연관관계 매핑 - 심화1, 상속관계 매핑

관계형 데이터베이스에는 객체 지향에서 다루는 상속이라는 개념이 없다. 대신에 슈퍼타입 서브타입 관계(Super-Type Sub-Type Relationship)라는 모델링 기법이 객체의 상속 개념과 가장 유사하다.

슈퍼타입 서브타입 관계 매핑

데이터베이스 설계 모델 중 객체 상속과 가장 비슷한 것은 슈퍼타입 서브타입 관계다. ORM에서 이야기하는 상속 관계 매핑은 객체의 상속 구조와 데이터베이스의 슈퍼타입 서브타입 관계를 매핑하는 것이다.

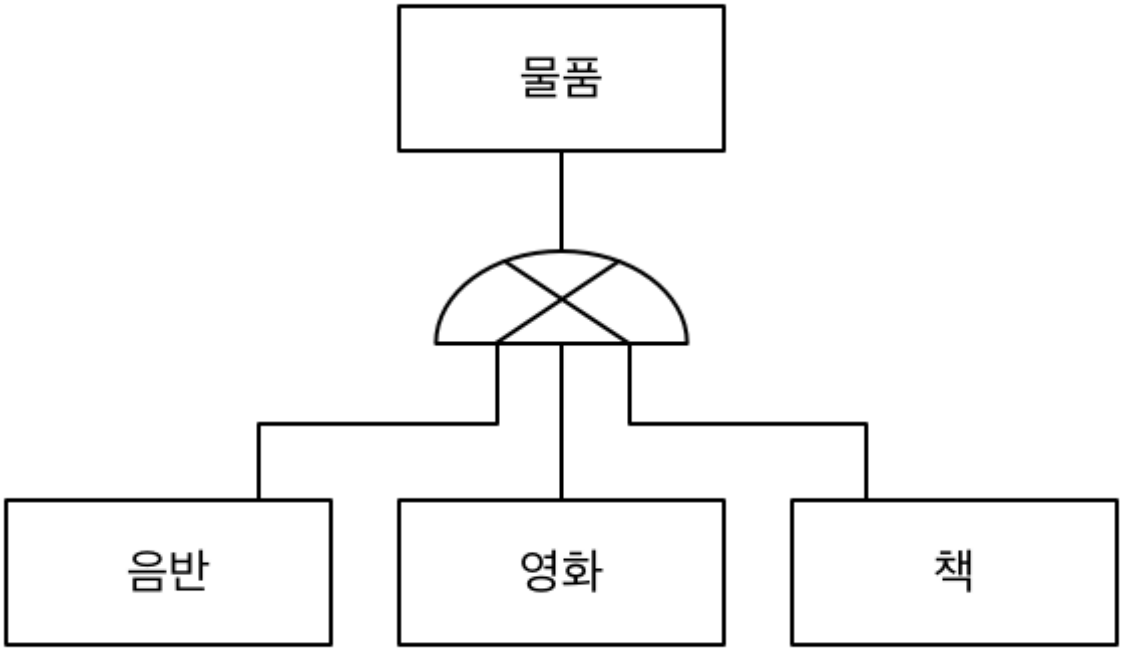

그림 - 슈퍼타입 서브타입 논리 모델

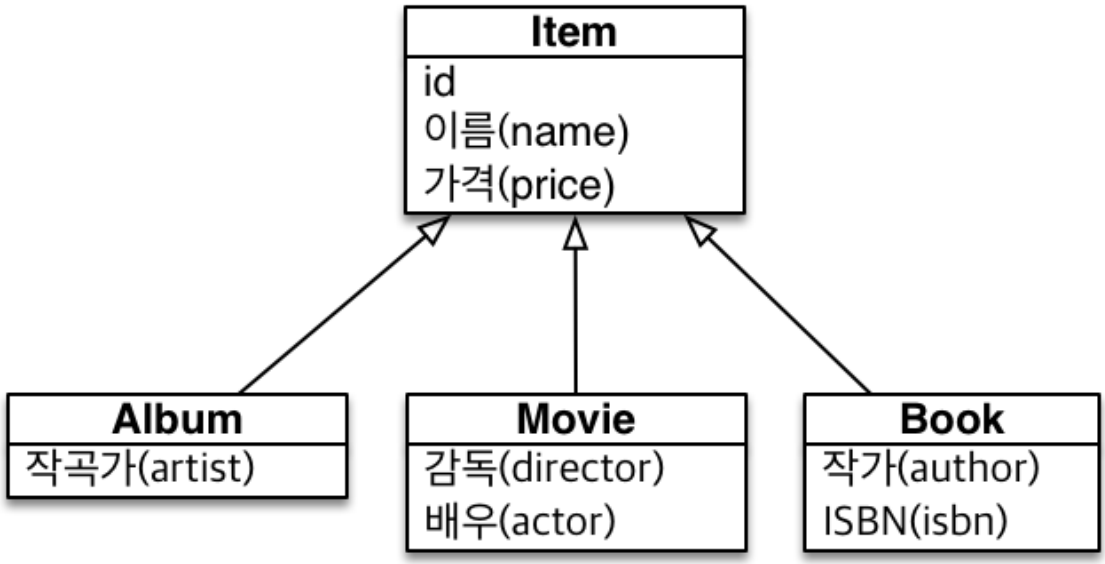

그림 - 객체 상속 모델

슈퍼타입 서브타입 논리 모델을 실제 물리 모델인 테이블로 구현할 때는 3가지 방법을 선택할 수 있다.

- **각각의 테이블로 변환**: 다음에 나오는 [그림 - JOINED TABLE]처럼 각각을 모두 테이블로 만들고 조회할 때 조인을 사용한다. JPA에서는 **조인 전략**이라 한다.
