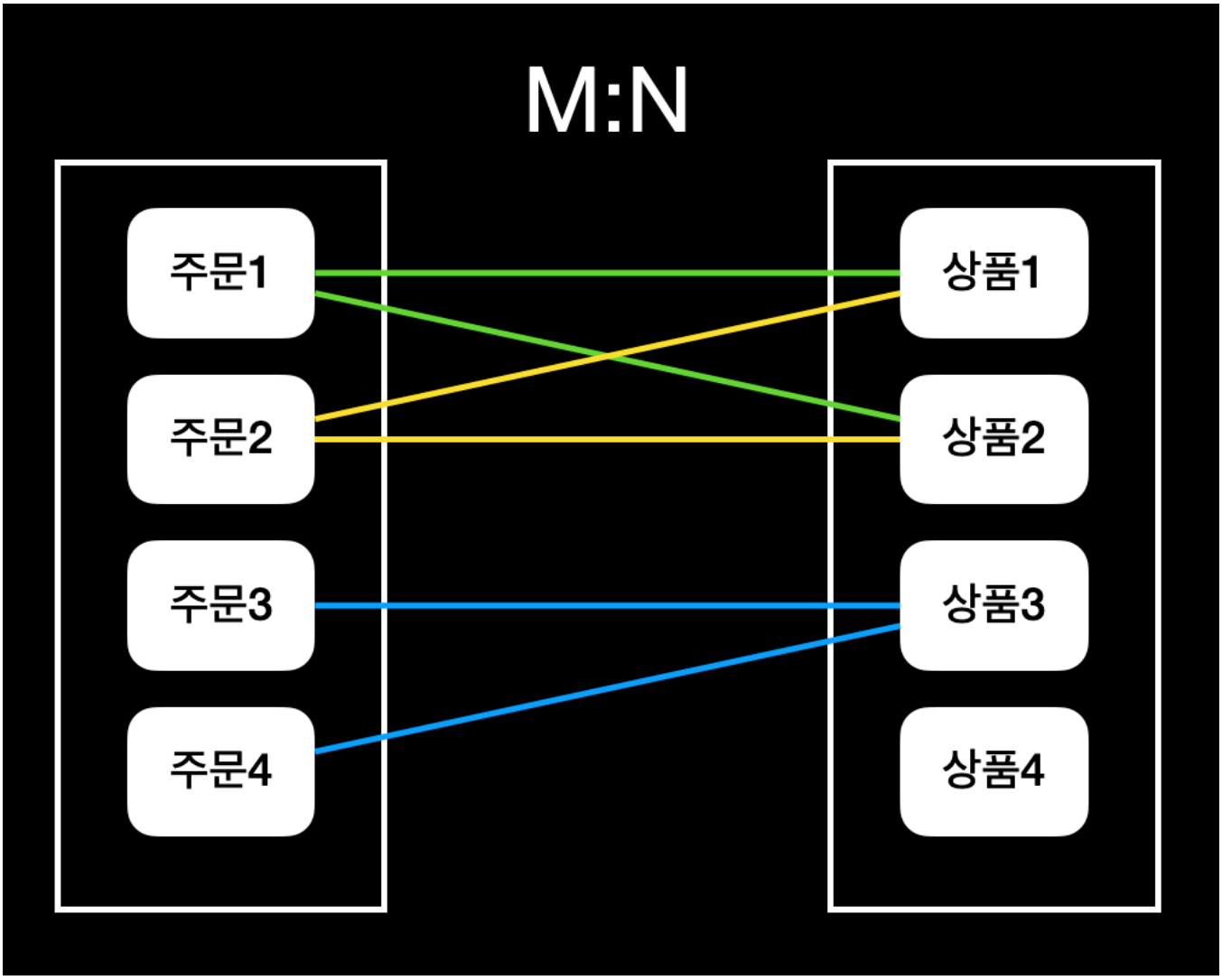
숫자를 기준으로 표기한다. (최대 하나를 가지면1, 하나 이상을 가질 수 있으면 N)

다대일(N:1)



- 다대일 관계는 단순히 일대다 관계의 반대이다.
- 앞의 예제에서 단순히 둘의 위치만 바꾸었다.
- 하나의 `주문`은 여러 `회원`에게 속할 수 있는가? 아니다. 하나의 주문은 반드시 한 명의 주인만 가진다. (일, 1)
  - 예를 들어 `주문1`은 `회원1`에만 속한다. 하나의 주문을 여러 회원이 함께 진행할 수는 없다.
- 한 명의 `회원`은 여러 번 `주문`할 수 있는가? 그렇다. (다, N)
  - 예를 들어 `회원1`은 쇼핑몰에서 여러 번 주문할 수 있다. `회원1`은 총 2번 주문해서 `주문1`, `주문2`와 관계를 맺는다.
- **결론**: `주문`과 `회원`은 **다대일(N:1)** 관계다.

다대다(M:N)



- 하나의 `주문`에 여러 개의 `상품`이 포함될 수 있는가? 그렇다. 요구 사항에 "여러 상품을 한 번에 주문"할 수 있다고 명시되어 있다. (다, N)
- 하나의 `상품`은 여러 `주문`에 포함될 수 있는가? 그렇다. '좋은 키보드'라는 상품은 네이트도 주문하고, 이철수도 주문할 수 있다. (다, M)
- **결론**: `주문`과 `상품`은 **다대다(M:N)** 관계다.

카디널리티 관계 쉽게 이해하기 - 다대다

카디널리티를 쉽게 이해하는 핵심은 인스턴스 하나의 입장으로 각각 나누어 생각해보면 된다.

- `주문1`의 입장에서 보면 `주문1`은 `상품1`, `상품2`와 같이 여러개의 상품을 가질 수 있다. 그래서 `?(주문) -> N(상품)` 관계가 된다. 여기서 핵심은 인스턴스 하나의 입장에서 상대편과의 관계를 봐야한다는 점이다.
- 반대로 `상품1`의 입장에서 보면 `상품1`은 `주문1`, `주문2`와 같이 여러 주문에 속한다. 그래서 `M(주문) <- ?(상품)` 관계가 된다. 여기서 핵심은 인스턴스 하나의 입장에서 상대편과의 관계를 봐야한다는 점이다.
- 이 둘을 합치면 `M(주문) - N(상품)` 관계를 알 수 있다.
- 카디널리티는 엔티티 간의 관계에서 한 인스턴스가 가질 수 있는 최대의 숫자를 기준으로 표기한다.

카디널리티 표기 기준 정리(복습)

카디널리티는 엔티티 **인스턴스(행)**가 다른 엔티티의 인스턴스와 맺을 수 있는 **최대 숫자**를 기준으로 표기한다.

- **1 (하나):** 한 엔티티의 인스턴스가 다른 엔티티의 인스턴스와 **최대 1개**의 관계를 가질 수 있을 때 사용한다.
- **N (하나 이상):** 한 엔티티의 인스턴스가 다른 엔티티의 인스턴스와 **1개 이상**, 즉 여러 개의 관계를 가질 수 있을 때 사용한다.

관계의 필수 여부, 참여도 분석하기

참여도(Optionality)는 한 엔티티의 인스턴스가 관계에 반드시 참여해야 하는지(필수), 아니면 참여하지 않을 수도 있는지(선택)를 결정한다.

다시 `회원`과 `주문`의 관계를 예로 들어보자.

