**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();

};

//逃走中のステートクラス。

class StateEscape : public IState{

public:

void Update();

void Draw();

};

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。

void Enemy::ChangeState(State state)

{

if( currentState != NULL ){

delete currentState;

}

if( state == State\_Search ){

currentState = new StateSearch;

}else if( state == State\_Tracking ){

currentState = new StateTracking;

}else if( state == State\_Battle ){

currentState = new StateBattle;

}else if( state == State\_Escape ){

currentState = new StateEscape;

}

}

EnemyクラスのUpdateとDraw関数は下記のようになります。

void Enemy::Update()

{

currentState->Update();

}

void Enemy::Draw()

{

currentState->Draw();  
}

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。