***Основные паттерны ООП (на примере C++)***

По материалам сайта

[http://cpp-reference.ru/patterns/](http://cpp-reference.ru/patterns/introduction/)

Паттерны поведения

<http://cpp-reference.ru/patterns/behavioral-patterns/strategy/>

<http://cpp-reference.ru/patterns/behavioral-patterns/template-method/>

Порождающие паттерны

<http://cpp-reference.ru/patterns/creational-patterns/abstract-factory/>

<http://cpp-reference.ru/patterns/creational-patterns/factory-method/>

<http://cpp-reference.ru/patterns/creational-patterns/singleton/>

Структурные паттерны

<http://cpp-reference.ru/patterns/structural-patterns/adapter/>

<http://cpp-reference.ru/patterns/structural-patterns/decorator/>

<http://cpp-reference.ru/patterns/structural-patterns/facade/>

В русскоязычной литературе обычно встречаются несколько вариантов перевода оригинального названия design patterns - паттерны проектирования, шаблоны проектирования

**Паттерн Template Method (шаблонный метод)**

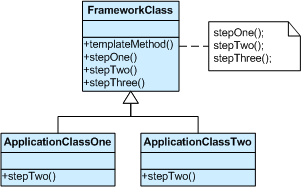
Назначение паттерна Template Method

Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры.

Базовый класс определяет шаги алгоритма с помощью абстрактных операций, а производные классы их реализуют.

Решаемая проблема

Имеются два разных, но в тоже время очень похожих компонента. Вы хотите внести изменения в оба компонента, избежав дублирования кода.



## Реализация паттерна Template Method

1. Стандартизуйте основу алгоритма в шаблонном методе базового класса.
2. Для шагов, требующих особенной реализации, определите "замещающие" методы.
3. Производные классы реализуют "замещающие" методы.

#include <iostream>

using namespace std;

class Base

{

    void a()

    {

        cout << "a  ";

    }

    void c()

    {

        cout << "c  ";

    }

    void e()

    {

        cout << "e  ";

    }

    // 2. Для шагов, требующих особенной реализации, определите

    //    "замещающие" методы.

    virtual void ph1() = 0;

    virtual void ph2() = 0;

  public:

    // 1. Стандартизуйте основу алгоритма в шаблонном методе

    //    базового класса

    void execute()

    {

        a();

        ph1();

        c();

        ph2();

        e();

    }

};

class One: public Base

{

   // 3. Производные классы реализуют "замещающие" методы.

     /\*virtual\*/void ph1()

    {

        cout << "b  ";

    }

     /\*virtual\*/void ph2()

    {

        cout << "d  ";

    }

};

class Two: public Base

{

     /\*virtual\*/void ph1()

    {

        cout << "2  ";

    }

     /\*virtual\*/void ph2()

    {

        cout << "4  ";

    }

};

int main()

{

  Base \*array[] =

  {

     new One(), new Two()

  };

  for (int i = 0; i < 2; i++)

  {

    array[i]->execute();

    cout << '\n';

  }

}

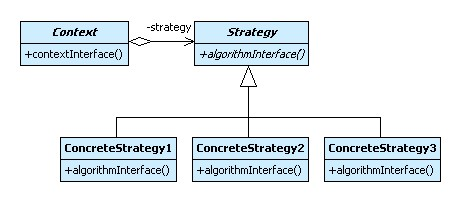
Вывод программы:

a b c d e a 2 c 4 e

См. example\_singleton\_template\_method.cpp.cpp

**Структура паттерна Strategy**

UML-диаграмма классов паттерна Strategy



#include <iostream>

#include <string>

using namespace std;

// Иерархия классов, определяющая алгоритмы сжатия файлов

class Compression

{

public:

virtual ~Compression() {}

virtual void compress( const string & file ) = 0;

};

class ZIP\_Compression : public Compression

{

public:

void compress( const string & file ) {

cout << "ZIP compression" << endl;

}

};

class ARJ\_Compression : public Compression

{

public:

void compress( const string & file ) {

cout << "ARJ compression" << endl;

