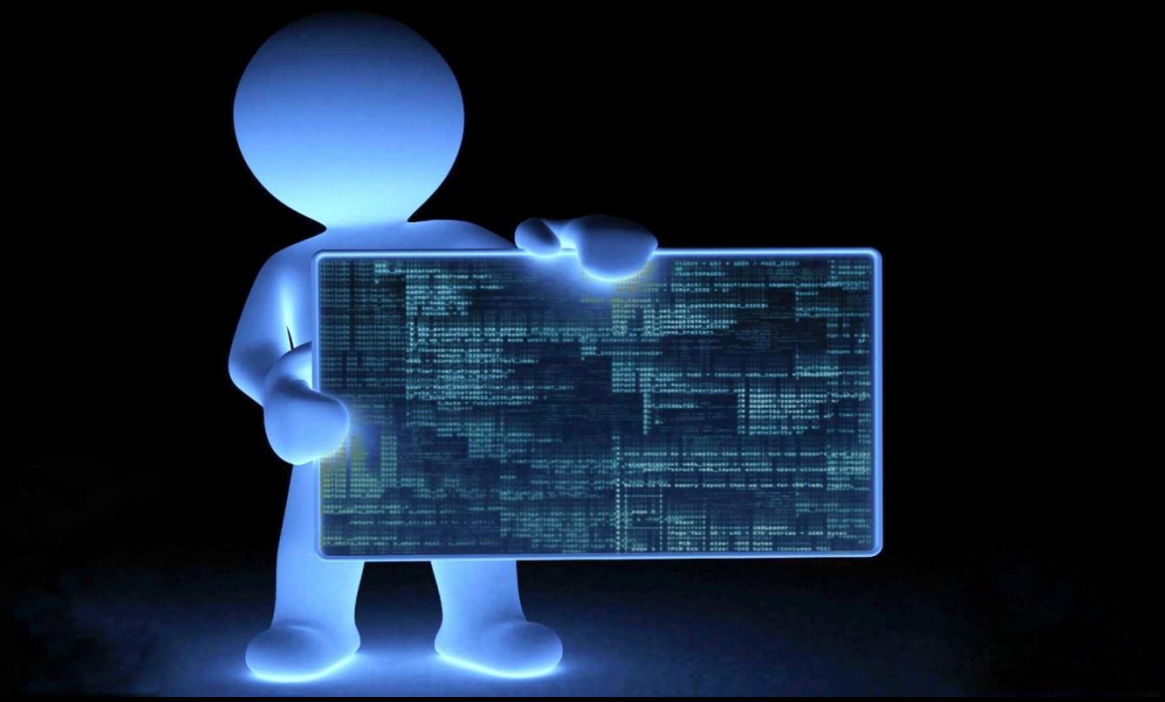


텀프로젝트 발표

객체지향개발론

20141361129 최은빈

Content

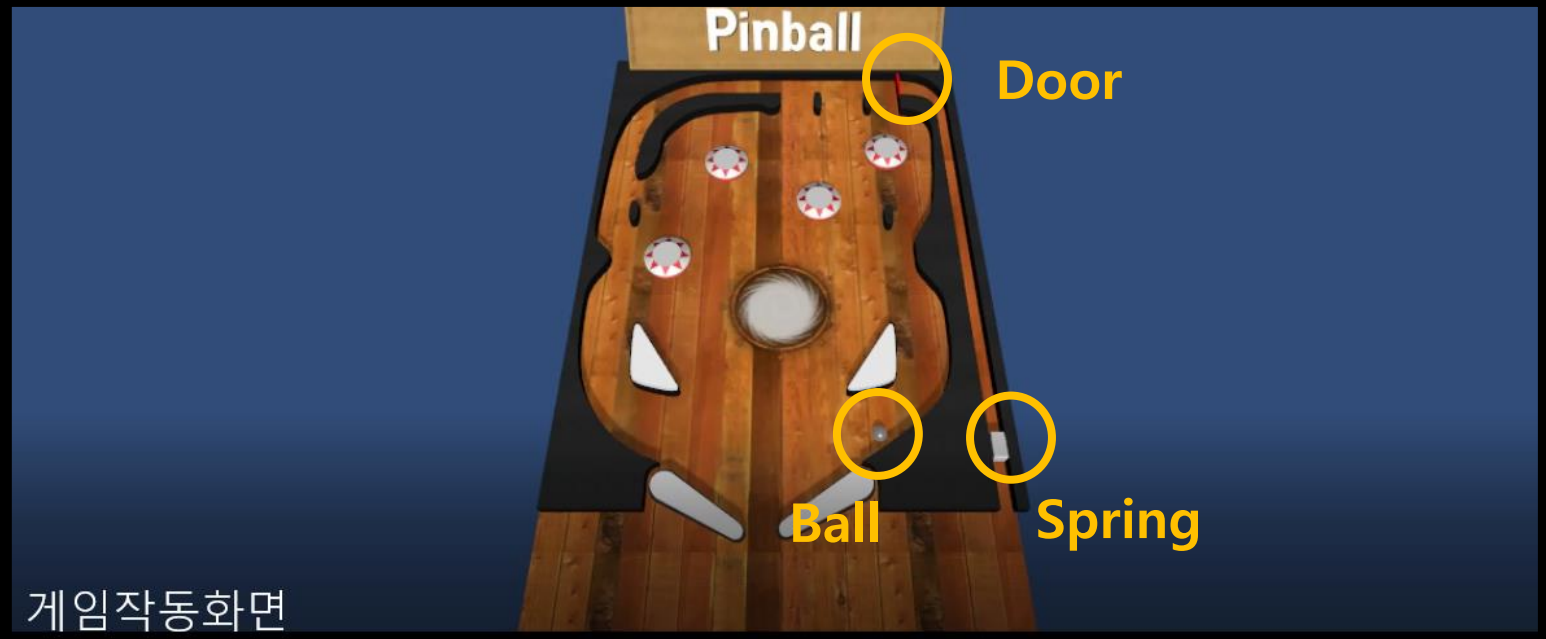
Explain the content we are treated

- 프로젝트 소개
- 적용한 설계 패턴 소개
- 패턴의 장단점

프로젝트 소개

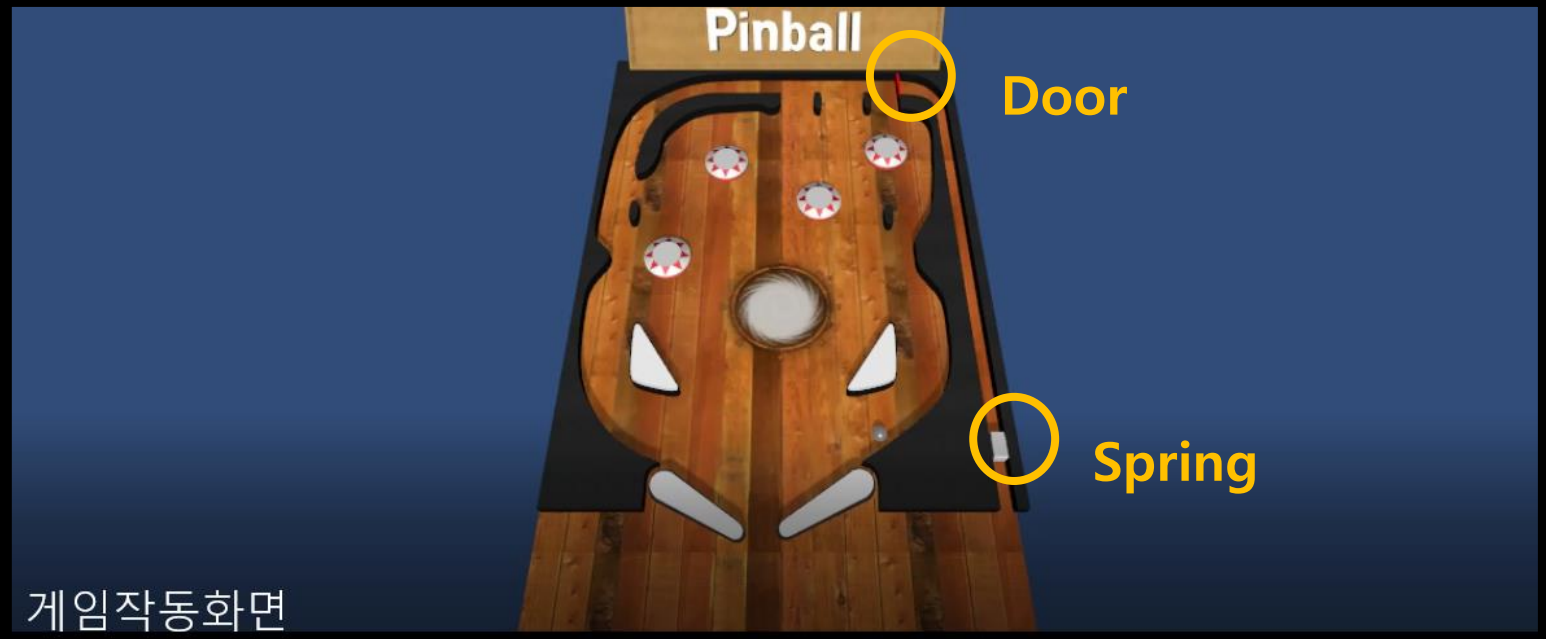
프로젝트 소개

프로젝트 소개



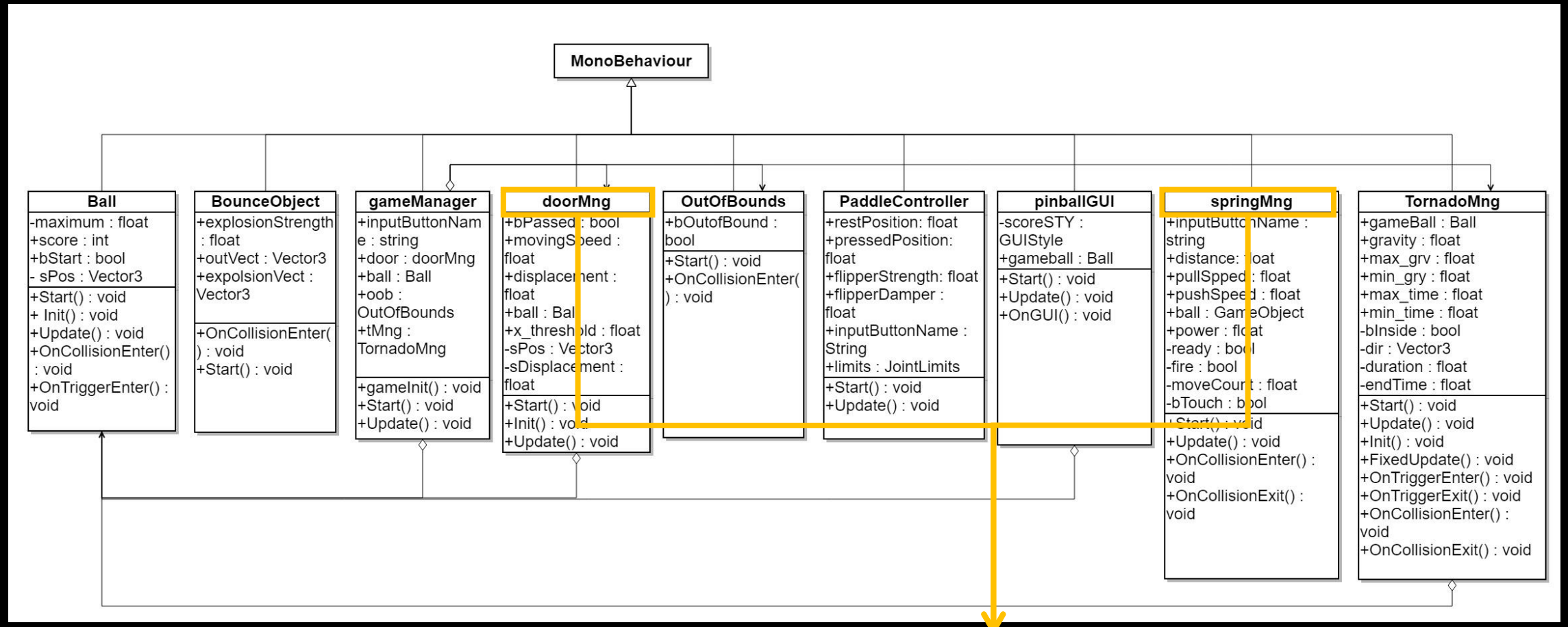
적용한 설계 패턴 소개

① State



적용한 설계 패턴 소개

① State



State 패턴 구현

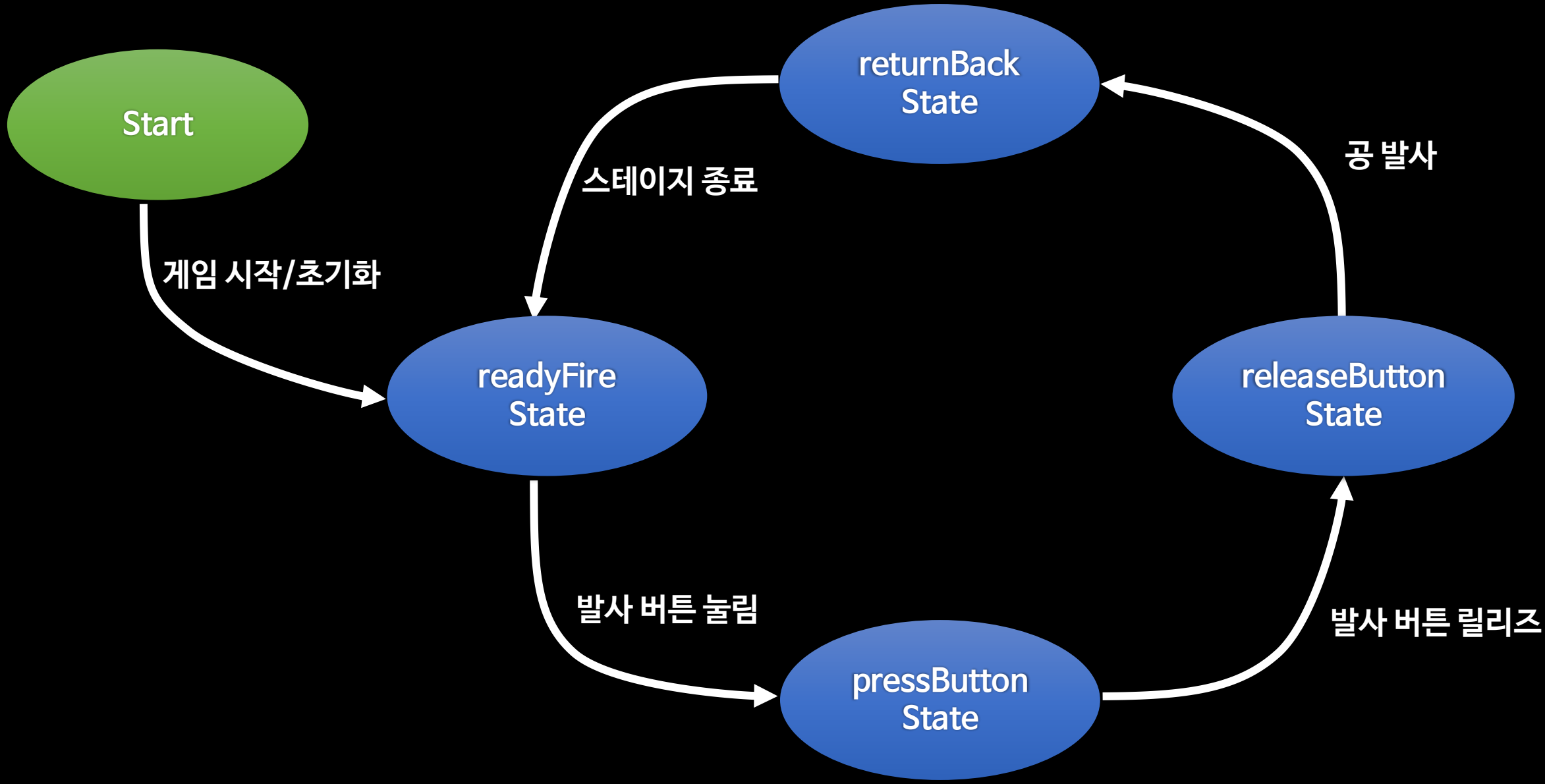
적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현

State 패턴 전 ←

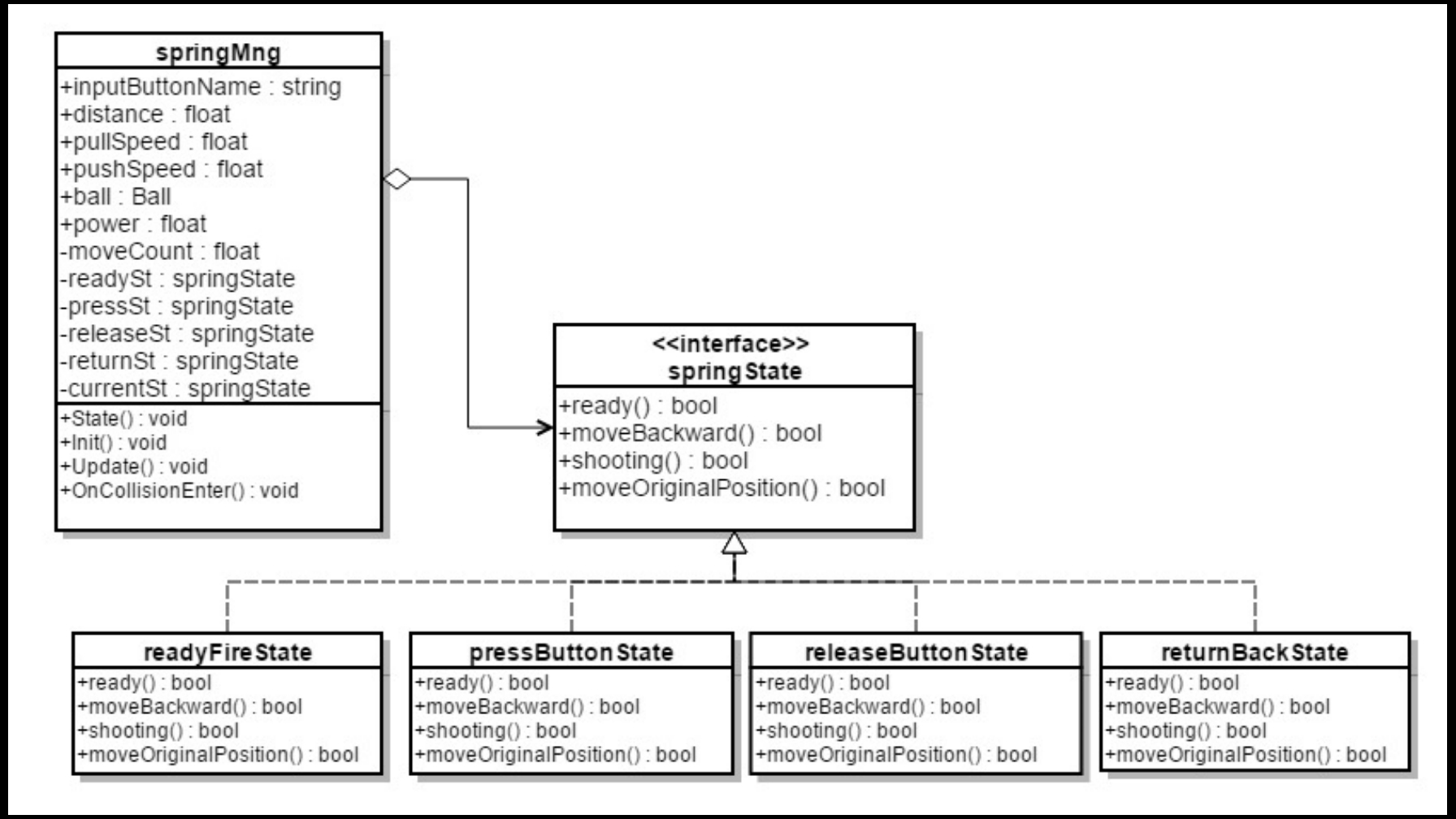
```
// Update is called once per frame
public void Update ()
{
