Command Pattern

결론 및 정리

정의

> *Command Pattern 은*
>
> *요청("어떠한 동작들을 할 수 있는(Receiver 의 Method : **Action1, 2 ... ()**) 객체(**Receiver**)" 에게, "뭘 해달라는 것") 을*
>
> *객체로 만들어 관리하는 Pattern 이다.*

- **Requeest=Command** 는 "어떠한 동작들을 할 수 있는(Receiver 의 Method : **Action1, 2 ... ()**) 객체(**Receiver**)" 에게, "뭘 해달라는 것 적혀있는(**execute()**)" **객체**다.
  - 이렇게 Request 자체가 Command Class 라는 형태로 감싸여져 있다는 것이 중요하다.
- Command 를 추상화 / 캡슐화 하기 위해서 Command Interface 와 Concrete Command Class 로 나눈다.
- 여기에 Command 를 실제로 만드는 **Client** 와 / Command 들을 받아 관리하고, 실제로 해당 Command 를 실행시켜주는 **Invoker**가 추가되었다고 생각하면 쉽다.
- 요청을 객체로 포장하였기 때문에, 오는 장점이 굉장히 많다.
  - Command를 execute 해주는 친구 (Invoker) 와, 그에 따라 실제로 action(동작) 하는 친구 사이의 결합을 느슨히 할 수 있다.
  - Invoker 가 Command 를 쌓아두고 어떻게 다루냐에 따라, 요청의 대기, 실행 취소 등을 구현할 수 있다.

구현

0. Receiver - 실질적으로 동작을 수행하는 Class 는 만들어져 있다 가정한다.
1. Command Interface 를 만든다. (excute() method 포함)
2. Concrete Command Class 를 만든다. 이는 Receiver 를 Instance Variable 로 포함하는 등 Receiver (요청을 수신 후 실제 동작을 수행하는 객체) 와 관계를 가지고 있어야 하며, Receiver 내부의 action()을 execute()에서 적절히 실행하도록 execute를 implements 해야한다.
3. Command Interface 를 변수로서 가지는 Invoker Class 를 제작한다.
   Command 를 관리하는 기능들 또한 Invoker Class 가 가질 수 있다.
4. Client Class는 Concrete Command Class 를 만들고, 이를 Invoker Class의 Command Interface Type 의 변수에 넣거나 해준다.

5. Invoker Class 는 이에 따라 Command 를 실행하거나 관리하는 등의 역할을 수행한다. 일반적인 경우, Invoker Class 의 제작 또한 Client Class 가 함께 담당하는 듯하다.

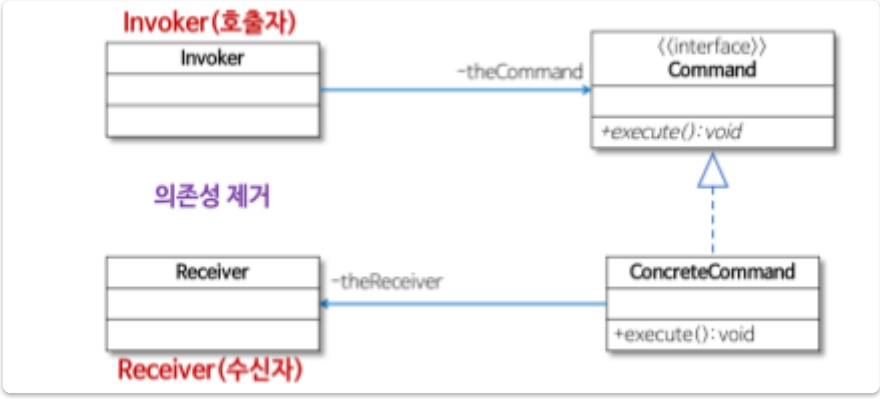
Context & Problem & Merit

- 요청을 객체로 포장하였기 때문에, 오는 장점이 굉장히 많다.
  - Command를 execute 해주는 친구 (Invoker) 와, 그에 따라 실제로 action(동작) 하는 친구 사이의 결합을 느슨히 할 수 있다.
  - Invoker 가 Command 를 쌓아두고 어떻게 다루냐에 따라, 요청의 대기, 실행 취소 등을 구현할 수 있다.

