디자인패턴

-CHAPTER1-

SOULSEEK

목차

단일체(SINGLETON)

1. 단일체(SINGLETON)

디자인 패턴?

- 객체지향을 설계하기 위해 사용하는 객체와 인터페이스를 어떤 상황의 문제에 대한 해법으로 사용되어지는 기법
- 특정한 전후 관계에서 일반적 설계 문제를 해결하기 위해 상호 교류하는 수정 가능한 객체와 클래스들에 대한 설명

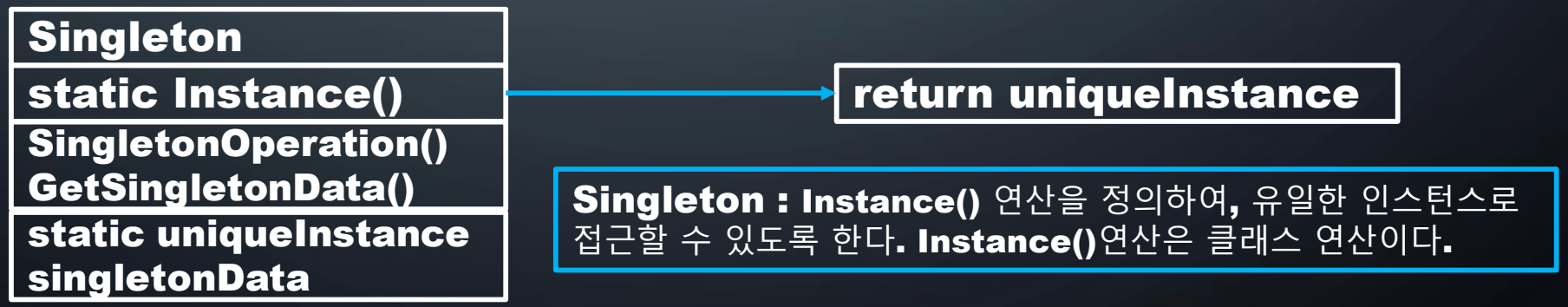
단일체(Singleton)

- 오직 한 개의 클래스 인스턴스만을 갖도록 보장하고, 이에 대한 전역적인 접근점을 제공한다.

활용성

- 클래스의 인스턴스가 오직 하나여야 함을 보장하고 잘 정의된 접근점으로 모든 사용자가 접근할 수 있도록 해야 할 때.
- 유일한 인스턴스가 서브클래싱으로 확장되어야 하며, 사용자는 코드의 수정없이 확장된 서브클래스의 인스턴스를 사용할 수 있어야 할 때

구조



1. 단일체(SINGLETON)

특징

1. 유일하게 존재하는 인스턴스로의 접근을 통제한다.
2. 이름 공간을 좁힌다.
3. 연산 및 표현의 정제를 허용한다.
4. 인스턴스의 개수를 변형하기 자유롭다.
5. 클래스 연산을 사용하는 것보다 훨씬 유연한 방법이다.
6. 다른 패턴들의 구현 방법으로 많이 사용된다.

```
class  Singleton // 클래스 명
{
      static Singleton * m_hThis; // 인스턴스를 넘겨줄 객체
public:
      //인스턴스가 생성되어 있지 않다면 생성하고 존재하면 반환한다.
      static Singleton * GetInstance()
      {
            if (m_hThis == NULL)
                  m_hThis = new Singleton;

            return m_hThis;
      }

Singleton();
~ Singleton();
};
```

1. 단일체(SINGLETON)

학습과제

1. 예제 파일의 리뷰를 들은 뒤 문제 파일 폴더에 있는 파일을 확인하고 문제를 해결해 보자.
2. Student 관리 프로그램을 Singleton으로 바꿔보자.

복합체(COMPOSITE)

2. 복합체(COMPOSITE)