    if(Input.GetButton(inputButtonName)){   //키보드 입력이나 터치 입력이 되었을 경우
        if(moveCount < distance){                   //이동 도달 거리 보다 현재 거리가 작으면
            transform.Translate(0,0,-pullSpeed * Time.deltaTime);   //이동 명령
            moveCount += pullSpeed * Time.deltaTime;                //이동 정도 저장
            fire = true;                                            //발사 준비 상태
        }
    }
    else if(moveCount > 0){
        if(fire && ready){  //발사 준비가 끝났다면,
            ball.transform.TransformDirection(Vector3.forward * 50);    //볼 움직이
            ball.GetComponent<Rigidbody>().AddForce(0, 0, moveCount * power);   //볼에 파워 인가

            fire = false;
            ready = false;
        }

        //초기 위치로 수렴하기 위해
        if(moveCount < (pushSpeed * Time.deltaTime))
        {
            transform.Translate(0,0,moveCount); //이동
            moveCount = 0;
            fire = false; //fire
        }
        else{   //빠른 속도로 발사
            transform.Translate(0,0,pushSpeed * Time.deltaTime);
            moveCount -= pushSpeed * Time.deltaTime;
        }
    }
}
public void OnCollisionEnter(Collision collision) {
    if(collision.gameObject.tag == "Ball"){     //볼과 충돌 했다면 준비 상태
        ready = true;
    }
}
```