- **통합 테이블로 변환**: 다음에 나오는 [그림 - SINGLE TABLE]처럼 테이블을 하나만 사용해서 통합한다. JPA에서는 **단일 테이블 전략**이라 한다.
- **서브타입 테이블로 변환**: 다음에 나오는 [그림 - CONCRETE TABLE]처럼 서브 타입마다 하나의

테이블을 만든다. JPA에서는 **구현 클래스마다 테이블 전략**이라 한다.

[그림 - 객체 상속 모델]을 3가지 방법으로 매핑해보자.

- 조인 전략 (Joined Strategy)

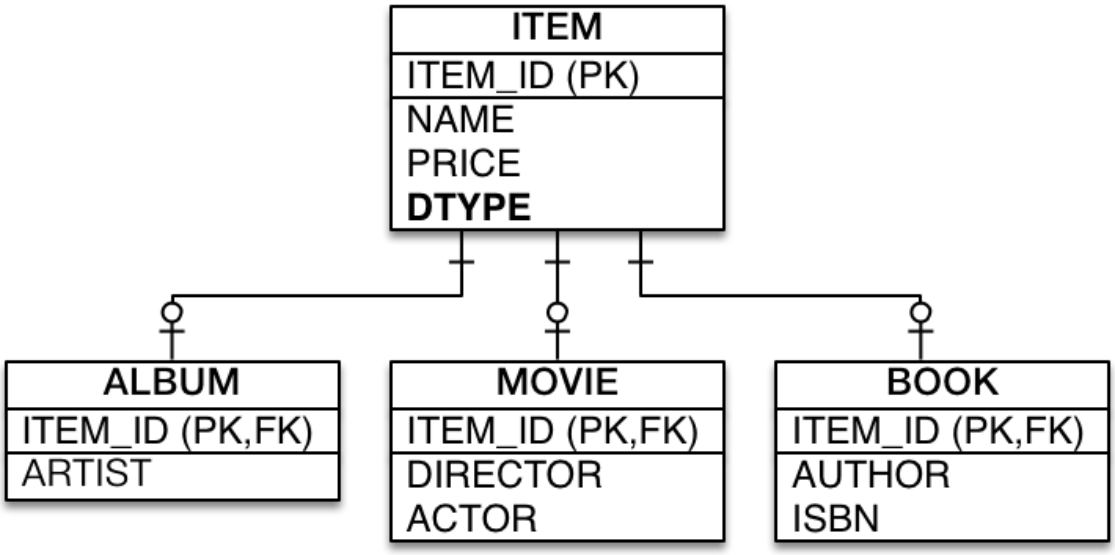

그림 - JOINED TABLE

조인 전략은 엔티티 각각을 모두 테이블로 만들고 자식 테이블이 부모 테이블의 기본 키를 받아서 기본 키 + 외래 키로 사용하는 전략이다. 따라서 조회할 때 조인을 자주 사용한다. 이 전략을 사용할 때 주의할 점이 있는데 객체는 타입으로 구분할 수 있지만 테이블은 타입의 개념이 없다. 따라서 타입을 구분하는 컬럼을 추가해야 한다. 여기서는 `DTYPE` 컬럼을 구분 컬럼으로 사용한다.

조인 전략을 사용한 예제 코드를 보자.

