Перегрузка операторов new и delete для отдельных классов

Операторы new и delete можно перегрузить. Для этого есть несколько причин:

* Можно увеличить производительность за счёт кеширования: при удалении объекта не освобождать память, а сохранять указатели на свободные блоки, используя их для вновь конструируемых объектов.
* Можно выделять память сразу под несколько объектов.
* Можно реализовать собственный "сборщик мусора" (garbage collector).
* Можно вести лог выделения/освобождения памяти.

Операторы new и delete имеют следующие сигнатуры:

void \*operator new(size\_t size);

void operator delete(void \*p);

Оператор new принимает размер памяти, которую необходимо выделить, и возвращает указатель на выделенную память.

Оператор delete принимает указатель на память, которую нужно освободить.

class A {

public:

void \*operator new(size\_t size);

void operator delete(void \*p);

};

void \*A::operator new(size\_t size) {

printf("Allocated %d bytes\n", size);

return malloc(size);

}

void A::operator delete(void \*p) {

free(p);

}

Вместо функций malloc и free можно использовать глобальные операторы ::new и ::delete.

Рекомендуется не производить в операторе new (особенно в глобальном) какие-либо операции с объектами, которые могут вызвать оператор new. Например, для вывода текста используется функция printf, а не объект std::cout.

Операторы new и delete, объявленные внутри класса, функционируют **подобно статическим функциям** и вызываются для данного класса и его наследников, для которых эти операторы не переопределены.

Переопределение глобальных операторов new и delete

В некоторых случаях может потребоваться перегрузить глобальные операторы new и delete. Они находятся не в пространстве имён std, а в глобальном пространстве имён.

Глобальные операторы new и delete вызываются для примитивных типов и для классов, в которых они не переопределены. Они имеют такие же сигнатуры, что и рассмотренные выше операторы new и delete.

*Вызов операторов new и delete для разных типов.*

// Для примитивных типов вызываются глобальные ::new и ::delete

int \*i = new int;

delete i;

// Для класса A вызываются переопределённые A::new и A::delete

A \*a = new A;

delete a;

// Для класса C операторы new и delete не переопределены,

// поэтому вызываются глобальные ::new и ::delete

C \*c = new C;

delete c;

Оператор new и исключения

Стандартный оператор new генерирует исключение std::bad\_alloc в случае, если не удаётся выделить достаточно памяти.

*Листинг 4. Пример реализации оператора new, генерирующего исключение при нехватке памяти.*

void \*operator new(size\_t size) {

void \*p = malloc(size);

if (p == NULL) {

throw std::bad\_alloc();

}

return p;

}

Если конструктор класса генерирует любое исключение, память, выделенная под объект, автоматически освобождается.

Операторы new[] и delete[]

При создании массива вызывается другая форма операторов new и delete. Они имеют следующие сигнатуры:

void \*operator new[](size\_t size);

void operator delete[](void \*p);

Параметры этих функций аналогичны параметрам обычных операторов new и delete.

Массив объектов создаётся в два этапа:

1. Выделяется память, необходимая для размещения массива объектов.
2. Для каждого из элементов массива вызывается конструктор по умолчанию.

Удаляется массив объектов тоже в два этапа:

1. Для всех элементов массива вызываются деструкторы.
2. С помощью оператора delete освобождается память, занимаемая массивом.

*Листинг 5. Пример использования операторов new[] и delete[].*

// У класса A должен быть конструктор по умолчанию

A \*a = new A[4];

// ...

delete [] a;

Placement new

Можно вызвать new на уже выделенной памяти. Такая форма оператора new называется placement new. Она может понадобиться в операционных системах реального времени и встраиваемых системах, чтобы жёстко закрепить адрес объекта.

*Пример создания объекта с помощью placement new.*

// p - указатель на некий статический буфер

// Явный вызов конструктора

A \*a = new(p) A;

// Явный вызов деструктора

a->A::~A();

Реализация placement new выглядит следующим образом:

void \*operator new(size\_t size, void \*p) {

return p;

}

Оператор new, не генерирующий исключение

Если требуется, чтобы оператор new не генерировал исключение в случае нехватки памяти, а возвращал ноль, можно использовать оператор new с параметром std::nothrow. Этот параметр имеет тип std::nothrow\_t.

// Объявление

void \*operator new(size\_t size, const std::nothrow\_t &nt);

// Пример использования

A \*a = new(std::nothrow) A;

Другие формы оператора new

Синтаксис языка C++ позволяет создавать собственные формы оператора new.

// Объявление

void \*operator new(size\_t size, std::string &str);

// Пример использования

A \*a = new("Object a") A;

Отношения между классами

https://habrastorage.org/getpro/habr/post_images/ca8/dca/2a5/ca8dca2a537a1ec8044e88984c3f8b02.png — агрегация (aggregation) — описывает связь «часть»–«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».

https://habrastorage.org/getpro/habr/post_images/4de/b17/e66/4deb17e6696fc05e610d73ca47b1a49a.png — композиция (composition) — подвид агрегации, в которой «части» не могут существовать отдельно от «целого».

https://habrastorage.org/getpro/habr/post_images/43b/4e7/9ac/43b4e79ac9d10a4f8b57859b019c7c24.png — зависимость (dependency) — изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.

https://habrastorage.org/getpro/habr/post_images/aa6/efe/a5f/aa6efea5f552569c7fa6ce4e5603e684.png — обобщение (generalization) — отношение наследования или реализации интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

Существует три типа шаблонов:

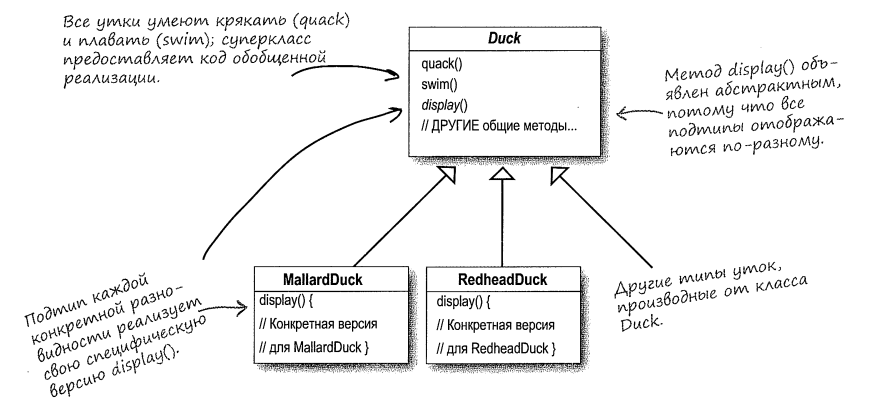
* структурные;
* порождающие;
* поведенческие.

**Структурные** шаблоны определяют отношения между классами и объектами, позволяя им работать совместно.

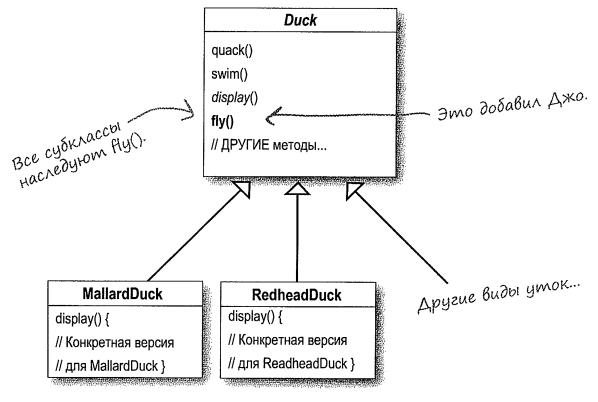
**Порождающие** шаблоны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

**Поведенческие** шаблоны используются для того, чтобы упростить взаимодействие между сущностями.

**ПАТЕРН СТРАТЕГИЯ**

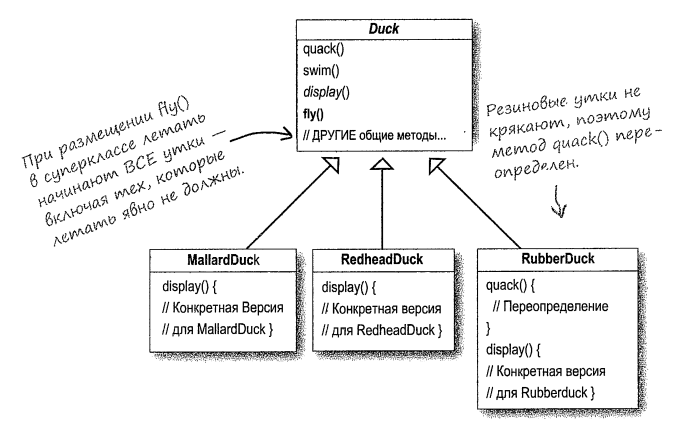
****

Необходимо, чтобы утки начали летать, поэтому добавляем метод fly(). Таким образом все утки научаться летать.



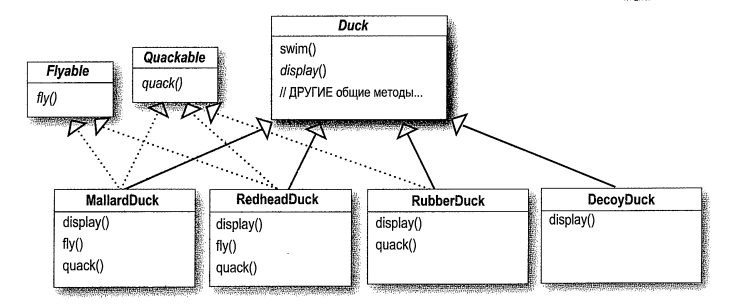
Возникла проблема, как оказалось резиновые утки не должны летать. Другими словами локальное изменение кода привело к нелокальному побочному эффекту.

Казалось бы, в этой ситуации наследование идеально подходит для повторного использования кода, но с сопровождением возникают проблемы.



Один из выходов это реализовать пустое переопределение метода fly для резиновых уток, но при добавлении ещё одного типа уток (например, деревянных) нужно будет переопределять метод quack()

Таким образом, поскольку не все утки должны крякать и летать, то эти методы желательно вынести из суперкласса. Например, следующим образом.

****

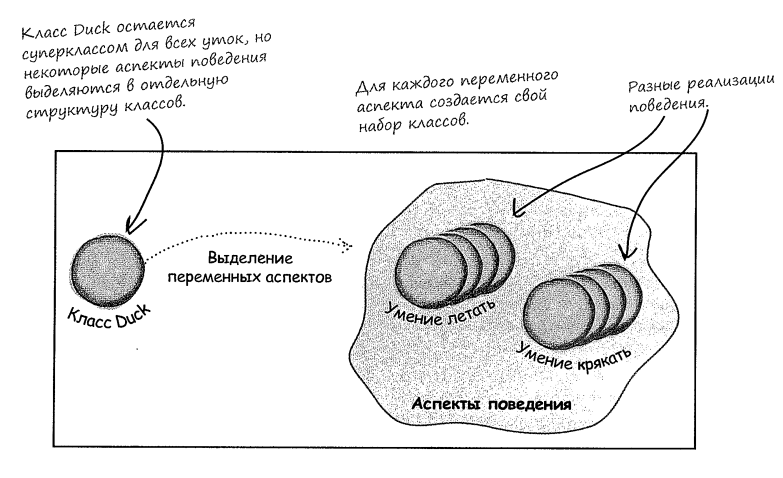
Реализовав их в виде интерфейсных классов. Это уже лучше, но тоже не выход. Так как разные типы уток могут летать и крякать по-разному, следовательно, придётся переопределять эти методы в соответствующих подклассах.

Кроме того, желательно, чтобы мы могли динамически во время выполнения менять поведение объектов (уток). Например, из всего множества летающих уток одну подстрелили, и она стала на время не летающей.

***Нам нужно сделать так, чтобы вносимые изменения оказывали минимальное влияние на существующий код.***

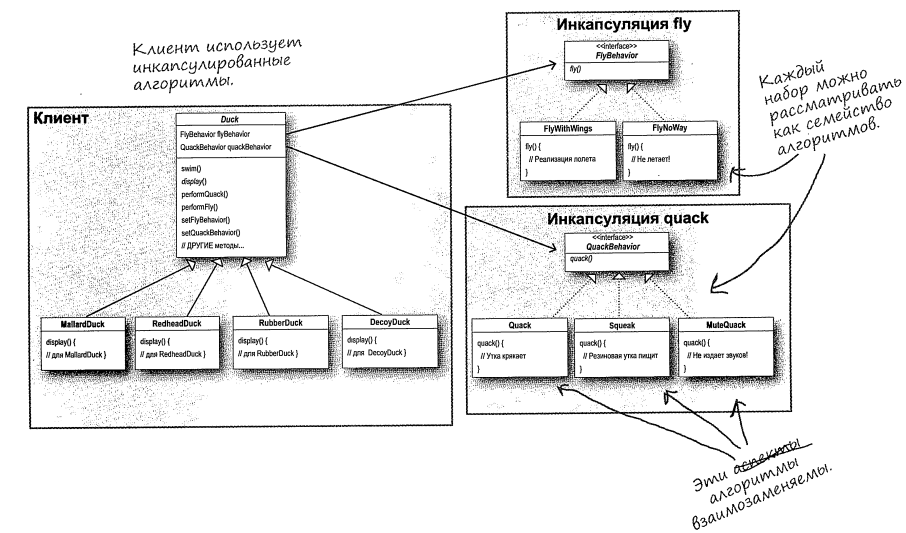
Другими словами, нужно инкапсулировать изменяемые аспекты программы, чтобы они не влияли на работу остального кода. ***Это один из главных принципов паттернов. Все паттерны обеспечивают возможность изменения некоторой части системы независимо от других частей.***

Чтобы отделить эти аспекты поведения от класса Duck, мы выносим оба метода за пределы класса Duck и создаём новый набор классов для представления каждого аспекта.



Второй принцип проектирования: **Программируйте на уровне интерфейса, а не на уровне реализации.**

Поэтому мы создаём суперклассы FlyBehavior и QuackBehavior , а каждая реализация аспекта поведения будет представлена конкретной реализацией этого интерфейса.



