1. 팩토리 패턴

1번.txt 파일에는 먼저 제가 맨 처음에 유니티에서 작성한 맵 제작 스크립크가 있습니다 해당 코드에선 게임 시작시 미리 게임매니저에 준비해둔 랜덤 맵 생성 변수를 이용하여 생긴 변수를 이용하여 플레이어와 오브젝트를 그때그때 생성해주는 코드입니다

디자인 패턴을 적용시키기 전의 이 코드에서는 swichcase문으로 하나하나 나눠서 조잡하게 맵 생성을 하였습니다. 그러다보니 이 코드에서 뭔가 내용을 추가하려면 하나하나 손수 추가시켜야 하고 다른 코드에서 조건이 바뀌면 해당 코드의 내용 역시 변화애햐 할 번거로움 역시 존재합니다 그리고 코드에 길고 반복되는 코드가 가득해 가독성 및 효율성 역시 좋지 못합니다

이 코드의 문제점:

1 - 생성 로직이 조잡하다

MakeObject함수에서 맵 번호에 따라서 조건문을 이용하여 프리팹을 생성하는데 이는 굉장히 코드를 복잡하게 만들고 수동적으로 하나하나 코드를 준비시켜야 하기 때문에 생산성도 떨어지는 결과를 이야기합니다.

2 - 많은 중복 코드

해당 스크립트에서는 많은 중복된 코드가 등장합니다 단순히 오브젝트를 생성함에도 지형을 만들 때 한번 캐릭터를 만들 때 한번 배경을 만들 때 한번 등등 계속하여 같은 내용의 생성하는 코드가 등장합니다 이는 코드의 가독성을 감소시키고 유지보수때 실수나 문제를 일으킬 가능성을 높입니다

2번.txt파일은 이를 팩토리 패턴을 적용시켜서 리팩토링한 코드입니다MakeObject에서 생성하는 와정에서 CreatObject라는 함수를를 새로 추가하여 코드의 반복되는 부분을 줄였고 수많은 Instantiate코드를 하나의 return Instantiate을 가진 CreatObject로 바꾸었습니다

이 코드에서 해결된 문제점:

1 - 줄어든 중복 코드

이 스크립트에서는 이전의 스크립트와 달리 많은 중복되던 부분을 새로운 함수 CreatObject라는 함수를 만들어 재사용 함으로써 이를 해결하였습니다 이젠 단 하나의 Instantiate만 가지고도 코드의 해결이 가능합니다

2 - 직관적인 생성 로직

이전의 코드에선 생성부분을 하나하나 코드로 처리하여 굉장히 직관적이지 못했습니다 지형생성이 잘못된거 같아서 확인을 하려 해봐도 코드에서 지형이 생성하는 부분이 어딘지 하나하나 읽어서 찾아봐야 하는 문제점이 있었습니다. 대신 새로 바꾼 코드에선 함수 하나당 한 종류씩 단 4줄의 코드만으로 파악이 가능해져서 손쉽게 디버깅을 할 수 있게됩니다

2. 커맨드 패턴

3번.txt파일의 경우 팀 프로젝트에서 작성한 코드입니다
이전엔 코드가 하나하나 하드코딩이 외어 있어서 프로젝트에서 팀원끼리 진행하는 동안 오류수정 및 기획 수정 과정에서 첨부터 다시 코드를 읽느라 시간을 소모하는 생산성 저하 원인이 있었고 새로운 내용을 추가하거나 삭제할때도 다른 코드랑 충돌이 나지 않는지 하나하나 재확인을 해야했으며 이 때문에 팀원간 회의시간도 지나치게 길어지고 정작 집중해야할건 다른 쪽인데 시간을 제일 많이쓰는건 복잡하게 섞인 코드 확인이었습니다.
이 코드의 문제점:

1 - 코드의 캡슐화가 부족함

동작을 하고 자원을 소비하는 부분의 코드가 플레이어스크립트, ui스크립트에 까지 넓게 펼쳐진 상황이라서 하나 기능을 수정하는데만 몇가지 스크립트 들을 보고 확인해야 했습니다.

2 - 코드의 유연성 및 확장성 부족

코드가 여기저기 꼬여있고 하드코딩이 되어있다 보니 비슷한 역할을 하는 새로운 기능 즉 이동에서 토글하여 달리기 추가라는 간단한 코드를 만드는데도 여기갔다 저기갔다 정신없이 수정을 해야했습니다.

3 - 코드의 유지보수성 저하

1,2번이 중첩되자 이동부분에서 미끄러지는 피드백이 들어왔을 때 저는 하드코딩되고 여기저기 분산된 코드들을 다시 훑어보느라 한참을 걸려서 이동을 간신히 고쳤습니다 이와 같이 코드의 상태가 좋지 않아서 유지보수에 어려움이 있었습니다.

