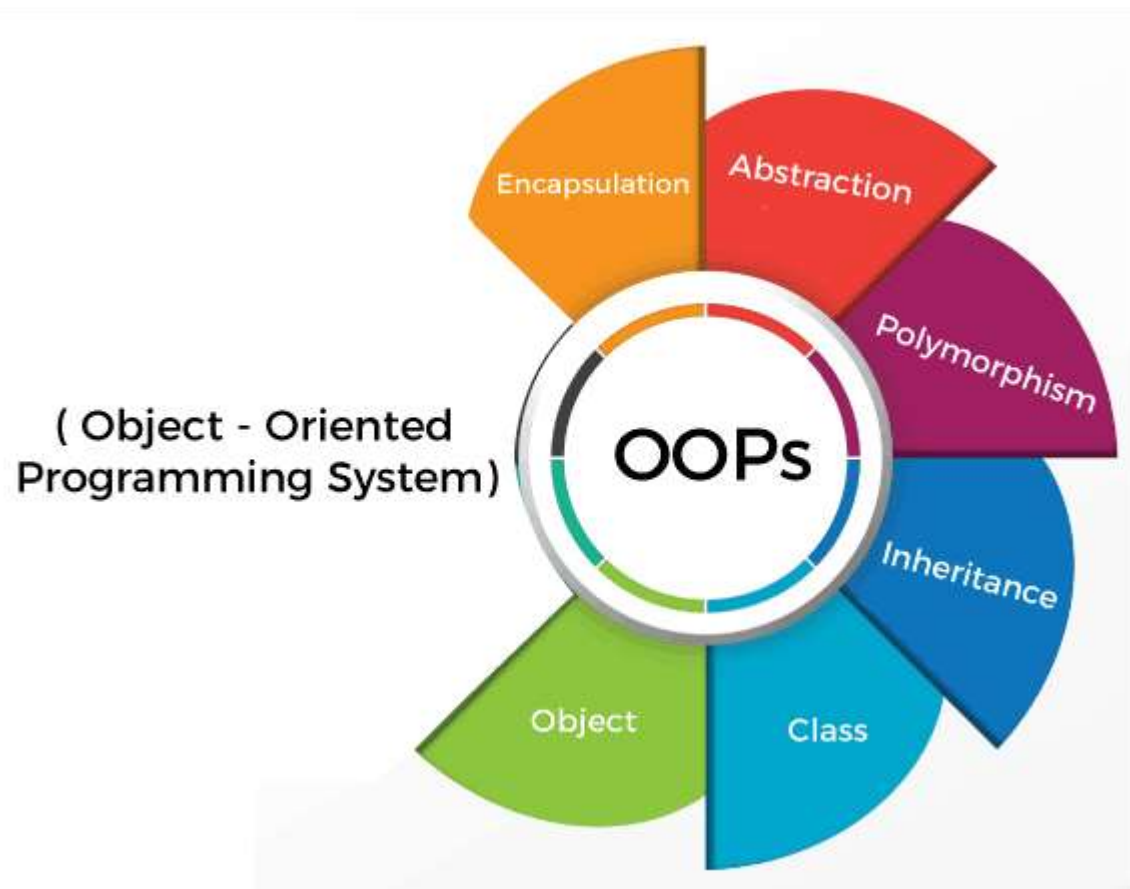


객체지향을 지향하기



객체지향 == 캡슐추다?

객체지향이 뭔데?

객체지향...? 꼭 해야돼?

객체지향? 그거 어떻게 하는건데...

객체지향이 뭔데?

