* **プロジェクトの流れ**
* **プロジェクトの構造の説明**

このプロジェクトはMVVM設計モデルを参考して開発しています。MVVMとは機能を「Model」（モデル）、「View」（ビュー）、「ViewModel」（ビューモデル）の三つの役割に分離して実装し、それらが連携して処理を進める方式です。「Model」はデータの管理や手続きを扱い、「View」は「Model」からの指示を受けてユーザーへの表示や出力を行います。「ViewModel」は両者の変更を監視し、相手方への反映が必要な場合に通知を行います。（https://ja.wikipedia.org/wiki/Model\_View\_ViewModel）

このプロジェクトでは「BaseData」と「Data」は「Model」部分です。「BaseData」はCSVなどテキストファイルを読み込んでゲームの進捗状況に対応しないデータです。例えば「CharacterBaseData」は「CharacterData.csv」から読み込んだゲームにいるキャラクターの基本情報です。ゲームを始めてから終わるまで変わらないです。「Data」はゲームの進捗状況により変わるデータです。例えばキャラクターのHPとかLevelとかです。

次は「View」部分です。「View」部分は「UIPanel」と「Object」を二つ種類に分けています。「UIPanel」はゲーム内表示するUIの部分です。「Object」はゲーム世界のものです。例えばプレイヤーのモデルとか弾のモデルとかみたいものです。

最後は「ViewModel」部分です。このプロジェクトでMVVM設計モデルの「ViewModel」通りものはありませんが、一部の役割は「EventMessenger」になりました。「EventMessenger」は「InputManager」と「View」からメッセージを受け取って「Model」にお知らせします。

* **処理の流れの説明**

最初プレイヤーの操作を処理するところは「InputManager.cs」です。「InputManager.cs」でプレイヤーの操作に対応するデリゲート（delegate）とイベント（Event）があります。例えば、Gamepadのスティックに対応するデリゲートは　InputDataWithVector2(Vector2 value)　です。このデリゲートを利用して作ったイベントは　MoveEvent　です。MoveEvent　は名前通りキャラクターの移動する時に使うイベントです。Gamepadの左スティックが動いたら、この動作をMoveEventに登録されているオブジェクトを通知します。そしてイベントを受信登録したオブジェクトはこの動作を処理します。

Event

動きを通知

プレイヤー

設備の入力

実行

InputManager

Event

もしプレイヤーは画面のボタンを押すとかの操作したら、この操作を処理するのは「InputManager」ではなくUIの「View」です。それぞれの「View」はプレイヤーの操作に対して、「EventMessenger」を利用し「Data」の状態を更新するための指示内容を表すメッセージを持っています。そして、「Data」は「View」からのメッセージを受け取って自分自身の状態を更新します（void Update()）。「View」は「Data」に通知できるため、まず「EventMessenger」を使って「View」の動き通知（メッセージ）の受け取りを登録しなければなりません。(delegate void EventDataHandler<T>(T t))

動きの通知受け取り登録

Data

ボタン動きを監視

UI

Buttonの動き

EventMessenger

Data

Data

　「View」からプレイヤーの操作のメッセージの通知を実行し（void Launch(EventMsg msg)）、「Data」にメッセージを渡します。そして、渡されたメッセージの内容に応じて「Data」は状態を更新します（void Update()）。最後状態が更新されたことを検知した「View」が書き換わります。

View

更新

Data

通知

UI

Buttonの動き

EventMessenger

View

更新

Data

このような処理する理由は単方向のデータフローを構築し、データの流れを見失いづらいことです。ゲーム内の動きも把握しやすいです

* **プログラミングの考え**

　例えば、ゲーム開発中にプランナーからこんな企画内容がありました。

1. ヒーローはモンスターに攻撃できます。
2. モンスターは攻撃を受けたらHPが減少します。
3. 武器によりモンスターに与えるダメージが違います。

この設計を実装してみたら、次のようになりました。

モンスターのクラス

1. public class Monster
2. {
3. public String Name { get; set; }
4. public int HP { get; set; }
6. public Monster(String name,int hp)
7. {
8. Name = name;
9. HP = hp;
10. }
11. }

