0. Содержание

Monday, June 4, 2018 15:20

36. В чем отличие синтаксической и логической константности методов?
37. Для чего используется ключевое слово mutable?
38. Что такое конструктор?
39. В каких случаях используется перегрузка конструкторов?
40. Какую цель может преследовать создание приватного конструктора?
41. Каким образом и в какой последовательности происходит инициализация полей объекта?
42. Для чего используется ключевое слово explicit?
43. В чем заключается предназначение конструктора по умолчанию?
44. Что такое деструктор, для чего он используется?
45. Каков порядок вызова деструкторов при разрушении объекта?
46. В какой момент вызывается деструктор объекта?
47. Каково время жизни объекта?
48. Зачем нужен виртуальный деструктор?
49. Как осуществляется работа с динамической памятью в C/C++?
50. В чем различие delete и delete[]?
51. Что подразумевается под идиомой RAII?
52. Перечислите основные подходы к обработке ошибок.
53. Для чего предназначен механизм обработки исключительных ситуаций?
54. Что такое исключение?
55. Какие типы данных допустимы для использования в качестве объектов exception?
56. Как происходит возбуждение исключения?
57. Кто отвечает за обработку возникших исключительных ситуаций?
58. Что такое раскрутка стека?
59. Где и для чего используется спецификатор throw?
60. Где и для чего используется спецификатор noexcept?
61. К чему приводит вызов throw без аргументов?
62. Что такое exception-safe операция?
63. Что такое делегирующие конструкторы?
64. Что вы можете сказать о генерации исключений в конструкторе/деструкторе?
65. Что такое ассоциация?
66. Что такое композиция и агрегация, чем они отличаются?
67. Время жизни агрегируемого объекта меньше времени жизни агрегата?
68. Какие классы называются дружественными, для каких целей используется это отношение?
69. В каком случае можно говорить об отношение «реализация»?
70. Как представлены объекты в памяти при использовании механизма наследования?
71. Какие существуют типы наследования, чем они различаются?
72. Наследуются ли конструкторы и деструкторы?
73. Наследуются ли приватные поля базового класса?

74. Что такое виртуальная функция?
75. Как осуществить вызов базовой реализации функции при её переопределении в дочернем классе?
76. Как связаны виртуальные функции и полиморфизм?
77. Что такое переопределение функций?
78. Работает ли переопределение для приватных функций?
79. Что такое таблица виртуальных функций?
80. Как себя ведут виртуальные функции в конструкторе и деструкторе?
81. В каких случаях допустимо приведение указателей/ссылок на дочерний класс к базовому?
82. Что такое чистая виртуальная функция?
83. Какой класс называется абстрактным?
84. Как в C++ реализуются интерфейсы?
85. Что такое перегрузка функций?
86. Как ведет себя перегрузка при наследовании?
87. Опишите процесс выбора функции среди перегруженных.
88. Чем отличаются механизмы раннего и позднего связывания?
89. Что такое множественное наследование?
90. Что такое ромбовидное наследование?
91. Какой существует механизм разрешения проблемы ромбовидного наследования в C++?
92. Как реализовано приведение типов в Си?
93. Что такое статическое приведение типов?
94. Что такое динамическое приведение типов?
95. Что такое константное приведение типов?
96. Что такое интерпретирующее преобразование типов?
97. Как работает преобразование в Си-стиле на языке C++?
98. Что такое умные указатели?
99. Опишите принцип работы boost::scoped_ptr.
100. Опишите принцип работы std::auto_ptr.
101. Опишите принцип работы std::shared_ptr.
102. Опишите принцип работы std::weak_ptr.
103. В чем особенности работы умных указателей с массивами?
104. Какие группы операторов в C++ вам известны?
105. Что такое перегрузка операторов, для чего она используется?
106. Для каких типов допустима перегрузка операторов?
107. Где может быть объявлена перегрузка оператора?
108. Какие особенности у перегрузки операторов инкремента и декремента?
109. Как ведут себя операторы с особым порядком вычисления при перегрузке?
110. Наследует ли производный класс перегруженные операторы? Да
111. Как защитить объект от копирования?
112. Для чего предназначен механизм RTTI, как его использовать?
113. Что такое шаблоны классов?

