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:

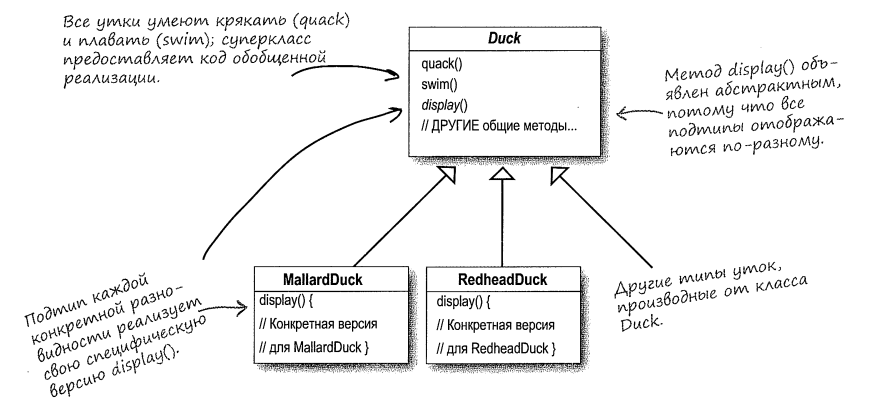
* структурные;
* порождающие;
* поведенческие.

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

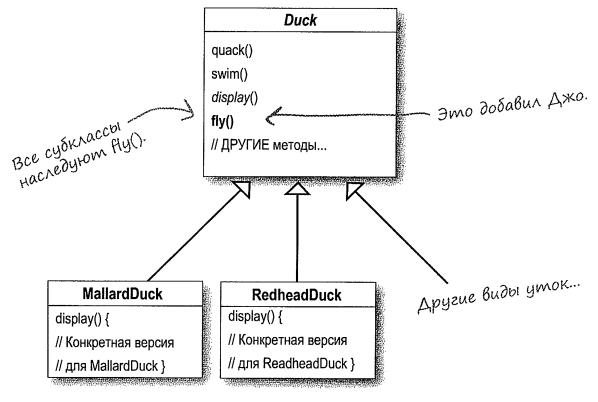
**Порождающие** шаблоны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

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

**ПАТЕРН СТРАТЕГИЯ**

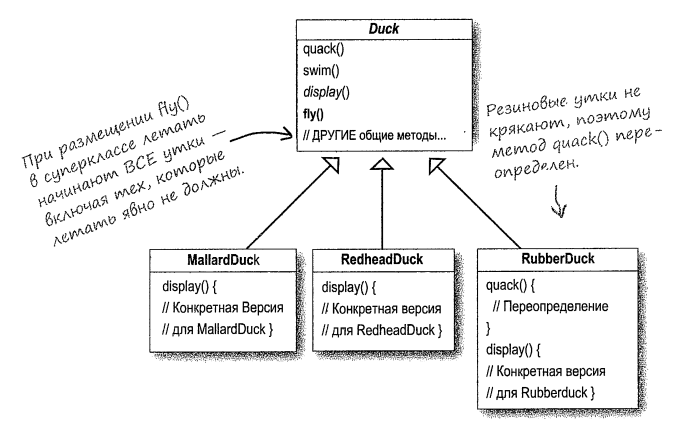
****

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



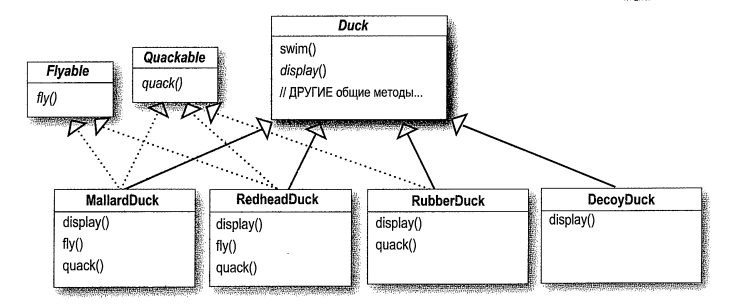
Возникла проблема, как оказалось резиновые утки не должны летать. Другими словами локальное изменение кода привело к нелокальному побочному эффекту.

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.

