ООП: Архитектурное проектирование и паттерны программирования (09.03.04 – программная инженерия)

Архитектурное проектирования и паттерны программирования

Семестр 16 недель
Лекции: 16 часов (1 пара в 2 недели)
Лабораторные работы: 32 часа на подгруппу == 1 пара в неделю на подгруппу
Расчетное задание
Экзамен

Лекции:
Крючкова Елена Николаевна
Лабораторные работы
ПИ-01(А,Б) Крючкова Елена Николаевна
ПИ-02(А, Б) Рахманин Данила Сергеевич

Литература

1. Эрик Фримен и др. Паттерны проектирования
2. Эрих Гамма и др. Паттерны проектирования
3. Мартин Фаулер, Кендалл Скотт. UML. Основы.
4. Крючкова Е.Н., Старолетов С.М. ООП: Архитектурное проектирование и паттерны программирования. – Барнаул, 2020

Темы лекций

1. качество ПО (2 часа)
2. базовые паттерны (2 часа)
3. структурные паттерны (4 часа)
4. порождающие паттерны (2часа)
5. паттерны поведения (4 часа)
6. паттерн MVC (1 час)
7. антипаттерны (1 час)

Лабораторные работы

1 Code review -2 часа

2 Проект системы - первая итерация - 4 часа

3 Делегирование и proxy - 2 часа

4 Структурные 1 (Adapter, Decorator , Composite, Iterator) - 4 часа

5 Структурные 2 (Bridge,Flyweight, Facade, Information Expert) - 4 часа

6 Порождающие 1 (Factory method , Abstract factory , Singleton , Prototype, Object Pool) - 4 часа

7 Порождающие 2 (Builder,) – 2 часа

8 Поведения 1 (State , Memento , Observer) – 4 часа

9 Поведения 2 (Command, Indirection , Visitor) – 4 часа

10 Итоговое занятие - 2 часа

Расчетное задание

Содержание отчета по РЗ

1. Описание предметной области
   - 1.1 Общая характеристика решаемых задач в предметной области
   - 1.2 Характеристика поставленной задачи
   - 1.3 Действующие объекты и функционал
   - 1.4 Возможные расширения системы
2. Объекты, интерфейсы и классы проектируемой системы
   - 2.1 Перечень классов и объектов
   - 2.2 Назначение классов
   - 2.3 Основные методы классов
   - 2.4 Отношения между классами
3. Пример реализации подсистемы на основе принципа делегирования
   - 3.1 Диаграмма классов
   - 3.2 Назначение классов
   - 3.3 Логи работы программы

Тема 0. Качество ПО или зачем менять код, если и так все работает

Стив Макконнелл «Совершенный код»

Правила хорошего кода по версии GeekBrains

О чем будем говорить сегодня?

1. Основные понятия ООП (повторение - залог успеха!)
2. Принципы хорошего кода. SOLID
3. Основы языка UML, виды диаграмм
4. Понятие паттерна проектирования
5. Три класса паттернов

Вспомним основные понятия ООП

1. Класс - это комплексный тип данных, элементы которого данные и функции (методы).
2. Объект (экземпляр класса) – данное, поведение которого полностью определяется классом.
3. Наследование – создание производных классов
4. Полиморфизм – изменение поведения (новый метод)
5. Инкапсуляция - объединение данных и работающих с ними методов в одном классе
6. Интерфейс - набор публичных методов без реализации
7. Абстрактный класс – могут быть данные и часть методов может быть реализована
8. Сигнатура метода

Принципы создания хорошего кода

В программе все должно быть прекрасно:

и стиль написания кода,

и ее текст,

и структура.

1. Соблюдайте единый Code style

1. соблюдайте переносы фигурных скобок и отступы
2. соблюдайте разрядку — ставьте пробелы там, где они улучшают читабельность кода; особенно это важно в составных условиях, например, условиях цикла
3. соблюдайте правило вертикали — части одного запроса или условия должны находиться на одном отступе

2. Не используйте «магические числа» (Magic numbers - антипаттерн)

1. Используйте именованные константы, чтобы был понятен смысл константы
2. если в ходе работы понадобится сделать расчёт с высокой точностью, придётся искать все вхождения константы в коде (3.14, 3.1415, ...)
3. коллеги-программисты могут не помнить на память значение использованной вами константы — тогда они просто не узнают её в коде

3. Используйте осмысленные имена для переменных, функций, классов

1. Если идентификатор - невнятный набор символов, это не только мешает разработчикам, которые участвуют в проекте, но и приводит к бесконечному количеству комментариев.
2. Переименовав функцию, можно избавиться от комментариев — её имя будет само говорить о том, что она делает.
3. самодокументируемый код — переменные и функции именуются таким образом, что при взгляде на код понятно, как он работает

4. В начале «внешних» методов проверяйте входные данные

1. будущие пользователи могут вводить любые данные, которые могут вызвать сбои в работе программы
2. Во всех вещах нужно соблюдать баланс. Проверка входных данных — обязательное условие только для торчащего наружу кода,

5. Реализуйте при помощи наследования только отношение «является». В остальных случаях – композиция

1. Композиция — паттерн более простой для дальнейшего понимания написанного кода.
2. Можно придерживаться такого правила: выбирать наследование, только если нужный класс схож с классом-предком и не будет использовать методы других классов.

6. Отделяйте интерфейс от реализации

1. Используйте заголовочные файлы и файлы реализации при использовании с.C/C++

7. Делайте методы компактными и / или разделенными на блоки

1. делайте метод компактным так, чтобы один метод выполнял одну задачу
2. отдельные небольшие методы дают в результате хороший код, разделённый на блоки, в которых содержится реализация каждой из функций. Но огромное количество мелких методов усложняет код
3. Длинный код разбивайте на логические блоки, отделенные пустыми строками, в пределах одной процедуры. Не мешайте все в кучу, пожалейте тех, кто будет работать с вашим кодом

8. Не используйте преждевременную оптимизацию

1. **Использовать побитовый сдвиг вместо операций деления и умножения — не лучшая затея**
2. **Мощный оптимизатор компилятора это сделает лучше вас**

ВЫВОД : Код пишется для людей

1. **код пишется в первую очередь для тех, кто будет его сопровождать.**
2. **Сопровождаемость – легкость использования написанного кода, минимизация возможности появления ошибок при его изменении**
3. **Код будет жить долго, если он предусматривает дальнейшее расширение и модификацию**

Принципы SOLID

ООП позволяет программистам комбинировать сущности, объединённые общей целью или функционалом, в отдельных классах.

Принципы SOLID - это стандарт программирования, для создания хорошей архитектуры.

Плохая архитектура -> код получается негибким, даже небольшие изменения в нём могут привести к проблемам.

Принципы SOLID : S

S: Single Responsibility Principle (Принцип единственной ответственности).

Класс должен:

- существовать с единственной целью,
- решать лишь одну задачу,
- ответственен лишь за что-то одно.

Пример: по событию от мыши надо подсчитать стоимость покупки или найти и показать аналоги. Класс обработки событий НЕ выполняет соответствующие действия.

Принципы SOLID : O

O: Open-Closed Principle (Принцип открытости-закрытости).

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации. Если требуется изменить поведение класса, надо создать наследника.

Пример.

Расчет налога на автомобиль: «электромобиль» и «автомобиль с бензиновым двигателем» – это наследники класса / интерфейса «автомобиль».

Принципы SOLID : L

L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).

Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.

Если в коде проверяется тип класса, значит принцип подстановки нарушается.

Пример SOLID : L

```
class Animal {…};
class Horse : public Animal { …};
class Tiger : public Animal {…};
class Cat : public Animal {…};

Animal* zoo[4]  = {new Tiger(),new Tiger(),
                   new Horse(), new Cat()};
for   (int i=0; i < 4; i++)
          zoo[i] -> feed();
```

Принципы SOLID : I

I: Interface Segregation Principle (Принципразделенияинтерфейса).

Клиент не должен зависеть от интерфейсов, которые он не использует. Много специальных интерфейсов лучше, чем один интерфейс общего назначения.