- `회원` 입장에서 `주문`과의 관계는 필수인가, 선택인가?
  - 모든 회원이 반드시 주문을 해야만 하는가? 아니다. 가입만 하고 한 번도 주문하지 않은 회원이 있을 수 있다. 따라서 `회원`의 참여는 **선택(Optional)**이다.
- `주문` 입장에서 `회원`과의 관계는 필수인가, 선택인가?
  - 모든 주문은 반드시 특정 회원에게 속해야 하는가? 그렇다. 어떤 회원인지도 모르는 유령 주문은 존재할 수 없다. 따라서 `주문`의 참여는 **필수(Mandatory)**다.

카디널리티와 참여도 분석을 종합하면 관계를 아주 명확하게 정의할 수 있다.
"**한 명의 회원**은 여러 개의 주문을 할 수도 있고 안 할 수도 있다. 하지만 **하나의 주문**은 반드시 단 한 명의 회원에게 속해야만 한다."

이처럼 카디널리티와 참여도를 정의함으로써 우리는 비즈니스 규칙을 데이터 모델에 정확하게 녹여낼 수 있다.

관계의 표현과 외래 키

개념적 모델링 단계는 관계형 데이터베이스를 포함한 특정 기술에 종속되지 않는다. 쉽게 이야기해서 개념적 모델링을 하고 나면 그 결과를 관계형 데이터베이스 뿐만 아니라 NoSQL 같은 곳에도 사용할 수 있다.

개념적 모델링 단계에서는 엔티티 간의 관계를 **관계선(Relationship)**으로 표현하며, 외래 키 같은 특정 기술의 구현적 요소는 포함하지 않는 것이 원칙이다. 예를 들어서 개념적 모델링 단계에서는 외래 키를 표현하기 위해 `주문`에 `회원id`를 포함하지 않는다. 이후에 **논리적 모델링 단계**에서 `주문`에 `회원id`를 포함해서 이런 관계선이 외래 키로 표현된다. 참고로 논리적 모델링 단계는 관계형 데이터베이스에 맞는 구조로 모델링 하는 단계이다. 따라서 외래 키 같은 내용이 등장한다.

1. **주문 엔티티의 속성**은 주문 자체의 고유한 특성만 포함한다.
    - 주문 번호, 주문 일시, 주문 상태, 배송지, 최종 결제 금액
2. **회원과 주문의 관계**는 별도의 관계선으로 표현한다.
    - "회원이 주문을 한다" (1:N 관계)
    - 이때 외래 키를 표현하기 위해 `회원id`를 주문 엔티티에 속성으로 넣지 않음

> **실무에서는 개념적 모델링과 논리적 모델링을 따로 구분하지 않고 함께 진행하는 경우가 많다. 따라서 처음부터 외래 키 값을 포함해서 함께 설계를 진행하기도 한다.**

ERD 완성하기

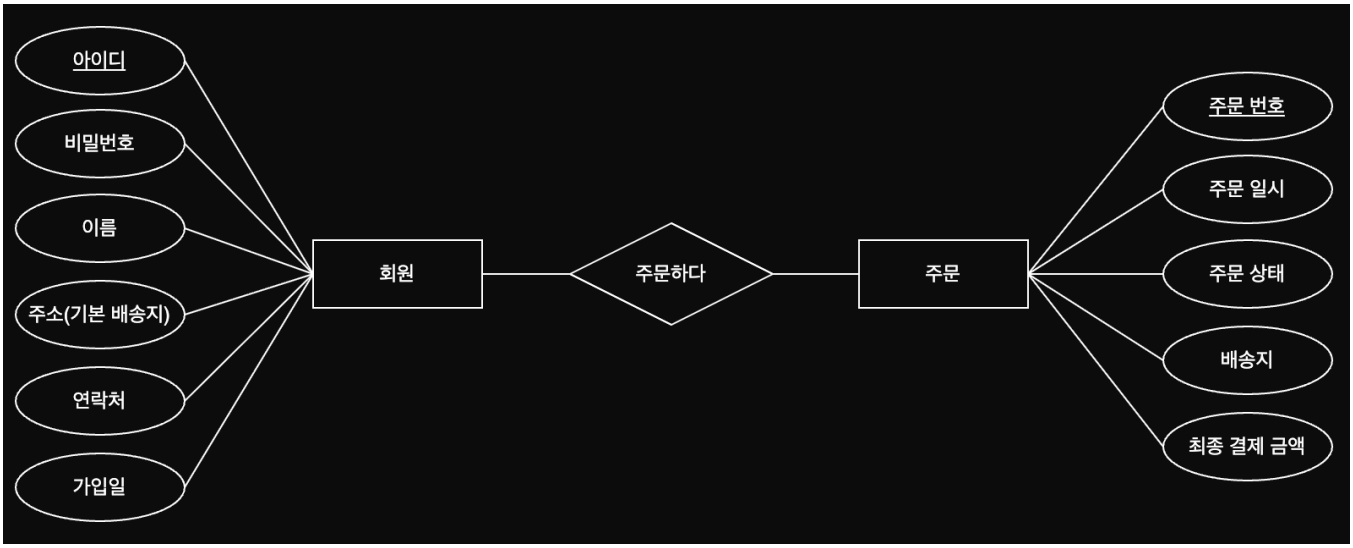
지금까지 분석한 모든 내용, 즉 엔티티, 속성, 식별자, 관계, 카디널리티, 참여도를 하나의 그림으로 표현한 것이 바로 **ERD(Entity-Relationship Diagram)**다. ERD는 복잡한 데이터 구조를 시각적으로 표현해서 개발자, 기획자 등 모든 관계자가 시스템의 청사진을 한눈에 파악하고 소통할 수 있게 도와주는 설계에서 가장 강력한 도구다.

> **그림 - 실무 이야기**
> 실무에서는 백마디 설명보다 잘 다듬어진 한 장의 그림이 훨씬 더 많은 사람들을 이해시키고 설득할 수 있다.

표기법의 이해와 실무 선택

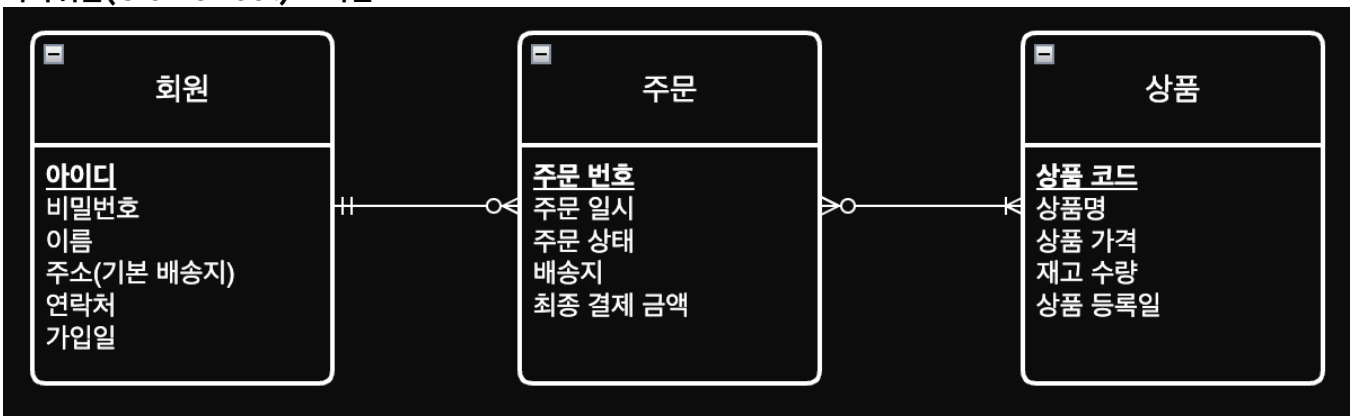
ERD를 그리는 방법에는 대표적으로 '피터 첸 표기법'과 '까마귀발 표기법'이 있다.