4번 .txt파일의 경우 이를 커맨드 패턴을 이용하여 리팩토링한 코드입니다
새로운 명령객체 PickupItemCommand와 PickupBoxCommand를 이용하여 각 동작들을 캡슐화 하였습니다.
이 코드의 개선된 부분:

1 - PlayerMovement클래스는 실행의 책임만을 갖도록 변경되었음

이전의 PlayerMovement클래스는 수많은 동작들을 직접 처리하고 이동시키고 하는 복잡한 일을 맡아서 update문의 가독성이 감소하였으나 이제는 단순 함수의 실행만을 하기에 책임 역시 간단해졌습니다. 코드의 단일 책임원칙을 지킴으로써 가속성와 유지보수가 더 쉬워집니다

2 - 코드의 유지보수가 쉬워짐

이 코드에서 상자와 관련된 기능을 손본다고 하더라도 해당 부분의 역할을 하는 함수를 찾아가서 바꾸는 간단한 방식으로 수정이 가능해 졌습니다. 역할이 부분부분으로 잘 분담된 덕에 내가 원하는게 있다면 해당 함수의 위치로 이동하여 손쉽게 기능을 수정할 수 있게 되었습니다.

3. 옵저버 패턴

5번.txt파일의 경우 새롭게 만든 ui코드입니다

총 4개의 스크립트가 존재하고 StaminaSubject, StaminaObserver, Health Subject, HpObserver 가 존재합니다

이 코드들은 스태미너, 체력의 상태 변화를 감지해 스태미너 바, 체력의 시각효과를 최신화 합니다 그리고 다른 기능을 쉽게 추가할 수 있도록 옵저버 패턴을 적용한 상태입니다

이 코드의 장점:

1 - 서브젝트와 옵저버의 분리

서브젝트 클래스들은 스태미너의 상태를 관리하고 옵저버 클래스들은 스태미너의 상태 변경을 감지하고 처리하는 역할을 수행합니다. 이러한 코드들로 인해 코드의 응집성이 개선되고 변경에 대한 영향이 줄어듭니다.

2 - 유연성과 확장성

새로운 옵저버를 추가하여 새로운 기능을 추가하면서도 기존의 옵저버를 수정할 필요 없이 새롭게 제작이 가능합니다.

3 - 재사용성

다른 상황 다른 환경에서도 다른 객체나 상태 변경에 따른 옵저버를 구현할 수 있습니다 스태미너가 아니라 스킬의 충전바나 경험치바 등등에 쉽게 응용이 가능합니다.

4. 스테이트 패턴

6번.txt파일의 경우 슈팅게임에서 동전의 코드입니다

이전의 코드에서는 상태 변환에 명확한 구조가 없었고 로직이 중복되어있거나 캡슐화가 제대로 되어있지 않아서 동작은 하지만 수정 및 유지보수에 큰 문제가 있는 상황이었습니다.

이 코드의 문제점

1 - 상태관리의 불일치

이 코드에선 돈이 플레이어에게 도착하기 전 랜덤한 방향으로 튀어나가는 경우가 있는데 이 상태 전환에 대한 로직이 명확한 구조로 되어있지도 않고 여러개의 함수로 나뉘어져 있었습니다.

2- 코드 중복과 가독성

코드들이 생각나는대로 대충 짜여져 있으며 일부 로직도 중복되어 있으며 가독성 역시 좋지가 않았습니다 튕겨나가는 로직과 도착 후 정지하는 로직이 여기저기 중복되어 사용되어 있엇고 이 때문에 가독성 역시 떨어집니다.

7번 .txt파일은 이를 수정하여 스테이트 패턴이라는 상태관리를 이용한 디자인 패턴을 적용하였습니다. 기존에는 if문과 bool변수로 직접 관리하던 반면 새롭게 바뀐 코드는 MoneyState라는 추상 클래스를 도입하고, 이를 상속받는 ReadyState와

GoToPlayerState 클래스를 구현하여 각각의 상태에 따른 동작을 캡슐화했습니다
두 번째 코드는 상태 전환 로직을 보다 명확하게 표현하고 중복된 코드를 제거하여 가독성을 올리고 유지보수에 쉽게 만들었습니다.
이 코드의 개선된 부분:

1 - 명확한 상태 관리

상태에 대한 구분 없이 대충 짠 코드로 모여있던 전과는 달리 상태 구분을 명확히 할 수 있도록 state개념을 도입하여 문제를 해결하였습니다.

2 - 중복된 코드 해결

여기저기 흩어진 중복된 코드들을 한곳에 몰아서 상태에 따라 로직을 이행하도록 변경하여 가독성을 개선하였습니다.

3 - 유지 보수성

새로운 코드를 추가하거나 상태 전환 로직을 수정할 때 해당 상태에 대한 클래스만 수정하면 되므로 다른 부분에 영향을 주지 않고 유연하게 확장 할 수 있습니다