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。

2 State パターン

2.1 概要

この章では古くからゲームで活用されてきたアクションゲームのエネミーの finite state machin(有限状態機械)を例にして State パターンについて見ていきましょう。前節で勉強した Strategy パターンととてもよく似たパターンとなります。

2.2 状態のカプセル化

Strategy パターンはアルゴリズムを流動的な要素としてカプセル化を行っていました。State パターンではオブジェクトの状態を流動的な要素としてカプセル化します。では下記のような仕様のエネミーの実装を考えてみましょう

普段はダンジョンを徘徊しているが、プレイヤーキャラクターが一定距離以内に入ってきたらプレイヤーを追尾する。

ゲームでよくある仕様のエネミーの実装だと思います。真っ先に思いつく実装は下記のようなものでしょうか。

```
void Enemy::Update()
{
  if(isTrackingPlayer == true){
      //プレイヤーを追尾中
  }else{
      //ダンジョンを徘徊中。
  }
}
```

isTrackingPlayer という bool 型の変数を使って実装しています。このプログラムはエネミーの仕様が単純なものであるうちは何の問題もないでしょう。むしろ小さい仕様であるならこれくらい単純なコードの方が優れています。しかし、エネミーの仕様が複雑になってきたのであれば、リファクタリングを検討した方が良いでしょう。では、エネミーの仕様が下記のように変更された場合を考えましょう。

普段はダンジョンを徘徊しているが、プレイヤーキャラクターが一定距離以内に入ってきたらプレイヤーを追尾する。そしてプレイヤーに追いつくとバトルが始まる。バトルで HP が 1/3 以下になったらバトルから逃亡する。

これを先ほどのようにフラグを使って実装すると下記のようになるでしょう。

```
Void Enemy::Update()
{
   if(isTrackingPlayer == true){
       //プレイヤーを追いかけ中。
   }else if(isBattle == true){
      //バトル中
   }else if(isEscape == true){
      //逃走中。
   }else{
      //徘徊中。
   }
}
```

さて、この実装は先ほどと違ってある問題があります。それは**isTrackingPlayer フラグが true になる時は isBattle フラグ、isEscape フラグを false にする必要がある**ということです。このプログラムをあなただけが保守、拡張しているのであれば、そのルールを厳密に守ればいいだけです。しかし、現実のソフトウェア開発ではあなた以外のプログラマがこのコードの保守、拡張を行うことは珍しいことではありません。そのとき、そのプログラマは果たして正しくフラグの操作を行ってくれるでしょうか？あなたが詳細にコメントを記載していたとしてもそれを期待するのは愚かなことです。そして、あなた自身も一年後にはそのルールを忘れ去ってしまうでしょう。昨日の自分は他人という言葉がソフトウェア開発の世界ではよく言われます。保守、拡張しやすいコートを書くというのは、他人のためよりも自分のためである場合がほとんどです。

では、このコードを FSM を使用するコードに設計変更しましょう。FSM はオブジェクトが同時に取りうる状態は 1 つだけという原則があります。先ほどの Enemy クラスに enum State 型を下記のように定義しましょう。

```
enum State{
    State_Search,    //徘徊中
    State_Tracking, //プレイヤーを追尾中
    State_Battle,    //戦闘中
    State_Escape,   //逃走中
```

```
};
```

そして Enemy クラスに State 型のメンバ変数の state を追加します。

```
class Enemy{
 State state;          //状態を表す変数。
        ・
        ・
        ・
};
```

Enem::Update 関数は下記のように変更します。

```
void Enemy::Update()
{
      if( state == State_Search ){
            //徘徊中。
      }else if( state == State_Tracking){
            //プレイヤーを追いかけ中。
      }else if( state == State_Battle){
            //バトル中
      }else if( state == State_Escape ){
            //逃走中。
      }
}
```

Update のコード自体は大差ないように思えるかもしれませんが、しかし元々の実装とは異なり state は一つの状態しか指さないことが保証されています。このように FSM はオブジェクトが持ちうる有限の状態を定義して、同時に取りうる状態は 1 つだけであることを保証するものとなります。

2.3 if 文から State パターンへのリファクタリング

先ほどのコードには、1 章で勉強した Strategy パターンと同様の問題があるのが分かるでしょうか？Enemy クラスの実装には各状態の処理が記述されていて、凝集度が低くなっています。また、例えばエネミーの状態に応じて描画の仕方を変更することもあるでしょう。武器を取り出すエネミーもいるかもしれません。その場合、Enemy クラスの中に state 変数を使用した if~else 文が点在することになります。

では、この問題を解決するために、Enemy クラスの状態に関するコードで State パターンを使用するようにリファクタリングを行いましょう。まず、IState というインターフェースクラスを作成しましょう。

```
class IState{
public:
	virtual void Update() = 0;
	virtual void Draw() = 0;
};
```