}

};

class RAR\_Compression : public Compression

{

public:

void compress( const string & file ) {

cout << "RAR compression" << endl;

}

};

// Класс для использования

class Compressor

{

public:

Compressor( Compression\* comp): p(comp) {}

~Compressor() { delete p; }

void compress( const string & file ) {

p->compress( file);

}

private:

Compression\* p;

};

int main()

{

Compressor\* p = new Compressor( new ZIP\_Compression);

p->compress( "file.txt");

delete p;

return 0;

}

Достоинства паттерна Strategy

Систему проще поддерживать и модифицировать, так как семейство алгоритмов перенесено в отдельную иерархию классов.

Паттерн Strategy предоставляет возможность замены одного алгоритма другим в процессе выполнения программы.

Паттерн Strategy позволяет скрыть детали реализации алгоритмов от клиента.

Недостатки паттерна Strategy

Для правильной настройки системы пользователь должен знать об особенностях всех алгоритмов.

Число классов в системе, построенной с применением паттерна Strategy, возрастает.

**Реализация паттерна Factory Method**

Рассмотрим оба варианта реализации паттерна Factory Method на примере процесса порождения военных персонажей для нашей стратегической игры. Ее подробное описание можно найти в разделе Порождающие паттерны. Для упрощения демонстрационного кода будем создавать военные персонажи для некой абстрактной армии без учета особенностей воюющих сторон.

**FactoryMethod.cpp**

Реализация паттерна Factory Method на основе обобщенного конструктора

// #include <iostream>

#include <vector>

enum Warrior\_ID { Infantryman\_ID=0, Archer\_ID, Horseman\_ID };

// Иерархия классов игровых персонажей

class Warrior

{

public:

virtual void info() = 0;

virtual ~Warrior() {}

// Параметризированный статический фабричный метод

static Warrior\* createWarrior( Warrior\_ID id );

};

class Infantryman: public Warrior

{

public:

void info() {

cout << "Infantryman" << endl;

}

};

class Archer: public Warrior

{

public:

void info() {

cout << "Archer" << endl;

}

};

class Horseman: public Warrior

{

public:

void info() {

cout << "Horseman" << endl;

}

};

// Реализация параметризированного фабричного метода

Warrior\* Warrior::createWarrior( Warrior\_ID id )

{

Warrior \* p;

switch (id)

{

case Infantryman\_ID:

p = new Infantryman();

break;

case Archer\_ID:

p = new Archer();

break;

case Horseman\_ID:

p = new Horseman();

break;

default:

assert( false);

}

return p;

};

// Создание объектов при помощи параметризированного фабричного метода

int main()

{

vector<Warrior\*> v;

v.push\_back( Warrior::createWarrior( Infantryman\_ID));

v.push\_back( Warrior::createWarrior( Archer\_ID));

v.push\_back( Warrior::createWarrior( Horseman\_ID));

for(int i=0; i<v.size(); i++)

v[i]->info();

// ...

}

Представленный вариант паттерна Factory Method пользуется популярностью благодаря своей простоте. В нем статический фабричный метод createWarrior() определен непосредственно в полиморфном базовом классе Warrior. Этот фабричный метод является параметризированным, то есть для создания объекта некоторого типа в createWarrior() передается соответствующий идентификатор типа.

**С точки зрения "чистоты" объектно-ориентированного кода у этого варианта есть следующие недостатки:**

**Так как код по созданию объектов всех возможных типов сосредоточен в статическом фабричном методе класса Warrior, то базовый класс Warrior обладает знанием обо всех производных от него классах, что является нетипичным для объектно-ориентированного подхода.**

**Подобное использование оператора switch (как в коде фабричного метода createWarrior()) в объектно-ориентированном программировании также не приветствуется.**

Указанные недостатки отсутствуют в классической реализации паттерна Factory Method.

