Chapter-10 UpdateMethodパターン

10.1【ハンズオン】UpateMethodパターンを実装する。

　では、サンプルプログラムのUpdateMethod_01を利用して、UpdateMethodパターンを実装していきます。UpdateMethod_01/UpdateMethod_01.slnを開いてください。

step-1 ゲームワールドクラスの宣言を作成する。

　まずは、ゲームの世界を制御するクラスを作ります。このクラスがゲームループの実行や、後で追加するゲームオブジェクトの管理を行います。では、まずはクラス宣言から作っていきましょう。GameWorld.hを開いて、該当するコメントの箇所に次のプログラムを入力してください。
[GameWorld.h]

```
// step-1 ゲームワールドクラスの宣言を作成する。
#include "IGameObject.h"

/// <summary>
/// ゲームワールド
/// </summary>
class GameWorld
{
public:
    /// <summary>
    /// ゲームループを実行
    /// </summary>
    void ExecuteGameLoop();
    /// <summary>
    /// 新しいゲームオブジェクトを追加。
    /// </summary>
    /// <typeparam name="T">作成するゲームオブジェクトの型</typeparam>
    /// <typeparam name="...CtorArgs">コンストラクタに渡す可変長引数</typeparam>
    /// <param name="...ctorArgs">コンストラクタに渡す可変長引数</param>
    /// <returns></returns>
    template<class T, class... CtorArgs>
    T* NewGameObject(CtorArgs... ctorArgs)
    {
        T* newObj = new T(ctorArgs...);
        m_gameObjectList.push_back(newObj);
        return newObj;
    }
    /// <summary>
    /// ゲームオブジェクトを破棄。
    /// </summary>
    /// <param name="go">作成するゲームオブジェクト</param>
    void DeleteGameObject(IGameObject* go)
    {
        // DeleteGameObjectはUpdate関数を呼び出している最中に呼ばれる可能性が非常に高い。
        // そのため、ここでリストからの削除、メモリの解放を行うと不正メモリアクセスが起きる可能性が発生する。
```

```
            // そのため、ここではリストから削除をせずに、削除フラグのみを立てて、遅れて削除を行う。
            go->SetMarkDelete();
        }
    private:
        std::list< IGameObject*> m_gameObjectList;          // ゲームオブジェクトのリスト。
    };
```

このクラスはゲームの世界を表すクラスです。後で実装するIGameObjectを継承したクラスのインスタンスのリストを保持しています。このリストに登録されているオブジェクトのUpdate()関数がゲームループから呼び出されるようになります。

NewGameObject()関数とDeleteGameObject()関数はIGameObjectを継承したゲームオブジェクトの生成と削除に関する関数です。NewGameObject()関数は任意のゲームオブジェクトを生成できる必要があるので、テンプレート関数として定義されています。また、任意のゲームオブジェクトクラスのコンストラクタに可変長個の引数を渡せるようにしたかったので、C++11で追加された可変長引数テンプレートも使用しています。DeleteGameObject()関数はコメントにも記載している通り、安全性のために遅延削除を行う実装になっています。

step-2 ゲームワールドクラス定義を実装する。

では続いて、ゲームワールドクラスの定義を実装していきます。GameWorld.cppを開いて、該当するコメントの箇所に次のプログラムを入力してください。

[GameWorld.h]

```
// step-2 ゲームワールドクラス定義を実装する。
void GameWorld::ExecuteGameLoop()
{
    while (true) {
        IGameObject::UpdateArg updateArg;
        updateArg.world = this;
        updateArg.frameDuration = 1.0f / 60.0f; // 60fps固定

        //【注目】登録されているオブジェクトの更新関数を呼び出す。
        for (auto go : m_gameObjectList) {
            go->Update(updateArg);
        }

        //【注目】削除フラグが立っているゲームオブジェクトを破棄する。
        m_gameObjectList.remove_if([&](IGameObject* obj) {
            if (obj->IsMarkDelete()) {
                delete obj;
                return true;
            }
            return false;
        });
        // 垂直同期をエミュレート。
        Sleep(16);
    }
}
```

GameLoopクラスのNewGameObject()関数とDeleteGameObject()関数はすでにヘッダーファイルでインライン関数として実装しているので、cppファイルで実装したのはゲームループの実行関数のみです。ExecuteGameLoop()関数の中で登録されているゲームオブジェクトの更新関数の呼び出しを行っている箇所と、削除フラグを見てゲームオブジェクトを破棄している箇所に注目してください。この部分がまさにUpdateMethodパターンのキモとなる部分です。ゲームワールドに登場してくるゲームオブジェクトカプセル化され隠蔽されているため、GameWorldクラスの中には各オブジェクトの詳細な実装は定義されていません。GameWorldクラスは各ゲームオブジェクトに対して、Update()関数を呼び出して、状態を更新しろという命令を出しているだけです。ゲームオブジェクトの詳細な実装とGamewWorldの実装が完全に分離されているため、各種ゲームオブジェクトの実装の変更による影響をうけなくなっています。

step-3 ゲームオブジェクトの基底クラスを宣言する。

続いて、ゲームオブジェクトの基底クラスを宣言します。IGameObject.hを開いて、次のプログラムを入力してください。 [IGameObject.h]

```
/// <summary>
/// ゲームオブジェクトの基底クラス。
/// </summary>
class IGameObject
{
public:
    /// <summary>
    /// 更新関数に渡す引数。
    /// </summary>
    struct UpdateArg {
        GameWorld* world;       // ゲームワールド
        float frameDuration;    // 1フレームの間隔(単位:秒)。
    };
    /// <summary>
    /// 毎フレーム呼ばれる更新関数。
    /// </summary>
    virtual void Update(const UpdateArg& arg) = 0;
    /// <summary>
    /// 削除の印を付ける。
    /// </summary>
    void SetMarkDelete()
    {
        m_isDelete = true;
    }
    /// <summary>
    /// 削除の印がついている？
    /// </summary>
    /// <returns></returns>
    bool IsMarkDelete() const
    {
        return m_isDelete;
    }
private:
```

```
    bool m_isDelete = false;    // 削除フラグ。
};
```