Поведение не наследуется, а предоставляется правильно выбранным объектом. Другими словами, вместо наследования используется композиция.

Таким образом, системы, созданные на основе композиции, обладают значительно большей гибкостью. Они позволяют не только инкапсулировать поведение, но и изменять его во время выполнения.

**Паттерн Стратегия** определяет семейство алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. Он позволяет модифицировать алгоритм независимо от их использования на стороне клиента.

***Выводы:***

***Инкапсулируйте, то, что изменяется.***

***Отдавайте предпочтение композиции перед наследованием.***

***Программируйте на уровне интерфейсов, а не реализации.***

Лабораторная работа №1

Паттерн стратегия

Домашнее задание: Изучить, теоретический материал. Написать в отчёте определение паттерна стратегия.

Лабораторное задание

1. Набрать программу.

// Паттерн Стратегия

#include "stdafx.h"

#include "iostream"

using namespace std;

class FlyBehavior {

public:

virtual void fly() = 0;

};

class FlyWithWings : public FlyBehavior {

public:

virtual void fly() { cout << "\n I fly "; }

};

class FlyNoWay : public FlyBehavior {

public:

virtual void fly() { cout << "\n I no fly "; }

};

class QuackBehavior {

public:

virtual void quack() = 0;

};

class Quack : public QuackBehavior {

public:

virtual void quack() { cout << "\n I Quack "; };

};

class Squack : public QuackBehavior {

public:

virtual void quack() { cout << "\n I pishu "; };

};

class MuteQuack : public QuackBehavior {

public:

virtual void quack() { cout << "\n I no Quack "; };

};

class Duck {

private:

FlyBehavior \* pFly;

QuackBehavior \* pQuack;

public:

Duck() :pFly(NULL), pQuack(NULL) {}

Duck(FlyBehavior \* pF, QuackBehavior \* pQ) : pFly(pF), pQuack(pQ) {}

virtual ~Duck() { delete pFly; delete pQuack; }

void swim() { cout << "\n I swim "; }

virtual void display() = 0;

void PerformFly() { if(pFly!=NULL)pFly->fly(); }

void PerformQuack() { if (pQuack != NULL) pQuack->quack(); }

void SetFlyBehavior(FlyBehavior \* p) {

delete pFly;

pFly = p;

}

void SetQuackBehavior(QuackBehavior \* p) {

delete pQuack;

pQuack = p;

}

};

class RubberDuck : public Duck {

public:

RubberDuck() :Duck() {}

RubberDuck(FlyBehavior \* pF, QuackBehavior \* pQ) : Duck(pF, pQ) {}

void display() { cout << "\n I am RubberDuck"; }

};

class DecoyDuck : public Duck {

public:

DecoyDuck() :Duck() {}

DecoyDuck(FlyBehavior \* pF, QuackBehavior \* pQ) : Duck(pF, pQ) {}

void display() { cout << "\nI am DecoyDuck"; }

};

void main() {

RubberDuck Rd;

Rd.display();

Rd.PerformFly();

getchar();

Rd.SetFlyBehavior(new FlyWithWings());

Rd.PerformFly();

getchar();

Rd.SetFlyBehavior(new FlyNoWay());

Rd.PerformFly();

getchar();

DecoyDuck Dd(new FlyWithWings(), new Squack());

Dd.display();

Dd.PerformFly();

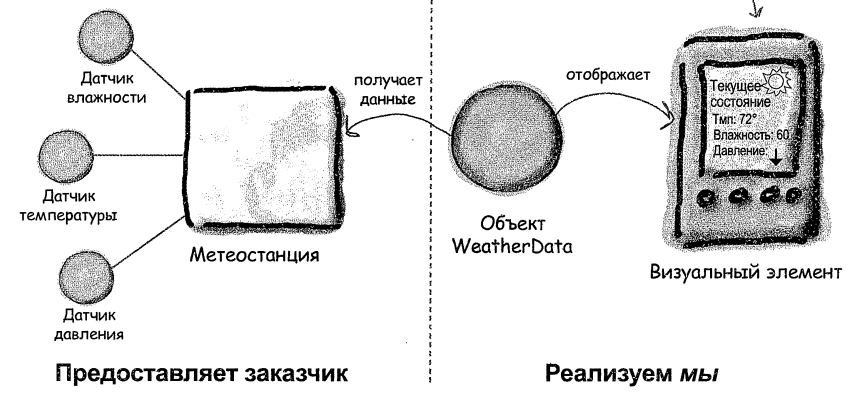
Dd.PerformQuack();

}

1. Доопределить поведение «быстрый полёт» для объектов класса «дикая утка».
2. Доопределить поведение «громкое кряканье» для объектов класса «домашняя утка».
3. Продемонстрировать, что поведение «быстрый полёт» и «громкое кряканье» могут демонстрировать объекты и других классов, производных от Duck.
4. Написать программу, которая рисует на экране воинов различных типов (пехотинец, моряк, десантник), которые могут использовать как типичное для них оружие (винтовка, кортик, автомат, сапёрная лопата, граната и т.д.), так и нет. Смена оружия у воина, должна приводить к изменению его облика на экране.

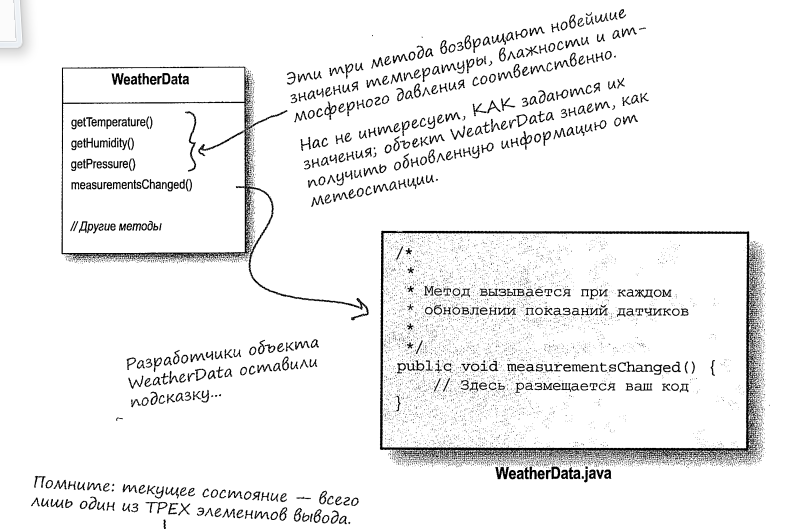
**Паттерн Наблюдатель**

Задача:



Объект WeatherData умеет получать от физической метеостанции обновленные данные. Затем объект WeatherData обновляет изображение для трех основных элементов: текущего состояния (температура, влажность и давление), статистики и прогноза.

Наша задача — создать приложение, которое использует данные объекта WeatherData для обновления текущих условий, статистики и прогноза погоды.



Метод measurementsChanged() необходимо реализовать так, чтобы он обновлял изображение для трех элементов: текущего состояния, статистики и прогноза.

* Класс WeatherData содержит get-методы для показаний трех датчиков: температуры, влажности и атмосферного давления.
* Метод measurementsChanged() вызывается при появлении новых метеорологических данных. (Мы не знаем, как он вызывается, да это и неважно; достаточно знать, что он вызывается.)
* Необходимо реализовать три экрана вывода, использующих метеорологические данные: экран текущего состояния, экран статистики и экран прогноза. Эти экраны должны обновляться каждый раз, когда у объекта WeatherData появляются новые данные.
* Система должна быть расширяемой — другие разработчики будут создавать новые экраны вывода, а пользователи могут добавлять и удалять их в своих приложениях. В настоящее время определены всего три вида экранов (текущее состояние, статистика и прогноз).

Первое, что приходит 6 голову

Первый вариант реализации, который приходит в голову: мы следуем совету разработчиков WeatherData и включаем свой код в метод measuremenlsChanged():

public class WeatherData {

// Объявления переменных экземпляров

public void measurementsChanged() {

float temp = getTemperature(); // Чтобы получить обновленные данные

float humidity = getHumidity () ; //мы вызываем (уже реализованные)

float pressure = getPressure (); //get -методы класса WeatherData

//Обновляем показания. Запрос на перерисовку каждого элемента

//с передачей обновлённых значений

currentConditionsDisplay.update(temp, humidity, pressure);

statisticsDisplay.update(temp, humidity, pressure);

forecastDisplay.update(temp, humidity, pressure);

// Другие методы WeatherData

}

}

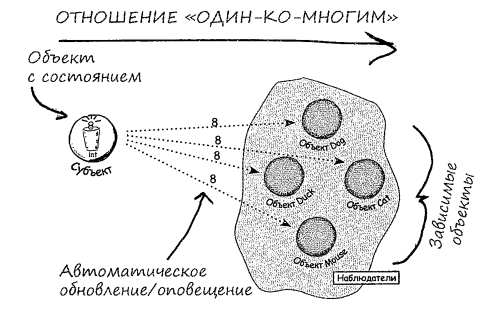
Недостаток. Программа на уровне конкретной реализации, мы не сможем добавлять и удалять визуальные элементы без внесения изменения в программу.

Принцип организации паттерна Наблюдатель

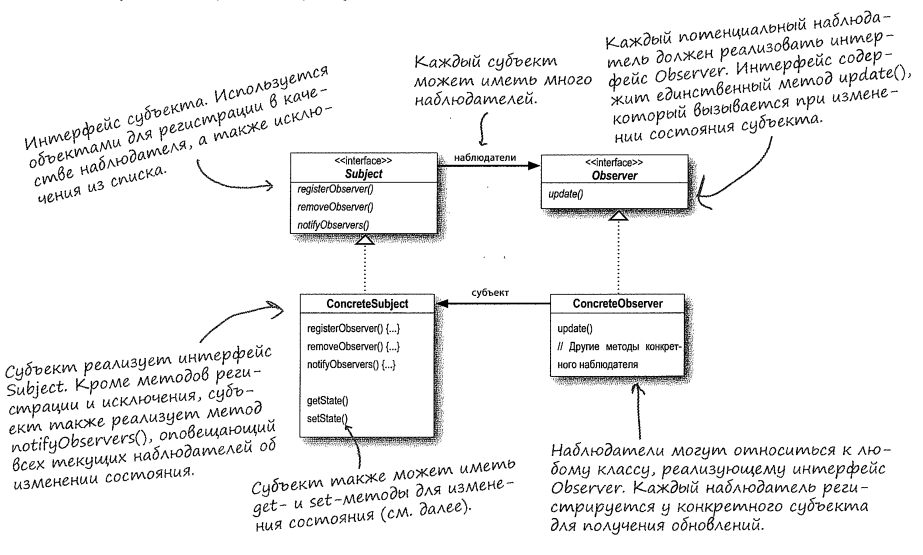
Издатель + Подписчик = Паттерн наблюдатель.

Т.е. можно подписаться на получение уведомлений и можно отказаться от них.

**Паттерн Наблюдатель определяет отношение типа «один-ко-многим» между объектами. Когда состояние одного объекта изменяется, все зависимые объекты получают оповещения.**



Паттерн Наблюдатель: диаграмма классов.



**Сила слабых связей**

***Если два объекта могут взаимодействовать, не обладая практически никакой информацией друг о друге, такие объекты называют слабосвязанными.***

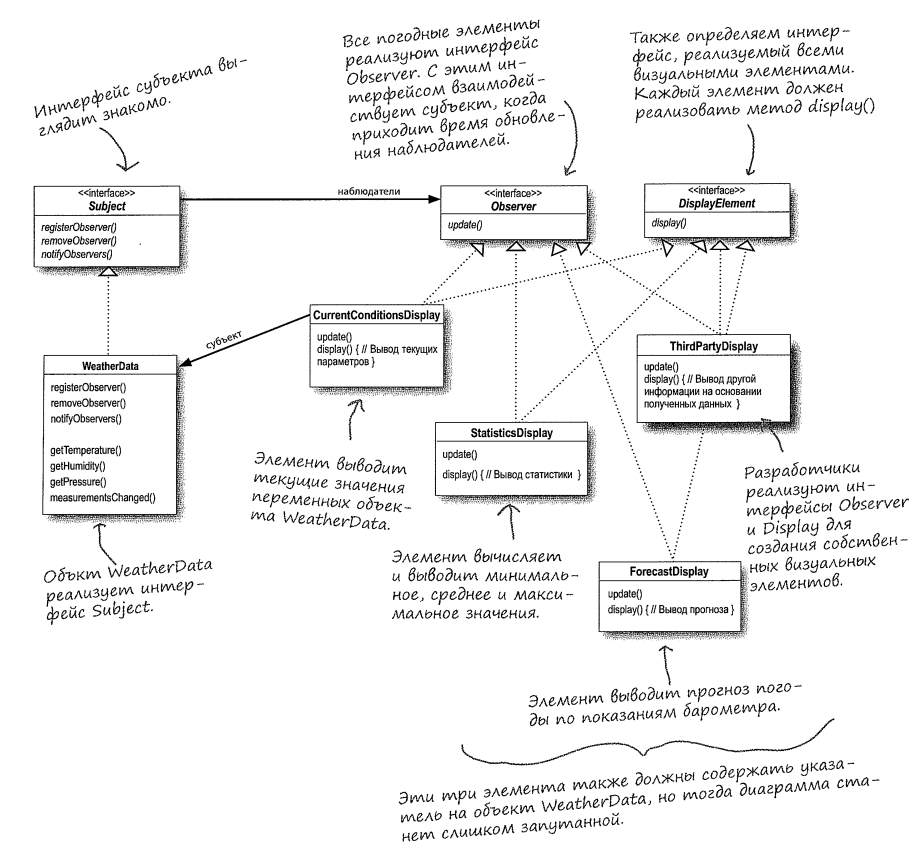
В архитектуре паттерна Наблюдатель между субъектами и наблюдателями существует слабая связь.

Почему?