114. Что такое шаблоны функций?
115. Как осуществляется вывод аргументов шаблона?
116. Что такое специализация шаблонов?
117. Что такое шаблон проектирования?
118. Factory Method
119. Abstract Factory
120. Builder
121. Prototype
122. Singleton
123. Adapter
124. Bridge
125. Composite
126. Decorator
127. Façade
128. Flyweight
129. Proxy
130. Chain of Responsibility
131. Command
132. Iterator
133. Mediator
134. Memento
135. Observer
136. State
137. Strategy
138. Template method
139. Visitor

I. Введение в C и C++

18 февраля 2018 г. 21:33

Введение в C и C++

Парадигма - способ концептуализации
Концептуализация - поиск признаков, по которым возможно отличить одно от другого
Парадигма программирования - способ разделения подходов к написанию программ. Парадигма не определяется однозначно языком программирования.

ООП - на данный момент самая распространённая парадигма. Это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

В C++ нет сообщений как таковых, отправка сообщения - вызов метода. Поэтому в C++ нельзя "послать" сообщение, если получатель не может ответить.

Классы - типы данных, определяемые пользователем. Это объекты в C++.

Архитектура фон Неймана

- Двоичное кодирование
- Однородность памяти - в памяти лежат как данные, так и программы
- Адресность - всю память можно представить в виде нумерованных ячеек
- Программное управление - программа состоит из ограниченного набора команд. Набор зависит от архитектуры процесса.

Парадигмы программирования

Императивные - программа набор инструкций, данных к исполнению. Может выполняться непосредственно ЦП.
Пример - Ассемблер, C

- **Процедурный** стиль - программа разбивается на модули. Модули переиспользуются, и это облегчает изменение программы и уменьшает размер программы.
- **Структурное программирование** - основано на теории, что с помощью циклов и условий можно разработать любой алгоритм. Это уменьшает количество безусловных переходов.
- **Объектно-ориентированное программирование** - программа разбивается на объекты, хранящиеся в классах.

Декларативные - программа получает на вход не "как", а "что" делать. Для выполнения нужен посредник

- **Функциональный** стиль
- **Логическое программирование**

Пример - SQL

Основные принципы ООП

- **Абстракция** - позволяет очертить набор характеристик для каждой сущности, отсекая ненужные в рамках предметной области
- **Иерархическая классификация** - программирование от целого к частному. Программирование идёт по "слоям" - слои взаимодействуют через интерфейс
- **Инкапсуляция**
  - Связывает код и данные. В классе описываются и данные, и методы для работы с ними
  - Защита от внешнего воздействия. Есть модификаторы доступа, определяющие, кто может получить доступ к данным.
  - Выделение интерфейса - разделение методов на публичные (интерфейс) и приватные
  - Сокрытие деталей реализации
  - В C++ основа инкапсуляции - класс
- **Наследование** - механизм, с помощью которого один объект приобретает свойство другого
  - Наследуются атрибуты
  - Наследование возможно, если один объект повторяет свойства другого
  - Основной смысл - наследники должны быть уникальными
  - Уменьшает дублирование кода
- **Полиморфизм** - множество реализаций для одного интерфейса. Механизм, позволяющий скрыть за интерфейсом класс действий
  - **Статический** - если несколько методов с одинаковым именем, из которых выбирается подходящий. Это перегрузка функций, методов, операторов.
  - **Динамический** - определяется во время выполнения. Относится к виртуальным методам и функциям.
  - **Параметрический** - относится к обработке значений разных типов идентичным образом. "Противоположен" статическому полиморфизму.

