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。

3 Template Method パターン

3.1 概要

この章では 3D モデルの表示プログラムを例に挙げて Template Method パターンについて見ていきましょう。

3.2 アルゴリズムの細部の違いをカプセル化する

Template Method パターンも Strategy パターンと同様にアルゴリズムの処理を入れ替えるパターンとなります。Strategy パターンとの違いは、Strategy パターンはアルゴリズムをごっそりと入れ替えるパターンであったのに対して、Template Method パターンはアルゴリズムの骨格は変えずに細部を入れ替えるパターンとなります。では、下記の仕様のモデルクラスの実装を考えてみてください。

Model クラスは DirectX の API を使用しており、Draw メンバ関数をコールされることによって、3D モデルが描画される。3D モデルデータはスキン付きのモデルデータとスキンなしのモデルデータが混在しており、前者は DrawSkinModel 関数、後者は DrawNonSkinModel 関数を呼び出すことで描画できるものとする。Model クラスには HasSkinWeight というスキンあり、なしを判断するメンバ関数があり、その関数が true を返せばスキンあり、false を返せばスキンなしだと判定することができる。

では、上記の仕様を実現する Model クラスの定義を見てみましょう。

```
class Model{
public:
	void Draw();
private:
	void DrawSkinModel();
	void DrawNonSkinModel();
	bool HasSkinWeight ();
};
```

public メンバ関数に Draw。そして、private メンバ関数に DrawSkinModel、DrawNonSkinModel、HasSkinWeight を定義しています。では、Model::Draw 関数を見てみましょう。

```
void  Model::Draw()
{
    if(HasSkinWeight()){
        //スキンがある。
        DrawSkinModel();
    }else{
        //スキンがない。
        DrawNonSkinModel();
    }
}
```

HasSkinWeight 関数が true を返してきたら、DrawSkinModel を呼び出して、false を返してきた場合は DrawNonSkinModel を呼び出しています。特に何も問題がない実装です。
さて、このようなモデルクラスを作っていた時に、下記のような仕様が追加されたことを考えてみてください。

今の Model クラスと同様の仕様の OpenGL を使用したモデル表示処理を実装して欲しい。

DirectX と OpenGL では３D モデルを表示させるための API が異なります。そのため、DirectX 版と OpenGL 版とでは DrawSkinModel、DrawNonSkinModel、HasSkinWeight 関数の中身の実装が異なるはずです。

さて、あなたはどのようにこれを実装するでしょうか？例えば、ModelOpenGL といった新しいクラスを作成する場合を考えてみましょう。

```
class ModelOpenGL{
public:
    void Draw();
private:
    void DrawSkinModel();
    void DrawNonSkinModel();
    bool HasSkinWeight ();
};
```

Draw 関数は次のようになるでしょう。

```
void ModelOpenGL::Draw()
{
    if(HasSkinWeight()){
        //スキンがある。
        DrawSkinModel();
    }else{
        //スキンがない。
        DrawNonSkinModel();
    }
}
```

これで DrawSkinModel、DrawNonSkinModel、HasSkinWeight 関数を OpenGL の API を使用して実装すれば完成です。

さて、ここで ModelOpenGL::Draw 関数と Model::Draw 関数の実装に全く違いがないことに気づいてもらえたでしょうか。
Mode::Draw 関数と ModelOpenGL::Draw 関数はアルゴリズムの骨格は同じで、細部の DrawSkinModel、DrawNonSkinModel、HasSkinModel の中身の実装にしか差異がありません。そして、この差異の部分が今回の流動的な要素となります。では、この流動的な要素をカプセル化していきましょう。まず、AbstractClass 役になる IModel クラスを作成しましょう。

```
cass IModel {
public:
    void Draw();
private:
    virtual void DrawSkinModel() = 0;
    virtual void DrawNonSkinModel() = 0;
    virtual bool HasSkinWeight() = 0;
};
```

IModel クラスは純粋仮想関数の DrawSkinModel、DrawNonSkinModel、HasSkinWeight を定義しているので抽象クラスです。では IModel::Draw 関数を見てみましょう。

```
void IModel::Draw()
{
    if(HasSkinWeight()){
        DrawSkinModel();
    }else{
        DrawNonSkinModel();
    }
}
```

