

DESIGN PATTERNS

COMMAND PATTERN

ONUR ERYILMAZ



DESIGN PATTERNS NEDİR?

- Design Patterns terimi mimar Christopher Alexander tarafından A Pattern Language isimli kitabında ilk olarak karşımıza çıkmıştır.
- Christopher Alexander kitabında Design Patterns'in açıklamasını şu şekilde yapmıştır;

  «Her bir pattern(düzen) çevremizde oluşan ve kendini tekrar eden problemleri tanımlar ve bu problemler için tekrar tekrar kullanılabilecek ana çözümü sunar.»
- Alexander bu tanımı binalar ve şehirler için kullanmış olsa da bu tanım yazılımdaki nesne yönelimli design pattern'ler için de geçerlidir.
- Türkçe olarak «Tasarım Şablonu» olarak kullanılabilir.
- Kod bağımsız olarak tanımlanır fakat uygulamada yazılım dillerinde farklılıklar gösterebilir.
- En önemli kaynak olarak «Design Patterns: Elements of Reusable Object Oriented Software» gösterilebilir.

DESIGN PATTERNS NEDİR?(DEVAM)

- Bir Design Pattern genel olarak 4 ana tanımlamadan oluşur:

  - Name(İsim) : Problemin, çözümünün ve sonuçlarının anlaşılabileceği bir iki kelimeden oluşan bir isim.

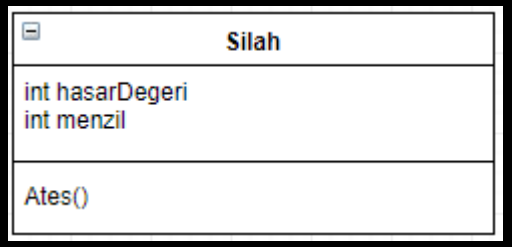
  - Problem : Pattern'in hangi problem(ler) oluştuğunda kullanılabileceği. Problemin genel açıklaması.

  - Solution(Çözüm): Class'ların ve Obje'lerin bu problemi çözmekte nasıl kullanılabileceği.

  - Consequenses(Sonuçlar): Pattern'i yazılıma uyguladığımızda ne gibi sonuçlar doğurabileceği.(Zaman, Performans, Yazılım esnekliği vs.)

DESIGN PATTERNS NEDİR?(DEVAM)

- Bir yazılımda birden fazla Design Pattern kullanılabilir. Birbirleri ile sık kullanılan Design Pattern'ler olduğu gibi birbirine alternatif Design Pattern'ler de bulunmaktadır.
- Object Oriented Programming(OOP) Languages(Nesne Yönelimli Programlama Dilleri) ile kullanılır. (C#, C++, Java vs.)
- OOP'de objeler(nesneler) sınıf (class) olarak adlandırılan şablonlardan türetilir. Bir nesne bir sınıfın örneğidir.



- Design Pattern'ler kendi içlerinde amaçlarına göre 3 temel kategoriye ayrılır.
  - Creational : Obje yaratımı ile ilgili pattern'ler.
  - Structural: Objelerin düzeni ile ilgili pattern'ler.
  - Behavioral: Objelerin birbirleri ile iletişimi ve sorumlulukları ile ilgili pattern'ler.

COMMAND PATTERN

- Gang of Four'un bu patterni tanımlaması;
  «Bir isteği obje olarak içerir, böylelikle isteği yapanları farklı istekler,sıralı istekler ve log istekleri ile parametrize edebilmeyi sağlar, ayrıca geri döndürülebilir.»