続いて、IState を継承してインターフェースを実装する各状態クラスを作成します。

```
//徘徊中のステートクラス。
class StateSearch : public IState{
public:
	void Update();
	void Draw();
};
//プレイヤーを追跡中のステートクラス。
class StateTracking : public IState{
public:
	void Update();
	void Draw();
};
//バトル中のステートクラス。
class StateBattle : public IState{
	void Update();
	void Draw();
};
```

```
1  //逃走中のステートクラス。
2  class StateEscape : public IState{
3  public:
4      void Update();
5      void Draw();
6  };
7  ―――――――――――――――――――――――――――――――――――――――――――――――――
8  Enemy は状態に合わせて適切なクラスのインスタンスを生成するように変更します。
9  ―――――――――――――――――――――――――――――――――――――――――――――――――
10 void Enemy::ChangeState(State state)
11 {
12     if( currentState != NULL ){
13         delete currentState;
14     }
15     if( state == State_Search ){
16         currentState = new StateSearch;
17     }else if( state == State_Tracking ){
18         currentState = new StateTracking;
19     }else if( state == State_Battle ){
20         currentState = new StateBattle;
21     }else if( state == State_Escape ){
22         currentState = new StateEscape;
23     }
24 }
25 ―――――――――――――――――――――――――――――――――――――――――――――――――
26 Enemy クラスの Update と Draw 関数は下記のようになります。
27 ―――――――――――――――――――――――――――――――――――――――――――――――――
28 void Enemy::Update()
29 {
30     currentState->Update();
31 }
32 void Enemy::Draw()
33 {
34     currentState->Draw();
35 }
36 ―――――――――――――――――――――――――――――――――――――――――――――――――
```

これで、Enemy の状態に関する責任は IState の派生クラスに委譲され、凝集度の高いクラスに改良されました。

さて、ここまでだと Strategy パターンと大差がないように感じると思います。実際、このままだとカプセル化を行ったものがアルゴリズムなのか状態なのかという違いしかありません。では Startegy パターンと State パターンの違いを見ていきましょう。

2.4 Strategy パターンとの違い

State パターンと Strategy パターンでは概念レベルでは流動的な要素がオブジェクトの状態とアルゴリズムという違いがあります。では実装レベルでの違いはなんでしょうか。

State パターンは状態を表すパターンであるため、頻繁に状態が切り替わることが考えられます。そのため、頻繁に状態の new、delete が発生することになります。一般的に処理が重いメモリアロケーションが発生する new/delete はソフトウェアの品質を損ねることになります。特に高いリアルタイム性が求められるゲームプログラムにおいて、頻繁なメモリ確保はご法度です。そのため、State パターンを使用する場合頻繁に new/delete を行うのではなく、状態に切り替わった時に Enter、Leave 関数を呼び出して初期化・終了処理を行う方法を検討すべきです。では実装を見ていきましょう。まず、IState クラスに Enter、Leave を追加します。

```
class IState{
public:
	virtual void Update() = 0;
	virtual void Draw() = 0;
	virtual void Enter() = 0;
	virtual void Leave() = 0;
};
```

各種状態を表す派生クラスは Enter、Leave 関数に正しく初期化、終了処理を実装します。

```
//徘徊中のステートクラス。
class StateSearch : public IState{
public:
	void Update();
	void Draw();
	void Enter();
	void Leave();
```

```
};
//プレイヤーを追跡中のステートクラス。
class StateTracking : public IState{
public:
	void Update();
	void Draw();
	void Enter();
	void Leave();

};
//バトル中のステートクラス。
class StateBattle : public IState{
	void Update();
	void Draw();
	void Enter();
	void Leave();

};

//逃走中のステートクラス。
class StateEscape : public IState{
public:
	void Update();
	void Draw();
	void Enter();
	void Leave();
};
```

Enemy クラスは各種状態のメンバ変数を保持します。

```
class Enemy{
	StateSearch	stateSearch;
	StateTracking	stateTracking;
	StateBattle	stateBattle;
	StateEscape	stateEscape;
	IState*		currentState;
```

```
};
```

Enemy::ChangeState 関数は下記のように変更します。

```
void Enemy::ChangeState(State state)
{
	if( currentState != NULL ){
		//終了関数を呼び出す。
		currentState->Leave();
	}
	if( state == State_Search ){
		currentState = &stateSearch;
	}else if( state == State_Tracking ){
		currentState = &stateTracking;
	}else if( state == State_Battle ){
		currentState = &stateBattle;
	}else if( state == State_Escape ){
		currentState = &stateEscape;
	}
	//初期化関数を呼び出す。
	currentState->Enter();
}
```

これで状態が切り替わるたびに発生していた new/delete が除去できます。

State パターンは Strategy パターンの拡張パターンと言えます。注意して欲しいのは Strategy パターンとの違いは、new/delete を行わないということではありません。new/delete を行う方が好ましい場合はそちらを行ってください。最も重要な違いはカプセル化する要素が状態なのかアルゴリズムなのか？という点です。ただ、状態をカプセル化する場合は頻繁に切り替えが発生することが多いため、実装には注意を払ってください。