Делим «толстые» интерфейсы на более специфичные, избавляя клиентов от ненужного влияния.

Принципы SOLID : D

D: Dependency Inversion Principle (Принцип инверсии зависимостей).

Модули верхних уровней не должны зависеть от модулей нижних уровней.

Зависимости модулей всех уровней строятся на Абстракциях.

Цель - уменьшение связности совокупности классов программы.

Тема 1. Базовые шаблоны проектирования

Зачем?

1. Наверняка вашу задачу или ее аналог кто-то когда-то решал. **Опыт других разработчиков** надо использовать.
2. Даже если вы отлично спроектировали свое приложение, со временем оно должно меняться, иначе оно умрет. Вносимые изменения должны оказывать **минимальное влияние на существующий код**
3. ООП дает способ построения обобщенной модели в терминах программирования. Объекты программы должны взаимодействовать   на основе **простых и гибких принципов.**

Почему?

В целом паттерны представляют собой некую архитектурную конструкцию, помогающую описать и решить определенную общую задачу проектирования.

Они приобрели такую популярность потому, что программисты понимают, что не стоит изобретать велосипед, а использование паттернов часто бывает полезным как отдельному разработчику, так и целой команде.

Паттерны - основа для обсуждения проекта в группе разработчиков.

Для решения каких проблем разработаны паттерны?

1. Оповещать объекты о наступлении событий, причем объекты могут отказаться в дальнейшем от такого оповещения.
2. Наделить свои или чужие объекты новыми возможностями без модификации кода класса
3. Создавать уникальные объекты, существующие в единственном экземпляре
4. Заставить объекты имитировать интерфейс, которыми они не обладают
5. …

Что такое GoF и GRASP?

«Банда четырёх» в программировании (*Gang of Four*, сокращённо *GoF*) — распространённое название группы четырех авторов (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес), выпустивших книгу Design Patterns

GRASP – это набор принципов проектирования по версии Крэга Лармана - автора книги Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development

GRASP

принципы

Polymorphism (Полиморфизм)
Low Coupling (Низкая связность)
High Cohesion (Высокое зацепление)
Protected Variations (Устойчивый к изменениям)

паттерны

Information Expert (Информационные эксперт)
Creator (Создатель)
Controller (Контроллер)
Pure Fabrication (Чистая выдумка или чистое синтезирование)
Indirection (Посредник)

Тоже известная штука: Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: “**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**”.

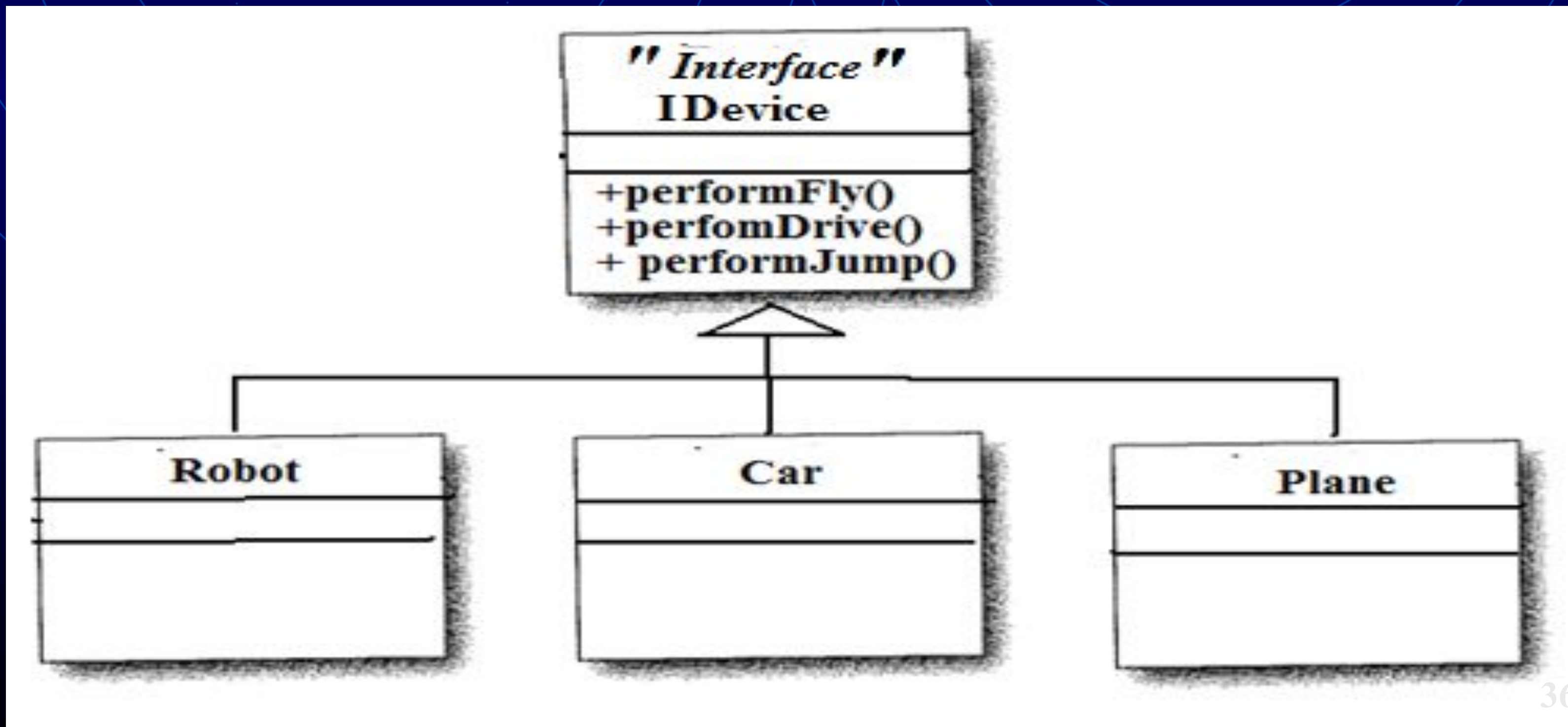
Полиморфизм (Polymorphism)

Полиморфизм позволяет обрабатывать альтернативные варианты поведения на основе типа и заменять подключаемые компоненты системы. Обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса.

Тоже известная штука: Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: “**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**”.

Полиморфизм

Все альтернативные реализации приводятся к общему интерфейсу



UML (Unified Modeling Language) Унифицированный язык моделирования

Язык UML - это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования при разработке программных систем.

Диаграммы

- **Классов** (наследование, ассоциация, агрегация, композиция, …)
- **Взаимодействия** (последовательности, …)
- …

Диаграмма классов

В зависимости от цели выбирается точка зрения, исходя из которой строится диаграмма классов (спецификация, реализация, концептуальная)

Область видимости:

"-" private

"+" public

"#" protected

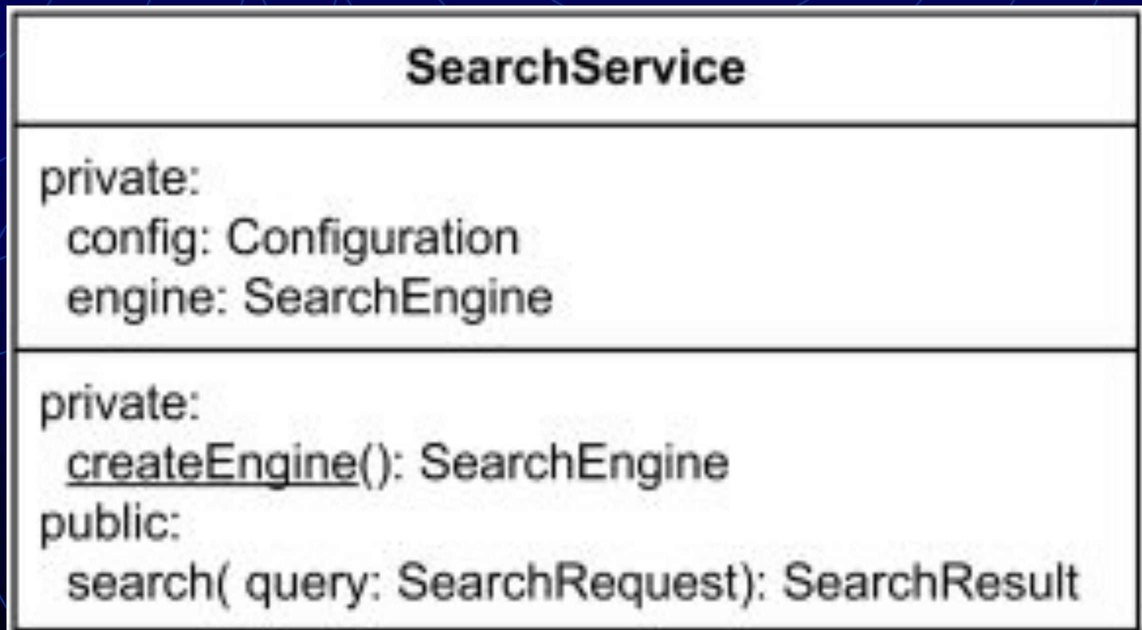
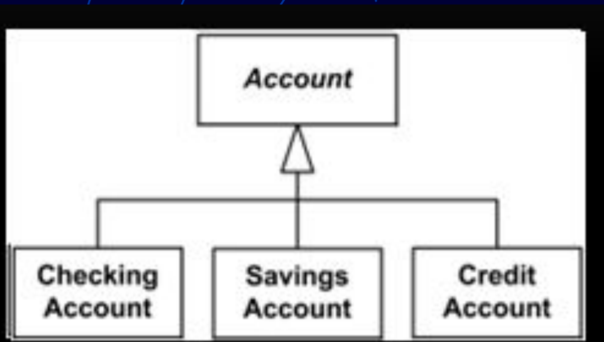
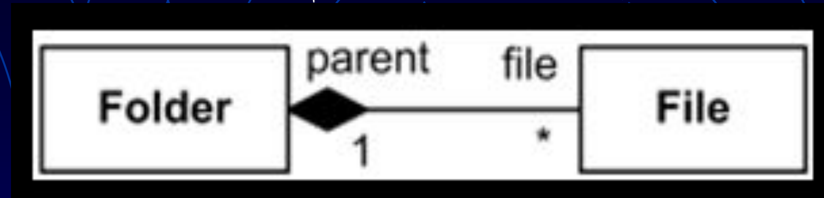


Диаграмма классов - отношения

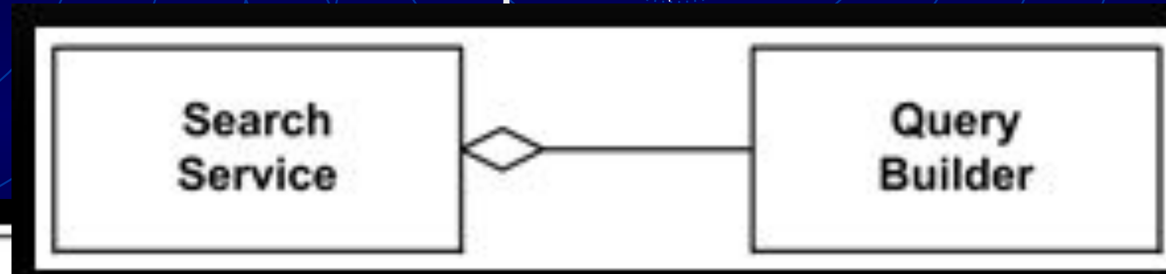
Наследование



Композиция



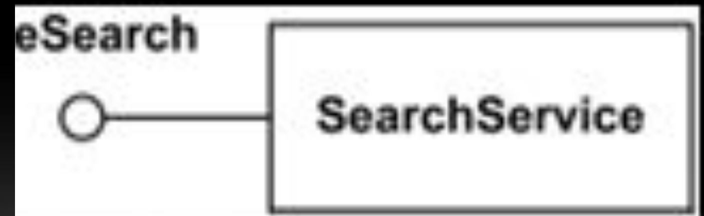
Агрегация



Ассоциация

Зависимость

Реализация интерфейса

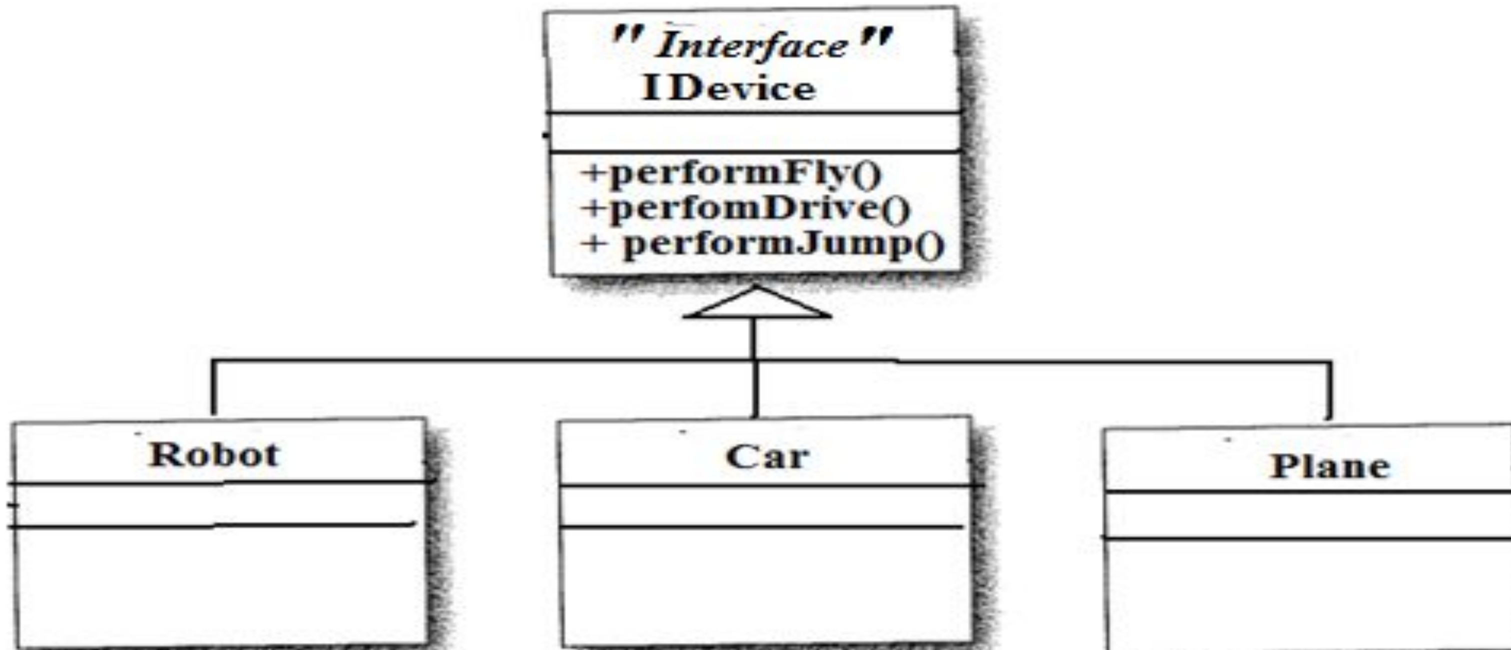


Тоже известная штука: Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: "**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**".

