단위 기능 합성을 위한 클래스 분류

이름 : 조대훈

목차

0. 개요

0.1 기능의 집합으로 완성되는 게임 개발

게임 개발은 여러 가지 작은 기능들이 모여서 완성됩니다. 개발자들은 자연스럽게 작은 기능 단위 개발을 수행하고, 이러한 작은 기능들을 합쳐 보다 큰 기능 단위 개발을 완성해 갑니다.

0.2 기능 단위 접근

저는 이러한 기능 단위 개발을 보며 '**각 기능의 성격별 구현 방식**'에 규칙성이 있을 것이라 생각하였고, 이를 문서화시키면 차후 개발에 큰 도움이 될 것이라 생각했습니다. **'기능의 성격 또는 특징'**에 맞춘 접미어를 사용하여 클래스들을 분류하였고, 규칙성을 찾기 시작했습니다. 개발 진행 초기에는 예상대로 '기능의 특징별' 규칙이 조금씩 보이기 시작했습니다.

0.3 문제점 발생

그러나 개발이 진행되면서 새로운 기능이 추가되거나, 설계 당시 생각하지 못했던 부분들이 생기게 되었습니다. 그럴 때마다 규칙이 조금씩 수정되고, 심지어는 '기능의 성격'이 조금씩 변질되는 경우도 존재했습니다. 개발이 진행될수록, 규칙성은 점점 사라져 가는 것이 보였습니다. 처음에는 제대로 가고 있는 느낌이 들었는데 어느 시점에서 문제가 생긴 것인지 고민하기 시작했습니다. 그리고 저는 문제의 원인을 찾게 되었습니다.

1. 원인과 해결 방안

제가 전제를 잘못 세웠던 것이기 때문에, 사실 원인이라고 할 것은 없습니다.

1.1 원인

'단위 기능' 개발에서 규칙성을 정하는 것을, '기능 합성' 부분까지 적용하려고 했던 것이 문제였습니다. 조금만 생각해보면, 두 부분은 서로 다른 개념이라는 것을 알 수 있습니다. 그러니, 서로 다른 규칙 기준이 필요했습니다. '기능 구현' 단계에서 '기능 합성' 단계를 구별하지 않고 규칙성을 만들려고 하면, 다음과 같은 상황을 겪게 됩니다.

- J 개발자는 객체를 특정 지점에 바로 위치시키는 기능을 만들었습니다.
- 객체의 위치를 지정하는 코드를 Positioner 라는 접미어를 사용하여 표현하기로 하였습니다.
- 그런데, 추가된 기획에서 베지어 곡선을 통한 동적 이동 기능이 있으면 좋겠다고 합니다.
- J 개발자는 동적 이동 기능 정도야, Positioner 클래스에 포함할 수 있다고 생각하고 추가합니다.
- 그런데, 이후 추가 기획에서 이동한 객체가 비활성화되기를 원했습니다.
- Positioner 접미어에 '비활성화' 기능은 조금 어울리지 않지만 따로 나누기에는 애매하여 추가하기로 합니다.
- J 개발자는 고민하다가 GPT 의 도움으로 PositionActivator 라는 접미어를 사용하기로 하였습니다.
- 그런데, 이후 추가 기획에서 이동된 객체가 '삭제'되는 기능도 필요하다고 합니다.
- J 개발자는 울며 겨자먹기로 PositionActivatorAndDestroyer 라는 접미어를 사용하기로 하였습니다.
- 물론, 이미 코드는 책임이 과해진 상태여서, 이 코드는 나중에 재사용하기 애매해졌습니다.

조금 억지스러운 예시이지만, 이런 식으로 '기능 구현' 단계에서 '기능 합성' 단계를 구별하지 않고 규칙성을 만들려고 하면, 완성된 클래스는 확장성도 재사용성도 부족한 결과물이 될 확률이 높습니다.

1.2 해결방안

'단위 기능 구현'과 '기능 합성 구현'을 나누는 것입니다.

1.2.1 단위 기능 구현

- 수많은 기능이 있기에, 정형화된 방식이 거의 없습니다. 그저 해당 기능을 많이 접하면서, 기능을 최소한의 책임 클래스로 나누어 관리하는 방법을 찾아야 합니다.
- 알고리즘이 적용되는 부분이기도 합니다.

1.2.2 기능 합성을 위한 클래스 분류

- '한 개의 단위 기능'을 '한 개의 MVC 그룹 형태'로 만듭니다.
- '단위 기능의 주요 로직'은 MVC 그룹의 Model 클래스에서 구현됩니다.

1.2.3 기능 합성 구현

- 'n 개의 MVC 그룹' (즉, n 개의 단위 기능)이 하나의 '독립화된 데이터 클래스'를 공유하게 합니다.
- 하나의 'Observer 패턴이 적용된 독립화된 데이터 클래스'를 사용하면, 각 기능은 서로 합성된 느낌을 갖게 됩니다.