피터 첸(Peter Chen) 표기법



- 피터 첸 표기법은 엔티티는 사각형, 관계는 마름모, 속성은 타원으로 표현하는 전통적인 방식이다. 이런 개념을 처음 만든 표기법이라 학술적으로 의미가 있지만, 실무에서는 사용하지 않는다. 그림이 복잡해지고 공간을 많이 차지하기 때문이다. 강의에서도 피터 첸 표기법은 사용하지 않겠다.
- 예제에서 회원과 주문만 간단히 표현했는데도 너무 많은 공간을 차지한다.

까마귀발(Crow's Foot) 표기법



- 현재 실무에서 가장 널리 사용되는 **사실상의 표준**이다.

> **용어 - 사실상 표준(de facto standard)**
> 사실상 표준은 국가나 국제기구가 공식적으로 정한 표준은 아니지만, 시장에서 널리 사용되어 사실상 표준처럼 자리 잡은 기술이나 제품을 의미한다. 'De Facto'는 '사실상'이라는 의미의 라틴어이다.

왜 실무에서는 까마귀발 표기법을 사용할까?

1. **정보 밀도와 가독성**: 관계를 별도의 마름모로 그리지 않고 엔티티를 잇는 선 자체에 기호로 표현하므로, 훨씬 적은 공간에 더 많은 정보를 깔끔하게 표현할 수 있다.
2. **직관적인 표현**: 관계의 'Many(다)' 쪽을 나타내는 기호가 까마귀의 발 모양과 닮았다고 해서 이런 이름이 붙었다.

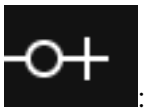
기호 자체가 관계의 수량(카디널리티)을 직관적으로 보여준다.

3. **대부분의 도구 지원**: MySQL Workbench, DBeaver, ERDCloud 등 우리가 사용하는 대부분의 데이터베이스 모델링 도구가 까마귀발 표기법을 기본으로 지원한다. 이 표기법만 알면 어떤 도구든 쉽게 사용할 수 있다.

까마귀발 표기법의 핵심 기호

- 엔티티는 사각형 상자로 표현한다.
- 관계는 두 엔티티를 잇는 직선으로 표현한다.
- 선의 양 끝에 기호를 붙여 **카디널리티**와 **참여도**를 동시에 표현한다.
  - `|` : 1 (One)
  - `<` : 다 (Many, 까마귀발 모양)
    -  : `<` 다는 실제로는 이렇게 발가락3개의 모양이다. 키보드로 이런 표현이 어려워서 `<`로 표현했다.
  - `0` : 0 (Zero)

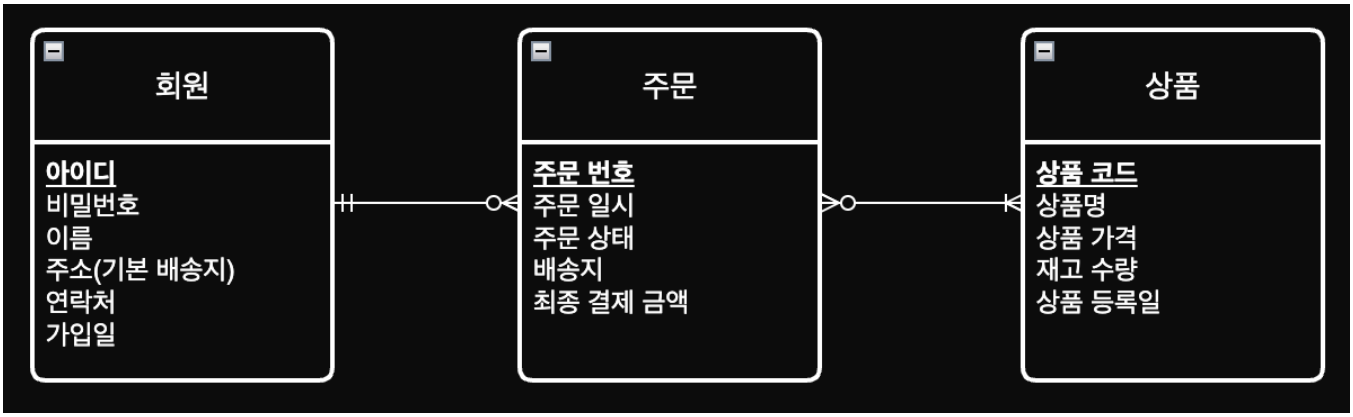
이 기호들을 조합하여 관계를 표현한다.

- `0|` (Zero or One): 0개 또는 1개 (선택적 1)
  -  : 실제 표현
- `||` (One and Only One): 반드시 1개 (필수적 1)
  -  : 실제 표현
- `0<` (Zero or Many): 0개 또는 여러 개 (선택적 N)
  -  : 실제 표현
- `|<` (One or Many): 1개 또는 여러 개 (필수적 N)
  -  : 실제 표현

이 표기법에 따라 우리의 '쇼핑몰' 관계를 다시 표현해보자.

- **회원 : 주문**
  - 한 명의 회원은 주문을 0개 이상 가질 수 있다(`0<`). - 선택적 N
  - 하나의 주문은 반드시 한 명의 회원을 가져야 한다(`||`). - 필수적 1
  - 최종 관계: `회원 (||)---(0<) 주문`
- **주문 : 상품 (M:N 관계)**

- M:N 관계는 ERD에서 두 엔티티를 직접 연결하고 양쪽에 `<` 기호를 표기한다.
- 하나의 주문에는 상품이 1개 이상 포함되어야 한다(`|<`). - 필수적 N
- 하나의 상품은 여러 주문에 포함될 수 있으며, 아직 한 번도 주문되지 않았을 수도 있다(`0<`). - 선택적 N
- 최종 관계: `주문 (>0)---(|<) 상품`



