Object 1장

객체, 설계 (티켓 판매 애플리케이션 구현하기)

> *소극장을 경영하는데 이벤트를 기획하였다. 이벤트 내용은 간단한데 추첨을 통해 선정된 관람객에게 공연을 무료로 관람할 수 있는 초대장을 발송하는 것이다.*
>
> *한가지 염두에 둬야 할 점은 이벤트에 당첨된 관람객과 그렇지 못한 관람객은 다른방식으로 입장 시켜야 한다는 것이다. 이벤트에 당첨된 관람객은 초대장을 티켓으로 교환한 후에 입장 할 수 있다. 이벤트에 당첨되지 않은 관람객은 티켓을 구매해야만 입장할 수 있다. 따라서 관람객을 입장시키기 전에 이벤트 당첨 여부를 확인해야 하고 이벤트 당첨자가 아닌 경우에는 티켓을 판매한 후에 입장 시켜야한다.*

먼저 이벤트 당첨자에게 발송되는 초대장을 구현하는 것으로 시작하자. 초대장이라는 개념을 구현한 Invitation은 공연을 관람할 수 있는 초대일자를 인스턴스 변수로 포함하는 간단한 클래스다.

```
public class Invitation {
    private LocalDateTime when; //초대일자
}
```

공연을 관람하기 원하는 모든 사람들을 티켓을 소지하고 있어야만 한다. Ticket 클래스를 추가하자.

```
public class Ticket {
    private Long fee; //요금

    public Long getFee() {
        return fee;
    }
}
```

이벤트 당첨자는 티켓으로 교환할 초대장을 가지고 있을 것이다. 당첨되지 않은 관람객은 티켓을 구매할 수 있는 현금을 보유하고 있을 것이다. 따라서 관람객이 가지고 올 수 있는 소지품은 초대장, 현금, 티켓 세가지 뿐이다.

관람객은 소지품을 보관할 용도로 가방을 들고 올 수 있다고 가정하자. Bag 클래스를 추가해보자

```
public class Bag {

    private Long amount;
```

```
    private Invitation invitation;
    private Ticket ticket;

    public boolean hasInvitation() {
        return invitation != null;
    }

    public void setTicket(Ticket ticket) {
        this.ticket = ticket;
    }

    public void minusAmount(Long amount) {
        this.amount -= amount;
    }

    public void plusAmount(Long amount) {
        this.amount += amount;
    }
}
```

입장하고자 하는 관객을 담는 클래스로 각각의 관객은 가방을 가지고 있다.

```
    public class Audience {
        private Bag bag;

        public Audience(Bag bag) {
            this.bag = bag;
        }

        public Bag getBag() {
            return bag;
        }
    }
```

티켓오피스에서는 티켓을 가지고 있고 티켓이 판매되면 티켓을 줄이고 가격을 높이는 등의 기능을 제공한다.

```
    import java.util.ArrayList;
    import java.util.Arrays;
    import java.util.List;

    public class TicketOffice {
        private Long amount;
        private List<Ticket> tickets = new ArrayList<>();

        public TicketOffice(Long amount, Ticket ... tickets) {
            this.amount = amount;
            this.tickets.addAll(Arrays.asList(tickets));
        }
```

```
    public Ticket getTicket() {
        return tickets.remove(0);
    }

    public void minusAmount(Long amount) {
        this.amount -= amount;
    }

    public void plusAmount(Long amount) {
        this.amount += amount;
    }

}
```

판매원은 자신이 소속된 티켓오피스를 가지고 있다.

```
package kail.study.java.objectsbook.ticket.domain;

public class TicketSeller {
    private TicketOffice ticketOffice; // 티켓 오피스를 가지고 있다.

    public TicketSeller(TicketOffice ticketOffice) {
        this.ticketOffice = ticketOffice;
    }

    public TicketOffice getTicketOffice() {
        return ticketOffice;
    }
}
```