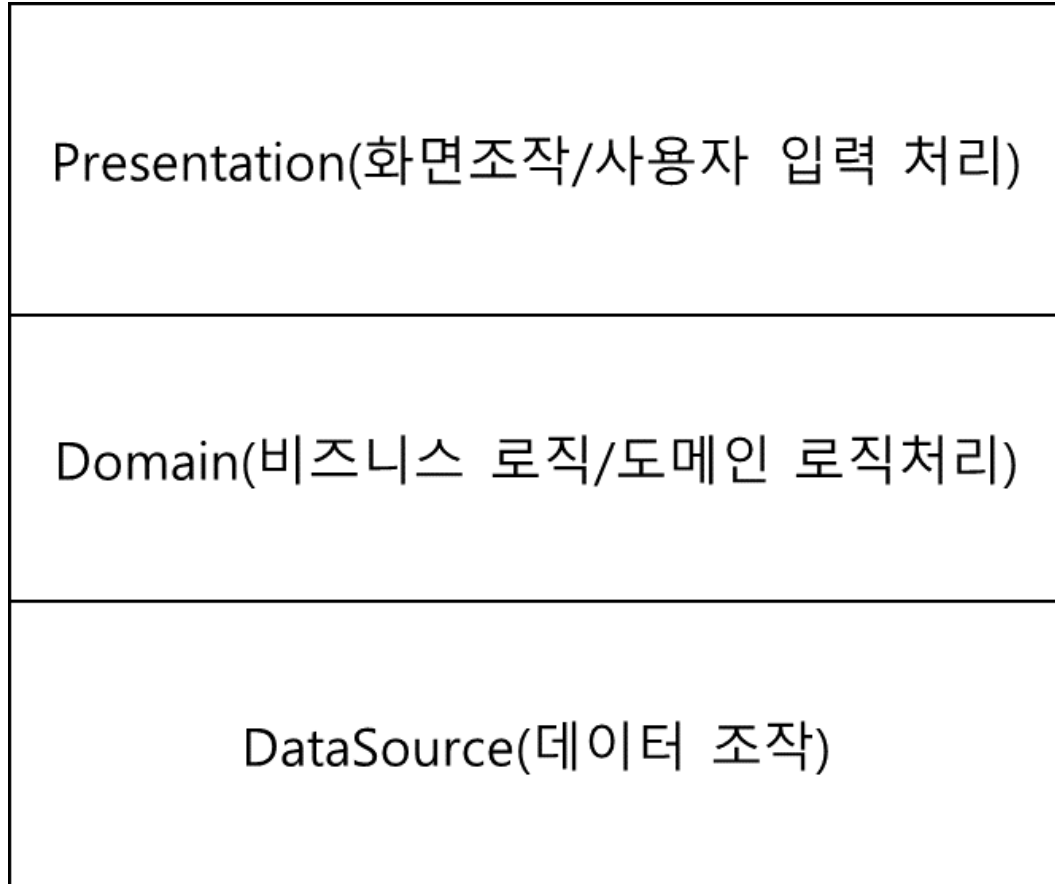
서로 다른 **역할/책임**을 가진 객체 간에
메시지를 통해 **협력관계**를 구축하는 것

객체지향? 그거 어떻게 하는건데?

코드로 알아보기위해

먼저 아키텍처를 만들어보자!

계층형 아키텍처



유사한 관심사에 관한 내용을 특정 계층에 몰아넣어

유연성/재사용성을 챙길 수 있다.

계층형 아키텍처를 설계하는 방법

절차지향으로 계층형 아키텍처를 설계하는 패턴

Transaction Script
Pattern

객체지향으로 계층형 아키텍처를 설계하는 패턴

Domain Model Pattern

예제 – “스즈메의 문단속” 영화 예매하기

영화	
영화 정보를 알고 있다. 가격을 계산한다.	할인 정책과 협력한다.

상영	
상영 정보를 알고 있다. 예매 정보를 생성한다.	영화와 협력한다. 손님과 협력한다.

할인 정책	
할인 정책을 알고있다. 할인된 가격을 계산한다.	할인 규칙과 협력한다.

할인 규칙	
할인 규칙을 알고있다. 할인 여부를 판단한다.	

예제를 **Transaction Script**로 구현해 보자!

