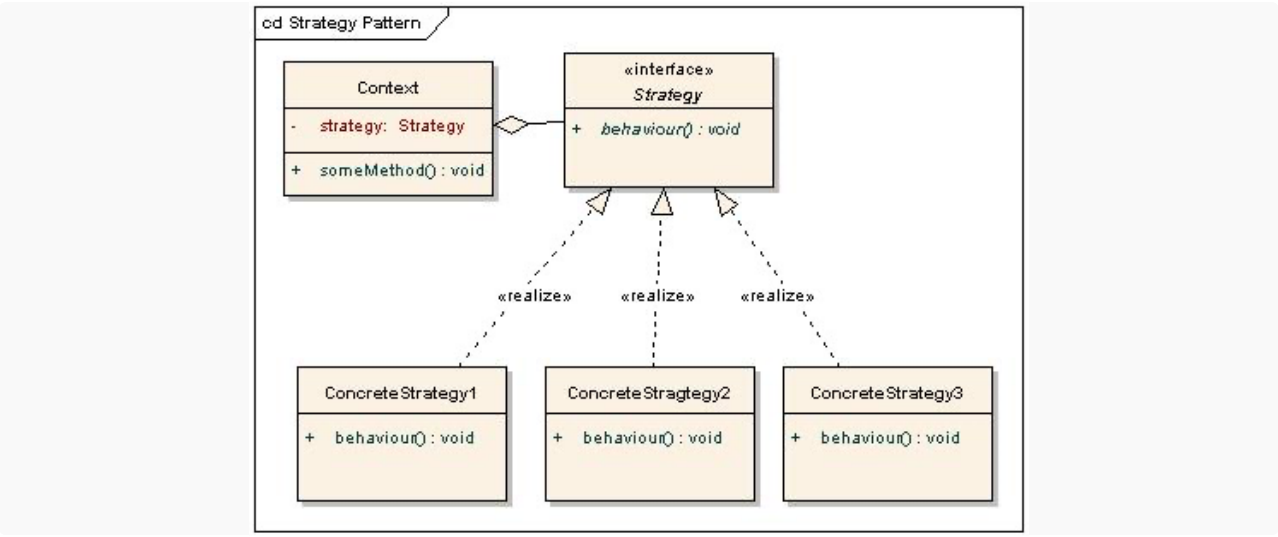
스트래티지 패턴 (Strategy Pattern)



스트래티지 패턴의 개념과 특징

스트래티지 패턴(Strategy Pattern)은 객체 지향 디자인 패턴 중 하나로, 알고리즘 군을 정의하고 각각을 캡슐화하여 교환해서 사용할 수 있도록 만든 패턴입니다. 쉽게 말해, 같은 문제를 해결하기 위해 여러 알고리즘이 필요할 때, 이를 객체화하여 교환 가능하게 만들어 주는 패턴입니다.

스트래티지 패턴의 특징은 다음과 같습니다.

1. 각 알고리즘을 캡슐화하여 독립적인 행동을 만듭니다.
2. 클라이언트는 알고리즘을 직접 선택하지 않습니다.
3. 새로운 알고리즘을 추가하기 쉽습니다.
4. 코드 재사용성을 높입니다.
5. 상속을 사용하지 않고도 다형성을 구현할 수 있습니다.

스트래티지 패턴을 사용하면, 알고리즘을 자유롭게 교체하면서 동작을 변경할 수 있습니다. 예를 들어, 정렬 알고리즘을 구현할 때, 버블정렬, 퀵정렬, 삽입정렬 등 다양한 알고리즘이 있습니다. 이 때, 스트래티지 패턴을 사용하면 각각의 정렬 알고리즘을 클래스로 정의하고, 클라이언트에서는 어떤 알고리즘을 사용할지 선택하여 사용할 수 있습니다. 이렇게 구현하면, 새로운 정렬 알고리즘을 추가할 때마다 기존 코드를 수정할 필요 없이 새로운 알고리즘을 추가할 수 있습니다.

스트래티지 패턴의 구성 요소

컨텍스트(Context)

스트래티지 패턴을 이용하고자 하는 클래스입니다. 컨텍스트는 인터페이스를 통해 스트래티지를 호출하며, 스트래티지 패턴을 이용하여 동적으로 알고리즘을 변경할 수 있습니다.

