# Полиморфизм

Вызов обычных функций и методов происходит через механизм, называемый статическим или ранним связыванием (static (early) binding). Т.е. во время компиляции для кода функции выделяется память в сегменте кода и затем соответствующий адрес связывается с именем функции.

class Base {

public:

void method() {

cout << "Базовый класс";

}

};

class Derived : public Base {

public:

void method() {

cout << "Производный класс";

}

};

Base b;

Derived d;

b.method(); *//Базовый класс*

d.method(); *//Производный класс*

Base \*b = new Derived;

Derived \*d = new Derived;

b->method(); *// Базовый класс*

d->method(); *// Производный класс*

Для реализации полиморфизма необходимо позднее связывание (late (dynamic) binding).

Т.е. связывание, осуществляемое во время выполнения программы. В С++ поздним связыванием обладают указатели на функции, а в ООП виртуальные функции или методы.

class Base {

public:

virtual void method() {

cout << "Базовый класс";

}

};

class Derived : public Base {

public:

*/\* virtual \*/* void method() {

cout << "Производный класс";

}

};

Base \*b = new Derived;

Derived \*d = new Derived;

b->method(); *// Производный класс*

d->method(); *// Производный класс*

Когда в базовом классе объявляется хотя бы одна виртуальная функция, то для всех полиморфных классов создается таблица виртуальных методов (virtual function table).

Это одномерный массив указателей на функции, количество элементов массива = количеству виртуальных функций в классе. В таблице виртуальных функций содержатся настоящие адреса методов. В базовом классе объявляется поле \_\_vfptr (адрес массива), который наследуется всеми производными классами. Механизм позднего связывания позволяет осуществить полиморфизм, но мы платим за это памятью и быстродействием. Если в коде используется полиморфизм, то необходимо объявлять деструктор базового класса виртуально. Иначе при разрушении объектов будет вызываться деструктор базового класса.

**Полиморфизм** означает хранение различных объектов полиморфных классов в одном месте и использования одинакового синтаксиса вызова методов.

Часто не требуется создавать объекты базового класса, он нужен только для построения иерархии. Поэтому следует объявить этот класс абстрактным. Для того чтобы класс стал абстрактным нужно добавить ключевое слово abstract хотя бы в один метод.

class Base {

virtual void f();

}

Пример. Реализовать имитатор скринсейвера «звездное небо». Необходимо зажигать и гасить несколько типов звезд.

*//starnight.h*

#ifndef \_STARNIGHT

#define \_STARNIGHT

#include <window.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <memory.h>

#include <conio.h>

#define PI 3.3141592

class BaseStar {

    static HWND hwnd; *//Дескриптор окна*

    static HDC hdc; *//Контекст устройства*

    RECT rect;

    BaseStar(int width, int height, int x, int y) {

        rect.left = x - width / 2;

        rect.right = x + width / 2 + width % 2;

        rect.top = y - height / 2;

        rect.bottom = y + height / 2 + height % 2;

    }

public:

    virtual void show();

    virtual ~BaseStar() { I

InvalidateRect(hwnd, &red, TRUE);

    }

    friend class StarsNight;

};

class EllipseStar : public BaseStare {

    public:

        EllipseStar(int width, int height, int x, int y)

            :BaseStar (int width, int height, int x, int y) {}

    virtual void show() {

        selectObject(

            hdc,

            CreateSolidBrush(RGB(rand() % 256, rand() % 256, rand() % 256))

        );

        Ellipse(hdc, rect.left, rect.top, rect.right. rect.bottom);

    }

}

class PolygonStar : public BaseStar {

    POINT \*p;

    int countCorners;

    public:

        PolygonStar(int width, int height, int x, int y, int n) : BaseStar(width, height, x, y) {

        p = (POINT\*) calloc(n, sizeof(POINT));

        double alfa = PI / 1000, a = width / 2., b = height / 2., delta = 2 \* PI / n, tgAlfa;

        for (int i = 0; i < n; ++i) {

            tgAlfa = tan(alfa);

            double tmpx = 1. / sqrt(1 / a / a + tgAlfa \* tgAlfa / b / b);

            if (alfa > PI / 2 && alfa < 3 \* PI / 2)

                tmpx = -tmpx;

            double tmpy = tgAlfa \* tmpx;

            p[i].x = x + (long) tmpx;

            p[i].y = y + (long) tmpy;

            alfa += delta;