- 데이터를 어떻게 다룰 지/활용할지를 철저하게 구분해서 사용한다.
- DB와 매핑하기 위한 엔티티 클래스와 DAO를 만들어 놓고 시작하는 것이 그 예시이다.
- 비즈니스 로직을 만든 이후, 그 로직을 처리하기 위한 DAO들을 주입 받는 방식이 일반적인 Transaction Script Pattern이다.

Transaction Script – ReservationService

```
public class ReservationService {

    @Transactional
    public Reservation reserveShowing(int customerId, int showingId, int audienceCount) {

        Showing showing = showingDAO.selectShowing(showingId);
        Movie movie = movieDAO.selectMovie(showing.getMovieId());
        List<Rule> rules = ruleDAO.selectRules(movie.getId());

        Rule rule = findRule(showing, rules);
        Money fee = movie.getFee();
        if (rule != null) {
            fee = calculateFee(movie);
        }

        Reservation result = makeReservation(customerId, showingId, audienceCount, fee);
        reservationDAO.insert(result);

        return result;
    }
}
```

```
private Rule findRule(Showing showing, List<Rule> rules) {
    for (Rule rule : rules) {
        if (rule.isTimeOfDayRule()) {
            if (showing.isDayOfWeek(rule.getDayOfWeek()) &&
                showing.isDurationBetween(rule.getStartTime(), rule.getEndTime())) {
                return rule;
            }
        } else {
            if (rule.getSequence() == showing.getSequence()) {
                return rule;
            }
        }
        return null;
    }
}

private Money calculateFee(Movie movie) {
    Discount discount = discountDAO.selectDiscount(movie.getId());

    Money discountFee = Money.ZERO;
    if (discountFee != null) {
        if (discountFee.isAmountType()) {
            discountFee = Money.wons(discount.getFee());
        } else if (discount.isPercentType()) {
            discountFee = movie.getFee().times(discount.getPercent());
        }
    }
    return movie.getFee().minus(discountFee);
}

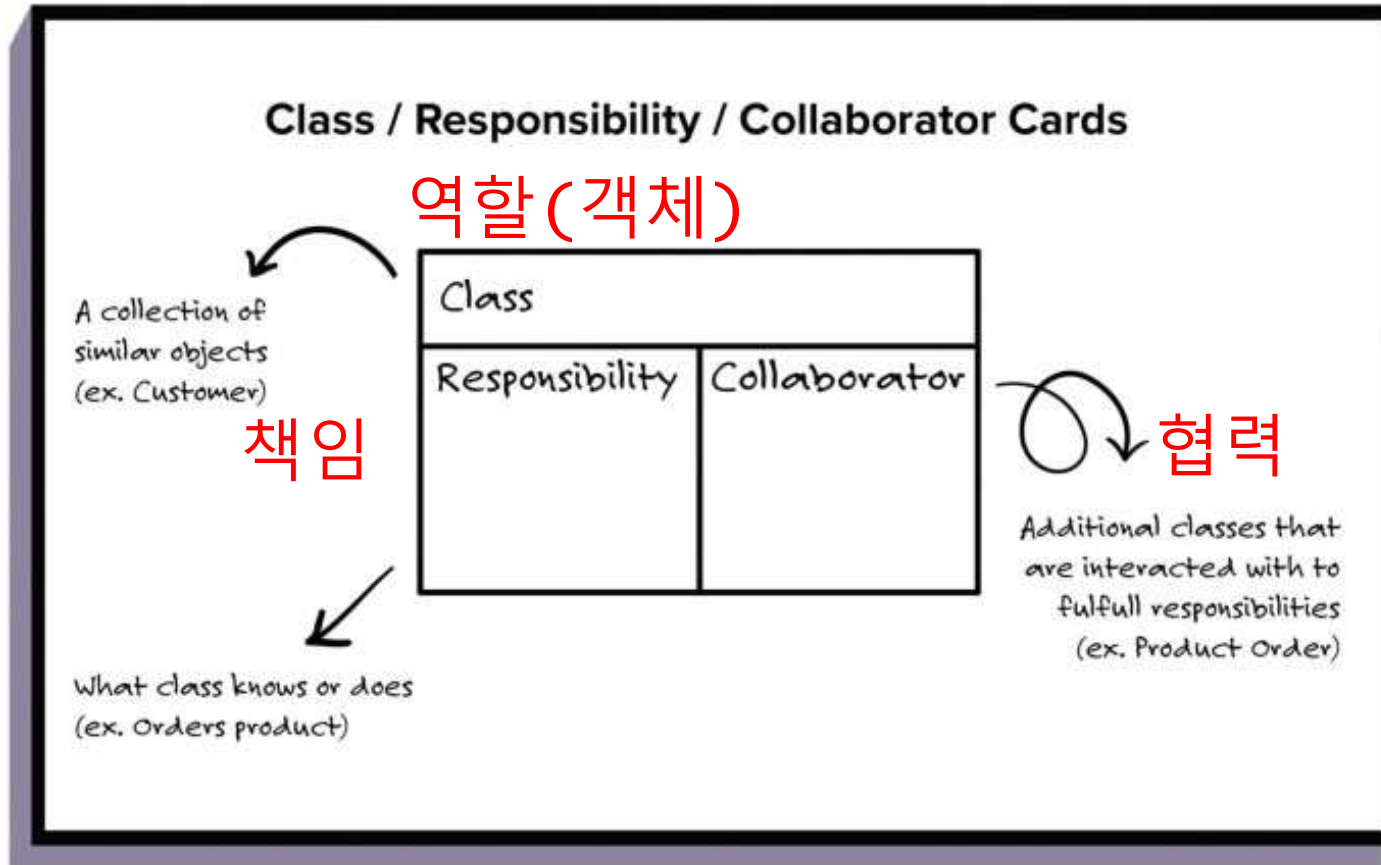
private Reservation makeReservation(
    int customerId, int showingId, int audienceCount, Money payment
) {
    Reservation result = new Reservation();
    result.setCustomerId(customerId);
    result.setShowingId(showingId);
    result.setAudienceCount(audienceCount);
    result.setFee(payment);

    return result;
}
```