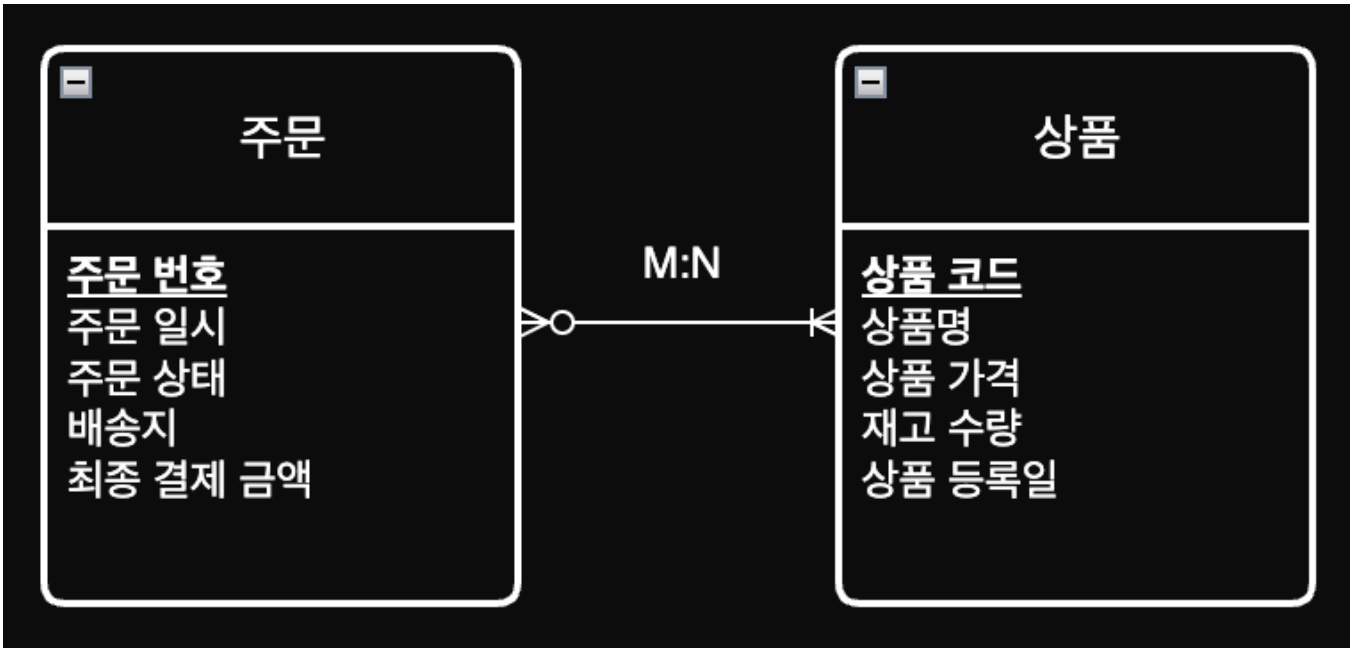
- 까마귀발 표기법을 읽을 때 가장 중요한 원칙은 **"한 엔티티 쪽 끝에 있는 기호는 반대편 엔티티에 대한 규칙을 설명한다"**는 것이다.
- `회원`은 여러 `주문`을 가질 수 있다.
- `회원`은 `주문`을 하지 않을 수 있다. (선택적 관계)
- `주문`은 반드시 하나의 `회원`에 포함되어야 한다. (필수적 관계)

이 모든 것을 종합하면 '쇼핑몰'의 초기 개념적 모델 ERD가 완성된다.

> **ERD 작성**
> 실제 ERD를 작성하는 것은 뒤에 실습에서 draw.io 툴을 사용해서 진행하겠다.

연관 엔티티 - 다대다 관계 해결

M:N 관계의 두 가지 문제점과 해결책



앞서 우리는 주문과 상품의 관계가 M:N(다대다)이라고 결론 내렸다. "하나의 주문에는 여러 상품이 포함될 수 있고, 하나의 상품은 여러 주문에 포함될 수 있다." 개념적으로는 타당하지만, 이 모델을 가지고 실제 데이터베이스 테이블을 만들려고 하면 두 가지 큰 문제에 부딪치게 된다.

문제 1: M:N 관계는 물리적으로 구현할 수 없다

개념적으로 M:N 관계를 그리는 것은 쉽지만, 이것을 실제 데이터베이스의 테이블 구조로 옮길 수는 없다. 왜 그럴까?

관계형 데이터베이스는 '테이블'이라는 2차원 표 형태를 사용하며, 관계는 한 테이블의 **기본 키(Primary Key)**를 다른 테이블의 **외래 키(Foreign Key)**로 포함시켜 표현한다.

이 방식을 M:N 관계에 적용해 보겠다. '네이트(회원id: 1)'가 주문 1001번으로 상품 101(키보드)과 상품 102(마우스)를 주문한 상황이다.

시도 1: 주문 테이블에 상품id 컬럼 추가하기 (1:N 방식)

주문 테이블에 주문한 상품의 ID를 저장해 보자.

주문 테이블

주문id (PK)	회원id	주문 일시	상품id (FK?)
1001	1	2025-08-06	101
???	1	2025-08-06	102

문제점: 주문 1001에 키보드(101)를 담았더니, 마우스(102)를 담을 방법이 없다. 마우스를 담기 위해 새로운 주문 행을 만들면, 그것은 더 이상 주문 1001이 아니게 된다. 이 구조는 하나의 주문에 단 하나의 상품만 가질 수 있으므로 "여러 상품을 한 번에 주문"한다는 요구 사항을 만족시키지 못한다.

시도 2: 컬럼 계속 추가하기

그렇다면 한 주문에 포함될 수 있는 상품의 최대 개수를 정해두고, 그만큼 컬럼을 미리 만들어두는 것은 어떨까? 예를 들어 한 번에 최대 5개의 상품을 주문할 수 있다고 가정해 보자.

주문 테이블 (잘못된 설계)

주문id (PK)	회원id	주문 일시	product1_id	product2_id	product3_id	...
1001	1	2025-08-06	101	102	NULL	...
1002	3	2025-08-07	103	NULL	NULL	...
1003	1	2025-08-08	102	103	104	...

이 방법 역시 심각한 문제점을 가지고 있다.

- **확장성 부재**: 만약 어떤 고객이 한 번에 6개의 상품을 주문하고 싶다면 어떻게 해야 할까? 데이터베이스 관리자는 `ALTER TABLE` 명령어로 `product6_id` 컬럼을 추가해야 한다. 비즈니스의 요구 사항이 바뀔 때마다 데이터베이스의 구조를 변경해야 하는 것은 매우 딱딱하고 확장성 없는 설계다.
- **공간 낭비**: 대부분의 주문은 한두 개의 상품으로 이루어질 것이다. 5개의 컬럼을 만들어두면, 대다수 주문 데이터에서 `product3_id`, `product4_id`, `product5_id` 컬럼은 항상 `NULL` 값으로 채워져 저장 공간을 낭비하게 된다.
- **데이터 검색의 어려움**: "무소음 마우스(ID: 102)가 포함된 모든 주문을 찾아달라"는 간단한 요청에 대한 SQL이 매우 복잡해진다. `WHERE product1_id = 102 OR product2_id = 102 OR product3_id = 102 OR ...` 와 같이 모든 컬럼을 OR 조건으로 엮어야 한다. 이는 쿼리 성능을 저하시키고 유지보수를 어렵게 만든다.

이처럼 컬럼을 무한정 늘리는 방식은 근본적인 해결책이 될 수 없다.

시도 3: 한 컬럼에 여러 값 욱여넣기 (절대 피해야 할 방식)

그렇다면 `상품ids` 라는 컬럼을 만들고, 여기에 주문한 상품 ID들을 텍스트로 모두 넣어보면 어떨까?

주문 테이블 (잘못된 설계)

주문id (PK)	회원id	주문 일시	상품ids
1001	1	2025-08-06	"101, 102"
1002	3	2025-08-07	"103"
1003	1	2025-08-08	"102, 103, 104"

문제점: 이 방식은 관계형 데이터베이스의 근간을 흔드는 최악의 설계다.

- **데이터 검색의 어려움**: "무소음 마우스(ID: 102)가 포함된 모든 주문을 찾아달라"는 간단한 요청에 `상품ids LIKE '%102%'` 와 같은 복잡하고 비효율적인 텍스트 검색을 해야 한다. 만약 상품 ID가 1102라면 이것까지 함께 검색되는 끔찍한 오류가 발생할 수 있다.
- **데이터 수정 및 삭제의 복잡성**: 주문 1003에서 '모니터(ID: 103)'만 빼달라는 요청이 오면, "102, 103, 104"라는 문자열을 읽어와서 103 부분만 잘라내고 다시 저장해야 한다. 데이터가 늘어날수록 관리는 매우 어려워진다.
- **원자성 위반**: 데이터베이스의 제1 정규형, 즉 "테이블의 모든 칸은 원자적인(더 이상 쪼갤 수 없는) 값 하나만 가져야 한다"는 원칙을 위반한다. (정규형은 뒤에서 설명한다.)

