－ －

2014年1月20日 初版

板垣 衛

改訂履歴

|  |  |  |  |
| --- | --- | --- | --- |
| 版 | リリース | 担当 | 改訂内容 |
| 初版 | 2014年1月20日 | 板垣 衛 | （初版） |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目次

[ 概略 1](#_Toc378256309)

[ 目的 1](#_Toc378256310)

[ 「デザインパターン」とは？ 1](#_Toc378256311)

[▼ GoFの23種類のデザインパターン 1](#_Toc378256312)

[ 生成に関するパターン 2](#_Toc378256313)

[ 構造に関するパターン 2](#_Toc378256314)

[ 振る舞いに関するパターン 3](#_Toc378256315)

[ ゲームプログラミングにおけるデザインパターンの活用 4](#_Toc378256316)

[▼ Observerパターン（Listner） 4](#_Toc378256317)

[▼ Adapterパターン 8](#_Toc378256318)

[▼ Proxyパターン（Wrapper） 14](#_Toc378256319)

[▼ Decoratorパターン（Filter） 24](#_Toc378256320)

[▼ Strategyパターン（Policy） 24](#_Toc378256321)

[▼ Template Methodパターン 25](#_Toc378256322)

# 概略

ゲームプログラミングにも効果的に適用できるデザインパターンについて解説。

# 目的

本書は、ゲームプログラミングに有効活用できるデザインパターンを示すことを目的とする。

それ以上に、「デザインパターン」という「共通認識」をプログラマーに提供することを目的とする。

「デザインパターン」や「UML」といった共通の知識は、プログラマー間の意思疎通をより円滑なものにする。

# 「デザインパターン」とは？

生産性を向上させるためのプログラミングのパターンのこと。

GoF（Gang of Four = 四人組）と呼ばれる4人の著者がまとめた23種類のパターンが特に有名。いずれもオブジェクト指向に基づくプログラミングパターンである。

狭義には「デザインパターン」と言えばこの23パターンのことを指すが、他のパターンも存在する。

社内でゲームに特化したパターンを独自に蓄積していくといった取り組みも良いかもしれない。

## GoFの23種類のデザインパターン

本書は、この「23のパターン」の中から、ゲームプログラミングに役立つパターンを幾つかピックアップして解説する。

まずは23のパターンを全て列挙する（Wikipediaの記述をそのまま引用し、一部捕捉を加える）。

なお、本書で解説するパターンについては赤い太字で表記する。解説はしないが、ゲームでも有用なパターンを緑字で表記。

### 生成に関するパターン

**Abstract Factory** **【アブストラクトファクトリー】（抽象的な工場）**

関連する一連のインスタンスを状況に応じて適切に生成する方法提供する。

**Builder** **【ビルダー】（構築者）**

複合化されたインスタンスの生成過程を隠蔽する。

**Factory Method** **【ファクトリーメソッド】（工場メソッド）**

実際に生成されるインスタンスに依存しない、インスタンスの生成方法を提供する。

**Prototype** **【プロトタイプ】（原型）**

同様のインスタンスを生成するために、原型のインスタンスを複製する。

**Singleton** **【シングルトン】（単体）**

あるクラスについて、インスタンスが単一であることを保証する。

### 構造に関するパターン

**Adapter** **【アダプター】（接続）**

元々関連性のない２つのクラスを接続するクラスを作る。

**Bridge** **【ブリッジ】（橋渡し）**

クラスなどの実装と、呼び出し側の間の橋渡しをするクラスを用意し、実装を隠蔽する。

**Composite** **【コンポジット】（合成）※階層**

再帰的な構造を表現する。

**Decorator** **【デコレーター】（装飾者）**

あるインスタンスに対して、動的に付加機能を追加する。Filterとも呼ばれる。

**Facade** **【ファサード】（外見）**

複数のサブシステムの窓口となる共通のインターフェースを提供する。

**Flyweight** **【フライウェイト】（軽量級）**

多数のインスタンスを共有し、インスタンスの構築のための負荷を減らす。

**Proxy** **【プロキシー】（代理人）**

共通のインターフェースをもつインスタンスを内包し、利用者からのアクセスを代理する。Wrapperとも呼ばれる。

### 振る舞いに関するパターン

**Chain of Responsibility** **【チェインオブレスポシビリティ】（責任の連鎖）**

イベントの送受信を行う複数のオブジェクトを鎖状につなぎ、それらの間をイベントが渡されてゆくようにする。

**Command** **【コマンド】（命令）**

複数の異なる操作について、それぞれに対応するオブジェクトを用意し、オブジェクトを切り替えることで操作の切り替えを実現する。

**Interpreter** **【インタープリタ】（通訳）**

構文解析のために、文法規則を反映するクラス構造を作る。

**Iterator** **【イテレータ】（繰り返し）**

複数の要素を内包するオブジェクトのすべての要素に順にアクセスする方法を提供する。反復子。

**Mediator** **【メディエイター】（調停者）**

オブジェクト間の相互作用を仲介するオブジェクトを定義し、オブジェクト間の結合度を低くする。

**Memento** **【メメント】（形見）**

データ構造に対する一連の操作のそれぞれを記録しておき、以前の状態の復帰または操作の再現が行えるようにする。

**Observer**  **【オブザーバー】（観察者）**

インスタンスの変化を他のインスタンスから監視できるようにする。Listnerとも呼ばれる。

**State** **【ステート】（状態）**

オブジェクトの状態を変化させることで、処理内容を変えられるようにする。

※「ステートマシン」として有名。有限オートマトンの実装に使われる。

**Strategy** **【ストラテジー】（戦略）**

データ構造に対して適用する一種のアルゴリズムをカプセル化し、アルゴリズムの切り替えを容易にする。

**Template Method** **【テンプレートメソッド】（ひな形メソッド）**

あるアルゴリズムの途中経過で必要な処理を抽象メソッドに委ね、その実装を変えることで処理が変えられるようにする。

**Visitor**  **【ビジター】（訪問者）**

データ構造を保持するクラスと、それに対して処理を行うクラスを分離する。

# ゲームプログラミングにおけるデザインパターンの活用

幾つかのデザインパターンを、実際にゲームプログラミングに適用することを想定したサンプルを示す。

## Observerパターン（Listner）

* 用途①： 何らかの状態を**観察**して、変化があったときに、関係するオブジェクトに通知したい場合。いわゆるイベントコールバック。もしくは、Delegate（デリゲート＝委譲）。
* 用途②： 共通処理に対して、タイトル固有の挙動を与えたい場合。