==== Item ====

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name; //이름
    private int price;   //가격
    ...
```

```
}
```

==== Album ====

```
@Entity
@DiscriminatorValue("A")
public class Album extends Item {

    private String artist;
    ...
}
```

==== Movie ====

```
@Entity
@DiscriminatorValue("M")
public class Movie extends Item {

    private String director; //감독
    private String actor;    //배우
    ...
}
```

매핑 정보를 분석해보자.

- `@Inheritance(strategy = InheritanceType.JOINED)` : 상속 매핑은 부모 클래스에 `@Inheritance` 를 사용해야 한다. 그리고 매핑 전략을 지정해야 하는데 여기서는 조인 전략을 사용하므로 `InheritanceType.JOINED` 를 사용했다.
- `@DiscriminatorColumn(name = "DTYPE")` : 부모 클래스에 구분 컬럼을 지정한다. 이 컬럼으로 저장된 자식 데이터를 구분할 수 있다. 기본값이 `DTYPE` 이므로 `@DiscriminatorColumn` 으로 줄여 사용해도 된다.
- `@DiscriminatorValue("M")` : 엔티티를 저장할 때 구분 컬럼에 입력할 값을 지정한다. 만약 영화 엔티티를 저장하면 구분 컬럼인 `DTYPE` 에 값 `M` 이 저장된다.

기본값으로 자식 테이블은 부모 테이블의 ID 컬럼명을 그대로 사용하는데 만약 자식 테이블의 기본 키 컬럼명을 변경하고 싶으면 `@PrimaryKeyJoinColumn` 을 사용하면 된다. 다음 예를 보자.

==== Book ====

```
@Entity
@DiscriminatorValue("B")
@PrimaryKeyJoinColumn(name = "BOOK_ID") //ID 재정의
public class Book extends Item {

    private String author; //작가
    private String isbn;   //ISBN
    ...
}
```

`BOOK` 테이블의 `ITEM_ID` 기본 키 컬럼명을 `BOOK_ID` 로 변경했다.

조인 전략 정리

- 장점
  - 테이블이 정규화된다.
  - 외래 키 참조 무결성 제약조건을 활용할 수 있다.
  - 저장공간을 효율적으로 사용한다.

- 단점
  - 조회할 때 조인이 많이 사용되어 성능이 저하될 수 있다.
  - 조회 쿼리가 복잡하다.
  - 데이터를 등록할 INSERT SQL을 두 번 실행한다.

- 특징
  - JPA 표준 명세는 구분 컬럼을 사용하도록 하지만 하이버네이트를 포함한 몇몇 구현체는 구분 컬럼(`@DiscriminatorColumn`) 없이도 동작한다.

- 관련 어노테이션
  - `@PrimaryKeyJoinColumn` , `@DiscriminatorColumn` , `@DiscriminatorValue`

- 단일 테이블 전략 (Single-Table Strategy)

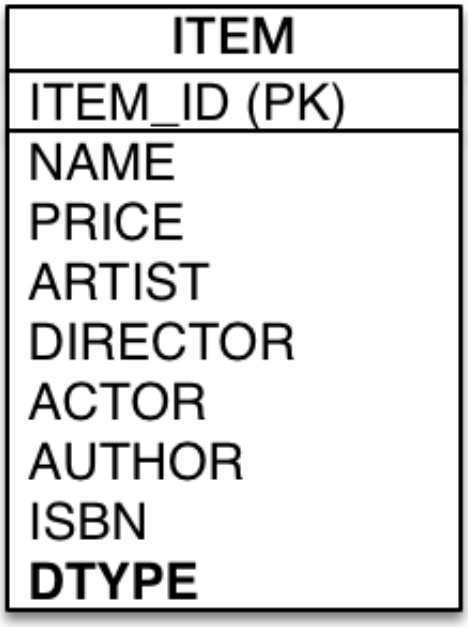

그림 - SINGLE TABLE

단일 테이블 전략은 이름 그대로 테이블을 하나만 사용한다. 그리고 구분 컬럼(`DTYPE`)으로 어떤 자식 데이터가 저장되었는지 구분한다. 조회할 때 조인을 사용하지 않으므로 일반적으로 가장 빠르다. 이 전략을 사용할 때 주의점은 자식 엔티티가 매핑한 컬럼은 모두 `null` 을 허용해야 한다는 점이다. 왜냐하면 예를 들어 `Book` 엔티티를 저장하면 `ITEM` 테이블의 `AUTHOR` , `ISBN` 컬럼만 사용하고 다른 엔티티와 매핑된 `ARTIST` , `DIRECTOR` , `ACTOR` 컬럼은 사용하지 않으므로 `null` 이 입력되기 때문이다.

==== Item ====

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name; //이름
    private int price;   //가격
    ...
}
```

==== Album ====

```
@Entity
@DiscriminatorValue("A")
public class Album extends Item { ... }
```

==== Movie ====

```
@Entity
@DiscriminatorValue("M")
public class Movie extends Item { ... }
```

==== Book ====

```
@Entity
@DiscriminatorValue("B")
public class Book extends Item { ... }
```

전략을 `InheritanceType.SINGLE_TABLE` 로 지정하면 단일 테이블 전략을 사용한다. 테이블 하나에 모든 것을 통합하므로 구분 컬럼을 필수로 사용해야 한다. 단일 테이블 전략의 장단점은 하나의 테이블을 사용하는 특징과 관련 있다.