결론적으로, 두 테이블만으로는 M:N 관계를 표현할 방법이 존재하지 않는다. 이것이 우리가 마주한 첫 번째 문제다.

문제 2: 관계에 속한 데이터를 저장할 장소가 없다

두 번째 문제는 M:N 관계가 만들어지는 그 '순간'에만 의미를 갖는 데이터들을 저장할 장소가 없다는 점이다.

여기 주문과 상품 엔티티의 현재 속성 목록(테이블)이 있다.

주문 테이블

주문id (PK)	회원id	주문 일시	배송지
1001	1	2025-08-06	서울시 강남구
...	...	...	...

상품 테이블 - 주문 시점

상품id (PK)	상품명	가격	재고수량

101	기계식 키보드	**120000**	48
102	무소음 마우스	45000	118
...	...	...	...

'네이트'가 주문 1001로 '무소음 마우스' 1개를 45,000원에 '기계식 키보드'를 2개를 120,000원에 구매했다고 가정해보자.

상품 테이블 - 현재

상품id (PK)	상품명	가격	재고수량
101	기계식 키보드	**135000**	48
102	무소음 마우스	45000	118
...	...	...	...

그런데 상품의 가격은 달라질 수 있다. 기계식 키보드의 가격은 120,000 ➜ 135,000원으로 변경되었다.

기계식 키보드의 주문 수량(2개)과 주문 당시 가격(120,000원) 정보는 어디에 저장해야 할까?

주문 테이블에 저장할 수 있을까?

주문id (PK)	회원id	...	주문수량	주문가격
1001	1	...	3	???

주문 테이블에 `주문수량`, `주문가격` 컬럼을 추가한다고 가정해보자. `주문수량`에 총수량 3을 넣으면, 키보드를 몇 개, 마우스를 몇 개 샀는지 알 수 없다. `주문가격`은 더 심각하다. 키보드 가격과 마우스 가격 중 무엇을 넣어야 할까? 총액을 넣자니 이후에 각 상품의 주문 가격을 알 수 없게 된다.

상품 테이블에 저장할 수 있을까?
`상품` 테이블은 현재 상품 정보를 관리하는 곳이지, 과거의 특정 주문 정보를 저장하는 곳이 아니다. 이 테이블에 주문 수량이나 주문 당시 가격을 추가하는 것은 말이 되지 않는다.

이처럼 주문 수량과 주문 당시 가격은 주문이나 상품에 속한 속성이 아니다. 이들은 **주문과 상품이 관계를 맺음으로써 발생하는 속성**이다. 이런 추가 속성도 누군가는 데이터를 관리해야 한다. **데이터를 관리하는 역할은 바로 엔티티의 몫**이다.

해결책: 관계를 엔티티로 승격하자

이 두 가지 문제를 한 번에 해결하는 방법은 M:N 관계를 **연관 엔티티(Associative Entity)**로 바꾸는 것이다. 연관 엔티티는 두 엔티티의 관계를 나타내는 새로운 엔티티를 만들어, 기존의 M:N 관계를 두 개의 1:N, N:1 관계로 풀어내는 방식이다.

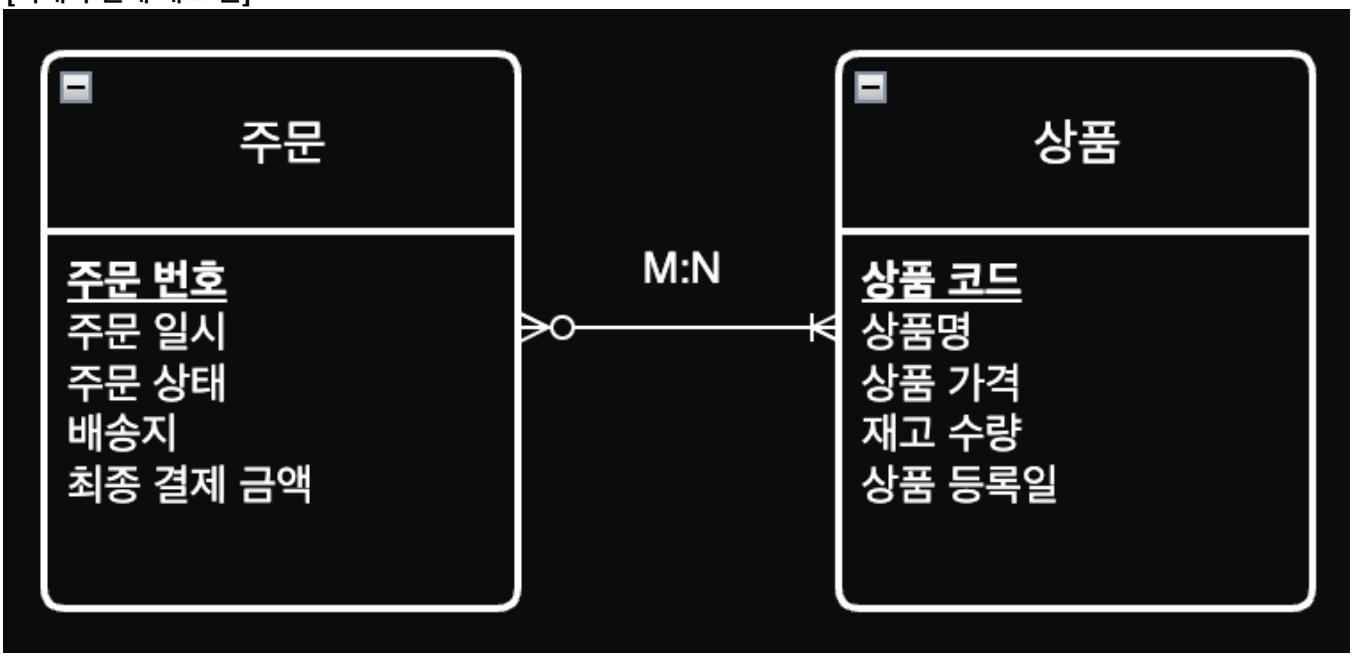
쉽게 이야기해서 관계를 엔티티로 만드는 것이다.

주문과 상품의 M:N 관계를 해소하기 위해 **주문 항목(order_item)**이라는 새로운 연관 엔티티를 만들자.

- M:N 관계를 두 개의 1:N 관계로 분해하여 문제 1을 해결한다.
- 새로운 엔티티에 관계 속성을 저장하여 문제 2를 해결한다.

이 새로운 구조가 실제 데이터로 어떻게 표현되는지 예시를 통해 확인해 보자.