스트래티지(Strategy)

알고리즘을 캡슐화한 인터페이스입니다. 이 인터페이스는 컨텍스트에서 호출되어 실행됩니다. 스트래티지는 여러 개의 구현 클래스를 가질 수 있으며, 이들은 모두 동일한 인터페이스를 구현합니다.

구체적인 스트래티지(Concrete Strategy)

스트래티지의 구현 클래스입니다. 스트래티지 인터페이스를 구현하여 알고리즘을 실제로 수행합니다. 각각의 구체적인 스트래티지는 독립적으로 변경될 수 있으며, 실행 시에 컨텍스트에서 원하는 스트래티지를 선택하여 사용합니다.

스트래티지 패턴과 상속의 차이점

상속은 부모 클래스의 특징을 자식 클래스가 물려받는 관계를 나타내며, 자식 클래스에서는 부모 클래스에서 정의한 메서드를 오버라이드하여 동작을 변경할 수 있습니다. 이러한 상속 관계는 코드의 재사용성을 높여주며, 코드의 일관성과 가독성을 유지할 수 있습니다.

반면, 스트래티지 패턴은 동일한 문제를 해결하기 위한 여러 알고리즘을 정의하고, 각 알고리즘을 캡슐화하여 이들을 상호 교환 가능하게 만드는 패턴입니다. 즉, 알고리즘을 사용하는 클라이언트와 알고리즘 자체를 분리하고, 클라이언트가 실행 시점에 알고리즘을 선택할 수 있게 만듭니다.

스트래티지 패턴은 상속과 달리 각 알고리즘을 독립적인 클래스로 구현하며, 클라이언트와 알고리즘 클래스 간에는 인터페이스를 통한 의존성이 있습니다. 이를 통해 알고리즘 클래스를 쉽게 교체하거나 확장할 수 있습니다. 또한, 상속과 달리 실행 시점에 알고리즘을 선택하기 때문에 더욱 유연한 코드를 작성할 수 있습니다.

따라서, 상속은 부모 클래스에서 정의한 기능을 자식 클래스가 물려받는 관계를 나타내며 코드 재사용성을 높여주고, 스트래티지 패턴은 동일한 문제를 다양한 알고리즘으로 해결하기 위한 패턴으로 코드 유연성을 높여줍니다.

스트래티지 패턴의 장단점

장점

- 확장성: 알고리즘을 쉽게 추가하거나 변경할 수 있습니다. 새로운 전략을 구현하고 기존의 전략과 교체하는 것이 쉽습니다.
- 유지보수성: 각 전략을 별도로 구현하기 때문에 코드가 간결해집니다. 따라서 코드 수정 및 유지보수가 쉬워집니다.
- 테스트 용이성: 전략을 단위 테스트하기 쉽습니다. 각 전략을 별도로 구현하기 때문에 개별적으로 테스트할 수 있습니다.

단점

- 클래스가 많아질 수 있음: 많은 수의 전략을 구현하기 위해서는 많은 클래스가 필요합니다. 이로 인해 클래스의 개수가 증가하고 코드 구조가 복잡해질 수 있습니다.
- 런타임 오버헤드: 전략을 선택하는 데 필요한 추가 로직이 필요하기 때문에 런타임 오버헤드가 발생할 수 있습니다. 하지만 이는 대부분 무시할 수 있는 수준입니다.

스트래티지 구현 예제

```
// 전략 객체를 정의합니다.
const strategyAdd = {
  execute: function(num1, num2) {
    return num1 + num2;
  }
}

const strategySubtract = {
  execute: function(num1, num2) {
    return num1 - num2;
  }
}

const strategyMultiply = {
  execute: function(num1, num2) {
    return num1 * num2;
  }
}

// 컨텍스트 객체를 정의합니다.
function Calculator(strategy) {
  this.strategy = strategy;

  this.executeStrategy = function(num1, num2) {
    return this.strategy.execute(num1, num2);
  }
}

// 전략을 사용하여 계산합니다.
const calculatorAdd = new Calculator(strategyAdd);
console.log(calculatorAdd.executeStrategy(5, 3)); // 출력 결과: 8

const calculatorSubtract = new Calculator(strategySubtract);
console.log(calculatorSubtract.executeStrategy(5, 3)); // 출력 결과: 2

const calculatorMultiply = new Calculator(strategyMultiply);
console.log(calculatorMultiply.executeStrategy(5, 3)); // 출력 결과: 15
```