Code Control 애셋의 Message 시스템과 이를 이용한 UI Event Helper 라이브러리를 소개한다. 먼저 Message 시스템은 <https://assetstore.unity.com/packages/tools/utilities/code-control-30871> 의 Code Control이라는 애셋에서 사용하는 일종의 글로벌 이벤트 관리 시스템이다.

이벤트 시스템은 프로그래밍을 하는 과정에서 발생하는 클래스간의 의존성 문제를 회피하며 코드의 재사용성을 늘리는 기법 중 하나인데, Code Control의 Message 시스템은 이러한 이벤트 시스템을 글로벌로 관리하여 극단적으로 코드간의 의존성을 제거하는 라이브러리다. 클래스간에 함수 호출이 필요한 경우 해당 Class의 Instance를 얻어오는 것이 아닌 서로 통신할 Message를 정의하고, 필요한 시점에 해당 메시지를 BroadCasting 하여 등록된 이벤트 리스너의 함수들을 호출해준다.

이를 이용했을 때의 장점은 클래스 간 의존성이 완전히 제거되어 클래스의 확장, 추가, 수정 등의 변화에 대응하기 쉬워지며, 다른 클래스의 Instance를 얻어올 필요가 없기 때문에 메시지 정의만 잘 해주면 결국 각 Message를 통한 Action단위의 코드 구현만을 하면 돼서 구현이 간편해지는 부분이 있다.

반대로 단점은 일반적으로 메시지 방식으로 구현하게 될 경우 어느 클래스에서든 해당 메시지에 접근이 가능하게 되어 구조화를 제대로 못할경우 오히려 객체지향에서 벗어나는 스파게티 코드가 발생할 수 있고, Generic 메시지를 구현하는 과정에서 Type Reflection 코드 등이 빈번하게 일어나게 되어 영향이 크진 않겠지만 일반적인 함수 호출보다는 성능이 떨어지게 된다.

다만 코드 의존성을 없애고 변화에 유연한 구조를 가질 수 있다는 장점을 살려 메시지 방식을 부분적으로 사용하는 것은 생산성을 늘릴 수 있으므로 부분적으로 사용하면 더욱 좋을 듯 하다. 요즘에는 오히려 의존성을 없애는 것이 아닌 반대로 의존성을 명확하게 만들어주는 Zenject 등의 라이브러리들이 있긴 한 것 같은데, 이는 나중에 따로 스터디하여 적용해보도록 하겠다.

메시지 시스템의 사용법으로는 간단히

public class CustomMsg : Message

{

public int val;

public CustomMsg(int val)

{

this.val = val;

}

}

와 같이 CodeControl.Message 를 상속받는 커스텀 메시지를 선언하고, 필요 시 해당 메시지에 멤버변수를 추가한다. 멤버변수를 추가할 경우엔 해당 Message를 날릴 때 마치 함수의 입력인자를 담아 보내듯이 활용할 수 있다.

이를 활용하는 쪽에선

void Awake()

{

Message.AddListener<CustomMsg>(OnCustomMsg);

}

void OnDestroy()

{

Message.RemoveListener<CustomMsg>(OnCustomMsg);

}

void OnCustomMsg(CustomMsg msg)

{

Debug.Log(msg.val);

}

와 같이 받고자 하는 Message를 매개변수로 받는 함수를 정의한 후 제너릭 함수를 통한 Message.AddListener, RemoListener로 리스너 등록/해제를 해주면 된다.

이후엔 해당 메시지를 호출하고자 하는 시점에서

Message.Send<CustomMsg>(new CustomMsg(500));

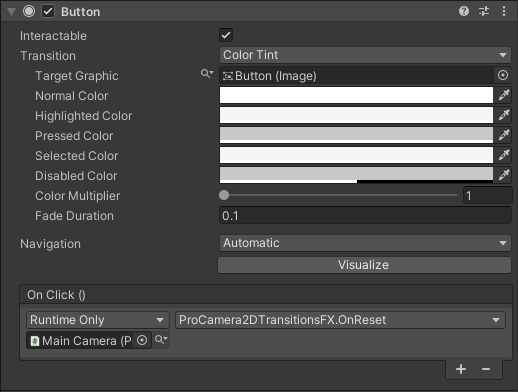
처럼 Message.Send 호출을 통해 현재 등록된 Listener 함수들에 새로 생성된 Message 객체를 전달하여 함수를 호출할 수 있다.

이로써 서로 다른 클래스에서 각자의 정보를 모르더라도 Message를 통한 로직 구현이 가능해진다.

이러한 메시지 시스템을 사용할 경우 메인 로직 구현에도 도움이 되지만, 이 기능을 활용하여 UGUI의 UI Event를 효율적으로 관리할 수 있는 UI Event Helper 시스템을 만들었다.

기존의 UGUI 이벤트의 경우 해당 버튼, 토글 컴포넌트 등에 OnClick() 이벤트 등에서 인스펙터를 활용하여 호출될 함수를 지정해주거나, 코드에서 button 인스턴스를 얻어와서 button.onclick.AddListener(); 등을 통해 이벤트 함수를 등록해주는 과정을 거쳐야 했다.