Наследование

```
class  Car : public Idevice  {
 …
};
```

public (+) private (*) protected (-)

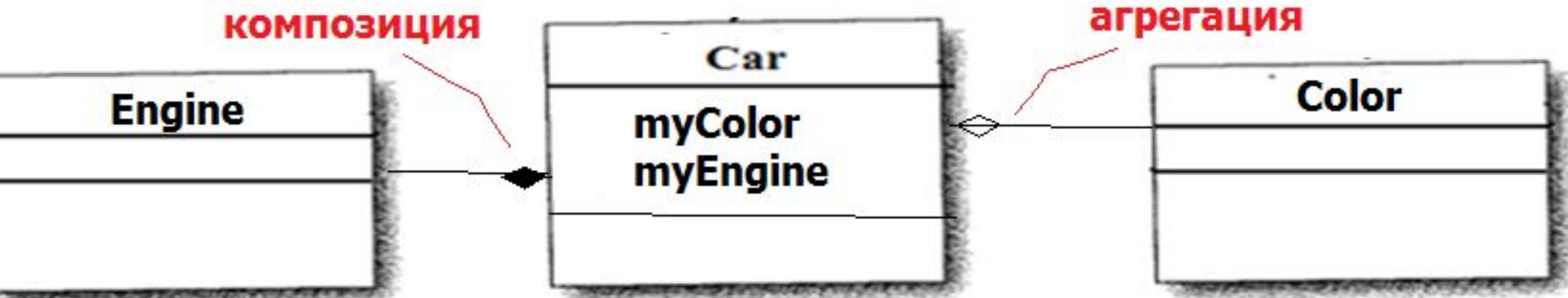


Тоже известная штука. Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: "**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**"

Агрегация и композиция

```
class  Car {
     Color   *   myColor;        // агрегация
     Engine      myEngine;       //  композиция
…
};
```

Удаляем объект класса Car удаляется myEngine
 остается myColor



Переходим к проблемам проектирования и принципам реализации

Как спроектировать объекты, чтобы изменения в объекте не затрагивали других?

Как избежать ситуации, когда при изменении кода объекта придется вносить изменения в множество других объектов системы?

Тоже известная штука: Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: “**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**”.

Низкая связность (Low Coupling)

Если объекты в приложении сильно связанны, то любое изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот поэтому необходимо, чтобы код был слабо связан и зависел от абстракций.

Например, если наш класс Xclass реализует интерфейс IXclass и другие объекты зависят именно от IXclass, т.е. от абстракции, то когда мы захотим внести изменения, касающиеся Xclass – нам нужно будет всего лишь подменить реализацию.

Тоже известная штука: Low Coupling или Слабая связанность. Если объекты в приложении сильно связанны то любой изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает баги. Вот по-этому везде пишут что необходимо чтобы код был слабо связан и зависел от абстракций.
Например если наш класс Sale реализует интерфейс ISale и другие объекты зависят именно от ISale, т.е. от абстракции, то когда мы захотим внести изменения касательно Sale – нам нужно будет всего лишь подменить реализацию.
Low Coupling встречается и в SOLID принципах в виде – Dependency Injection. Сейчас можно часто услышать такой принцип. Но суть остается прежней: "**Программируйте на основе абстракций (интерфейс, абстрактный класс и т.п.), а не реализаций**".

Низкая связность (Low Coupling)

Вывод:

Программируйте на основе абстракций (интерфейс, абстрактный **класс и т.п.), а не реализаций**.

Высокое зацепление (High Cohesion)

High Cohesion или высокое зацепление относится к слабой связанности, они идут в паре и одно всегда приводит к другому.

Класс должен иметь какую-то одну ответственность (Single responsibility principle),

Высокое зацепление -пример

ХОРОШО:

Класс Sale (продажа) - все ответственности, которые касаются продаж (вычисление общей суммы , формирование чека и т.п.)

Класс Payment (платеж). – все ответственности, которые касаются оплаты

ПЛОХО:

Класс SaleAndPayment - одни члены класса, которые касаются Sale, будут между собой достаточно тесно связанны, и также члены класса, которые оперируют с Payment, между собой тесно связаны Но в целом сцепленность класса SaleAndPayment будет низкой, так как мы имеем дело с двумя обособленными частями в одном целом

Высокое зацепление (High Cohesion) - вывод

Программируйте так, чтобы один класс имел единственную зону ответственности, и, следовательно, был сильно сцеплен внутри.

Устойчивый к изменениям (Protected Variations)

Суть данного принципа : определить “точки изменений” и зафиксировать их в абстракции (интерфейсе).

Необходимо определить места в системе, где поведение может изменится и выделить абстракцию, на основе которой будет происходить дальнейшее программирование с использованием этого объекта.

Устойчивый к изменениям (Protected Variations) - вывод

Необходимо обеспечить устойчивость интерфейса.

Если будет много изменений, связанных с объектом, он считается не устойчивым, тогда его нужно выносить в абстракцию, от которой он будет зависеть.

Что такое паттерны проектирования?

Простое определение:

«Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново»
(Кристофер Александр)
2`5

Шаблоны проектирования.

Базовые шаблоны

Delegation и Delegation Event Model

Interface и Abstract Superclass

Proxy или Surrogate

Делегирование (Delegation)

Задача:

Построить игру, в которой есть автомобили. Автомобили умеют передвигаться по земле на колесах.

Расширение:

Добавили самолеты, которые умеют летать и передвигаться по земле на колесах.

Расширение:

Добавим роботов, которые умеют передвигаться по земле по-разному (некоторые на колесах, некоторые на ногах). Роботы летать не умеют, но некоторые из них умеют прыгать

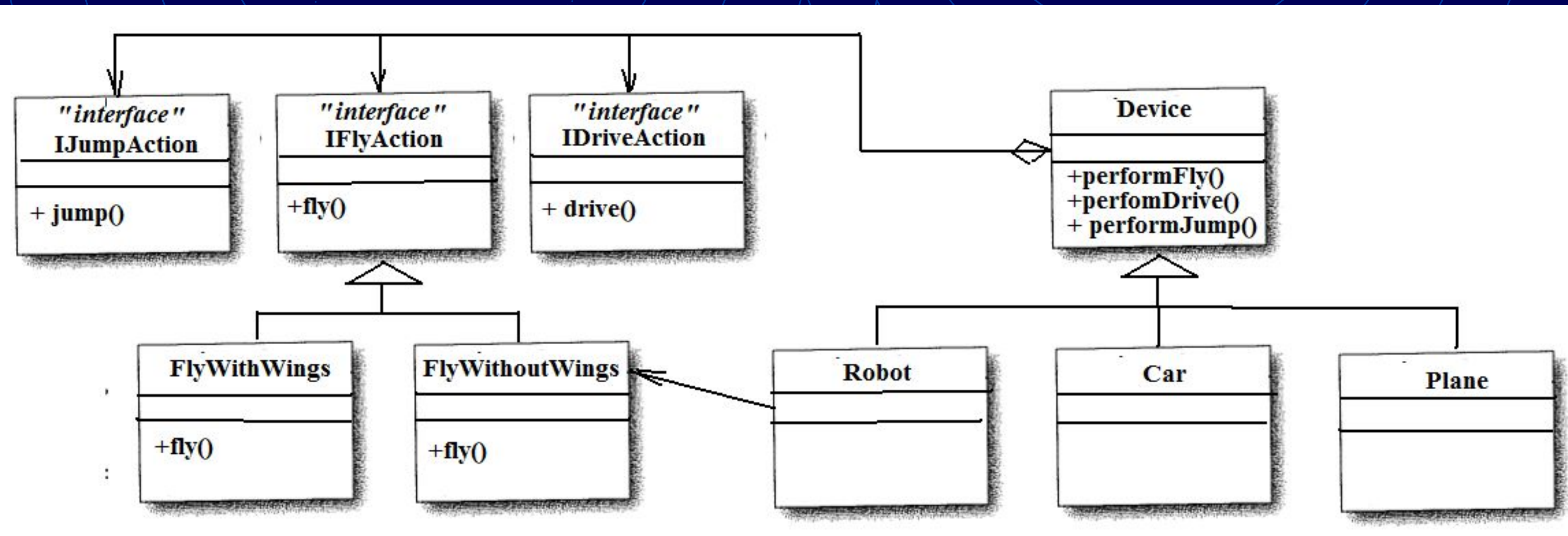
Расширение: …

Делегирование (Delegation)

Наследование как основной принцип создания устройств приводит к огромному количеству разнотипных вариантов.

Выход – делегировать выполнение другому классу.