****

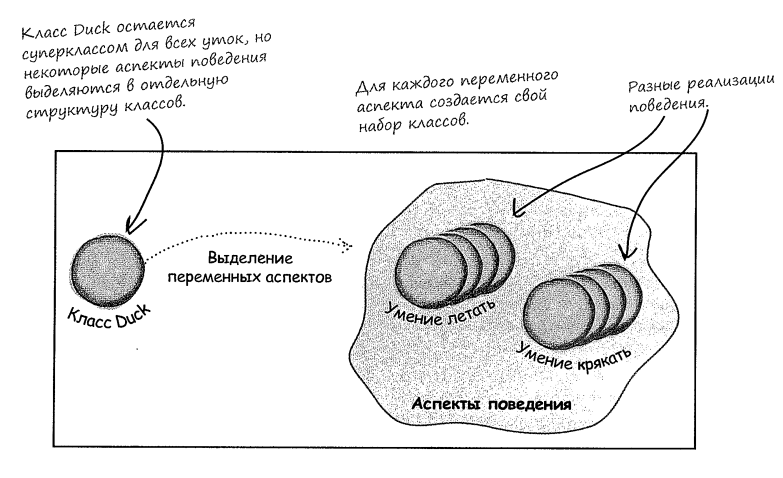
Реализовав их в виде интерфейсных классов. Это уже лучше, но тоже не выход. Так как разные типы уток могут летать и крякать по-разному, следовательно, придётся переопределять эти методы в соответствующих подклассах.

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.***

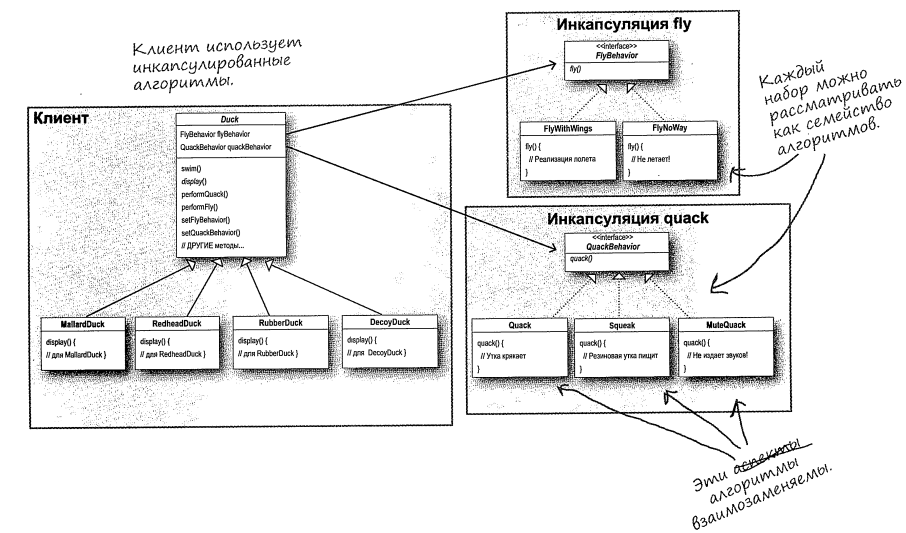
Другими словами, нужно инкапсулировать изменяемые аспекты программы, чтобы они не влияли на работу остального кода. ***Это один из главных принципов паттернов. Все паттерны обеспечивают возможность изменения некоторой части системы независимо от других частей.***

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();

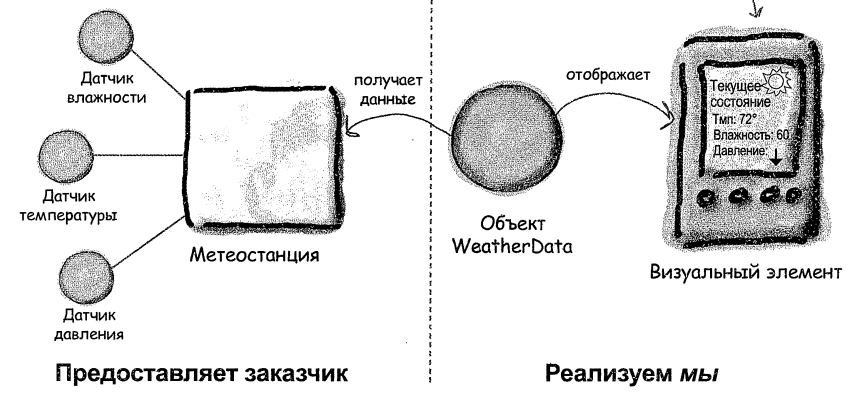
Dd.PerformQuack();

}

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.

