オブジェクト指向と
ゲームプログラミング

▷　フレームワーク編 － 第１３回　シーンの作成　◁

ゲームの状態にあわせた関数の呼び出し

ゲームでは、ゲームの状態にあわせた処理を行います。たとえば、オープニング、タイトル、メイン、エンディングなどの状態が考えられるとき、これらを処理するための関数を作成しておき、進行状態にあった関数を呼び出します。たとえば、タイトルを処理するTitleScene関数、メイン部分を処理するMainScene関数、エンディングを処理するEndingScene関数を作成しておき、ゲームがどこまで進んでいるかを判断してこれらを呼び出します。
もっとも簡単な実装は、以下のようにゲームの状態を変数に格納しておき、これを判断してどの関数を呼び出すのかを決定する方法です。

```
// ゲーム処理関数
void GameProc()
{
    // GameStateはゲームの状態を格納する変数
    switch(GameState) {
      case 0:       // タイトル
        Title();    // タイトルを処理する関数の呼び出し
        break;

      case 1:       // ステージ１
        Stage1();   // ステージ１を処理する関数の呼び出し
        break;

      case 2:       // ステージ２
        Stage2();   // ステージ２を処理する関数の呼び出し
        break;

      case 9:       // エンディング
        Ending();   // エンディングを処理する関数の呼び出し
        break;
    };
}
```

このようなswitch-case文やif-else文による多分岐は、とても簡単でわかりやすく、ゲーム以外でも多くのプログラムで使用されています。しかし、状態がたくさん考えられる場合は、それにあわせてたくさんの分岐を記述することになります。ゲームの状況が１００ほどある場合、caseやif-elseで１００分岐させることになり、caseやelseが１００個も延々と続くコードになってしまいます。さらに、Windowsアプリケーションでは、アクティブ状態と非アクティブ状態があるので、それぞれの分岐後にさらなる分岐が加わり、全部で２００分岐となってしまいます。また、GameProc関数が呼び出されるたびに、どの関数を呼び出すのかをいちいち判断する必要があるので、効率がよいとはいえません。
オブジェクト指向では、このような多分岐をStateパターンを用いて設計、実装することにより、状態にあった処理を効率的に行うことができます。

Stateパターンによる状態遷移

オブジェクト指向では、「もの」をクラスとして表現しますが、Stateパターンでは「状態」というものをクラスとして表現します。Stateとは「状態(ものごとのありさま)」を意味します。「状態」をクラスとして表現していれば、クラスを切り替えることによって「状態の変化」が表せます。

Stateパターンで上記のswitch-case文を書き換えるには、次のようにします。まず、以下のような基本となるクラス(またはインタフェース)CGameStateを用意します。

```
class CGameState {
public:
    virtual void ActiveProc() = 0;      // アクティブ状態時の処理をする関数
    virtual void NonActiveProc() = 0;   // 非アクティブ状態時の処理をする関数
};
```

そして、それぞれの状態に対応したクラスCTitle, CStage1, CStage2, CEndingをCGameStateから派生させ、仮想関数ActiveProc, NonActiveProcをオーバーライドします。

```
class CTitle : public CGameState {
public:
    virtual void ActiveProc();
    virtual void NonActiveProc();
};

class CStage1 : public CGameState {
public:
    virtual void ActiveProc();
    virtual void NonActiveProc();
};

class CStage2 : public CGameState {
public:
    virtual void ActiveProc();
    virtual void NonActiveProc();
};

class CEnding : public CGameState {
public:
    virtual void ActiveProc();
    virtual void NonActiveProc();
};
```

これで、それぞれの状態(シーン)をクラスとしてまとめることができます。ゲーム処理クラス本体(仮にCGameProcとします)には、CGameStateクラスへのポインタm_pGameStateを持たせ、ゲーム処理が必要になるたびにm_pGameStateのメンバ関数を呼び出します。

```
class CGameProc {
public:
    void Run(bool inActive)
    {
        if(inActive)
            m_pGameState->ActiveProc();
        else
            m_pGameState->NonActiveProc();
    }

private:
    CGameState*   m_pGameState;
};
```

m_pGameStateには、ゲームの状態に応じてCTitle, CStage1, CStage2, CEndingのいずれかのインスタンスをセットします。こうすることで、延々と続くswitch-case文から解放されます。

```
// 例：タイトルを設定
delete m_pGameState;
m_pGameState = new CTitle;

// ステージ１へ遷移
delete m_pGameState;
m_pGameState = new CStage1;
```

課　　題

ゲームの状態を表現したクラスを作成し、状態にあった処理を効率的に行うことができるようにしましょう。

(１)ゲームの状態(シーン)を表現するクラスの基本クラスCGameSceneを作成します。CGameSceneクラスのヘッダファイル(GameScene.hpp)を以下のように作成しましょう。