적용한 설계 패턴 소개


Start
게임 시작/초기화
readyFire State
발사 버튼 눌림
pressButton State
발사 버튼 릴리즈
releaseButton State
공 발사
returnBack State
스테이지 종료

적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현



적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현

```
public interface springState
{
    bool ready();
    bool moveBackward();
    bool shooting();
    bool moveOriginalPosition();
}
```

적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현

```
public class readyFireState : springState //발사를 기다리는 상태
{
    public bool ready()
    {
        Debug.Log("Ready");
        return true;
    }
    public bool moveBackward()
    {
        return false;
    }
    public bool shooting()
    {
        return false;
    }
    public bool moveOriginalPosition()
    {
        return false;
    }
}
```

```
public class pressButtonState : springState //발사 키가 눌린 상태
{
    public bool ready()
    {
        return false;
    }
    public bool moveBackward()
    {
        return true;
    }
    public bool shooting()
    {
        return false;
    }
    public bool moveOriginalPosition()
    {
        return false;
    }
}
```

적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현

```
public class releaseButtonState : springState  //발사 키가 릴리즈 된 상태
{
    public bool ready()
    {
        return false;
    }
    public bool moveBackward()
    {
        return false;
    }
    public bool shooting()
    {
        return true;
    }
    public bool moveOriginalPosition()
    {
        return false;
    }
}
```

```
public class returnBackState : springState  //발사대를 원래 위치로 복귀
{
    public bool ready()
    {
        return false;
    }
    public bool moveBackward()
    {
        return false;
    }
    public bool shooting()
    {
        return false;
    }
    public bool moveOriginalPosition()
    {
        return true;
    }
}
```

적용한 설계 패턴 소개

① State - springMng 클래스 상태 구현

State 패턴 후

```
void Update ()
{
    if (Input.GetButton(inputButtonName))
    {   //키보드 입력이 되었을 경우
        if (currentSt.ready())          //레디 상태였다면,
            currentSt = pressSt;        //키가 눌린 상태로 전환
    }
    else {          //키가 안눌린 경우
        if(currentSt.moveBackward())    //키가 눌렸던 상태 였다면
            currentSt = releaseSt;      //키가 릴리즈 된 상태로 전환
    }

    if (currentSt.moveBackward())   //발사대를 뒤로 당겨야 하는 상황이면 (키가 눌려 있는 상태)
    {
        if (moveCount < distance)
        {                   //이동 도달 최대 거리 보다 현재 거리가 작으면
            transform.Translate(0, 0, -pullSpeed * Time.deltaTime);   //이동 명령
            moveCount += pullSpeed * Time.deltaTime;                //이동 정도 저장
        }
    }
    else if (currentSt.shooting())  //발사를 해야하는 상황이라면 (키가 릴리즈 된 상태),
    {
        ball.transform.TransformDirection(Vector3.forward * 50);    //볼 이동
        ball.GetComponent<Rigidbody>().AddForce(0, 0, moveCount * power);   //볼에 파워 인가
        currentSt = returnSt;
    }
    else if (currentSt.moveOriginalPosition()) {
        if (moveCount < (pushSpeed * Time.deltaTime))
        {
            transform.Translate(0, 0, moveCount); //이동
            moveCount = 0;
            currentSt = readySt;
        }
        else
        {   //빠른 속도로 발사
            transform.Translate(0, 0, pushSpeed * Time.deltaTime);
            moveCount -= pushSpeed * Time.deltaTime;
        }

    }
}
void OnCollisionEnter(Collision collision) {
    if(collision.gameObject.tag == "Ball"){     //볼과 충돌 했다면 준비 상태
        currentSt = readySt;
    }
```

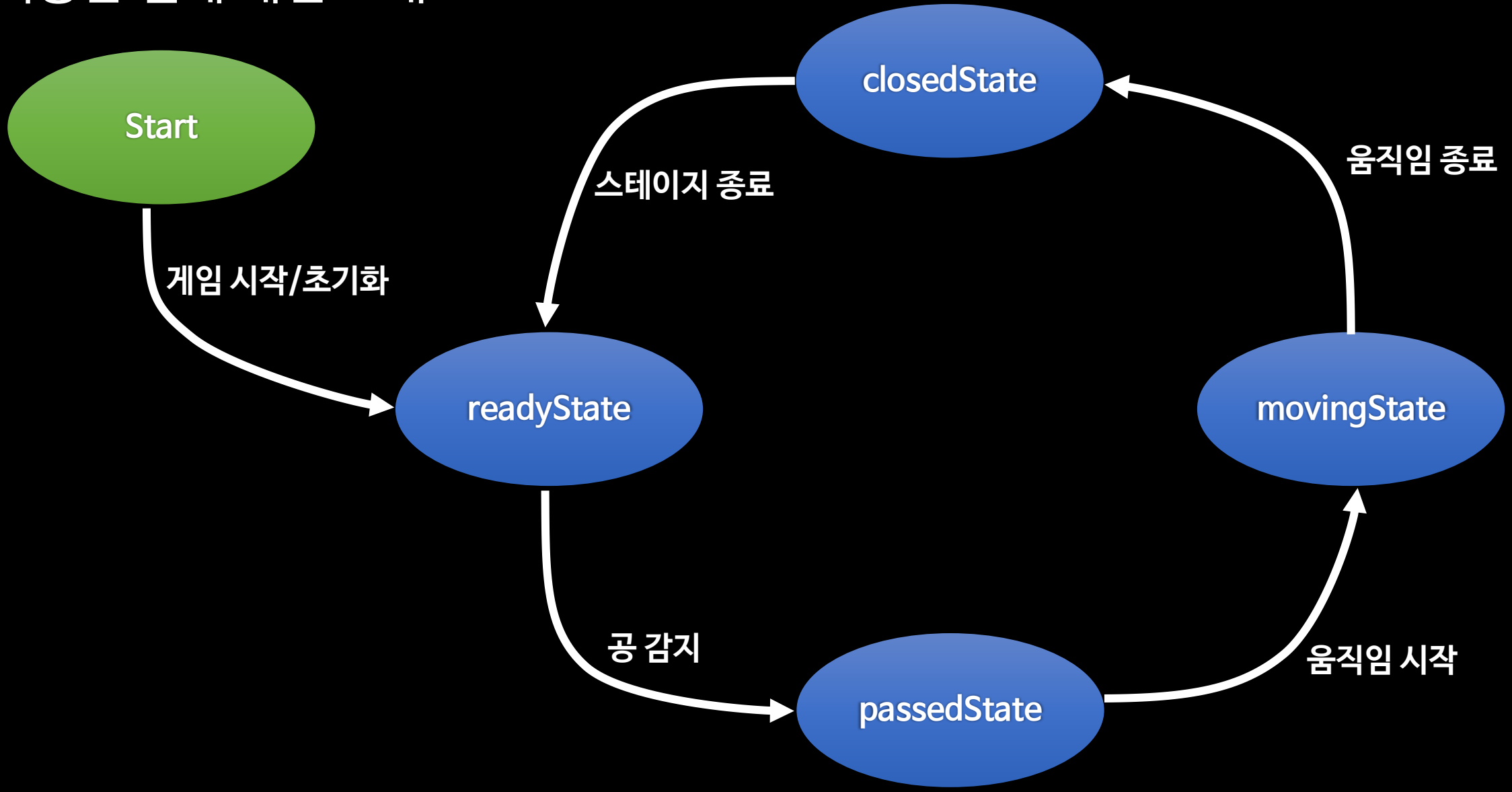
적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현