実装例：

|  |
| --- |
| //================================================================================  //Observerパターン  //----------------------------------------  //【準備】C++11非対応のコンパイラなら、override, final キーワードを何もしないマクロで代用  //#define override  //#define final  //----------------------------------------  //--------------------  //【ライブラリ側】委譲インターフェース  template<class T>  class IDelegate  {  public:  //更新処理  virtual void update(T& target) = 0;  };  //--------------------  //【ライブラリ側】委譲オブジェクト登録クラス  template<class T>  class CDelegateRegister  {  public:  //委譲オブジェクト登録  void registDelegate(IDelegate<T>\* delegate){ m\_delegate = delegate; }  //更新処理  void update()  {  printf("CDelegateRegister<T>::update() : m\_delete->update()\n");  if (m\_delegate)  {  m\_delegate->update(m\_target);  }  }  public:  //コンストラクタ  CDelegateRegister(T& target) : m\_target(target), m\_delegate(nullptr) {}  private:  //フィールド  T& m\_target;//対象オブジェクト  IDelegate<T>\* m\_delegate;//委譲オブジェクト  };  //--------------------  //【ライブラリ側】共通システム１クラス  class CCommonSystem1  {  public:  //委譲オブジェクト登録  void registDelegate(IDelegate<CCommonSystem1>\* delegate){ m\_delegateRegister.registDelegate(delegate); }  //更新処理  void update()  {  printf("CCommonSystem1::update()\n");  //更新処理の委譲  m\_delegateRegister.update();  }  //メッセージ取得  const char\* getMessage() const{ return "CCommonSystem1's message"; }  public:  //コンストラクタ  CCommonSystem1() : m\_delegateRegister(\*this) {}  private:  //フィールド  CDelegateRegister<CCommonSystem1> m\_delegateRegister;//委譲オブジェクト  };  //--------------------  //【ライブラリ側】共通システム２クラス  class CCommonSystem2  {  public:  //委譲オブジェクト登録  void registDelegate(IDelegate<CCommonSystem2>\* delegate){ m\_delegateRegister.registDelegate(delegate); }  //更新処理  void update()  {  printf("CCommonSystem2::update()\n");  //更新処理の委譲  m\_delegateRegister.update();  }  //システム名取得  const char\* getSystemName() const{ return "Common System 2"; }  public:  //コンストラクタ  CCommonSystem2() : m\_delegateRegister(\*this) {}  private:  //フィールド  CDelegateRegister<CCommonSystem2> m\_delegateRegister;//委譲オブジェクト  };  //----------------------------------------  //--------------------  //【タイトル側】独自システム  class CMySystem  {  public:  //共通処理１向けの拡張更新処理  void updateSystem1(CCommonSystem1& target)  {  printf("CMySystem::updateSystem1() : target.getMessage()=\"%s\"\n", target.getMessage());  }  //共通処理２向けの拡張更新処理  void updateSystem2(CCommonSystem2& target)  {  printf("CMySystem::updateSystem2() : target.getSystemName()=\"%s\"\n", target.getSystemName());  }  private:  //共通処理システム１向けのオブザーバー（委譲オブジェクト）  class CObserver1 : public IDelegate<CCommonSystem1>  {  public:  //更新処理  void update(CCommonSystem1& target) override  {  m\_this.updateSystem1(target);  }  public:  CObserver1(CMySystem& me) : m\_this(me) {}  private:  CMySystem& m\_this;  };  //共通処理システム２向けのオブザーバー（委譲オブジェクト）  class CObserver2 : public IDelegate<CCommonSystem2>  {  public:  //更新処理  void update(CCommonSystem2& target) override  {  m\_this.updateSystem2(target);  }  public:  CObserver2(CMySystem& me) : m\_this(me) {}  private:  CMySystem& m\_this;  };  public:  //コンストラクタ  CMySystem(CCommonSystem1& sys1, CCommonSystem2& sys2) :  m\_observer1(\*this),  m\_observer2(\*this)  {  //オブザーバー登録  sys1.registDelegate(&m\_observer1);  sys2.registDelegate(&m\_observer2);  }  private:  //フィールド  CObserver1 m\_observer1;//共通処理システム１向けのオブザーバー  CObserver2 m\_observer2;//共通処理システム２向けのオブザーバー  };  **//----------------------------------------**  **//Observerパターンテストメイン関数**  **void testObserver()**  {  printf("\n- testObserver() -\n\n");  CCommonSystem1 sys1; //共通システム１  CCommonSystem2 sys2; //共通システム２  CMySystem my\_sys(sys1, sys2); //独自システム  //共通システムの更新処理  sys1.update();  sys2.update();  //※共通システム側の処理実行時に、オブザーバーによって、独自システム側の処理が委譲される  } |