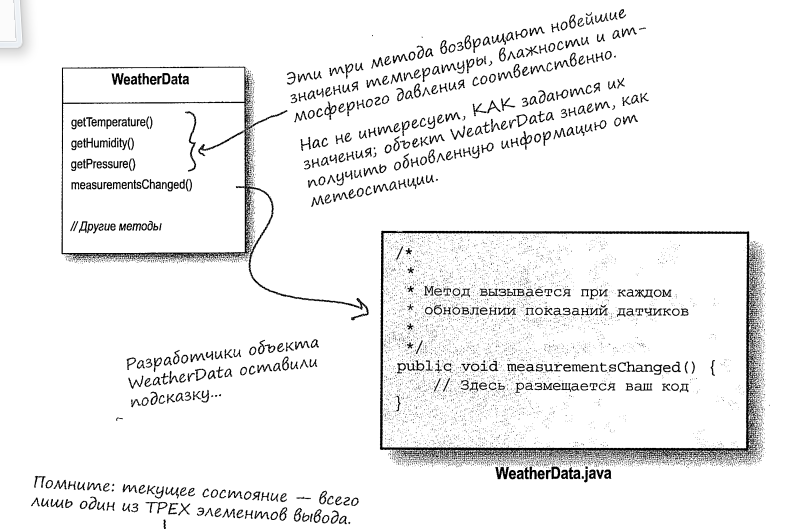
**Паттерн Наблюдатель**

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

}

}

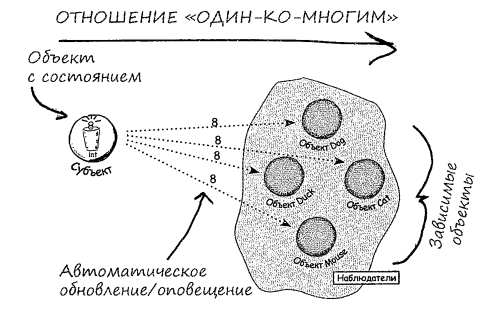
Недостаток. Программа на уровне конкретной реализации, мы не сможем добавлять и удалять визуальные элементы без внесения изменения в программу.

Принцип организации паттерна Наблюдатель

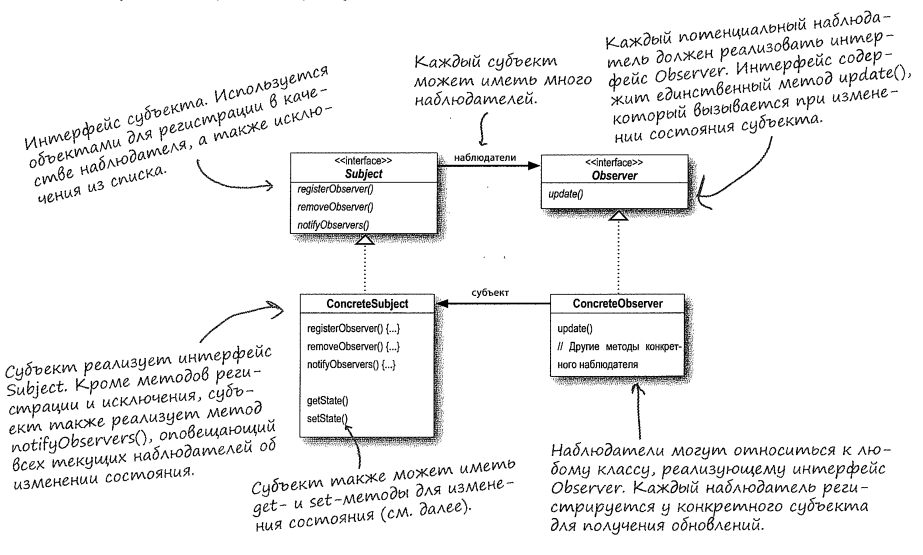
Издатель + Подписчик = Паттерн наблюдатель.