State 패턴 전 ←

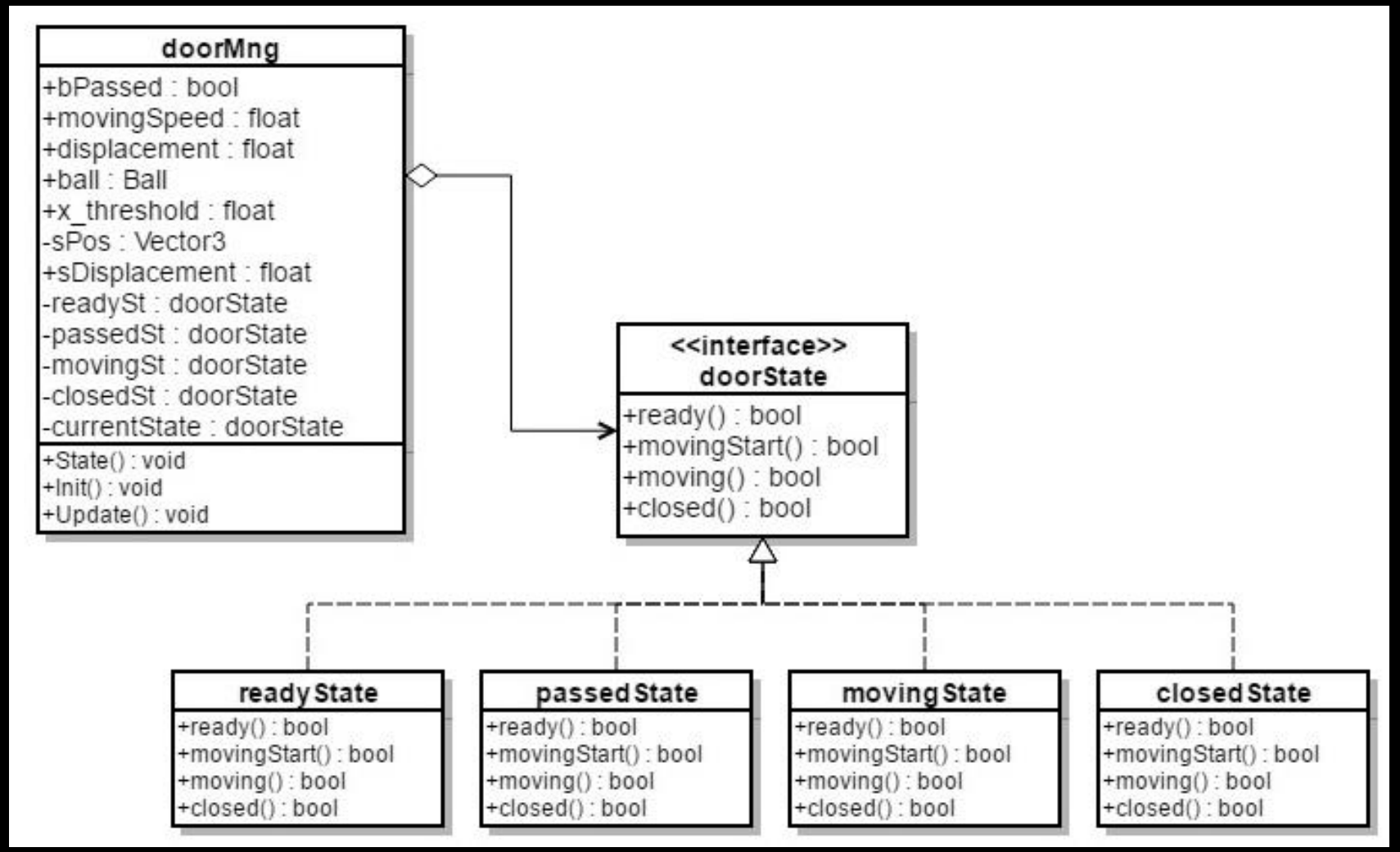
```
public void Start()
{
    sPos = this.transform.position;     //초기 시작 위치 저장
    sDisplacement = displacement;       //초기 시작 변위 저장
}

public void Init()
{
    this.transform.position = sPos;     //초기 시작 위치로 초기화
    this.bPassed = false;               //공이 지나 갔는지 여부를 저장하는 변수를 초기화
    this.displacement = sDisplacement;  //초기 시작 변위 저장
}

public void Update()
{
    if (ball.transform.position.x < x_threshold)
    {  //볼의 위치가 임계치 이내로 지나 갔는지 확인.
        ball.bStart = true;         //볼의 시작 변수를 참으로 초기화
        bPassed = true;             //공이 지나 갔다는 정보를 저장
    }
    if (bPassed)
    {                       //공이 이미지나 갔다면
        if (displacement < (movingSpeed * Time.deltaTime))
        {    //특정 위치에 도달하지 못한 경우
            transform.Translate(0, 0, displacement);       //이동
            displacement = 0;
            bPassed = false;
        }
        else if (displacement == 0)
        {
            displacement = 0;
            bPassed = false;
        }
        else
        {
            transform.Translate(0, 0, movingSpeed * Time.deltaTime);
            displacement -= movingSpeed * Time.deltaTime;
        }
    }
    //Debug.DrawRay(this.transform.position,this.transform.forward*10, Color.red, 50f,false);
}
```

적용한 설계 패턴 소개


Start
게임 시작/초기화
readyState
공 감지
passedState
움직임 시작
movingState
움직임 종료
closedState
스테이지 종료

적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현



적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현

```
public interface doorState
{
    bool ready();
    bool movingStart();
    bool moving();
    bool closed();
}
```

적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현

```
public class readyState : doorState //공이 지나가길 기다리는 상태
{
    public bool ready()
    {
        return true;
    }
    public bool movingStart()
    {
        return false;
    }
    public bool moving()
    {
        return false;
    }
    public bool closed()
    {
        return false;
    }
}
```

```
public class passedState : doorState //공이 지나간 상태
{
    public bool ready()
    {
        return false;
    }
    public bool movingStart()
    {
        return true;
    }
    public bool moving()
    {
        return false;
    }
    public bool closed()
    {
        return false;
    }
}
```

적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현

```
public class movingState : doorState //공이 지나가고 문이 움직이고 있는 상태
{
    public bool ready()
    {
        return false;
    }
    public bool movingStart()
    {
        return false;
    }
    public bool moving()
    {
        return true;
    }
    public bool closed()
    {
        return false;
    }
}
```

```
public class closedState : doorState //문이 다 닫힌 상태
{
    public bool ready()
    {
        return false;
    }
    public bool movingStart()
    {
        return false;
    }
    public bool moving()
    {
        return false;
    }
    public bool closed()
    {
        return true;
    }
}
```