이 과정에서 Runtime에서 버튼 이벤트를 등록해줘야 하는 경우, 버튼 연결 대상이 바뀌는 경우 등 UI 기능에 수정사항이 생길 경우 바꿔야할 코드 의존성이 늘어나게 되고, 해당 버튼 인스턴스를 얻어올 수 있는 UI Manager 구현 등이 필요하게 된다.



- 기존 Button Event 등록 방식 –

이를 해결하기 위해 Message 방식을 통해 버튼, 토글 이벤트 등이 발생했을 경우 단순히 미리 정의해둔 Message를 Send하게끔 하고, 버튼 이벤트를 받는 쪽에선 해당 Message의 Listener를 붙여서 구현할 수 있게 해준다.

먼저 UI Event의 Base Message는 주로 쓰이는 변수들을 기반으로 아래와 같이 구현하고, 추상함수인 Send를 구현한다.

public abstract class VoidMessageBase : Message

{

public abstract void Send();

}

public abstract class StringMessageBase : Message

{

public string val;

public abstract void Send();

}

public abstract class BoolMessageBase : Message

{

public bool val;

public abstract void Send();

}

이후에 해당 메시지들을 상속받는 Custom Message들의 경우

public class StartButtonClickMsg : VoidMessageBase

{

public override void Send()

{

Message.Send(this);

}

}

public class JumpButtonClickMsg : VoidMessageBase

{

public override void Send()

{

Message.Send(this);

}

}

public class RetryButtonClickMsg : VoidMessageBase

{

public override void Send()

{

Message.Send(this);

}

}

위와 같이 Message.Send(this)를 구현하여 본인의 인스턴스를 보낼 수 있게 한다. 이는 UI Event Helper 쪽에서 Send하기위한 Custom Message의 정확한 Type을 모르기 때문에 Generic함수를 호출할 수 없어서 생기는 문제로.. 외부에서 Message Send를 호출하는 것이 아닌 추상함수를 통한 내부에서 직접 본인의 인스턴스를 보낼 수 있게 해주기 위함이다.

[RequireComponent(typeof(Button))]

public class UIButtonClickMsgHelper : SerializedMonoBehaviour

{

[TypeFilter("GetFilteredTypeList")]

public VoidMessageBase msg;

public IEnumerable<Type> GetFilteredTypeList()

{

var q = typeof(VoidMessageBase).Assembly.GetTypes()

.Where(x => !x.IsAbstract)

.Where(x => typeof(VoidMessageBase).IsAssignableFrom(x));

return q;

}

private void Awake()

{

GetComponent<Button>().onClick.AddListener(OnClicked);

}

void OnClicked()

{

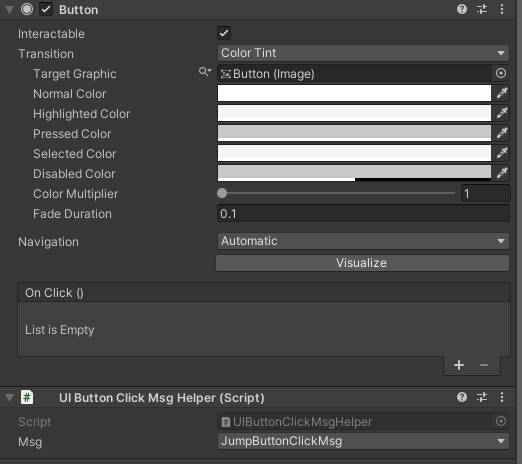
msg.Send();

}

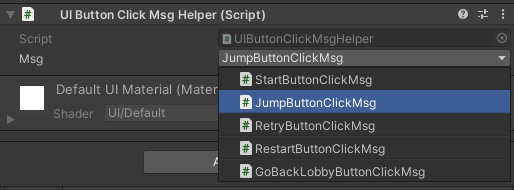
}

실제 사용하는 쪽에선 이런식으로 본인이 보낼 메시지를 지정하여 UI버튼의 OnClick 이벤트가 일어났을 때 해당 메시지를 Send하는 식으로 사용하게 된다. Helper 클래스를 구현하는 과정에서 Message 타입 지정을 편하게 하기 위해 OdinInspector의 TypeFilter를 이용해서 인스펙터에서 UI Message만 드롭다운 메뉴로 선택할 수 있게 하였다. Toggle, InputField 등의 다른 UI들에서도 비슷한 식으로 구현됐다.

에디터에서 사용할때는



이런식으로 각 UI Helper 컴포넌트를 붙여서 해당 UI의 이벤트가 발생했을 때 발생할 Message Type을 지정해주며 되며, Odin Inspector의 Type Filter 덕분에



처럼 정의된 UI Message들만 지정하여 선택할 수 있다.

이로인해 콘텐츠 로직에서 구현될때는 해당 UI 이벤트가 발생했을 때 전달되는 Message 들에 대해 Listener를 등록하여 처리하면 되어서 이벤트가 호출될 대상 함수가 변하거나, 런타임에서 다수의 이벤트가 등록될 때, 버튼들이 수시로 기능을 바꿀 때 등 여러가지 변화 상황에 있어서도 각 코드들과 버튼간에 의존성이 없기 때문에 유지보수가 간편해지는 효과를 갖게 된다.