단일 테이블 전략 정리

- 장점
  - 조인이 필요 없으므로 일반적으로 조회 성능이 빠르다.
  - 조회 쿼리가 단순하다.

- 단점
  - 자식 엔티티가 매핑한 컬럼은 모두 `null` 을 허용해야 한다.
  - 단일 테이블에 모든 것을 저장하므로 테이블이 커질 수 있다. 그러므로 상황에 따라서는 조회 성능이 오히려 느려질 수 있다.

- 특징
  - 구분 컬럼을 꼭 사용해야 한다. 따라서 `@DiscriminatorColumn` 을 꼭 설정해야 한다.
  - `@DiscriminatorValue` 를 지정하지 않으면 기본으로 엔티티 이름을 사용한다. (예 `Moive` , `Album` , `Book`)

- 구현 클래스마다 테이블 전략 (Table-per-Concrete-Class Strategy)

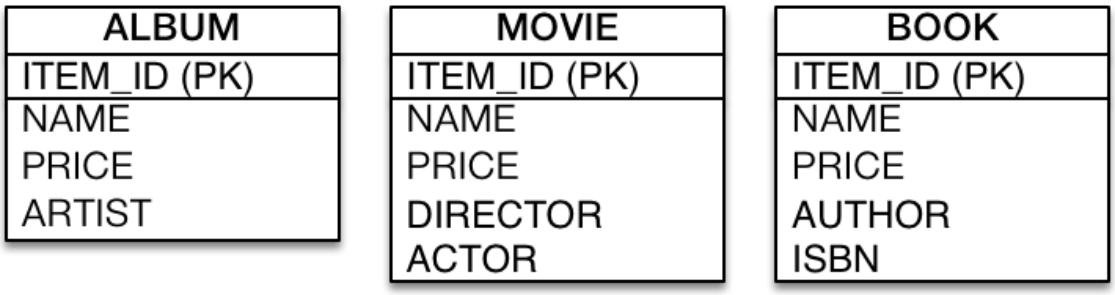

그림 - CONCRETE TABLE

이 전략은 자식 엔티티마다 테이블을 만든다. 그리고 자식 테이블 각각에 필요한 컬럼이 모두 있다.

==== Item ====

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name; //이름
    private int price;   //가격
    ...
}
```

==== Album ====