↓（処理結果）

|  |
| --- |
| - testObserver() -  CCommonSystem1::update() ←共通処理１  CDelegateRegister<T>::update() : m\_delete->update() ←オブザーバーに委譲  CMySystem::updateSystem1() : target.getMessage()="CCommonSystem1's message" ←独自処理  CCommonSystem2::update() ←共通処理２  CDelegateRegister<T>::update() : m\_delete->update() ←オブザーバーに委譲  CMySystem::updateSystem2() : target.getSystemName()="Common System 2" ←独自処理 |

## Adapterパターン

* 用途： 直接関連性のないオブジェクトを連携（**接続**）させたい場合。

例えば、画面上の敵、NPC、宝箱、スイッチなどといった様々なオブジェクトに対して、共通のターゲッティング処理を行いたい場合。この時、個々のオブジェクトに直接ターゲッティングに関する処理は実装したくない。





実装例：（あらかじめ用意されているバッファにクラスのインスタンスを作ることでメモリを効率的に利用する、配置newの活用テクニックも示す）

|  |
| --- |
| //================================================================================  //Adapterパターン  //----------------------------------------  //【準備】C++11非対応のコンパイラなら、override, final キーワードを何もしないマクロで代用  //#define override  //#define final  //----------------------------------------  //【準備】ベクトル演算型定義  typedef glm::vec3 vec3;//GLM利用  //----------------------------------------  //【準備】共通関数  //配列の要素数を返す（型安全版）  template<typename T, std::size\_t N1> inline std::size\_t lengthOfArray(const T(&var)[N1]){ return N1; }  //大きい方の値を返す（型安全版）  //template<typename T> inline T max(T var1, Tvar2){ return var1 > var2 ? var1 : var2; }  //配列の要素数を返す  //#define lengthOfArray(var) (sizeof(var1) / sizeof(var1[0])  //大きい方の値を返す  #define max(var1, var2) (var1 > var2 ? var1 : var2)  //----------------------------------------  //--------------------  //NPC  class CNPC //※何の共通インターフェースも実装していない  {  public:  //アクセッサ  const char\* getName() const { return m\_name; } //名前を取得  const vec3& getPos() const { return m\_pos; } //位置を取得  void setPos(const vec3 pos) { m\_pos = pos; } //位置を更新  float getRotY() const { return m\_rot\_y; } //Y軸の向きを取得  void setRotY(const float rot\_y) { m\_rot\_y = rot\_y; } //Y軸の向きを更新  public:  //コンストラクタ  CNPC(const char\* name, const vec3 pos, const float rot\_y) :  m\_name(name),  m\_pos(pos),  m\_rot\_y(rot\_y)  {}  private:  //フィールド  const char\* m\_name;//名前  vec3 m\_pos; //位置  float m\_rot\_y; //Y軸の向き  };  //--------------------  //宝箱  class CTBox //※何の共通インターフェースも実装していない  {  public:  //アクセッサ  int getID() const { return m\_id; } //IDを取得  const vec3& getPos() const { return m\_pos; } //位置を取得  float getRotY() const { return m\_rot\_y; } //Y軸の向きを取得  public:  //コンストラクタ  CTBox(const int id, const vec3 pos, const float rot\_y) :  m\_id(id),  m\_pos(pos),  m\_rot\_y(rot\_y)  {}  private:  //フィールド  const int m\_id; //ID  const vec3 m\_pos; //位置  const float m\_rot\_y;//Y軸の向き  };  //----------------------------------------  //--------------------  //外部システム１  namespace other\_system1  {  //変数  static int s\_npc\_num = 0;  static CNPC\* s\_npc[10];    //初期化  void init()  {  s\_npc[s\_npc\_num++] = new CNPC("太郎", vec3(10.f, 11.f, 12.f), 0.1f);  s\_npc[s\_npc\_num++] = new CNPC("次郎", vec3(20.f, 21.f, 22.f), 0.2f);  s\_npc[s\_npc\_num++] = new CNPC("三郎", vec3(30.f, 31.f, 32.f), 0.3f);  s\_npc[s\_npc\_num++] = new CNPC("四朗", vec3(40.f, 41.f, 42.f), 0.4f);  s\_npc[s\_npc\_num++] = new CNPC("五朗", vec3(50.f, 51.f, 52.f), 0.5f);  }    //周辺のNPC情報を収集  int findAroundNPC(CNPC\* ref\_npc[], const int ref\_max)  {  int ref\_num = 0;  for (int index = 0; index < s\_npc\_num && index < ref\_max; ++index)  {  ref\_npc[ref\_num++] = s\_npc[index];  }  return ref\_num;  }  }  //--------------------  //外部システム２  namespace other\_system2  {  //変数  static int s\_tbox\_num = 0;  static CTBox\* s\_tbox[10];  //初期化  void init()  {  s\_tbox[s\_tbox\_num++] = new CTBox(101, vec3(15.f, 15.f, 15.f), 1.1f);  s\_tbox[s\_tbox\_num++] = new CTBox(102, vec3(25.f, 25.f, 25.f), 1.2f);  s\_tbox[s\_tbox\_num++] = new CTBox(103, vec3(35.f, 35.f, 35.f), 1.3f);  s\_tbox[s\_tbox\_num++] = new CTBox(104, vec3(45.f, 45.f, 45.f), 1.4f);  s\_tbox[s\_tbox\_num++] = new CTBox(105, vec3(55.f, 55.f, 55.f), 1.5f);  }  //周辺の宝箱情報を収集  int findAroundTBox(CTBox\* ref\_tbox[], const int ref\_max)  {  int ref\_num = 0;  for (int index = 0; index < s\_tbox\_num && index < ref\_max; ++index)  {  ref\_tbox[ref\_num++] = s\_tbox[index];  }  return ref\_num;  }  }  //----------------------------------------  //--------------------  //ターゲット用アダプターインターフェース  class ITargetAdapter  {  public:  //アクセッサ  virtual void getIdentifier(char\* buff, const size\_t buff\_size) const = 0;//識別情報を取得  virtual const vec3& getPos() const = 0; //位置を取得  virtual float getRotY() const = 0; //Y軸の向きを取得  };  //--------------------  //NPC向けターゲット用アダプター  class CTargetAdapterNPC : public ITargetAdapter  {  public:  //識別情報を取得  void getIdentifier(char\* buff, const size\_t buff\_size) const override  {  sprintf\_s(buff, buff\_size, "NPC\"%s\"", m\_npc.getName());  }  //位置を取得  const vec3& getPos() const override  {  return m\_npc.getPos();  }  //Y軸の向きを取得  float getRotY() const override  {  return m\_npc.getRotY();  }  public:  //コンストラクタ  CTargetAdapterNPC(CNPC& npc) :  m\_npc(npc)  {}  private:  //フィールド  CNPC& m\_npc;//NPC  };  //--------------------  //宝箱向けターゲット用アダプター  class CTargetAdapterTBox : public ITargetAdapter  {  public:  //識別情報を取得  void getIdentifier(char\* buff, const size\_t buff\_size) const override  {  sprintf\_s(buff, buff\_size, "宝箱(%d)", m\_tbox.getID());  }  //位置を取得  const vec3& getPos() const override  {  return m\_tbox.getPos();  }  //Y軸の向きを取得  float getRotY() const override  {  return m\_tbox.getRotY();  }  public:  //コンストラクタ  CTargetAdapterTBox(CTBox& tbox) :  m\_tbox(tbox)  {}  private:  //フィールド  CTBox& m\_tbox;//宝箱  };  //----------------------------------------  //--------------------  //ターゲット処理用配置new  class CTargetCollector;  //コンストラクタ実行用に与えられたポインターをそのまま返すだけのnew  void\* operator new(std::size\_t size, const CTargetCollector\*, void\* buff){ return buff; }  //なにしないdelete（配置newとdeleteは一対で登録しないとダメ）  void operator delete(void\* mem, const CTargetCollector\*, void\* buff){}  //--------------------  //ターゲット収集処理クラス  class CTargetCollector  {  public:  //周辺のターゲット情報を収集  void findAroundTarget()  {  //周辺のNPC情報を収集  CNPC\* ref\_npc[4];  const int ref\_npc\_num = other\_system1::findAroundNPC(ref\_npc, lengthOfArray(ref\_npc));  //周辺の宝箱情報を収集  CTBox\* ref\_tbox[3];  const int ref\_tbox\_num = other\_system2::findAroundTBox(ref\_tbox, lengthOfArray(ref\_tbox));  //ターゲット情報に置き換え  m\_numTargets = 0;  {  //NPC情報をターゲット情報に置き換え  for (int index = 0; index < ref\_npc\_num; ++index)  {  if (m\_numTargets < MAX\_TARGETS)  {  try  {  m\_target[m\_numTargets]  = new(this, m\_targetBuff[m\_numTargets]) CTargetAdapterNPC(\*ref\_npc[index]);  ++m\_numTargets;  }  catch (...)  {  fprintf(stderr, "CTargetAdapterNPC allocation failure.\n");  }  }  }  //宝箱情報をターゲット情報に置き換え  for (int index = 0; index < ref\_tbox\_num; ++index)  {  if (m\_numTargets < MAX\_TARGETS)  {  try  {  m\_target[m\_numTargets]  = new(this, m\_targetBuff[m\_numTargets]) CTargetAdapterTBox(\*ref\_tbox[index]);  ++m\_numTargets;  }  catch (...)  {  fprintf(stderr, "CTargetAdapterTBox allocation failure.\n");  }  }  }  //ターゲット情報のソート（スクリーン座標で横並びに）  //...の代わりにシャッフル  time\_t timer;  time(&timer);  srand(static\_cast<unsigned int>(timer));  const int shuffle\_num = rand() % 20;  for (int num = 0; num < shuffle\_num; ++num)  {  const int index1 = rand() % m\_numTargets;  const int index2 = rand() % m\_numTargets;  ITargetAdapter\* target\_temp = m\_target[index1];  m\_target[index1] = m\_target[index2];  m\_target[index2] = target\_temp;  }  }  }  //ターゲット情報を取得  ITargetAdapter\* getTarget(const int index)  {  if (index < 0 || index >= m\_numTargets)  return nullptr;  return m\_target[index];  }  //ターゲット情報数を取得  int getTargetNum() const{ return m\_numTargets; }  public:  //コンストラクタ  CTargetCollector() :  m\_numTargets(0)  {}  private:  //フィールド  static const std::size\_t MAX\_TARGETS = 10; //ターゲット情報の最大数  static const std::size\_t MAX\_SIZE\_OF\_TARGET\_ADAPTER  = max(sizeof(CTargetAdapterNPC), sizeof(CTargetAdapterTBox));//ターゲット情報用バッファのサイズ  char m\_targetBuff[MAX\_TARGETS][MAX\_SIZE\_OF\_TARGET\_ADAPTER]; //ターゲット情報用のバッファ  ITargetAdapter\* m\_target[MAX\_TARGETS]; //ターゲット情報  int m\_numTargets; //ターゲット数  };  **//----------------------------------------**  **//Adapterパターンテストメイン関数**  **void testAdapter()**  {  printf("\n- testAdapter() -\n\n");    //外部システム１初期化  other\_system1::init();  //外部システム２初期化  other\_system2::init();  //周辺のターゲット情報を収集  CTargetCollector target\_coll;  target\_coll.findAroundTarget();  //ターゲット情報を表示  //※ターゲットがNPCか宝箱か気にせず一律の処理を行っている  for (int index = 0; index < target\_coll.getTargetNum(); ++index)  {  ITargetAdapter\* target = target\_coll.getTarget(index);  if (target)  {  //共通処理を呼んでいるが、アダプターの作用により、  //実際にはNPCと宝箱のそれぞれ固有の処理が実行されている  char identifier[128];  target->getIdentifier(identifier, sizeof(identifier));  const vec3& pos = target->getPos();  const float rot\_y = target->getRotY();  printf("[%d] \"%s\" pos=(%.1f, %.1f, %.1f) rot=(%.1f)\n",  　index, identifier, pos[0], pos[1], pos[2], rot\_y);  }  }  } |