Язык С - язык низкоуровневой разработки.
Особенности:

- Эффективность - работа напрямую с железом
- Стандартизированность
- Высокая обратная совместимость

- Относительная простота
- Доступность на большинстве платформ

Язык С реализует императивную и функциональную парадигмы, но возможна и реализация ООП.

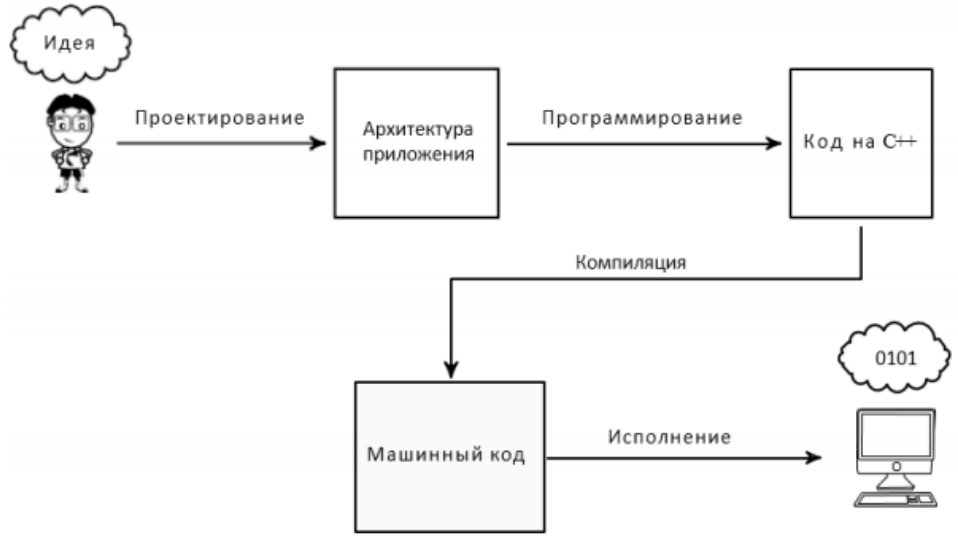
Язык C++

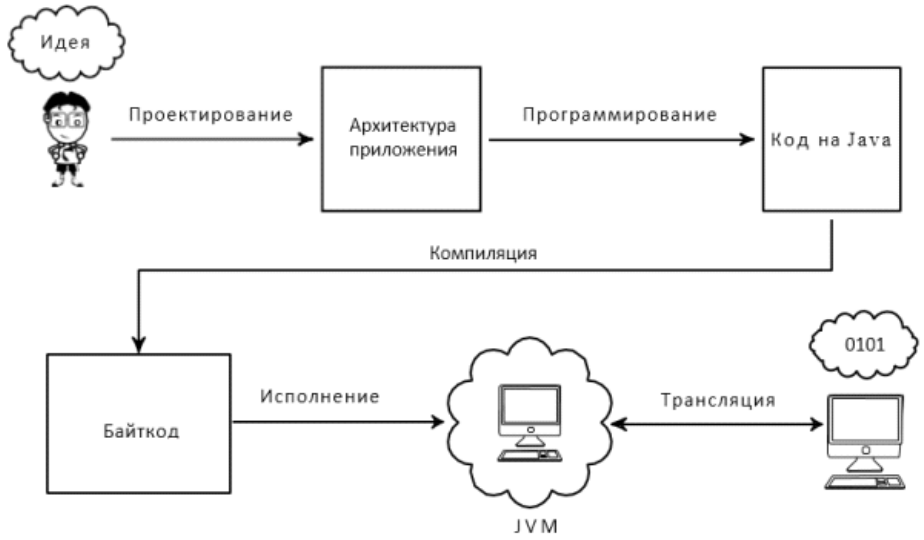
- Изначально транслировался в С
- Используется при разработке систем реального времени
- Код после компилятора сопоставим с С по производительности, но допускает более высокий уровень абстракции
- С++ унаследовал синтаксис С

Язык С++ - **мультипарадигмен**. Можно писать в процедурном, структурном, объектно-ориентированном, функциональном стиле.