프로그램을 담당하는 극장은 판매원이 있으며 각각의 도메인에서 데이터를 가져와 입장과 관련된 일을 처리한다.

```
public class Theater {
    private TicketSeller ticketSeller;

    public Theater(TicketSeller ticketSeller) {
        this.ticketSeller = ticketSeller;
    }

    public void enter(Audience audience) {
        if(audience.getBag().hasInvitation()) { //초대장을 가지고 있으면
            Ticket ticket = ticketSeller.getTicketOffice().getTicket();
            audience.getBag().setTicket(ticket);
        } else { //없으면 표를 산다
            Ticket ticket = ticketSeller.getTicketOffice().getTicket();
            audience.getBag().minusAmount(ticket.getFee());
            ticketSeller.getTicketOffice().plusAmount(ticket.getFee());
            audience.getBag().setTicket(ticket);
        }
```

```
        }
    }
```

문제점

위의 프로그래밍에서는 문제점을 가지고 있다. 로버트 마틴의 말에 의하면 소프트웨어 모듈은 3가지 특성을 가져야한다고 말한다.

- 정상적으로 동작해야 한다.
- 변경이 용이해야 한다.
- 이해하기 쉬워야 한다.

위 프로그램은 정상적으로 동작하는 티켓판매 과정임으로 1번 기준은 만족한다. 하지만 2번과 3번을 만족하지 못하고 있는데 그 이유에 대해서 생각해보자.

- 이해하기 어려운 코드
    - 위의 코드에서 `Theater` 클래스의 `enter` 메서드를 보자. 소극장은 관람객의 가방을 열어 그 안에 초대장이 들어 있는지 확인한다. 가방 안에 초대장이 있으면 판매원은 매표소에 보관돼 있는 티켓을 관람객의 가방 안으로 옮긴다. 가방 안에 초대장이 들어있지 않다면 관람객의 가방에서 티켓 금액만큼의 현금을 꺼내 매표소에 적립한 후에 매표소에 보관돼 있는 티켓을 관람객 가방 안으로 옮긴다.
    - 직관적으로 생각해봤을 때 우리는 표를 구매할 때 우리 스스로 가방을 열어 초대장을 확인하고 초대장이 없으면 우리가 직접 돈을 지불하고 계산하여 티켓을 받고 가방에 넣는다. 하지만 위의 예제에서는 우리는 단순히 수동적인 존재이며 극장 이라는 객체가 모든 일을 처리한다. 뿐만 아니라 매표소 직원도 직원의 허락없이 극장 자체가 티켓을 가져가고 돈을 적립하고 있다. 직관적으로 이해가 되는 가? **직관적으로 이해하기 어려운 코드는 이해하기 어려운 코드이다**
- 변경에 취약한 코드
    - 이해하기 어려운 코드보다 더 큰 문제는 변경에 취약한 코드라는 것이다. 앞에서 언급한 바와 같이 각각의 도메인은 데이터 로서의 역할만을 담당하고 있고 극장 이라는 클래스가 모든 일을 처리하고 있다. 이는 다시 말해 데이터에서 변화가 발생하는 경우 극장 에서의 로직은 언제나 변경되어야 하는 대상이 되며 극장 은 너무나 많은 도메인과 의존성을 맺고 있다고 볼 수 있다.
    - 객체지향에서 의존성은 너무나 당연한 것이다. 객체는 각자의 자율적인 방법으로 책임을 수행하지만 협력을 통해 전체와 소통하기 때문에 서로간의 의존 및 협력은 당연한 것이다. 하지만 위의 예제와 같이 과한 의존성이 나타나는 경우 유지보수의 어려움은 급증한다. 변경에 취약하지 않은 설계는 **요구사항의 변경이나수정사항에 대해 담당하고 있는 클래스 하나 혹은 최소한만이 변화되면 되는 설계이다.**

해결책 : 자율적인 객체

위의 코드가 문제시되고 있는 **변경에 취약한 구조, 이해하기 어려운 코드**를 변경해보자. 어떠한 방향으로 해결할 수 있을까? 우리가 문제시 되고 있는 부분을 더 간결하게 요약하면 아래와 같다. 즉 각각의 객체가 **자율적인 존재** 로서 본인의 역할과 책임을 수행하고 있지 않기 때문에 외부 클래스에서 이를 관리하게 되고 그렇기 때문에 직관적으로 이해하기 어렵다 라고 할 수 있다. **그럼 해결책은 당연히 각각의 객체에게 자율성을 부여하는 것이다.**

- 역할과 책임을 담당해야하는 객체가 수동적이라 직관적으로 이해하기 어렵다.
- 하나의 클래스가 너무 많은 곳에 의존함으로써 결합도가 높아진다.

