Strategy Pattern

결론 및 정리

Strategy Pattern 정리

정의

전략 패턴은 다음과 같다.

1. 작업 및 행동 "들" 이 있고, "각" 작업 및 행동이 "여러 형태 (여러 알고리즘)" 으로 구현될 수 있는 경우.
2. 해당 각 작업을 interface or abstract class 를 통해 맞닿는 부분을 만든다. 이렇게 만들어진 interface 혹은 abstract class 를 **Strategy** 라 한다.
    - Strategy 는 해당 작업 및 행동을 수행하는 method 를 포함하고 있어야 함.
3. 만들어진 Strategy를 기반으로 **Concrete Strategy** 를 제작한다. 이는 실제 구현된 Concrete Class 이며, 작업 및 행동에 대한 구체적인 형태 (알고리즘) 을 구현한다.
4. Strategy를 변수로서 가지는 Class 를 만들어, Concrete Strategy를 바꿔가며 해당 Class를 동작시킬 수 있다. 이렇게 Strategy Object 를 사용하는 Client Class를 **Context** 라 한다.

구현

위에 구현에 관한 내용도 작성되어 있으나, 더 자세히 작성하여 보자면 다음과 같다.

1. 각 작업의 추상적인 부분을 interface, 혹은 abstract class 로 제작한다.
    - 이 때 해당 작업을 변수로서 가지는 Context 가 실행시킬 동작을 해당 interface 등의 method 로 할당한다.
2. Strategy 기반의 Concrete Class 를 제작한다.
3. Interface / Abstract Class 를 변수 Type 으로서 가지는 Context 를 제작한다.

Context & Problem

어떤 때에 Strategy Pattern 을 사용하여야 할까?

> *단순하게 정리하자.*
> *각각의 동작이 다양한 알고리즘(동작 / 형태)로서 구현될 수 있을 때.*

(해당 동작을 잘 전환할 수 있게 하는 등의 효과를 가짐.)

Goal

Strategy Pattern 을 사용함으로써 무엇을 얻을 수 있을까?

단순하게 정리해서, Loose Coupling 과, 코드를 낭비하지 않는 효과와, Polymorphism 의 효과를 얻을 수 있음.

애초에, Context 나 Problem 이 이와 비슷한 의미를 가지기 때문에.

Design Pattern Principles

> *Design Principle 1 : 다변화 (vary) 될 수 있는 것이 뭐가 있는지 알고, 이를 다른 것과 분리하여 두라.*

> *Design Principle 2 : interface 를 최대한 이용하라.*
> *Dependency injection 등을 신경쓰라!*

> *Design Principle 3 : composition (has a) 가 inheritence 보다 권장된다. (Code 를 Flexible 하게 만들 수 있기 때문에)*

Structure, Design, Diagram에 대한 내용과 맞닿아 있다. 구조를 짜는 것. 따라서, 언어에 종속적이지 않으며, 소통에 있어서 잘 사용할 수 있다.

강의 들으며

> *중점적인 내용만 기록*

Class 가 가질 수 있는 행동이 여러 개 있을 때.
Duck 이라는 Class 가 있고, Flying이랑 Quacking 이라는 기능을 추가하여야 할 때의 예시.

1. 그냥 상속 쓰면, 해당 동작을 하면 안 되는 SubClass 에서 해당 동작을 막을 수 없음.
2. Interface 를 쓰면, Polymorphism 에 따른 추상화 및, 그 추상화 내용에 대해 호출하는 기능을 사용할 수 없음.

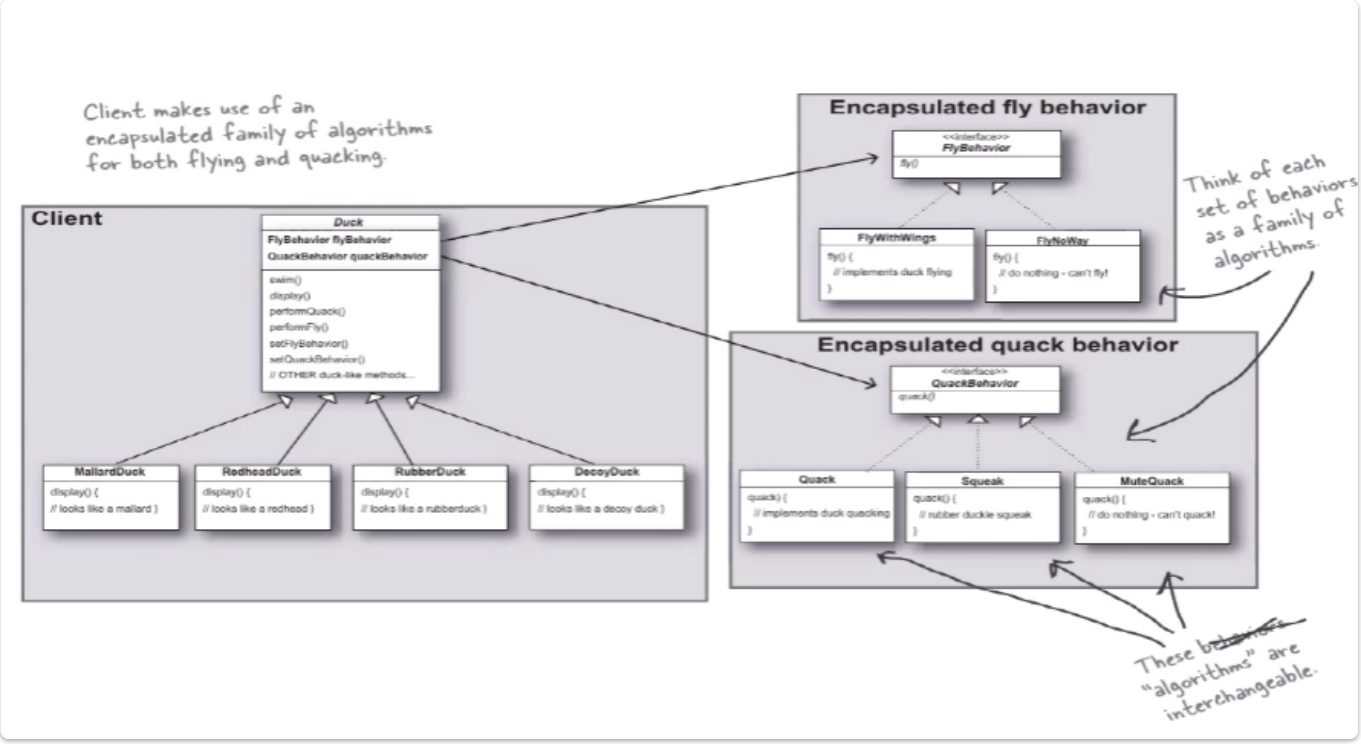
> *Design Principle 1 : 다변화 (vary) 될 수 있는 것이 뭐가 있는지 알고, 이를 다른 것과 분리하여 두라.*