* Единственное, что знает субъект о наблюдателе, — то, что тот реализует некоторый интерфейс (Observer). Ему не нужно знать ни конкретный класс наблюдателя, ни его функциональность... вообще ничего.
* Новые наблюдатели могут добавляться в любой момент. Так как субъект зависит только от списка объектов, реализующих интерфейс Observer, вы можете добавлять новых наблюдателей по своему усмотрению. Любого наблюдателя во время выполнения можно заменить другим наблюдателем или исключить его из списка — субъект этого не заметит.
* Добавление новых типов наблюдателей не требует модификации субъекта. Допустим, у нас появился новый класс, который должен стать наблюдателем. Вносить изменения в субъект не потребуется — достаточно реализовать интерфейс Observer в новом классе и зарегистрировать его в качестве наблюдателя. Субъект будет доставлять оповещения любому объекту, реализующему интерфейс Observer.
* Субъекты и наблюдатели могут повторно использоваться независимо друг от друга. Между ними не существует сильных связей, что позволяет повторно использовать их для других целей.
* Изменения в субъекте или наблюдателе не влияют на другую сторону.
* Благодаря слабым связям мы можем вносить любые изменения на любой из двух сторон — при условии, что объект реализует необходимый интерфейс субъекта или наблюдателя.

**На базе слабосвязанных архитектур строятся гибкие ОО-системы, которые хорошо адаптируются к изменениям благодаря минимальным зависимостям между объектами.**

Решение нашей задачи.



Ключевые моменты

1. Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами.
2. Субъекты обновляют наблюдателей через единый интерфейс.
3. Субъект ничего не знает о наблюдателях — кроме того, что они реализуют интерфейс Observer.
4. При использовании паттерна возможен как запрос, так и активная доставка данных от субъекта (запрос считается более «правильным»).
5. Работа кода не должна зависеть от порядка оповещения наблюдателей.

Принципы

1. Инкапсулируйте то, что изменяется.
2. Предпочитайте композицию наследованию
3. Программируйте на уровне интерфейсов.
4. Стремитесь к слабой связанности взаимодействующих объектов.

#include "stdafx.h"

#include <iostream>

#include <list>

#include <algorithm>

using namespace std;

/\* Класс, обеспечивающий семантику подсчёта ссылок.

\* Объект, на который ссылается указатель, автоматически

\* уничтожается при удалении последнего экземпляра CountedPtr

\* для данного объекта.

\*/

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class WeatherData;

class Observer;

class Subject {

public:

virtual void registerObserver(Observer\* ob) = 0;

virtual void removeObserver(Observer\* ob) = 0;

virtual void notifyObserver() = 0;

};

class Observer {

public:

virtual void update(float temperature, float humidity, float pressure) = 0;

};

class DisplayElement {

public:

virtual void display() = 0;

};

class CurrentConditionsDisplay : public Observer, DisplayElement {

private:

float temperature;

float humidity;

Subject\* weatherData;

public:

CurrentConditionsDisplay(Subject\* wD) : weatherData(wD) { weatherData->registerObserver(this); }

void update(float t, float h, float p) { temperature = t; humidity = h; display(); }

void display() { cout << "\n CurrentConditionsDisplay temperature = "<< temperature<<"\t humidity = "<< humidity; }

void removeObserver() { weatherData->removeObserver(this); }

~CurrentConditionsDisplay() { weatherData->removeObserver(this); }

};

class StatisticDisplay : public Observer, DisplayElement {

private:

float temperature;

float humidity;

Subject\* weatherData;

public:

StatisticDisplay(Subject\* wD) : weatherData(wD) { weatherData->registerObserver(this); }

void update(float t, float h, float p) { temperature = t; humidity = h; display(); }

void display() { cout << "\n StatisticDisplay temperature = " << temperature << "\t humidity = " << humidity; }

void removeObserver() { weatherData->removeObserver(this); }

~StatisticDisplay() { weatherData->removeObserver(this); }

};

typedef CountedPtr<Observer> ObserverPtr;

class WeatherData: public Subject {

private:

list<ObserverPtr> list\_Obs;

float temperature;

float humidity;

float pressure;

public:

void registerObserver(Observer\* ob) {list\_Obs.push\_front(ObserverPtr(ob)); }

void removeObserver(Observer\* ob) {

list<ObserverPtr>::iterator pos;

for (pos = list\_Obs.begin(); pos != list\_Obs.end(); ++pos)

if (&(\*(\*pos)) == ob) {

list\_Obs.erase(pos);

break;

}

}

void notifyObserver() {

list<ObserverPtr>::const\_iterator pos;

for (pos = list\_Obs.begin(); pos != list\_Obs.end(); ++pos) (\*pos)->update(temperature, humidity, pressure);

}

void measurementsChange() { notifyObserver();}

void setMeasurements(float t, float h, float p) {temperature = t; humidity = h; pressure = p; measurementsChange();}

~WeatherData() {list\_Obs.clear();}

};

int main()

{

WeatherData \* pWD = new WeatherData;

CurrentConditionsDisplay\* ccD1 = new CurrentConditionsDisplay(pWD);

CurrentConditionsDisplay\* ccD2 = new CurrentConditionsDisplay(pWD);

StatisticDisplay \* sD1 = new StatisticDisplay(pWD);

pWD->setMeasurements(10, 20, 30);

ccD2->removeObserver();

pWD->setMeasurements(40, 50, 60);

getchar();

delete pWD;

return 0;

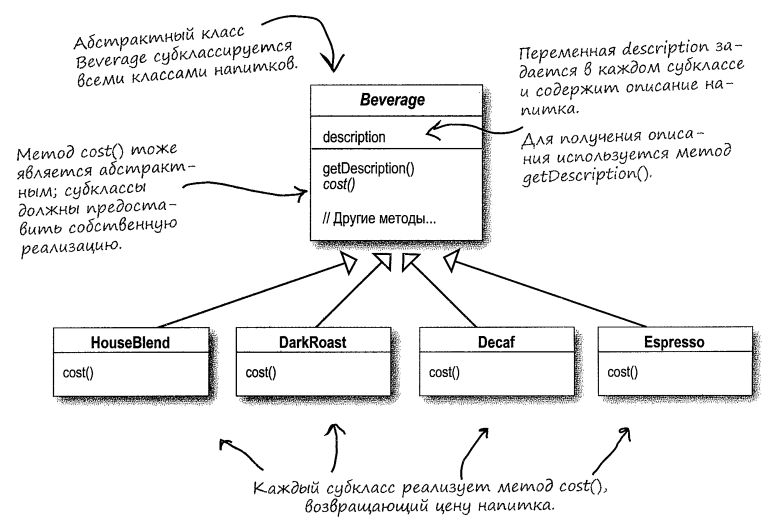
}

1. Набрать и отладить программу.
2. Предположим, что метеостанция стала измерять и солнечную активность. Допишите наблюдателя, который будет получать и выводить на экран эту информацию.
3. Модифицировать программу так, чтобы метеостанция присылала уведомление о любых новых данных, а наблюдатели после этого запрашивали требуемые им данные с метеостанции (модель запроса данных).
4. Модифицируйте программу так, чтобы наблюдатель мог регистрироваться на нескольких метеостанциях.

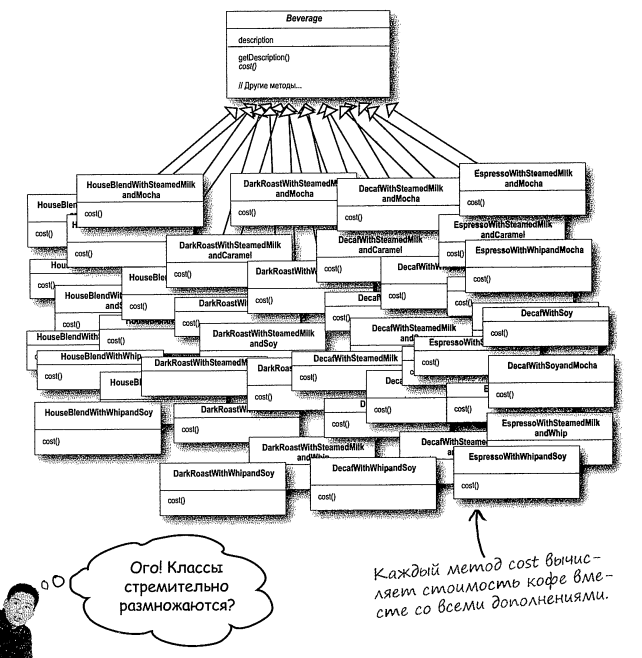
**Паттерн Декоратор**

Эту главу можно назвать «Взгляд на архитектуру для любителей наследования». Мы проанализируем типичные злоупотребления из области наследования, и вы научитесь декорировать свои классы во время выполнения с использованием разновидности композиции. Зачем? Затем, что этот прием позволяет вам наделить свои (или чужие) объекты новыми возможностями *без модификации кода классов*.

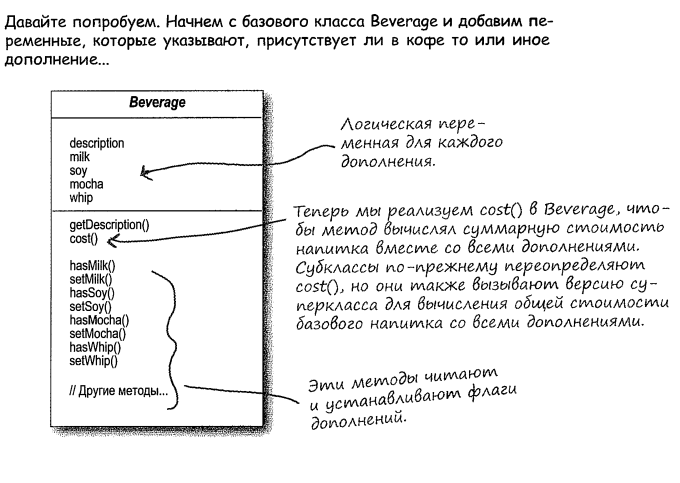
Задача. Сеть кофеин

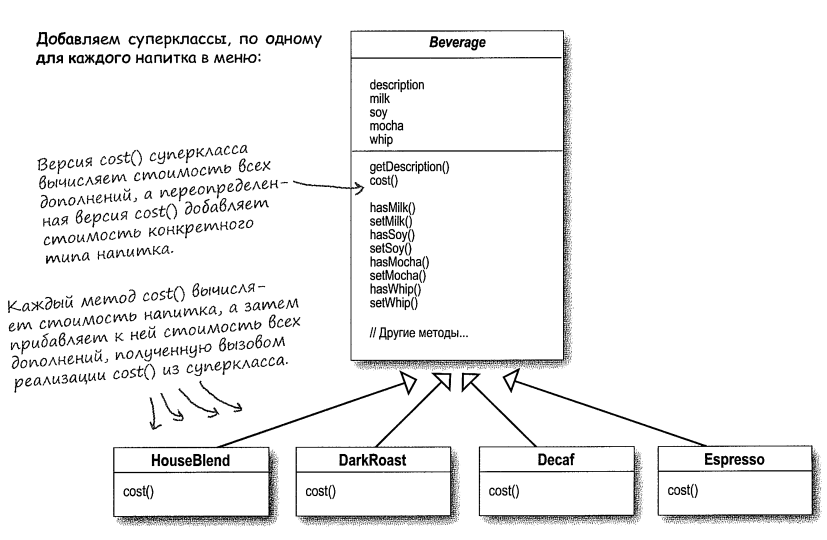


К кофе продаются различные добавки (пенка, шоколад и т.д.). Дополнения не бесплатны, поэтому они должны быть встроены в систему оформления заказов.



Другой подход



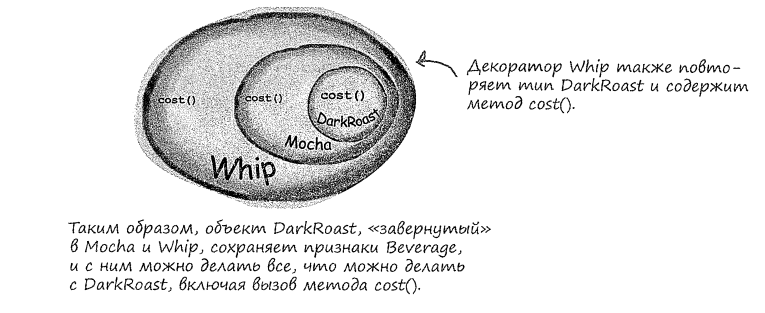


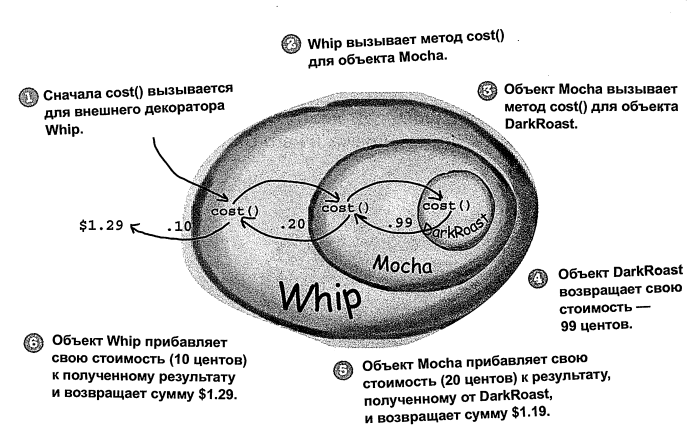
Берем объект DarkRoast;

Декорируем его объектом Mocha;

Декорируем его объектом Whip;

Вызываем метод cost() и пользуемся делегированием для прибавления стоимости дополнений.