Transaction Script를 Domain Model로 바꾸보자!

- CRC Card로 객체 간의 책임과 협력관계를 할당하기

CRC Card란!?



영화 예매 예제를 CRC Card로 그려보기

Shawing	
◦ 상영정보를 알고있다 ◦ 예매정보를 생성한다.	Movie Customer

Movie	
◦ 영화 정보를 알고있다. ◦ 가격을 계산한다	Discount Strategy

Discount Strategy	
◦ 할인정책을 알고있다. ◦ 할인된 가격을 계산한다.	Rule

Rule	
◦ 할인유형을 알고있다. ◦ 할인여부를 판단한다.	

Domain Model Pattern – Showing 구현

```
public class Showing {  
  
    // 예매 정보 생성은 Reservation에게 위임한다.  
    public Reservation reserve(Customer customer, int audienceCount) {  
        return new Reservation(customer, this, audienceCount);  
    }  
  
    // 금액 계산은 Movie에게 위임한다.  
    public Money calculateFee() {  
        return movie.calculateFee(this);  
    }  
}
```

Domain Model Pattern – Movie 구현

```
public class Movie {  
  
    // 예매 가격을 계산한다.  
    public Money calculateFee(Showing showing) {  
        return fee.minus(discountStrategy.caculateDiscountFee(showing));  
    }  
}
```

Domain Model Pattern – DiscountStrategy 구현

```
public abstract class DiscountStrategy {  
  
    // 할인 규칙에 해당하는지 확인하는 것을 Rule에게 위임하며  
    // 그 내용을 바탕으로 할인될 금액을 계산한다.  
    public Money calculateDiscountFee(Showing showing) {  
        for (Rule rule : rules) {  
            if (rule.isSatisfiedBy(showing)) {  
                return getDiscountFee(showing);  
            }  
        }  
        return Money.ZERO;  
    }  
}
```