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.***

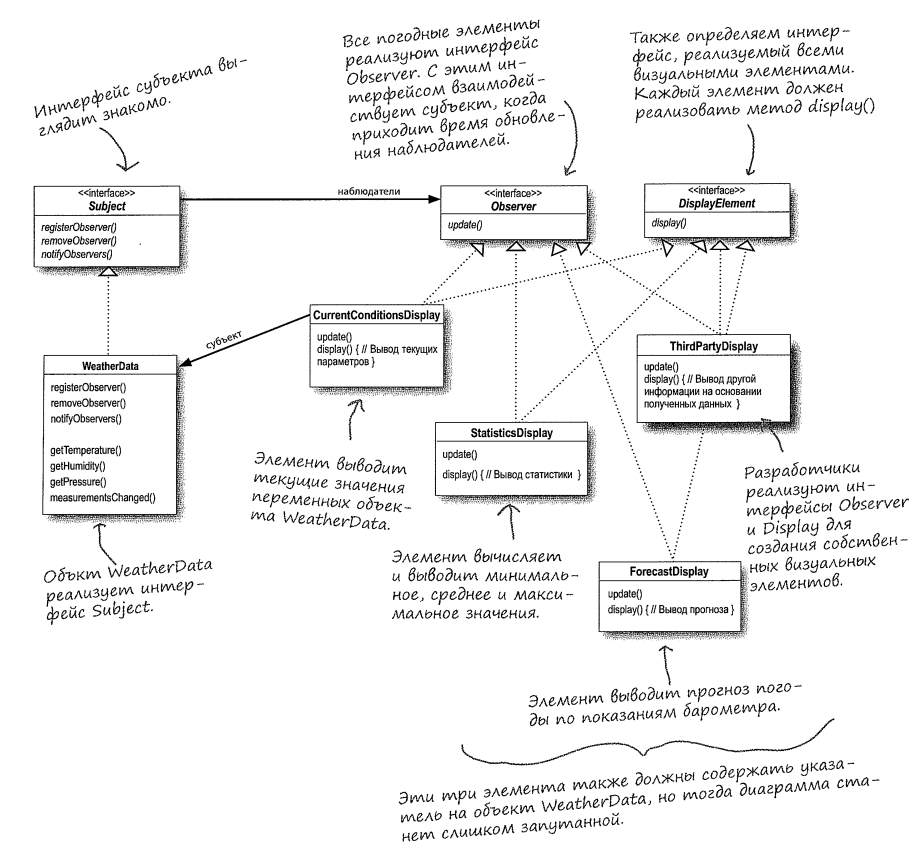
В архитектуре паттерна Наблюдатель между субъектами и наблюдателями существует слабая связь.

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;

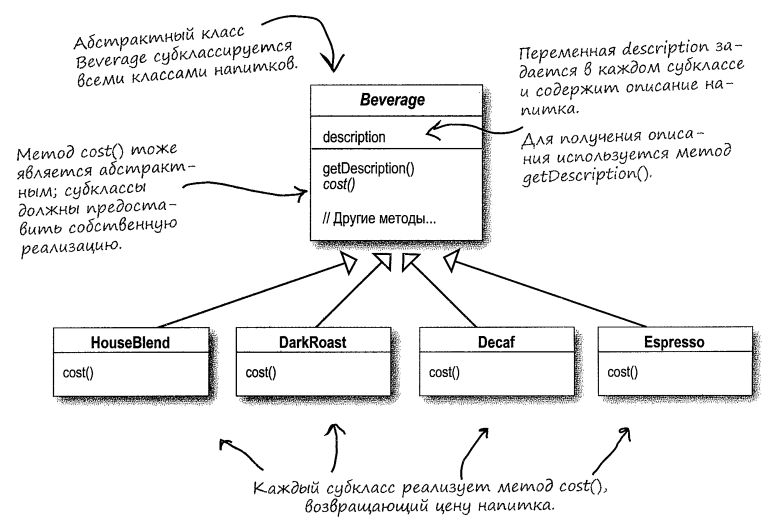
}

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.

