# State Pattern

# 결론 및 정리

## 정의

### FSM :

> *FSM 은 유한 상태 기계 (Finite State Machine) 의 약자로, 시스템의 동작을 모델링 및 제어하는 "모델 (모형)" 이라고 보면 됨.*

주어진 입력에 따라 시스템의 상태가 변화. 이에 따라 다른 동작이 수행.
내부의 주요 개념은 다음과 같다.

- Event : 시스템에서 발생하는 변화나 입력을 의미. 상태 전이를 유발, Trigger의 역할을 함.
- Transition (상태 전이) : 다음 상태로 전이하는(바뀌는) 동작을 의미함.
- State : 시스템이 취할 수 있는 상태. 상태는 시스템의 동작과 속성을 직접적으로 결정하는 요소임.

=> 이 FSM 의 개념이, 넓은 범위의 State Pattern이라 할 수 있으므로, 해당 모델을 먼저 설명하였음.

### State Pattern

객체의 내부 상태에 기반하여 동작을 변경하는 패턴. State 패턴은 상태에 따라 객체의 동작을 다르게 처리해야 할 때 사용. 상태 클래스 간의 전환을 통해 상태를 가지는 객체 (Context) 의 동작을 변경함.

- Context : State를 가지고, State가 변경 됨에 따라 다른 동작을 수행하는 Class.
- State Interface : State 의 기본이 되는 Interface. State의 영향을 받는 동작들이 interface 의 method 로서 구현되어 있음. Context 는 State 에서 이 method를 실행함으로써, 다른 동작을 실행시킬 것.
- Concrete State Class : State Interface 를 구현한 Class. 영향을 받는 동작들이 Override 되어 실제 알고리즘으로서 존재함. 각 Concrete State Class 마다 각 Method의 구현이 다를 것.

### State vs Strategy

> *결론 : 목적과 테마는 다름. 형태는 동일. (일부 다르긴 한데, 이에 대해서도 자세히 써볼 것. // 결국 목적이 다름에 따라 분화된 내용)*

- Strategy :
  - 목적 : 동작
  - 형태 차이 :
    - Context Class 가 여러 행동이 있다면, 여러 Strategy 를 가지는 것이 자연스러움.
    - 하나의 Strategy Interface는 하나의 동작을 의미하므로, 하나의 method를 가지는 것이 자연스러움. (여러 method 를 포함할 수 있긴 하나, 응집력, 관련성이 충분히 있어야 함.)
    - 각 Concrete Strategy가 서로를 알아둘 필요가 없음. (dependency)
- State
  - 목적 : 상태
    - Context Class 가 State군 (각 State 묶음) 을 여러 개 가질 수도 있긴 하겠지만 (아예 종류나 범주가 다른 State 라면) 자연스럽지는 아니함.
    - 해당 State Interface는 Context Class 가 상태가 바뀜에 따라 바뀔 수 있는 모든 Method 에 대한 method 를 가지고 있어야 하므로, Strategy Pattern 과 달리 method가 여러 개인 것이 자연스러움.
    - 각 Concrete State 가 Context Class 의 상태를 변경시키려면, 서로를 사용하여야 함.

## 구현

1. Context Class 가 있다 하자.
2. 상태에 따라 다른 코드를 실행하여야 하는 Method 들을 선언하는 곳에 집어넣은 Interface 인 State Interface를 만든다.
3. Context Class 에 State 를 Type 으로 가지는 Variable 'a' 을 하나 만들고, 기존 State에 따라 나뉘어 졌던 내용을 a.a내부method이름(~) 으로 대체한다.
4. 각 상태에 따른 Concrete State Class 를 만든다. 이는 해당 state 에 맞는 method 동작을 가지도록 한다.
5. Context Class에 Concrete State Class 를 넣어가며, 상태를 전환하면서 사용한다.

## Context, Problems, Merit

- State 를 바꿈에 따라 간단하게 Context Class (상태를 가지는 Class) 의 동작을 바꿀 수 있다.
- Encapsulated 되어 있는 것이나 마찬가지 이므로, Context Class 는 State 및 그 내부에 들어 있는 Class 에 대해 신경 쓸 필요가 없다.
- 기존 If 문 등을 사용하였을 때보다, Context Class의 코드가 훨씬 간결해진다.

---

---

---

# 강의 들으며

Game Programming Patterns -> 자료로서 이걸 제시해주심.

Object 의 State 를 Runtime 중에 변경하고, 이를 바탕으로 동작이 일어나도록 하는 것.

## FSM에 관한 개념 설명

- program의 일부분이 들어갈 수 있는 창구가 State 인데, 이러한 State를 유한개 가진 장치를 Finite State Machine 이라 함.
- FSM은 Software 이론에서의 Concept 임.
- 각 상태에서 전체 프로그램은 다으게 동작함.
- Program은 다른 상태로 전이 (transition) 할 수 있음.

FSM은 "Finite State Machine"의 약자로, 유한 상태 기계를 나타냅니다. FSM은 상태(State), 상태 전이(Transition), 이벤트(Event) 등의 개념을 사용하여 시스템의 동작을 모델링하고 제어하는 수학적인 모델입니다. FSM은 다양한 분야에서 사용되며, 컴퓨터 과학, 전기 및 전자 공학, 자동화 등 다양한 영역에서 유용하게 적용됩니다.

> *즉, FSM 은 유한 상태 기계 (Finite State Machine) 의 약자로, 시스템의 동작을 모델링 및 제어하는 "모델 (모형)" 이라고 보면 됨.*

주어진 입력에 따라 시스템의 상태가 변화. 이에 따라 다른 동작이 수행. 내부의 주요 개념은 다음과 같다.