↓（処理結果）

|  |
| --- |
| - testAdapter() -  [0] "NPC"太郎"" pos=(10.0, 11.0, 12.0) rot=(0.1) ←一回のループ処理で、2種類のオブジェクトの処理が  [1] "宝箱(103)" pos=(35.0, 35.0, 35.0) rot=(1.3) 入り乱れて呼びだれていることがわかる  [2] "宝箱(102)" pos=(25.0, 25.0, 25.0) rot=(1.2)  [3] "NPC"四朗"" pos=(40.0, 41.0, 42.0) rot=(0.4)  [4] "宝箱(101)" pos=(15.0, 15.0, 15.0) rot=(1.1)  [5] "NPC"三郎"" pos=(30.0, 31.0, 32.0) rot=(0.3)  [6] "NPC"次郎"" pos=(20.0, 21.0, 22.0) rot=(0.2) |

## Proxyパターン（Wrapper）

* 用途①： 一時的にあるオブジェクトのふるまいを偽装（**代理**）したい場合。

例えば、キャラクターを扱うオブジェクトがあるが、特定の場面に限りローディングに時間がかかるため、ローディングマークを表示しながらも、クリックしたりなどの本来の反応は受け付けたい。ローディングマークの処理をキャラクターの基本処理に組み込むようなことをせず、この場面だけの特殊な処理として扱いたい。