**FactoryMethod2.cpp**

Классическая реализация паттерна Factory Method

#include <iostream>

#include <vector>

// Иерархия классов игровых персонажей

class Warrior

{

public:

virtual void info() = 0;

virtual ~Warrior() {}

};

class Infantryman: public Warrior

{

public:

void info() {

cout << "Infantryman" << endl;

};

};

class Archer: public Warrior

{

public:

void info() {

cout << "Archer" << endl;

};

};

class Horseman: public Warrior

{

public:

void info() {

cout << "Horseman" << endl;

};

};

// Фабрики объектов

class Factory

{

public:

virtual Warrior\* createWarrior() = 0;

virtual ~Factory() {}

};

class InfantryFactory: public Factory

{

public:

Warrior\* createWarrior() {

return new Infantryman;

}

};

class ArchersFactory: public Factory

{

public:

Warrior\* createWarrior() {

return new Archer;

}

};

class CavalryFactory: public Factory

{

public:

Warrior\* createWarrior() {

return new Horseman;

}

};

// Создание объектов при помощи фабрик объектов

int main()

{

InfantryFactory\* infantry\_factory = new InfantryFactory;

ArchersFactory\* archers\_factory = new ArchersFactory ;

CavalryFactory\* cavalry\_factory = new CavalryFactory ;

vector<Warrior\*> v;

v.push\_back( infantry\_factory->createWarrior());

v.push\_back( archers\_factory->createWarrior());

v.push\_back( cavalry\_factory->createWarrior());

for(int i=0; i<v.size(); i++)

v[i]->info();

// ...

}

Классический вариант паттерна Factory Method использует идею полиморфной фабрики. Специально выделенный для создания объектов полиморфный базовый класс Factory объявляет интерфейс фабричного метода createWarrior(), а производные классы его реализуют.

Представленный вариант паттерна Factory Method является наиболее распространенным, но не единственным. Возможны следующие вариации:

Класс Factory имеет реализацию фабричного метода createWarrior() по умолчанию.

Фабричный метод createWarrior() класса Factory параметризирован типом создаваемого объекта (как и у представленного ранее, простого варианта Factory Method) и имеет реализацию по умолчанию. В этом случае, производные от Factory классы необходимы лишь для того, чтобы определить нестандартное поведение createWarrior().

Результаты применения паттерна Factory Method

Достоинства паттерна Factory Method

Создает объекты разных типов, позволяя системе оставаться независимой как от самого процесса создания, так и от типов создаваемых объектов.

Недостатки паттерна Factory Method

В случае классического варианта паттерна даже для порождения единственного объекта необходимо создавать соответствующую фабрику.

Реализация паттерна Abstract Factory

Приведем реализацию паттерна Abstract Factory для военной стратегии "Пунические войны". При этом предполагается, что число и типы создаваемых в начале игры боевых единиц идентичны для обеих армий. Подробное описание этой игры можно найти в разделе Порождающие паттерны.

#include <iostream>

#include <vector>

// Абстрактные базовые классы всех возможных видов воинов

class Infantryman

{

public:

virtual void info() = 0;

virtual ~Infantryman() {}

};

class Archer

{

public:

virtual void info() = 0;

virtual ~Archer() {}

};

class Horseman

{

public:

virtual void info() = 0;

virtual ~Horseman() {}

};

// Классы всех видов воинов Римской армии

class RomanInfantryman: public Infantryman

{

public:

void info() {

cout << "RomanInfantryman" << endl;

}

};

class RomanArcher: public Archer

{

public:

void info() {

cout << "RomanArcher" << endl;

}

};

class RomanHorseman: public Horseman

{

public:

void info() {

cout << "RomanHorseman" << endl;

}

};

// Классы всех видов воинов армии Карфагена

class CarthaginianInfantryman: public Infantryman

{

public:

void info() {

cout << "CarthaginianInfantryman" << endl;

}

};

class CarthaginianArcher: public Archer

{

public:

void info() {

cout << "CarthaginianArcher" << endl;

}

};

class CarthaginianHorseman: public Horseman

{

public:

void info() {

cout << "CarthaginianHorseman" << endl;

}

};

// Абстрактная фабрика для производства воинов

class ArmyFactory

{

public:

virtual Infantryman\* createInfantryman() = 0;

virtual Archer\* createArcher() = 0;

virtual Horseman\* createHorseman() = 0;

virtual ~ArmyFactory() {}

};