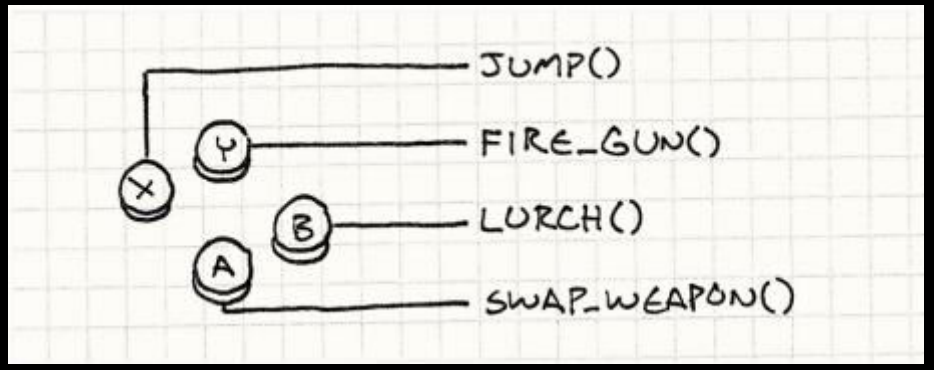
- Game Programming Patterns kitabının yazarı ise bu tanımı karışık ve anlaşılmaz buluyor. Ve kendi tanımlarını yapıyor;
  «Command somutlaştırılmış metod(fonksiyon) çağrılarıdır.»
  Somutlaştırılmış metod çağrısını ise obje gibi çağrılabilen metodlar olarak tanımlıyor.

- Command Design Pattern, Behavioral(Davranışsal) pattern'lere dahil ediliyor.

- Command Design Pattern ile kullanılan 5 ana terim vardır. Bunlar;
  - Receiver(Alıcı) => İşlemlerin olduğu gerçek nesne.
  - Command(Komut) => Receiver nesnesindeki işlemleri çalıştırabilecek, Concrete Command nesnelerinin uygulaması gereken soyut class.
  - Concrete Command(Somut Komut) => Command nesnelerini uygulayan receiver'daki metodların çalıştırılacağı nesne.
  - Invoker(Çağırıcı) => İçerisinde en az 1 adet concrete command bulunduran ve command nesnelerindeki metodu çalıştıran nesne.
  - Client(Gönderen) => Komutu(İsteği) gönderen nesne.

COMMAND PATTERN(DEVAM)

- Oyun yapımında Command Pattern'in sık kullanıldığı alan Input Manager'lardır.
- Örnek;

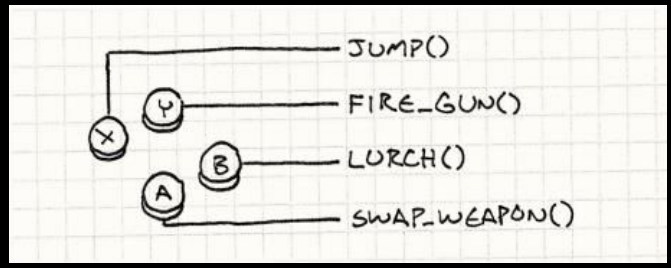



```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();
  else if (isPressed(BUTTON_Y)) fireGun();
  else if (isPressed(BUTTON_A)) swapWeapon();
  else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

Eğer oyunumuzda oyuncuların bu tuşları değiştirmesini istemiyorsak yukarıdaki kod işe yarayabilir. Fakat günümüzde bir çok oyun oyuncuların istedikleri tuşlara istedikleri komutları atayabilmelerine imkan veriyor.

Bunu destekleyebilmek açısından Jump(), FireGun(), Lurch() ve SwapWeapon() metodlarını değiştirilebilir hale getirmemiz gerekiyor. Bunu yapmak için de bu komutları nesneleştirmemiz gerekiyor. Command Pattern de tam da bu tip bir iş için biçilmiş kaftan gözüküyor.

COMMAND PATTERN ÖRNEK




```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
};
```

Öncelikle tetiklenebilen ana bir Command class'ı tanımlıyoruz.

```
class JumpCommand : public Command
{
public:
  virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
  virtual void execute() { fireGun(); }
};
```

Daha sonra her bir buton komutu için bu class'tan türemiş alt class'ları oluşturuyoruz.

```
class InputHandler
{
public:
  void handleInput();

  // Methods to bind commands...