```
@Entity
public class Album extends Item { ... }
```

==== Movie ====

```
@Entity
public class Movie extends Item { ... }
```

==== Book ====

```
@Entity
public class Book extends Item { ... }
```

`InheritanceType.TABLE_PER_CLASS`를 선택하면 구현 클래스마다 테이블 전략을 사용한다. 이 전략은 자식 엔티티마다 테이블을 만든다. 일반적으로 추천하지 않는 전략이다.

- 장점
  - 서브 타입을 구분해서 처리할 때 효과적이다.
  - `not null` 제약조건을 사용할 수 있다.

- 단점
  - 여러 자식 테이블을 함께 조회할 때 성능이 느리다. (SQL에 UNION을 사용해야 한다.)
  - 자식 테이블을 통합해서 쿼리하기 어렵다.

- 특징
  - 구분 컬럼을 사용하지 않는다.

이 전략은 데이터베이스 설계자와 ORM 전문가 둘다 추천하지 않는 전략이다. 조인이나 단일 테이블 전략을 고려하자.

@MappedSuperclass

지금까지 학습한 상속 관계 매핑은 부모 클래스와 자식 클래스를 모두 데이터베이스 테이블과 매핑했다.

부모 클래스는 테이블과 매핑하지 않고 부모 클래스를 상속 받는 자식 클래스에게 매핑 정보만 제공하고 싶으면 `@MappedSuperclass`를 사용하면 된다.

`@MappedSuperclass`는 비유를 하자면 추상 클래스와 비슷한데 `@Entity`는 실제 테이블과 매핑 되지만 `@MappedSuperclass`는 실제 테이블과는 매핑되지 않는다. 이것은 **단순히 매핑 정보를 상속할 목적**으로만 사용된다.

예제를 통해 `@MappedSuperclass`를 알아보자.

그림 - `@MappedSuperclass` 설명 테이블

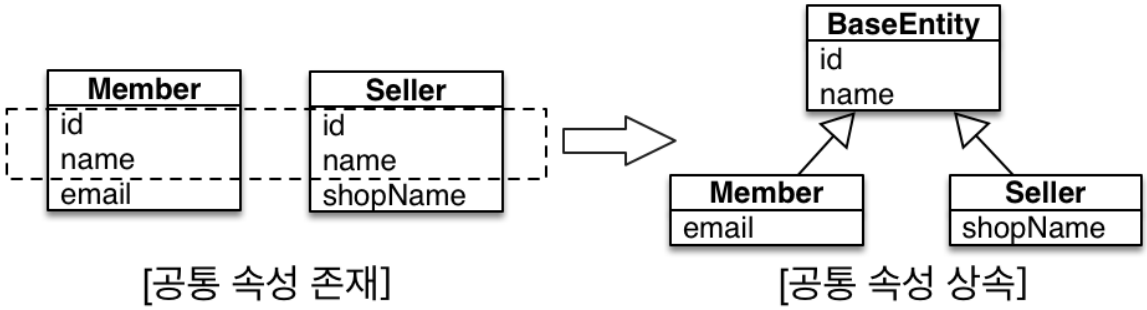

그림 - `@MappedSuperclass` 설명 객체

회원(`Member`)과 판매자(`Seller`)는 서로 관계가 없는 테이블과 엔티티다. 테이블은 그대로 두고 객체 모델의 `id` , `name` 두 공통 속성을 부모 클래스로 모으고 객체 상속 관계로 만들어보자.

==== `BaseEntity` ====

```
@MappedSuperclass //**
public abstract class BaseEntity {

    @Id @GeneratedValue
    private Long id;
    private String name;
    ...
}
```

==== `Member` ====

```
@Entity
public class Member extends BaseEntity {

    //ID 상속
    //NAME 상속
    private String email;
    ...
```

```
}
```

==== `Seller` ====

```
@Entity
public class Seller extends BaseEntity {

    //ID 상속
    //NAME 상속
    private String shopName;
    ...
}
```

`BaseEntity`에는 객체들이 주로 사용하는 공통 매핑 정보를 정의했다. 그리고 자식 엔티티들은 상속을 통해 `BaseEntity`의 매핑 정보를 물려받았다. 여기서 `BaseEntity`는 테이블과 매핑할 필요가 없고 자식 엔티티에게 공통으로 사용되는 매핑 정보만 제공하면 된다. 따라서 `@MappedSuperclass`를 사용했다.

재정의

부모로부터 물려받은 매핑 정보를 재정의 하려면 `@AttributeOverrides`나 `@AttributeOverride`를 사용하고, 연관관계를 재정의 하려면 `@AssociationOverrides`나 `@AssociationOverride`를 사용한다.

==== 매핑 정보 재정의 ====

```
@Entity
@AttributeOverride(name = "id", column = @Column(name = "MEMBER_ID"))
public class Member extends BaseEntity { ... }
```

부모에게 상속 받은 `id` 속성의 컬럼명을 `MEMBER_ID`로 재정의 했다.

둘 이상을 재정의 하려면 `@AttributeOverrides`를 사용하면 된다.