따로 정리하며

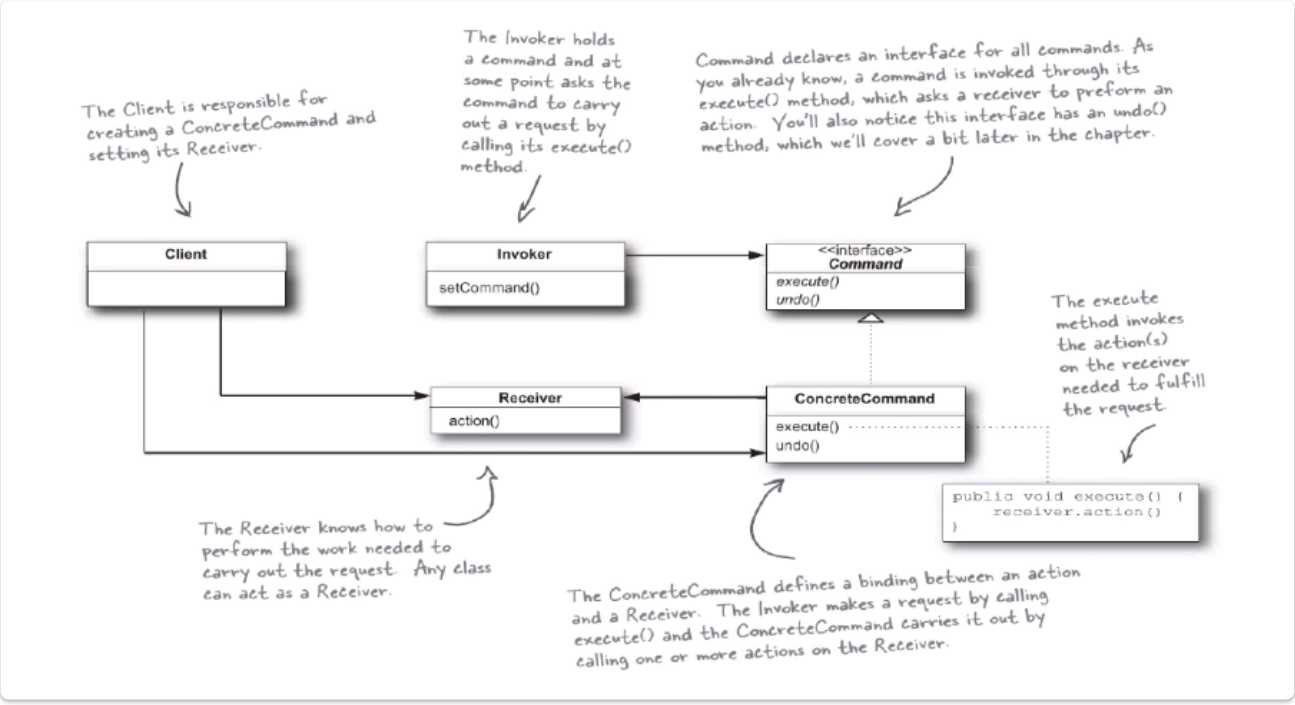
23.08.02 의문

1. Request (요청) 을 객체로 캡슐화 한다는 것 자체를 지금 제대로 이해하지 못했음.
2. 이거 보니까 이해가 되는 것 같기도 하고...?



  - Request = Command 라고 생각하자.
  - Invoker 는 실제로 그 요청이 수행될 지 말지를 정하는 시스템이다. Command interface type 을 변수로서 가지고 있다.
  - Concrete Command 는 Command 의 Concrete Class. 여기에는 "Receiver : 실제로 일을 수행하는 사람"이 들어있고, 수행하는 일에 대한 실제로 구현된 method (implemented execute) 가 들어있다.

3. 어느 정도 이해가 갔으니 이거 살펴보자.



- Command Pattern 의 가장 중점은, "요청(Request)(실행하고자 하는 특정 작업 또는 동작)" 을 객체로 캡슐화 한다는 것에 있다.
- Concrete Command Class 가, 그 요청을 실제로 캡슐화 한 것인데. 캡슐화라고 표현하는 이유는, 해당 동작을 실제로 수행하는 객체인 Receiver 와, 실제로 수행되는 동작인 Receiver 내의 Method ("action()") 가 밖으로 드러나지 않기 때문이다.
  - 밖으로 드러나지 않는 이유는, Concrete Command Class 는 Command Interface 를 통해서만 사용될 것이기 때문이다. 이 Command Interface는 execute() method 만을 가진다.
- Command Interface Type 을 변수로서 가지며, Concrete Command Class 를 해당 변수로 받아와 사용해주는 친구는, Invoker 다.
- Concrete Command Class 를 만들어 주는 친구, 실제로 요청을 제작하는 친구는 Client 이다.