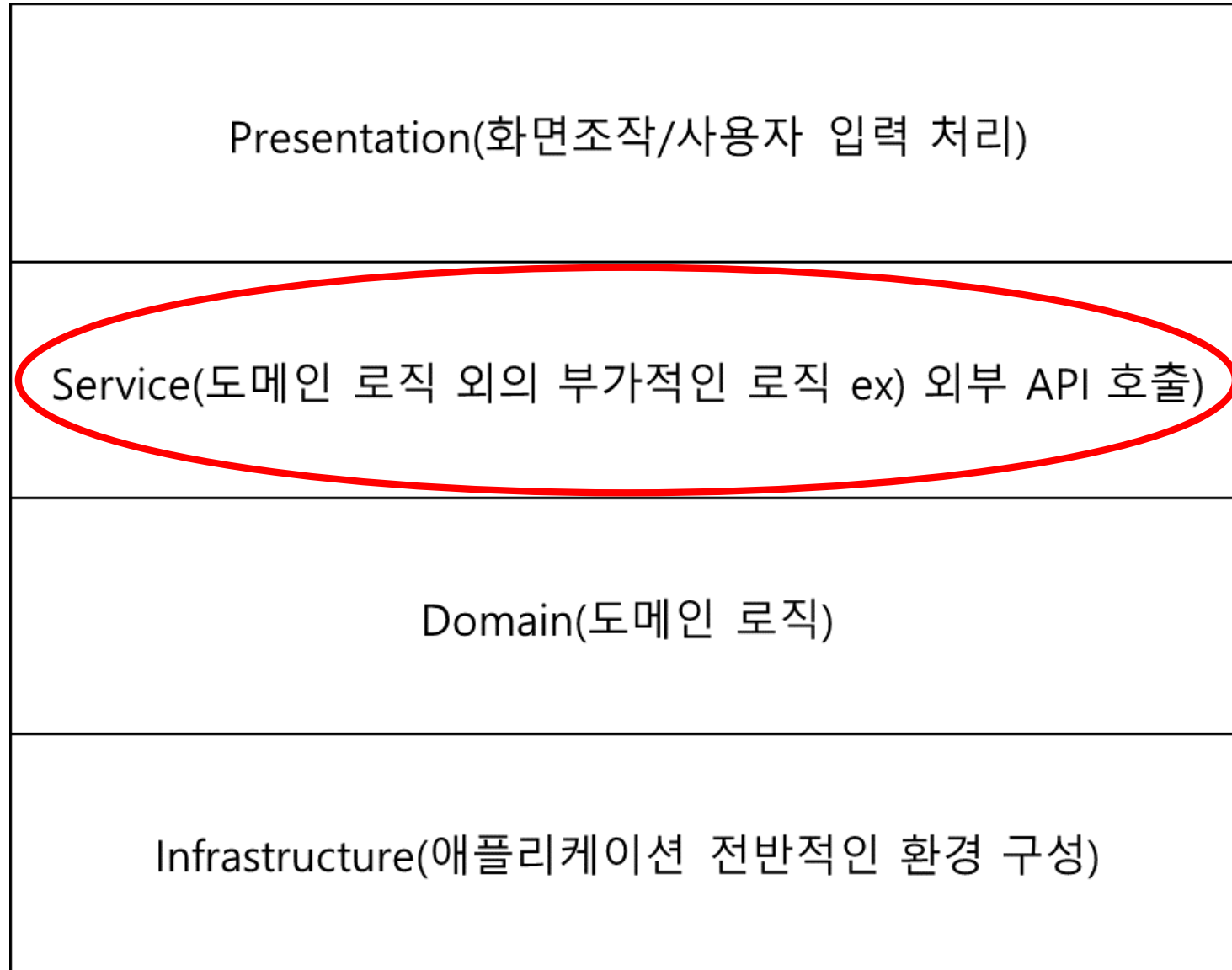
Domain Model Pattern – Rule 구현

```
public interface Rule {  
    boolean isSatisfiedBy(Showing showing);  
}
```


Domain Model Pattern – Key Point

- 객체에 대한 데이터는 객체(본인)이 직접 다루자
- 본인의 데이터를 처리할 때 외부 계층으로부터 영향 받지 말자

Domain Model Pattern Architecture



Service 계층이 Domain 계층을 보호한다.