* 用途②： スマートポインター。

一見して普通のオブジェクト（クラス）のポインターに見せつつ、内部で参照カウンタをカウントして、参照がなくなったら自己消滅する。



実装例①：（イテレーターを使って処理の記述を簡潔にする方法も示す）

|  |
| --- |
| //================================================================================  //Proxyパターン(1)  //----------------------------------------  //【準備】C++11非対応のコンパイラなら、override, final キーワードを何もしないマクロで代用  //#define override  //#define final  //----------------------------------------  //--------------------  //キャラクターインターフェースクラス  //※このインターフェースを、本来のキャラクタークラスとプロキシークラスが共有することで、  //　プロキシーが代理として振る舞うことができるようにする  class CCharacter;  class ICharacter  {  public:  //処理更新  virtual void update() = 0;  //描画処理  virtual void draw() = 0;  //構築済みフラグ取得  virtual bool isBuilt() const = 0;  //本体取得  virtual const CCharacter\* getReal() const = 0;  };  //--------------------  //キャラクタークラス  class CCharacter : public ICharacter  {  public:  //処理更新  void update() override  {  //処理経過  ++m\_step;    //2ステップ以上経過したら、1/4の確率で構築完了ということにする  if (m\_step > 2 && rand() % 4 == 0)  {  m\_isBuilt = true;  }  }  //描画処理  void draw() override  {  printf("This is real \"%s\"!\n", m\_name);  }  //構築済みフラグ取得  bool isBuilt() const override  {  return m\_isBuilt;  }  //本体取得  const CCharacter\* getReal() const override  {  return this;  }  public:  //アクセッサ  const char\* getName() const { return m\_name; }//名前取得  int getStep() const { return m\_step; } //処理ステップ取得  public:  //コンストラクタ  CCharacter(const char\* name) :  m\_name(name),  m\_step(0),  m\_isBuilt(false)  {}  private:  //フィールド  const char\* m\_name;//名前  int m\_step; //処理ステップ  bool m\_isBuilt; //構築済みフラグ  };  //----------------------------------------  //--------------------  //ローディング中のキャラクター代理クラス  class CCharacterProxy : public ICharacter  {  public:  //処理更新  virtual void update() override  {  m\_realObj->update();//委譲  }  //描画処理  virtual void draw()  {  printf("Now is loading \"%s\"... (step=%d)\n", m\_realObj->getName(), m\_realObj->getStep());  }  //構築済みフラグ取得  bool isBuilt() const override  {  return false;  }  //本体取得  const CCharacter\* getReal() const override  {  return m\_realObj;  }  public:  //本来の実体をセット  void setRealObj(CCharacter\* real\_obj)  {  m\_realObj = real\_obj;  }  public:  //コンストラクタ  CCharacterProxy() :  m\_realObj(nullptr)  {}  private:  //フィールド  CCharacter\* m\_realObj;//本来の実体  };  //--------------------  //キャラクターコレクション  class CCharacterCollection  {  public:  //--------------------  //イテレータークラス  class CIterator  {  public:  //イテレーター用型宣言  typedef CIterator type;  typedef ICharacter\* value\_type;  public:  //イテレーター用オペレーター  operator value\_type() { return m\_chara; } //キャスト  operator const value\_type() const { return m\_chara; }//constキャスト  value\_type operator->() { return m\_chara; } //参照  const value\_type operator->() const { return m\_chara; } //const参照  ICharacter& operator\*() { return \*m\_chara; } //実体化  const ICharacter& operator\*() const { return \*m\_chara; }//const実体化  type& operator++() { ++m\_index; m\_chara = m\_coll[m\_index]; return \*this; }//前置インクリメント  type operator++(int) { type prev = \*this; ++(\*this); return prev; } //後置インクリメント  type& operator--() { --m\_index; m\_chara = m\_coll[m\_index]; return \*this; }//前置デクリメント  type operator--(int) { type prev = \*this; --(\*this); return prev; } //後置デクリメント  bool operator==(const type& ite) const//比較  {  if (m\_chara == nullptr && ite.m\_chara == nullptr)  return true;  if (m\_chara == nullptr || ite.m\_chara == nullptr)  return false;  return m\_chara->getReal() == ite->getReal();  }  bool operator!=(const type& ite) const//比較  {  return !(\*this == ite);  }  public:  //コンストラクタ  CIterator(CCharacterCollection& coll, const int index) :  m\_coll(coll),  m\_index(index)  {  m\_chara = m\_coll[index];  }  private:  //フィールド  CCharacterCollection& m\_coll;//キャラクターコレクション  int m\_index; //コレクションのカレントインデックス  ICharacter\* m\_chara; //キャラクターオブジェクト  };  public:  //イテレータ―用メソッド  CIterator begin() { CIterator ite(\*this, 0); return ite; } //先頭イテレーターを返す  CIterator end() { CIterator ite(\*this, m\_charaNum); return ite; }//終端イテレーターを返す  bool empty() const { return m\_charaNum == 0; } //要素が空か？  int size() const { return m\_charaNum; } //要素数を返す  ICharacter\* operator[](const int index) //要素を返す　※ここでプロキシーに対応  {  if (index < 0 || index >= m\_charaNum)  return nullptr;  CCharacter\* realChara = m\_chara[index];  ICharacter\* chara = realChara;  if (!realChara->isBuilt())  {  //構築が済んでいない場合、プロキシ―を返す  static CCharacterProxy proxy;//プロキシーのインスタンス（再利用する：一種のFlyweightパターン）  proxy.setRealObj(realChara);  chara = &proxy;  }  return chara;  }  public:  //キャラ追加  void addChara(const char\* name)  {  if (m\_charaNum >= CHARA\_MAX)  return;  CCharacter\* chara = new CCharacter(name);  if (chara)  {  m\_chara[m\_charaNum++] = chara;  }  }  public:  //コンストラクタ  CCharacterCollection() :  m\_charaNum(0)  {}  //デストラクタ  ~CCharacterCollection()  {  //全て削除  for (int index = 0; index < m\_charaNum; ++index)  {  if (m\_chara[index])  {  delete m\_chara[index];  m\_chara[index] = nullptr;  }  }  }  private:  //フィールド  static const int CHARA\_MAX = 32;//キャラ情報の最大数  CCharacter\* m\_chara[CHARA\_MAX]; //キャラ情報  int m\_charaNum; //キャラ情報数  };  //自作for\_each  template<class T, class F>  inline void my\_for\_each(T ite, T end, F functor)  {  for (; ite != end; ++ite)  {  functor(ite);  }  }  //自作for\_each（関数オブジェクトがパラメータを受け取るバージョン）  template<class T, class F, class P1>  inline void my\_for\_each(T ite, T end, F functor, P1& param1)  {  for (; ite != end; ++ite)  {  functor(ite, param1);  }  }  //----------------------------------------  //--------------------  //共通処理システム  namespace common\_system  {  //キャラクター更新処理（関数オブジェクト）  struct Update  {  inline void operator()(ICharacter\* chara)  {  chara->update();  }  };    //キャラクター描画処理（関数オブジェクト）  struct Draw  {  inline void operator()(ICharacter\* chara)  {  chara->draw();  }  };  //キャラクター構築完了チェック  struct isBuiltAll  {  inline void operator()(ICharacter\* chara, bool& is\_built\_all)  {  if (!chara->isBuilt())  {  is\_built\_all = false;  }  }  };  };  **//----------------------------------------**  **//Proxyパターン(2)テストメイン関数**  **void testProxy1()**  {  printf("\n- testProxy1() -\n\n");  //キャラクター生成  CCharacterCollection chara\_coll;  chara\_coll.addChara("太郎");  chara\_coll.addChara("次郎");  chara\_coll.addChara("三郎");  chara\_coll.addChara("四郎");  chara\_coll.addChara("五郎");  //全キャラクターの構築が完了するまで処理  for (int step = 0; ; ++step)  {  printf("- step: %d -\n", step);  my\_for\_each(chara\_coll.begin(), chara\_coll.end(), common\_system::Update());//処理更新  my\_for\_each(chara\_coll.begin(), chara\_coll.end(), common\_system::Draw()); //描画  bool is\_built\_all = true;  my\_for\_each(chara\_coll.begin(), chara\_coll.end(), common\_system::isBuiltAll(), is\_built\_all);  if (is\_built\_all)  {  printf("- finish. -\n");  break;  }  }  } |

↓（処理結果）

|  |
| --- |
| - testProxy1() -  - step: 0 -  Now is loading "太郎"... (step=1)  Now is loading "次郎"... (step=1)  Now is loading "三郎"... (step=1)  Now is loading "四郎"... (step=1)  Now is loading "五郎"... (step=1)  - step: 1 -  Now is loading "太郎"... (step=2)  Now is loading "次郎"... (step=2)  Now is loading "三郎"... (step=2)  Now is loading "四郎"... (step=2)  Now is loading "五郎"... (step=2)  - step: 2 -  Now is loading "太郎"... (step=3)  Now is loading "次郎"... (step=3)  Now is loading "三郎"... (step=3)  Now is loading "四郎"... (step=3)  Now is loading "五郎"... (step=3)  - step: 3 -  This is real "太郎"! ←ローディングが済むまでは、  This is real "次郎"! プロキシー側の描画処理が実行されていることがわかる  Now is loading "三郎"... (step=4)  Now is loading "四郎"... (step=4)  This is real "五郎"!  - step: 4 -  This is real "太郎"!  This is real "次郎"!  This is real "三郎"!  Now is loading "四郎"... (step=5)  This is real "五郎"!  - step: 5 -  This is real "太郎"!  This is real "次郎"!  This is real "三郎"!  This is real "四郎"!  This is real "五郎"!  - finish. - |