－GameScene.hpp－

```
/*
================================================================================================
                                オブジェクト指向ゲームプログラミング
     Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
================================================================================================

  【対象ＯＳ】
     Microsoft Windows2000/XP

  【コンパイラ】
     Microsoft Visual C++ 2005

  【プログラム】
     GameScene.hpp
                ゲームシーン基本クラスヘッダ

  【履歴】
     * Version    1.00       2005/03/dd  hh:mm:ss

================================================================================================
*/

#pragma ????

/************************************************************************************************/
/*                                        インクルードファイル                                  */
/************************************************************************************************/
#include <※ここは各自考えましょう>

/************************************************************************************************/
/*                                      ゲームシーン基本クラス定義                              */
/************************************************************************************************/
class CGameScene {
public:
    CGameScene() {}
    ??????? ~CGameScene() {}

    virtual int ActiveProc()    { return 0; }
    virtual int NonActiveProc() { ::WaitMessage(); return 0; }

private:
    CGameScene(const CGameScene&);
    CGameScene& operator=(const CGameScene&);
};
```

仮想関数ActiveProc, NonActiveProcは、純粋仮想関数ではなく、基本的な処理が記述されています。このように、基底クラスで派生クラス共通の処理または処理の枠組みを定義し、派生クラスでその具体的な内容を定義するようなデザインパターンをTemplate Methodパターンと呼びます。

新たなシーンを作成する場合は、CGameSceneを継承したクラスを定義します。仮想関数ActiveProc, NonActiveProcは、動作を変更したいときだけオーバーライドします。(※NonActiveProc関数で使われているWaitMessage関数は、メッセージが発生するまで待機状態に入る関数です)

(２)CGameSceneクラスのメンバは、以下のとおりです。

CGameScene　コンストラクタ

CGameSceneオブジェクトを構築します。

~CGameScene　デストラクタ

CGameSceneオブジェクトを解放します。

ActiveProc　オーバーライド可能な関数：処理

アプリケーションがアクティブ状態時のシーン処理を行います。

試virtual int ActiveProc();

Return　シーン続行：０　シーン変更：０以外の正数　アプリケーション終了：負

NonActiveProc　オーバーライド可能な関数：処理

アプリケーションが非アクティブ状態時のシーン処理を行います。

試virtual int ActiveProc();

Return　シーン続行：０　シーン変更：０以外の正数　アプリケーション終了：負

(３)機能テストのためのシーンを表現するCTestSceneクラスを作成します。このクラスでは、アクティブ状態時の処理だけ変更したいので、ActiveProc関数のみオーバーライドします。
以下のプログラムをGameScene.hppに追加しましょう。

```
/*****************************************************************************************************/
/*                                        テストシーンクラス定義                                       */
/*****************************************************************************************************/
class CTestScene : ?????? ?????????? {
public:
    CTestScene();
    ??????? ~CTestScene();

    virtual int ActiveProc();

private:
    // ここに、機能テストで必要となる変数や関数を定義します
};
```

新たなシーンを作成する場合、専用のヘッダファイルを作成し、そこで定義しても構いませんが、すべてのシーンをGameScene.hppに定義しておけば、このヘッダをインクルードするだけですべてのシーンを使用できるという利点があります。

(４)CTestSceneクラスのソースファイル(TestScene.cpp)を以下のように作成しましょう。

－TestScene.cpp－

```
/*
=====================================================================================================
                                オブジェクト指向ゲームプログラミング
     Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
=====================================================================================================

  【対象ＯＳ】
     Microsoft Windows2000/XP

  【コンパイラ】
     Microsoft Visual C++ 2005

  【プログラム】
     TestScene.cpp
                テストシーンクラス

  【履歴】
     * Version    1.00       2005/03/dd  hh:mm:ss

=====================================================================================================
*/

/*****************************************************************************************************/
/*                                        インクルードファイル                                         */
```

```
/*****************************************************************************************************/
※ここは各自考えましょう

/*****************************************************************************************************/
/*                                        デフォルトコンストラクタ                                     */
/*****************************************************************************************************/
CTestScene::CTestScene()
{
    // ここに、機能テストの初期化処理を記述します
}

/*****************************************************************************************************/
/*                                          デストラクタ                                               */
/*****************************************************************************************************/
CTestScene::~CTestScene()
{
    // ここに、機能テストの解放処理を記述します
}

/*****************************************************************************************************/
/*                                        アクティブ処理                                               */
/*****************************************************************************************************/
int CTestScene::ActiveProc()
{
    // ここに、機能テストのメイン処理を記述します

    return 0;   // シーン続行
}
```

(５)ゲームの状態(シーン)は、ゲーム処理の一部であると考えられます。以下のプログラムを適切な場所に追加し、CGameSceneクラスを適切なクラスに集約させましょう。

```
CGameScene*   m_pScene;
```