  (기능이 데이터를 변경하면, Observer 의 Notify 를 통해 서로 연동하여 동작하기 때문입니다.)

2. 기능 합성을 위한 클래스 분류

독립된 데이터 클래스를 사용하여 각 '기능들의 합성 방식'에 규칙을 만들 수 없을까 고민하던 도중, 옛날에 공부했던 MVC 패턴이 생각났습니다. 당시 저는 Model 클래스에 Data 클래스와 Logic 클래스를 넣어서 사용했습니다. 이러한 사용 방식으로 '객체 간에 데이터 사용을 위한 클래스 참조'가 너무 과해져 사용을 포기했었지만, MVC 패턴은 데이터를 잘 참조하면 기능을 구현하는 것에는 문제가 없었습니다.

저는 MVC 패턴에서 사용하는 클래스 분류법을 가져와 규칙성을 만들기 시작했고, 예측대로 클래스들이 어느 정도 규칙성을 갖게 되었습니다. 그리고 이러한 '합성된 기능 집단'을 System 이라 부르기로 했습니다.

과거에 MVC 패턴을 즐겨 사용하던 덕분에, Model, Controller, View 의 의미를 클래스 이름을 지을 때 자주 사용하게 되었습니다. 저는 System 내부 클래스를 총 5 가지로 분류합니다.

2.1 Data 클래스

- 게임 내 기능 구현을 위한 정보들이 기록 및 저장되어 있는 클래스입니다.

2.1.1 필드 멤버

- 각종 형식의 데이터 변수들이 존재합니다.
- 데이터의 동적 변경됨을 공지하기 위한 Observer 관련 변수가 존재합니다.

2.1.2 접근 방법

- Singleton 패턴을 통해 독립적으로 존재합니다.

2.1.3 초기화 방법

- 게임의 실행, 로그인 성공, 특정 시스템 실행 등 다양한 시점에서 초기화 수행합니다.

2.1.4 책임

- 소유하고 있는 데이터의 Get 과 Set 기능 제공합니다.
- 각 자료구조에 알맞은 추가 메소드 제공합니다.

2.2 InteractionView 클래스

- 화면에 보이는 UI/UX 가 컴포넌트하는 클래스입니다.
- 사용자와 상호작용을 인식하는 역할을 합니다.

2.2.1 필드 멤버

- 상호작용에 의해 수행되어야 할 기능을 호출할 Controller 클래스를 참조하고 있습니다.

2.2.2 접근 방법

- 유니티의 Input 클래스, UnityEngine.EventSystems 네임스페이스 내의 인터페이스 상속, 유니티 UI 버튼 클릭 등, 다양한 방식으로 접근 가능합니다.
  (UI 버튼, 마우스, 키보드 인식 등, 유니티 엔진에서 구현이 되어 있는 경우가 많습니다.)

2.2.3 초기화 방법

- [SerializedField]를 통해 Controller 클래스만 참조하면, 딱히 초기화할 멤버가 없습니다.

2.2.4 책임

- 사용자의 입력을 인식하고 Controller 클래스의 메소드를 호출하는 메소드가 존재합니다.

2.3 Controller 클래스

- 상호작용의 기능을 수행하기 위해 접근하는 클래스입니다.
- '기능 수행'을 위한 Model 클래스에 대한 접근성을 제공합니다.

2.3.1 필드 멤버

- Model 클래스를 참조합니다.
- 특정 Data 의 변경에 영향을 받는 기능의 경우, 데이터 클래스의 ObserverSubject 를 참조합니다.

2.3.2 접근 방법

- MonoBehaviour 를 상속하고 있어서, [SerializedField]를 통해 접근하면 됩니다.

2.3.3 초기화 방법

- Awake()에서 Model 클래스를 정의합니다.
- OnEnable()과 OnDisable()에서 ObserverSubject 의 Register, Remove 를 수행합니다.

2.3.4 책임

- 'Data 의 변경됨을 인식'하는 Observer-Subscriber 의 Update 메소드가 존재합니다.
- 'InteractionView 클래스'가 기능 수행을 위해 접근할 메소드가 존재합니다.

2.4 Model 클래스

- 데이터 가공에 필요한 로직을 구현합니다.

2.4.1 필드 멤버

- '기능 합성을 위해 공유하여 사용하는 데이터 클래스'를 참조합니다.
- 데이터를 가공하기 위해 필요한 다른 데이터 클래스들을 참조합니다.
- 데이터 가공을 수행하는 '특정 클래스'를 참조하는 경우도 있습니다.
  (데이터 가공 로직이 복잡한 경우, '특정 클래스'에 기능을 위임하여 구현합니다.)