적용한 설계 패턴 소개

① State - doorMng 클래스 상태 구현

State 패턴 후

```
void Start()
{
    currentState = readySt;
    if (currentState.ready())
    {
        sPos = this.transform.position;     //초기 시작 위치 저장
        sDisplacement = displacement;       //초기 시작 변위 저장
    }
}

public void Init()
{
    this.transform.position = sPos;      //초기 시작 위치로 초기화
    this.displacement = sDisplacement;   //초기 시작 변위 저장
    currentState = readySt;              //대기 상태로 초기화
}
```

```
void Update()
{
    if (currentState.ready())       //레디 상태이고,
    {
        if (ball.transform.position.x < x_threshold)
        {  //볼의 위치가 임계치 이내로 지나 갔는지 확인.
            ball.bStart = true;                 //볼의 시작 변수를 참으로 초기화
            currentState = passedSt;            //움직임 확인 상태로 전환
        }
    }
    if (currentState.movingStart())         //움직임이 시작 됐다면,
        currentState = movingSt;            //자동으로 움직임 상태로 전환

    if (currentState.moving()) //움직임 상태면
    {                                   //공이 이미지나 갔다면
        if (displacement < (movingSpeed * Time.deltaTime))
        {    //특정 위치에 도달하지 못한 경우
            transform.Translate(0, 0, displacement);        //이동
            displacement = 0;
            currentState = closedSt;    //움직임이 끝났기 때문에 닫힘 상태로 전환
        }
        else if (displacement == 0)
        {
            displacement = 0;
            currentState = closedSt;    //움직임이 끝났기 때문에 닫힘 상태로 전환
        }
        else
        {
            transform.Translate(0, 0, movingSpeed * Time.deltaTime);
            displacement -= movingSpeed * Time.deltaTime;
        }
    }
    //Debug.DrawRay(this.transform.position,this.transform.forward*10, Color.red, 50f,false);
}
```

패턴의 장단점

① State

상태 패턴의 장점

1. 새 자식클래스를 정의하여 새로운 상태를 쉽게 추가 가능

2. 각 상태를 클래스를 이용해 표현하기 때문에
상태 전이를 한눈에 파악할 수 있음

패턴의 장단점

① State

상태 패턴의 단점

1. 클래스의 개수가 많아짐

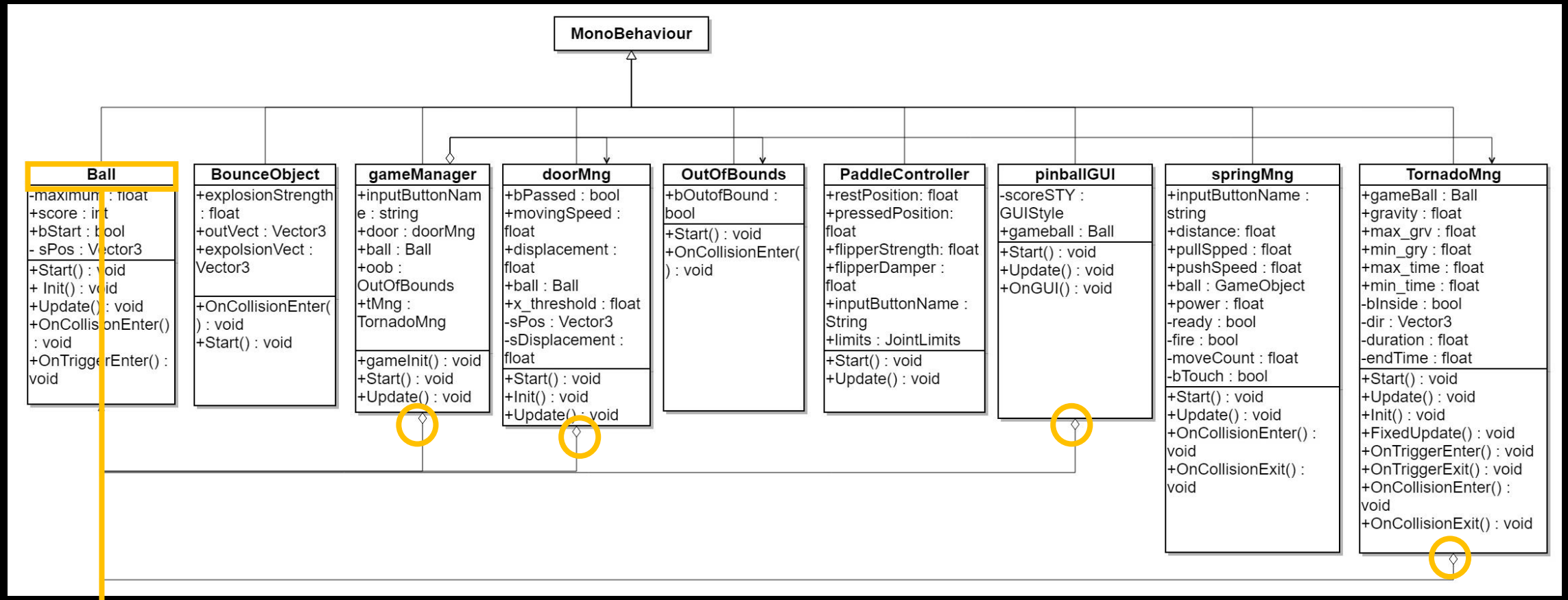
2. 클래스에 코드 중복이 생길 수 있음

적용한 설계 패턴 소개

② Singleton

적용한 설계 패턴 소개

② Singleton



→ Singleton으로 구현

적용한 설계 패턴 소개

② Singleton

```
private static Ball _instance = null;

public static Ball getInstance() {
    return _instance;
}
public void setTrans(Vector3 t) {
    this.transform.TransformDirection(t);
}
public void setAddForce(float z)
{
    this.GetComponent<Rigidbody>().AddForce(0, 0, z);
}

//Awake is always called before any Start functions
void Awake()
{
    //Check if instance already exists
    if (_instance == null)
    {
        //if not, set instance to this
        _instance = this;
    }
    //If instance already exists and it's not this:
    else if (_instance != this)
    {
        //Then destroy this. This enforces our singleton pattern, meaning there can only ever be o
        Debug.Log("Destroy!!");
        Destroy(gameObject);
    }

    //Sets this to not be destroyed when reloading scene
    DontDestroyOnLoad(gameObject);
}
```

객체 생성 ← 현재 객체가 존재하지 않는 경우

Destroy ← 현재 객체가 존재하는 경우

적용한 설계 패턴 소개

② Singleton

gameManager 클래스

```
//게임 매니저
public class gameManager : MonoBehaviour
{
        public string inputButtonName = "Reset";    //리셋 버튼 이름 저장
        public movingDoor door;          //도어 객체 생성
        //public Ball ball;                //볼 객체 생성
        public OutOfBounds oob ;         //외곽 판정 객체 생성
        public TornadoMng tMng;          //토네이도 객체 생성

        // Use this for initialization
        public void gameInit (){
            Ball.getInstance().Init();    //볼 객체 초기화
            door.Init();
            tMng.Init ();   //토네이도 객체 초기화
            oob.bOutofBound = false;    //공위치가 외곽 위치에 있는지 판별하는 변수 초기화
        }
        public void Start ()
        {

        }
```

doorMng 클래스

```
void Update()
{
    if (Ball.getInstance().transform.position.x < x_threshold)
    {  //볼의 위치가 임계치 이내로 지나 갔는지 확인.
        Ball.getInstance().bStart = true;        //볼의 시작 변수를 참으로 초기화
        bPassed = true;             //공이 지나 갔다는 정보를 저장
    }
    if (bPassed)
    {                       //공이 이미지나 갔다면
        if (displacement < (movingSpeed * Time.deltaTime))
        {    //특정 위치에 도달하지 못한 경우
            transform.Translate(0, 0, displacement);       //이동
            displacement = 0;
            bPassed = false;
        }
        else if (displacement == 0)
        {
            displacement = 0;
            bPassed = false;
        }
        else
        {
            transform.Translate(0, 0, movingSpeed * Time.deltaTime);
            displacement -= movingSpeed * Time.deltaTime;
        }
    }
```