Делегирование (Delegation) «задача о машинках»



Делегирование (Delegation)

```
// интерфейсы действий
#ifndef __ACTIONS
#define __ACTIONS
class IFlyAction{
public:
	virtual  void   fly() =0; // интерфейс не имеет реализации
};

class IJumpAction{
public:
	virtual  void   jump() =0; // интерфейс не имеет реализации
};

class IDriveAction{
public:
	virtual   void   drive() =0; // интерфейс не имеет реализации
};
#endif
```

Делегирование (Delegation)

```
// классы делегатов
#ifndef __BEHAVIOUR
#define __BEHAVIOUR
#include <stdio.h>
#include <stdlib.h>
#include "actions.h"

//ЛЕТАЕМ
<Классы для  выполнения полетов>
//ПРЫГАЕМ
<классы для выполнения прыжков>
…
#endif
```

Делегирование (Delegation)

```
////ЛЕТАЕМ
class  FlyWithWings : public IFlyAction {
// класс поведения для устройств, которые умеют летать
public:
     void fly(){
          printf ("I am flying!\n");
     }
};

class  FlyWithoutWings : public IFlyAction {
// класс поведения для устройств, которые НЕ умеют летать
public:
     void fly(){
          printf ("I can not fly...\n");
     }
};
```

Делегирование (Delegation)

```
 /// ПРЫГАЕМ
class  JumpWithLegs : public IJumpAction{
        // класс поведения для устройств, которые умеют прыгать
public:
     void jump(){
          printf ("I am jumping!\n");
     }
};

class  JumpWithoutLegs : public IJumpAction{
      // класс поведения для устройств, которые НЕ умеют прыгать
public:
     void jump(){
          printf ("I can not jump...\n");
     }
};
```

Делегирование (Delegation)

```
/// ЕЗДИМ
class  DriveWithWheels : public IDriveAction{
  // класс поведения для устройств на колесах
public:
      void drive(){
            printf ("I can drive with high velocity!\n");
      }
};

class  DriveWithoutWheels : public IDriveAction{
          // класс поведения для устройств, которые НЕ имеют колес
public:
      void drive(){
            printf ("I can drive slowly...\n");
      }
};
```

Делегирование (Delegation)

```
#ifndef __DEVICE
#define __DEVICE

#include "behaviour.h"
#include "actions.h"

class Device{        // абстрактный класс устройства
public:
    IFlyAction    *   flyAction;
    IJumpAction *  jumpAction;
    IDriveAction * driveAction;
    Device(){}
            ~Device();
      // делегируем выполнение операции классам поведения :
    void performFly(){              flyAction -> fly();     }
    void performJump(){             jumpAction -> jump();     }
    void performDrive(){            driveAction -> drive();     }

};
```

Делегирование (Delegation)

```
// конкретный класс «Самолет», который умеет летать и
// ездить

class Plane : public  Device{
public:
    Plane (){
                flyAction = new FlyWithWings();
            driveAction = new DriveWithWheels;
            jumpAction= new JumpWithoutLegs;
    }
};
```

Делегирование (Delegation)

```
// конкретный класс «Автомобиль», который
// умеет ездить

class Car : public  Device{
public:
    Car(){
            flyAction = new FlyWithoutWings;
            driveAction = new DriveWithWheels;
            jumpAction = new JumpWithoutLegs;


    }
};
```

Делегирование (Delegation)

```
// конкретный класс «Робот», который умеет прыгать
// и медленно передвигаться

class Robot : public  Device{
public:
    Robot(){
        flyAction = new FlyWithoutWings;
        driveAction = new DriveWithoutWheels;
        jumpAction  = new JumpWithLegs;
    }

};
```

Делегирование (Delegation)

```
int main(){   // создаем объекты устройств
printf("  Robots\n");
Robot   robot1,   robot2;
robot1.performJump();
robot1.performDrive();
robot1.performFly();
robot2.performJump();
robot2.performDrive();
robot2.performFly();

  // добавим колеса роботу номер 2 :
robot2.driveAction = new DriveWithWheels;  //  утечка  памяти!!
printf("\n\n  Robot 1 after modification\n");
robot1.performDrive();
robot2.performDrive();
```

Результат работы программы

```
c:\MY_PROGRAMMS\pattern_1_2\Debug\pattern_1_2.exe
  Robots
I am jumping!
I can drive slowly...
I can not fly...
I am jumping!
I can drive slowly...
I can not fly...


  Robot 1 after modification
I can drive slowly...
I can drive with high velocity!


  Car
I can not jump...
I can drive with high velocity!
I can not fly...


  Plane
I can not jump...
I can drive with high velocity!
I am flying!


   List of devices
I can drive slowly...
I can drive with high velocity!
I can drive with high velocity!
I can drive with high velocity!


 Petr:
A am swiming!
I play football!
```

'pattern_1_2.exe': Loade

Все устройства выполняют передвижение

```
printf("\n\n   List of devices \n");

Device device[10] = {robot1, robot2, car1, plane1};
for (int index =0; index <4; index ++)
        device[index].performDrive();
```

Результат работы программы

```
c:\MY_PROGRAMMS\pattern_1_2\Debug\pattern_1_2.exe
 Robots
I am jumping!
I can drive slowly...
I can not fly...
I am jumping!
I can drive slowly...
I can not fly...


 Robot 1 after modification
I can drive slowly...
I can drive with high velocity!


 Car
I can not jump...
I can drive with high velocity!
I can not fly...


 Plane
I can not jump...
I can drive with high velocity!
I am flying!


  List of devices
I can drive slowly...
I can drive with high velocity!
I can drive with high velocity!
I can drive with high velocity!


 Petr:
A am swiming!
I play football!
```

Конфигурирование системы

В программе мы создали классы устройств с заранее выбранным типом поведения всех объектов данного класса.
Задача: формировать объекты одного класса с разным поведением.

Основа реализации изменения поведения объектов базируется на конфигурировнии объектов при их создании или динамически в процессе работы.

Делегирование – это инструмент для конфигурирования системы.

Конфигурирование объекта

Как и ранее, создадим разных делегатов :

```
class  DriveFast : public IDriveAction{
   void drive(){// класс поведения для быстрой езды
public:
      cout <<"I can drive with high velocity!" << endl;
   }
};
class  DriveSlow : public IDriveAction{
public:
   void drive(){// класс поведения для медленной езды
      cout << "I can drive, but my velocity is slow " <<
endl;
   }
};
```

Конфигурирование объекта

… и для выполнения полета

```
class  FlySlow : public IFlyAction {
public:
   void fly(){     //  летать с невысокой скоростью
      cout <<"I am flying, but my speed is slow" << endl;
   }
};
class  FlyHiper : public IFlyAction {

public:
   void fly(){       //  гиперзвуковая скорость полета
      cout << "I am flying at supersonic speed" << endl;
   }
};
```

Конфигурирование системы

А в классе самолетов изменим конструктор, который получает на вход список делегатов:

```
class Plane : public  Device{
public:
   Plane (string n, IFlyAction* f, IDriveAction* d){
      flyAction = f;
      driveAction = d;
      setName(n);
      jumpAction= new JumpWithoutLegs;
   }
};
```

При этом остальное поведение объекта не меняется

Конфигурирование всей системы

```
 int main(){   // создаем объекты делегатов
FlyHiper * v1 = new FlyHiper();
FlySlow * v0 = new FlySlow();
DriveSlow * d0 = new DriveSlow();
DriveFast * d1 = new DriveFast();
         // создаем и конфигурируем самолеты
cout << "  airplanes" << endl;
Plane* SU_57_x  = new Plane("SU_57_x", v1, d1),
     * SU_57_y  = new Plane("SU_57_y", v1, d1),
      // два истребителя
    * Ruslan  = new Plane("Ruslan", v0, d1),
       // большой транспортный самолет
     * dron  = new Plane("dron", v0, d0);
```