- Event : 시스템에서 발생하는 변화나 입력을 의미. 상태 전이를 유발, Trigger의 역할을 함.
- Transition (상태 전이) :다음 상태로 전이하는 동작을 의미함.
- State : 시스템이 취할 수 있는 상태. 상태는 시스템의 동작과 속성을 직접적으로 결정하는 요소임.

=> 아마 이 FSM 의 개념이 State Pattern 의 개념과 굉장히 닮아 있기 때문에, 이를 서술하는 것으로 보임. 또 FSM 은, 하드웨어 시스템이어도 그냥 적용될 수 있는, 추상적인 개념이자 범위가 넓은 개념이니까!

## Gumball machine FSM

## 다시 돌아와서, State Pattern

구현 관련 내용 설명
constant(final : 변경 불가능한) 나 enum 생성. 이들은 상태가 될 것임.
그 후, 해당 상태를 지닐 수 있는 Instance Variable (Field) 를 만들기.

이후, 실질적으로 구현을 진행함. 해당 object 가 할 수 는 모든 일 작성하기. 그리고 이 method, 행동들은 실제로 구현될 것이고, 이는 state를 바꾸는 동작을 포함할 수 있음.

State 에 따른 행동의 분기는 if 문을 통해 구현되었음.

=> 만약 여기 나온 대로 구현되어 있으면, 각 동작 (method) 들은, 각 state 에 대한 분기를 가지므로, 너무 비효율적이게 됨!
=> 확장되거나 할 때도 마찬가지. 각 행동에 해당하는 method 들마다 state에 대한 분기를 가지는 것은 좋지 않다!

=> 바꾼 건 어떻게 동작할 것인가?
State를 interface 로 두고, State 내에 해당 행동에 관한 내용이 존재한다.
이를 구현하는 Concrete Class 가 직접적인 행동을 결정한다.
해당 State는 직접적으로 이를 가질 object 를 가진다.

Method 중 찐또배기를 한다면, State를 바꾸고, 실질적인 동작이 이루어질 것임!
이걸 사용하는 친구는 자신 내부에 든 State 를 신경 쓸 필요가 없음.
State 가 끝나는, 더 전환될 게 없는 부분이라면, End를 할 필요가 있을 수도 있음.

state의 내부 상태에 따라서, 이걸 사용하는 친구가 다른 동작을 하도록 하는 게 state의 주요 임무.
용어를 설명해보자.

- Context : State를 가지고, State가 변경 됨에 따라 다른 동작을 수행하는 친구.
- State Interface : State 의 기본이 되는 Interface. State의 영향을 받는 동작들이 interface 의 method 로서 구현되어 있음.
- Concrete State Class : State Interface 를 구현한 Class. 영향을 받는 동작들이 Override 되어 실제 알고리즘으로서 존재함.

Strategy 와 비슷하다는 느낌을 받은 이유 :

- Strategy는 각 Task 들이 존재하고, 그 Task 들이 여러 기능으로 동작할 수 있을 때 해당 이를 Strategy Interface 와 여러 개의 Concrete Strategy 를 입력함으로써 해결함. 각 Strategy가 의미하는 바는, 각각에 해당하는 행동이고, Context 는 이걸 쓰는 것. 행동이므로, 여러 다른 Strategy 가 들어갈 수 있으며, Strategy 는 보통 하나로 대변됨.
- State 는 애초에 Object 의 상태를 나타내는 것. State라는 interface 가 있고, Context 는 이를 하나만 가지는 것이 일반적일 것.
  단, 하나의 행동으로서 나타났던 Strategy 와는 달리, State는 상태에 따라서 변화하는 Context의 행동에 대해서 모두 interface의 method 로서 구현되어 있으며. 이는 차이를 야기함. 아니, 차이임.

결국 형태 자체는 동일한 듯?
차이라고 한다면,

- Strategy 는 동작에 집중.
- State 는 상태에 집중.

=> 이에 따라 오게 되는 차이점 :

- Strategy Pattern 은 Context Class 가 여러 Strategy 를 가지는 것이 자연스럼. 또한 하나의 Strategy Interface는 하나의 동작을 의미하므로, 하나의 method를 가지는 것이 자연스러움. 물론, 여러 method 가 포함될 수 있긴 하나, 동일한 환경에서 연쇄적으로 나오거나 하는 등 - 관련되어 있는 기능이 아니라면, 따로 하나 더 Strategy 를 만드는 게 나을 듯.
  (Chat GPT 답 : 여러 개의 메서드를 동일한 동작의 다른 측면을 다루는 것으로 판단되는 경우, 해당 메서드들을 하나의 인터페이스에 포함할 수 있습니다. 그러나 이 경우 메서드들이 상호 연관되어야 하며, 함께 사용될 수 있는 의미 있는 단위여야 합니다. 메서드들 간에 응집력이 있고 서로 종속적인 경우에만 하나의 인터페이스에 포함하는 것이 좋습니다. - 관련성과 일련성. 적절한 분리와 응집.)
- State Pattern 은 Context Class 가 여러 State를 가질 수도 있긴 하겠지만 (아예 종류나 범주가 다른 State 라면) 자연스럽지는 아니함. 또한, 해당 State Interface는 Context Class 가 상태가 바뀜에 따라 바뀔 수 있는 모든 Method 에 대한 method 를 가지고 있어야 하므로, Strategy Pattern 과 달리 여러 개인 것이 자연스럽다.
- 또한, State Pattern 은 각 State가 서로 전환하려면, 서로를 알아두는 구조이기도 하겠구나! Strategy Pattern 은 Concrete Strategy 가 서로를 알 필요가 전혀없다!
- 목적과 테마가 다르고. 구조는 엇비슷하나, 목적이 다르기에 오는 차이가 존재한다.