デザインパターンの活用

– デザインパターンをゲームプログラミングに役立てる –

2014 年 2 月 24 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 24 日	板垣 衛	（初稿）

■ 目次

■ 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」をプログラマー間で共有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」「共通の言語」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 種類のパターンがとくに有名。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」（フィルター）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」（ラッパー）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。
  イテレータは非常に広範囲に用いられている。
- 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」（リスナー）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などに使われる。

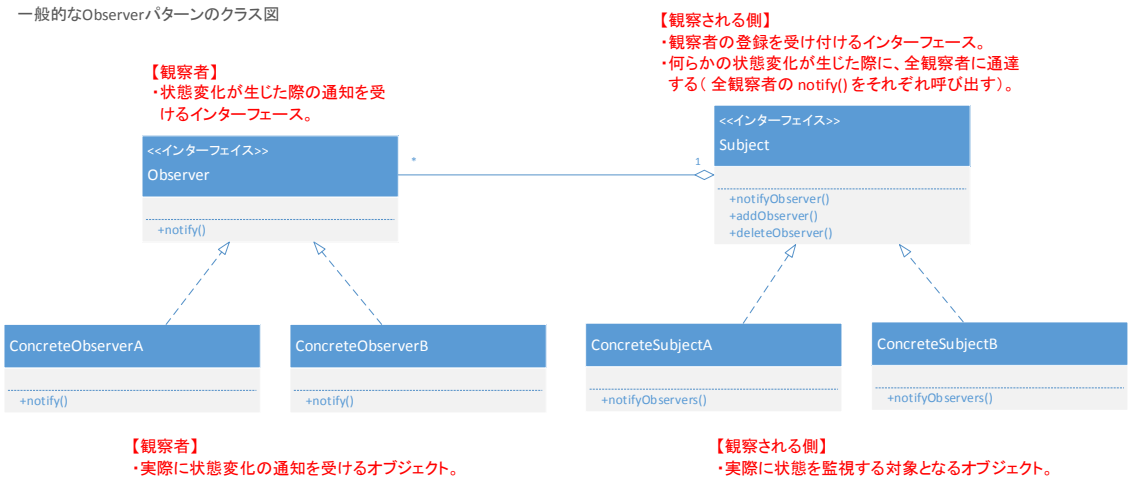
- **Strategy** ..... **【ストラテジー】（戦略）**
  データ構造に対して適用する一種のアルゴリズムをカプセル化し、アルゴリズムの切り替えを容易にする。「Policy」（ポリシー）と同等。
- **Template Method** ..... **【テンプレートメソッド】（ひな形メソッド）**
  あるアルゴリズムの途中経過で必要な処理を抽象メソッドに委ね、その実装を変えることで処理が変えられるようにする。
- **Visitor** ..... **【ビジター】（訪問者）**
  データ構造を保持するクラスと、それに対して処理を行うクラスを分離する。

■ 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
}
```

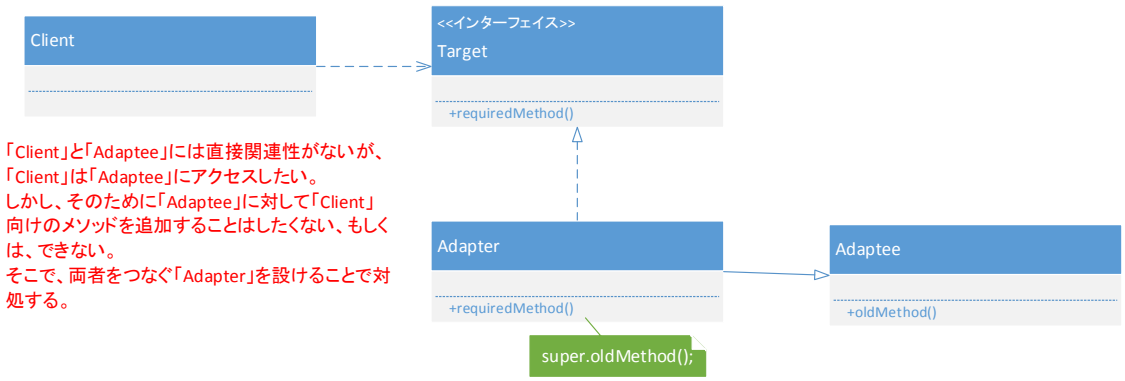
↓（処理結果）