IModel::Draw 関数は HasSkinWeight 関数が true を返してきたら DrawSkinModel 関数を呼び出して、false を返してきたら DrawNonSkinModel 関数を呼び出すというアルゴリズムの骨格を実装しています。続いて、これらの純粋仮想関数を実装している派生クラスを実装する必要があります。では、ConcreteClass 役になる、ModelDirectX クラスと ModelOpenGL クラスを見てみましょう。

```
//DirectX を使用したモデル表示クラス。
class ModelDirectX : public IModel{
private:
    void DrawSkinModel();
    void DrawNonSkinModel();
    bool HasSkinWeight ();
};
//OpenGL を使用したモデル表示クラス。
class ModelOpenGL : public IModel{
private:
    void DrawSkinModel();
    void DrawNonSkinModel();
    bool HasSkinWeight ();
};
```

あとはユーザーが用途に応じて適切なモデルクラスのインスタンスを生成して、Draw 関数を呼べばいいことになります。例えば次のような使い方です。

```
class Player{
public:
    Player();
    ~Player();
    void Draw();
private:
    IModel* model;
};

Player::Player()
{
    if(…){
        //OpenGL を使用する場合は ModelOpenGL のインスタンスを生成する。
        model = new ModelOpenGL();
    }else if(…){
        //DirectX を使用する場合は ModelDirectX のインスタンスを生成する。
        model = new ModelDirectX();
    }
}
void Draw()
{
    model->Draw();
}
```

3.3 まとめ

TemplateMethod パターンはアルゴリズムの実装を変えるという観点から見ると Strategy パターンと同じになります。Strategy パターンとの違いは、アルゴリズムの骨格が同じである場合は全てをごっそりと入れ替えるのではなく、細部を入れ替えるというものです。関数(Method)の雛形(Template)があるパターンであるため、TemplateMethod パターンと呼ばれています。

4 Observer パターン

4.1 概要

この章ではオブジェクトの状態を観察するプログラムで使われる Observer パターンについて見ていきましょう。

4.2 オブジェクトの変化を観察するデザインパターン

Observer パターンはあるオブジェクトの変化を観察するデザインパターンです。例えばパックマンの敵キャラクターの AI で考えてみましょう。パックマンの敵キャラクターはプレイヤーがパワーアップアイテムを食べて無敵状態になると、プレイヤーから逃げ出すという仕様があります。これは「敵キャラクターはプレイヤーの状態変化を**観察していて**、プレイヤーの状態が無敵になったら逃げ出す」ということです。

ではこの処理を実現するための簡単な実装例を見てみましょう。また、敵キャラクターはステートパターンを使用して FSM を実装しているものとします。

```
void Enemy::Update()
{
            .
            .
            .
    if(   player->IsChangeStatusTrigger()
        && player->GetStatus() == PLAYER_STATUS_INVINCIBLE
    ){
        //プレイヤーの状態が変わった And プレイヤーの状態が無敵状態になったので
        //逃げ状態に遷移する。
        status = new EscapeState;
    }

            .
            .
            .
```

player->IsChangeStatusTrigger()はプレイヤーの状態が変わった 1 フレームだけ true を返えします。player->GetStatus()は現在の状態を返します。つまり、上記のコードはプレイヤーの状態が無敵状態に変わったフレームで敵が逃げ状態に遷移するコードです。

とてもシンプルで分かりやすい実装なのではないでしょうか。小さなプログラムであればこれで十分でしょう。小さなプログラムにわざわざ複雑性を盛り込む必要はありません。ただし、このプログラムがある程度大きくなったときにいくつかの問題が想定されます。では次の節から想定される問題点を見ていきましょう。

4.2.1 仕様変更や追加によってオブジェクトの変化を見逃す可能性が高くなる

例えば下記のような仕様追加を考えてみてください。

「パックマンがあるアイテムを食べると、敵キャラクターの動きを停止することができるようになる。」

敵キャラクターの動きを止める方法はいくつか考えられますが、今回は下記のように Update 関数で return するように実装するようにしました。

```
void Enemy::Update
{

    if( timeStop ){
        //時間停止されていたら return する。
        return ;
    }
        .
        .
        .
    if( player->IsChangeStatusTrigger()
        && player->GetStatus() == PLAYER_STATUS_INVINCIBLE
    ){
        //プレイヤーの状態が変わった And プレイヤーの状態が無敵状態になったので
        //逃げ状態に遷移する。
        status = new EscapeState;
    }
        .
        .
        .
```

さて、上記のコードには次のような不具合があるのが分かるでしょうか？

「時間停止中にプレイヤーが無敵状態になると逃げ状態に遷移できない。」

timeStop 変数が true になっていると Update 関数を return する実装になっているためプレイヤーの状態変化の瞬間が監視できません。player->IsChangeStatusTrigger()は状態が変化した 1 フレームだけ true を返す関数だったことに注意してください。その 1 フレームを逃してしまうと false を返すようになります。