private:
  Command* buttonX_;
  Command* buttonY_;
  Command* buttonA_;
  Command* buttonB_;
};
```

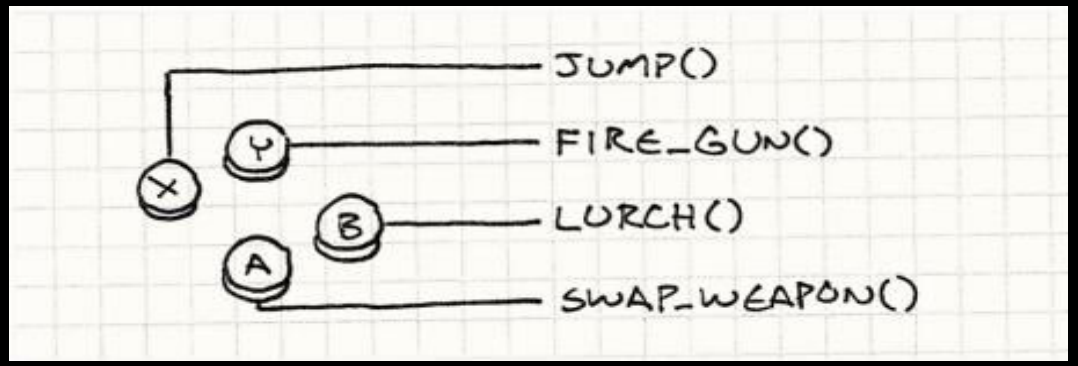
Daha sonra InputHandler class'ı içerisinde butonları birer Command nesnesi olarak tanımlıyoruz.
Bu class'ın içerisinde HandleInput() adında bir metodumuz da mevcut.

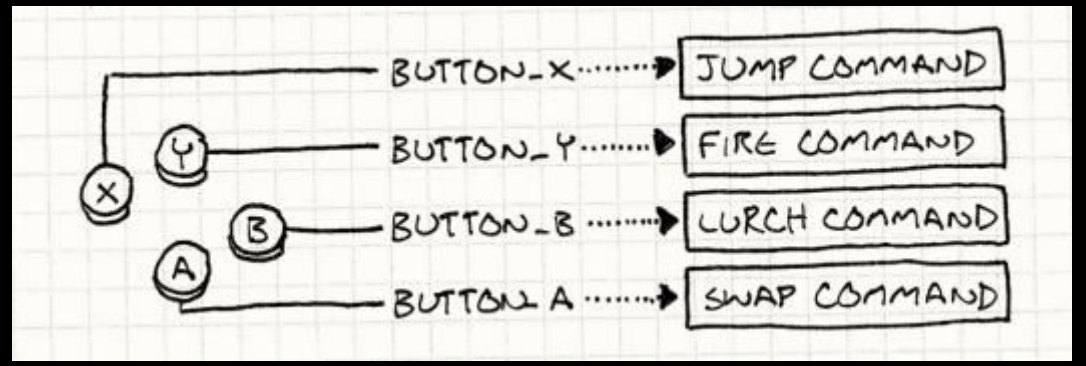
COMMAND PATTERN ÖRNEK(DEVAM)

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();
  else if (isPressed(BUTTON_Y)) fireGun();
  else if (isPressed(BUTTON_A)) swapWeapon();
  else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();
  else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

HandleInput metodu içerisinde hangi tuşa basıldığında hangi Command'ın tetiklenmesi gerekeceğini belirtiyoruz.





Görüldüğü üzere artık butonlar direk olarak metodun kendisine bağlı değil arada bir Command ara yüzü mevcut.

COMMAND PATTERN ÖRNEK(DEVAM)

Bu sistem eğer bu komutlar ile yönetilecek oyun objesi sadece oyuncunun yönettiği karakter ise işe yarar. Peki yaptığımız oyun bir strateji oyunu ise ve komutlar ile oyundaki birimleri yöneteceksek?

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute(GameActor& actor) = 0;
};
```

Oluşturduğumuz Command class'ında bulunan ve Command'ın çalıştırılmasını sağlayan execute metodunu actor parametresi alacak şekilde güncelliyoruz.

```
class JumpCommand : public Command
{
public:
  virtual void execute(GameActor& actor)
  {
    actor.jump();
  }
};
```

Command class'ından türeyen JumpCommand class'ındaki execute metodunu parametre ile gelen actor oyun objesinin Jump metodunu çalıştırması için güncelliyoruz.

```
Command* InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) return buttonX_;
  if (isPressed(BUTTON_Y)) return buttonY_;
  if (isPressed(BUTTON_A)) return buttonA_;
  if (isPressed(BUTTON_B)) return buttonB_;

  // Nothing pressed, so do nothing.
  return NULL;
}
```

InputHandler class'ında bulunan HandleInput metodu artık bir Command döndürecek. Çünkü gelen komutu gerçekleştirecek actorün hangi actor olduğunu bilmiyor(daha doğrusu bunu bilmek InputHandler'ın görevi değil).