```
- 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
```

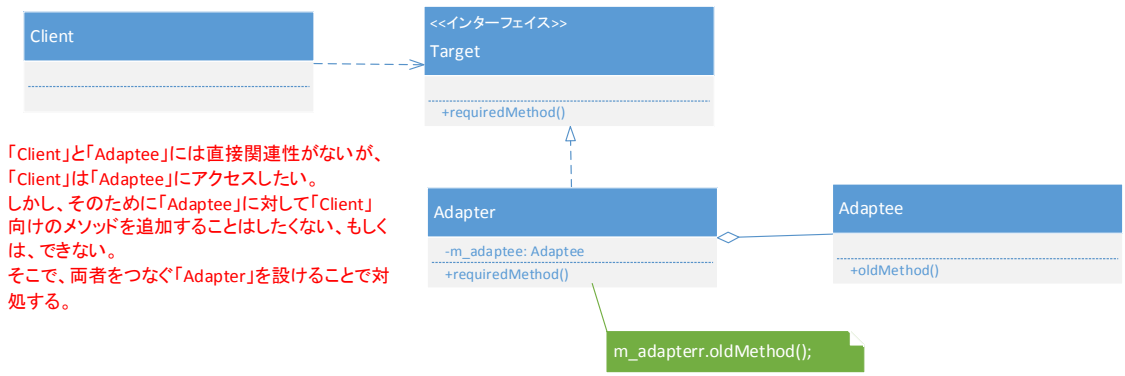
▼ Adapter パターン

・ 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。

一般的なAdapterパターンのクラス図（1）：継承を用いるパターン



一般的なAdapterパターンのクラス図（2）：委譲を用いるパターン



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）

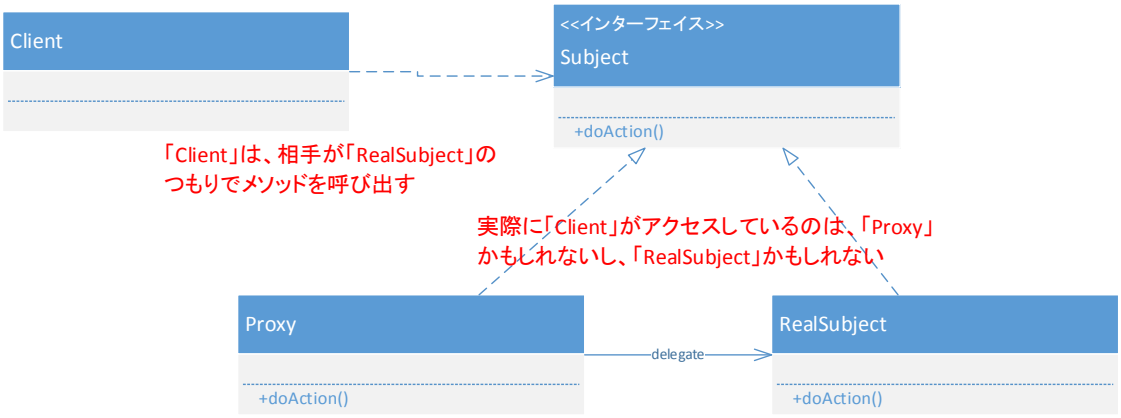
```
- testAdapter() -