// Фабрика для создания воинов Римской армии

class RomanArmyFactory: public ArmyFactory

{

public:

Infantryman\* createInfantryman() {

return new RomanInfantryman;

}

Archer\* createArcher() {

return new RomanArcher;

}

Horseman\* createHorseman() {

return new RomanHorseman;

}

};

// Фабрика для создания воинов армии Карфагена

class CarthaginianArmyFactory: public ArmyFactory

{

public:

Infantryman\* createInfantryman() {

return new CarthaginianInfantryman;

}

Archer\* createArcher() {

return new CarthaginianArcher;

}

Horseman\* createHorseman() {

return new CarthaginianHorseman;

}

};

// Класс, содержащий всех воинов той или иной армии

class Army

{

public:

~Army() {

int i;

for(i=0; i<vi.size(); ++i) delete vi[i];

for(i=0; i<va.size(); ++i) delete va[i];

for(i=0; i<vh.size(); ++i) delete vh[i];

}

void info() {

int i;

for(i=0; i<vi.size(); ++i) vi[i]->info();

for(i=0; i<va.size(); ++i) va[i]->info();

for(i=0; i<vh.size(); ++i) vh[i]->info();

}

vector<Infantryman\*> vi;

vector<Archer\*> va;

vector<Horseman\*> vh;

};

// Здесь создается армия той или иной стороны

class Game

{

public:

Army\* createArmy( ArmyFactory& factory ) {

Army\* p = new Army;

p->vi.push\_back( factory.createInfantryman());

p->va.push\_back( factory.createArcher());

p->vh.push\_back( factory.createHorseman());

return p;

}

};

int main()

{

Game game;

RomanArmyFactory ra\_factory;

CarthaginianArmyFactory ca\_factory;

Army \* ra = game.createArmy( ra\_factory);

Army \* ca = game.createArmy( ca\_factory);

cout << "Roman army:" << endl;

ra->info();

cout << "\nCarthaginian army:" << endl;

ca->info();

// ...

}

Вывод программы будет следующим:

Roman army:

RomanInfantryman

RomanArcher

RomanHorseman

Carthaginian army:

CarthaginianInfantryman

CarthaginianArcher

CarthaginianHorseman

Достоинства паттерна Abstract Factory

Скрывает сам процесс порождения объектов, а также делает систему независимой от типов создаваемых объектов, специфичных для различных семейств или групп (пользователи оперируют этими объектами через соответствующие абстрактные интерфейсы).

Позволяет быстро настраивать систему на нужное семейство создаваемых объектов. В случае многоплатформенного графического приложения для перехода на новую платформу, то есть для замены графических элементов (кнопок, меню, полос прокрутки) одного стиля другим достаточно создать нужный подкласс абстрактной фабрики. При этом условие невозможности одновременного использования элементов разных стилей для некоторой платформы будет выполнено автоматически.

Недостатки паттерна Abstract Factory

Трудно добавлять новые типы создаваемых продуктов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован. Например, если для нашей стратегической игры нужно будет ввести новый вид военной единицы - осадные орудия, то надо будет добавить новый фабричный метод, объявив его интерфейс в полиморфном базовом классе AbstractFactory и реализовав во всех подклассах. Снять это ограничение можно следующим образом. Все создаваемые объекты должны наследовать от общего абстрактного базового класса, а в единственный фабричный метод в качестве параметра необходимо передавать идентификатор типа объекта, который нужно создать. Однако в этом случае необходимо учитывать следующий момент. Фабричный метод создает объект запрошенного подкласса, но при этом возвращает его с интерфейсом общего абстрактного класса в виде ссылки или указателя, поэтому для такого объекта будет затруднительно выполнить какую-либо операцию, специфичную для подкласса.

Паттерн Adapter (адаптер, wrapper, обертка)

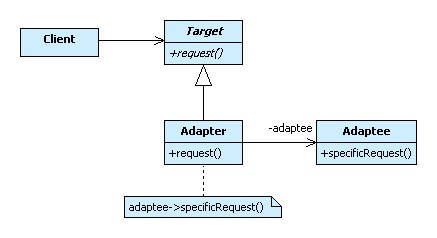
Назначение паттерна Adapter

Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter (адаптер).