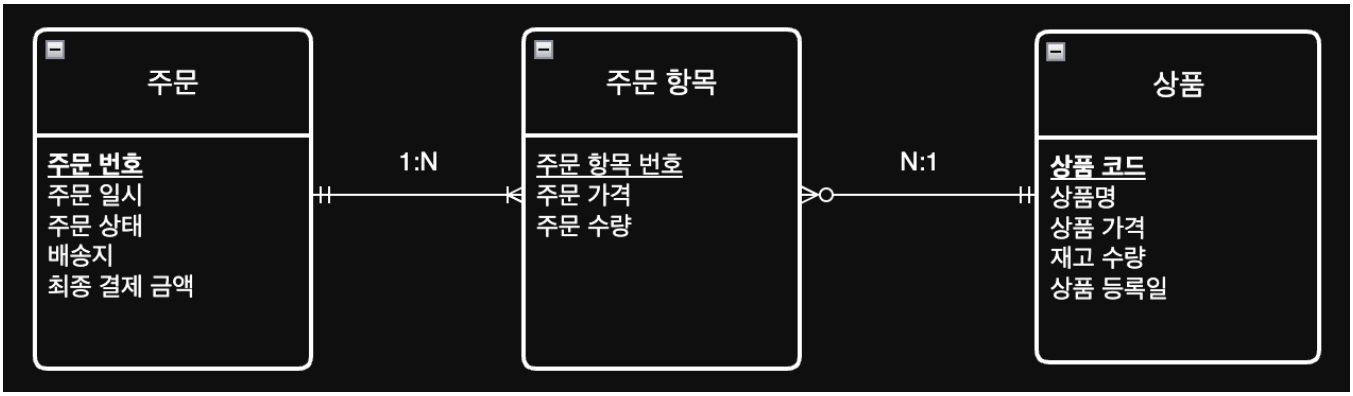
[다대다 관계 해소 전]



`주문` - `상품` 관계(M:N)

- 하나의 `주문`에는 최소 하나 이상의 `상품`이 포함된다.
- 하나의 `상품`은 여러 번 주문될 수도, 아직 한 번도 주문되지 않을 수도 있다.

[다대다 관계 해소 후]



- `주문 항목`이라는 연관 엔티티를 만들었다.

`주문` - `주문 항목` 관계 (1:N)

- 하나의 `주문`에는 최소 하나 이상의 `주문 항목`이 포함된다.
- 하나의 `주문 항목`은 특정한 `주문` 한 건에만 속한다.

`주문 항목` - `상품` 관계 (N:1)

- 하나의 `주문 항목`은 반드시 하나의 `상품`을 가져야 한다.
- 하나의 `상품`은 여러 번 주문될 수도, 아직 한 번도 주문되지 않을 수도 있다.

> **연관 엔티티의 다양한 용어**
> 개념적 모델링 단계의 **연관 엔티티(Associative Entity)**는 논리/물리적 모델링 단계에서는 **연결 테이블(Link Table)**, **조인 테이블(Join Table)**, **매핑 테이블(Mapping Table)** 등 다양한 이름으로 불린다.
> 어떤 용어를 사용하든 이는 **다대다 관계를 해소하고 관계에 대한 추가 데이터를 저장하기 위해 두 개 이상의 테이블을 연결하는 중간 테이블**을 의미한다.

최종 모델의 테이블 예시

지금은 개념적 모델링 단계이지만 이해를 돕기 위해 다대다 관계를 해소하는 경우 실제 테이블이 이떻게 만들어지는지 간단히 살펴보자. 필드는 최소화 했다.

회원 테이블

회원id (PK)	로그인id	회원명
1	nate123	네이트
2	chulsoo	이철수
3	younghee	김영희

상품 테이블

상품id (PK)	상품명	상품가격
101	기계식 키보드	135000
102	무소음 마우스	45000
103	4K 모니터	350000

주문 테이블

주문id (PK)	회원id (FK)	주문 일시
1001	1	2025-08-06
1002	3	2025-08-07
1003	1	2025-08-08
1004	2	2025-08-08

주문_항목 테이블 - 연결 테이블

주문_항목id (PK)	주문id (FK)	상품id (FK)	주문가격	주문수량
201	1001	101	120000	2
202	1001	102	45000	1
203	1002	103	350000	1
204	1003	102	45000	1
205	1004	101	135000	1
206	1004	103	350000	1

결과 분석

문제 1 해결: `주문_항목` 테이블은 `주문id`와 `상품id`를 각각의 컬럼으로 가짐으로써 M:N 관계를 완벽하게 풀어낸다. 주문 1001은 `주문_항목` 테이블의 두 행(201, 202)을 통해 두 개의 상품과 연결된다.

네이트가 주문한 주문 1001 내용으로만 추려서 확인해보자.

주문 테이블 (1001)

주문id (PK)	회원id (FK)	주문 일시
1001	1	2025-08-06

주문_항목 테이블 - 연결 테이블 (1001)

주문_항목id (PK)	주문id (FK)	상품id (FK)	주문가격	주문수량
201	1001	101	120000	2
202	1001	102	45000	1

상품 테이블 (1001 관련)

상품id (PK)	상품명	상품가격
101	기계식 키보드	135000
102	무소음 마우스	45000

1001 주문 내용 확인

- '네이트'는 주문 1001로 '무소음 마우스' 1개를 45,000원에 '기계식 키보드'를 2개를 120,000원에 구매한 것을 확인할 수 있다.
- 상품 테이블의 기계식 키보드 가격은 네이트가 주문한 120,000원 보다 올라서 135,000원으로 변경되었다.
- 네이트는 한 번에 2개의 상품을 주문했다.

문제 2 해결: `주문_항목` 테이블은 `주문가격`(주문 당시 가격)과 `주문수량` 속성을 가질 완벽한 공간이 된다.

- 주문 1001의 키보드는 할인 기간이라 120,000원에 2개 구매했다.
- 주문 1004의 키보드는 할인이 끝나 135,000원에 1개 구매했다.

주문_항목 테이블 - 연결 테이블(1001, 1004)

주문_항목id (PK)	주문id (FK)	상품id (FK)	주문가격	주문수량
201	1001	101	120000	2
205	1004	101	135000	1

이처럼 연관 엔티티를 도입함으로써, 물리적으로 구현이 불가능하고 중요한 데이터를 저장할 수 없었던 M:N 관계의 두 가지 문제를 모두 깔끔하게 해결할 수 있었다.

실무 팁 - M:N 관계는 반드시 풀고, 필요한 속성을 찾아라

개념적 모델링에서 M:N 관계를 발견했다면, 두 가지를 거의 확신해도 좋다.

- 첫째, 이 관계는 논리적/물리적 모델링 단계에서 거의 100% 연관 엔티티(연결 테이블)로 해소된다. M:N 관계를 그대로 두고는 테이블을 설계할 수 없기 때문이다.
- 둘째, 실무에서 마주하는 대부분의 M:N 관계는 그 자체로 추가적인 속성을 가진다. 우리 예제의 주문 수량, 주문 당시 가격처럼 말이다. 때로는 '주문 시점에 각 상품에 적용된 할인율' 같은 속성이 더 필요할 수도 있다. 특히 실무에서는 테이블에 데이터(행)를 추가한 일시(날짜)와 같은 정보를 기본으로 저장하기 때문에 대부분의 경우 추가적인 속성이 필요하다.

따라서 M:N 관계를 발견하면 "아, 여기에는 연관 엔티티가 필요하겠구나"라고 생각하는 것을 넘어, "이 관계가 만들어지는 순간에만 의미를 갖는 데이터는 무엇일까?"를 적극적으로 질문하고 찾아내야 한다.

용어 사전

프로젝트의 모든 이해 관계자들은 모두 같은 비즈니스 용어를 사용하는 것이 중요하다.
그리고 개념적, 논리적 모델링 단계에서는 비즈니스 담당자들과의 원활한 소통을 위해 '회원', '주문', '상품'과 같이 한글 용어를 사용하는 경우가 많다. 물리적 모델링 단계에서는 이 한글 용어들을 정한 규칙에 따라 영어로 정확하게 변환해야 한다.