적용한 설계 패턴 소개

② Singleton

pinballGUI 클래스

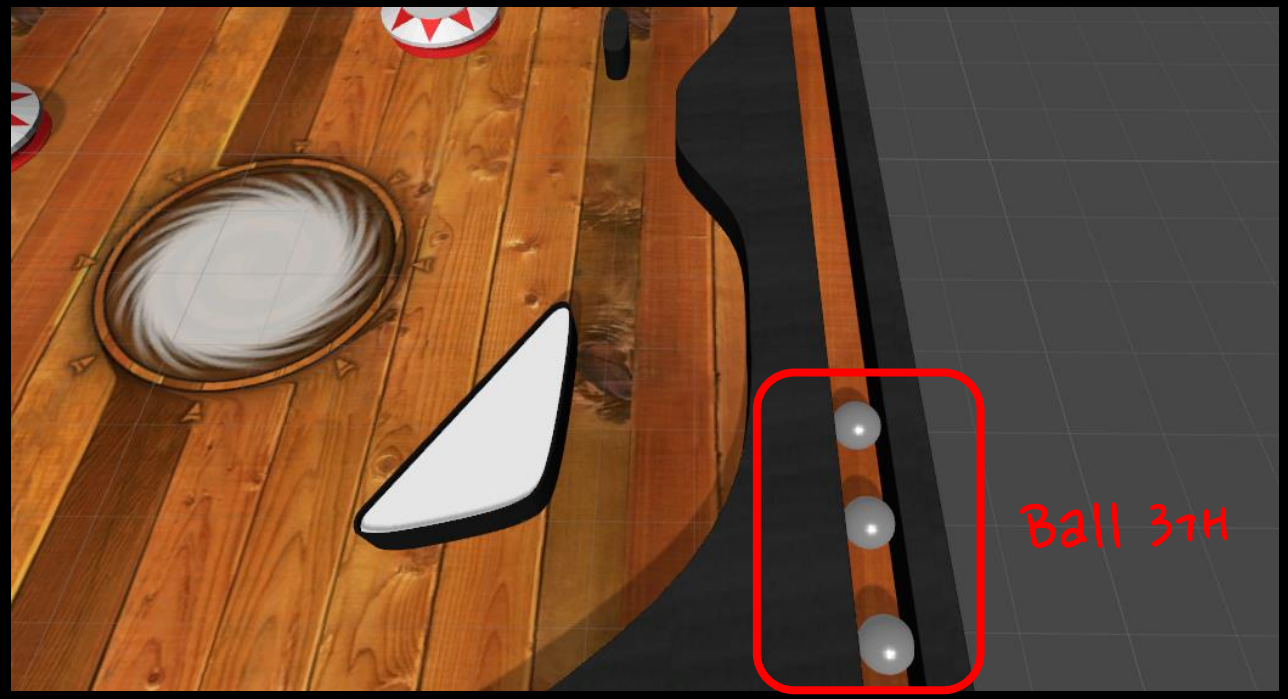
```
//간단한 GUI 클래스
public class pinballGUI : MonoBehaviour {

    private GUIStyle scoreSTY = new GUIStyle(); //GUIStyle 객체 생성
    //public Ball gameball;   //볼 객체
    // Use this for initialization
    void Start () {
        scoreSTY.alignment = TextAnchor.MiddleCenter;   //가운데 정렬
        scoreSTY.fontSize = 20;                         //폰트 크기 20
        //scoreSTY.c
    }

    // Update is called once per frame
    void Update () {

    }
    void OnGUI() {
        GUI.color = Color.white;        //흰색
        GUI.TextField (new Rect (10, 10, 150, 20), "Score : " + Ball.getInstance().score) ;

    }
```

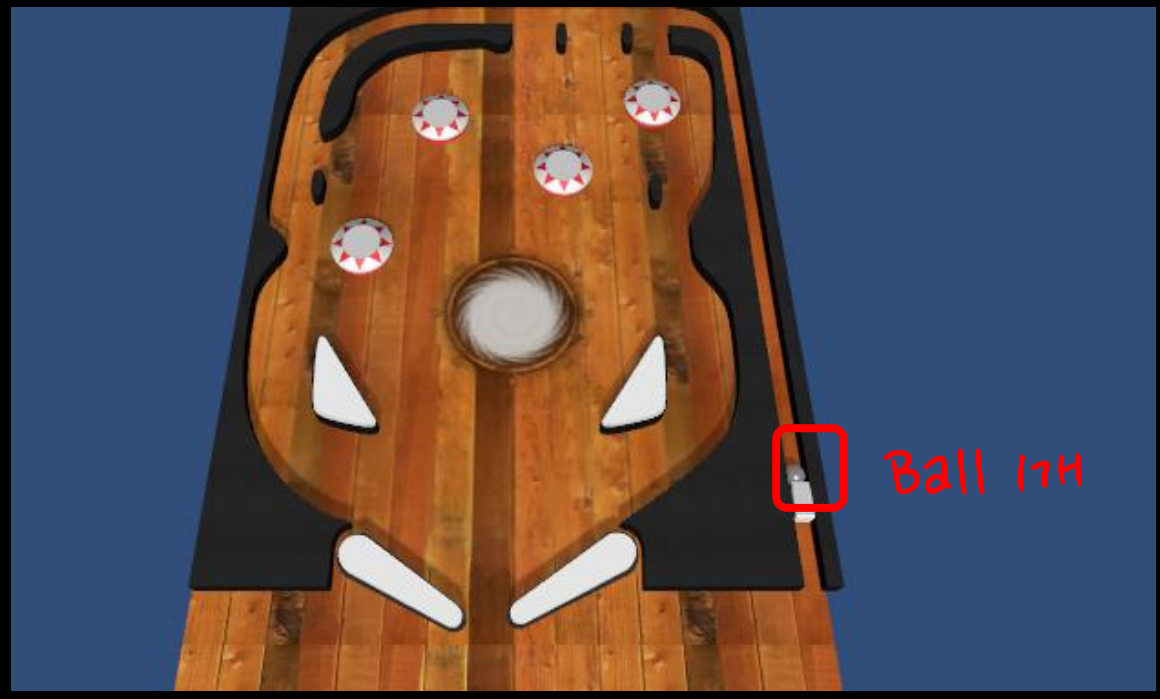
SpringMng 클래스

```
        else if(moveCount > 0){
            //Shoot the ball
            //Debug.Log(ball.transform.position);
            if(fire && ready){  //발사 준비가 끝났다면,

                Ball.getInstance().setTrans(Vector3.forward * 50);
                Ball.getInstance().setAddForce( moveCount * power);   //볼에 파워 인가

                fire = false;
                ready = false;
            }

            //초기 위치로 수렴하기 위해
            if(moveCount < (pushSpeed * Time.deltaTime))
            {
                transform.Translate(0,0,moveCount); //이동
                moveCount = 0;
                fire = false; //fire
            }
            else{   //빠른 속도로 발사
                transform.Translate(0,0,pushSpeed * Time.deltaTime);
                moveCount -= pushSpeed * Time.deltaTime;
            }
        }
    }
    void OnCollisionEnter(Collision collision) {

        if(collision.gameObject.tag == Ball.getInstance().tag) // 살아남는 놈
        {     //볼과 충돌 했다면 준비 상태
            ready = true;
        }
    }
    void OnCollisionExit(Collision collisionInfo) {


        if (collisionInfo.gameObject.tag == Ball.getInstance().tag)
        { //볼과 충돌이 끝났다면 준비 상태를 거짓으로 초기화
            ready = false;
```