実装②：スマートポインター

|  |
| --- |
| //================================================================================  //Proxyパターン(2)：簡易版スマートポインター  //----------------------------------------  //簡易版スマートポインターのテンプレートクラス  //Boost C++ の smart\_ptr のように、削除処理を指定するような機能はなし  template<class T>  class CSmartPtr  {  private:  //参照情報型  struct T\_REF\_INFO  {  //フィールド  T\* m\_realObj; //本当のオブジェクト  int m\_refCount;//参照カウンタ  //コンストラクタ  T\_REF\_INFO(T\* real\_obj, const int ref\_count) :  m\_realObj(real\_obj),  m\_refCount(ref\_count)  {}  T\_REF\_INFO(T\_REF\_INFO& info) :  m\_realObj(info.m\_realObj),  m\_refCount(info.m\_refCount)  {}  };  public:  //オペレータを実装して本来のオブジェクトを偽装（代理）  T\* operator->(){ return m\_refInfo ? m\_refInfo->m\_realObj : nullptr; }  const T\* operator->() const { return m\_refInfo ? m\_refInfo->m\_realObj : nullptr; }  T& operator\*(){ return \*m\_refInfo->m\_realObj; }  const T& operator\*() const { return \*m\_refInfo->m\_realObj; }  //代入演算子：スマートポインターを代入  CSmartPtr<T>& operator=(CSmartPtr<T>& obj)  {  //すでに参照しているならなにもしない  if (m\_refInfo == obj.m\_refInfo)  return \*this;    //参照カウンタをカウントダウン  release();    //参照情報をコピー  m\_refInfo = obj.m\_refInfo;    //参照カウンタをカウントアップ  addRef();    return \*this;  }  //代入演算子：ポインターを代入  CSmartPtr<T>& operator=(T\* real\_obj)  {  //すでに参照しているならなにもしない  if (m\_refInfo && m\_refInfo->m\_realObj == real\_obj)  return \*this;    //参照カウンタをカウントダウン  release();    if (real\_obj)//nullptr 代入時はなにもしない  {  //参照情報を生成  m\_refInfo = new T\_REF\_INFO(real\_obj, 1);  }  return \*this;  }  private:  //参照カウンタカウントアップ  void addRef()  {  if (!m\_refInfo)  return;  ++m\_refInfo->m\_refCount;  }  //参照カウンタをカウントダウン  void release()  {  if (!m\_refInfo)  return;  if (m\_refInfo->m\_refCount > 0)  {  --m\_refInfo->m\_refCount;  if (m\_refInfo->m\_refCount == 0)  {  //参照カウンタが0になったらオブジェクトを削除  if (m\_refInfo->m\_realObj)  {  delete m\_refInfo->m\_realObj;//本当のオブジェクトを削除  m\_refInfo->m\_realObj = nullptr;  }  delete m\_refInfo;//参照情報も削除  }  }  m\_refInfo = nullptr;  }  public:  //コンストラクタ  CSmartPtr(T\* real\_obj)  {  //参照情報を生成  m\_refInfo = new T\_REF\_INFO(real\_obj, 1);  }  //コピーコンストラクタ  CSmartPtr(CSmartPtr<T>& smart\_ptr)  {  //参照情報をコピー  m\_refInfo = smart\_ptr.m\_refInfo;    //参照カウンタをカウントアップ  addRef();  }  //デストラクタ  ~CSmartPtr()  {  //参照カウンタをカウントダウン  release();  }  private:  //フィールド  T\_REF\_INFO\* m\_refInfo;//参照情報  };  //----------------------------------------  //テスト用クラス  //※これ自体は何のインターフェースも持たない普通のクラス  //　このクラスをスマートポインターで扱う  class CTest  {  public:  //アクセッサ  const char\* getName() const { return m\_name; }  public:  //コンストラクタ  CTest(const char\* name)  {  printf("CTest::Ctest(\"%s\")\n", name);  m\_name = name;  }  //デストラクタ  ~CTest()  {  printf("CTest::~Ctest() : m\_name=\"%s\"\n", m\_name);  }  private:  //フィールド  const char\* m\_name;//名前  };  **//----------------------------------------**  **//Proxyパターン(2)テストメイン関数**  **void testProxy2()**  {  printf("\n- testProxy2() -\n\n");  //スマートポインターで CTest のインスタンスを生成  //自由にコピーしても、適切な開放を行ってくれる  //【注意】  //関数にスマートポインターを渡す時は、値渡しにしないと  //関数内でdeleteが発生する事に注意。  printf("(begin function)\n");  CSmartPtr<CTest> obj1 = new CTest("太郎");  printf("[01] obj1->getName()=\"%s\"\n", obj1->getName());  printf(" (\*obj1).getName()=\"%s\"\n", (\*obj1).getName());  CSmartPtr<CTest> obj2 = new CTest("次郎");  printf("[02] obj2->getName()=\"%s\"\n", obj2->getName());  CSmartPtr<CTest> obj3 = new CTest("三郎");  printf("[03] obj3->getName()=\"%s\"\n", obj3->getName());  {  printf("(begin block)\n");  CSmartPtr<CTest> obj4 = new CTest("四朗");  printf("[04] obj4->getName()=\"%s\"\n", obj4->getName());  CSmartPtr<CTest> obj5 = new CTest("五郎");  printf("[05] obj5->getName()=\"%s\"\n", obj5->getName());  printf("obj2 = obj4\n");  obj2 = obj4;//スマートポインターの代入：「次郎」の delete が行われる  printf("[06] obj1->getName()=\"%s\"\n", obj1->getName());  printf(" obj2->getName()=\"%s\"\n", obj2->getName());  printf(" obj3->getName()=\"%s\"\n", obj3->getName());  printf(" obj4->getName()=\"%s\"\n", obj4->getName());  printf(" obj5->getName()=\"%s\"\n", obj5->getName());  printf("obj1 = obj2\n");  obj1 = obj2;//スマートポインターの代入：「太郎」の delete が行われる  printf("[07] obj1->getName()=\"%s\"\n", obj1->getName());  printf(" obj2->getName()=\"%s\"\n", obj2->getName());  printf(" obj3->getName()=\"%s\"\n", obj3->getName());  printf(" obj4->getName()=\"%s\"\n", obj4->getName());  printf(" obj5->getName()=\"%s\"\n", obj5->getName());  //ブロック終了：「五郎」のdelete が行われる  printf("(end block)\n");  }  printf("[08] obj1->getName()=\"%s\"\n", obj1->getName());  printf(" obj2->getName()=\"%s\"\n", obj2->getName());  printf(" obj3->getName()=\"%s\"\n", obj3->getName());  printf("obj3 = nullptr\n");  obj3 = nullptr;//obj3を明示的に解放：「三郎」のdelete が行われる  printf("[09] obj1->getName()=\"%s\"\n", obj1->getName());  printf(" obj2->getName()=\"%s\"\n", obj2->getName());  // printf(" obj3->getName()=\"%s\"\n", obj3->getName());//エラー  printf("obj2 = nullptr\n");  obj2 = nullptr;//obj2を明示的に解放：「四朗」のdelete は行われない  printf("[10] obj1->getName()=\"%s\"\n", obj1->getName());  // printf(" obj2->getName()=\"%s\"\n", obj2->getName());//エラー  // printf(" obj3->getName()=\"%s\"\n", obj3->getName());//エラー  //obj1は明示的な解放しない  printf("(end function)\n");  //関数終了：関数を抜ける時に「四朗」の delete が行われる  } |