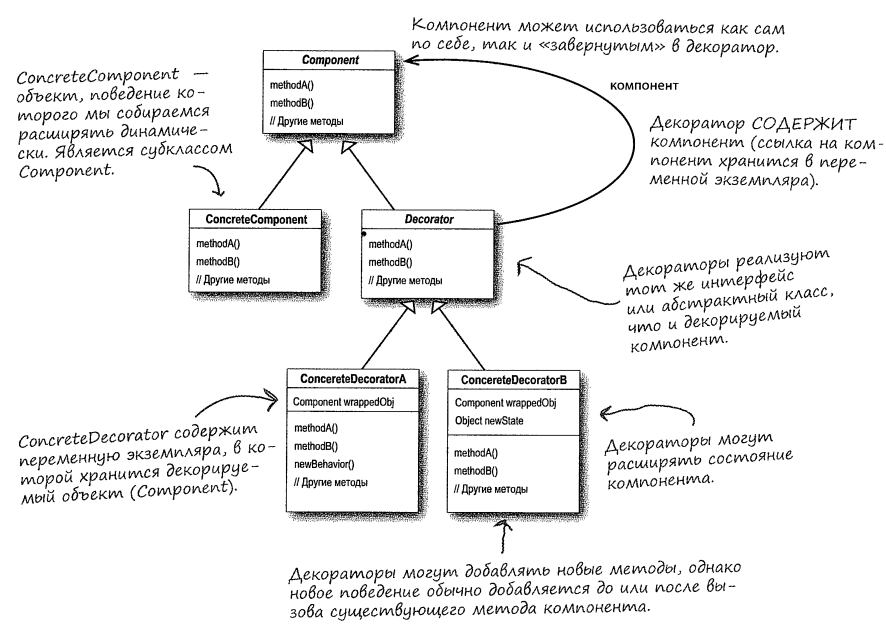


Что известно

1. Декораторы имеют тот же супер тип, что и декорируемые объекты.
2. Объект можно «завернуть» в один или несколько декораторов.
3. Так как декоратор относится к тому же супер типу, что и декорируемый объект, мы можем передать декорированный объект вместо исходного.
4. Декоратор добавляет свое поведения до и (или) после делегирования операций декорируемому объекту, выполняющему остальную работу.
5. Объект может быть декорирован в любой момент времени, так что мы можем декорировать объекты динамически и с произвольным количеством декораторов.

Паттерн Декоратор динамически наделяет объект новыми возможностями и является гибкой альтернативой наследованию в области расширения функциональности.

Диаграмма классов



Выводы

* Наследование — одна из форм расширения, но оно не всегда обеспечивает гибкость архитектуры.
* Следует предусмотреть возможность расширения поведения без изменения существующего кода.
* Композиция и делегирование часто используются для динамического добавления нового поведения.
* Паттерн Декоратор предоставляет альтернативу субклассированию в области расширения поведения.
* Типы декораторов соответствуют типам декорируемых компонентов (соответствие достигается посредством наследования или реализации интерфейса).
* Декораторы изменяют поведение компонентов, добавляя новую функциональность до и (или) после (или даже вместо) вызовов методов компонентов.
* Компонент может декорироваться любым количеством декораторов.
* Декораторы обычно прозрачны для клиентов компонента (если клиентский код не зависит от конкретного типа компонента

#include "stdafx.h"

#include <iostream>

#include <string>

using namespace std;

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class Beverage {

protected:

string description;

public:

virtual string GetDiscription() { return description; }

virtual double cost() = 0;

};

typedef CountedPtr<Beverage> pBeverage;

class CondimentDecorator : public **Beverage** {

protected:

**pBeverage beverage;**

CondimentDecorator(pBeverage pBg) : beverage(pBg) {}

};

**Мэри:** Посмотри на диаграмму классов. CondimentDecorator расширяет класс Beverage. Это наследование, верно?

**Сью:** Верно. Здесь принципиально то, что декораторы должны относиться к тому же супертипу, что и декорируемые объекты. Таким образом, наследование применяется для согласования типов, а не для обеспечения поведения.

**Мэри:** Хорошо, я понимаю, что декораторы должны обладать таким же «интерфейсом», что и компоненты, потому что они должны использоваться вместо компонентов. Но откуда тогда берется поведение?

**Сью:** Объединяя декоратор с компонентом, мы добавляем новое поведение. Это поведение не наследуется от суперкласса, а добавляется посредством композиции.

**Мэри:** Выходит, мы субклассируем абстракный класс Beverage только для приведения к нужному типу, а не для наследования его поведения. Поведение формируется в результате композиции декораторов с базовыми компонентами и другими декораторами.

**Сью:** Точно.

**Мэри:** Начинаю понимать. Композиция объектов дает нам значительную гибкость в смешении напитков и дополнений. Очень элегантно.

**Сью:** Да. Если бы мы воспользовались наследованием, то все поведение определялось бы статически во время компиляции. Другими словами, мы могли бы использовать только то поведение, которое нам предоставляет суперкласс — или которое мы переопределяем в субклассе. Композиция делает возможным произвольное смешивание декораторов... во время выполнения.

**Мэри:** И, насколько я поняла, мы можем в любой момент реализовать новые декораторы для добавления нового поведения.

**Сью:** Вот именно.

**Мэри:** Остался последний вопрос. Если наследуется только тип компонента, то почему не воспользоваться интерфейсом вместо абстрактного класса Beverage?

**Сью:** Потому что когда нам передали этот код, у Starbuzz уже был абстрактный класс Beverage. Изменения в существующем коде всегда нежелательны; не стоит «исправлять» код, если можно просто воспользоваться абстрактным классом.

class Espresso : public Beverage {

public:

Espresso() {description = "Espresso";}

double cost() { return 1.99; }

};

class HouseBlend : public Beverage {

public:

HouseBlend() { description = "HouseBlend"; }

double cost() { return 0.89; }

};

class Mocha : public CondimentDecorator {

public:

Mocha(pBeverage bg) : CondimentDecorator(bg) {}

public:

string GetDiscription() {return (beverage->GetDiscription() + string(", Mocha"));}

double cost() { return 0.20 + beverage->cost(); }

};

class Soy : public CondimentDecorator {

public:

Soy(pBeverage bg) : CondimentDecorator(bg) {}

public:

string GetDiscription() { return (beverage->GetDiscription() + string(", Soy")); }

double cost() { return 0.16 + beverage->cost(); }

};

int main()

{

pBeverage pBg = pBeverage(new Espresso());

cout << pBg->GetDiscription() + " $ "<<pBg->cost();

cout << "\n";

pBeverage pB2 = pBeverage(new HouseBlend());

pB2 = pBeverage(new Mocha(pB2));

pB2 = pBeverage(new Mocha(pB2));

pB2 = pBeverage(new Soy(pB2));

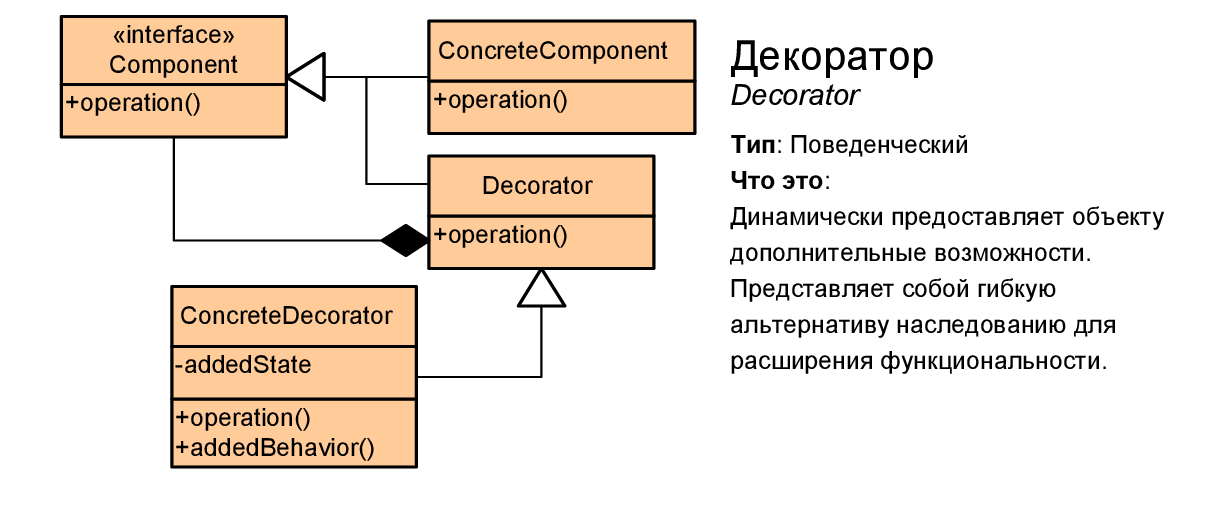
cout << pB2->GetDiscription()+ " $ " << pB2->cost();

getchar();

return 0;

}

1. Набрать и отладить программу.
2. Создать ещё два вида кофе и приправ.
3. Модифицировать программу так, чтобы можно было выводить меню (стоимость напитков и добавок к ним) на экран.
4. Кофе стали разливать в маленькую, среднюю и большую чашки. Для этого в класс Beverage добавили две функции getSize() и SetSize(). От размера стоимость кофе и добавок к ним изменилась. Модифицируйте программу так, чтобы учесть эти изменения.
5. Клиенты стали заказывать кофе: горячее, тёплое, холодное.



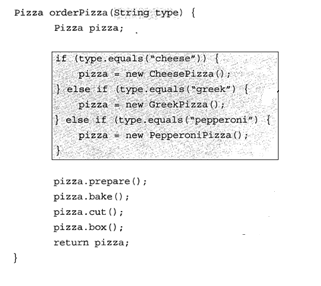
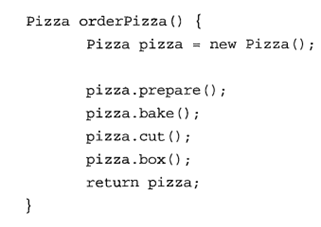
ПРОСТАЯ ФАБРИКА

Создание объектов отнюдь не сводится к простому вызову оператора new. Оказывается, создание экземпляров не всегда должно осуществляться открыто; оно часто создает проблемы сильного связывания. Паттерн Фабрика спасет вас от неприятных зависимостей.

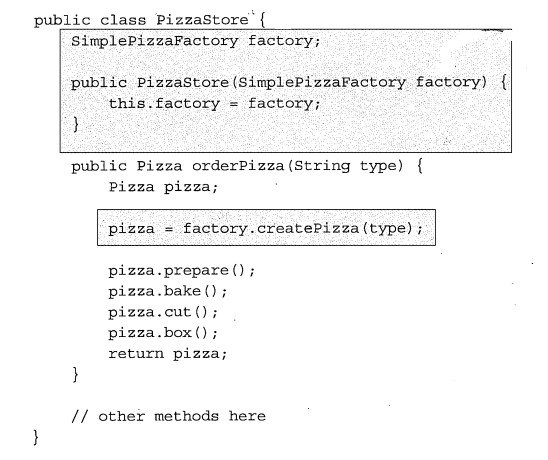
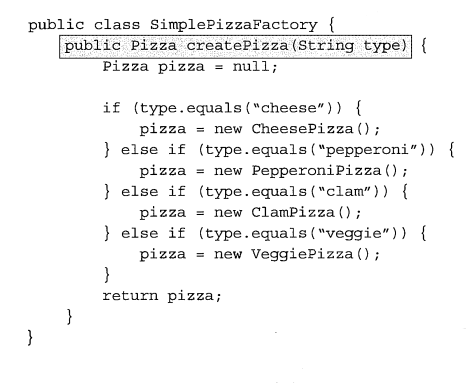
Видим new— подразумеваем конкретный.

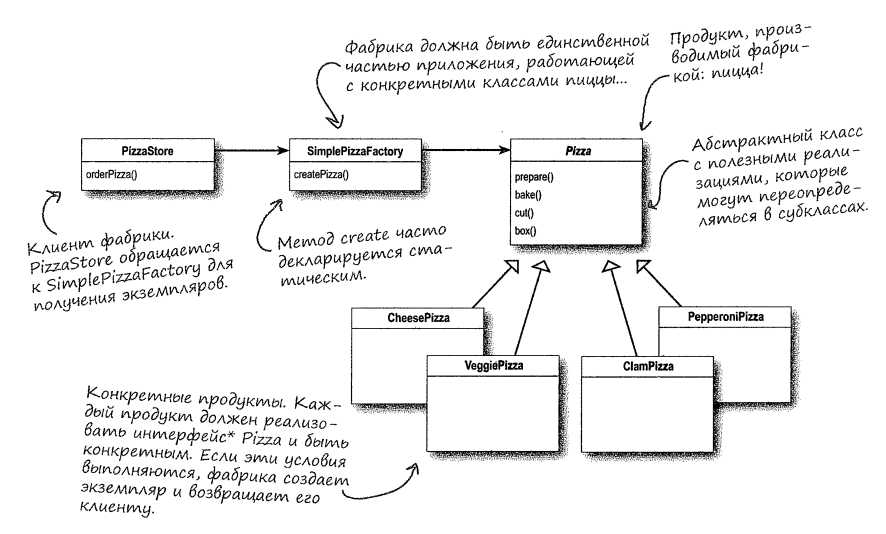
При использовании new вы создаете экземпляр конкретного класса, поэтому эта операция относится к уровню реализации, а не интерфейса. А вы уже знаете, что привязка кода к конкретному классу делает его менее гибким и устойчивым к изменениям.

Пример: Мы видим, что одна часть кода остаётся неизменной, а другая меняется.



Следовательно, можно предположить, что её можно и нужно инкапсулировать.