적용한 설계 패턴 소개

② Singleton





콘솔 화면

Destroy!!
UnityEngine.Debug:Log(Object)

Destroy!!
UnityEngine.Debug:Log(Object)

패턴의 장단점

② Singleton

싱글톤의 장점

1. 객체가 빈번히 생성되는 것을 막을 수 있음
   - 이는 결국 메모리 공간 절약으로 이어짐

2. 객체 초기화 로직을 반복할 필요가 없음

패턴의 장단점

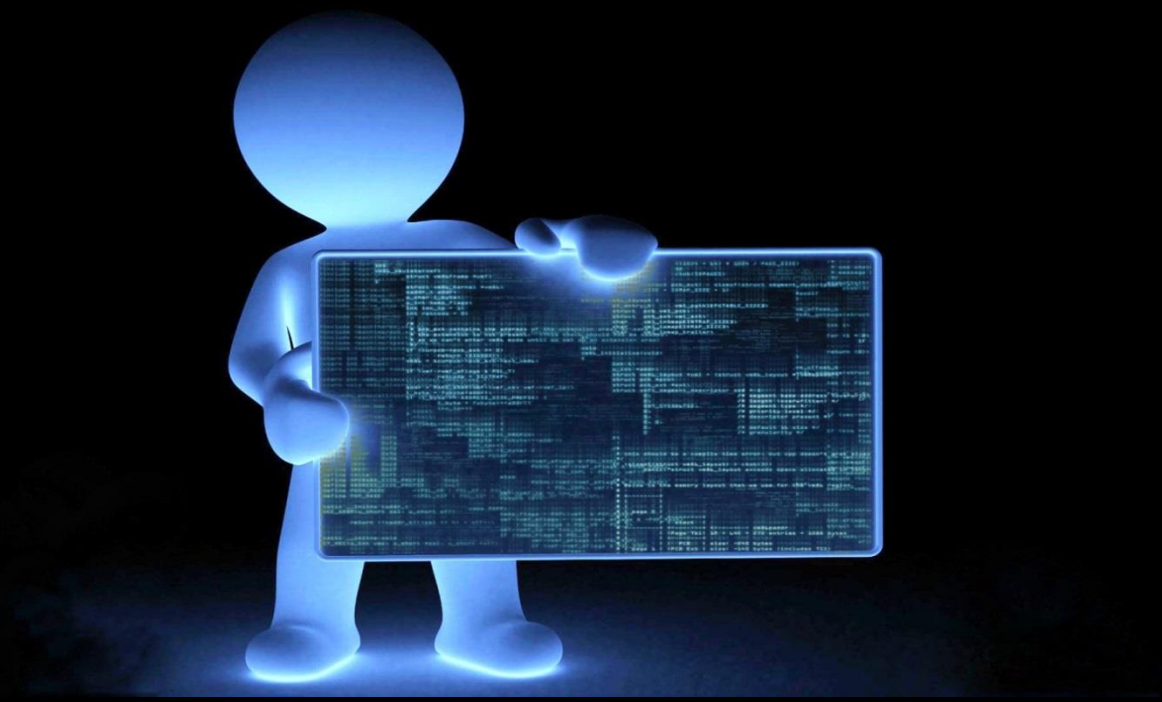
② Singleton

싱글톤의 단점

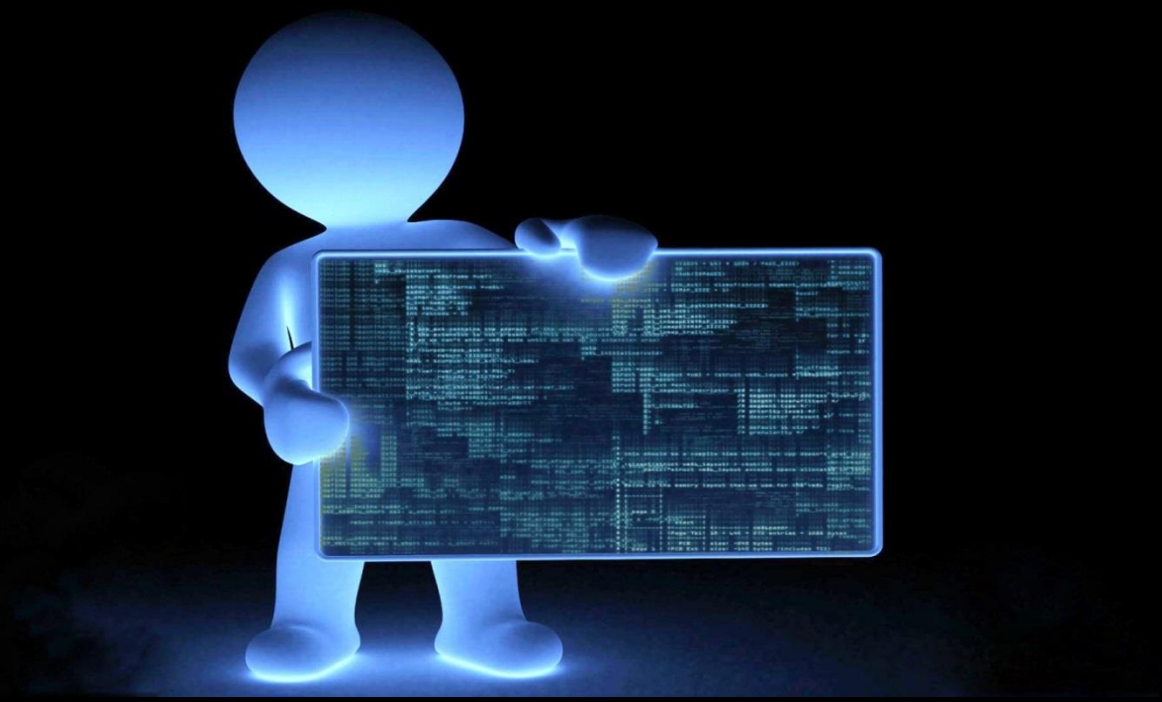
1. 단일 객체를 공유하므로 멀티 쓰레드 환경을 반드시 고려해줘야함

2. 클래스끼리의 결합도(Coupling)을 높이기 때문에
다른 부분 수정 시 변경해야하는 부분이 많아짐

3. 기존의 단일 객체를 제외한 새로운 객체를 만들어야할 때
코드 수정이 많을 수 있음
- 프로그램의 특성을 잘 이해하는 것이 필요함



들어주셔서 감사합니다



Q & A