티켓을 판매하는 기능은 극장 의 역할이 아니다. 판매원 이 해야하는 역할이다. 이를 위해 기존에 극장 에서 담당하는 enter 의 내부 메서드를 판매원 이 담당할 수 있도록 변경하였다.

```
public class Theater {
    private TicketSeller ticketSeller;

    public Theater(TicketSeller ticketSeller) {
        this.ticketSeller = ticketSeller;
    }

    public void enter(Audience audience) {
        ticketSeller.toSell(audience); // 캡슐화
    }
}
public class TicketSeller {
    private TicketOffice ticketOffice;

    public TicketSeller(TicketOffice ticketOffice) {
        this.ticketOffice = ticketOffice;
    }

    public void toSell(Audience audience) {
        if(audience.getBag().hasInvitation()) {
            Ticket ticket = ticketOffice.getTicket();
            audience.getBag().setTicket(ticket);
        } else {
            Ticket ticket = ticketOffice.getTicket();
            audience.getBag().minusAmount(ticket.getFee());
            ticketOffice.plusAmount(ticket.getFee());
            audience.getBag().setTicket(ticket);
        }
    }
}
```

기존의 문제점에서 극장 이 판매원과 무관하게 티켓을 가져가고 요금을 적립하는 행위들이 있었기에 이해하기 어려운 코드가 발생하였고 이러한 행위는 객체간의 결합도를 높이고 있었기에 문제가 되었다. 하지만 위와 같이 판매원 이라는 객체가 본인의 역할을 수행하도록 변경하면 기존에 있던 아래의 메소드가 사라지며 외부에서 판매 와 관련된 활동이 불가능해진다. 이를 조금 어려운 말로 변경하면 객체 내부의 세부적인 사항을 캡슐화 하여 외부 객체에게 내부 접근을 불가능하게 하면 객체 사이의 결합도를 낮출 수 있기 때문에 변경에 유연해진다.

극장 은 더이상 스스로 판매원의 일을 할 수 없고 판매원에게 sellTo 라는 메세지를 통해서만 판매를 할 수 있다. 이와 같은 맥락에서 다른 부분도 고칠점을 찾아보자. 아까 언급했던 것처럼 **관객이 직접 가방을 꺼내지 않고 극장이 이를 꺼내는 것이 문제가 되었다.** 이 부분도 같이 고쳐보자.

```
public class TicketSeller {
    private TicketOffice ticketOffice;

    public TicketSeller(TicketOffice ticketOffice) {
        this.ticketOffice = ticketOffice;
    }

    public void toSell(Audience audience) {
```

```
            ticketOffice.plusAmount(audience.buy(ticketOffice.getTicket()));
        } // 판매원은 관객의 티켓 금액을 티켓오피스에 더해준다.
    }
    public class Audience { // 관객은 스스로 티켓 여부를 판단하고 티켓의 금액을 지불한다.
        private Bag bag;

        public Audience(Bag bag) {
            this.bag = bag;
        }

        public Long buy(Ticket ticket) {
            if(bag.hasInvitation()) { //초대장 여부확인
                bag.setTicket(ticket);
                return 0L;
            }else {
                bag.setTicket(ticket); // 스스로 티켓 금액 지불
                bag.minusAmount(ticket.getFee());
                return ticket.getFee();
            }
        }
    }
```

기존에 극장 에서 가방을 꺼내어 돈을 가져가고 했던 부분이 리팩토링을 통해 판매원 이 하도록 변경하였다. 하지만 여전히 관객은 수동적인 존재로 스스로 가방에서 돈을 꺼내지 않고 판매원에 의존하여 티켓을 구매하고 있다. 하지만 현실세계에서는 그렇지 않다. 현실세계와 대응되도록 위와 같이 변경하였다.

이를 통해 관객은 스스로 티켓의 여부를 판단하고 판매원이 제공하는 티켓을 본인의 가방에 넣고 발생한 금액을 지불하는 형태를 가진다. 구매하는 로직이 변경되더라도 관객클래스에서만 로직을 변경하면 다른 클래스는 영향을 받지 않는다. 즉 의존성이 낮아지고 결합도가 낮아졌다고 볼 수 있다.