COMMAND PATTERN ÖRNEK(DEVAM)

```
Command* command = inputHandler.handleInput();
if (command)
{
  command->execute(actor);
}
```

Input Handler class'ının HandleInput metodundan dönen Command'ı actor ile birlikte çalıştırıyoruz.

Command Pattern'in tanımında Command'ların geri döndürülebilir olduğundan bahsediliyordu. Bu özelliğin oyunlardaki yansımasına bakalım.
Turn-based(Sıra tabanlı) bir strateji oyunu yaptığımızı düşünelim ve bu oyunda oyuncu yaptığı hamleleri geri alabilmeli.

```
class MoveUnitCommand : public Command
{
public:
  MoveUnitCommand(Unit* unit, int x, int y)
  : unit_(unit),
    x_(x),
    y_(y)
  {}

  virtual void execute()
  {
    unit_->moveTo(x_, y_);
  }

private:
  Unit* unit_;
  int x_, y_;
};
```

Burada bir Command class'ından türetilmiş bir MoveUnitCommand class'ı görüyoruz. Class'ın içerisinde Unit olarak tanımlı bir obje(ilerletilecek obje) ve int türünde tanımlanmış x ve y değerleri bulunuyor. Class içerisinde Command class'ından gelen execute metodu ile bu class'a özel bir de MoveUnitCommand metodu(constructor) görüyoruz. Bir önceki örnekteki Command class'ından farklı olarak burada execute bir actor almıyor yani MoveUnitCommand direk olarak oynatılacak oyun objesine bağlanmış durumda. Yani genel bir move(ilerle) komutu değil.

COMMAND PATTERN ÖRNEK(DEVAM)

```
Command* handleInput()
{
  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
    // Move the unit up one.
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    // Move the unit down one.
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  // Other moves...

  return NULL;
}
```

Önceki örneklerde InputHandler class'ı gelen komutu butona basıldığında direkt olarak tetikliyordu. Burada ise her butona basıldığında yeni bir Command nesnesi üretiyor ve yeni üretilen Command nesnesini döndürüyor.

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
  virtual void undo() = 0;
};
```

Command'lerimizi geri alınabilir yapmak için içerisine undo metodunu ekleyerek güncelliyoruz.

COMMAND PATTERN ÖRNEK(DEVAM)

```
class MoveUnitCommand : public Command
{
public:
  MoveUnitCommand(Unit* unit, int x, int y)
  : unit_(unit),
    xBefore_(0),
    yBefore_(0),
    x_(x),
    y_(y)
  {}

  virtual void execute()
  {
    // Remember the unit's position before the move
    // so we can restore it.
    xBefore_ = unit_->x();
    yBefore_ = unit_->y();