4.2.2 状態変化の条件が変更された場合の修正箇所が多くなる

敵キャラクターがプレイヤーから逃げ出す条件に仕様が追加されるということはよくあることです。

「プレイヤーの HP が一定値以下だったら逃げ出す」

その場合は下記のように実装を変更することが考えられます。

```
void Enemy::Update()
{
                ・
                ・
                ・
    if(   player->IsChangeStatusTrigger()
        && player->GetStatus() == PLAYER_STATUS_INVINCIBLE
        && player->GetHP() <= ESCAPE_HP
    ){
        //プレイヤーの状態が変わった And プレイヤーの状態が無敵状態になったので
        //逃げ状態に遷移する。
        status = new EscapeState;
    }

                ・
                ・
                ・
```

さて、この条件が記述されているのは Enemy::Update 関数だけではないかもしれません。Enemy::Draw 関数に記述されているかもしれませんし、敵が逃げ出すと BGM も変わるかもしれませんね。そのため、サウンドプログラムにも上記のような条件式があるかもしれません。このように多数の箇所に状態変化を監視するコードが書かれていると漏れることなくそれを修正する必要があります。*1

4.2.3 パフォーマンス的なデメリット

Enemy の Update 関数でプレイヤーの状態変化を監視しているため、例えば Enemy が 1000 体存在している場合、すべて同じ結果を返すのに 1000 回の条件判定が行われています。条件判定の処理によっては無視できないパフォーマンスの低下を招くことでしょう。

*1 もちろん、この複雑な条件自体を関数化することでかなりマシなコードにはなります。それでもその関数に渡す引数が変わるなど変更は発生するものです。

4.3 Observer パターンを使用した場合の実装例

前節で見たいくつかの問題点は、Observer パターンを使用しなくても解決することはできます。ここで紹介する Observer パターンは上記のような問題点の解決作の一つとして考えてください。そしてこのデザインパターンは色々なゲームエンジン、GUI システムで採用されている優秀なデザインパターンであるため、多くのケースでより良い解決方法になります。

今回実装する Observer パターンには下記の登場人物がいます。

・状態変化を監視されるオブジェクト(ConcreteSubject 役)

今回のパックマンの例ですと、状態変化を開始されるオブジェクトはプレイヤーです。

・状態変化を通知されるオブジェクト(ConcreteObserver 役)

今回のパックマンの例ですと、状態変化を通知する必要があるのはエネミー、サウンドプログラムなどです。

・状態変化を通知される処理のインターフェースクラス(Observer 役)

状態変化の変更を受け取るオブジェクトがエネミー、サウンドプログラムなどいくつか考えられるため、通知するためのインターフェースとなる Observer クラスが必要になります。

では実装を見ていきましょう。

まず、状態変化を通知されるインターフェースクラスの Observer 役を見てみましょう。

```
class IPlayerStateListener{
public:
    //無敵状態に遷移したことを通知。
    virtual void NotifyChangeStateInvincible () = 0;
};
```

Player クラスは状態変化を通知する必要のあるオブジェクトのリストを保持しています。

```
class Player{
      ・
      ・
      ・
private:
    std::list<IPlayerStateListener*>    listenerList;          //!<プレイヤーの状態の監視者。
};
```

そして、オブサーバーを追加するための AddStateListener 関数と RemoveStateListener 関数を追加します。

```
void Player:: AddStateListener (IPlayerStateListener* listener )
{
      listenerList.push_back( listener );
}

void Player:: RemoveStateListener ( IPlayerStateListener* listener )
{
      //削除するリスナーを検索。
```

```
        auto delObj = std::find(listenerList.begin(), listenerList.end(), listener);
        if(delObj !=  listenerList.end()){
            //削除するオブサーバーが見つかった。
            listenerList.erase(delObj);
        }
}
```

Player::ChangeState 関数では、自分の状態を監視してリスナーに対して通知を行っています。

```
void Player::ChangeState( int nextState )
{
        if( nextState == PLAYER_STATUS_INVINCIBLE){
           //次の状態が無敵状態なら、リスナーに通知を行う。
            for(auto listener : listenerList ){
                listener->NotifyChangeStateInvincible();
            }
        }
            .
            .
            .
```