> *즉 지금까지의 과정은 극장에서 모든 것을 처리하던 프로세스에서 극장, 판매원, 관객이 각각을 자율적인 존재로 변경함으로써 의존하고 있는 대상을 줄이고 이해 가능한 코드로 만듦으로서 변경에 유연하게 대처할 수 있도록 한 것이다.*

추가해결

리팩토링한 위의 코드에서도 여전히 같은 맥락의 문제가 존재한다. 그 문제는 바로 가방 이다. 현실세계에서 가방은 인간에 의해 열려지고 안에 있는 물건들이 관리된다. 하지만 객체지향의 세계에서는 각각의 객체는 자율성을 가지고 가방이나 무생물, 개념이라 할지라도 각각 자율성을 지닌 **독립적인 존재**가 된다. 이와 같은 논리로 인해 객체지향은 **현실세계를 모방하는 것이 아닌 재창조하는 과정** 이라고 설명되며 각각의 존재들은 **의인화 및 은유** 를 통해 표현된다.

그렇다면 위의 예제에서 가방과 TicketOfiice는 여전히 수동적인 형태로 본인의 자율성 없이 외부의 객체들에게 의존하고 있다. 이러한 부분을 개선해보자.

가방이 `hold` 라는 메서드를 통해 직접 초대장을 확인하고 인자로 입력받은 초대장을 가방에 입력하는 형태로 변경함으로써 가방에게 자율권을 부여한다.

```
public class Bag {
 private Long amount;
 private Invitation invitation;
 private Ticket ticket;

 public Bag(Long amount) {
     this(null,amount);
 }

 public Bag(Invitation invitation, long amount) {
     this.amount = amount;
     this.invitation = invitation;
 }

 public Long hold(Ticket ticket) { // 스스로 초대장을 확인
     if(hasInvitation()) {
         setTicket(ticket);
         return 0L;
     } else { // 없으면 티켓을 스스로 구매
         setTicket(ticket);
         minusAmount(ticket.getFee());
         return ticket.getFee();
     }
 }

 private boolean hasInvitation() {
     return invitation != null;
 }

 public boolean hasTicket() {
     return ticket != null;
 }

 private void setTicket(Ticket ticket) {
     this.ticket = ticket;
 }

 private void minusAmount(Long amount) {
     this.amount -= amount;
 }

 public void plusAmount(Long amount) {
     this.amount += amount;
 }
}
```

티켓오피스도 외부에서 Get을 통해 사용하지 않고 본인의 로직을 본인이 처리하도록 변경하였다.

```
 import java.util.ArrayList;
 import java.util.Arrays;
```

```
    import java.util.List;

    public class TicketOffice {
        private Long amount;
        private List<Ticket> tickets = new ArrayList<>();

        public TicketOffice(Long amount, Ticket ... tickets) {
            this.amount = amount;
            this.tickets.addAll(Arrays.asList(tickets));
        }

        //자율적인 티켓 판매 외부에서 가져오지 않고 객체 안에서 처리
        public void sellTicketTo(Audience audience) {
            plusAmount(audience.buy(getTicket()));
        }

        private Ticket getTicket() {
            return tickets.remove(0);
        }

        public void minusAmount(Long amount) {
            this.amount -= amount;
        }

        private void plusAmount(Long amount) {
            this.amount += amount;
        }

    }
```

결과적으로 외부에서 그 값을 가져와 직접 연산하던 클래스들은 담당 객체에게 메세지를 보내는 형태로 변형되었고

요구사항이 변경되었을 때 구현체만 변경해주면 되는 형태가 되었다.

```
public class Audience {

    private Bag bag;

    public Audience(Bag bag) {
        this.bag = bag;
    }

    public Long buy(Ticket ticket) {
        return bag.hold(ticket); //자율적인 가방
    }
   }


 public class TicketSeller {

        private TicketOffice ticketOffice;

        public TicketSeller(TicketOffice ticketOffice) {
```

```
            this.ticketOffice = ticketOffice;
        }

        public void toSell(Audience audience) {
            ticketOffice.sellTicketTo(audience); //자율적인 티켓오피스
        }
    }
```