(６)(５)を追加したクラスのコンストラクタ初期化子に、以下のプログラムを追加し、メンバ変数m_pSceneが適切に初期化されるようにしましょう。

```
m_pScene(NULL)     または    m_pScene(new CGameScene)
```

(７)シーンを生成する処理を追加します。

シーンの生成は、以下のように、古いシーンのインスタンスをdeleteで解放したあと、新しいシーンのインスタンスをnewで生成し、m_pScene変数に代入することで行います。

```
delete m_pScene;              // 古いシーンの解放
m_pScene = new CTestScene;    // 新しいシーンの生成
```

この場合、生成するインスタンスのクラス名をあらかじめ知っておく必要があります。また、改良などによりクラス名が変更になったり、別のクラスを生成したくなった場合、該当箇所を探し出して修正しなければなりません。
CGameSceneクラスを継承したクラスのインスタンスを生成しつつ、実際に生成されるインスタンスのクラスを隠蔽したり、固定したくない場合があります。このような場合、Factory Methodパターンと呼ばれる「インスタンスを生成する工場」というクラスまたは関数を導入します。

Factory Methodパターンを導入するため、各シーンに固有のIDを割り当てます。以下のプログラムをGameScene.hppに追加しましょう。

```
/*****************************************************************************************************/
/*                                        ゲームシーンID                                               */
/*****************************************************************************************************/
enum SCENE {
    SCN_NONE,   // シーンなし
    SCN_TEST    // テストシーン
};
```

(８)シーンを解放するReleaseScene関数を適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          シーン解放                                               */
/*****************************************************************************************************/
void CGameProc::ReleaseScene()
{
    delete m_pScene;
    m_pScene = NULL;
}
```

ヒント：シーンの作成や解放は、ゲーム処理の機能と考えられます。

(９)Factory Methodパターンを用いてシーンを生成するCreateScene関数の足りない部分を補い、適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          シーン生成                                               */
/*****************************************************************************************************/
bool CGameProc::CreateScene(const int inSceneID)
{
    ReleaseScene();     // 古いシーンの解放

    switch(inSceneID) {
      case SCN_NONE:    m_pScene = new CGameScene();    break;
      case SCN_TEST:    m_pScene = new ??????????();    break;
      default:          return false;
    }

    return true;
}
```

CreateScene関数により、シーンの生成は、具体的なクラス名ではなく、シーンのIDを指定することで行えます。存在しないIDが渡されたときは、プログラムを終了する、例外を発生させる、「何もしない」「基本的な処理のみ行う」といったクラスのインスタンスを生成する、などの処理を行います。
また、この関数では、メンバ変数m_pSceneに生成したインスタンスを代入していますが、ほとんどの場合、生成したインスタンスのポインタを返します。

Factory Methodパターンを導入する利点は、具体的なクラス名を知らなくてもインスタンスの生成が行えるという点と、インスタンスを生成する処理が一カ所にまとめられるという点が挙げられます。クラスの変更や追加が起こっても、関数内の修正だけで済みます。
欠点は、switch-case文による多分岐をなくそうしているオブジェクト指向の概念に違反する点と、派生クラスの定義が、ひとつのソースファイルに集積されるという点です。CGameProc::CreateScene関数を実装するファイルでは、CGameSceneクラスから派生したすべてのクラス(caseで分岐するすべてのクラス)をインクルードしなければならないため、コンパイル時や保守時の依存関係を複雑化することにつながります(これらの欠点を改善する策ももちろんありますが、今回は取り上げません)。

(10)CGameProc::Run関数をシーン処理関数ActiveProcおよびNonActiveProcの戻り値を考慮したものに変更しましょう。

```
bool CGameProc::Run(const bool inActive)
{
    int   ret;
    if(inActive)
        ret = m_pScene->ActiveProc();
    else
        ret = m_pScene->NonActiveProc();

    if(ret == 0)
        return true;            // シーン継続
    if(ret < 0)
        return false;           // ゲーム終了

    return ??????????(ret);     // 新しいシーンの生成
}
```

(２)の仕様より、ActiveProc関数およびNonActiveProc関数の戻り値が０のときは同じシーンの継続を表しているのでtrueを返します。負の場合は、ゲーム終了を示しているのでfalseを返します。これ

ら以外の場合(０より大きい数)は、シーン変更を示しており、戻り値をそのままシーンIDとしてシーン生成関数に渡し、新しいシーンの生成を行います。

(11)CGameProcクラスのデストラクタをに以下のプログラムを追加し、シーンのインスタンスが確実に解放されるようにしましょう。

```
ReleaseScene();
```

(12)アプリケーションの初期化処理に以下のプログラムを追加し、最初のシーンを設定しましょう。

```
// 初期シーン生成
m_GameProc.CreateScene(SCN_TEST);
```

(13)アプリケーションの解放処理に以下のプログラムを追加し、シーンが正しいタイミングで解放されるようにしましょう。

```
m_GameProc.ReleaseScene();      // シーン解放
```