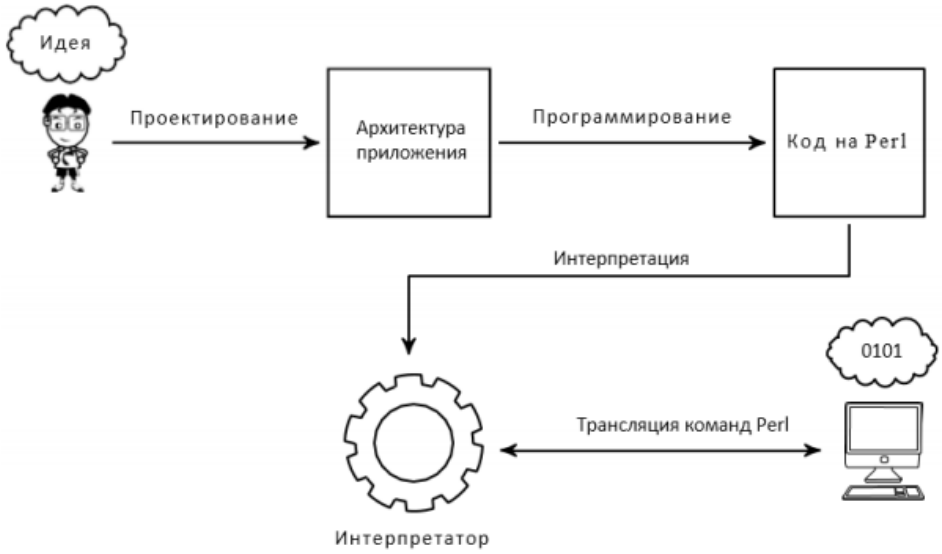
С++ сочетает в себе свойства как высокоуровневых (ООП, , так и низкоуровневых языков.

Так как у С++ под каждую платформу требуется свой компилятор, то у С++ отсутствуют неявный накладные расходы. Например, в Java компиляция происходит не в машинный код, исполняемый на ЦП, а в байт-код, исполняемый JVM.





Интерпретация - выполнение кода шаг за шагом, построчно. Пример - Perl, Python



Компиляция - трансляция всего текста программы в машинный код или в байт-код.

Разбиение проекта на файлы

- Удобнее работать
- Структурирование кода
- Проще командная работа
- Частичная рекомпиляция

Структура проекта:

- src - исходные коды .cpp
- include - файлы с прототипами .h или .hpp

Статическая и динамическая типизация

Статическая типизация - каждая сущность имеет свой тип, определяемый на этапе компиляции. Такой подход позволяет заранее предусмотреть размер переменной и определить вызываемые в конкретных местах функции

Динамическая типизация - каждая переменная связывается с типом в момент присваивания значения, а не в момент объявления

Сильная и слабая типизация

Сильная типизация (strong typing) - строгие правила компиляции, не допускающие неявного приведения типов и т.п.

Слабая типизация (weak typing) - компилятор будет выполнять всевозможные неявные преобразования, даже если это приведёт к потере точности.

II. Принципы выполнения программ C++

19 февраля 2018 г. 13:52

Принципы выполнения программ C++

Все современные архитектуры строятся на основе архитектуры фон Неймана.

Сегментация памяти - память разделена на условные блоки - кода и данных.
В современных ОС сегменты кода защищены от записи, в отличие от сегментов данных. Это сделано для того, чтобы осложнить выполнение вредоносного кода на компьютере.
При запуске выделяются два типа сегментов данных:

- Сегмент глобальных данных - хранит глобальные переменные
- Стек - хранит локальные переменные

В случае нехватки можно освобождать предыдущие сегменты или запрашивать новые. Если адрес сегмента не выделен, обращение к нему пресекаются.