```
@Entity
@AttributeOverrides({
      @AttributeOverride(name = "id", column = @Column(name = "MEMBER_ID")),
      @AttributeOverride(name = "name", column = @Column(name = "MEMBER_NAME"))
})
public class Member extends BaseEntity { ... }
```

정리

`@MappedSuperclass`의 특징

- 테이블과 매핑되지 않고 자식 클래스에 엔티티의 매핑 정보를 상속하기 위해 사용한다.
- `@MappedSuperclass`로 지정한 클래스는 엔티티가 아니므로 `em.find()`나 JPQL에서 사용할 수 없다.
- 이 클래스를 직접 생성해서 사용할 일은 거의 없으므로 추상 클래스로 만드는 것을 권장한다.

정리하자면 `@MappedSuperclass`는 테이블과는 관계가 없고 단순히 엔티티가 공통으로 사용하는 매핑 정보를 모아주는 역할을 할 뿐이다. ORM에서 이야기하는 진정한 상속 매핑은 이전에 학습한 객체 상속을 데이터베이스의 슈퍼타입 서브타입 관계와 매핑하는 것이다.

`@MappedSuperclass`를 사용하면 등록일자, 수정일자, 등록자, 수정자 같은 여러 엔티티에서 공통으로 사용하는 속성을 효과적으로 관리할 수 있다.