2.4.2 접근 방법

- Controller 클래스가 갖는 메소드를 통해서만 접근 가능합니다.

2.4.3 초기화 방법

- Controller 클래스의 Awake()에서 Model 클래스가 정의되는 순간에만 초기화됩니다.

2.4.4 책임

- 데이터를 가공을 수행하여 수정(Set)하는 메소드가 존재합니다.
  (Observer 와 관련된 데이터를 수정한 경우, Observer 의 Notify 를 호출해야 합니다.)

2.5 View 클래스

- 화면에 보이는 UI / UX 가 컴포넌트하는 클래스입니다.
- 사용자에게 정보를 제공해주는 역할을 수행합니다.
- View 클래스는 1 개 이상 존재합니다.

2.5.1 필드 멤버

- 자신의 UI / UX 를 구성하는데 사용되는 '데이터 클래스'들을 참조합니다.
- '동적으로 변동되는 데이터'에 영향을 받는 경우, 데이터 클래스에 정의된 ObserverSubject 를 참조합니다.

2.5.2 접근 방법

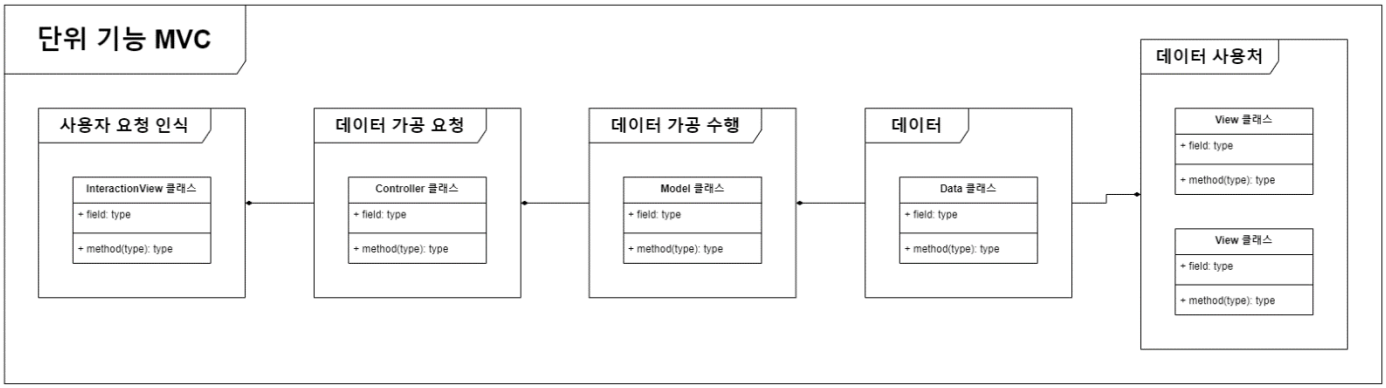
- MonoBehaviour 를 상속하고 있어서, [SerializedField]를 통해 접근하면 됩니다.