Конфигурирование всей системы

```
int main(){          // создаем объекты делегатов
FlyHiper * v1 = new FlyHiper();
FlySlow * v0 = new FlySlow();
DriveSlow * d0 = new DriveSlow();
DriveFast * d1 = new DriveFast();
          // создаем и конфигурируем самолеты
cout << "  airplanes" << endl;
Plane* SU_57_x  = new Plane("SU_57_x", v1, d1),
     * SU_57_y  = new Plane("SU_57_y", v1, d1),
       //  два истребителя
    * Ruslan  = new Plane("Ruslan", v0, d1),
       // большой транспортный
     * dron  = new Plane("dron", v0, d0);,
```

Конфигурирование системы

```
Device* device[10] = {SU_57_x, SU_57_y, Ruslan, dron };
```

Конфигурирование системы

```
Device* device[10] =
        {SU_57_x, SU_57_y, Ruslan, dron };

for (int index =0; index < 4; index ++) {
        cout << "start:" << endl;
        device[index]->performDrive();
        cout << "fly:"  << endl;
        device[index]->performFly();
        cout << "plane landing:"  << endl;
        device[index]->performDrive();
        cout << "plane stop:"  << endl << endl;;
  }
```

абота

```
C:\...atterns\01_делегат\DelegatConfig>deleg
  airplanes
start:
SU_57_x I can drive with high velocity!
fly:
SU_57_x I am flying at supersonic speed
plane landing:
SU_57_x I can drive with high velocity!
plane stop:

start:
SU_57_y I can drive with high velocity!
fly:
SU_57_y I am flying at supersonic speed
plane landing:
SU_57_y I can drive with high velocity!
plane stop:

start:
Ruslan I can drive with high velocity!
fly:
Ruslan I am flying, but my speed is slow
plane landing:
Ruslan I can drive with high velocity!
plane stop:

start:
dron I can drive, but my velocity is slow
fly:
dron I am flying, but my speed is slow
plane landing:
dron I can drive, but my velocity is slow
```

Что и когда делать?

А) Нужны

- типы классов с разным поведением,
- объекты одного класса имеют одинаковое поведение

 делегат статически определяется в классе-наследнике

Б) Нужны

- объекты одного класса с разным поведением

 динамически при создании конфигурируем объект

В) Нужны объекты, меняющие свое поведение в процессе работы

 Динамически меняем делегата

Вариант реализации:

создать в классе Device указатели на обработчики с пустым поведением, чтобы при создании устройства подписывать его только на выполняемые им действия

```
IFlyAction    *   flyAction = new emptyFlyAction();
IJumpAction *  jumpAction = new emptyJumpAction();
IDriveAction * driveAction = new emptyDrive Action();

class emptyDriveAction : public IDriveAction{
  // класс  «пустого» поведения
public:
    void drive(){};
};
```

Задача о множестве действий у одного объекта

Задача:

Есть несколько видов спорта. Надо построить класс спортсмена, который занимается определенным видом спорта.

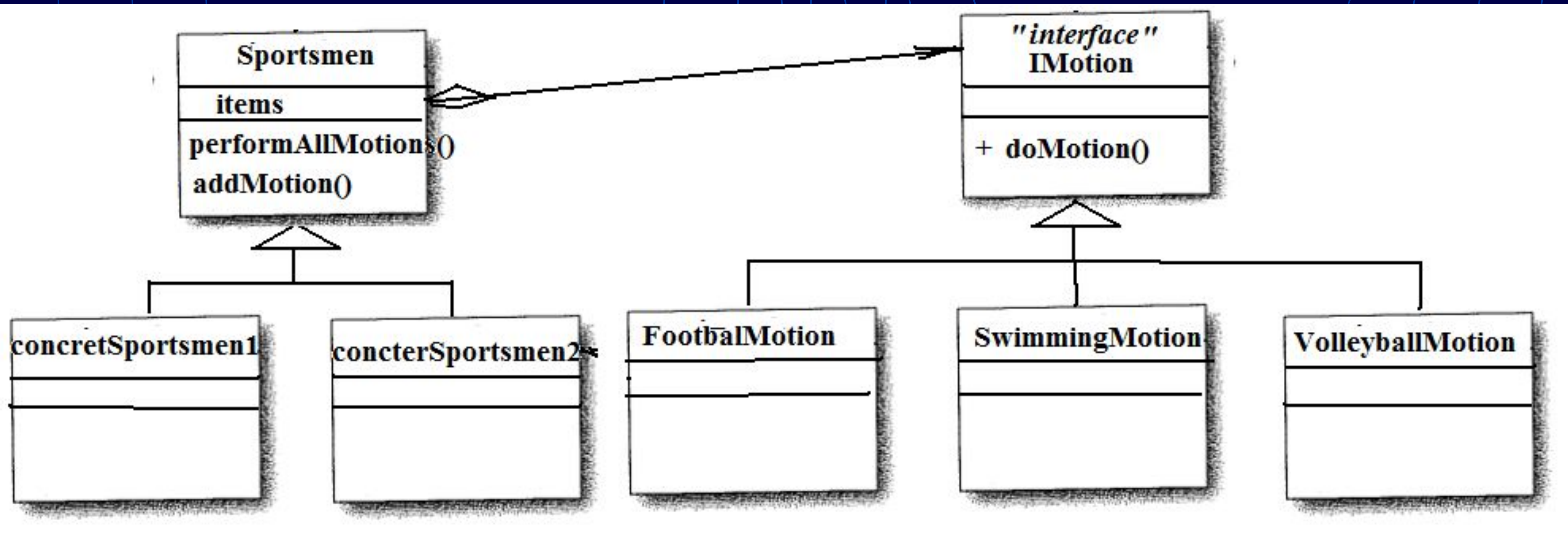
Расширение:

Можем добавить новые виды спорта.

Расширение:

Один спортсмен может заниматься разными видами спорта.

Задача о множестве действий у одного объекта



Список делегатов у объекта

```
// интерфейсы действий
#ifndef __MOTION
#define __MOTION

class IMotion {   //  интерфейс
public:
    virtual void doMotion() = 0;
};

 //  здесь конкретные делегаты
…
#endif
```

Список делегатов у объекта

```
// конкретные делегаты:

class SwimmingMotion : public  IMotion {
public:
    void doMotion(){printf("A am swiming! \n");}
};

class FootballMotion : public  IMotion {
public:
    void doMotion(){printf("I play football! \n");}
};

class VolleyballMotion : public  IMotion {
public:
    void doMotion(){printf("I play volleyball! \n");}
};
```

Подписка

```
typedef   IMotion * ptrMotion;
   //  класс спортсмен
class Sportsmen{
private:
    vector <ptrMotion> items;
public:
    void performAllMotions();
    void addMotion(Motion *newMotion);
    Sportsmen(){    items.clear();          }
          ~Sportsmen();
};
```

Подписка

```
void perfomAllMotions(){
    for (vector<ptrMotion>::iterator
            it = items.begin();
            it != items.end();  it++) {
                (*it)->doMotion();
    }
}

void addMotion(Motion *newMotion){
        items.push_back(newMotion);
}
```

Подписка

```
Sportsmen * Petr = new Sportsmen();
Sportsmen * Vera = new Sportsmen();
SwimmingMotion *typeSwim = new SwimmingMotion;
FootballMotion *typeFoot = new FootballMotion;
VolleyballMotion *typeVoll = new VolleyballMotion;

printf("\n\n Petr:\n");
Petr->addMotion(typeSwim);
Petr->addMotion(typeFoot);
Petr->performAllMotions();

printf("\n\n Vera:\n");
Vera->addMotion(typeSwim);
Vera->addMotion(typeVoll);
Vera->performAllMotions();
```

Результат работы программы

c:\MY_PROGRAMMS\pattern_1_2\Debug\pattern_1_2.exe

```
   Robots
I am jumping!
I can drive slowly...
I can not fly...
I am jumping!
I can drive slowly...
I can not fly...


   Robot 1 afte modofication
I can drive slowly...
I can drive with high velocity!


   Car
I can not jump...
I can drive with high velocity!
I can not fly...


   Plane
I can not jump...
I can drive with high velocity!
I am flying!


    List of devices
I can drive slowly...
I can drive with high velocity!
I can drive with high velocity!
I can drive with high velocity!


  Petr:
A am swiming!
I play football!


  Vera:
A am swiming!
I play volleyball!
```

Proxy – заместитель
или
Surrogate - суррогат

Заместитель – суррогат настоящего объекта.