↓（処理結果）

|  |
| --- |
| - testProxy2() -  (begin function)  CTest::Ctest("太郎")  [01] obj1->getName()="太郎"  (\*obj1).getName()="太郎"  CTest::Ctest("次郎")  [02] obj2->getName()="次郎"  CTest::Ctest("三郎")  [03] obj3->getName()="三郎"  (begin block)  CTest::Ctest("四朗")  [04] obj4->getName()="四朗"  CTest::Ctest("五郎")  [05] obj5->getName()="五郎"  obj2 = obj4  CTest::~Ctest() : m\_name="次郎" ←処理と見比べると、参照が失われたことが自動的に判断されて  [06] obj1->getName()="太郎" delete が実行されていることがわかる  obj2->getName()="四朗"  obj3->getName()="三郎"  obj4->getName()="四朗"  obj5->getName()="五郎"  obj1 = obj2  CTest::~Ctest() : m\_name="太郎"  [07] obj1->getName()="四朗"  obj2->getName()="四朗"  obj3->getName()="三郎"  obj4->getName()="四朗"  obj5->getName()="五郎"  (end block)  CTest::~Ctest() : m\_name="五郎"  [08] obj1->getName()="四朗"  obj2->getName()="四朗"  obj3->getName()="三郎"  obj3 = nullptr  CTest::~Ctest() : m\_name="三郎"  [09] obj1->getName()="四朗"  obj2->getName()="四朗"  obj2 = nullptr  [10] obj1->getName()="四朗"  (end function)  CTest::~Ctest() : m\_name="四朗" |

## Decoratorパターン（Filter）

別紙の「カメラ処理の効率化手法」にて解説。

なお、DecoratorパターンとProxyパターンは構造的に同じである。両者は目的が異なるため、区別される。

Proxyパターンが「機能の置き換え（代理）」を目的としたクラスの「利用者」側の用いるパターンであるのに対して、Decoratorパターンは「機能の追加」を目的としたクラスの「提供者」側が用いるパターンである。

## Strategyパターン（Policy）

別紙の「効果的なテンプレートテクニック」にて解説。

## Template Methodパターン

別紙の「効果的なテンプレートテクニック」にて解説。

■■以上■■

索引

２

23のパターン 1

Ａ

Abstract Factory 2

Adapter 2, 8

Ｂ

Bridge 2

Builder 2

Ｃ

Chain of Responsibility 3

Commadn 3

Composite 2

Ｄ

Decorator 2

Delegate 4

Ｆ

Facade 2

Factory Method 2

Filter 2

Flyweight 2

Ｇ

GoF 1

Ｉ

Interpreter 3

Iterator 3

Ｌ

Listner 3, 4

Ｍ

Mediator 3

Memento 3

Ｏ

Observer 3, 4

Ｐ

Policy 24

Prototype 2

Proxy 3, 14

Ｓ

Singleton 2

State 3

Strategy 4, 24

Ｔ

Template Method 4, 25

Ｖ

Visitor 4

Ｗ

Wrapper 3, 14

あ

アダプター 2, 8

アブストラクトファクトリー 2

い

イテレータ 3

イベントコールバック 4

インタープリタ 3

お

オブザーバー 3, 4

こ

コマンド 3

コンポジット 2

し

シングルトン 2

す

ステート 3

ステートマシン 3

ストラテジー 4, 24

スマートポインター 14

ち

チェインオブレスポンシビリティ 3

て

デコレーター 2

デザインパターン 1

デリゲート 4

テンプレートメソッド 4, 25

ひ

ビジター 4

ビルダー 2

ふ

ファクトリーメソッド 2

ファサード 2

フライウェイト 2

ブリッジ 2

プロキシー 3, 14

プロトタイプ 2

め

メディエイター 3

メメント 3

ゆ

有限オートマトン 3

以　上