        }

    }

    ~PolygonStar() {

        free(p);

    }

        virtual void show() {

        SelectObject(

            hdc,

            CreateSolidBrush(RGB (rand() % 256, rand() % 256, rand() % 256));

        );

        Polygon(hdc, p, countCorners);

        }

}

HWND BaseStar::hwnd = GetConsoleWindow();

HDC BaseStar::hdc = GetDC(BaseStar::hwnd);

class StarNight {

    RECT clRect;

    int countStart;

    int minWidth, maxWidth, minHeight, maxHeight;

    int pause;

    BaseStar \*\*bs;

    public:

        StarNight(

            int c = 30,

            int minW = 25,

            int maxW = 50,

            int minH = 25,

            int maxH = 50,

            int p = 333

        ) {

            countStar = c;

            minWidth = minW;

            maxWidth = maxW;

            minHeight = minH;

            maxHeight = maxH;

            pause = p;

            bs = new BaseStar\*[countStars];

            memset(bs, 0, sizeof(BaseStar\*) \* countStars);

            GetClientRect(BaseStar::hwnd, & clRect);

            srand((unsigned int) time(0));

        };

        ~StarNight() {

            hide();

            delete []bs;

        };

        void show() {

            for (int i =0; i < count; ++i) {

                int width = rand() % (maxWidth - minWidth + 1) + minWidth;

                int height = rand() % (maxHeight - minHeigth + 1) + minHeight;

                int x = rand() % (clRect.right - clRect.left - width) + width / 2;

                int y = rand() % (clRect.bottom - clRect.top - height) + height / 2;

                if (rand() % 2)

                    bs[i] = new EllipseStar(width, height, x, y)

                else

                    bs[i] = new PolygonStar(width, height, x, y, rand % 8 + 3);

                bs->show();

                sleep(pause);

            }

            int i = 0;

            while (!\_kbhit()) {

                delete bs[i];

                int width = rand() % (maxWidth - minWidth + 1) + minWidth;

                int height = rand() % (maxHeight - minHeigth + 1) + minHeight;

                int x = rand() % (clRect.right - clRect.left - width) + width / 2;

                int y = rand() % (clRect.bottom - clRect.top - height) + height / 2;

                if (rand() % 2)

                    bs[i] = new EllipseStar(width, height, x, y)

                else

                    bs[i] = new PolygonStar(width, height, x, y, rand % 8 + 3);

                bs->show();

                sleep(pause);

                i = (i + 1) & countStars;

            }

            \_getch();

            hide();

        };

        void hide() {

            for (int i = 0; i < countStart; ++i) {

                delete bs[i];

                bs[i] = 0;

            }

        }

}

#endif

*//starsNight.cpp*

#include "stdafx.h"

#include "starNight.h"

#include <iostream>

int main() {

    StarNight sn;

    std::cout << "Begin StarNight\n";

    sn.show();

    std:cout << "End StarNight";

    return 0;

}

# Паттерное проектирование

Проектирование объектно-ориентированных систем является непростой задачей. Необходимо разработать иерархии классов, определить их интерфейсы, определить отношение между классами и объектами. Оказалось, что нет необходимости решать каждую задачу в нуля, можно использовать решения, которые оказались удачными в прошлом. Первый каталог паттернов проектирования был описан в книге 1994 года «Приемы объектно-ориентированного проектирования». Паттерны классифицируются следующим образом.

## 1 классификация

**Пораждающие паттерны**

* Фабрика

**Поведенческие паттерны**

Распределяет обязанности между классами и объектами

* Комманда
* Состояние
* Стратегия

**Структурные**

Объеденяет классы и объекты в более крупные структуры

* Декоратор
* Адаптер
* Мост

### 2 классификация

Зависит от того относится ли к классам или бъектам.

Паттерны классов описывают отношение между классами посредствам наследования. Данные отношение определяются на стадии компиляции.

