Chapter-1 オブジェクト指向

1.1 はじめに

私が初めて学んだ言語はC言語でした。C言語は「手続き型言語」と呼ばれている言語です。C言語の次に私が学んだ言語はC++言語です。C++言語はC言語にクラスなどのオブジェクト指向パラダイムを付与したスーパーセット言語です。 私が初めてC++言語を学んだ時に、オブジェクト指向って必要なのか？と思ったことを覚えています。別にクラスなんて使わなくても、ゲームは実装できるじゃないか？なぜ、こんなものを勉強しなくてはいけないのだ？と不思議に感じました。純粋なオブジェクト指向言語である、Javaなどからオブジェクト指向に触れた人は、そもそもこんな感想は持たないかもしれませんが、多くのC++の入門書では、最初にC言語の部分を学んでいき、途中からクラスなどを利用した、C++のオブジェクト指向の部分を学んでいくことになります。そのため、私と同じような感想を持った人も多いのではないかと思います。本書では、なぜオブジェクト指向を使うのか？オブジェクト指向を使うと何がいいのか？これらを具体的なゲームでのデザインパターンの利用とともに学んでいきます。

また、本書で扱うデザインパターンはGoFが「オブジェクト指向における再利用のためのデザインパターン」で提唱したデザインパターンと、Robert Nystromが「Game Programming Patterns」で提唱したデザインパターン、そして結城 浩が「Java言語で学ぶデザインパターン入門 マルチスレッド編」で紹介された、マルチスレッドでのデザインパターン、これら３つの書籍で紹介されたデザインパターンからいくつかを学んでいきます。

1.2 なぜオブジェクト指向で作るのか？

オブジェクト指向について見ていく前に、なぜオブジェクト指向で作るのか？についてみていこうと思います。結論から言うと、オブジェクト指向でプログラミングをした方が、ある程度大きなソフトウェアであれば、開発をするのが楽になるからです。私はオブジェクト指向プログラミングを行った際に、大きなソフトウェアの開発効率が上がる理由として、次の３点があると考えています。

1. 可読性の向上
2. 保守性の向上
3. 再利用性の向上

1.2.1 可読性の向上

可読性とはプログラムの読みやすさです。適切なオブジェクト指向設計を行うと、小さな意味のある単位、でソースファイルは分割されていきます。例えば、Player.cpp、Enemy.cpp、Boss.cpp、PlayerAnimation.cppなどです。すると、どのソースファイルに目的のプログラムが書かれているのかを見つけ出すことが容易になります。先ほどの例で行けば、プレイヤーに関する処理はPlayer.cpp、敵キャラクターに関する処理はEnemy.cpp、プレイヤーのアニメーションに関する処理はPlayerAnimation.cppのように。また、この本の本題である、デザインパターンを使うことで、凝集度が高く、結合度が低いプログラムを設計することが可能になります。凝集度が高く、結合度が低いということがどういうことかというと、簡潔に説明すると「あるモジュールの処理に、そのモジュールに関係が薄いプログラムが少ないということです。」ここで言っているモジュールはクラスに置き換えて読んでもらって構いません。例えば、敵キャラクターのソースファイルのEnemy.cppが、凝集度が高く、結合度が低い実装になっていると、Enemy.cppには、敵キャラクターの処理に関係がないプレイヤーのプログラムだとか、ギミックのプログラムなどがあまり出てこない、ということです。です　　ので、敵キャラクターのソースファイルを読んでいるプログラマは、目的の処理のみに集中してコードを読むことができるようになります。

1.2.2 保守性の向上

保守性とは、プログラムのメンテナス、変更のしやすさを指しています。保守性は先ほどの可読性と通じる部分もあります。プログラムを意味のある単位で分割ができていると、メンテナンス、変更をする際に、目的となるソースファイルをすぐに見つけることができます。また、保守性はクラスのカプセル化を行うことで、高めることもできます。しばしば、オブジェクト指向設計の入門書にはカプセル化とは、クラスのデータメンバを隠ぺいするものである、というように説明されていますが、カプセル化とは、ありとあらゆるものを隠ぺいすることを指します。例えばそれが、データメンバであったり、実装の詳細であったり、インターフェースであったり。カプセル化を行うことで、ありとあらゆるものの詳細を隠ぺいすることができるようになるため、その詳細をこっそりと変更することが容易になります。隠ぺいしているものを変更するだけなので、変更したことをその他大勢のプログラムに教える必要がなくなるのです。これについてはデザインパターンを見ていく際に、もっと詳細に説明します。