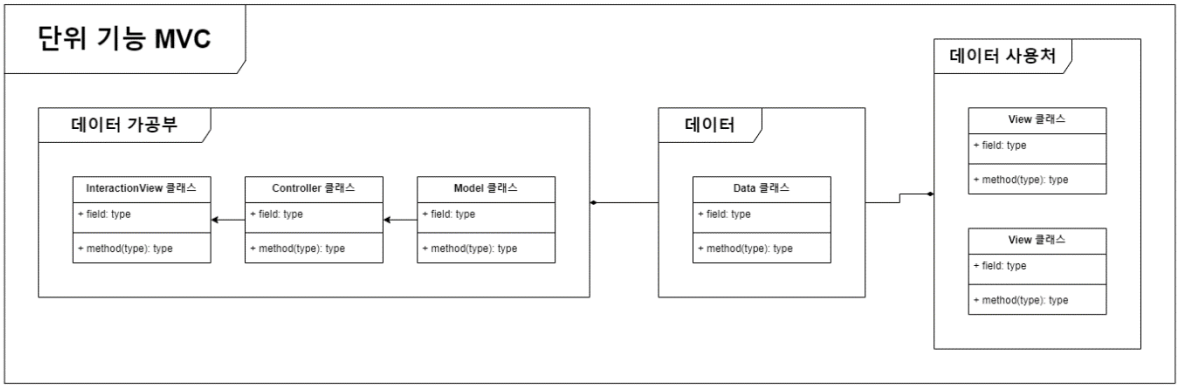
2.5.3 초기화 방법

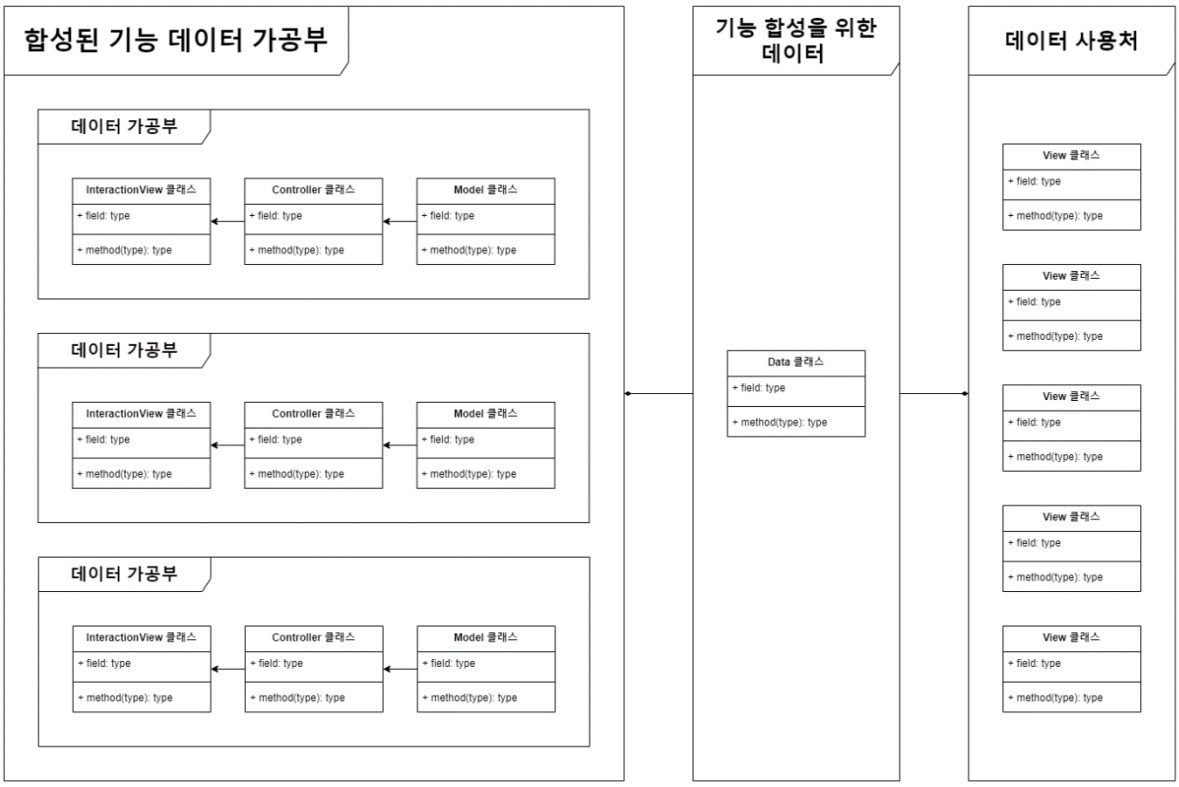
- Awake()에서 데이터 클래스를 참조합니다.
- OnEnable()과 OnDisable()에서 ObserverSubject 의 Register, Remove 를 수행합니다.

2.5.4 책임

- 'Data 의 변경됨을 인식'하는 Observer-Subscriber 의 Update 메소드가 존재합니다.
- '데이터 클래스'를 참조하여 UI / UX 를 갱신하는 메소드가 존재합니다.


단위 기능 MVC
사용자 요청 인식
InteractionView 클래스
+ field: type
+ method(type): type
데이터 가공 요청
Controller 클래스
+ field: type
+ method(type): type
데이터 가공 수행
Model 클래스
+ field: type
+ method(type): type
데이터
Data 클래스
+ field: type
+ method(type): type
데이터 사용처
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type


단위 기능 MVC
데이터 가공부
InteractionView 클래스
+ field: type
+ method(type): type
Controller 클래스
+ field: type
+ method(type): type
Model 클래스
+ field: type
+ method(type): type
데이터
Data 클래스
+ field: type
+ method(type): type
데이터 사용처
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type


합성된 기능 데이터 가공부
데이터 가공부
InteractionView 클래스
+ field: type
+ method(type): type
Controller 클래스
+ field: type
+ method(type): type
Model 클래스
+ field: type
+ method(type): type
데이터 가공부
InteractionView 클래스
+ field: type
+ method(type): type
Controller 클래스
+ field: type
+ method(type): type
Model 클래스
+ field: type
+ method(type): type
데이터 가공부
InteractionView 클래스
+ field: type
+ method(type): type
Controller 클래스
+ field: type
+ method(type): type
Model 클래스
+ field: type
+ method(type): type
기능 합성을 위한 데이터
Data 클래스
+ field: type
+ method(type): type
데이터 사용처
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type
View 클래스
+ field: type
+ method(type): type