> **참고:** 엔티티(`@Entity`)는 엔티티(`@Entity`)이거나 `@MappedSuperclass`로 지정한 클래스만 상속받을 수 있다.

[실전 예제] - 4. 상속 관계 매핑하기

> **예제 코드 :** `ch07-model4`

다음 요구사항이 추가되었다.

- 상품의 종류는 음반, 도서, 영화가 있고 이후 더 확장될 수 있다.
- 모든 데이터는 등록일과 수정일이 있어야 한다.

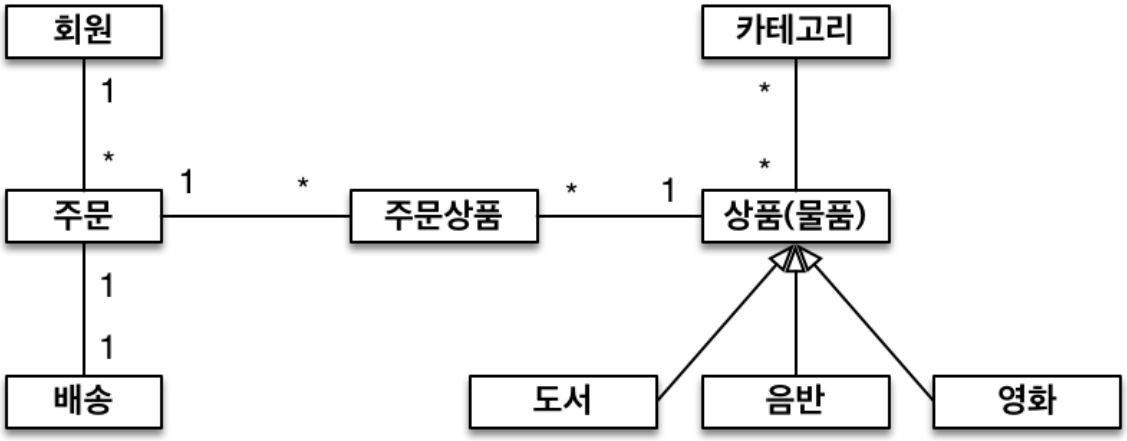

그림 - UML4

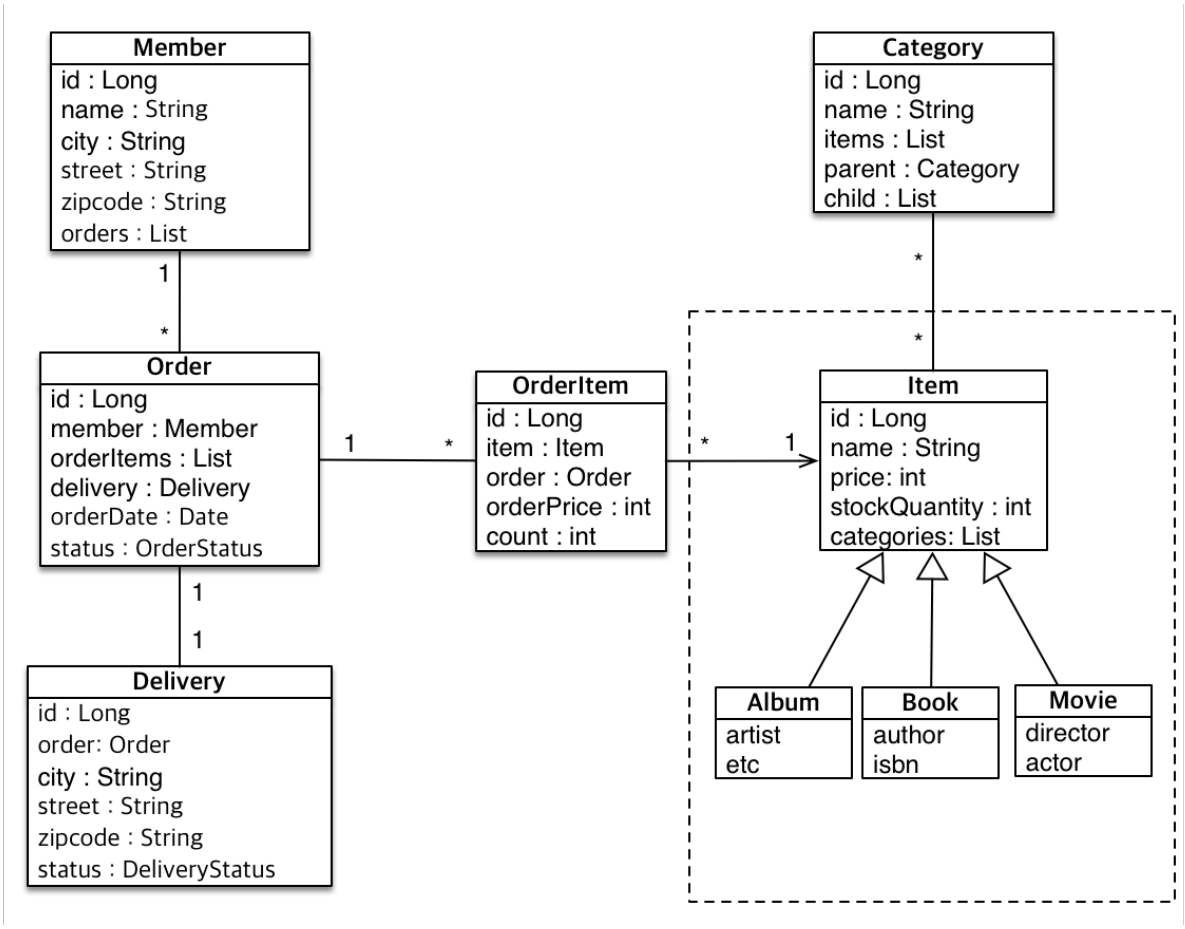

그림 - UML 상세4

엔티티를 상속관계로 만들고 공통 속성은 `Item` 엔티티에 두었다. 그리고 요구사항대로 `Album`, `Book`, `Movie` 자식 엔티티를 추가했다.

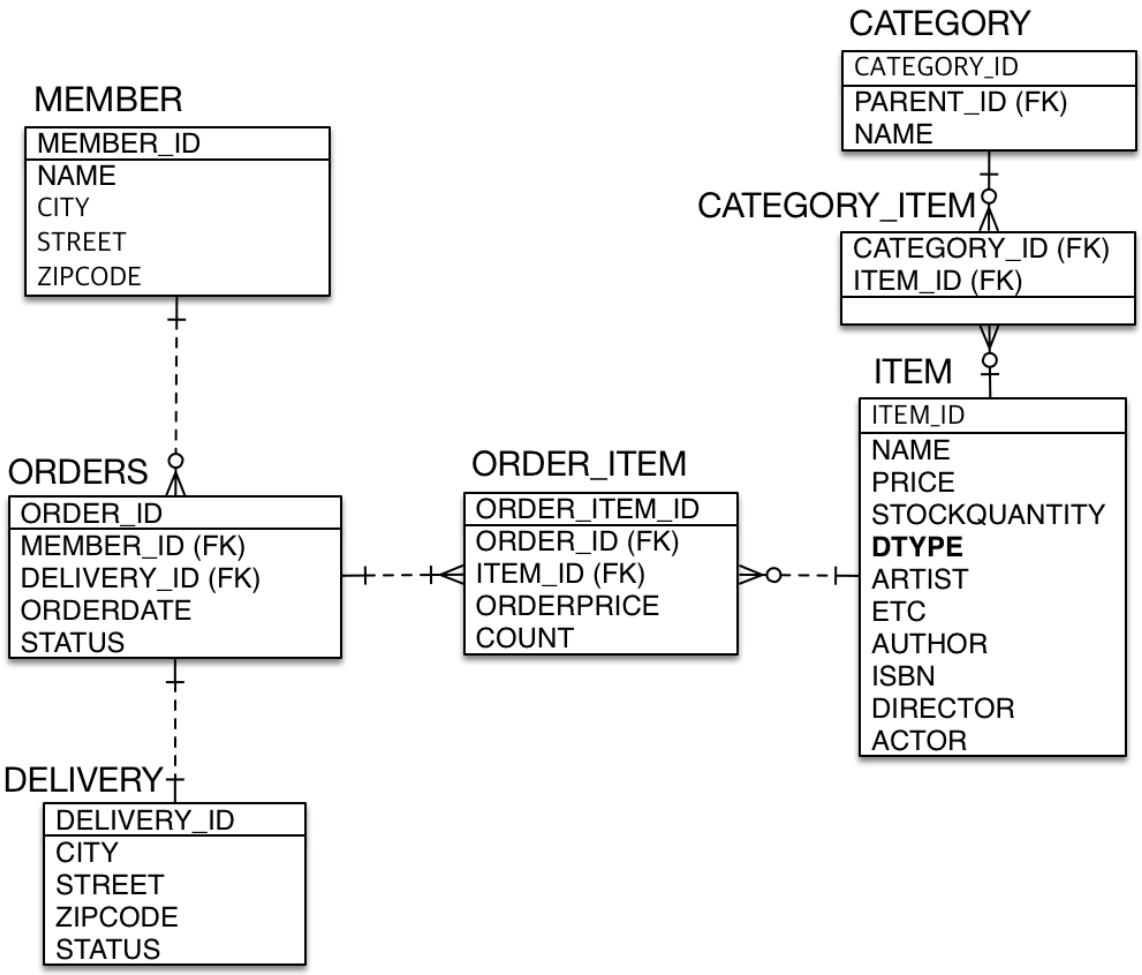

그림 - ERD4

상속 관계를 테이블 하나에 통합하는 단일 테이블 전략을 선택했다. 따라서 `ITEM` 테이블 하나만 사용하고 `DTYPE` 이라는 컬럼으로 자식 상품을 구분한다.

상속 관계 매핑하기

상품 클래스를 `jpabook.model.entity.item` 이라는 패키지로 이동했다. 이 패키지에는 상품과 상품의 자식 클래스들을 모아두었다. 그리고 상품 클래스는 직접 생성하지 않으므로 `abstract` 를 추가해서 추상 클래스로 만들었다.

==== 상품(`Item`) ====

