유니티와 디자인 패턴 2강

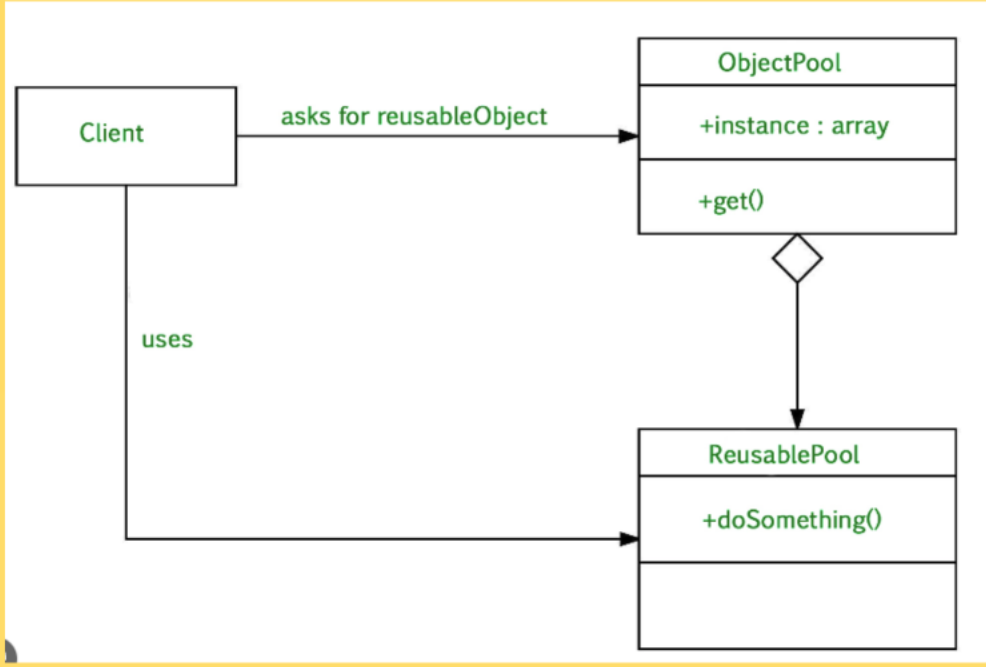
- 김하연 튜터

오브젝트 풀 패턴으로 최적화하기

- 프레임 속도를 유지하려면, 자주 생성되는 요소를 일부 메모리에 예약하는게 좋음
- 최근 죽인 적을 메모리에서 없애는 대신 다시 사용할 수 있도록 오브젝트 풀에 추가
- 엔티티의 새로운 인스턴스를 로드하는 초기의 초기화 비용이 들지 않음
- Unity에는 오브젝트 풀링이 API에 구현되어 있음

오브젝트 풀 패턴

- ObjectPool: 객체의 풀을 관리함. 즉, 객체를 생성하고, 관리하고, 파기하는 책임
- ReusablePool: 실제로 재사용될 객체임. ObjectPool이 여기 객체들을 관리함.
- Client: 클라이언트는 필요할 때 ObjectPool에서 ReusablePool 객체를 요청하고, 사용 후에는 다시 풀에 반환함.