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

Классическая реализация паттерна Adapter



Приведем реализацию паттерна Adapter. Для примера выше адаптируем показания температурного датчика системы климат-контроля, переведя их из градусов Фаренгейта в градусы Цельсия (предполагается, что код этого датчика недоступен для модификации).

#include <iostream>

using namespace std;

// Уже существующий класс температурного датчика окружающей среды

class FahrenheitSensor

{

public:

// Получить показания температуры в градусах Фаренгейта

float getFahrenheitTemp() {

float t = 32.0;

// ... какой то код

return t;

}

};

class Sensor

{

public:

virtual ~Sensor() {}

virtual float getTemperature() = 0;

};

class Adapter : public Sensor

{

public:

Adapter( FahrenheitSensor\* p ) : p\_fsensor(p) {

}

~Adapter() {

delete p\_fsensor;

}

float getTemperature() {

return (p\_fsensor->getFahrenheitTemp()-32.0)\*5.0/9.0;

}

private:

FahrenheitSensor\* p\_fsensor;

};

int main()

{

Sensor\* p = new Adapter( new FahrenheitSensor);

cout << "Celsius temperature = " << p->getTemperature() << endl;

delete p;

return 0;

}

**Паттерн Фасад**

Клиент использует только "фасад".

Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Помните, Adapter делает работающими вместе два существующих интерфейса, не создавая новых.

Adapter и Facade в являются "обертками", однако эти "обертки" разных типов. Цель Facade – создание более простого интерфейса, цель Adapter – адаптация существующего интерфейса. Facade обычно "обертывает" несколько объектов, Adapter "обертывает" один объект.

Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.

Реализация паттерна Facade

Разбиение системы на компоненты позволяет снизить ее сложность. Ослабить связи между компонентами системы можно с помощью паттерна Facade. Объект "фасад" предоставляет единый упрощенный интерфейс к компонентам системы.

В примере ниже моделируется система сетевого обслуживания. Фасад FacilitiesFacade скрывает внутреннюю структуру системы. Пользователь, сделав однажды запрос на обслуживание, затем 1-2 раза в неделю в течение 5 месяцев справляется о ходе выполнения работ до тех пор, пока его запрос не будет полностью обслужен.

#include <iostream>

class MisDepartment

{

public:

void submitNetworkRequest()

{

\_state = 0;

}

bool checkOnStatus()

{

\_state++;

if (\_state == Complete)

return 1;

return 0;

}

private:

enum States

{

Received, DenyAllKnowledge, ReferClientToFacilities,

FacilitiesHasNotSentPaperwork, ElectricianIsNotDone,

ElectricianDidItWrong, DispatchTechnician, SignedOff,

DoesNotWork, FixElectriciansWiring, Complete

};

int \_state;

};

class ElectricianUnion

{

public:

void submitNetworkRequest()

{

\_state = 0;

}

bool checkOnStatus()

{

\_state++;

if (\_state == Complete)

return 1;

return 0;

}

private:

enum States

{

Received, RejectTheForm, SizeTheJob, SmokeAndJokeBreak,

WaitForAuthorization, DoTheWrongJob, BlameTheEngineer,

WaitToPunchOut, DoHalfAJob, ComplainToEngineer,

GetClarification, CompleteTheJob, TurnInThePaperwork,

Complete

};

int \_state;

};

class FacilitiesDepartment

{

public:

void submitNetworkRequest()

{

\_state = 0;

}

bool checkOnStatus()

{

\_state++;

if (\_state == Complete)

return 1;

return 0;

}

private:

enum States

{

Received, AssignToEngineer, EngineerResearches,

RequestIsNotPossible, EngineerLeavesCompany,

AssignToNewEngineer, NewEngineerResearches,

ReassignEngineer,EngineerReturns,

EngineerResearchesAgain, EngineerFillsOutPaperWork,

Complete

};

int \_state;

};

class FacilitiesFacade

{

public:

FacilitiesFacade()

{

\_count = 0;

}

void submitNetworkRequest()

{

\_state = 0;

}

bool checkOnStatus()

{

\_count++;

/\* Запрос на обслуживание получен \*/

if (\_state == Received)

{

\_state++;

/\* Перенаправим запрос инженеру \*/

\_engineer.submitNetworkRequest();

cout << "submitted to Facilities - " << \_count

<< " phone calls so far" << endl;

}

else if (\_state == SubmitToEngineer)

{

/\* Если инженер свою работу выполнил,

перенаправим запрос электрику \*/

if (\_engineer.checkOnStatus())

{

\_state++;

\_electrician.submitNetworkRequest();

cout << "submitted to Electrician - " << \_count

<< " phone calls so far" << endl;

}

}

else if (\_state == SubmitToElectrician)

{

/\* Если электрик свою работу выполнил,

перенаправим запрос технику \*/

if (\_electrician.checkOnStatus())

{

\_state++;

\_technician.submitNetworkRequest();

cout << "submitted to MIS - " << \_count

<< " phone calls so far" << endl;

}

}

else if (\_state == SubmitToTechnician)

{

/\* Если техник свою работу выполнил,

то запрос обслужен до конца \*/

if (\_technician.checkOnStatus())

return 1;

}

/\* Запрос еще не обслужен до конца \*/

return 0;

}

int getNumberOfCalls()

{

return \_count;

}

private:

enum States

{

Received, SubmitToEngineer, SubmitToElectrician,

SubmitToTechnician

};

int \_state;

int \_count;

FacilitiesDepartment \_engineer;

ElectricianUnion \_electrician;

MisDepartment \_technician;

};

int main()

{

FacilitiesFacade facilities;

facilities.submitNetworkRequest();

/\* Звоним, пока работа не выполнена полностью \*/

while (!facilities.checkOnStatus())

;

cout << "job completed after only "

<< facilities.getNumberOfCalls()

<< " phone calls" << endl;

}

Вывод программы:

submitted to Facilities - 1 phone calls so far

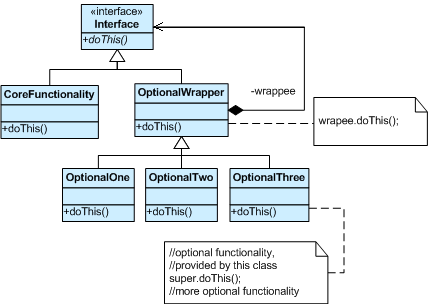
submitted to Electrician - 12 phone calls so far

submitted to MIS - 25 phone calls so far

job completed after only 35 phone calls

Пример паттерна Decorator

Паттерн Decorator динамически добавляет новые обязанности объекту. Украшения для новогодней елки являются примерами декораторов. Огни, гирлянды, игрушки и т.д. вешают на елку для придания ей праздничного вида. Украшения не меняют саму елку, а только делают ее новогодней.



Декоратором является OptionalWrapper

Паттерн Decorator: до и после

До

Используется следующая иерархия наследования:

#include <iostream>

using namespace std;

class A {

public:

virtual void do\_it() {

cout << 'A';

}

};

class AwithX: public A {

public:

/\*virtual\*/

void do\_it() {

A::do\_it();

do\_X();

};

private:

void do\_X() {

cout << 'X';

}

};

class AwithY: public A {

public:

/\*virtual\*/

void do\_it() {

A::do\_it();

do\_Y();

}

protected:

void do\_Y() {

cout << 'Y';

}

};

class AwithZ: public A {

public:

/\*virtual\*/

void do\_it() {

A::do\_it();

do\_Z();

}

protected:

void do\_Z() {

cout << 'Z';

}

};

class AwithXY: public AwithX, public AwithY

{

public:

/\*virtual\*/

void do\_it() {

AwithX::do\_it();

AwithY::do\_Y();

}

};

class AwithXYZ: public AwithX, public AwithY, public AwithZ

{

public:

/\*virtual\*/

void do\_it() {

AwithX::do\_it();

AwithY::do\_Y();

AwithZ::do\_Z();

}

};

int main() {

AwithX anX;

AwithXY anXY;

AwithXYZ anXYZ;

anX.do\_it();

cout << '\n';

anXY.do\_it();

cout << '\n';

anXYZ.do\_it();

cout << '\n';

}

Вывод программы:

1

2

3

AX

AXY

AXYZ

После

Заменим наследование делегированием.