Заместитель прикидывается настоящим объектом, а на самом деле или взаимодействует с ним или просто работает «по умолчанию».

Proxy - заместитель

Типы заместителей:

1 – удаленный заместитель. При сетевой реализации заместитель действует как представитель удаленного объекта.

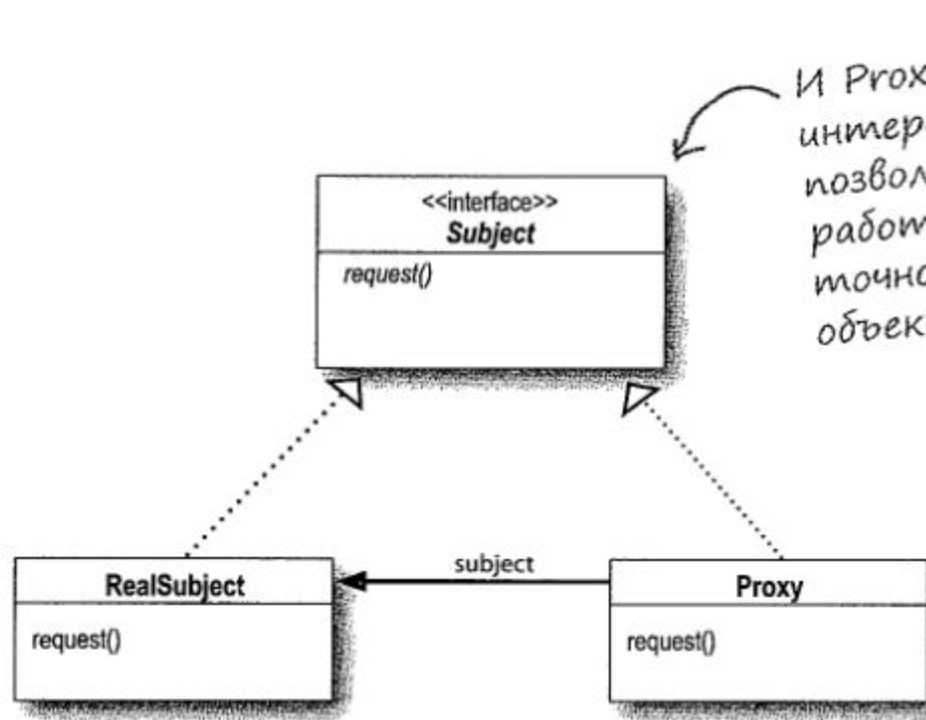
2 - виртуальный заместитель. Управляет доступом к ресурсу, создание которого требует больших затрат. Заместитель создает объект только тогда, когда это необходимо

3 – защитный заместитель. Контролирует доступ к ресурсу в соответствии с системой привилегий

4 – фильтрующий заместитель. Управляет доступом к группам ресурсов

5 – синхронизирующий заместитель. Обеспечивает безопасный доступ из нескольких потоков к объекту

Proxy - заместитель



И Proxy, и RealSubject реализуют интерфейс Subject. Это позволяет любому клиенту работать с заместителем Proxy точно так же, как с реальным объектом RealSubject.

RealSubject — объект, выполняющий фактическую работу; заместитель управляет доступом к нему.

Заместитель часто создает экземпляры или управляет созданием RealSubject.

Proxy хранит ссылку на Subject, чтобы передавать запросы Subject по мере надобности.

Proxy - пример

```
class Math {

  // класс, для которого создадим Proxy

public:
    virtual void sum()=0;
    virtual void sub()=0;
    virtual void mult()=0;
    virtual void div()=0;
};
```

Proxy - пример

```
class M1 :        public Math  {
//  настоящий класс для обработки данных
public:
   int a,b;
   virtual void sum() {        cout << "Sum: " << a+b << endl;     }
   virtual void sub()  {        cout << "Sub: " << a-b << endl;        }
   virtual void mult() {        cout << "Mult: " << a*b << endl;    }
   virtual void div() {
      if( b == 0) {            cout << "Div by zero!\n";
      } else {
         cout << "Div: " << a*b << endl;
      }
   }
   M1(int inA, int inB)     {        a = inA;        b = inB;     }
};
```

Proxy - пример

```
class ProxyM1 :        public Math  {
private:
    M1 *prox;
    void log() { cout << "a=" << prox->a << ", b=" << prox->b << endl; }

public:
    virtual void sum() {        log();        prox->sum();    }
    virtual void sub()  {       log();        prox->sub();    }
    virtual void mult() {       log();        prox->mult();   }
    virtual void div()   {       cout << "No div!" << endl;     }

    ProxyM1(int inA, int inB) {
        prox = new M1(inA,inB);
                // здесь Proxy создает реальный объект M1
    }
    ~ProxyM1() {       delete prox;    }
};
```

Proxy - пример

```
int main(){
    Math *t = new M1(6,0);
    Math *p = new ProxyM1(6,0);
    cout << "M1\n";
    t->sum();
    t->sub();
    t->mult();
    t->div();
    cout << "\nProxyM1\n";
    p->sum();
    p->sub();
    p->mult();
    p->div();
    delete p;
    delete t;
    return 0;
}
```

Proxy – работа

```
C:\LECTURES\Patterns>main.exe
M1
Sum: 6
Sub: 6
Mult: 0
Div by zero!

ProxyM1
a=6, b=0
Sum: 6
a=6, b=0
Sub: 6
a=6, b=0
Mult: 0
No div!

C:\LECTURES\Patterns>_
1Help  2UserMn  3View  4Edit  5Copy  6R
```

1.

Задание на лабораторную работу № 2 и №3

1. Рассмотреть задачу в неформальной постановке
2. Перечислить список объектов, у каждого из них указать свойства и выполняемые функции
3. Для некоторых объектов перечислить возможные расширения свойств и функционала
4. Ввести новые возможности в систему (новые свойства, действия , объекты, взаимодействия объектов и т.д.)
5. Сформировать предварительный перечень классов и их обязанности (первая итерация)

Задание на лабораторную работу № 4

1. Рассмотреть задачу в начальной постановке
2. Предложить реализацию различного поведения объектов на основе делегирования
3. Сформировать перечень интерфейсов
4. Сформировать перечень классов и их обязанности
5. Построить диаграмму классов
6. Проверить выполнение принципов низкой связности и высокого зацепления.
7. Реализовать систему
8. Реализовать Proxy для работы с некоторым объектом и для контроля доступа к объекту

Пример задания

Прикладная область для выполнения лабораторных работ: «Интерактивные головоломки для детей»,

Тема: Логическая задача «Волк, Коза, Капуста».

Лодочник должен перевести на лодке с одного берега на другой три объекта: волка, козу и капусту. В лодке, кроме самого лодочника, может поместиться только один перевозимый объект, поэтому лодочнику придется совершить несколько рейсов. Но, если он оставит без присмотра на одном берегу волка и козу, то волк съест козу. А если вместе на берегу окажутся коза и капуста, то коза съест капусту Задача: как лодочник должен перевезти в целости и сохранности все три объекта с одного берега на другой?

Пример: базовые объекты

1. Лодка: умеет загружать и выгружать пассажиров, перемещаться в заданном направлении. Лодка: может иметь разный двигатель.
2. Перевозимые объекты: требуют специальных условий для перевозки, так как могут обладать свойством опасности для других объектов.
3. Лодочник: выбирает стратегию перевозки объектов, обеспечивает контроль безопасности объектов, дает команду объектам переместиться в лодку или из лодки, управляет лодкой. Лодочник может иметь множество стратегий для выбора перевозимых объектов.