Представление кода в памяти. В процессе компиляции каждая функция преобразуется в набор машинных инструкций. Адрес начала данного сегмента кода равен адресу функции, т.е. фактически адрес функции - адрес первого операнда. После компиляции программы, почти вся информация о типах стирается.

Ход выполнения программы. Адрес следующей инструкции хранится в регистре IP. Процессор считывает команду по адресу, выполняет его и сдвигает IP. Есть функции, которые меняют IP принудительно (условный, безусловный переход). Так, вызов функции меняет IP.
При компиляции на место оператора вызова функции подставляется только её имя. Зависимости разрешает линковщик. Если линковщик не нашел нужную функцию, то он выдаст ошибку (undefined reference)

Чтобы избежать повторного объявления функции при включениях, можно использовать конструкцию с IFNDEF или #pragma once

Стек вызовов
Отличие обычного стека от стека вызовов в том, что возможно обращение к любой ячейке стека, а не только к верхней. Обычно стек вызовов небольшой(32КБ - 4МБ), но его минимальный размер можно определить при компиляции.
В стеке вызовов хранятся локальные (объявленные внутри функции) и временные (переменные, хранящие промежуточные результаты)

переменные, входные параметры.
Когда функция завершается, стек-фрейм освобождается. Стек-фрейм - область памяти, выделяющаяся при входе в функцию. Помимо переменных и входных параметров, хранит адрес возврата.
Новый стек-фрейм создается на месте предыдущего и может хранить мусор, оставшийся от старой функции. Поэтому память нужно переинициализировать.
Память для return на самом деле тоже выделяется на стеке. Вызывающая данную функция может прочесть данные со стека.

Ссылки и указатели

Указатель - переменная, хранящая адрес некоторой ячейки памяти.

```
int   value   = 3;    // Переменная типа int
int * pointer = &value;   // В указатель записывается адрес переменной value
*pointer = 42;        // Запись нового значения по адресу, на который указывает pointer
```

* - оператор **разыменования**
& - оператор **взятия адреса**
Если внутрь функции передать не переменную, а указатель, то работа будет вестись с исходным адресом переменной.

Арифметика указателей

- `(start+index)` - сдвиг на index ячеек int вправо
- `(start-index)` - сдвиг на index ячеек int влево
- `(end-start)` - расстояние между указателями
- `start[index] = *(start+index)`

Ссылки - "обертки" для указателей, которые позволяют избежать операторов разыменования и взятия адреса.
Стандарт C++ не описывают реализацию ссылок, поэтому реализация зависит от конкретного компилятора.
Ссылка не может быть не инициализирована и не может быть переининциализирована, в отличие от указателя. Также у ссылки нет нулевого значения.

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Указатели на функцию

Функция, как и переменная, имеет место в памяти, значение которого может быть присвоено указателю.
Синтаксис:
`<Тип возвращаемого значения> (*имя)(<Типы переменных>)`
Пример:

```
void f(int);
void (*p1)(int) = &f;
void (*p2)(int) = f;
```

Lvalue и Rvalue

Lvalue (locator value) – объект, который имеет идентифицируемое место в памяти.

Rvalue – отрицание lvalue – выражение, которое не представляет собой объект, который имеет идентифицируемое место в памяти.

Основное отличие (которое долгое время было единственным) – lvalue можно изменять, rvalue – нельзя (но в стандарте C++11 можно иметь ссылки на rvalue и тем самым их изменять)

Пример:

```
int var;
int& foo() { return var; }
int foo1() { return var; }
int main(){ //Оператор присваивания слева принимает lvalue
    var = 10;  //Можно, var - lvalue
    foo() = 20; //Можно, & возвращает lvalue - ссылку
    foo1() = 20; //Нельзя, foo1() - rvalue
}
```

Все lvalue, которые не являются массивом, функцией и не имеет неполный тип, могут быть преобразованы в rvalue. Обратное неверно – там, где ожидается lvalue, rvalue использовать нельзя.