1.2.3 再利用性の向上

オブジェクト指向言語で言われる再利用とは実装の再利用と、設計の再利用の二つを指します。実装の再利用は、既存のプログラムを再利用するというものです。例えば、本書に付属しているMiniEngineなどのクラスは、学生が作成する多くのゲームで利用されることを想定しているため、再利用性の高いプログラムとなっています。実装の再利用は、移譲と継承の二つの選択肢があります。これについては、後ほど詳細に解説しますが、多くの場合で、実装の再利用を行いたい場合は、継承より移譲を選択したほうが良い設計となります。実装の継承はクラスの爆発など多くの問題をはらんでいます。設計の再利用は、本書のテーマであるデザインパターンと強く関連している内容となります。

1.3 オブジェクト指向とは

さて、オブジェクト指向とはいったいどのようなものなのでしょうか。私が初めてC++を学んだ時に、「オブジェクト指向とは、現実世界のモノに着目してプログラミングをすることである」と学びました。現実世界のモノがクラス、そのモノを操作するための命令がメンバ関数になるといった感じです。例えば、レースゲームを作っている際に、車のプログラミングをするのであれば、Carクラスを作成します。また、車を操作するための「ブレーキをかける」、「アクセルを踏む」といった操作をメンバ関数として用意します。これをプログラミングすると、次のようなコードになります。

```
// 車クラス
class Car{
public:
  // ブレーキをかける。
  void Brake();
  // アクセルを踏み込む。
  void Accele();
  // 走らせる。
  void Run();
}
```

このCarクラスは次のコードのように、インスタンスを作成して利用されると思います。

```
Car c;
if( g_pad[0].IsPress( enButtonA ) ){
```

```
    // ゲームコントローラーのAボタンが押されているのでアクセルをかける。
    c.Accele();
  }
  if( g_pad[0].IsPress(enButtonB)){
    // ゲームコントローラーのBボタンが押されているのでブレーキをかける。
    c.Break();
  }
  // 車を走らせる
  c.Run();
```

現実世界のモノをクラスにして、そのモノに対する操作をメンバ関数にするというのは、理解しやすい話なので、入門書などによく書かれているのだと思われます。しかし、私はこの話を聞いたときに、クラスをゲームのプログラムで、作ることはほとんどないのではないか？と考えてしまいました。この話を聞いたときに、私はプログラミング学んで１年ほどで、ちょうどC言語を利用して、小さな2Dゲームを作っていたころだったと思います。そのゲームは、プレイヤーキャラクターが出てきて、マップを徘徊している５体の敵キャラクターをすべて倒したらゲームクリアといったゲームでした。このゲームはC言語で作っていたのですが、C++を利用して作ったらどうなるか？と考えたのです。そうすると、せいぜい３つ程度のクラスしか思いつきませんでした。それは、プレイヤークラス、エネミークラス、背景クラスです。ひょっとすると当時は背景クラスは思いついていなかったかもしれません。この時私は、「これは私が作った小さなゲームだからそうなのかもしれない」と思い、市販のゲームでもいくつのクラスが作れるのか考えてみました。それでも結果に大きな違いはなく、せいぜい、10個程度のクラスしか思いつきませんでした。なぜ、こんなことになってしまったのか？それは「オブジェクト指向とは、現実世界のモノに着目してプログラミングをすることである」という説明のせいです。この説明は、オブジェクト指向に対する、大変狭いものの見方です。これは正確には「オブジェクト指向プログラミングでは、現実世界のモノをプログラミングすることも**も**できる」だったのです。現実世界のモノをプログラミングするというのは、あくまでオブジェクト指向プログラミングの一部であり、イコールではなかったわけです。では、オブジェクト指向の正しい定義は何なのでしょうか。オブジェクト指向プログラミングとは、ソフトウェアを作るうえで実装する必要のある、概念、機能を実装するための、振る舞いとデータを一つにまとめてプログラミングをしていくことです。現実世界のモノに着目するわけではないのです。ソフトウェアを作る際に、解決する必要のある問題領域、概念、機能に着目して、プログラミングを行うのです。例えば、先ほどの私が作ったゲームの例で行きましょう。そのゲームには、敵キャラクターがいたので、敵のプログラムを実装する必要があります。ですので、Enemyクラスなどを実装するのは容易に想像できます。さて、この敵キャラクターを実装するためには、どのような機能を実装する必要があるのでしょうか。この敵キャラクターはプレイヤーキャラクターを発見していないときは、マップをランダムに徘徊していました。これも解決する必要がある問題領域です。このゲームでは、敵キャラクターはマップをランダムに徘徊できる必要があるのです。そこで、EnemyMoveRandomのようなクラスを作ることが考えられます。他にはどうでしょうか。おそらく敵キャラクターの絵を画面に表示する機能も必要だったはずです。そこで、EnemyRendererというクラスを作ることも考えられます。敵キャラクターをアニメーションさせる必要もあったかもしれません。そうすると、EnemyAnimatorというクラスが必要になるかもしれません。ここまでに上げてきた、EnemyMoveRandom、EnemyRenderer、EnemyAnimatorは「現実世界のモノに着目してプログラミングする」という考え方に縛られていると生まれてこないクラスです。これらのクラスは、オブジェクト指向とは「オブジェクト指向とは、実装する必要のある、概念、機能をオブジェクトととらえてプログラミングすることである」と大きくとらえることで、生み出すことができるクラスです。