이때 **용어 사전**을 만들어 관리하면 매우 유용하다.

용어 사전 예시

분류	명칭	전체 영문명	축약어	설명	관련 시스템 요소
엔티티	회원	`member`		서비스를 이용하는 사용자. 모든 사용자 정보의 기준이 되는 핵심 엔티티.	`member` 테이블
	상품	`product`		판매하는 물건. `prod`로 축약 가능하나 `product` 사용을 권장.	`product` 테이블
	주문	`order`		상품 구매 행위의 결과로 생성되는 데이터 집합. `order`는 예약어이므로 테이블명은 `orders` 사용.	`orders` 테이블
	항목	`item`		주문 내역, 장바구니 등 목록을 구성하는 개별 요소.	`order_item` 테이블
주요 속성	식별자	`identifier`	`id`	데이터를 고유하게 식별하는 번호. `[엔티티명]_id` 형식으로 사용.	`회원id`, `상품id`
	이름, 명	`name`		대상을 지칭하는 명칭.	`member_name`, `product_name`
	가격	`price`		물건의 가치. `product.price`(현재가), `order_item.주문가격`(주문시점가) 등 문맥에 맞게 사용.	`product.price`, `order_item.주문가격`

	주소	address	addr	위치 정보. ship과 조합하여 ship_addr 등으로 사용.	member.address, orders.ship_address
재고 물류	재고	stock		판매를 위해 보유 중인 상품의 재고.	product.stock_quantity
	SKU 재고 관리 단위	Stock Keeping Unit	SKU	재고를 관리하는 가장 작은 단위. **같은 상품이라도 옵션(색상, 사이즈)이 다르면 SKU가 다르다.**	product_option.sku_code
	배송	shipping	ship	상품을 고객에게 보내는 행위. ship_addr(배송지), ship_fee(배송비) 등으로 확장.	orders.ship_address
비즈니스 지표	PNL 손익	Profit and Loss	PNL	특정 기간 동안의 수익, 비용, 이익을 나타내는 지표. 경영 분석의 핵심.	BI 대시보드, 통계/정산 시스템
	MAU 월간 활성 사용자	Monthly Active Users	MAU	한 달간 서비스에 한 번 이상 방문한 중복 없는 사용자 수. 서비스 활성화 핵심 지표.	member, login_history 기반으로 집계
	ARPU 객단가	Average Revenue Per User	ARPU	1인당 평균 구매액. (총매출 / 구매 고객 수)	orders 테이블 기반으로 집계

기타	관리자	`administrator`	`admin`	시스템 관리 권한을 가진 사용자.	`member.role`

용어 사전이 왜 필요한가?

"이거 영어로 뭐라고 부르지?" 프로젝트를 진행하다 보면 개발자들 사이에서 이런 대화가 정말 많이 오간다. 한 번 상상해보자.

- **팀원 A**: "회원은 영어로 `member`가 자연스럽지." ➜ `member` 테이블을 만든다.
- **팀원 B**: 우리 서비스는 고객이니까 `customer`가 맞아." ➜ 코드 변수명은 `customer`로 쓴다.
- **팀원 C**: 우리 서비스의 대상은 사용자니까 `user`가 맞아." ➜ 코드 변수명은 `user`로 쓴다.

이런 시스템은 유지보수하기가 극도로 어렵고, 새로운 팀원이 투입되었을 때 시스템 구조를 파악하는 데 엄청난 시간과 비용을 낭비하게 만든다.

용어 사전은 이런 혼란을 막기 위한 **'프로젝트 용어의 헌법'**과 같다. 프로젝트 초기에 모든 구성원이 합의하여 비즈니스 용어와 실제 물리적인 데이터 이름을 매핑해두면, 모두가 일관된 용어를 사용하여 소통하고 개발할 수 있다. 이는 사소해 보이지만, 장기적으로는 개발 생산성과 시스템의 안정성을 극대화하는 매우 중요한 활동이다.

실무 중심의 용어 사전

단순히 단어를 나열하는 것을 넘어, 실무에서는 보다 체계적이고 상세한 용어 사전을 사용한다. 용어를 **역할별로 분류**하고, **약어와 전체 영문명을 명확히 구분**하며, **실제 시스템에서 어떻게 사용되는지** 구체적인 예시를 함께 기록하는 것이 좋다.

이렇게 잘 정리된 용어 사전은 단순한 명명 규칙을 넘어, 시스템의 구조와 비즈니스 로직을 이해하는 좋은 가이드가 된다. 그리고 이렇게 만든 용어 사전은 앞으로 우리가 만들 모든 데이터베이스 객체와 애플리케이션 코드의 기준이 된다.

단일어(Single Word) 중심

그렇다면 용어 사전을 어떻게 만드는 것이 가장 효율적일까? `회원ID`는 `회원id`, `상품가격`은 `product_price`처럼 복합 명사를 통째로 정의하는 방식보다는, **가장 작은 단위의 단일어**를 중심으로 사전을 구성하는 것이 좋다.

만약 `회원`을 `member`로, `상품`을 `product`로, `ID`를 `id`로, `가격`을 `price`로 각각 정의했다고 생각해보자. 이제 우리는 이 단어들을 **조립**해서 일관된 용어를 만들어낼 수 있다.

- 회원id = `member` + `id` = `member_id`
- 상품 가격 = `product` + `price` = `product_price`

이 방식은 `member_identifier`, `prd_price` 와 같은 변형을 원천적으로 차단하고, 시스템 전체의 예측 가능성을 극대화한다. 개발자는 사전에 정의된 단어를 조합하기만 하면 되므로, 새로운 컬럼이나 변수명을 만드는 데 드는 고민의 시간을 줄이고 혼란의 여지를 없앨 수 있다.

복합어를 용어 사전에 정의하는 것의 단점
복합어를 용어 사전에 하나하나 정의하는 방식은 다음과 같은 단점이 있다.

- **용어 사전의 비대화**: `회원ID`, `회원명`, `회원주소`, `상품ID`, `상품명`, `상품가격` ... 이렇게 모든 복합어를 개별적으로 등록하면 용어 사전이 급격하게 커지고 복잡해진다. 아마도 모든 엔티티에 있는 거의 모든 속성 수 만큼 커질 것이다. 또한 새로운 속성이 추가될 때마다 사전에 새로운 항목을 계속해서 만들어야 한다.
- **일관성 유지의 어려움**: 만약 누군가 '회원 번호'라는 용어를 `member_num`으로 사전에 추가했다고 가정해보자. 다른 개발자는 `회원id`를 사용하고 있었을 수 있다. 이처럼 비슷한 의미의 용어가 다른 이름으로 등록될 가능성이 높아져 시스템 전반의 명명 규칙이 깨지기 쉽다.
- **유연성 및 재사용성 저하**: 단일어를 정의하면 `id`라는 용어를 `member`, `product`, `order` 등 다양한 엔티티에 일관되게 재사용할 수 있다. 하지만 `회원id`를 통째로 정의하면, `상품id`를 만들어야 할 때 `회원id`라는 정의는 아무런 도움이 되지 않는다.

결론적으로, 잘 정의된 단일어 사전은 레고 블록처럼 조합하여 무한한 용어를 일관되게 만들어내는 강력한 기반이 된다.