#include "stdafx.h"

#include <iostream>

#include <string>

using namespace std;

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class Pizza {

public:

virtual void prepare() { cout<<"\n prepare pizza"; }

virtual void bake() { cout<<"\n bake pizza"; }

virtual void cut() { cout << "\n cut pizza"; }

virtual void box() { cout << "\n box pizza"; }

};

class CheesePizza : public Pizza {

virtual void prepare() { cout << "\n prepare cheesePizza"; }

virtual void bake() { cout << "\n bake cheesePizza"; }

virtual void cut() { cout << "\n cut cheesePizza"; }

virtual void box() { cout << "\n box cheesePizza"; }

};

class PepperoniPizza : public Pizza {

virtual void prepare() { cout << "\n prepare PepperoniPizza"; }

virtual void bake() { cout << "\n bake PepperoniPizza"; }

virtual void cut() { cout << "\n cut PepperoniPizza"; }

virtual void box() { cout << "\n box PepperoniPizza"; }

};

typedef CountedPtr<Pizza> pPizza;

class SimplePizzaFactory {

pPizza ppizza;

public:

pPizza createPizza(string type) {

if (type == "cheese") ppizza = pPizza(new CheesePizza());

else if (type == "pepperoni") ppizza = pPizza(new PepperoniPizza());

else ppizza = pPizza(new Pizza());

return ppizza;

}

};

class PizzaStore { //пиццерия

SimplePizzaFactory factory;

public:

PizzaStore(SimplePizzaFactory fct) :factory(fct) {}

pPizza orderPizza(string type) { //заказ пиццы

pPizza ppizza = factory.createPizza(type);

ppizza->prepare();

ppizza->bake();

ppizza->cut();

ppizza->box();

return ppizza;

}

};

int main()

{

SimplePizzaFactory spf;

PizzaStore ps(spf);

ps.orderPizza("");

ps.orderPizza("cheese");

ps.orderPizza("pepperoni");

getchar();

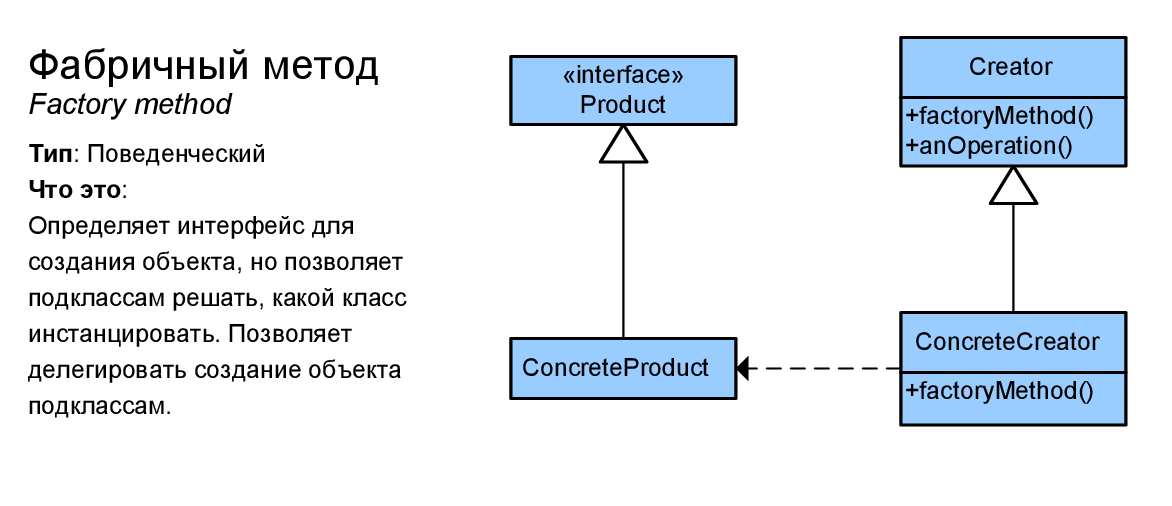
return 0;

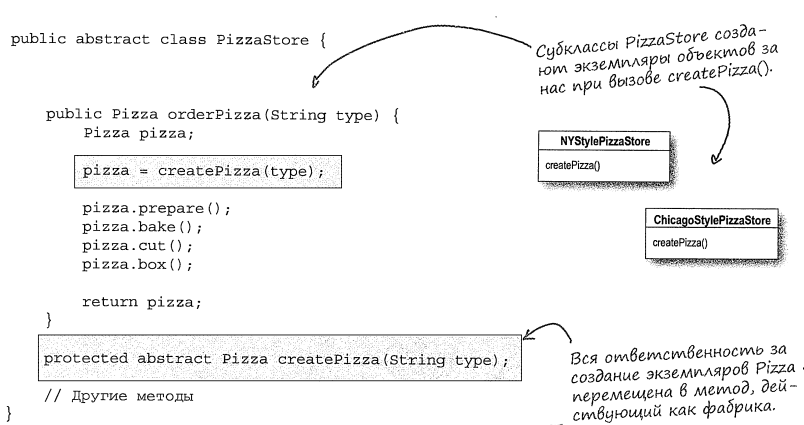
}

1. Набрать и отладить программу.
2. Создать ещё два вида пиццы.
3. Модифицировать программу так, чтобы можно было менять тип фабрики в классе PizzaStore, т.е. сделать SimplePizzaFactory абстрактным классом, от которого будут создаваться потомки, которые будут учитывать региональные особенности приготовления пицц cheese и pepperoni.

**Паттерн Фабричный метод**

Паттерн Фабричный Метод определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом, Фабричный Метод делегирует операцию создания экземпляра субклассам.





#include "stdafx.h"

#include <iostream>

#include <string>

#include <vector>

using namespace std;

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class Pizza {

protected:

string name;

string dough;

string sauce;

vector<string> ls;

public:

virtual void prepare() {

cout << "\n Preparing" + name;

for (int i = 0; i < ls.size(); i++) cout << " " + ls[i];

}

virtual void bake() { cout << "\n Bake for 25 minutes at 350"; }

virtual void cut() { cout << "\n Cuting the pizza into diagonal slices"; }

virtual void box() { cout << "\n Place pizza in official PizzaStore box"; }

};

typedef CountedPtr<Pizza> pPizza;

class NYStyleCheesePizza : public Pizza {

public:

NYStyleCheesePizza() {

name = "NY Style Sauce and cheese pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

ls.push\_back(string("Grated Reggiano Chese"));

}

virtual void prepare() { cout << "\n prepare NYStyleCheesePizza"; }

virtual void bake() { cout << "\n bake cheesePizza"; }

virtual void cut() { cout << "\n cut cheesePizza"; }

virtual void box() { cout << "\n box cheesePizza"; }

};

class NYStylePepperoniPizza : public Pizza {

virtual void prepare() { cout << "\n prepare NYStylePepperoniPizza"; }

virtual void bake() { cout << "\n bake PepperoniPizza"; }

virtual void cut() { cout << "\n cut PepperoniPizza"; }

virtual void box() { cout << "\n box PepperoniPizza"; }

};

class MsStyleCheesePizza : public Pizza {

virtual void prepare() { cout << "\n prepare MsStyleCheesePizza"; }

virtual void bake() { cout << "\n bake cheesePizza"; }

virtual void cut() { cout << "\n cut cheesePizza"; }

virtual void box() { cout << "\n box cheesePizza"; }

};

class PizzaStore { //пиццерия

public:

pPizza orderPizza(string type) { //заказ пиццы

pPizza pizza = createPizza(type);

pizza->prepare();

pizza->bake();

pizza->cut();

pizza->box();

return pizza;

}

protected:

virtual pPizza createPizza(string type)=0;

};

class NYPizzaStore: public PizzaStore {

public:

pPizza createPizza(string type) {

pPizza ppizza;

if (type == "cheese") ppizza = pPizza(new NYStyleCheesePizza());

else if (type == "pepperoni") ppizza = pPizza(new NYStylePepperoniPizza());

else ppizza = pPizza(new Pizza());

return ppizza;

}

};

int main()

{

NYPizzaStore NYps;

NYps.orderPizza("");

NYps.orderPizza("cheese");

NYps.orderPizza("pepperoni");

getchar();

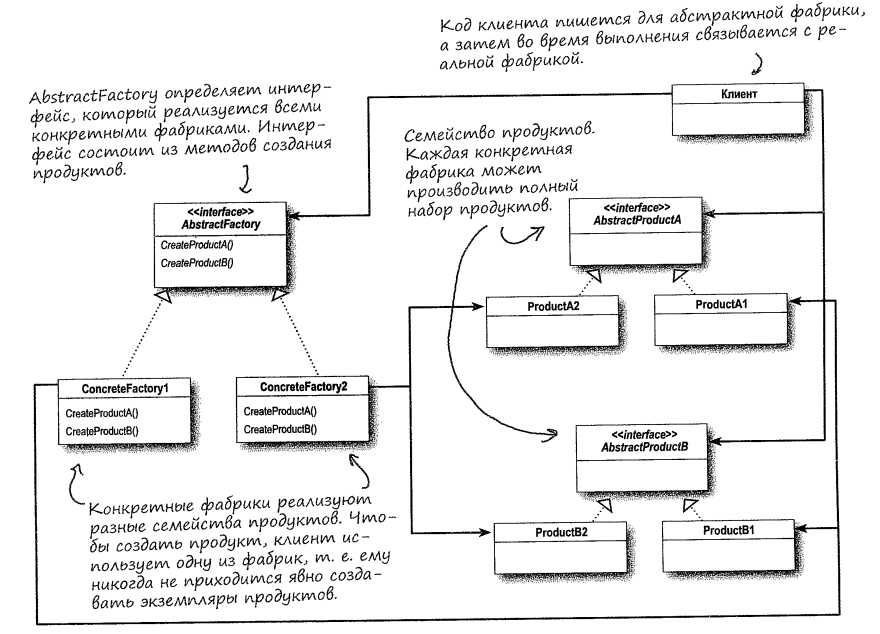
return 0;

}

1. Набрать и отладить программу.
2. Дописать программу так, чтобы было реализовано три вида пиццерий, в которых готовят три вида пицц.
3. Распечатать меню пиццерий.
4. Упросить программу, избавившись от класса CountedPtr.

ПАТТЕРН Абстрактная фабрика

Паттерн Абстрактная Фабрика предоставляет интерфейс создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.



В нашем примере класс Pizza является клиентом, которая содержит указатели на интерфейс фабрика и продукты.

Методы Абстрактной Фабрики часто реализуются как фабричные методы. И это вполне логично. Задача Абстрактной Фабрики - определить интерфейс для создания набора продуктов. Каждый метод этого интерфейса отвечает за создание конкретного продукта, и мы реализуем субкласс Абстрактной Фабрики, который предоставляет эти реализации. Таким образом, фабричные методы являются естественным способом реализации методов продуктов в абстрактных фабриках.

* Все фабрики инкапсулируют создание объектов.
* Простая Фабрика, хотя и не является полноценным паттерном, обеспечивает простой механизм изоляции клиентов от конкретных классов.
* Фабричный Метод основан на наследовании: создание объектов делегируется субклассам, реализующим фабричный метод для создания объектов.
* Абстрактная Фабрика основана на композиции: создание объектов реализуется в методе, доступ к которому осуществляется через интерфейс фабрики.
* Все фабричные паттерны обеспечивают слабую связанность за счет сокращения зависимости приложения от конкретных классов.
* Задача Фабричного Метода — перемещение создания экземпляров в субклассы.
* Задача Абстрактной Фабрики — создание семейств взаимосвязанных объектов без зависимости от их конкретных классов.

Задание: набрать программу приведённую ниже.

#include "stdafx.h"

#include <iostream>

#include <string>

using namespace std;

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class Dough { protected: string name; double cost; }; //Тесто

class ThinDough : public Dough { public: ThinDough() { name = "thin dough"; } };

class FatDough: public Dough { public: FatDough() { name = "fat dough"; }};

class Sauce { protected: string name; double cost; };//Соус

class TomatoSauce: public Sauce { public: TomatoSauce() { name = "tomato sauce"; }};

class KetchupSauce: public Sauce { public: KetchupSauce() { name = "ketchup sauce";}};

class Cheese { protected: string name; double cost; };//Сыр

class HardCheese: public Cheese { public: HardCheese() { name = "hard cheese";}};

class SoftCheese: public Cheese { public: SoftCheese() { name = "soft cheese";}};

class Clams { protected: string name; double cost; };//Мидии

class FreshClams: public Clams { public: FreshClams() { name = "fresh clams";}};

class FrozenClams: public Clams { public: FrozenClams() { name = "frozen clams";}};

//---------------------------------------------------------------------------------------------

typedef CountedPtr<Dough> pDough;

typedef CountedPtr<Sauce> pSauce;

typedef CountedPtr<Cheese> pCheese;

typedef CountedPtr<Clams> pClams;

//фабрика ингредиентов

class PizzaIngredientFactory {

public:

virtual pDough createDough() = 0;

virtual pSauce createSauce() = 0;

virtual pCheese createCheese() = 0;

virtual pClams createClams() = 0;

};

//фабрика ингредиентов в Нью-Йорке

class NYPizzaIngredientFactory : public PizzaIngredientFactory {

pDough createDough() { return pDough(new ThinDough()); }

pSauce createSauce() { return pSauce(new TomatoSauce()); }

pCheese createCheese() { return pCheese(new HardCheese()); }

pClams createClams() { return pClams(new FreshClams()); }

};

//фабрика ингредиентов в Чикаго

class CHPizzaIngredientFactory : public PizzaIngredientFactory {

pDough createDough() { return pDough(new FatDough()); }

pSauce createSauce() { return pSauce(new KetchupSauce()); }

pCheese createCheese() { return pCheese(new SoftCheese()); }

pClams createClams() { return pClams(new FrozenClams()); }

};

typedef CountedPtr<PizzaIngredientFactory> pPizzaIngredientFactory;

//-------------------------------------------------------------------------------------

class Pizza {//пицца

protected:

pPizzaIngredientFactory pingredientFactory; //Интерфейс "Абстрактная фабрика"

string name;

//Интерфейсы "Абстрактные продукты"

pDough pdough; //Тесто

pSauce psauce; //Соус

pCheese pcheese; //Сыр

pClams pclams; //Мидии

public:

virtual void prepare(){ //Из каких продуктов готовиться пицца

cout << string("\n\n Preparing ") + string("default pizza");

}

virtual void bake() { cout << "\n Bake for 25 minutes at 350"; }

virtual void cut() { cout << "\n Cutting the pizza into diagonal slices"; }

virtual void box() { cout << "\n Place pizza in official PizzaStore box"; }

void SetName(string nm) { name = nm; }

string GetName() { return name; }

};

typedef CountedPtr<Pizza> pPizza;

class CheesePizza : public Pizza {//пицца с сыром

public:

CheesePizza(pPizzaIngredientFactory pF) { pingredientFactory = pF; name = "CheesePizza"; }

void prepare() {

cout << "\n\n Preparing " + name;

pdough = pingredientFactory->createDough();

psauce = pingredientFactory->createSauce();

pcheese = pingredientFactory->createCheese();

