객체지향개발론및실습

Laboratory 9. State 패턴

1. 목적

- 객체의 내부 상태가 변경되었을 때 행동 패턴을 바꿀 수 있도록 해주는 상태 패턴을 실습한다.
- 일반적으로 객체지향에서 객체는 상태를 가지며, 이 상태에 따라 행위의 결과가 달라질 수 있다. 이 때문에 모든 종류의 객체를 상태 패턴을 사용하여 설계하면 효과적이라고 생각할 수 있다. 하지만 상태 패턴은 행위의 결과가 달라지는 것을 효과적으로 모델링 해주는 것이 아니라 상태에 따라 해당 행위를 할 수 있는지 여부가 달라지는 경우에 효과적이다. 특히, 상태가 구체적(상태 전이도가 제시 가능함)이며, 다수의 행위가 상태에 영향을 받을 때 효과적이다.
- 상태 패턴을 사용하면 각 상태를 나타내는 별도 클래스를 정의하며, 객체는 상태 변경에 따라 유지하는 상태 객체를 변경한다.
- 상태 패턴은 크게 상태 객체에서 상태를 전이하는 상태 중심 전이 방식과 문맥 객체에서 상태를 전이하는 문맥 중심 전이 방식으로 구현이 가능하다.
- 보통 상태 패턴의 사용을 통해 문맥 클래스를 조건문 없이 간결하게 작성하고 해당 상태를 수행해야 하는 것을 상태 객체에 위임하는 것이 의도이기 때문에 상태 중심 전이 방식이 이 의도에 충실한 구현 방법이다.
- 문맥 객체를 다수 생성하여 사용할 경우에는 상태 객체를 너무 많이 생성할 수 있기 때문에 상태 객체를 공유하는 것이 효과적일 수 있다. 상태 중심 전이 방식에서는 상태 객체의 전이 메소드를 호출할 때 문맥 객체를 전달하는 방법을 사용할 수 있다.
- 자바는 상태 객체를 열거형 상수로 구현하면 자동으로 공유하는 형태가 되며, 한 눈에 전체 상태 관련 정보를 볼 수 있기 때문에 여러 가지 이점이 있다.

2. 개요

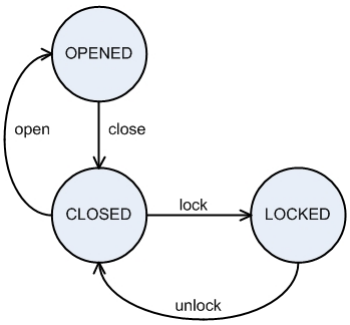

그림 9.1: 문 상태도

- 그림 9.1에 주어진 일반 문에 대한 상태도를 바탕으로 기존 방법과 상태 패턴을 이용한 방법으로 구현해봄으로써 그 차이를 경험해본다.

3. 기존 방법

- 상태 패턴을 모르는 경우 상태에 따라 다른 행동을 하도록 프로그래밍하는 일반적인 방법은 아래와 같이 상태를 나타내는 열거형을 사용하는 것이다.
- 각 언어별 소스는 el에서 다운받을 수 있다.
- 문 예제는 상태에서 특정 행위를 하였을 때 상태 변경을 하는 것 외에 할 일이 없어 상태 패턴을 활용하여 코드를 바꾸더라도 그것의 이점을 잘 느끼지 못할 수 있다.

```
public class Door {
    public enum State {OPENED, CLOSED, LOCKED}
    private State currentState = State.CLOSED;
    public void open(){
        switch(currentState){
        case OPENED:
            System.out.println("이미 열려 있음");
            break;
        case CLOSED:
            System.out.println("문이 열림");
            currentState = State.OPENED;
            break;
        case LOCKED:
            System.out.println("잠금을 해제해야 열 수 있음");
        } // switch
    }
    public void close(){
    }
    public void lock(){
    }
    public void unlock(){
    }
}
```

4. 상태 중심 전이 방식 실습

- 상태 객체에서 상태를 전이하는 상태 중심 전이 방식의 상태 패턴을 이용하여 **Door**를 구현한 자바 버전이 el에 제시되어 있다.
  - 자바. 이 버전을 열거형을 사용하도록 변경한다. 열거형이므로 상태 객체를 공유하는 형태이며, 문맥은 **changeState(DoorState state)** 형태의 상태 전이를 위한 메소드를 하나만 정의하면 된다.
  - 파이썬, C++: 상태 중심 전이 방식의 상태 패턴을 이용하여 구현한다. 싱글톤으로 구현하지 않고, 상태 객체를 클래스 멤버로 만들어 공유하는 형태로 구현한다. 문맥은 상태만큼 **changeToState()** 형태의 상태 전이를 위한 메소드를 제공해야 한다.
    - 파이썬도 열거형을 이용하여 구현할 수 있지만 각 열거형 상수에 특정 기능을 추가할 수 없어, 열거형에 추가하는 전이 메소드에서 조건문의 사용이 필요하다. 이 때문에 파이썬의 경우 열거형에 전이 메소드를 구현하는 방식은 좋은 구현 방식이 아니다.
- 상태 패턴을 이용하여 클래스를 정의하기 위해서는 상태 전이가 일어날 수 있는 문맥의 행위들을 **interface**로 정의해야 한다. 그다음 이 **interface**를 구현한 각 상태 클래스를 정의해야 한다. 문 예제에서 상태 클래스가 구현해야 하는 **interface**는 다음과 같다.

```
public interface DoorState {
    void open();
    void close();
    void lock();
    void unlock();
}
```