* Шаблонный метод
* Адаптер
* Фабричный метод
* Интерпретатор

Паттерны объектов определяет отношение по средствам композиции. Отношение определяется на стадии выполнения.

* Стратегия
* Посетитель
* Компоновщик

В общем случае паттерн состоит из 4 элементов.

**Имя**. Составшись на имя проектировщик сразу же определяет проблему. Ее решения и их последствия.

**Задача.** Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст.

**Решение.** Описание элементов дизайна, отношений между ними и функций каждого элемента.

**Результаты.** Следствие применения паттерна и разного рода компромисы.

**Паттерн проектирования –** Описание взаимодействия объектов и классов, адаптированных для общих задач проектирования в конкретном контексте. Именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые позволяют Применить его для создания Повторно используемого дизайна. Вычлениет классы и объекты, их роль, отношения и функции.

## Стратегия

Программист работает в компании, выпускающей игру «Иммитатор утинного пруда». В пруду плавают и крякают различные утки. Разработан класс Duck, от которого наследуются конкретные виды уток.

class Duck {

    ab void quack();

    virtual void swim();

    virtual void display();

}

class MallardDuck : public Duck {}

class RedheadDuck : public Duck {}

Руководство решило добавить возможность полёта. 1 вариант – добавить метод fly() в Duck. В результате оказалось, что летать стали и резиновые утки. То есть новое поведение оказалось неподходящим для некоторых объектов. В классе резоновой утки переопределен метод quack(), т.к. эти утки не крякают, а пищат. По аналогии можно было бы переопределить метод fly(). Но руководство сообщило, что продукт должен будет часто обновляться и придется искать и переопределять fly() и quack() для каждого подкласса, включенного в программу.

Программист решил изменить структуры следующим образом. Методы летания и издавания звуков перенести в соответствующий интерфейс.

**class** Duck {

**public** **abstract** swim();

**public** **abstract** display();

}

**interface** Quackable {

**void** quack();

}

**interface** Flyable {

**void** fly();

}

**class** MallardDuck **extends** Duck **implements** Quackable, Flyable {

    @Override

**public** display();

    @Override

**public** quack();

    @Override

**public** fly();

}

**class** RubberDuck **extends** Duck **implements** Quackable {

    @Override

**public** display();

    @Override

**public** quack();

}

**class** DecoyDuck **extends** Duck {

    @Override

**public** display();

}

Остается проблема одинакового переопределения методов в разных классах.

Необходимо выделить переменные составляющие и инкапсулировать их, чтобы позднее их можно было изменять без воздействия на постоянные составляющие.

Создадим 2 набора классов. 1 для полета, 1 для кряканья.

**interface** FlyBehavior {

**void** fly();

}

**interface** QuackBehavior {

**void** quack();

}

**class** FlyWithWings **implements** FlyBehavior {

    @Override

**void** fly();

}

**class** FlyHowag **implements** FlyBehavior {

    @Override

**void** fly();

}

**class** Quack **implements** QuackBehavior {

    @Override

**void** quack();

}

**class** MuteQuack **implements** QuackBehavior {

    @Override

**void** quack();

}

**class** Squack **implements** QuackBehavior {

    @Override

**void** quack();

}

Далее свяжем эти иерархии с супер классом посредствам композиции.

сlass Duck {

    FlyBehavior flyBehavior;

    QuackBehavior quackBehavior;

**public** swim();

**public** display();

}

В результате появилась возможность менять аспекты поведения во время выполнения программы.

## Observer

Разработка ПО для метеостанции. Вместе с оборудованием поставляется исходный код класса WeatherDate, отслеживающий текущие погодные условия (температура, влажность, давление). Необходимо создать приложение, содержащее 3 визуальных элемента (текущая сводка, статистика погоды, прогноз). Все данные следует обновлять в реальном времени. Когда объект WeatherDate получает новые данные. Необходимо предусмотреть возможность расширения приложения. Для этого определить программный интерфейс, чтобы другие разработчики могли погдключать собственные визуальные элементы.