- 도메인 로직을 처리하기 위한 준비 작업을 한다.
- 도메인 로직을 처리하는데 발생하는 후처리(예외, 반환값 셋팅 등 ...)

Domain Model Pattern – ReservationService

Service는 그저 객체끼리 협력할 수 있도록 메시지를 보낼 뿐이다.

```
public class ReservationService {  
  
    @Transactional  
    public Reservation reservationShowing(int reserveId, int showingId, int audienceCount) {  
        Customer customer = customerRepository.find(reserveId);  
        Showing showing = showingRepository.find(showingId);  
        Reservation reservation = showing.reserve(customer, audienceCount);  
  
        reservationRepository.save(reservation);  
        return reservation;  
    }  
}
```

대놓고 비교해 봅시다.

Transaction Script Pattern (Fat Service)

```
public class ReservationService {

    @Transactional
    public Reservation reserveShowing(int customerId, int showingId, int audienceCount) {

        Showing showing = showingDAO.selectShowing(showingId);
        Movie movie = movieDAO.selectMovie(showing.getMovieId());
        List<Rule> rules = ruleDAO.selectRules(movie.getId());

        Rule rule = findRule(showing, rules);
        Money fee = movie.getFee();
        if (rule != null) {
            fee = calculateFee(movie);
        }

        Reservation result = makeReservation(customerId, showingId, audienceCount, fee);
        reservationDAO.insert(result);

        return result;
    }

    private Money calculateFee(Movie movie) {
    private Rule findRule(Showing showing, List<Rule> rules) {
    private Reservation makeReservation(
        int customerId, int showingId, int audienceCount, Money payment
    ) {
        Reservation result = new Reservation();
        result.setCustomerId(customerId);
        result.setShowingId(showingId);
        result.setAudienceCount(audienceCount);
        result.setFee(payment);

        return result;
    }
}
```

Domain Model Pattern (Thin Service)

```
public class ReservationService {

    @Transactional
    public Reservation reservationShowing(int reserveId, int showingId, int audienceCount) {
        Customer customer = customerRepository.find(reserveId);
        Showing showing = showingRepository.find(showingId);
        Reservation reservation = showing.reserve(customer, audienceCount);

        reservationRepository.save(reservation);
        return reservation;
    }
}
```

대놓고 비교해 봅시다.

Transaction Script Pattern (Fat Service)

```
public class ReservationService {

    @Transactional
    public Reservation reserveShowing(int customerId, int showingId, int audienceCount) {

        Showing showing = showingDAO.selectShowing(showingId);
        Movie movie = movieDAO.selectMovie(showing.getMovieId());
        List<Rule> rules = ruleDAO.selectRules(movie.getId());

        Rule rule = findRule(showing, rules);
        Money fee = movie.getFee();
        if (rule != null) {
            fee = calculateFee(movie);
        }

        Reservation result = makeReservation(customerId, showingId, audienceCount, fee);
        reservationDAO.insert(result);

        return result;
    }

    private Money calculateFee(Movie movie) {
    private Rule findRule(Showing showing, List<Rule> rules) {
    private Reservation makeReservation(
        int customerId, int showingId, int audienceCount, Money payment
    ) {
        Reservation result = new Reservation();
        result.setCustomerId(customerId);
        result.setShowingId(showingId);
        result.setAudienceCount(audienceCount);
        result.setFee(payment);

        return result;
    }
}
```

Domain Model Pattern (Thin Service)

```
public class ReservationService {

    @Transactional
    public Reservation reservationShowing(int reserveId, int showingId, int audienceCount) {
        Customer customer = customerRepository.find(reserveId);
        Showing showing = showingRepository.find(showingId);
        Reservation reservation = showing.reserve(customer, audienceCount);

        reservationRepository.save(reservation);
        return reservation;
    }
}
```

어떤 패턴을 써야되죠?