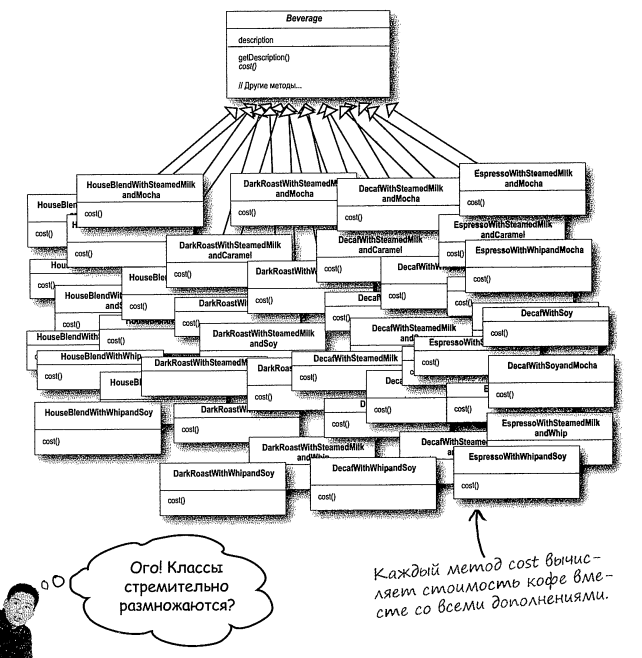
**Паттерн Декоратор**

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*.

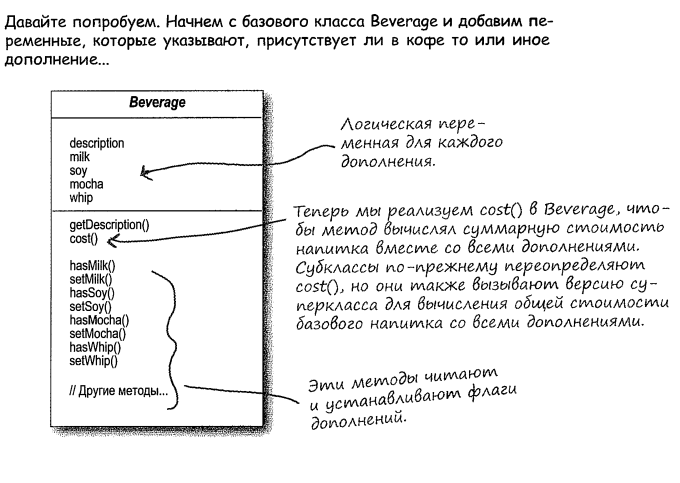
Задача. Сеть кофеин

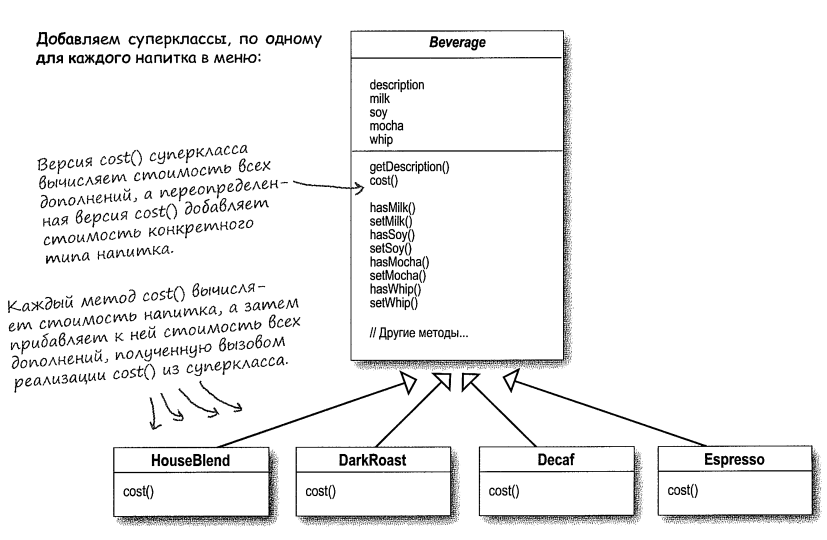


К кофе продаются различные добавки (пенка, шоколад и т.д.). Дополнения не бесплатны, поэтому они должны быть встроены в систему оформления заказов.