続いて、Enemy クラスを見ていきましょう。Enemy クラスはプレイヤーの状態変化を監視しますので、Obserber 役の IPlayerStateListener クラスを継承します。

```
class Enemy : public IPlayerStateListener{
            .
            .
            .
    void NotifyChangeStateInvincible ()
            .
            .
            .
};
```

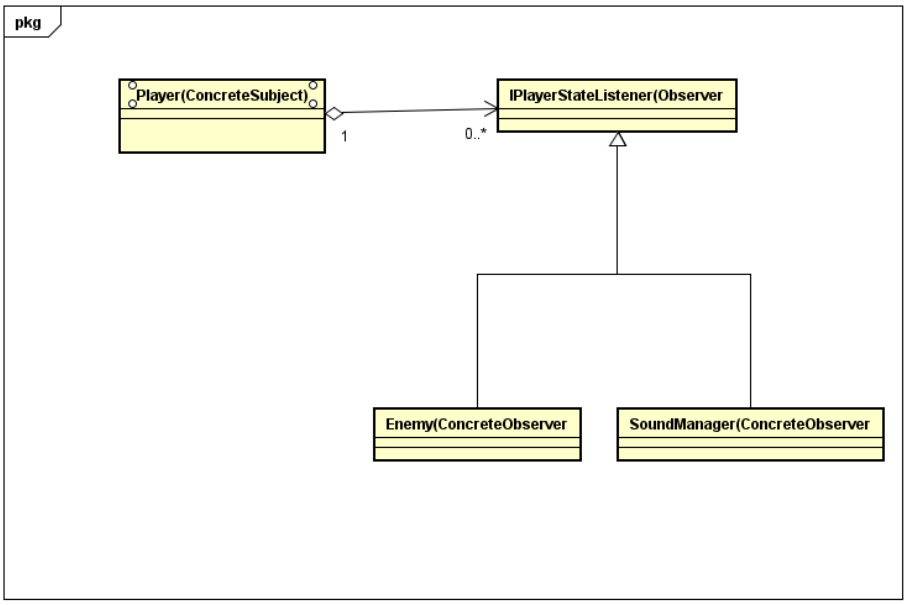
Enemy のコンストラクタとデストラクタで Player にリスナーを追加します。

```
//コンストラクタ。
Enemy::Enemy()
{
        player->AddPlayerStateListener(this);
}
//デストラクタ。
Enemy::~Enemy()
{
        player->RemovePlayerStateListener(this);
}
```

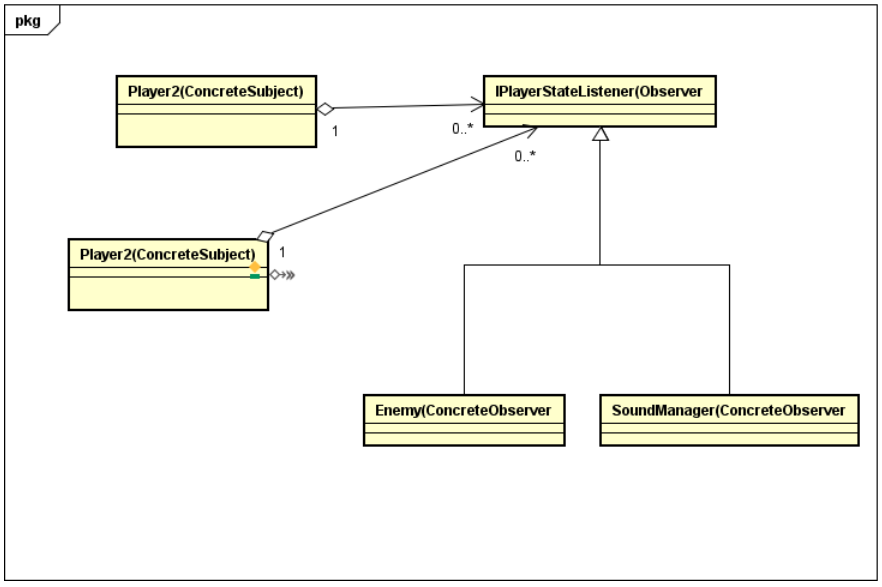
これで、Player の状態が変更されたら Enemy クラスの NotifyChangeStateInvincible が自動的にコールされるようになりました。Player の状態を調べる責任は Player クラスに移譲され凝集度が上がりました。Observer パターンを採用する前はいたるところに点在していたコードが一箇所にまとまることになります。