오브젝트 풀 패턴

장점
- 예측할 수 있는 메모리 사용
- 성능 향상

단점
- 이미 C#이 메모리 최적화가 뛰어나서 굳이 필요없다는 썰이 있음
- 예측 불가능한 객체 상태

유니티와 디자인 패턴 2강

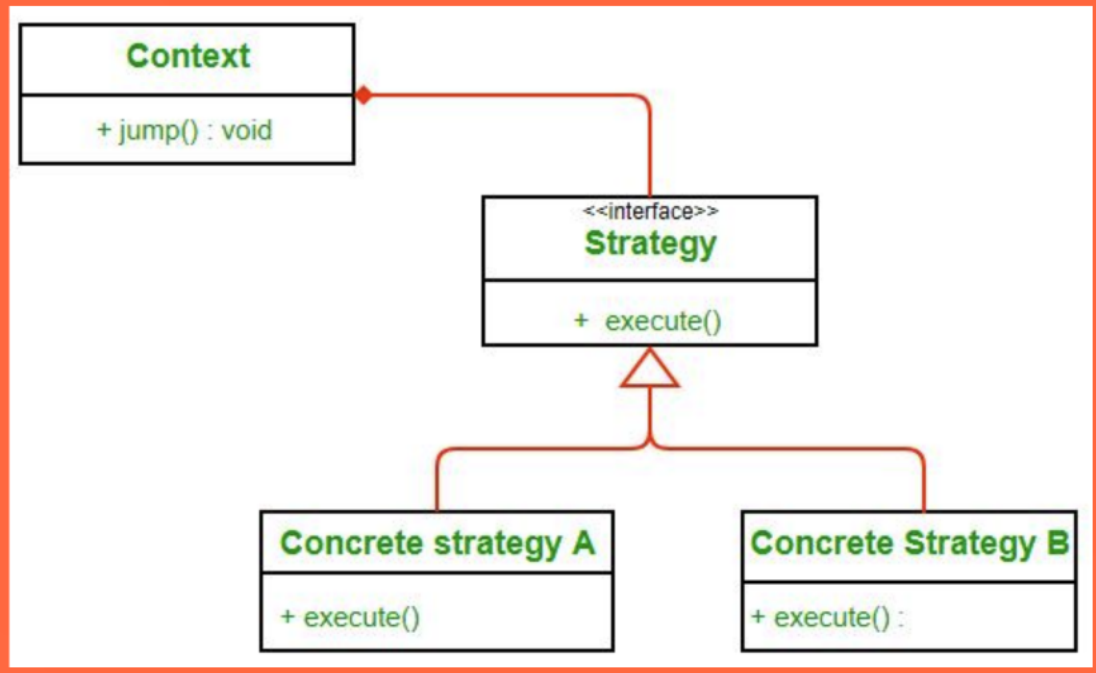
- 김하연 튜터

전략 패턴으로 드론 구현하기

- 드론의 다양한 동작을 구현하는 상황
- 런타임에 특정 동작을 객체에 바로 할당할 수 있음

전략 패턴 개요

- Context: 자신의 작업을 수행하는데 필요한 전략을 선택하는 클래스.
- Strategy: 전략 인터페이스를 구현한 클래스들로, 특정 행동을 제공함
- Client: 클라이언트에서 context 클래스를 생성



전략 패턴 개요

장점
- 캡슐화가 잘 될 수 있음
- 런타임에 객체가 사용하는 알고리즘을 교환할 수 있음.

단점
- 전략 패턴과 상태 패턴이 혼동될 수 있음. 구조가 유사하지만 의도가 매우 다름.
    - 전략 패턴: 같은 문제를 해결하는 여러 알고리즘이 있을 때, 이들 중 하나를 런타임에 선택해야할 때. (ex. 상황에 따라 적합한 정렬 알고리즘 고르기), 즉, **알고리즘의 선택에 중점**
    - 상태 패턴: 객체가 여러 상태를 가지고 있고, 상태에 따라 행동이 달라져야할 때. 즉, **상태에 따른 행동 변경**

유니티와 디자인 패턴 2강

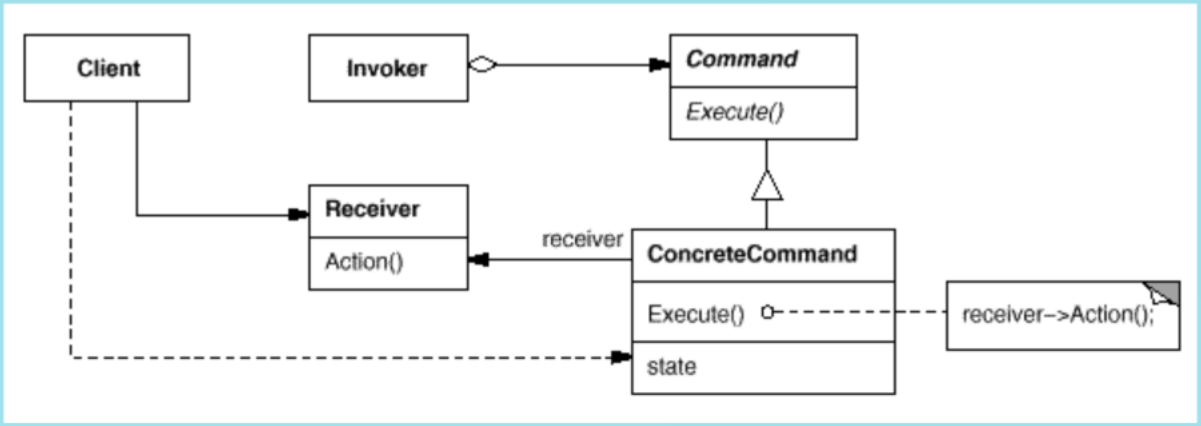
- 김하연 튜터

커맨드 패턴으로 리플레이 시스템 구현하기

- 커맨드 패턴은 게임 내에서 발생하는 모든 행동 (이동, 점프)을 명령으로 캡슐화를 할 수 있음. 그리고 이 명령들은 모두 쉽게 기록이 됨!
- 기록을 재생하여 리플레이 시스템을 구현할 수 있음

커맨드 패턴 개요

- Client: 커맨드 객체를 생성, 그 커맨드가 어떤 receiver와 연결될지를 결정함
- Invoker (호출자): 커맨드를 받아서 실행함.
- Command (커맨드): 실행될 모든 명령에 대한 인터페이스
- Receiver (수신자): 실제로 작업을 수행할 객체.



커맨드 패턴 개요

장점
- 분리: 실행하는 객체와 호출하는 객체가 분리됨
- 명령하는 것을 큐에 넣어서 리플레이, 매크로, 명령 큐 등을 구현할 수 있음.

단점
- 각각의 명령을 하나의 클래스로 구현해야되서 좀 복잡함.

유니티와 디자인 패턴 2강

- 김하연 튜터