우주에 변하지 않는 유일한 것은 '변한다'는 사실 뿐이다.

- 헤라클레이토스

변화에 면역이 더 잘 되어있는 패턴이 승자다!

변화 적용 - 영화 예매 시 “중복 할인 혜택” 추가

변화 적용 - 영화 예매 시 “중복 할인 혜택” 추가

```
public class ReservationService {

    @Transactional
    public Reservation reserveShowing(int customerId, int showingId, int audienceCount) {

        Showing showing = showingDAO.selectShowing(showingId);
        Movie movie = movieDAO.selectMovie(showing.getMovieId());
        List<Rule> rules = ruleDAO.selectRules(movie.getId());

        Rule rule = findRule(showing, rules);
        Money fee = movie.getFee();
        if (rule != null) {
            fee = calculateFee(movie);
        }

        Reservation result = makeReservation(customerId, showingId, audienceCount, fee);
        reservationDAO.insert(result);

        return result;
    }
}
```

변화 적용 - 영화 예매 시 “중복 할인 혜택” 추가

Transaction Script Pattern

```
// 기존 할인 혜택이 하나만 적용 되었을 때
private Money calculateFee(Movie movie) {
    Discount discount = discountDAO.selectDiscount(movie.getId());

    Money discountFee = Money.ZERO;
    if (discountFee != null) {
        if (discountFee.isAmountType()) {
            discountFee = Money.wons(discount.getFee());
        } else if (discount.isPercentType()) {
            discountFee = movie.getFee().times(discount.getPercent());
        }
    }
    return movie.getFee().minus(discountFee);
}
```



```
// 중복 할인 혜택이 적용 될 때
private Money calculateFee(Movie movie) {
    List<Discount> discounts = discountDAO.selectDiscounts(movie.getId());

    Money discountFee = Money.ZERO;
    for (Discount discount : discounts) {
        if (discountFee != null) {
            if (discountFee.isAmountType()) {
                discountFee = Money.wons(discount.getFee());
            } else if (discount.isPercentType()) {
                discountFee = movie.getFee().times(discount.getPercent());
            }
        }
    }
    return movie.getFee().minus(discountFee);
}
```

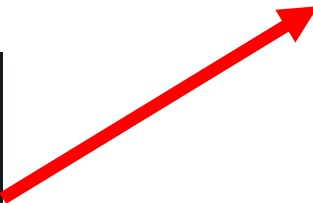
Transaction Script Pattern 에 변화를 적용 한다면...

- 기존 동작하고 있던 비즈니스 로직에 포함된 `private method`를 바꿔 끼워줘야 한다.

변화 적용 - 영화 예매 시 “중복 할인 혜택” 추가

Domain Model Pattern

```
public class Movie {  
  
    // 예매 가격을 계산한다.  
    public Money calculateFee(Showing showing) {  
        return fee.minus(discountStrategy.calculateDiscountFee(showing));  
    }  
}
```



```
public class OverlappedDiscountStrategy extends DiscountStrategy {  
    private List<DiscountStrategy> discountStrategies = new ArrayList<>();  
  
    public OverlappedDiscountStrategy(DiscountStrategy... discountStrategies) {  
        this.discountStrategies = Arrays.asList(discountStrategies);  
    }  
  
    @Override  
    protected Money getDiscountFee(Showing showing) {  
        Money result = Money.ZERO;  
  
        for (DiscountStrategy strategy : discountStrategies) {  
            result = result.plus(strategy.calculateDiscountFee(showing));  
        }  
        return result;  
    }  
}
```

Domain Model Pattern 에 변화를 적용한다면...