step-4 GameWorldクラスを利用する。

step-3までの実装で、生成したゲームオブジェクトの自動アップデート機能を持っているゲームワールドクラスが完成しました。では、ゲームワールドクラスを利用するコードを記述しましょう。main.cppを開いて、該当するコメントの箇所に次のプログラムを入力してください。 [main.cpp]

```
// step-4 GameWorldクラスを利用する。
// ゲームワールドクラスのオブジェクトを定義。
GameWorld gameWorld;
// ゲームループを実行。
gameWorld.ExecuteGameLoop();
```

さて、これでUpdateMethodパターンを利用できるゲームワールドクラスは作成できました。しかし、まだゲームオブジェクトが一切実装されていないので、実行しても何もおきません。では、次のハンズオンで簡単な敵キャラクターのゲームオブジェクトを実装していきましょう。

10.2【ハンズオン】UpateMethodパターンを利用する。

10.1のハンズオンでUpdateMethodパターンを利用したゲームワールドを実装しました。では、10.2ではUpdateMethodパターンを利用して、簡単なゲームプログラムを実装してみましょう。今回作成するゲームは、一定時間で骸骨のモンスターが現れるだけのゲームです。では、
UpdateMethod_01/UpdateMethod_01.slnを開いてください。

step-1 IGameObjectを継承して、インゲームの処理を制御するクラスを宣言する

まずはインゲームを制御するInGameクラスを追加しましょう。このクラスの仕事は一定時間で骸骨のモンスターをスポーンすることです。では、InGame.hを開いて、該当するコメントの箇所に次のプログラムを入力してください。
[InGame.h]

```
// step-1 IGameObjectを継承して、インゲームの処理を制御するクラスを宣言する。
#include "IGameObject.h"

// インゲームクラス宣言
class InGame : public IGameObject
{
public:
    // 毎フレーム呼ばれる更新処理
    void Update(const UpdateArg& arg) override;
private:
    float m_spawnTimer = 0.0f;      // 敵キャラクターのスポーンタイマー。(単位:秒)
    int m_spawnSkeletonCount = 0;   // スポーンしたスケルトンの数。
};
```

step-2 インゲームの処理を制御するクラス定義を実装する

続いて、InGameクラスの定義を実装していきましょう。InGame.cppを開いて次のプログラムを入力してください。
[InGame.cpp]

```
// step-2 インゲームの処理を制御するクラス定義を実装する。
void InGame::Update(const UpdateArg& arg)
{
    m_spawnTimer += arg.frameDuration;
    if (m_spawnTimer > 5.0f) {
        // 5秒経過でスケルトンを生成する。
        m_spawnSkeletonCount++;
        m_spawnTimer = 0.0f;
        //【注目】スケルトンゲームオブジェクトを生成する。
        arg.world->NewGameObject<Skeleton>(m_spawnSkeletonCount);
    }
}
```

5秒経過すると、骸骨のゲームオブジェクトを生成している点に注目してください。

step-3 ゲームオブジェクトを継承したスケルトンクラスを宣言する

step-3～step-4にかけては骸骨のモンスターを実装していきます。まずは、IGameObjectを継承して、Skeletonクラスの宣言を作成します。Skeleton.hを開いていて、該当するコメントの箇所に次のプログラムを入力してください。
[Skeleton.h]

```
// step-3 ゲームオブジェクトを継承したスケルトンクラスを宣言する。
class Skeleton : public IGameObject
{
public:
    // コンストラクタ
    // no : スケルトンの番号。
    Skeleton(int no);
    // 毎フレーム呼ばれる更新処理。
    void Update(const UpdateArg& arg) override;
private:
    float m_shoutTimer = 0.0f;  // 叫びタイマー
    float m_deathTimer = 0.0f;  // 死亡タイマー。
    int m_no = 0;               // スケルトン番号。
};
```

step-4 ゲームオブジェクトを継承したスケルトンクラスを定義する

続いて、スケルトンクラスの定義を実装します。該当するコメントの箇所に次のプログラムを入力してください。
[Skeleton.cpp]

```
// step-4 ゲームオブジェクトを継承したスケルトンクラスを定義する。
Skeleton::Skeleton(int no)
{
    m_no = no;
    printf("スケルトン%02dが現れた。\n", no);
}
void Skeleton::Update(const UpdateArg& arg)
{
    // 叫びタイマーを加算。
    m_shoutTimer += arg.frameDuration;
    // 死亡タイマーを加算。
    m_deathTimer += arg.frameDuration;
    if (m_shoutTimer > 2.0f) {
        // ２秒経過したので叫ぶ。
        printf("スケルトン%02dが咆哮をあげた。\n", m_no);
        // 叫びタイマーをリセット。
        m_shoutTimer = 0.0f;
    }
    if (m_deathTimer > 10.0f) {
        // １０秒経過したので死亡。
        printf("スケルトン%02dが消滅した。\n", m_no);
        // 自分自身をゲームワールドを削除する。
        arg.world->DeleteGameObject(this);
    }
}
```

step-5 インゲームクラスのゲームオブジェクトを生成。

　では、これで最後です。インゲームクラスのゲームオブジェクトを生成して、インゲームの処理を毎フレーム実行できるようにしましょう。main.cppを開いて次のプログラムを入力してください。
[main.cpp]

```
// step-5 インゲームクラスのゲームオブジェクトを生成。
gameWorld.NewGameObject<InGame>();
```