C++와 파이썬은 추상 클래스로 **DoorState**를 정의하면 된다.

- 일반적인 상태 중심 전이 방식에서는 상태 객체는 문맥 객체를 멤버 변수로 유지한다. 하지만 이렇게 구현하면 상태 객체를 공유할 수 없다. 따라서 상태 객체를 공유하고 싶으면 **DoorState**의 각 메소드는 다음과 같이 변경해야 한다.

```
1 void open(Door door);
```

- 열거형 버전은 기본적으로 상태 객체를 공유하겠다는 것이므로 이 형태로 메소드를 이용해야 한다.
- 파이썬은 싱글톤으로 각 상태 객체를 구현할 수 있지만 상태 객체를 클래스 멤버로 정의하여 공유할 수 있다. 파이썬은 이 방식으로 구현한다.
- 자바 8 이후에는 메소드 선언 대신에 빈 메소드 형태의 기본 메소드로 정의하면 상태를 구현하는 클래스를 더 편리하게 작성할 수 있다.
- C++와 파이썬도 **DoorState**에 빈 메소드로 각종 전이 메소드를 정의하면 각 상태 클래스는 그 상태에서 전이가 발생하는 메소드에 대해서만 구현하면 된다.
- 상태 전이 기반 구현 방식에서는 상태 객체에서 문맥의 상태를 전이할 수 있어야 한다. 상태가 총 n개가 있다고 가정하면 문맥 객체는 이를 위한 **getOpenedState()**와 같은 n개의 getter와 **changeState(DoorState state)**와 같은 setter를 하나 제공하거나 상태 변경을 위한 **changeToOpenedState()**와 같은 메소드를 n개 제공해야 한다. 열거형 버전에서는 상태 객체를 공유하는 형태이기 때문에 **changeState(DoorState state)** 하나만 정의하면 된다.
- C++와 파이썬의 경우 싱글톤으로 만들면 자바의 열거형 버전처럼 하나의 메소드만 정의하면 되지만 클래스 멤버로 공유하는 방식에서는 **changeToOpenedState()**와 같은 메소드를 n개 제공해야 한다.
- 이 실습의 각종 상태 클래스를 구현할 때 불필요한 콘솔 출력은 포함하지 않아도 된다.

5. 문맥 중심 전이 방식 실습

- 문맥 중심 전이 방식을 사용하는 자바 버전도 el에 제시되어 있다.
  - 자바: 이 버전을 열거형을 사용하도록 변경한다. 이 버전은 상태 객체를 반환하는 형태로 구현하며, 상태 전이가 고정되어 있으므로 문맥 객체를 전이 메소드의 인자로 전달하지 않는다.
  - C++와 파이썬: 문맥 중심 전이 방식의 상태 패턴을 이용하여 구현한다. 상태 전이가 고정되어 있으므로 문맥 객체를 전이 메소드의 인자로 전달하지 않으며, 상태 클래스를 **싱글톤**으로 정의하고, 상태 객체를 반환하는 형태로 구현해야 한다.
- 문맥 중심 전이 방식은 문맥에서 전이가 이루어지지만 전이를 해야 하는 여부는 상태 객체가 알려주어야 한다. 이를 위해 전이 메소드가 **boolean** 값을 반환하도록 만들 수 있고, 상태 객체 자체를 반환하도록 만들 수 있다.
- 상태 객체를 반환하는 방식의 경우 상태 전이가 고정되어 있지 않으면 문맥 객체에 대한 접근 없이 적절한 상태 객체를 반환하기 어렵다.
- 지금 모델링하고 있는 문은 상태 전이가 고정적이기 때문에 상태 객체를 반환하는 방법으로 문맥 객체에 대한 접근 없이 구현할 수 있다.
- 열거형으로 구현하거나 싱글톤으로 구현하지 않으면 상태 객체가 문맥 객체에 대한 접근이 필요하며, 문맥 객체가 유지하는 상태 객체를 반환하는 getter 메소드를 제공해 주어야 상태 객체를 반환하는 방법으로 구현할 수 있다.
- **boolean** 값을 반환하는 방법에서 문맥 객체는 반환값에 따라 상태를 스스로 변경해야 하므로 조건문을 제거하기 위해 상태 패턴을 도입하였는데, 계속 조건문을 사용해야 하는 문제점이 있다.
- 하지만 상태 객체를 반환하는 방법으로 구현하면 이 문제를 부분적으로 해결할 수 있다.