- 구현체를 바꿔 끼워주기만 하면 된다! 기존 코드를 수정하지 않고 DiscountStrategy를 확장한 서브 클래스를 하나 더 만든 것 뿐이다.
- 기존 코드의 변경을 담은 채 확장에는 열어놓은 OCP 원칙이 적용된 것이다.
- 이는 의존성의 방향이 추상화로 향하는 DIP의 원칙에 맞게 설계해 놓은 덕이다.
- 추상화에만 의존하고 있다면 사용하는 클라이언트 측은 구현체가 어떻게 바뀌던지 상관없다.

어떤 것을 추상화 하면 좋을까?

- 변경 가능성이 적은 것
- 일찍, 올바르게 결정하고 싶은 것

추상화를 꼭 해야하나? 어떤 장점이 있을까?

- Domain Model Pattern에서 Domain과 관련된 추상화를 뽑아내다보면 객체 간의 협력 관계를 자연스럽게 구축하게 된다.

어떻게 진짜 개발 하는데에 적용할 수 있을까?

- 테이블 주도 개발 (뒤틀린 TDD)
- 도메인 주도 개발 (DDD)

테이블 주도 개발

1. 테이블을 설계한다.
2. 객체와 테이블을 매핑한다.
3. 해당 테이블에서 수행할 수 있는 CRUD 및 기타 Query를 작성한다.
4. 비즈니스 로직을 작성한다.

테이블 주도 개발은 조금! 아쉬운 점이 있다.

- Entity 클래스를 테이블과 매핑하기 위해서만 사용하게 된다.
- 비즈니스 로직보다 DB 접근로직에 우선하여 개발을 진행하게 된다.
- 객체 간의 관계가 희미해진다.

도메인 주도 개발

1. 테이블 구조는 딱히 의식하지 않고 비즈니스 로직을 개발한다.
2. DB에 의존하지 않고 테스트 코드를 활용하여 비즈니스 로직을 작성하고 검증해도 좋다.
3. 비즈니스 로직에 확신이 생길 때 객체와 테이블을 매핑한다.

도메인 주도 개발은 뭐가 더 좋을까?

- 백엔드 엔지니어로써 가장 중요한 **비즈니스 로직**에 집중할 수 있다.
- DDL이나 객체를 매핑하는데 큰 시간을 안들여도 된다.
 - Hibernate가 제공하는 DDL 옵션을 사용하면 테스트 환경에서는 무리가 없다.
- 개발 도중 스키마가 변경되어도 기존 비즈니스 로직에 문제가 발생하지 않는다.

테이블 주도 개발 vs 도메인 주도 개발 - 개발 흐름 비교

테이블 주도 개발



데이터베이스 서버에 항상 의존관계를 가진다.

도메인 주도 개발



데이터베이스 서버 필요함

데이터베이스 서버가 없는 상태에서 개발 가능

결론 - 1

객체지향이란 객체 간의 협력관계를 구축함으로써 어떤 문제를 해결하고자 하는 것이다.

결론 - 2

객체 간의 협력관계를 구축하는 것은 관심사 분리를 통한 계층형 아키텍처를 기반으로 한다.

결론 - 3

변경 되지 않는 것은 없다.

결론 - 4

변경에 대응하기 좋은 아키텍처가 장기적으로 보면 좋을 수 있다.

결론 - 5

변경에 조금 더 좋은 설계를 기반으로 코드를 작성해볼 수 있는 것은 Domain Model Pattern이다.

결론 - 6

Domain Model Pattern은 알고 보면 도메인 주도 개발(DDD)이다.

결론

객체지향

도메인 모델 패턴(도메인 주도 개발)

ORM

계층형 아키텍처

협력 관계 설계하
기

결론 - 인용

정답은 없습니다.

일단은 본인이 속한 조직에 맞춰가시고, 조직 내에서 어느정도 힘이 생겼을 때
본인이 생각하기에 더 좋다고 생각하는 방식을 도입하면 됩니다.

- 조영호, 박재성