- 요청이 객체로 캡슐화 되는 것이 가장 중점적이며, 캡슐화된 요청이 Command Interface 와 Concrete Command Class다.
  - **요청** : 요청은 실질적인 동작 그 자체는 아닌, "무언가를 해달라고 하는 것.(청하는 것)"이다.
  - **Request 의 의미** : Command Pattern 에서 **"Request" 는 "요구하는 동작 - method (action())"이 담긴 객체**다. 요구하는 동작을 담으려면, 객체지향 프로그래밍에서는 당연히, 그 요구하는 동작을 수행할 수 있는 주체(Receiver)도 포함된다.
  - **Receiver** : 즉, 요청이 요구하는 동작을 실제로 실행하는 Class 를 의미한다.
  - Request 를 **execute** 한다는 것은, 요구하는 동작 (receiver 의 action) 을 수행하는 것을 의미한다.
-

- 더더 정리하면, Requeest=Command 는 "어떠한 동작들을 할 수 있는(Receiver 의 Method : Action1, 2 ... ()) 객체(Receiver)" 에게, "뭘 해달라는 것 적혀있는(execute())" **객체**다.
  - 이렇게 Request 자체가 Command Class 라는 형태로 감싸여져 있다는 것이 중요하다.
- Command 를 추상화 / 캡슐화 하기 위해서 Command Interface 와 Concrete Command Class 로 나눈다.
- 여기에 Command 를 실제로 만드는 Client 와 / Command 들을 받아 관리하고, 실제로 해당 Command 를 실행시켜주는 Invoker가 추가되었다고 생각하면 쉽다.
- 요청을 객체로 포장하였기 때문에, 오는 장점이 굉장히 많다.
  - Command를 execute 해주는 친구 (Invoker) 와, 그에 따라 실제로 action(동작) 하는 친구 사이의 결합을 느슨히 할 수 있다.
  - Invoker 가 Command 를 쌓아두고 어떻게 다루냐에 따라, 요청의 대기, 실행 취소 등을 구현할 수 있다.

