참고 자료

- https://refactoring.guru/design-patterns/strategy

Intent

분류

- behavioral design pattern
    - object의 behavior을 캡슐화하고 요청을 object에 위임하는 패턴

정의

- 알고리즘의 집합을 정의하고, 각각 개별 클래스로 놓은 뒤 그 객체들을 바꿀 수 있도록 하는 패턴

Problem

상황

- 네비게이션 앱을 생각해봅시다.
- 개발 초기에 자연 경관을 볼 수 있는 루트를 알려주는 기능을 만들었습니다.

- 시간이 흘러 점점 요구사항이 점점 추가됩니다.

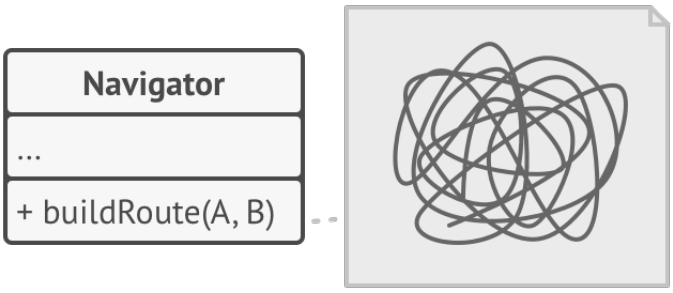
1. 도보로 걸을 수 있는 루트를 제공해주는 기능을 추가해주고...
2. 대중교통을 통해 갈 수 있는 루트를 제공해주는 기능을 추가해주고...
3. 심지어 특정 도시를 경유하여 갈 수 있는 루트를 제공해주는 기능을 추가하였습니다.

문제

main에서 경로를 제공하는 로직을 작성했어서, 매 기능이 추가될 때마다 main class의 코드가 2배로 길어지는 문제가 생겼습니다.

또 분리되지 않았기 때문에 알고리즘을 수정 및 보완할 때마다 버그가 발생합니다.

어떻게 해야 좋은 구조가 될까요?



Solution

핵심

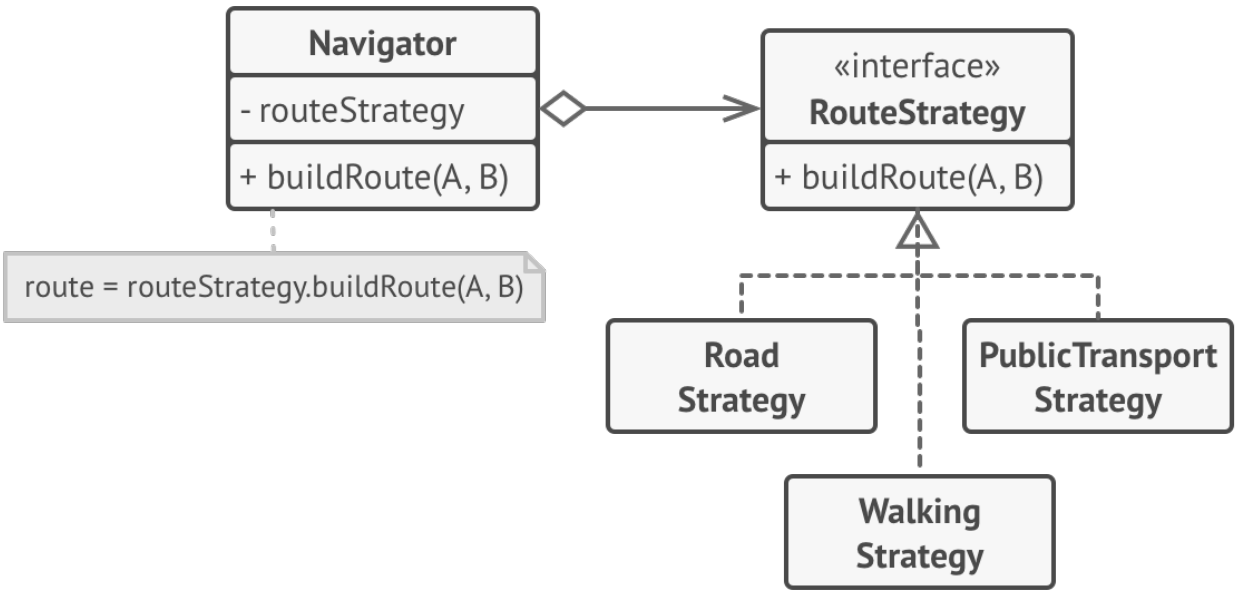
각 알고리즘을 하나의 클래스로 분리하고 인터페이스를 두어 알고리즘을 사용합니다.

방법

용어

Navigator는 **context**라 부르고, 특정 루트를 선택하는 알고리즘들은 **strategy**라 부릅니다.

> context라 불리는 이유는 strategy를 사용하는 **맥락**의 의미를 강조하고 싶어 context라하는 것 같네요.



1. Navigator(context)

- 네비게이터라는 클래스를 만들고 그 안에 strategies를 참조할 수 있는 field를 지정합니다.
- 그리고 네비게이터는 루트 찾는 로직을 strategy에게 위임합니다.
- 이 strategy는 인터페이스 타입으로 선언되어 있어 특정 길을 찾는 알고리즘에 종속되지 않습니다.