Обсуждение. Используйте агрегирование вместо наследования для декорирования "основного" объекта. Тогда клиент сможет динамически добавлять новые обязанности объектам, в то время как архитектура, основанная на множественном наследовании, является статичной.

#include <iostream>

using namespace std;

class Interface {

public:

virtual ~Interface(){}

virtual void do\_it() = 0;

};

class DoCoreTask: public Interface {

public:

~DoCoreTask() {

cout << "DoCoreTask dtor" << '\n';

}

/\*virtual\*/

void do\_it() {

cout << "DoCoreTask";

}

};

class OptionalWrapper: public Interface {

public:

OptionalWrapper(Interface \*inner) {

m\_wrappee = inner;

}

~OptionalWrapper() {

delete m\_wrappee;

}

/\*virtual\*/

void do\_it() {

m\_wrappee->do\_it();

}

private:

Interface \*m\_wrappee;

};

class X: public OptionalWrapper {

public:

X(Interface \*core): OptionalWrapper(core){}

~X() {

cout << "X dtor" << " ";

}

/\*virtual\*/

void do\_it() {

OptionalWrapper::do\_it();

cout << 'X';

}

};

class Y: public OptionalWrapper {

public:

Y(Interface \*core): OptionalWrapper(core){}

~Y() {

cout << "Y dtor" << " ";

}

/\*virtual\*/

void do\_it() {

OptionalWrapper::do\_it();

cout << 'Y';

}

};

class Z: public OptionalWrapper {

public:

Z(Interface \*core): OptionalWrapper(core){}

~Z() {

cout << "Z dtor" << " ";

}

/\*virtual\*/

void do\_it() {

OptionalWrapper::do\_it();

cout << 'Z';

}

};

int main() {

Interface \*myX = new X(new DoCoreTask);

Interface \*myXY = new Y(new X(new DoCoreTask));

Interface \*myXYZ = new Z(new Y(new X(new DoCoreTask)));

myX->do\_it();

cout << '\n';

myXY->do\_it();

cout << '\n';

myXYZ->do\_it();

cout << '\n';

delete myX;

delete myXY;

delete myXYZ;

}

Вывод программы:

DoCoreTaskX

DoCoreTaskXY

DoCoreTaskXYZ

X dtor DoCoreTask dtor

Y dtor X dtor DoCoreTask dtor

Z dtor Y dtor X dtor DoCoreTask dtor

Паттерн проектирования Decorator по шагам

Создайте "наименьший общий знаменатель", делающий классы взаимозаменяемыми.

Создайте базовый класс второго уровня для реализации дополнительной функциональности.

Основной класс и класс-декоратор используют отношение "является".

Класс-декоратор "имеет" экземпляр "наименьшего общего знаменателя".

Класс Decorator делегирует выполнение операции объекту "имеет".

Для реализации каждой дополнительной функциональности создайте подклассы Decorator.

Подклассы Decorator делегируют выполнение операции базовому классу и реализуют дополнительную функциональность.

Клиент несет ответственность за конфигурирование нужной функциональности.

--

Decorator позволяет изменить внешний облик объекта, Strategy – его внутреннее содержание.

--

Singleton Мэйерса

// Singleton.h

class Singleton

{

private:

Singleton() {}

Singleton( const Singleton&);

Singleton& operator=( Singleton& );

public:

static Singleton& getInstance() {

static Singleton instance;

return instance;

}

};

Результаты применения паттерна Singleton

Достоинства паттерна Singleton

Класс сам контролирует процесс создания единственного экземпляра.

Паттерн легко адаптировать для создания нужного числа экземпляров.

Возможность создания объектов классов, производных от Singleton.

Недостатки паттерна Singleton

В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.