[0] "NPC"太郎"" pos=(10.0, 11.0, 12.0) rot=(0.1)
[1] "宝箱(103)" pos=(35.0, 35.0, 35.0) rot=(1.3)
[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)
```

←一回のループ処理で、2種類のオブジェクトの処理が入り乱れて呼びだれていることがわかる

▼ 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。

一般的なProxyパターンのクラス図



「Proxy」は「Client」に対して、「RealSubject」のように見せかけることができる（代理人になる）。
「Proxy」（代理人）は、「Client」からの要求に基づいて、その内容次第では「RealSubject」に働きかけることもある。

実装例①：（イテレータ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:
	//--------------------
	//イテレータクラス
```

```
	class CIterator
	{
	public:
		//イテレータ用型宣言
		typedef CIterator type;
		typedef ICharacter* value_type;
	public:
		//イテレータ用オペレータ
		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; }		//先頭イテレータを返す
	CIterator end() { CIterator ite(*this, m_charaNum); return ite; }//終端イテレータを返す
	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、プロキシーを返す
			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、カメラ処理の拡張方法として解説。

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、「ポリシー」の実装方法として解説。

▼ 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、「CRTP」の活用方法として解説。

▼ Singleton パターン

別紙の「効率化と安全性のためのロック制御」にて、「スレッドセーフなシングルトンパターン」の実装方法を解説。

Proxy パターンと組み合わせた実装を行っている。

▼ Abstract Factory パターン ／ Chain Of Responsibility パターン

別紙の「開発の効率化と安全性のためのリソース管理」にて、Abstact Factory パターンを応用したリソースのビルド処理を解説。

Abstract Factory パターンに、Chain Of Responsibility パターンと Singleton パターンを組み合わせた実装のサンプルを示している。

▼ Iterator パターン

Iterarot パターンの説明を明記したドキュメントはとくに用意していない。

その代わり、STL 互換、および、STL と同等のイテレータは随所で使用している。

別紙の「プログラミング禁則事項」には、STL を禁止する代わりに、STL のアルゴリズムを利用するためのコンテナクラスの自作方法の紹介としている。

また、別紙の「効果的なテンプレートテクニック」には、コーディングを最適化するために「for_each」を自作する方法を示している。

▼ Composite パターン

Composite パターンは、Iterator パターンと同様に、もはや普遍化したパターンと言える。説明を明記したドキュメントはとくに用意していないが、使いどころは多い。

ファイルシステムのディレクトリ構造を扱うのに適したパターン。

階層構造で連結されたオブジェクトのそれぞれに固有の振る舞いを実装する場合に使用する。別紙の「ゲーム全体を円滑に制御するためのシーン管理」で説明する「シーングラフ」のようなデータに応用できる。

▼ Prototype パターン

Prototype パターンのサンプルはとくに用意していない。

Prototype は初期化のパターンを登録しておいて、新規オブジェクト生成時にそのクローンを用いる手法である。

コンストラクタやフィールド操作でオブジェクトを初期化する手間を省き、再利用したい状態のオブジェクトを保存しておくことで、それと同じ状態の新規オブジェクトを素早く生成する。

具体的な使用例として、別紙の「ゲーム制御のためのメモリ管理方針」にて、スラブアロケータと組み合わせた手法を説明する。

▼ Flyweight パターン

Flyweight パターンのサンプルはとくに用意していない。

Flyweight パターンは、（主に）不変のオブジェクトを共有するために用いる。最初にオブジェクトが必要になった時にインスタンスを生成してプールし、以後同じオブジェクトが必要になったらプールから取り出して再利用・共有する。

ゲームにも使いどころは十分考えられる。別紙の「開発の効率化と安全性のためのリソース管理」で示すリソース共有も一種の Flyweight パターンである。

具体的な使用例として、別紙の「ゲーム制御のためのメモリ管理方針」にて、スラブアロケータ、および、Prototype パターンと組み合わせた手法を説明する。

■■以上■■

■ 索引

T

V

W

あ

い

お

こ

し

す

ち

て

ひ

ふ

め

ゆ

デザインパターンの活用

以　上