6. 과제 음료수 캔 자판기

- 이론 수업에서 살펴본 껌볼기기는 아주 간단한 형태의 자동판매기이다.
- 자동판매기는 상태 패턴으로 구현하기 매우 적합한 형태이다.
- 일반적인 자동판매기는 껌볼기기와 달리 판매하는 제품이 여러 종류가 있고, 최신 자동판매기는 다양한 지불 방법을 제공하고 있다,
- 과제로 음료수 캔을 판매하는 자동판매기를 구현해 본다.
- JavaFX, PyQt6, Qt6를 이용한 기본 프로젝트를 제공한다. 제공된 프로젝트가 올바르게 동작하도록 상태 패턴을 사용하여 완성해야 하다. **`VendingMachine`** 클래스의 **`insertCash`**, **`selectItem`**, **`cancel`**을 완성해야 한다. 또 **`clearItems`**, **`setItems`**, **`removeItem`**에도 상태가 변할 수 있는 메소드이므로 상태 변화와 관련된 코드의 추가가 필요하다.
- 모델링하는 음료수 캔 자동판매기의 특성
  - 지불 방법은 현금을 투입하는 방법만 고려한다.
  - 현금은 10원, 50원, 100원, 500짜리 동전과 1,000원, 5,000원, 10,000원짜리 지폐를 사용할 수 있다.
  - 재고가 있어야 구매가 가능하다.
  - 재고가 있고, 현금을 투입하더라도 정확한 거스름을 지급할 수 있어야만 구매가 가능하다. 예를 들어 800원 짜리 음료 캔 하나를 구입하고자 할때, 200원의 거스름을 줄 동전이 없으면 1,000원짜리 지폐 하나를 이용해서는 해당 캔을 구입할 수 없다.
- 상태 중심 전이 방식으로 구현하는 것이 편리하다.
- 구현하기 전에 상태 전이도를 먼저 그리고, 그것을 바탕으로 구현해야 한다.
- 사용자와 상호작용하는 GUI가 있기 때문에 구매할 수 없는 상품을 선택하는 등과 같은 행위는 발생할 수 없다.
- 제시된 프로그램을 검토해 보면 자동판매기와 고객 또는 관리자가 상호작용할 수 있는 부분은 다음과 같음
  - **`insertCash`**: 고객이 동전 또는 지폐를 투입하는 행위
  - **`selectItem`**: 고객이 특정 제품을 선택
  - **`cancel`**: 고객이 취소 버튼을 선택 (투입하였지만 사용하지 않은 금액은 반환되어야 함)
  - **`dispenseItem`**: 선택한 제품을 제공함
  - **`setCash`**: 관리자가 돈을 보충함
  - **`setItems`**: 관리자가 제품을 보충함 (상태 변화 필요)
  - **`clearItems`**: 관리자가 제품을 모두 제거함 (상태 변화 필요)
- 관리자와 고객의 상호작용이 함께 모델링되어 있어 복잡함 측면이 있다. 관리자와 상호작용하는 부분은 상태 패턴을 모델링할 때 포함하지 않아도 된다.
- **`selectItem`**의 경우 **`ChangeNotAvailableException`** 예외를 발생할 수 있다. 고객이 투입한 돈으로 선택한 음료를 구매할 수 있지만 필요한 거스름을 제공하지 못해 구매가 불가능한 경우 다른 음료를 구매할 수 있는 경우와 구매할 수 있는 것이 없는 경우를 구분해야 하며, 전자의 경우에는 예외를 생성할 때 **`false`**를 인자로 전달해야 한다.
- 자판기에서 거스름은 가장 높은 액면가 지폐 또는 동전을 이용하여 지불하도록 구현해야 한다. 예를 들어 1,700원을 거스름으로 주어야 하는데, 현재 자판기에는 1,000원 짜리가 2개, 500원 짜리가 4개, 100원 짜리가 10개 있으면, 1,000원 짜리 하나, 500원 짜리 하나, 100원 짜리 2개로 거스름을 구성하여 준다.
- **`CashRegister`**라는 클래스를 통해 자판기가 유지하는 돈, 사용자 투입한 돈, 거스름 돈을 모델링한다. 사용자가 돈을 투입하면 이 돈은 자판기에 유지하는 돈에 추가되는 형태로 구현한다. 그 이유는 거스름은 사용자가 투입한 돈까지 고려하여 만들 수 있기 때문이다.
- GUI와 별도로 **`VendingMachine`** 클래스를 테스트할 수 있는 테스트 프로그램을 제공한다. 처음에는 GUI와 무관하게 이 테스트 프로그램을 이용하여 개발하는 것이 더 효과적일 수 있다.