객체지향개발론및실습

Laboratory 8. State 패턴

1. 목적

- 객체 내부 상태가 변경되었을 때 행동 패턴을 바꿀 수 있도록 해주는 State 패턴을 실습함

2. 개요

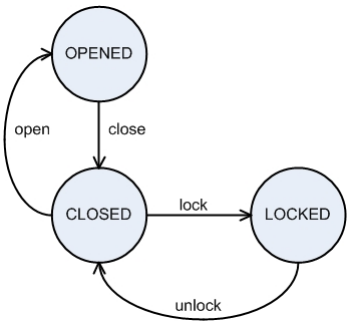

그림 2.1: 문 상태도

- 그림 2.1에 주어진 일반 문에 대한 상태도를 바탕으로 기존 방법과 state 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.
- 완전한 소스는 el에서 다운받을 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. 실습 1. 상태 중심 전이 방법

- 상태 객체에서 상태를 전이하는 상태 중심 전이 방법의 State 패턴을 이용하여 Door를 구현하시오.
- 기본 소스는 el에 제공됨

4.1 DoorState 인터페이스

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

- 인터페이스 대신에 추상 클래스로 정의하고 각 메소드는 기본적으로 예외를 발생하도록 구현할 수 있음. 이렇게 하면 이 상태를 구현하는 클래스들을 작성하는 것이 편리할 수 있음
- el에 제시된 소스를 보면 출력문이 많이 있음. 이 문들은 실제 응용에서는 사용되지 않는 부분임. 구현 과정을 확인하기 위해 추가한 부분임을 인식하고 상태 패턴을 이해할 필요가 있음

4.2 Opened, Closed, Locked 클래스

- 모두 DoorState 인터페이스를 구현하여 정의함
- 예)

```
public class Opened implements DoorState {
    private Door door;
    public Opened(Door door){
        this.door = door;
    }
    @Override
    public void open() {
        System.out.println("이미 열려 있음");
    }
    @Override
    public void close() {
        System.out.println("문이 닫힘");
        door.changeToClosedState();
    }
    @Override
    public void lock() {
        System.out.println("열려 있는 상태에서는 잠글 수 없음");
    }
    @Override
    public void unlock() {
        System.out.println("문이 열려 있어 잠금을 해제할 필요가 없음");
    }
}
```

- 이 예에서 알 수 있듯이 오히려 코딩해야 하는 양은 늘어날 수 있음

- 이 예처럼 상태 객체에서 context 객체를 멤버로 유지할 수 있지만 상태 객체를 싱글톤으로 모델링하고 싶으면 context 객체를 모든 메소드의 인자로 전달하는 방법을 사용할 수도 있음

4.3 Door 클래스

```
public class Door {
    private final DoorState openedState = new Opened(this);
    private final DoorState closedState = new Closed(this);
    private final DoorState lockedState = new Locked(this);
    private DoorState currentState = closedState;
    public void open(){
        currentState.open();
    }
    public void close(){
        currentState.close();
    }
    public void lock(){
        currentState.lock();
    }
    public void unlock(){
        currentState.unlock();
    }
    public void changeToOpenedState(){
        currentState = openedState;
    }
    public  void changeToClosedState(){
        currentState = closedState;
    }
    public  void changeToLockedState(){
        currentState =  lockedState;
    }
}
```

- 테스트 프로그램은 el에 제공된 다음을 사용함

```
public class Test {
    public static void main(String[] args){
        Door door = new Door();
        door.close();
        door.lock();
        door.open();
        door.unlock();
        door.open();
    }
}
```

- 이 프로그램의 실행 결과는 다음과 같음. 출력되는 문구는 정확하게 일치할 필요는 없음.

```
문이 이미 닫혀 있음
문을 잠금
문이 잠겨 있어 열 수 없음
문의 잠금을 해제함
문이 열림
```

5. 실습 2. 문맥 중심 상태 전이 방법

- 실습 1에서 완성한 프로젝트를 복사한 후에 문맥 중심 상태 전이 방법을 사용하도록 DoorState **interface**를 다음과 같이 수정하고, 각 상태를 나타내는 클래스도 수정함

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

6. 실습 3. 열거형을 이용한 상태 패턴의 구현

- 이 실습에서는 자바의 열거형을 이용하여 상태 패턴을 구현함
- 문맥 중심 전이 방법에서는 상태 대신에 문맥에서 상태를 전이함. 이를 위해 상태 전이의 필요 여부를 **boolean** 변수를 통해 알리는 방법을 사용할 수 있음. 하지만 열거형을 사용할 경우 각 상태를 나타내는 객체가 열거형 상수이기 때문에 상태 객체 간에 밀접하게 결합되어 있음. 따라서 상태 전이가 필요하다 것만을 알리는 것이 아니라 어떤 상태로 전이해야 하는지를 반환하도록 구현할 수 있음. 따라서 이와 같은 구현은 정확하게 상태 중심, 문맥 중심으로 구분하기 어려운 측면이 있음. 이 실습에서는 상태 객체에서 상태를 반환하는 방법으로 구현함
- 열거형 `DoorState` 다음과 같이 구현함

```
public enum DoorState {
    OPENED{
    },
    CLOSED{
    },
    LOCKED{
    };
    public abstract DoorState open();
    public abstract DoorState close();
    public abstract DoorState lock();
    public abstract DoorState unlock();
}
```

만약 디버깅을 위한 콘솔에 문자열 출력이 필요 없으면 `open` 메소드의 기본 메소드를 구현하면 각 상태를 구현하기 편리해질 수 있음

```
public DoorState open(){
    return this;
}
```

- 실습1, 실습2, 실습3을 비교 분석하시오.