}

};

class ClamsPizza : public Pizza { //пицца с мидиями

public:

ClamsPizza(pPizzaIngredientFactory pF) { pingredientFactory = pF; name = "ClamsPizza"; }

void prepare() {

cout << "\n\n Preparing " + name;

pdough = pingredientFactory->createDough();

psauce = pingredientFactory->createSauce();

pclams = pingredientFactory->createClams();

}

};

//

//-------------------------------------------------------------------------------------

class PizzaStore { //пиццерия

public:

pPizza orderPizza(string type) { //заказ пиццы

pPizza pizza = createPizza(type); //создание пиццы

pizza->prepare();

pizza->bake();

pizza->cut();

pizza->box();

return pizza;

}

protected:

virtual pPizza createPizza(string type) = 0;

};

class NYPizzaStore : public PizzaStore {//Пиццерия в Нью-Йорке

public:

pPizza createPizza(string type) {//Используем фабрику ингредиентов из Нью-Йорка для приготовления разных видов пиццы

if (type == "cheese")return pPizza(new CheesePizza( pPizzaIngredientFactory (new NYPizzaIngredientFactory())));

if (type == "clams") return pPizza(new ClamsPizza( pPizzaIngredientFactory (new NYPizzaIngredientFactory())));

return pPizza(new Pizza);

}

};

//Прайс на продукты

//--------------------------------------------------------------------------------------

class Prise;

typedef CountedPtr<Prise> pPrise;

class Prise {

static bool fCreate;

static pPrise prise;

double costDough;

Prise() { costDough = 0.88; }

public:

static pPrise& getInstance() {

if (!fCreate) { fCreate = 1; return prise = pPrise(new Prise()); }

else return prise;

}

double GetCostDout() { return costDout; }

};

bool Prise::fCreate = 0;

pPrise Prise::prise;

//--------------------------------------------------------------------------------------

int main()

{

NYPizzaStore nyp;

pPrise pr = Prise::getInstance();

pr->GetCostDout();

pPizza nyCheesePizza = nyp.orderPizza("cheese");

nyp.orderPizza("clams");

nyp.orderPizza("");

getchar();

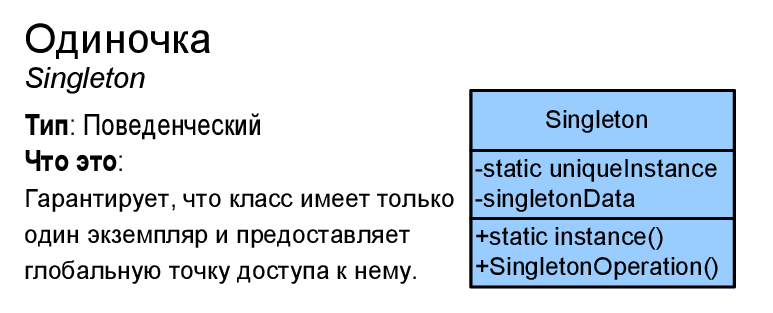
return 0;

}

1. Создайте пиццерию в чикагском стиле.
2. Доопределите класс Prise так, чтобы в нём содержалась информация о стоимости ингредиентов пиццы. Причём объект Prise должен быть общим (единственным) для всех пиццерий.
3. Выведите на экран, из каких компонентов состоит пицца, и сколько она стоит.
4. Модифицируйте код так, чтобы кроме вида пиццы можно было бы указывать и её размер.
5. Модифицируйте код так, чтобы вместе с пиццей можно было заказать и добавки к ней (декоратор).
6. Упростите программу, избавившись от класса CountedPtr.
7. Нарисуйте в отчёте диаграммы фабричного метода и абстрактной фабрики, поясните, в чём их разница.

ПАТТЕРН Уникальный объект.

Паттерн Одиночка направлен на создание уникальных объектов, существующих только в одном экземпляре. Из всех паттернов Одиночка имеет самую простую диаграмму классов; собственно, вся диаграмма состоит из одного единственного класса.



//Прайс на продукты

//--------------------------------------------------------------------------------------

class Prise;

typedef CountedPtr<Prise> pPrise;

class Prise {

static bool fCreate;

static pPrise prise;

double costDout;

Prise() { costDout = 0.88; }

public:

static pPrise& getInstance() {

if (!fCreate) { fCreate = 1; return prise = pPrise(new Prise()); }

else return prise;

}

double GetCostDout() { return costDout; }

};

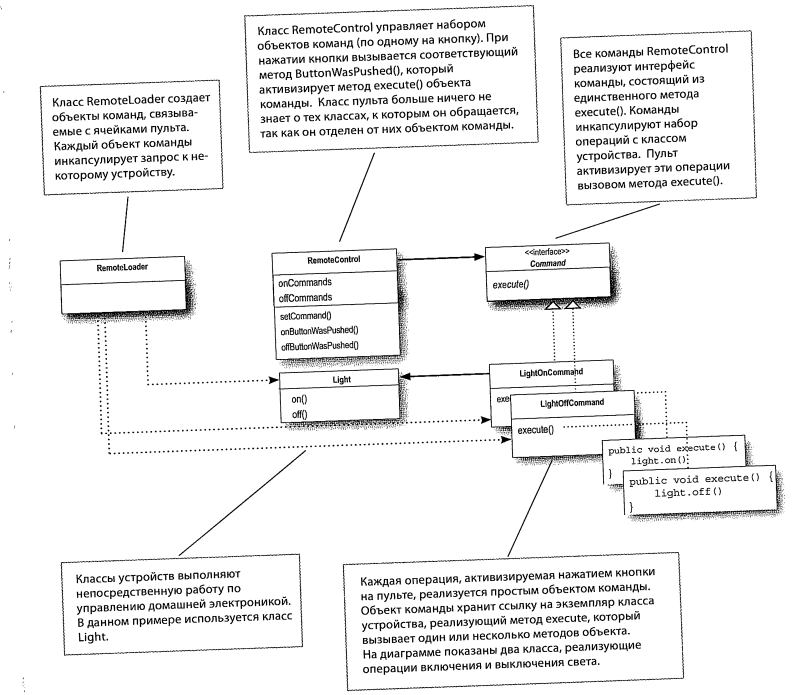
bool Prise::fCreate = 0;

pPrise Prise::prise;

* Конструктор закрытый. Объект существует в программе в одном экземпляре.
* Паттерн Одиночка гарантирует, что в приложении существует не более одного экземпляра данного класса.
* Паттерн Одиночка также предоставляет глобальную точку доступа к этому экземпляру.
* Проанализируйте ограничения по производительности и затратам ресурсов, тщательно выберите реализацию Одиночки для многопоточного приложения.
* Будьте внимательны при использовании загрузчиков классов; они могут привести к созданию нескольких экземпляров, а это противоречит основной цели паттерна.

ПАТЕРН КОМАНДА. Инкапсуляция вызова

Паттерн Команда инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.



* Паттерн Команда отделяет объект, выдающий запросы, от объекта, который умеет эти запросы выполнять.
* Объект команды инкапсулирует получателя с операцией (или набором операций).
* Инициатор вызывает метод execute() объекта команды, что приводит к выполнению соответствующих операций с получателем.
* Возможна параметризация инициаторов командами (даже динамическая во время выполнения).
* Команды могут поддерживать механизм отмены, восстанавливающий объект в состоянии до последнего вызова метода execute().
* Макрокоманды — простое расширение паттерна Команда, позволяющее выполнять цепочки из нескольких команд.
* В них также легко реализуется механизм отмены.
* На практике нередко встречаются «умные» объекты команд, которые реализуют запрос самостоятельно вместо его делегирования получателю.
* Команды также могут использоваться для реализации систем регистрации команд и поддержки транзакций.

Задание: Набрать программу.

#include "stdafx.h"

#include <iostream>

#include <vector>

#include <string>

using namespace std;

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0)

: ptr(p), count(new long(1)) {

}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw()

: ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

// Command

//-----------------------------------------------

class Command {

public:

virtual void execute() = 0;

virtual void undo() = 0;

};

class NoCommand: public Command {

public:

virtual void execute() { cout <<"\n NoCommand "; }

virtual void undo() { cout << "\n NoCommand "; }

};

// Light

//-----------------------------------------------

class Light {

string name;

public:

Light() { ; }

Light(string n) {name = n;}

void off() { cout << "\nOff light in " << name; }

void on() { cout << "\nOn light in " << name; }

};

class LightOnCommand : public Command {

Light& light;

public:

LightOnCommand(Light& light): light(light){}

virtual void execute() { light.on(); }

virtual void undo() { light.off(); }

};

class LightOffCommand : public Command {

Light& light;

public:

LightOffCommand(Light& light): light(light){}

virtual void execute() { light.off(); }

virtual void undo() { light.on(); }

};

//

//------------------------------------------------------

class Music {

int volume;

int prevVolume;

string name;

public:

Music(string name) :volume(0), prevVolume(0){ this->name = name; }

void on() { cout << "\n On music "<< name; }

void off() { cout << "\n Off music "<<name; }

void SetVolume(int vol) {

if (volume + vol >= 0 && volume + vol <= 25) { prevVolume = volume; volume = volume + vol; }

else if (vol < 0) { prevVolume = volume; volume = 0; }

else if (vol > 0) { prevVolume = volume; volume = 25; }

cout << "\n volume = "<< volume;

}

int GetPrevVolume() {return prevVolume; }

};

class MusicOnCommand: public Command {

Music& music;

public:

MusicOnCommand(Music& str) : music(str) {}

virtual void execute() { music.on(); }

virtual void undo() { music.off(); }

};

class MusicOffCommand : public Command {

Music& music;

public:

MusicOffCommand(Music& str) : music(str) {}

virtual void execute() { music.off(); }

virtual void undo() { music.on(); }

};

class MusicUpCommand : public Command {

Music& music;

public:

MusicUpCommand(Music& str) : music(str) {}

virtual void execute() { music.SetVolume(1); };

virtual void undo() { music.SetVolume(music.GetPrevVolume()); };

};

class MusicDownCommand : public Command {

Music& music;

public:

MusicDownCommand(Music& str) : music(str) {}

virtual void execute() { music.SetVolume(-1); };

virtual void undo() { music.SetVolume(music.GetPrevVolume()); };

};

typedef CountedPtr<Command> pCommand;

// RemoteControl

//------------------------------------------------------

class RemoteControl {

vector<pCommand> onCommand;

vector<pCommand> offCommand;

pCommand undoCommand;

public:

RemoteControl(): onCommand(7, pCommand(new NoCommand())), offCommand(7, pCommand(new NoCommand())), undoCommand(pCommand(new NoCommand())) {}

void setCommand(int slot, pCommand onCom, pCommand offCom) {onCommand[slot] = onCom; offCommand[slot] = offCom;}

void onButtonWasPushed(int slot) { onCommand[slot]->execute(); undoCommand = onCommand[slot];}

void offButtonWasPushed(int slot){ offCommand[slot]->execute(); undoCommand = onCommand[slot]; }

void undoButtonWasPushed() { cout << "\nundo"; undoCommand->undo(); }

};

int main()

{

RemoteControl RC;

Light lr("Living room");

Light br("Bath room");

RC.setCommand(0, pCommand(new LightOnCommand(lr)), pCommand(new LightOffCommand(lr)));

RC.setCommand(1, pCommand(new LightOnCommand(br)), pCommand(new LightOffCommand(br)));

Music mlr("Music in living room");

RC.setCommand(2, pCommand(new MusicOnCommand(mlr)), pCommand(new MusicOffCommand(mlr)));

//Громкость музыки

RC.setCommand(3, pCommand(new MusicUpCommand(mlr)), pCommand(new MusicDownCommand(mlr)));

RC.onButtonWasPushed(0);

RC.undoButtonWasPushed();

RC.offButtonWasPushed(0);

RC.onButtonWasPushed(1);

RC.offButtonWasPushed(1);

RC.onButtonWasPushed(2);

RC.offButtonWasPushed(2);

RC.onButtonWasPushed(3);

RC.offButtonWasPushed(3);

RC.onButtonWasPushed(3);

RC.undoButtonWasPushed();

RC.onButtonWasPushed(3);

RC.onButtonWasPushed(6);

RC.undoButtonWasPushed();

getchar();

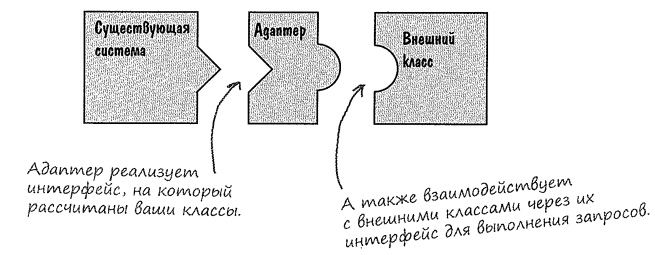
return 0;

}

1. В программе для каждого объекта управления создаётся свой класс команды. Это необходимо переделать. От абстрактного класса «команда» необходимо создать классы: одиночная команда, пустая команда и макрокоманда (объединение нескольких команд в одну). Объекты этих классов должны обеспечивать управление светом, музыкой, вентиляцией, кондиционированием и TV в помещении с 15-ти кнопочного (7 кнопок включить, 7 кнопок выключить и одна кнопка отменить) пульта управления.
2. Программу необходимо модифицировать так, чтобы вёлся журнал последовательности выполнения команд.
3. Модифицировать программу так, чтобы команды сначала помещались в очередь, и только после её переполнения первая команда в очереди выполнялась и удалялась из очереди.