Другой подход



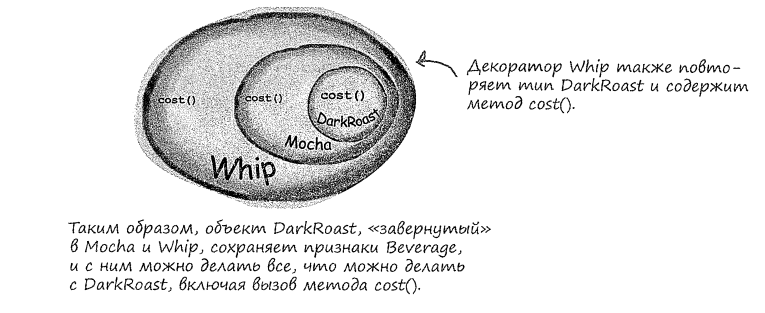


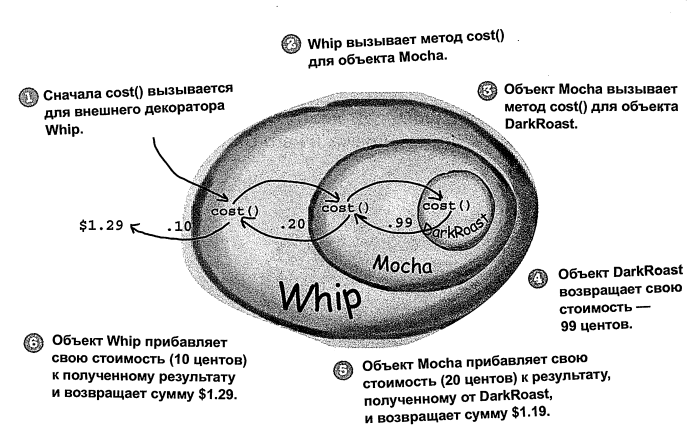
Берем объект DarkRoast;

Декорируем его объектом Mocha;

Декорируем его объектом Whip;

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



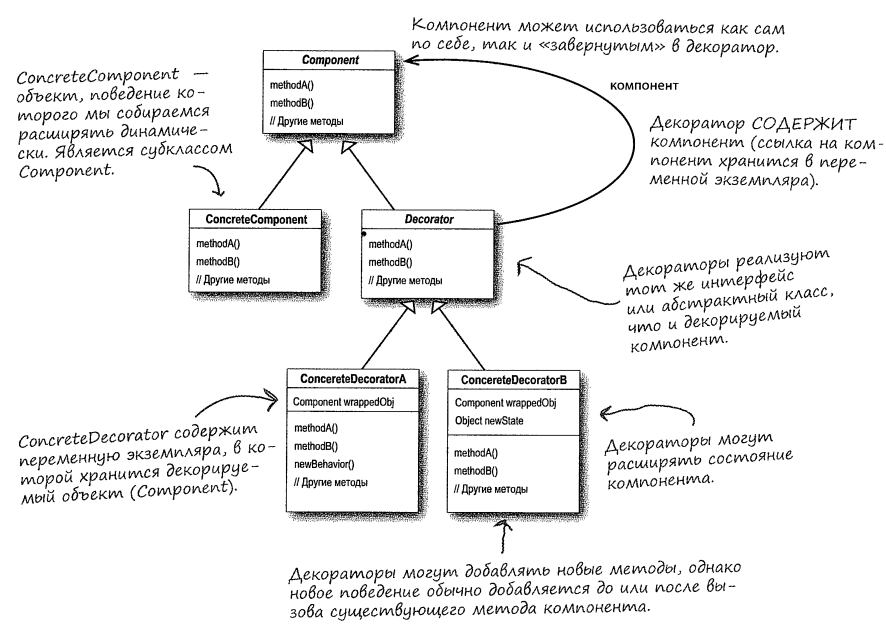


Что известно

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));