1.3 コンポーネント指向とオブジェクト指向

オブジェクト指向ほど聞きなじみがないかもしれませんが、コンポーネント指向という考え方があります。コンポーネント指向はソフトウエアを機能ごとに部品として分割し、必要に応じて組み合わせて使うという考え方です。この考え方で設計されているゲームエンジンがUnityです。Unityでは、敵キャラクターを実装したいときに、GameObjectというオブジェクトに、絵を描画するためのMeshRendererコンポーネント、アニメーションを再生するためのAnimatorコンポーネント、GameObjectをランダムに動かすためのRandomMoveコンポーネントなど複数のコンポーネントを組み合わせて、実装していきます。さて、ここで1.2節のEnemyクラスの実装の話を思い出してみてください。同じようなことを言っていることに気づいてもらえたでしょうか？1.2節でも敵キャラクターを実装するために、ランダム移動するEnemyMoveRandomクラスを作るだとか、絵を表示するためにEnemyRendererクラスを作るという話をしたと思います。実はコンポーネント指向と優れたオブジェクト指向が目指している方向はイコールなのです。また、コンポーネント指向は、その部品を容易に変更可能とします。例えば、ランダム移動ではなく、決められた経路上を移動する敵キャラクターを作りたい場合は、RandomMoveコンポーネントの代わりにPathMoveコンポーネントを使えばよいのです。多くのデザインパターン、特にGofが提唱した23個のデザインパターンの多くは、まさにソフトウェアを作る際に、部品を容易に交換可能にすることを目的としたものとなります。例えば、先ほどのUnityのGameObjectとComponentの関係などは、まさにGoFが提唱したBridgeパターンの亜種だと言えます。