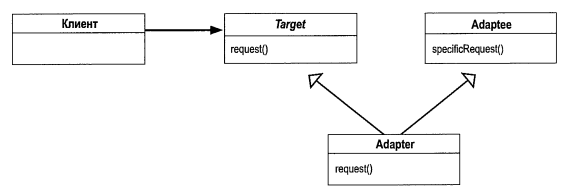
ПАТТЕРНЫ АДАПТЕР и ФАСАД

Объектно-ориентированные адаптеры

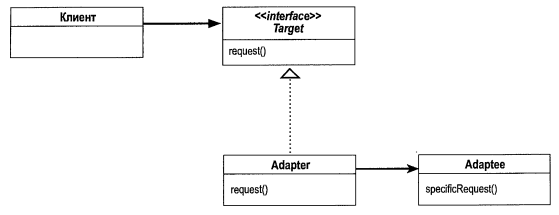


Паттерн Адаптер преобразует интерфейс класса к другому интерфейсу, на который рассчитан клиент. Адаптер обеспечивает совместную работу классов, невозможную в обычных условиях из-за несовместимости интерфейсов. Бываю два типа адаптеров.

Адаптер классов.



Адаптер объектов.



Набрать программу

#include "stdafx.h"

#include <iostream>

using namespace std;

//Утки

//--------------------------------------------

class Duck {

public:

virtual void quack() = 0;

virtual void fly() = 0;

};

class MallardDuck : public Duck {

public:

void quack() {cout << "\n Quack";}

void fly() { cout << "\n I am flying"; }

};

//Индюшка

//------------------------------------------------------------

class Turkey {

public:

virtual void gobble()=0;

virtual void fly() =0;

};

class WildTurkey : public Turkey {

public:

void gobble() { cout << "\n Gobble"; }

void fly() { cout << "\n I'm flying a short distance"; }

};

//Адаптер объектов класса Turkey

//-----------------------------------------------------------------------------------

class TurkeyAdapter : public Duck {

Turkey& turkey;

public:

TurkeyAdapter(Turkey& tk) : turkey(tk) {}

void quack() {turkey.gobble(); }

void fly() { turkey.fly();turkey.fly(); }

};

//Адаптер класса WildTurkey

//----------------------------------------------------------------------------------

class TurkeyAdapter2 : public Duck, public WildTurkey {

public:

TurkeyAdapter2( WildTurkey& turkey) : WildTurkey(turkey){}

void quack() { this->gobble(); }

void Duck::fly() { WildTurkey::fly(); WildTurkey::fly();}

};

//Функция, которая без адаптеров не смола бы работать с объектами Turkey

void testDuck(Duck& duck) {

duck.quack();

duck.fly();

}

int main()

{

MallardDuck duck;

WildTurkey turkey;

TurkeyAdapter turkeyadapter(turkey);

cout << "\n The Turkey says ...";

turkey.gobble();

turkey.fly();

cout << "\n The Duck says...";

testDuck(duck);

cout << "\n The TurkeyAdapter says...";

testDuck(turkeyadapter);

TurkeyAdapter2 turkeyadapter2(turkey);

cout << "\n The TurkeyAdapter2 says...";

testDuck(turkeyadapter2);

cout << "\n";

return 0;

}

1. Определите новый тип данных «Гусь», интерфейс которого не совпадает с интерфейсом класса Duck.
2. Определите классы «Дикий Гусь» и производный от него «Дикий Серый Гусь»
3. Напишите для класса «Дикий Гусь» и «Дикий Серый Гусь» адаптеры объектов и классов так, чтобы функция testDuck() могла с ними работать.

ПАТЕРН ФАСАД

Фасад не только упрощает интерфейс, но и обеспечивает логическую изоляцию клиента от подсистемы, состоящей из многих компонентов.

Фасад применяется для упрощения, а адаптер — для преобразования интерфейса к другой форме.

При использовании паттерна Фасад мы создаем класс, который упрощает и унифицирует набор более сложных классов, образующих некую подсистему. В отличие от многих других паттернов, Фасад относительно прост; в нем нет умопомрачительных абстракций, в которых приходится подолгу разбираться. Но от этого он не становится менее полезным; паттерн Фасад предотвращает появление сильных связей между клиентом и подсистемой и, как вы вскоре увидите, способствует соблюдению нового принципа объектно-ориентированного проектирования (п**ринципа минимальной информированности)**.

***Паттерн Фасад предоставляет унифицированный интерфейс к группе интерфейсов подсистемы. Фасад определяет высокоуровневый интерфейс, упрощающий работу с подсистемой.***

**Принцип минимальной информированности:** общайтесь только с близкими друзьями.

Но как добиться этой цели? Принцип дает некоторые рекомендации. Возьмем произвольный объект; согласно принципу, из любого метода этого объекта должны вызываться только методы, принадлежащие:

■ самому объекту;

■ объектам, переданным в параметрах метода;

■ любому объекту, созданному внутри метода;

■ любым компонентам объекта.

Ограничения кажутся вам слишком жесткими? Какой вред принесет вызов метода объекта, полученного в результате другого вызова? Дело в том, что он обращен к одной из подчастей другого объекта, а следовательно, увеличивает число объектов, о которых непосредственно «знает» текущий объект. В таких случаях принцип требует, чтобы объект выдавал этот запрос за нас (чтобы круг «друзей» оставался по возможности узким).

Пример:

//Принцип игнорируется. Здесь мы получаем объект thermometer от станции а затем вызываем метод getTemperature() самостоятельно.

public float getTemp() {

Thermometer thermometer = station.getThermometer ();

return thermometer.getTemperature();

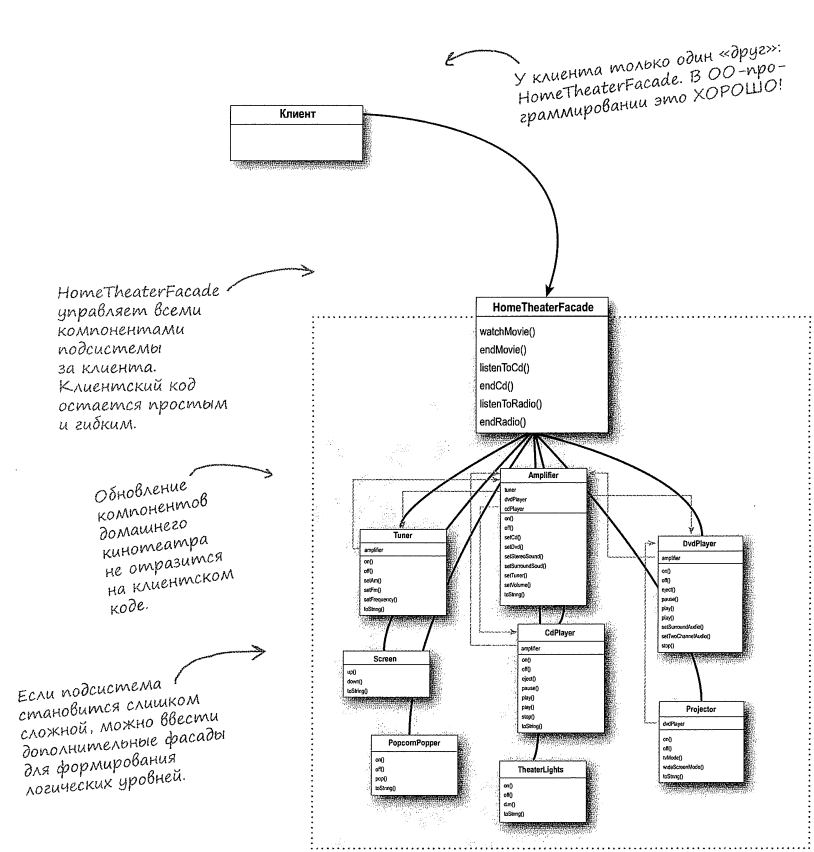
}

//Принцип соблюдается. В класс Station включается метод, который обращается с запросом к thermometer за нас. Тем самым сокращается количество классов, от которых зависит наш код.

public float getTempO {

return station.getTemperature()

}



1. Написать код, который реализует фасад для домашнего кинотеатра (использовать композицию).

* Если вам понадобится использовать существующий класс с неподходящим интерфейсом — используйте адаптер.
* Если вам понадобится упростить большой интерфейс или семейство сложных интерфейсов — используйте фасад.
* Адаптер приводит интерфейс к тому виду, на который рассчитан клиент.
* Фасад изолирует клиента от сложной подсистемы.
* Объем работы по реализации адаптера зависит от размера и сложности целевого интерфейса.
* Реализация фасада основана на композиции и делегировании.
* Существуют две разновидности адаптеров: адаптеры объектов и адаптеры классов. Для адаптеров классов необходимо множественное наследование.
* Для подсистемы можно реализовать несколько фасадов.

ПАТТЕРН СОСТОЯНИЕ

Паттерн Состояние управляет изменением поведения объекта при изменении его внутреннего состояния. Внешне это выглядит так, словно объект меняет свой класс.



Эта диаграмма классов полностью совпадает с диаграммой паттерна Стратегия.

Да, диаграммы классов практически совпадают, но эти два паттерна различаются своей целью.

В паттерне Состояние набор вариантов поведения инкапсулируется в объектах состояния; в любой момент времени контекст делегирует выполнение действий одному из этих объектов. Со временем текущее состояние переключается на другие объекты в соответствии с изменениями внутреннего состояния контекста, так что поведение контекста также изменяется со временем. При этом клиент обычно знает об объектах состояния очень мало (или вообще ничего не знает).

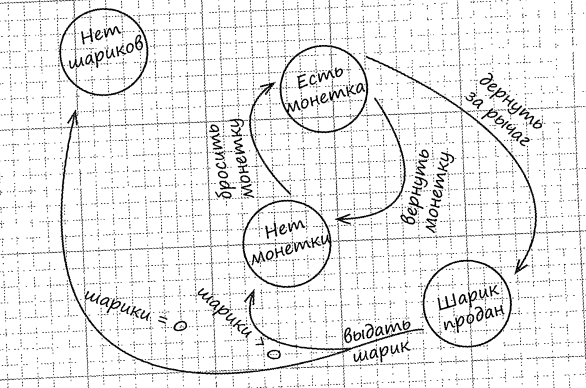
В паттерне Стратегия клиент обычно определяет объект стратегии, связываемый с контекстом. Хотя паттерн обеспечивает необходимую гибкость для изменения объекта стратегии во время выполнения, часто имеется объект стратегии, наиболее подходящий для объекта контекста. Например, в главе 1 некоторые утки инициализировались «типичным» поведением при полете, а для других (резиновых уток и приманок) выбиралось поведение, при котором они оставались на земле.

В общем случае паттерн Стратегия может рассматриваться как гибкая альтернатива субклассированию: если поведение класса определяется наследованием, класс жестко привязывается к этому поведению, даже если позднее его потребуется изменить. Паттерн Стратегия позволяет изменить поведение посредством композиции с другим объектом.

Паттерн Состояние может рассматриваться как замена многочисленных условных конструкций в коде контекста: если поведение инкапсулировано в объектах состояния, для изменения поведения контекста достаточно выбрать другой объект состояния.

Задача, которая приводит к этому паттерны. Конечный автомат, который выдаёт шарик с жевательной резинкой.

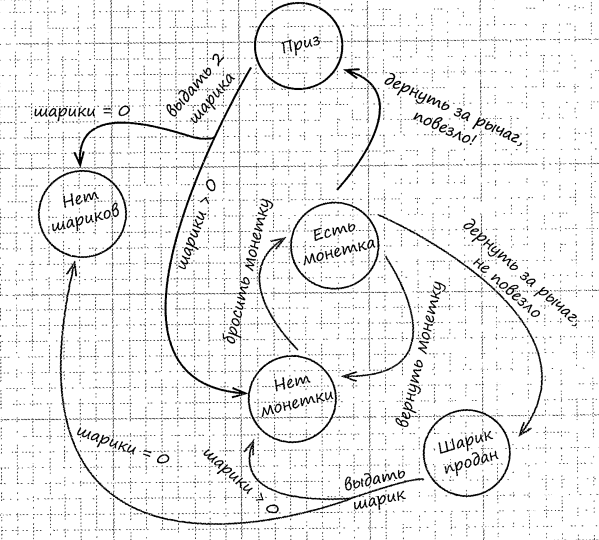
Диаграмма состояний.



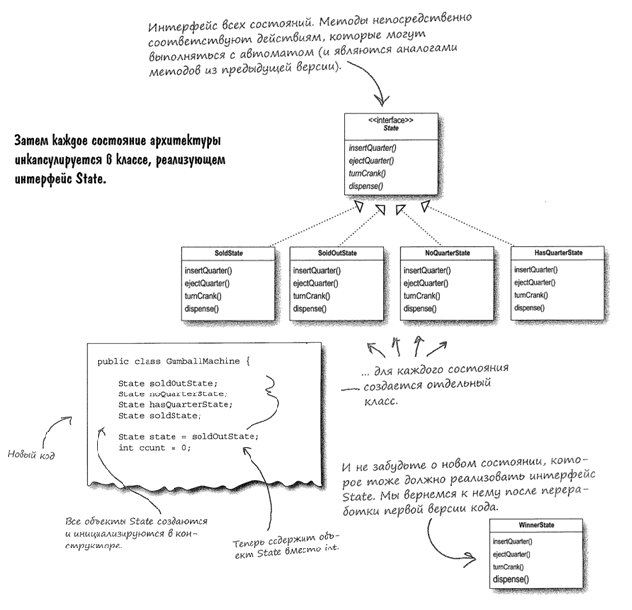
Итак, пришло время реализовать логику автомата для продажи жевательной резинки. Мы уже выяснили, что нам понадобится переменная экземпляра для хранения текущего состояния. Что касается действий, необходимо реализовать действия «бросить монетку», «вернуть монетку», «дернуть за рычаг» и «выдать шарик»; также следует реализовать состояние «Нет шариков».

Лабораторное задание.

1. Реализовать диаграмму состояний в виде программы работы автомата по продаже жевательной резинки.
2. После того как автомат реализован и протестирован, добавляем ещё одно условие. Случайным образом с вероятностью 10% покупатель может получить два шарика.



]



Зачем создавать отдельное состояние WinnerState? Почему бы не выдать два шарика в состоянии SoldState?