4.4 Subject 役の抽象化

前節までの例では Player の状態を Enemy やサウンドのプログラムが監視しているというものを考えていました。クラス図にすると下記のようなものです。



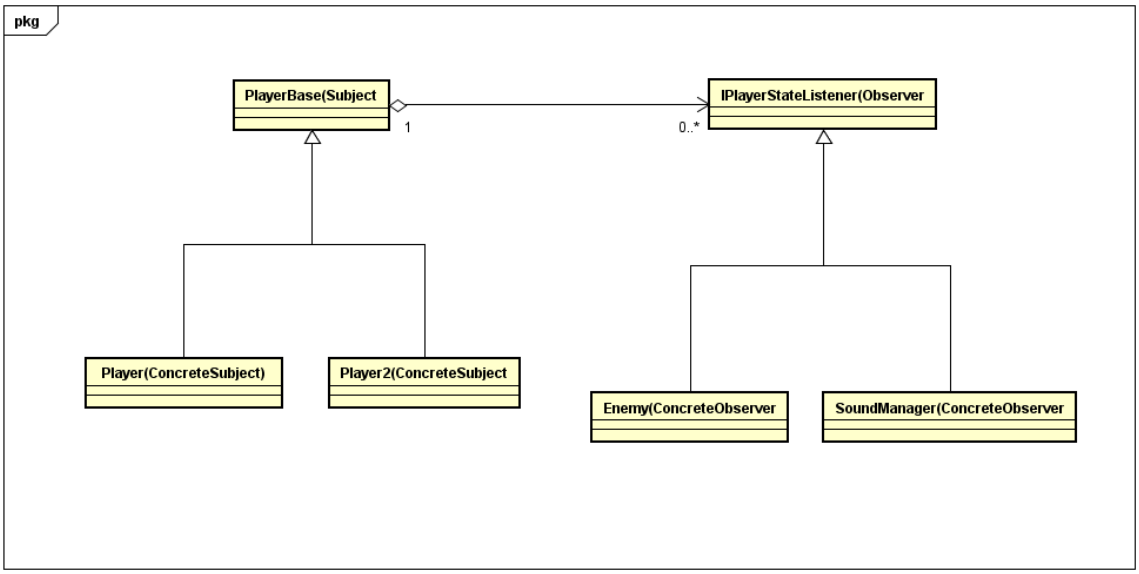
では、さらに考えを進めてみましょう。パックマンの仕様が 2P 対戦になった場合を考えてみてください。そして、プレイヤー 2 は少し性能が違うパックマンを操作するという仕様だったとします。その場合プログラマは Player2 というクラスを新しく作成して、下記のような設計にするかもしれません。



Enemy クラスと SoundManager クラスは Player と Player2 両方の状態の変化を監視する必要があります。そのため Player2 も ConcreteSubject 役のクラスになっています。Enemy クラスのコンストラクタとデストラクタには下記のようなコードが追加されます。

```
//コンストラクタ。
Enemy::Enemy()
{
  player->AddPlayerStateListener(this);
  player2-> AddPlayerStateListener(this);
}
//デストラクタ。
Enemy::~Enemy()
{
  player->RemovePlayerStateListener(this);
  player2->RemovePlayerStateListener(this);
}
```

では、もしこのゲームが４人対戦になったらどうでしょうか。そして全てのプレイヤーが特色の違う仕様だった場合、また同様に Enemy のコンストラクタとデストラクタを変更する可能性があります。もちろんすべて手動で追加を行っていってもかまいませんが、より良い設計として、下記のように PlayerBase といった基底クラスを作成してポリモーフィズムを活用することが一つの案として考えられます。



PlayerBase という基底クラスが作成されたため、プレイヤーは下記のように管理できます。

```
Player player;
Player2 player2;
PlayerBase* playerList[2];

void Init()
{
    playerList[0] = &player;
    playerList[1] = &player2;
}
```

Observer のリストは PlayerBase が保持するようになっているため、Enemy クラスのコンストラクタとデストラクタは下記のように実装できます。

```
//コンストラクタ。
Enemy::Enemy()
{
  for( PlayerBase* p : playerList){
    p->AddPlayerStateListener(this);
  }
}
//デストラクタ。
Enemy::~Enemy()
{
  for( PlayerBase* p : playerList){
    p->RemovePlayerStateListener(this);
  }
}
```

1

2 このように、ConcreteSubject 役にも抽象レイヤーとなる Subject 役を用意してやることに
3 よって、今後 PlayerBase を継承するクラスが増えたとしても ConcreteObsever クラスに変
4 更が発生することはなくなりました。