1 Strategy パターン

1.1 概要

この章ではゲームプログラムのエネミーの実装を例に上げて Strategy パターンについて見ていきましょう。

1.2 新たな欲求を取り扱うための方法

ソフトウェアの開発において、初期段階で要求仕様がガチっと決まって、以降仕様変更、追加が起きないということはまずありえません(悲しいことに)。特にゲーム開発は面白さという定義が曖昧なものを追求します。そのため仕様変更、追加というものは、他のソフトウェア開発よりも頻繁に発生します。デザインパターンの多くは、このような仕様変更、追加といった流動的な要素のカプセル化がテーマになっていて、この Strategy パターンも流動的な要素のカプセル化がテーマとなっているパターンです。

1.3 流動的な要素

Strategy パターンはアルゴリズムという流動的な要素をカプセル化します。では、エネミーのアルゴリズムで流動的な要素とは何があるか考えてみましょう。エネミーがプレイヤーを発見した時の移動の仕方(これがアルゴリズム)だけを考えても色々な種類があります。例えばフラフラしながらこちらに向かってくるエネミー、一直線にダッシュで向かってくるエネミー、経路探索を行って障害物を正しく避けて知性を感じさせるエネミーなどなど。これに Strategy パターンを使わずに実装すると下記のような実装になるでしょう。

```
class Enemy{
private:
  //プレイヤーを発見したときの移動処理。
  enum FindMoveType{
    eFindMoveType_FuraFura,       //フラフラしながら動いてくる。
    eFindMoveType_Dash,           //一直線にダッシュしてくる。
    eFindMoveType_PathFinding,    //経路探索を行って向かってくる
  };
  FindMoveType moveType;    //移動の種類
                  ・
                  ・
                  ・
};
```

このように moveType というメンバ変数を作成して、if 文や switch 文を使用してアルゴリズムを分岐させる方法が考えられます。

```
	switch(moveType){
		case eFindMoveType_FuraFura:
			//フラフラ向かってくる場合のアルゴリズムをここに記述。
			break;
		case eFindMoveType_Dash:
			//ダッシュで向かってくる場合のアルゴリズムをここに記述。
			break;
		case eFindMoveType_PathFinding:
			//経路探索を行って向かってくる場合のアルゴリズムをここに記述。
			break;
	}
```

敵がプレイヤーを発見中の移動処理なため、プレイヤーまで移動することを諦めるアルゴリズムもあるかもしれません。

```
	bool isEndFindMove = false;
	switch(moveType){
		case eFindMoveType_FuraFura:
			//フラフラ向かってくる場合のアルゴリズムをここに記述。
			break;
		case eFindMoveType_Dash:
			//ダッシュで向かってくる場合のアルゴリズムをここに記述。
			break;
		case eFindMoveType_PathFinding:
		//経路探索を行って向かってくる場合のアルゴリズムをここに記述。

			break;
	}
}
```

ではこの実装の問題点を見ていきましょう。

・凝集度が低いため、Enemy クラスの保守が困難になりやすい。

・仕様の追加が発生したときに、全ての switch、if 文にコードをもれなく追加する要がある。

例えば eFindMoveType_FuraFura のアルゴリズムに仕様変更が発生したときに、このような実装の場合はしばしばエネミーの全てのコードを調べることになります。これは eFindMoveType_FuraFura に関係する処理がエネミーのコードに点在していて、それを探す羽目になるからです。

1.4 流動的な要素のカプセル化

デザインパターン、オブジェクト指向の重要な考え方として責任の委譲があります。今回の例で流動的な要素は**プレイヤーを発見した時の移動アルゴリズム**です。先ほどの実装ではエネミークラスがエネミーの移動に関してまで責任をもっています。この責任を他のクラスに委譲することが Strategy パターンのキモになります。

まず、抽象化レイヤーとして、インターフェースクラスの IEnemyFindMove クラスを作成します。

```
class IEnemyFindMove{
public:
    virtual void FindMove() = 0;
    virtual bool IsEndFindMove();
};
```

そして IEnemyFindMove を継承して、インターフェースの中身を実装した各移動アルゴリズムのクラスを作成します。

```
class EnemyMove_FuraFura : public IEnemyFindMove{
public:
    void FindMove();
    bool IsEndFindMove();
};
class EnemyMove_Dash : public IEnemyFindMove{
public:
```

```
        void FindMove();
        bool IsEndFindMove();
    };
    class EnemyMove_PathFinding : public IEnemyFindMove{
    public:
        void FindMove();
        bool IsEndFindMove();
    };
```

Enemy クラスは IEnemyFindMove のポインタを保持するように変更します。また、コンテキストに合わせて適切な敵の発見アルゴリズムを使用する必要があるため、アルゴリズムクラスのインスタンスを生成するファクトリ関数が必要になるでしょう。

```
    class Enemy{
    private:
        enum FindMoveType{
            eFindMoveType_FuraFura,         //フラフラしながら動いてくる。
            eFindMoveType_Dash,             //一直線にダッシュしてくる。
            eFindMoveType_PathFinding,      //経路探索を行って向かってくる
        };
        FindMoveType moveType;      //移動の種類
        IEnemyFindMove*    enemyFindMove = NULL;
        void CreateEnemyFindMove()
        {
            switch(moveType){
            case eFindMoveType _FuraFura:
                enemyFindMove = new EnemyMove_FuraFura;
            case eFindMoveType_Dash:
                enemyFindMove = new EnemyMove_Dash;
            case eFindMoveType_PathFinding:
                enemyFindMove = new EnemyMove_PathFinding;
            }
        }
            ・
            ・
```

```
        ・
};
```

Enemy クラスの switch 文で分岐していたコードはポリモーフィズムを活用することで下記のようなコードに置き換えることができます。

```
enemyFindMove->FindMove();
        ・
        ・
        ・
bool isEndMove = enemyFindMove->IsEndMove();
```

Strategy パターンを使用するように設計を変更したことで、プレイヤーを発見時の移動に関する責任は IEnemyFindMove を継承した各派生クラスに委譲されました。エネミークラスは適切なアルゴリズムのインスタンスを生成する責任だけを持っており、移動アルゴリズムに関しての責任は一切持っていません。また、経路探索を行う移動アルゴリズムに仕様変更が発生した場合にプログラマはすぐに EnemyMove_PathFinding クラスを調べに行くようになるでしょう。新しい移動アルゴリズムの仕様が追加された場合は、IEnemyFindMove クラスを継承した新しいクラスを作成して、Enemy クラスの変更はインスタンスの生成する関数だけになります。このように、アルゴリズムを抽象クラス内にカプセル化することで、いつでも交換可能にするというのが Strategy パターンの本質になります。