2. RouteStragegy

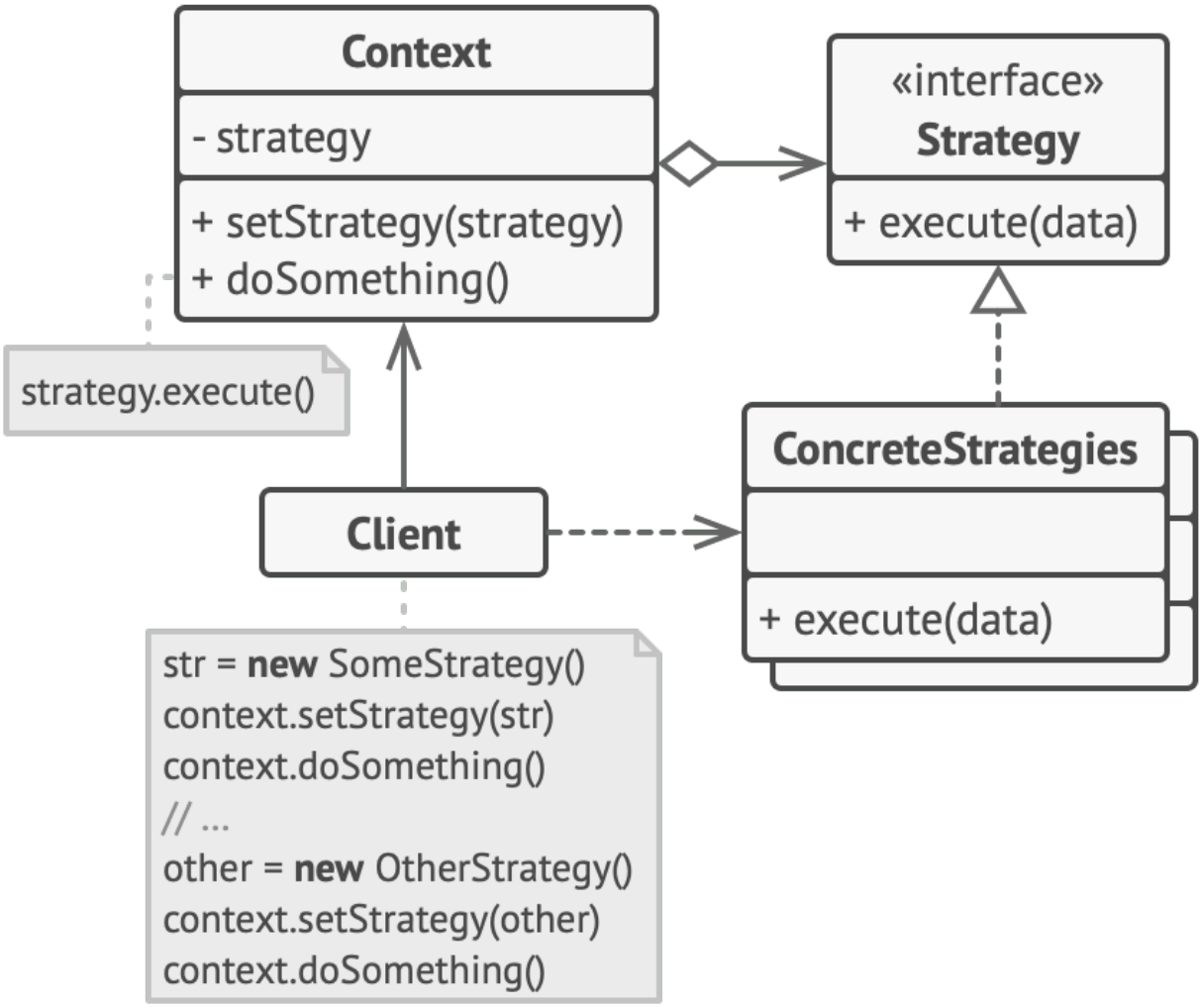
- 길을 찾는 알고리즘에 필요한 method를 선언합니다.

3. Concrete Strategy(ex: road, walking, publicTransport....)

- 매 새로운 알고리즘을 추가할 때마다 routeStrategy를 구현한 concrete Strategy를 만듭니다.
- 여기에서 알고리즘이 수행되는 것이죠.

이와같이 하면 main class가 지저분해질 필요도 없고, 길을 찾는 알고리즘들을 분리시킬 수 있게 됩니다.

구조



1. Context

- strategy interface로 선언된 reference를 통해 concrete strategies 중 하나와 소통합니다.
- 클라이언트가 현재 strategy object를 바꿀 수 있도록 setter 메서드를 노출합니다.

2. Strategy

- 모든 concrete strategy의 공통된 인터페이스입니다.
- strategy를 실행시키기 위해 context가 사용할 method를 정의합니다.

3. ConcreteStrategies

- context가 사용할 알고리즘을 구현합니다.

4.Client

- 특정한 strategy object를 생성하고 context에 넘겨줍니다. 다시말해서 context는 stratgy를 생성하고 변경할 책임이 없습니다.
- 또한 런타임 중 context의 현재 strategy object를 변경할 수 있습니다.