Класс WeatherDate содержит методы для считывания параметров и метод изменения показаний, который автоматически вызывается при изменении значения одного из 3 показаний. Разработчики советую добавлять в свой код данный метод.

void measurementChanged () {

    float t = getTemperature();

    float h = getHumodity();

    float p = getPressure();

    currentConditionDisplay.update(t, h, p);

    statisticDisplay.update(t, h, p);

    toreinstConditionDisplay.update(t, h, p);

}

При добавлении нового виджета придется менять метод measurementChanged(). Нет возможности подключать\отключать элементы.

Рассмотрим концепцию паттерна наблюдатель. Есть аналогия с подпиской на переодические издания. Подписка оформляется у конкретного издателя. При выходе очередного номера он доставляется подписчику. В паттерне наблюдатель издатель называется субъектом, а подписчик наблюдателем. Паттерн определяет отношение один ко многоим между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых элементов.

В ахрхитектуре паттерна реализована слабая связь между субъектом и наблюдателем. Единственное, что знает субъект о наблюдателе, что тот реализует интерфейс Observer.

## Proxy(заместитель)

Предоставляет суррогатный объект, управляющий доступом к другому объекту. Выделяют следующие варианты заместителей.  
**Удаленный заместитель**

В этой разновидности заместитель выполняет функции оригинального объекта.

**Виртуальный заместитель**

Представляет объект, создание которого связано с большими затратами.  
**Защитный заместитель**

Контролирует доступ к объекту в соответствии с системой привилегий.

Bridge(мост)

Предположим, необходимо реализовать пульт для телевизора. Понятно, что в дальнейшем будут меняться как пульты, так и телевизоры. Т.е. необходимо предусмотреть изменение как абстракции (пульт), так и реализации (телевизор).

Builder(cтроитель)

Инкапсулирует строение объекта и делит его на этапы. Предположим необходимо создать систему планирования экскурсий. Гости могут выбрать гостиницу, заказать билеты на мероприятие, ужин и т.д. Поездки могу различаться по количеству дней. И по составу развлекательной программы. Строитель позволяет инкапсулировать программу создания экскурсий, в результате чего будет создана соответствующая структура данных.

### Chain of resposibility(цепочка обязанностей)

Данный паттерн используется когда необходимо предоставить возможность обработки запроса нескольким объектам. Например, обработка электронной почты. Фирма выпустила неекий продукт, и должна сортировать почту по следубщим алгоритму: благодарности должны поступать руководству. Жалобы в юридический отдел, запросы на установку продуктов в отдел коммерческого развития, а спам должен удаляться. Разработаны эврестические детекторы, позволяющие определить к какой категории относится письмо. В паттерне цепочка обязанностей реализуется цепочка объектовв, последовательно обрабатывающих запрос. Каждый объект либо обрабатывает запрос, либо отправляет слудующим.

### Mediator(посредник)

Паттерн посредник используется для централизации сложных взаимодействий и управляющих операций между объектами. Данный паттерн формирует слабую связанность между объектами и избаляет объекты от необходимости явно ссылаться друг от друга. Пример – умный дом. Срабатывание будтльника дает сигнал кофеварке начать варить кофе. Но по выходным будильник не срабатывает. Включение посредника упрощает объекты, но при этом объекты извещают посредника об изменении своего состояния и отвечают на запросы посредника. Таким образом посредник содержит управляющую логику системы.

## Антипаттерны проектирования

Паттерн предоставляет решение задачи, а антипаттерн распространенное плохое решение задачи. С первого взгляда антипаттерн выглядит как хорошее решение, но после применения данное решение оказывается неудачным. Основные свойства антипаттерна:

**Антипаттерн показывает почему плохое решение выглядит привлекательным.**

**Антипаттерн объясняет почему данное решение плохо в долгосрочной перспективе.**

**Антипаттерн прдлагает правильный выход из ситуации.**

Золотой молоток

Проблема: требуется выбрать технологии для разработки. Контекст: технология, с которой хорошо знакому группа разработчиков плохо соответствует требованиям разрабатываемой системы. Сопутствующие факторы:

**Разработчики стремятся использовать знакомые технологии.**

**Группа разработки незнакома с другими технологиями.**

**Переход на новую технологию связан с риском.**

Предлагаемое решение. Использовать знакомую технологию.  
Рекоммендуемое решение. Повышение квалификации разработчиков и переход на новую технологию.