Состояния SoldState и WinnerState почти полностью идентичны, только WinnerState выдает два шарика вместо одного. Конечно, код выдачи двух шариков можно разместить и в SoldState. Недостаток такого решения заключается в том, что класс начинает представлять сразу ДВА состояния. Таким образом, за устранение дублирования приходится платить ясностью кода. Также стоит вспомнить о принципе: «Один класс — одна обязанность». Включая обязанности WinnerState в SoldState, вы наделяете SoldState ДВУМЯ обязанностями. А что произойдет, если рекламная акция завершится? Или изменится приз? Таким образом, речь идет о компромиссном решении, принятом на архитектурном уровне.

1. Набрать программу, реализующую объектно-ориентированный вариант реализации конечного автомата.

#include "stdafx.h"

#include <iostream>

#include <string>

using namespace std;

class State {

public:

//действия покупателя или автомата

virtual void insertQuarter() = 0; //покупатель вставил монетку в автомат

virtual void ejectQuarter() = 0;//покупатель нажал на кнопку возврата монетки

virtual void turnCrank() = 0;//покупатель дёрнул за рычаг, чтобы купить шарик

virtual void dispense() = 0;//автомат продал шарик

virtual string getState() = 0;//автомат сообщил в каком состоянии он находится

};

class GumBallMachine {

//возможные состояния автомата

State\* soldOutState; //в автомате нет шариков

State\* noQuarterState;//в автомат не брошена монетка

State\* hasQuarterState;//в автомат брошена монетка

State\* soldState;//в автомате есть шарики и он готов к продажам

//здесь запоминаем текущее состояние автомата

State\* state;

int count = 0;

public:

//Конструктор получает исходное количество шариков и сохраняет его в переменной.

//Он также создает экземпляр State для всех состояний.

GumBallMachine(int numberGumballs);

State\* getSoldOutState();

State\* getNoQuarterState();

State\* getHasQuarterState();

State\* getSoldState();

void insertQuarter();//вставить монету

void ejectQuarter();//вернуть монету

//Для метода dispense() в классе GumballMachine метод действия не нужен, потому что это внутреннее действие;

//пользователь не может напрямую потребовать, чтобы автомат выдал шарик.

//Однако метод dispense() для объекта State вызывается из метода turnCrank().

void turnCrank();//дёрнуть за рычаг (купить шарик)

//Этот метод позволяет другим объектам

// (в частности, нашим объектам State) перевести автомат в другое состояние.

void setState(State\* state);

//Вспомогательной метод releaseBall() отпускает шарик и уменьшает значение переменной count.

void releaseBall();

int getCount();

void getState();

~GumBallMachine();

};

//Состояния - монета отсутствует

class NoQuarterState : public State {

GumBallMachine& gumballMachine;

public:

NoQuarterState(GumBallMachine& gumballMachine):gumballMachine(gumballMachine){ }

//Если в автомат брошена монетка, вывести сообщение и перейти в состояние HAS\_QUARTER

void insertQuarter() {

cout << "\n You inserted a quarter";

gumballMachine.setState(gumballMachine.getHasQuarterState());

}

void ejectQuarter() { cout << "\n You haven't inserted a quarter"; } //Чтобы вернуть монетку, нужно сначала её бросить

void turnCrank() { cout << "\n You turned, but there's no quarter"; } // Нет монетки - нет шарика

void dispense() { cout << "\n You need to pay first"; } //Шарик выдаётся только за монетку

virtual string getState() { return string("\n NoQuarterState"); }

};

//

class HasQuarterState: public State {

//добавить генератор случайных чисел

GumBallMachine& gumballMachine;

public:

HasQuarterState(GumBallMachine& gumballMachine): gumballMachine(gumballMachine) {}

void insertQuarter() { cout << "\n You can't insert another quarter"; }

void ejectQuarter() {

cout<<"\nQuarter returned";

gumballMachine.setState(gumballMachine.getNoQuarterState());

}

void turnCrank() {

cout<<"\n You turned...";

gumballMachine.setState(gumballMachine.getSoldState());

}

void dispense() {

cout<< "\n No gumball dispensed";

}

virtual string getState() { return string("\n HasQuarterState"); }

};

class SoldState : public State {

GumBallMachine& gumballMachine;

public:

SoldState(GumBallMachine& gumballMachine) : gumballMachine(gumballMachine) { ; }

void insertQuarter() { cout << "\n Please wait, we're already giving you a gumball"; }

void ejectQuarter() { cout << "\nSorry, you already turned the crank"; }

void turnCrank() { cout << "\nTurning twice doesn't get you another gumball"; }

void dispense() {

if (gumballMachine.getCount() > 0) {

gumballMachine.setState(gumballMachine.getNoQuarterState());

}

else {

cout << "\nOops, out of gumbll !";

gumballMachine.setState(gumballMachine.getSoldOutState());

}

}

virtual string getState() { return string("\n SoldState"); }

};

class SoldOutState: public State {

GumBallMachine& gumballMachine;

public:

SoldOutState(GumBallMachine& gumballMachine) : gumballMachine(gumballMachine) { ; }

void insertQuarter() { cout<< "\nYou can't insert a quarter, the machine is sold out";}

void ejectQuarter() { cout << "\nYou can't eject, you haven't inserted a quarter yet"; }

void turnCrank() { cout << "\nYou turned, but there are no gumballs"; }

void dispense() { cout<< "\nNo gumball dispensed"; }

string getState(){ return string("\n SoldOutState");}

};

GumBallMachine::GumBallMachine(int numberGumballs) :soldOutState(new SoldOutState(\*this)),

noQuarterState(new NoQuarterState(\*this)), hasQuarterState(new HasQuarterState(\*this)), soldState(new SoldState(\*this)), count(numberGumballs) {

if (numberGumballs > 0) {

//Если количество шариков > 0. Устанавливается состояние NoQuarterState.

state = noQuarterState;

}

else state = soldOutState;

}

State\* GumBallMachine::getSoldOutState() { return soldOutState; }

State\* GumBallMachine::getNoQuarterState() { return noQuarterState; }

State\* GumBallMachine::getHasQuarterState() { return hasQuarterState; }

State\* GumBallMachine::getSoldState() { return soldState; }

void GumBallMachine::insertQuarter() { state->insertQuarter(); }

void GumBallMachine::ejectQuarter() { state->ejectQuarter(); }

void GumBallMachine::turnCrank() {

state->turnCrank();

state->dispense();

}

GumBallMachine::~GumBallMachine() {delete soldOutState; delete noQuarterState; delete hasQuarterState; delete soldState;}

//Этот метод позволяет другим объектам

// (в частности, нашим объектам State) перевести автомат в другое состояние.

void GumBallMachine::setState(State\* state) { this->state = state; }

//Вспомогательной метод releaseBall() отпускает шарик и уменьшает значение переменной count.

void GumBallMachine::releaseBall() {

cout << "\n A gumball comes rolling out the slot...";

if (count != 0) {

count = count - 1;

}

}

int GumBallMachine::getCount() { return count; }

void GumBallMachine::getState() {cout<< state->getState();}

int main()

{

GumBallMachine gumballMachine(5);

gumballMachine.insertQuarter(); gumballMachine.getState();

//Не правильные действия пользователя (не приводящие к изменению состояния)

gumballMachine.insertQuarter();

//Правильные действия пользователя

gumballMachine.turnCrank(); gumballMachine.getState();

//Не правильные действия пользователя (не приводящие к изменению состояния)

gumballMachine.ejectQuarter();

gumballMachine.turnCrank();

// и т.д.

return 0;

}

1. Добавить в программу состояние WinnerState.
2. Модифицировать программу так, чтобы автомат не мог продать следующий шарик, пока покупатель не забрал из автомата только что купленный. В свою очередь это означает, что при выпадении приза автомат выдаёт не два шарика, а возвращает монету за только что проданный шарик.
3. Реализовать функцию заправки автомата шариками.
4. Разработать меню для тестирования программы, т.е. например, будем считать, что нажатие на клавиатуре цифры 1 соответствует действию – «покупатель бросил монету в автомат»; 2 – «покупатель нажал на кнопку возврата монеты» и т.д. При этом на экран должны выдаваться сообщения, соответствующие состоянию автомата.

***Достоинства объектно-ориентированного подхода.***

Прежде всего, мы создали реализацию торгового автомата, которая на структурном уровне отличается от первой (процедурной) версии, но при этом сохранила прежнюю функциональность. Преимущества структурного изменения реализации:

■ Локализация поведения каждого состояния в отдельном классе.

■ Исключение многочисленных конструкций if, усложнявших сопровождение кода.

■ Каждое состояние закрыто для изменения, однако сам автомат открыт для расширения посредством добавления новых классов состояний.

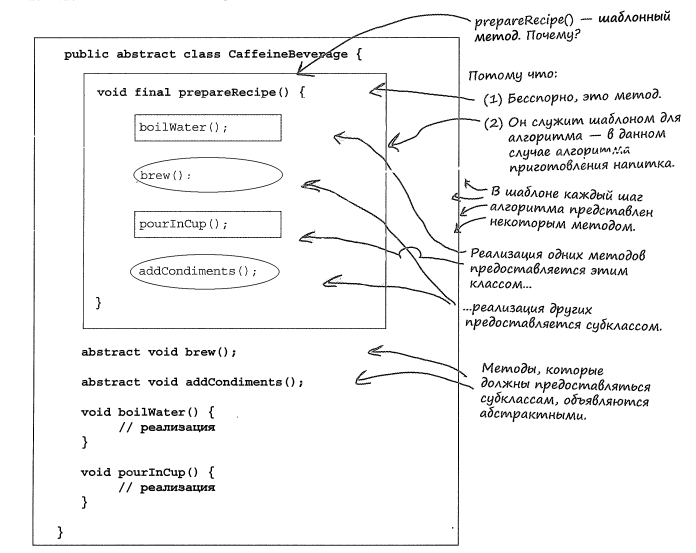
■ Создание кодовой базы и структуры классов, приближенной к диаграмме переходов автомата, и, следовательно — более простой для чтения и понимания.

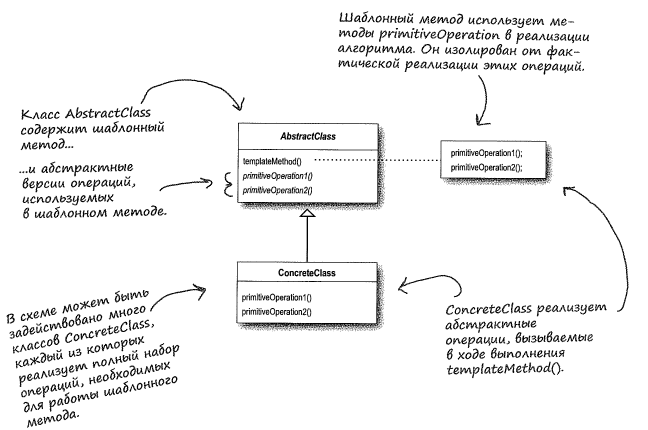
Основные выводы:

* Паттерн Состояние позволяет объекту иметь много разных вариантов поведения в зависимости от его внутреннего состояния.
* В отличие от процедурных конечных автоматов, состояние в этом паттерне представляется полноценным классом.
* Поведение контекста реализуется делегированием выполняемых операций текущему объекту состояния, с которым он связан посредством композиции.
* Инкапсуляция состояния в классе локализует его возможные изменения.
* Паттерны Состояние и Стратегия имеют похожие диаграммы классов, но решают разные задачи.
* Паттерн Стратегия обычно определяет в классе контекста поведение алгоритма.
* Паттерн Состояние изменяет поведение контекста в соответствии с изменениями его состояния.
* Переходами между состояниями могут управлять как классы состояний, так и классы контекстов.
* Применение паттерна Состояние обычно увеличивает количество классов в архитектуре.
* Классы состояний могут совместно использоваться несколькими экземплярами контекстов.

ПАТТЕРН ШАБЛОННЫЙ МЕТОД

Паттерн Шаблонный Метод задает «скелет» алгоритма в методе, оставляя определение реализации некоторых шагов субклассам. Субклассы могут переопределять некоторые части алгоритма без изменения его структуры.





«Перехватчиком» называется метод, объявленный абстрактным классом, но имеющий пустую реализацию (или реализацию по умолчанию). Он дает возможность субклассу «подключаться» к алгоритму в разных точках. Впрочем, субкласс также может проигнорировать имеющийся перехватчик.

Голливудский принцип Не вызывайте нас – мы вас сами вызовем.

Голливудский принцип помогает предотвратить «разложение зависимостей» — явление, при котором компоненты высокого уровня зависят от компонентов низкого уровня, которые зависят от компонентов низкого уровня, которые зависят... и т. д. Разобраться в архитектуре такой системы очень трудно.

Голливудский принцип позволяет компонентам низкого уровня подключаться к системе, но компоненты высокого уровня сами определяют, когда и как они должны использоваться. Иначе говоря, компоненты высокого уровня запрещают компонентам низкого уровня «проявлять инициативу».

Связь между Голливудским принципом и Шаблонным Методом достаточно очевидна: при проектировании с использованием паттерна Шаблонный Метод мы фактически запрещаем субклассам обращаться с вызовами к суперклассу.

1. Написать программу, реализующую паттерн шаблонный метод при приготовлении кофе и чая.
2. Использовать в программе паттерн фабричный метод.
3. Предусмотреть в шаблонном методе функции перехватчики.

Выводы

1. Шаблонный Метод определяет основные шаги алгоритма, оставляя субклассам возможность определения реализации этих шагов.
2. Паттерн Шаблонный Метод играет важную роль в повторном использовании кода.
3. Абстрактный класс Шаблонного Метода может определять конкретные методы, абстрактные методы и перехватчики.
4. Абстрактные методы реализуются субклассами.
5. Перехватчики не делают ничего или определяют поведение по умолчанию в абстрактном классе, но могут переопределяться в субклассах.
6. Чтобы субкласс не мог изменить алгоритм в Шаблонном Методе, объявите последний с ключевым словом final.
7. Голливудский принцип указывает на то, что решения должны приниматься модулями высокого уровня, которые знают, как и когда обращаться с вызовами к модулям низкого уровня.
8. Паттерны Стратегия и Шаблонный метод инкапсулируют алгоритмы; один использует наследование, а другой — композицию.
9. Фабричный Метод является специализированной версией Шаблонного Метода.