ヒーローのクラス

1. public class Hero
2. {
3. private Random \_random = new Random();
4. public String WeaponTag { get; set; }
5. public void Attack(Monster monster)
6. {
7. if (monster.HP <= 0)
8. {
9. Console.WriteLine("The monster is dead.");
10. return;
11. }
12. if (WeaponTag == "WoodSword")
13. {
14. monster.HP -= 20;
15. if (monster.HP <= 0)
16. {
17. Console.WriteLine("The monster is dead.");
18. }
19. else
20. {
21. Console.WriteLine("Monster " + monster.Name + "loses 20 HP.");
22. }
23. }
24. else if (WeaponTag == "IronSword")
25. {
26. monster.HP -= 50;
27. if (monster.HP <= 0)
28. {
29. Console.WriteLine("The monster is dead.");
30. }
31. else
32. {
33. Console.WriteLine("Monster " + monster.Name + "loses 50 HP.");
34. }
35. }
36. else if (WeaponTag == "MagicSword")
37. {
38. int loss = (\_random.NextDouble() < 0.5) ? 100 : 200;
39. monster.HP -= loss;
40. if (monster.HP <= 0)
41. {
42. Console.WriteLine("The monster is dead.");
43. }
44. else
45. {
46. Console.WriteLine("Monster " + monster.Name + "loses " + loss +" HP.");
47. }
48. }
49. else
50. {
51. Console.WriteLine("The hero has no weapon, cannot attack!");
52. }
53. }
54. }

Mainのクラス

1. static void Main(string[] args)
2. {
3. Monster monster1 = new Monster("MonsterA", 50);
4. Monster monster2 = new Monster("MonsterB", 50);
5. Monster monster3 = new Monster("Boss", 500);
6. Hero hero = new Hero();
7. hero.WeaponTag = "WoodSword";
8. hero.Attack(monster1);
9. hero.WeaponTag = "IronSword";
10. hero.Attack(monster2);
11. hero.Attack(monster2);
12. hero.WeaponTag = "MagicSword";
13. hero.Attack(monster3);
14. hero.Attack(monster3);
15. hero.Attack(monster3);
16. hero.Attack(monster3);
17. hero.Attack(monster3);
18. Console.ReadLine();
19. }

このプログラマは企画内容通り正しく行うことができましたが、プログラミングの設計からみると改善してほしいところがあります。

一つ目はヒーローのクラスです。ヒーローのクラスで似ている「if...else」構造がたくさんあります。二つ目は武器に対する処理はヒーローのクラスに行うべきではないと思います。もし武器を追加するなら、またヒーローのクラスの構造を変わらなければならないです。この二つ問題を解決するため、コードはつぎのようになりました。

攻撃のインターフェイス

1. interface IAttackStrategy
2. {
3. void AttackTarget(Monster monster);
4. }

木剣のクラス

1. public class WoodSword : IAttackStrategy
2. {
3. public void AttackTarget(Monster monster)
4. {
5. monster.Notify(20);
6. }
7. }

鉄剣のクラス

1. public class IronSword : IAttackStrategy
2. {
3. public void AttackTarget(Monster monster)
4. {
5. monster.Notify(50);
6. }
7. }

マジックソードのクラス

1. public class MagicSword : IAttackStrategy
2. {
3. private Random \_random = new Random();
4. public void AttackTarget(Monster monster)
5. {
6. int loss = (\_random.NextDouble() < 0.5) ? 100 : 200;
7. monster.Notify(loss);
8. }
9. }

モンスターのクラス

1. public class Monster
2. {
3. public String Name { get; set; }
4. private int HP { get; set; }
5. public Monster(String name,int hp)
6. {
7. this.Name = name;
8. this.HP = hp;
9. }
11. public void Notify(int loss)
12. {
13. if (this.HP <= 0)
14. {
15. Console.WriteLine("The monster is dead.");
16. return;
17. }
18. this.HP -= loss;
19. if (this.HP <= 0)
20. {
21. Console.WriteLine("The monster is dead.");
22. }
23. else
24. {
25. Console.WriteLine("Monster " + monster.Name + "loses " + loss +" HP.");
26. }
27. }
28. }

ヒーローのクラス

1. public class Hero
2. {
3. public IAttackStrategy Weapon { get; set; }
5. public void Attack(Monster monster)
6. {
7. this.Weapon.AttackTarget(monster);
8. }
9. }

Mainのクラス

1. static void Main(string[] args)
2. {
3. Monster monster1 = new Monster("MonsterA", 50);
4. Monster monster2 = new Monster("MonsterB", 50);
5. Monster monster3 = new Monster("Boss", 500);
6. Hero hero = new Hero();
7. hero.Weapon = new WoodSword();
8. hero.Attack(monster1);
9. hero.Weapon = new IronSword();
10. hero.Attack(monster2);
11. hero.Attack(monster2);
12. hero.WeaponTag = new MagicSword();
13. hero.Attack(monster3);
14. hero.Attack(monster3);
15. hero.Attack(monster3);
16. hero.Attack(monster3);
17. hero.Attack(monster3);
18. Console.ReadLine();
19. }

クラスが増えてきましたが、各クラス内のメソッドのコードが短く役割にも明確になっています。また、新しい武器を追加したい場合、IAttackStrategyを実装する新しいクラスを作成するだけで、キャラクターが新しい武器を装備できるようになります。これで機能の拡張や改善は簡単にできます。

以上の例は今プロジェクトで「Data」と「View/Object」です。