```
package jpabook.model.entity.item;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE) //** <1>
@DiscriminatorColumn(name = "DTYPE")                  //** <2>
```

```
public abstract class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name;        //이름
    private int price;          //가격
    private int stockQuantity;  //재고수량

    @ManyToMany(mappedBy = "items")
    private List<Category> categories = new ArrayList<Category>();
    //Getter, Setter
    ...
}
```

- <1> 상속 관계를 매핑하기 위해 부모 클래스인 `Item` 에 `@Inheritance` 어노테이션을 사용하고 `strategy` 속성에 `InheritanceType.SINGLE_TABLE` 을 선택해서 단일 테이블 전략을 선택했다.
- <2> 단일 테이블 전략은 구분 컬럼을 필수로 사용해야 한다. `@DiscriminatorColumn` 어노테이션을 사용하고 `name` 속성에 `DTYPE` 이라는 구분 컬럼으로 사용할 이름을 주었다. 참고로 생략하면 `DTYPE` 이라는 이름을 기본으로 사용한다.

==== 음반(`Album`) ====

```
package jpabook.model.entity.item;

import javax.persistence.DiscriminatorValue;

@Entity
@DiscriminatorValue("A")
public class Album extends Item {

    private String artist;
    private String etc;
    ...
}
```

==== 도서(`Book`) ====

```
package jpabook.model.entity.item;

@Entity
```

```
@DiscriminatorValue("B")
public class Book extends Item {

    private String author;
    private String isbn;
    ...
}
```

==== 영화(`Movie`) ====

```
package jpabook.model.entity.item;

@Entity
@DiscriminatorValue("M")
public class Movie extends Item {

    private String director;
    private String actor;
    ...
}
```

자식 테이블들은 `@DiscriminatorValue` 어노테이션을 사용하고 그 값으로 구분 컬럼(`DTYPE`)에 입력될 값을 정하면 된다. 각각 앞자리를 따서 `A` , `B` , `M` 으로 정했다.

@MappedSuperclass 매핑하기

두 번째 요구사항을 만족하려면 모든 테이블에 등록일과 수정일 컬럼을 우선 추가해야 한다. 그리고 모든 엔티티에 등록일과 수정일을 추가하면 된다. 이때 모든 엔티티에 등록일과 수정일을 직접 추가하는 것보다는 `@MappedSuperclass` 를 사용해서 부모 클래스를 만들어 상속 받는 것이 효과적이다.

=== 기본 부모 엔티티(`BaseEntity`) ===

```
package jpabook.model.entity;

import javax.persistence.MappedSuperclass;
import java.util.Date;

@MappedSuperclass
public class BaseEntity {

    private Date createdDate;       //등록일
    private Date lastModifiedDate;  //수정일
```

```
    //Getter, Setter
    ...
}
```

=== 회원(`Member`) ===

```
@Entity
public class Member extends BaseEntity { ... }
```

=== 주문(`Order`) ===

```
@Entity
@Table(name = "ORDERS")
public class Order extends BaseEntity { ... }
```

자동 생성된 DDL을 보면 상속받은 매핑 정보가 추가되어 있다.

```
create table Member (
    MEMBER_ID bigint not null,
    createdDate timestamp,      #**
    lastModifiedDate timestamp, #**
    city varchar(255),
    name varchar(255),
    street varchar(255),
    zipcode varchar(255),
    primary key (MEMBER_ID)
)
```