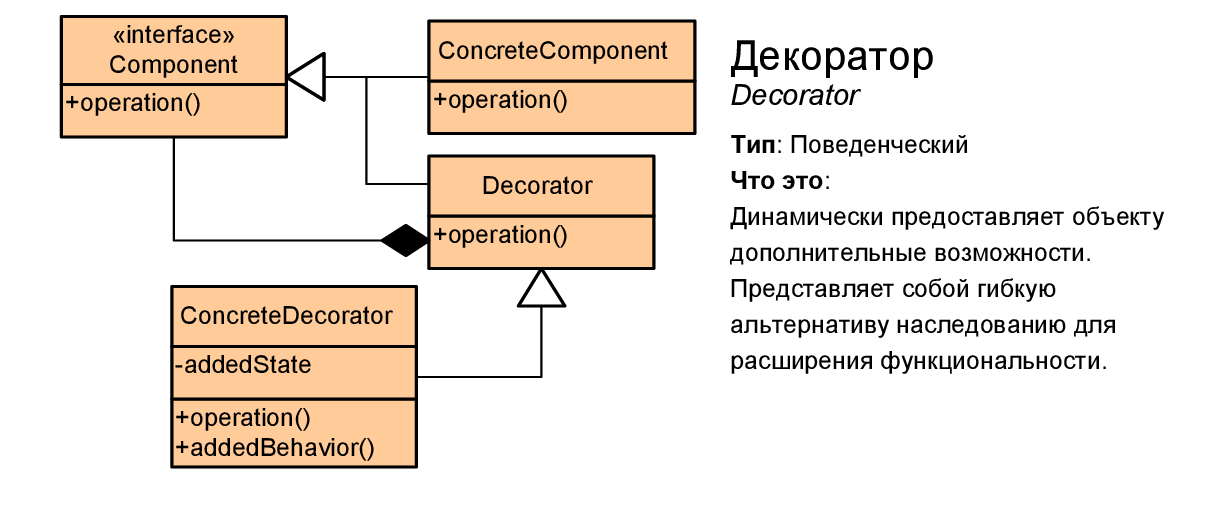
cout << pB2->GetDiscription()+ " $ " << pB2->cost();

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.



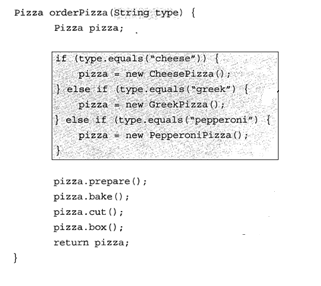
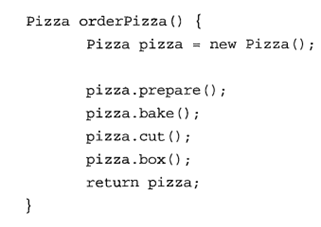
ПРОСТАЯ ФАБРИКА

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.

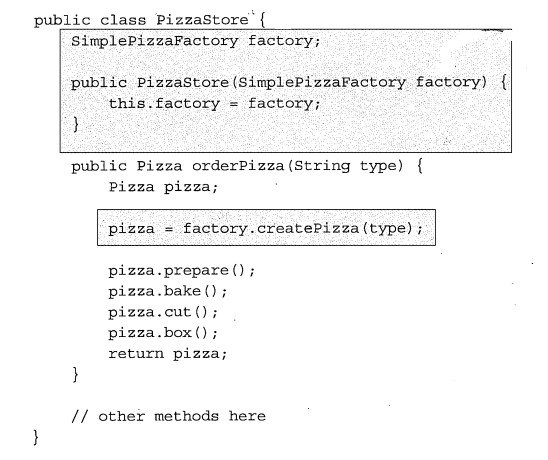
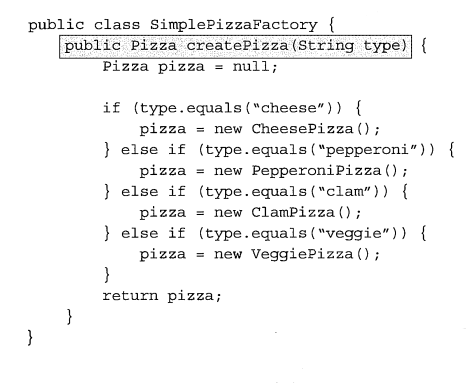
Видим new— подразумеваем конкретный.