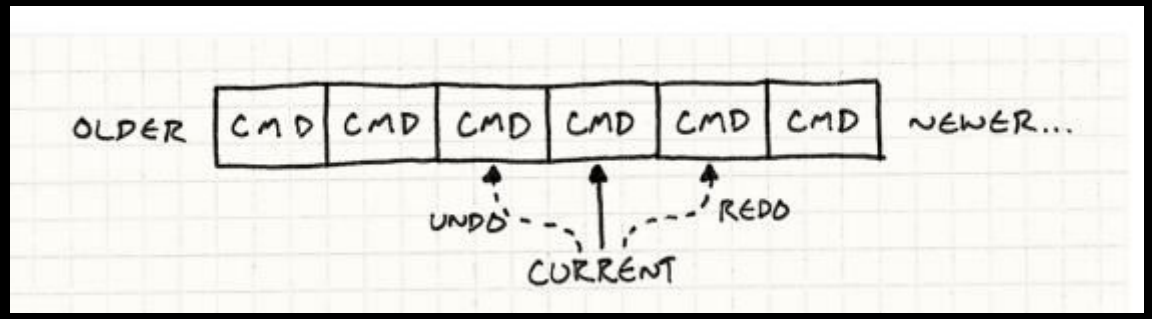
    unit_->moveTo(x_, y_);
  }

  virtual void undo()
  {
    unit_->moveTo(xBefore_, yBefore_);
  }

private:
  Unit* unit_;
  int xBefore_, yBefore_;
  int x_, y_;
};
```

Tabii bir de MoveUnitCommand ‘ı da bu şekilde güncellememiz gerekiyor. Command execute edildiğinde mevcut pozisyon bilgisi saklanacak ardından unit gelen pozisyon bilgisi ile yeni pozisyonuna ilerletilecek. Böylelikle oyuncu hamleyi geri almak istediğinde geri alma tuşuna bastığında Command’ın undo metodu çalışacak ve unit bir önceki pozisyonuna geri dönecek.

COMMAND PATTERN ÖRNEK(DEVAM)



Ve eğer Command'leri bir listede tutarsak, sadece bir kere değil birden fazla geri alma işlemi yapabilmek mümkün olur. Eğer oyuncu geri alma tuşuna basarsa şuanki Command'ın undo'su çalıştırılır ve şuanki Command bir önceki Command'a set edilir. Eğer oyuncu ileri alma tuşuna basarsa, şuanki Command'ın execute'u çalışır ve şuanki Command bir sonraki Command'a set edilir. Eğer oyuncu geri aldıktan sonra farklı bir Command uygularsa şuanki Command'ın ilerisindeki Command'lar silinir.

COMMAND PATTERN UNITY UYGULAMASI

Son örnekten yola çıkarak C# ile yazılmış bir uygulamaya bakalım. Uygulamada sahnede bulunan küp W,A,S,D tuşları ile ileri, geri, sağa ve sola hareket ettirilebilir. Ayrıca U tuşu ile Undo(geri alma) işlemi gerçekleştirilebilr. R tuşu ile geri alınan hareket Redo(ileri alma) yapılabilir. Son olarak da P tuşu ile şimdiye kadar yapılmış hareketler başlangıç noktasına kadar Undo edilebilir.

```
public abstract class Command{
    public virtual void Execute() { }
    public virtual void Undo() { }
}
```

Command class'ı oluşturulur.

Oluşturulan Command class'ından MoveCommand adlı bir class oluşturulur. (Command)

```
public class MoveCommand : Command {

    private GameObject gameObjectToMove;
    private int beforePosX;
    private int beforePosZ;
    private int posX;
    private int posZ;

    public MoveCommand(GameObject go, int x, int z)
    {
        gameObjectToMove = go;
        posX = x;
        posZ = z;
    }
    public override void Execute()
    {
        beforePosX = (int)gameObjectToMove.transform.position.x;
        beforePosZ = (int)gameObjectToMove.transform.position.z;

        gameObjectToMove.GetComponent<PositionChanger>().MoveToPosition(posX, posZ);
    }
    public override void Undo()
    {
        gameObjectToMove.GetComponent<PositionChanger>().MoveToPosition(beforePosX, beforePosZ);
    }
}
```

COMMAND PATTERN UNITY UYGULAMASI(DEVAM)

```
public class InputManager : MonoBehaviour {

    public CommandManager cmdManager;
    public GameObject cube;

    private void Update()
    {
        HandleCommandManagerInput();
        Command cmd = HandleMoveInput();
        if (cmd != null)
            cmdManager.HandleMoveCommands(cmd as MoveCommand);
    }

    public Command HandleMoveInput()
    {
        if (Input.GetKeyDown(KeyCode.W))
            return new MoveCommand(cube, (int)(cube.transform.position.x + 1), (int)cube.transform.position.z);
        else if(Input.GetKeyDown(KeyCode.S))
            return new MoveCommand(cube, (int)(cube.transform.position.x - 1), (int)cube.transform.position.z);
        else if(Input.GetKeyDown(KeyCode.A))
            return new MoveCommand(cube, (int)cube.transform.position.x, (int)(cube.transform.position.z+1));
        else if(Input.GetKeyDown(KeyCode.D))
            return new MoveCommand(cube, (int)cube.transform.position.x, (int)(cube.transform.position.z - 1));
        else
            return null;
    }
    private void HandleCommandManagerInput()
    {
        if (Input.GetKeyDown(KeyCode.U))
            cmdManager.Undo();
        else if (Input.GetKeyDown(KeyCode.R))
            cmdManager.Redo();
        else if (Input.GetKeyDown(KeyCode.P))
            StartCoroutine(cmdManager.RewindPlay());
    }
}
```

Input Manager Class'ı.(Client)

COMMAND PATTERN UNITY UYGULAMASI(DEVAM)