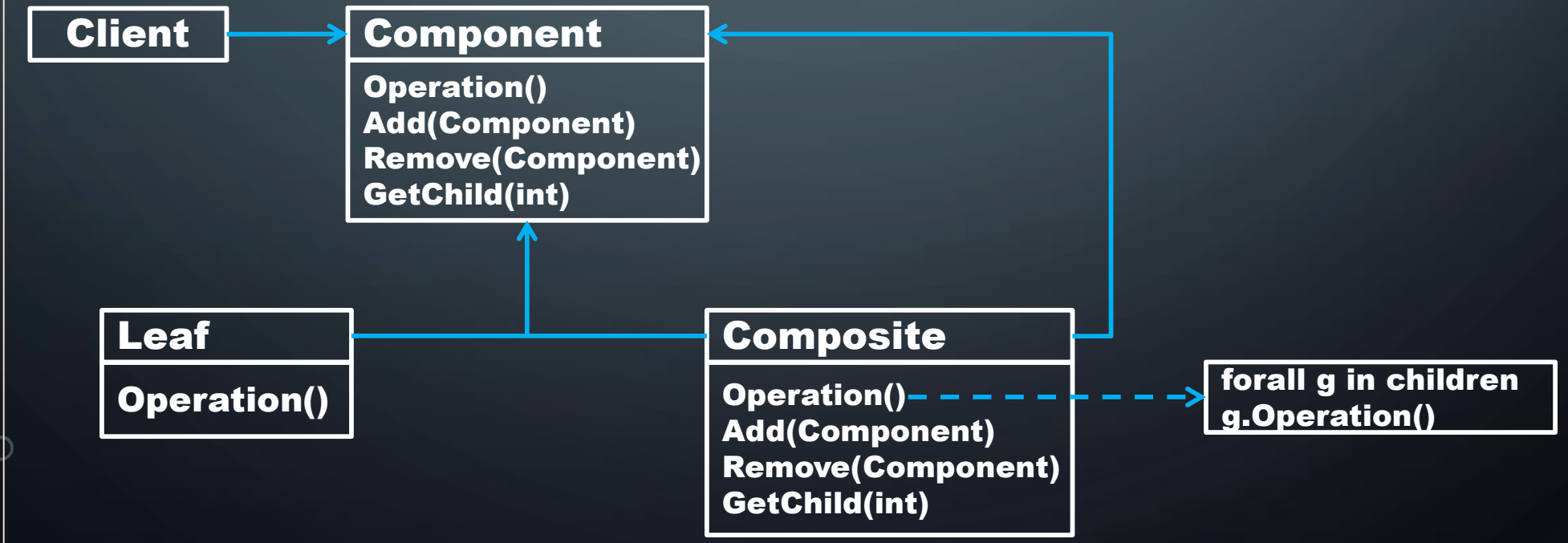
복합체(Composite)

- 부분과 전체의 계층을 표현하기 위해 객체들을 모아 트리구조로 구성.
- 사용자로 하여금 개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록 하는 패턴

활용성

- 부분 – 전체의 객체 개통을 표현하고 싶을 때
- 사용자가 객체의 합성으로 생긴 복합 객체와 개개의 객체 사이의 차이를 알지 않고도 자기 일을 알 수 있도록 만들고 싶을 때
- **Unity3D**와 **Unreal**에서 보이는 **Hierarchy**창이 이런 형태로 이루어져 있다.

구조



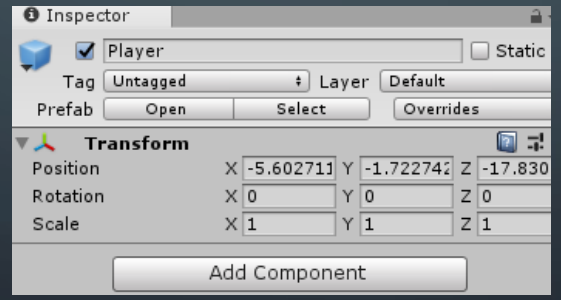
2. 복합체(COMPOSITE)

Component

- 집합 관계에 정의될 모든 객체에 대한 인터페이스를 정의.
- 모든 클래스에 해당하는 인터페이스에 대해서는 공통의 행동을 정의
- 전체 클래스에 속한 요소들을 관리하는데 필요한 인터페이스를 정의
- 순환 구조에서 요소들을 포함하는 전체 클래스로 접근하는 데 필요한 인터페이스 정의 및 구현

Composite

- 자식이 있는 구성요소에 대한 행동을 정의
- 자신이 복합하는 요소들을 저장하면서, **Component** 인터페이스에 정의된 자식 관련 연산을 구현.




```
anmator = GetComponent<Animator>();
```

```
namespace UnityEngine
{
    public class Component : Object
    {
        public Component();
        public GameObject gameObject { get; }
        public string tag { get; set; }
        public Component rigidbody { get; }
        public Component rigidbody2D { get; }
        public Component camera { get; }
        public Component light { get; }
        public Component animation { get; }
        public Transform transform { get; }
        public Component constantForce { get; }
        public Component audio { get; }
        public Component guiText { get; }
        public Component networkView { get; }
        public Component guiElement { get; }
        public Component guiTexture { get; }
        public Component collider { get; }
        public Component collider2D { get; }
        public Component renderer { get; }
```

추상 팩토리(ABSTRACT FACTORY) & 팩토리 메서드(FACTORY METHOD)

2. 추상 팩토리 & 팩토리 메서드

추상 팩토리(Abstract Factory)

- 서로 호환성이 있는 여러 서브 클래스들을 하나의 클래스에서 생성하여 제공하는 것
- 특정 클래스를 사용하기 위해서 특정 조건에 해당할 필요가 있지만, 그 조건과 같은 부모를 상속받은 서브 클래스가 여러 개 존재할 경우에 자칫 다른 서브 클래스를 생성할 경우를 최소화하기 위한 방법

AbstractFactoryEx와 **AbstractFactor** 예제파일을 확인하고 비교해 보자.

팩토리 메서드(Factory Method)

- 프로그램의 뼈대를 만들 때 많이 사용한다.
- 게임에서 **Initialized, Update, Finished**를 만들 때 사용한다.
- **Initialized** 클래스에서는 화면 **View**를 생성하고, **Update** 클래스에서는 **Draw**를, **Finished** 클래스에서는 해제를 한다.

FactoryMethod 예제 파일을 확인하고 알아보자.

학습과제

미로찾기 게임을 작성할 때를 가정하고 위의 패턴을 이용해서 구성을 해보자.