비즈니스 특화 용어도 포함해야 하는 이유

개발팀과 비즈니스팀(기획, 운영 등)은 종종 다른 언어를 사용한다. 예를 들어, 기획자는 "SKU 별로 재고를 관리해야 해요"라고 말하지만, 개발자는 SKU가 무엇인지 모를 수 있다.

SKU(Stock Keeping Unit), **PNL(Profit and Loss)** 과 같은 비즈니스 전문 용어나 약어를 용어 사전에 명확히 정의해두면 이러한 간극을 메울 수 있다. 이는 단순히 개발자 간의 소통을 넘어, 프로젝트에 참여하는 **모든 구성원의 공통 언어(Ubiquitous Language)**를 만드는 핵심적인 과정이다. 그리고 비즈니스의 요구 사항이 코드에까지 정확하게 반영되도록 돕는 중요한 다리 역할을 한다.

축약어 통일

축약어를 남발하면, 축약어의 원문을 이해하기 어려운 경우가 있다. 예를 들어서 `member_register_datetime`이

라고 하면 바로 이해가 되지만 `mem_reg_dt`라고 하면 이해가 어려울 수 있다.
축약어는 반드시 용어사전에 등록하고 모두의 동의하에 사용해야 한다.
예를 들어 `dt`, `ship`, `addr` 같은 축약어를 용어 사전에 등록해두면, 모두가 공통으로 사용하는 축약어를 쉽게 알 수 있다.

용어 사전은 개념적 모델링 단계에서 모두 완성되는 것은 아니다.
개념적, 논리적, 물리적 모델링을 거치면서 새로운 용어들이 등장하고, 또 협의 하에 기존 용어들을 변경하는 경우도 자주 있다. 서비스 오픈 이후에도 서비스가 운영되면서 새로운 기능이 추가되며, 새로운 용어가 등장할 수 있다.

> ✨ **실무 팁 - 용어 사전은 살아있는 문서다**
> 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.

> ✨ **실무 팁 - 용어 사전은 필수다.**
> 개인적으로 용어 사전은 실무에서 가장 쓸모있는 중요한 문서 중 하나라 생각한다.
> 용어 사전은 모든 이해 관계자들을 하나로 만들고, 특히 개발자들에게는 변수명을 짓는 시간을 줄여준다.
> 용어 사전은 데이터베이스는 물론이고 애플리케이션 코드에 까지 시스템의 모든 영역에 일관성을 부여한다.

정리

요구 사항 분석과 핵심 요소 식별

- 개념적 모델링은 기술적인 용어 대신 모두가 이해할 수 있는 그림과 용어로 소통하며, 만들려는 데이터 세상의 청사진에 대해 합의하는 과정이다.
- 요구 사항 분석 시 '명사'는 엔티티 또는 속성, '동사'는 관계가 될 확률이 높다.
- '주문하다'와 같은 행위(동사)의 결과로 생성되는 '주문 기록(명사)'처럼, 시스템이 정보를 저장하고 관리해야 할 대상은 엔티티로 도출한다.

엔티티란?

- 엔티티는 데이터를 저장하고 관리해야 할 유무형의 대상을 의미하며, 데이터베이스의 테이블에 해당한다.
- 좋은 엔티티는 업무 관련성, 식별 가능성, 2개 이상의 속성 보유, 인스턴스의 집합, 다른 엔티티와의 관계라는 특징을 가진다.
- 엔티티는 개념적인 틀(엑셀 시트)이며, 그 안에 담긴 실제 데이터 하나하나(행)를 인스턴스라고 한다.

엔티티 분류1

- 엔티티를 성격과 역할에 따라 분류하면 데이터의 본질을 파악하고 성능 최적화 같은 설계 전략을 수립하는 데 도움이 된다.
- **존재 형태**에 따라 물리적 실체가 있는 '유형 엔티티', 추상적 개념인 '개념 엔티티', 업무 행위를 기록하는 '사건 엔티티'로 분류한다.
- **역할 및 발생 시점**에 따라 독립적인 '기본 엔티티', 업무의 중심인 '중심 엔티티', 상세 내역인 '행위 엔티티'로 분류하며, 이는 데이터의 흐름과 개발 우선순위를 결정하는 기준이 된다.
- 사건, 행위 엔티티는 데이터가 폭발적으로 증가할 가능성이 높으므로, 설계 초기부터 인덱스, 파티셔닝, 아카이빙 전략을 고려해야 한다.

엔티티 분류2

- **존재 종속성**에 따라 다른 엔티티에 의존하지 않고 독립적으로 존재하는 '강한 엔티티'와, 다른 엔티티(소유 엔티티)가 있어야만 존재 의미를 가지는 '약한 엔티티'로 나뉜다.
- 이 분류는 부모 없는 '고아 데이터' 발생을 막아 데이터의 정합성을 보장한다.
- 다대다(M:N) 관계를 해소하기 위해 '연관 엔티티'를 사용하고, 공통점과 차이점이 있는 엔티티 그룹을 표현하기 위해 '슈퍼타입/서브타입' 구조를 사용한다.

속성과 식별자

- 속성(Attribute)은 엔티티가 가지는 구체적인 정보이며, 데이터베이스 테이블의 컬럼(Column)이 된다.
- 식별자(Identifier)는 엔티티 내의 각 인스턴스(행)를 유일하게 구별해주는 속성이며, 테이블의 기본 키(Primary Key) 역할을 한다.

카디널리티와 참여도

- 관계를 상세화하기 위해 카디널리티와 참여도를 정의한다.
- **카디널리티(Cardinality)**는 한 엔티티의 인스턴스가 다른 엔티티의 인스턴스와 맺을 수 있는 최대 관계 수(1:1, 1:N, M:N)를 나타낸다.
- **참여도(Optionality)**는 관계의 참여가 필수적인지(Mandatory) 선택적인지(Optional)를 나타낸다.

ERD 완성하기

- ERD(Entity-Relationship Diagram)는 엔티티, 속성, 관계 등을 시각적으로 표현한 데이터 구조의 청사진이다.
- 실무에서는 정보 밀도가 높고 직관적이며 대부분의 도구에서 지원하는 **까마귀발(Crow's Foot) 표기법**을 사실상

의 표준으로 사용한다.

- 까마귀발 표기법은 선의 양 끝에 기호를 붙여 카디널리티와 참여도를 동시에 표현한다.

연관 엔티티 - 다대다 관계 해결

- 다대다(M:N) 관계는 관계형 데이터베이스에서 물리적으로 구현할 수 없고, '주문 수량'이나 '주문 당시 가격'처럼 관계에 종속된 속성을 저장할 공간이 없는 문제를 가진다.
- 이 문제를 해결하기 위해 두 엔티티 사이에 **연관 엔티티(Associative Entity)**를 새로 만든다.
- 연관 엔티티는 기존의 다대다 관계를 두 개의 일대다(1:N) 관계로 풀어내어 문제를 해결하고, 관계에 대한 추가 속성을 저장할 공간을 제공한다.

용어 사전

- 용어 사전은 프로젝트의 모든 구성원이 사용하는 비즈니스 용어와 실제 데이터베이스 객체명을 일관되게 매핑하여 관리하는 문서이다.
- 용어 혼란을 막아 개발 생산성과 시스템 안정성을 높이는 매우 중요한 역할을 한다.
- `회원(member)`, `가격(price)`처럼 가장 작은 단위의 단일어를 중심으로 정의하고, 이를 조합하여 사용하는 방식이 효율적이다.
- 용어 사전은 프로젝트 진행에 따라 지속적으로 업데이트되는 **살아있는 문서**로 관리해야 한다.