```
public InputManager inputManager;
public Command currentCommand;
public int currentCommandIndex=0;
public List<Command> moveCommandList = new List<Command>();

public void HandleMoveCommands(MoveCommand moveCommand)
{
    if (currentCommandIndex < moveCommandList.Count - 1)
        moveCommandList.RemoveRange(currentCommandIndex + 1, (moveCommandList.Count - 1) - currentCommandIndex);

    moveCommandList.Add(moveCommand);
    currentCommand = moveCommand;
    currentCommandIndex = moveCommandList.Count - 1;
    currentCommand.Execute();
}
```

InputManager'dan gelen Command'ları işlemek için bir CommandManager class'ı oluşturuyoruz.(Invoker)

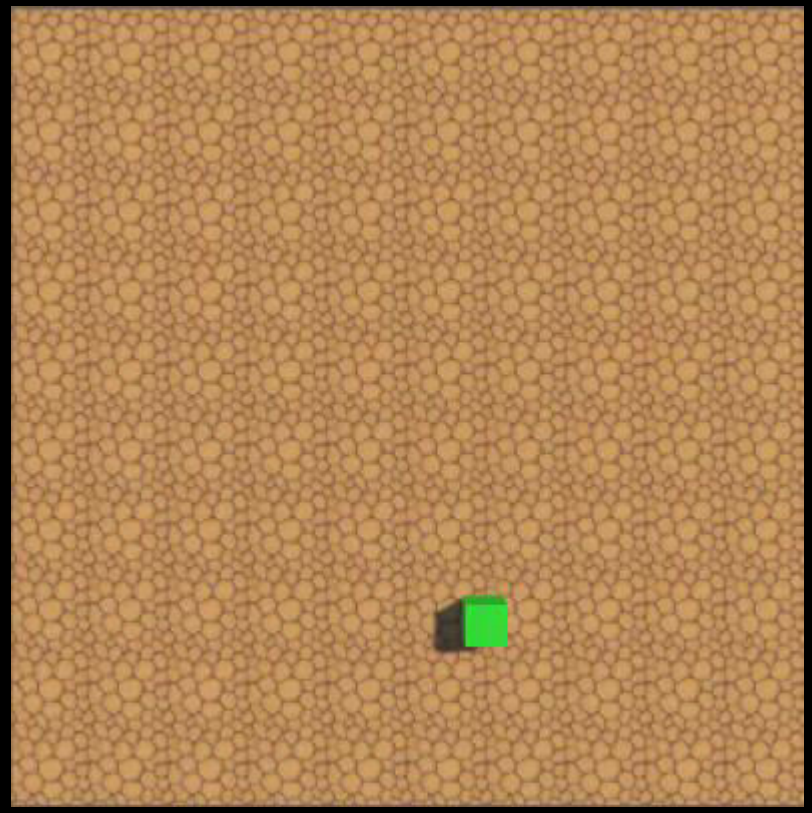
```
public void Undo()
{
    if (currentCommandIndex == 0)
        currentCommand.Undo();

    if (moveCommandList.Count > 1 && currentCommandIndex > 0)
    {
        currentCommand.Undo();
        currentCommand = moveCommandList[currentCommandIndex - 1];
        currentCommandIndex--;
    }
}
```

```
public IEnumerator RewindPlay()
{
    if(moveCommandList.Count > 1)
    {
        int lastIndex = moveCommandList.Count - 1;
        while(lastIndex != -1)
        {
            Undo();
            yield return new WaitForSeconds(1f);
            lastIndex--;
        }
    }
}
```

```
public void Redo()
{
    if (moveCommandList.Count > 1 && currentCommandIndex < moveCommandList.Count - 1)
    {
        currentCommand = moveCommandList[currentCommandIndex + 1];
        currentCommandIndex++;
        currentCommand.Execute();
    }
}
```

COMMAND PATTERN UNITY UYGULAMASI SONUÇ



DİNLEDİĞİNİZ İÇİN TEŞEKKÜRLER...