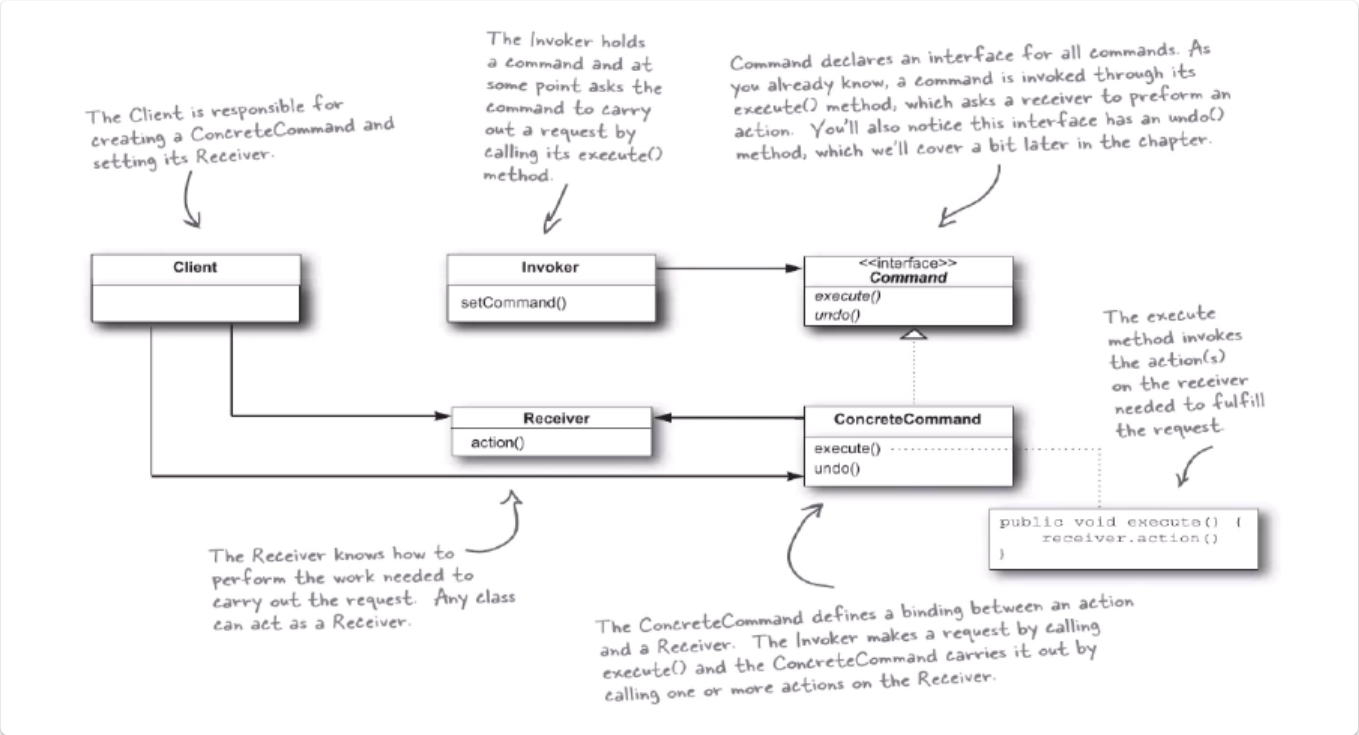
정의

Request (요청) 을 객체로 캡슐화하여 매개변수화, 요청의 대기, 실행 취소 등을 지원하는 패턴. Command 패턴은 요청을 객체로 캡슐화함으로써 요청을 수신하는 객체와 요청을 발송하는 객체 사이의 결합도를 낮추고, 실행 취소, 다시 실행 등의 기능을 제공할 수 있음.

이는 다음과 같은 요소로 구성되어 있음.

- Command Interface : 실행될 동작을 정의하는 인터페이스로, execute() 메서드가 포함되어 있음.
- Concrete Command Class : Command 인터페이스를 구현하여 실제로 동작을 수행하는 클래스. Encapsulate 되어 (Receiver 와, 그 내부 기능은 드러내지 않은 채 만들어 진 후, 이를 실행하는 것 (Excute) 만 밖으로 드러냈으므로, Encapsulate 되었다 볼 수 있을 것.) 독립적으로 존재하는 요청 (Request).
  보통 Receiver Class 를 Instance Variable 로 가지고 있어, 이 method 중 하나를 실행하는 동작을 Execute 에 넣는다.
- Invoker (호출자) 클래스: Command 객체를 사용하여 요청을 실행하는 객체.
- Receiver (수신자) 클래스: 요청을 수신하고 실제 동작을 수행하는 객체. 보통 ConcreteCommand Class 가 이 Receiver Class 와 연결되어 있다.
- Client Class : Command 를 만들어, Invoker 에게 전달해주는 Class 를 의미한다. 확인한 예시에서는, Main 이 이 내용을 가지고 있기도 했다.

구조는 다음과 같다.



구현

0. Receiver - 실질적으로 동작을 수행하는 Class 는 만들어져 있다 가정한다.
1. Command Interface 를 만든다. (excute() method 포함)
2. Concrete Command Class 를 만든다. 이는 Receiver 를 Instance Variable 로 포함하는 등 Receiver (요청을 수신 후 실제 동작을 수행하는 객체) 와 관계를 가지고 있어야 하며, 이를 이용한 execute() 를 실행하도록 override 해야한다.
3. Command Interface 를 변수로서 가지는 Invoker Class 를 제작한다.
   또한 해당 Class는 Client Class 가 Command 를 바꿀 수 있도록 SetCommand() 등,
   Command 를 바꿀 수 있는 내용을 포함하여야 한다.
4. => 여기 부분 부터 조금 다시. Client Method 가 Invoker Class 를 만들고, 이를 넣어가며 사용한다.

강의 들으며

Job / Task - Receiver - 그 Reciever 가 실질적 동작을 수행.
ex) Customer - Waiter (Pass the command) - Chef

Remote Control Project =>
많은 Device, 많은 동일한 동작 in other device
Remote Control Prototype

Create Order -> take Order -> Order Up -> order indicate make some food (indicate 는 많은 걸 가지고 있을 것임.) -> output

Who will do?
Waiter 는 그냥 Order 를 받고, 이행할 뿐임.
=> 잘 모르겠네!

Clinet : Customer
=> Make Command Object (execute())
SetCommand => Invoker
Invoker => Execute method.
Command 의 Execute => Reciver 가 실행.

Command Object Should have reciver.
Remote ControlI 의 예로 들어보면, Remote Controller 가 Invoker 인 것.
Test Program이 Client 임.

Invoker 는 Command 내에 뭐가 들었는지를 알 필요가 없음.
Request 를 Standalone Object (독립적인 object - 각 Concrete Command Object 를 의미) 로 Encapsulate 하고. 우리는 이를 독립적으로 구성했기 때문에, 쉽게 다른 곳으로 넘겨줄 수 있다.

=> 지금 약간 감이 잡힐랑 말랑 하는 느낌이라 한번 정리하고 넘어가는 게 좋을 것 같긴 함.