Пример: возможные расширения объектов

1. Баба Яга: следит за оставшимися без надзора объектами и планирует их похищение.
2. Ступа - транспорт Бабы Яги - умеет выполнять различные типы перемещения, при этом может маскироваться под лодку.

Пример: возможные расширения функционала

1. Лодка: может иметь разный двигатель (от резиновой моторки до атомной подлодки)
2. Перевозимые объекты могут обладать ядовитостью, радиоактивностью и т.д, то есть возможен набор вредоносностей
3. Лодочник : может иметь множество стратегий для безопасного выбора перевозимых объектов

Пример: перечень классов, интерфейсов, объектов

1. Класс "перевозимый объект" IPassenger
2. Конкретные объекты Волк, Коза, Капуста, ...
3. Класс "вредоносность" IDanger
4. Классы-наследники конкретных вредоносностей: Убивать, Съедать, Заражать, ...
5. Класс "набор вредоносностей" - IComposite
6. Наборы вредоносностей для конкретных объектов класса IPassenger
7. Класс "транспортное средство ITransport
8. Классы-наследники: Плавающие, Летающие, ...
9. Конкретные объекты: Лодка, Ступа, …

Пример: перечень классов, интерфейсов, объектов

1. Класс "мотор транспортного средства" - IEngine
2. Классы-нвследники: ручная тяга, двигатель, ...
3. Конкретные объекты: Весло, Мотор, Атомный двигатель...
4. Класс "водитель транспортного средства" IDriver
5. Конкретные объекты: Лодочник, Капитан, ...
6. Класс "стратегия решения задачи" IStrategy
7. Классы-наследники: Простая стратегия № 1, ...
8. Класс "состояние игры" IState IState
9. Классы-наследники конкретных состояний: состояние-1, состояние-2, ...
10. Класс "внешние силы - наблюдатели" IObserver
11. Конкретные объекты: Баба Яга, …

Пример: отношения между классами

1. ITransport --- композиция ----> IEngine
2. IDriver --- композиция ----> ITransport
3. IState --- агрегация ----> IStrategy
4. IDanger --- агрегация ----> IPassenger
5. IObserver --- агрегация ----> IPassenger
6. IComposite --- наследование ----> IDanger
7. IPassenger --- агрегация ----> Icomposite
8. …

Пример: простая реализация лабораторной работы №4 (делегирование и proxy)

Для выполнения задания по теме "Делегирование" реализуем простейший вариант системы, оставим в системе только следующие три перевозимых объекта, при этом клиент (функция main) самостоятельно по фиксированной стратегии задает порядок перевозимых объектов.

Пример: простая реализация - делегирование

1) IDriver делегирует ITransport
- действие "загрузить транспортное средство"
- действие "выполнить перевозку в пункт назначения"

2) ITransport делегирует IEngine
- действие "выполнить маршрут«

Продемонстрируем работоспособность системы для различных используемых транспортных средств с различными моторами.

Пример: простая реализация proxy

Введем в систему защитного заместителя Proxy для контроля доступности и безопасности нового состояния типа IState в соответствии с

системой вредоносностей объектов, находящихся в одном месте.

Примеры задач

1. Система управления и мониторинга грузоперевозками
2. Система бронирования билетов на театрально-зрелищные представления
3. Система управления режимом в инкубаторе
4. Система управления температурным режимом в автоматизированной теплице
5. Система управления кафе-автоматом
6. Система управления автоматом по продаже бутербродов
7. Система управления заводом-автоматом «кондитерская фабрика»
8. Система управления роботом-луноходом

9 Игра (по выбору)

10. Автоматическая система управления аэропортом
11. Умный дом
12. Система управления кинотеатром-автоматом

Замечания к реализации
(решение проблемы с параметрами разных типов)

Замечания к реализации

1. Иногда возникает необходимость реализовать класс-делегат, который содержит несколько функций с разным списком параметров.
2. Такая же проблема возникает при необходимости построить несколько разных наследников одного интерфейса делегатов, но при этом выполняемые функции у каждого наследника имеют разные типы параметров

Выход есть – создать класс параметров, физические данные которого обернуты в некоторый контейнер (например, структуру или класс). Рассмотрим пример с реализацией данных в форме структуры.

Физические данные

```
// эти  данные будут в качестве параметров команд
// собрали  данные в структуры

  struct str_1{  // какие-то данные
     int   a,b,c;      double d;
};
struct str_2{  // еще какие-то другие данные
  double   a,b,x,y;
};

// При необходимости добавим еще структуры,
//  что  не  повлияет  на  уже   готовый  код
```

Абстрактный класс параметров

```
class param{
public:
   void * data;
   virtual void *getParam() = 0;
  virtual  void  setParam(void *) = 0;
  …
};
```

При необходимости все функции для работы с параметрами можно дополнить.

Главное здесь - ***тип данных void****

Конкретные параметры - concrParam_1

Создадим параметры, содержащие данные типа str_1

```
class concrParam_1 : public param{  // параметры - str_1
public:
   concrParam_1(str_1 * z) {
      str_1 * y  = new str_1;
      data = (void *) y;
      y->a = z->a;
      y->b = z->b;
   }
   virtual void * getParam() {
      return (void *) (&data);
   }
};
```

Конкретные параметры - concrParam_2

Аналогично создадим другой класс параметров

```
class concrParam_2 : public param{  // параметры - str_2
public:
concrParam_2(str_2 * z) {
     str_2 * y  = new str_2;
     data = (void *) y;
     y->a = z->a;
     y->b = z->b;
   }
  virtual void * getParam() {
     return (void *) (data);
  }
};
```

Абстрактный класс роботов

```
// Теперь можем создавать любые  классы, функции которых
// имеют  параметры  созданного  абстрактного  класса
// Абстрактный класс роботов выполняет команды,
 // работая  с параметрами  абстрактного  класса
//  doWork   и doWork _2

class robot{
public:
  virtual void doWork(param*p) = 0;
  virtual void doWork_2(param*p) = 0;
  robot (){}
};

//   параметры  выглядят  одинаково,  но  будут   разными
```

Конкретный робот

```
class robot_1 : public robot {  //  У него параметры
    concrParam_1
public:
  void doWork(param * p) {
    concrParam_1 * myParam = (concrParam_1 *) p ;
    str_1   * y = (str_1 *) p -> data;
    cout << y->a << endl;
  }
  void doWork_2 (param * p) {
    concrParam_2 * myParam  = (concrParam_2 *) p ;
    str_2  * y = (str_2 *) p -> data;
    cout << y->a << endl;
  }
};
```

Теперь нет проблем с вызовом функций

```
int main(){
   robot * r1 = new robot_1();

 // функции с первым типом параметров
   str_1  z1; z1.a = 1000 ;  z1.b = 1000 ;
   str_1  z2; z2.a = 1 ;    z2.b = 2;
     param * a = new concrParam_1(&z1),
             * b= new concrParam_1(&z2);
   r1-> doWork(a);
   r1-> doWork(b);
  …
   …
```

Теперь нет проблем с вызовом функций

```
   …
// функции с другим типом параметров
   str_2  w1; w1.a = 900.77 ;  w1.b = 10.9900 ;
   str_2  w2; w2.a = 1.22 ;    w2.b = 2.19;
   param * a2 = new concrParam_2(&w1),
              * b2 = new concrParam_2(&w2);
   r1-> doWork_2(a2);
   r1-> doWork_2(b2);
return 0;
}
```

Все работает!

У первой функции первый параметр имел тип str_1, функция выводила его первое поле типа int. У второй параметр имел тип str_2, функция выводила его первое поле типа double.

Работа с функциями с разными типами параметров:

```
C:\...ES\Pattern
1000
1
900.77
1.22
```

Выводы

1. Создаем конструкции из параметров
2. Используем абстрактный класс параметров с указателем на данные типа void *
3. Создаем конкретные параметры с соответствующими данными типа (1)
4. В классах с выполняющимися функциями описываем функции с параметрами типа (2)
5. Каждая функция знает свой тип параметров и приводит указатель на параметр типа (2) к своему типу, а затем работает с параметрами соответствующего типа