1.4 責任の移譲

さて、1.3節で優れたオブジェクト指向とはコンポーネント指向であると説明しました。つまり、優れたオブジェクト指向とは、何かの処理を実現したい場合に、部品を組み合わせて実装していくということになります。では、この観点から考えた時に、何かの処理を実装するときには、どのような部品、機能が必要だろうか？と考えていく必要があります。これを考えるときにはまず、そのクラスがもっている責任について考えてみるのはよいアプローチです。責任について考えるというのは、そのクラスは要求を実現するために、何をする必要があるのか？ということについて考えていくということです。先ほどのマップをランダムに移動するEnemyクラスの実装にについて考えてみましょう。このEnemyクラスを実装するためには、1.2節で考えたように、マップ上をランダムに移動できる責任、絵を正しく画面に表示する責任、アニメーションを正しく再生する責任などがあります。Enemyクラスがクラス分割できていない状態は、これらの処理がすべてEnemyクラスに記述されていることになります。この時、Enemyクラスは大きな責任を背負っていることとなります。このような、大きすぎる責任を背負っているクラスのことを、まるで神のようなクラスだと揶揄して、GODクラスなどと呼びます。現実世界でも同じですが、大きな責任を背負っているものは、その責任の重さで押しつぶされてしまうものです。プログラミングを行うのは人間ですので、この原則はそのままプログラミングにも当てはまります。大きすぎる責任を追っているクラスは、往々にして、クラスの実装の行数が多くなります。そのようなクラスの処理を変更する場合、たとえ小さな変更であっても、多くの処理に影響を与えます。また、その変更の影響範囲を把握するのも困難です。例えば、敵キャラクターの描画処理に関するプログラムを変更した場合にも、その変更による影響範囲を確認するために、キャラクターの描画処理がどこに書かれているのかを把握する必要があります。また、キャラクターの描画処理というのは、アニメーションのプログラムと関連があるかもしれません。そうなると、アニメーションのプログラムが記述されている箇所も把握する必要が出てきます。もし、これが適切にクラス分割されていて、EnemyRendererクラスやEnemyAnimatorクラスなどがあれば、話は簡単です。描画処理はEnemyRendererクラスに記述されているはずですし、アニメーションを制御する処理はEnemyAnimatorクラスに記述されているからです。このように、大きなクラスから、別のクラスに責任を譲り渡すことを移譲といいます。今回のケースで言えば移譲とは単に、Enemyクラスの中で記述されていた描画処理をEnemyRendererクラスに、アニメーション制御に関する処理をEnemyAnimatorクラスに移動させるといったものです。そして、Enemyクラスはこれらのクラスのインスタンスをメンバ変数に持ち、あとの処理はよろしく！というように、それらのクラスに責任をなすりつけるのです。これが責任の移譲です。

1.5 【ハンズオン】Enemyクラスから移動処理を委譲してみよう。

さて、このハンズオンでは、Enemyクラスに実装されている、エネミーの移動処理を、別のクラスに委譲するハンズオンを行っていきます。ですので、ハンズオンを行う前に、もともとのEnemyクラスの実装を確認しておきましょう。Sample_01_01/Sample_01_01_Before.slnを立ち上げてください。立ち上がったらF5を押して、プログラムを実行してみてください。すると図1.1のようなゲームが実行されます。
[図1.1]

point-1 移動処理に関するメンバ変数。

では、今回ポイントとなるプログラムを確認してみましょう。Enemy.before.hを開いてください。リスト1.1のプログラムは移動処理に関するメンバ変数です。
[リスト1.1 Enemy.before.h]

```
// point-1 移動処理に関するメンバ変数
Vector3 m_moveSpeed;    // 移動速度。
int m_moveTimer = 0;    // 移動タイマー。
// point-1 ここまで
```

移動速度と移動に関係するタイマーがEnemyクラスのメンバ変数として保持されています。この二つのメンバ変数はこの後のハンズオンで、新しく追加されるEnemyRandomMoveクラスに移動します。

point-2 ランダム移動に関する処理。

続いて、ランダム移動に関する処理の実装を見ていきましょう。この処理もEnemyクラスに記述されています。Enemy.before.cppを開いてください。リスト1.2のプログラムが120フレームごとにランダムに移動速度を決定して、その移動速度をエネミーの座標に足し算して、エネミーを動かしているプログラムです。
[リスト1.2 Enemy.before.cpp]

```
// point-2 ランダム移動に関する処理
// ランダムに移動速度を決定。
if (m_moveTimer % 120 == 0) {
  // 120フレームで移動方向を変更する。
  std::random_device rd;
  m_moveSpeed.x = (rd() % 100) / 100.0f;
  m_moveSpeed.x -= 0.5f;
  m_moveSpeed.z = (rd() % 100) / 100.0f;
  m_moveSpeed.z -= 0.5f;
  // 正規化する。
  m_moveSpeed.Normalize();
  // 移動速度は0.3。
  m_moveSpeed *= 0.3f;
}
// 移動タイマーをインクリメント
m_moveTimer++;
// 移動速度を座標に加算。
m_position += m_moveSpeed;
// point-2 ここまで
```