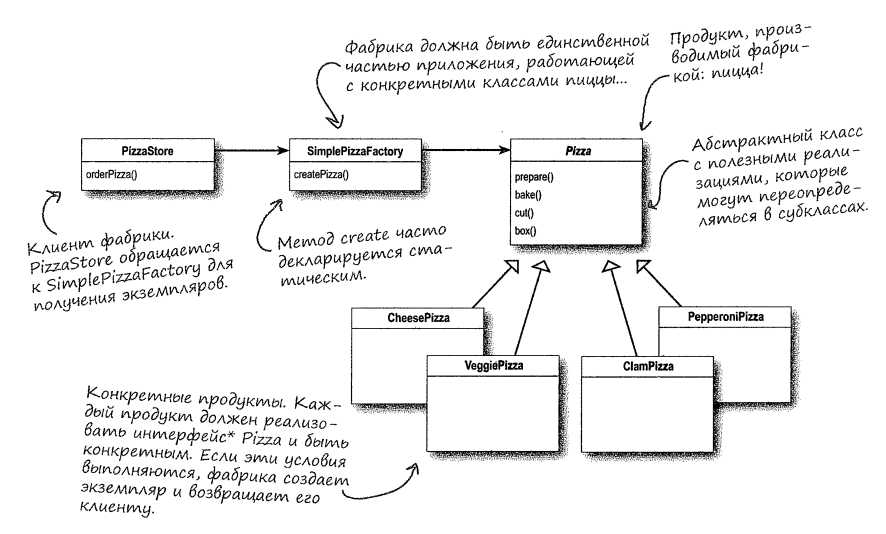
При использовании new вы создаете экземпляр конкретного класса, поэтому эта операция относится к уровню реализации, а не интерфейса. А вы уже знаете, что привязка кода к конкретному классу делает его менее гибким и устойчивым к изменениям.

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();

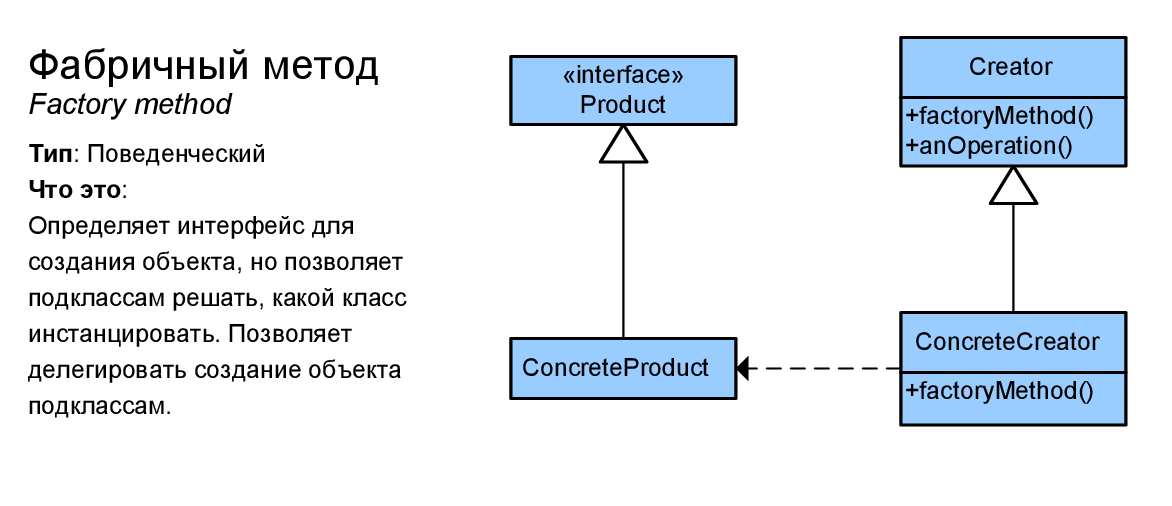
return 0;

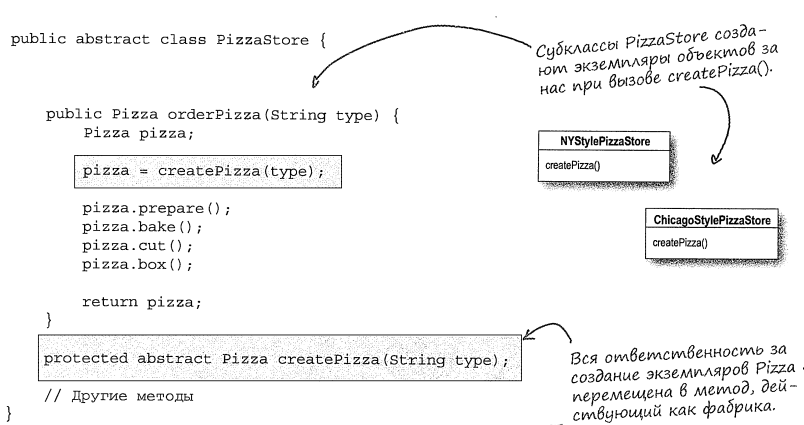
}

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.