=> 이 Design Pattern 원칙을 준수하면> => 각 Behavior Type 에 대한 Interface를 만들고, Duck 이 해당 Interface를 Implements 한 Class 를 가지고 있을 수 있도록 한다~!

> *Design Principle 2 : interface 를 최대한 이용하라.*
> *Dependency injection 등을 신경쓰라!*

그 이후 직접적인 구현의 방식을 보여주심.
Behavior 에 대한 interface 를 각각 만들고, 이를 구현하는 각 종류의 Behavior Class 들을 만듬.
interface 를 variable 로서 가지는 (association) 행동을 가지는 A Class 가 존재하여, 해당 Behavior Class 를 활용함.

A Class 는 가진 행동을 기반으로, 사용하는 method, getter 와 setter method 가 존재하며, Subclass 에선 이를 할당 받음.



이후 직접 코드 짜는 과정을 보여주심.
언제, 어떻게 쓰는지는 알겠는데. 이와 같은 동작의 경우를 제외하고는, 언제 이 구조를 사용할 수 있을지 약간 의문임.

=> 이에 관한 의문은 Strategy Pattern 의 내용을 보면 쉽게 알 수 있음.

> *Design Principle 3 : composition (has a) 가 inheritence 보다 권장된다. (Code 를 Flexible 하게 만들 수 있기 때문에)*

Strategy Pattern의 정의

- algorithms의 묶음으로서 존재하는 Design Pattern. 내부에 들어있는 각각을 encapsulates 하고, 그들을 바꿀 수 있도록 해줌. Client 가 어떤 algorithm을 사용할지를 선택할 수 있도록 함.

- 각 Task 에 대해 수행할 수 있는 여러개의 알고리즘이 존재할 때, 알고리즘을 무얼 선택할지를 runtime 중에 바꿀 수 있으므로, 이러한 경우 유의미함.

Duck 에서 각각의 Strategy 가 의미하는 것

- Algorithm : flying 이나 quacking 과 같은 행동들
- algorithms are encapsulated : Ducks 가 해당 알고리즘을 사용하나, method 로 구현되어 있어, 그게 어떻게 구현되어 있는지를 알지 못함.
- 하나의 알고리즘을 다른 알고리즘으로 바꾸기가 쉬움.
- 위는 문제상황과, 전략 패턴의 사용 및 용례를 알아보았음.
- 정확한 정의 자체는 여기에 나와 있는 내용이므로, 이를 집중적으로 알아볼 필요가 있을 듯함.

추가 조사 / 생각 내용

전략 패턴(Strategy Pattern)은 소프트웨어 디자인 패턴 중 하나로, 알고리즘군을 정의하고 각각을 캡슐화하여 상호 교환 가능하도록 만드는 방법을 제공합니다. 이 패턴은 알고리즘을 사용하는 클라이언트와 알고리즘을 구현하는 클래스들 사이의 결합도를 줄이고, 유연성과 확장성을 향상시키는 데 사용됩니다.

=>
하나의 알고리즘은, 하나의 "작업 / 역할" 을 의미함.
캡슐화란 데이터와 클래스를 하나의 단위로 묶고, 이를 외부에서 직접적으로 접근하지 못하도록 제한함으로써 데이터의 무결성을 보호하는 것. // 뭐 이리 어려운 개념인가 싶은데, 감싸서 드러내고 싶은 것만 드러내는 것이라고 생각하면 될 듯.
따라서 인터페이스 혹은 추상 클래스를 이용하여 Strategy (알고리즘의 묶음으로서의 역할을 수행하는 추상 클래스 혹은 인터페이스) 를 만들고, Strategy 를 바탕으로 Concrete Strategy 의 구체적인 내용을 호출할 수 있다면, 그건 캡슐화라고 볼 수 있을 듯. interface 나 abstract class 를 통해 접근함으로써 그 내부의 구체적인 내용 및 데이터를 은닉하고 있는 것이니까.