プログラム自体はたいしたプログラムではないので、特別な解説は入れません。重要なのは、今見ていったEnemyクラスの処理を、新たに追加するEnemyRandomMoveクラスに移行して責任を委譲するということです。

step-1 敵キャラクターのランダム移動クラスの宣言を実装する。

では、エネミーの移動処理の責任を委譲していくハンズオンを実施しましょう。
Sample_01_01/Sample_01_01.slnを開いてください。まずは、ランダム移動クラスの宣言を実装します。
EnemyRandomMove.hを開いてリスト1.3のプログラムを該当するコメントの箇所に入力してください。
[リスト1.3 EnemyRandomMove.h]

```
// step-1 敵キャラクターのランダム移動クラスの宣言を実装する。
class EnemyRandomMove {
public:
    // 移動処理を実行する関数。
    // 引数に移動させる座標の参照を受け取る。
    void Execute(Vector3& pos);
private:
    Vector3 m_moveSpeed;    //【注目】移動速度。
    int m_moveTimer = 0;    //【注目】移動タイマー。
};
```

EnemyRandomMoveのメンバ変数に注目してください。元々のEnemyクラスに定義されていたものと同じ変数が用意されています。

step-2 敵キャラクターのランダム移動クラスの定義を実装する。

続いて、実際にEnemyを移動させる処理を実装していきます。EnemyRandomMove.cppを開いて。リスト1.4のプログラムを入力してください。
[リスト1.4 EnemyRandomMove.cpp]

```
// step-2 敵キャラクターのランダム移動クラスの定義を実装する。
void EnemyRandomMove::Execute(Vector3& pos)
{
    // ランダムに移動速度を決定。
    if (m_moveTimer % 120 == 0) {
        // 120フレームで移動方向を変更する。
        std::random_device rd;
        m_moveSpeed.x = (rd() % 100) / 100.0f;
        m_moveSpeed.x -= 0.5f;
        m_moveSpeed.z = (rd() % 100) / 100.0f;
        m_moveSpeed.z -= 0.5f;
        // 正規化する。
        m_moveSpeed.Normalize();
        // 移動速度は0.3。
        m_moveSpeed *= 0.3f;
    }
    // 移動タイマーをインクリメント
```

```
    m_moveTimer++;
    // 移動速度を座標に加算。
    pos += m_moveSpeed;
}
```

見てもらえば分かるように、この処理も元々のEnemyクラスに実装されていたものとほとんど同じです。このExecute関数の引数に移動させるエネミーの座標の参照を渡すことで、移動させる処理を実現しています。

step-3 ランダム移動処理のインスタンスをEnemyクラスのメンバ変数に追加。

続いて、ランダム移動処理のインスタンスをEnemyクラスのメンバ変数として追加します。Enemy.hを開いて、リスト1.5のプログラムを入力してください。
[リスト1.5 Enemy.cpp]

```
// step-3 ランダム移動処理のインスタンスをEnemyクラスのメンバ変数に追加。
EnemyRandomMove m_randomMove; // ランダム移動処理。
```

Enemyクラスのメンバ変数から、元々のEnemyクラスに存在していた、移動速度を表すm_moveSpeedと移動タイマーのm_moveTimerがなくなっている点にも注目してください。これらの変数は、EnemyRandomMoveクラスに移動したので、Enemyクラスからはなくなっています。

step-4 ランダム移動処理のインスタンスをEnemyクラスのメンバ変数に追加。

では、これで最後です。エネミーを移動させる処理の実行をEnemyRandomMoveクラスに委譲しましょう。Enemy.cppにリスト1.6のプログラムを入力してください。
[リスト1.6 Enemy.cpp]

```
// step-4 移動処理をEnemyRandomMoveに委譲する。
m_randomMove.Execute(m_position);
```

入力出来たら実行してみてください。Sample_01_01_Before.slnを実行したときと同じように、エネミーが移動してたら完成です。

さて、いかがでしょうか。委譲といっても何も難しいことはありません。なんだ、こんな簡単なことを何を偉そうに語っているんだと思われた方もいるかもしれません。しかし、このなんてことのない、簡単な考え方が、非常に重要なことなのです。今回実装したプログラムが、なんてことない普通のプログラムに感じた方はその感覚を持ったまま次に進んでいってください。