Оператор '*' принимает rvalue, а возвращает lvalue:

```
int arr[] = {1, 2};
int* p = &arr[0];
*(p + 1) = 10;
```

Наоборот, ‘&’ принимает lvalue, а возвращает rvalue:

```
int var = 10;
int* a1 = &var; //Можно, var - lvalue
int* a2 = &(var + 1) //Нельзя, var+1 - rvalue, а & требует lvalue
&var = 20; //Нельзя, оператор присваивания требует lvalue
```

В стандарте C++11 были добавлены ссылки на Rvalue. Один из вариантов их применения - создание переносящих операторов присваивания и конструкторов. Их использование позволяет избежать создания внутри метода временного объекта и сэкономить память.

Пример:

```
MyClass& operator=(MyClass&& other)
{
    log("move assignment operator");
    std::swap(*this, other);
    return *this;
}
```

III. Stack Overflow. Структуры и классы

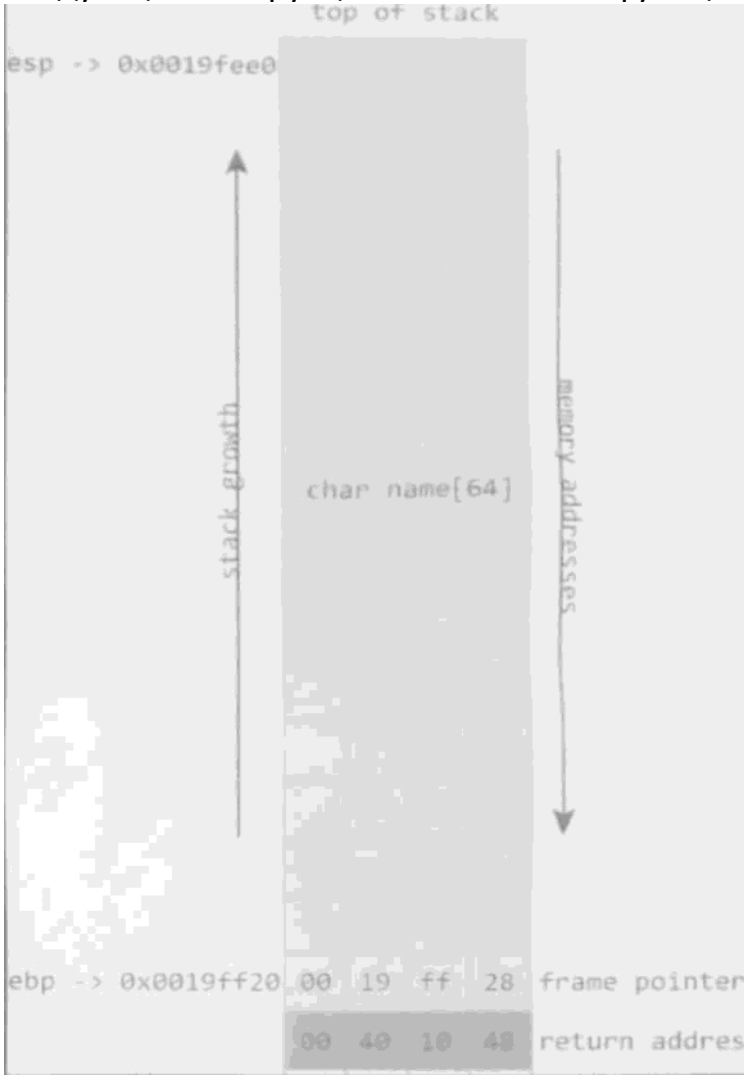
26 февраля 2018 г. 13:51

Атака на переполнение буфера

Для того, чтобы избежать пересечения приложений, в современных ОС каждой программе выделяется виртуальная память. Таким образом любой адрес для программы находится в её адресном пространстве.
Плюс виртуальной адресации - защищённость
Виртуальное пространство разделено на сегменты:

- Адресное пространство ядра - старшие адреса. Память, предназначенная для взаимодействия с ОС.
- Heap - множество адресов фиксированного размера, отводящееся для хранения динамически аллоцируемых данных
- Stack - хранит Fram'ы всех вызываемых функций

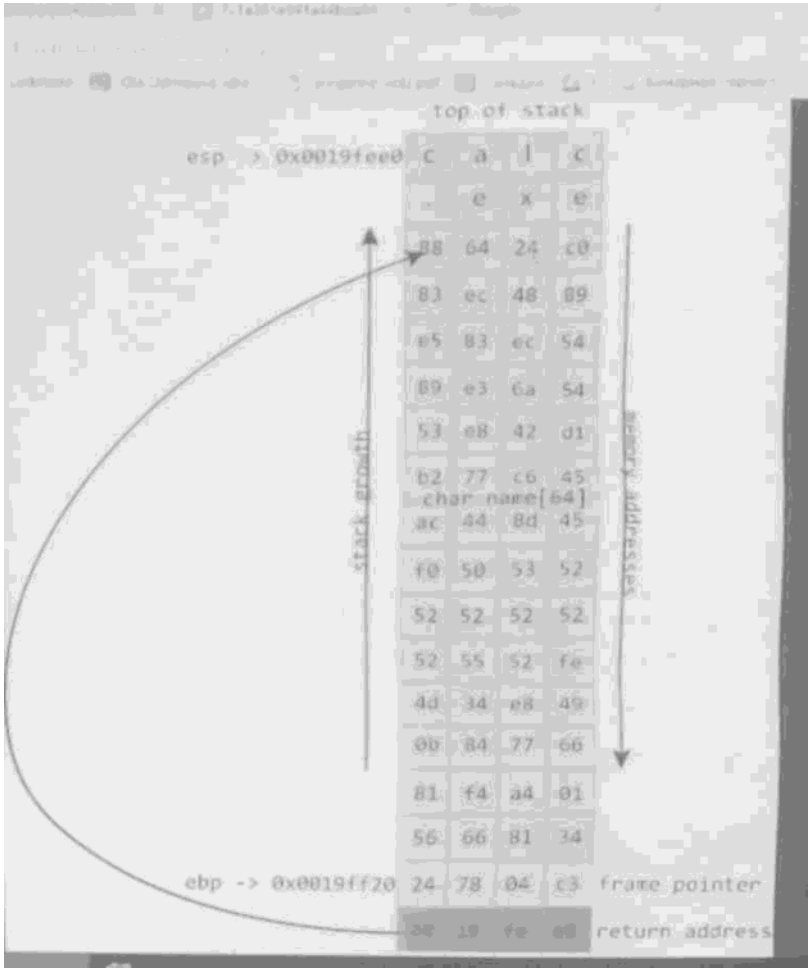
В стеке, помимо всего прочего, хранится адрес возврата - адрес следующей инструкция за вызванной функцией.



Нижние 4 байта (1 машинное слово x86) - адрес возврата
Регистр ebp хранит указатель на frame pointer.

Если направить return address в начало буфера, а в буфер поместить код,

код будет выполнен. Перезапись return adress'a обеспечивает переполнение буфера



Опасные методы - getchar, getc и т.п. опасны тем, что считывают данные до терминального символа. При этом может переполнится буфер. Первая вредоносная программа, эксплуатирующая переполнение буфера - *червь Моррuса*. Он копировал сам себя и в короткое время попал почти на каждое устройство, подключенное к Интернету, но вредоносных действий не совершал. В начало буфера червь Морриса зашил код, получающий рут-права и сканирующий 79-й порт.

Одно из средств защиты - ограничение для записи некоторых страниц (например, страниц кода), но вызывает проблемы с JIT (Just-In-Time) компиляторами

Группировка данных в C++. Структуры и классы

Перечисления - контексты для описания диапазона значений. Каждому элементу соответствует целочисленное значение. Нумерация идет с нуля по возрастанию. Перечисления могут быть преобразованы в целочисленные типы, но не наоборот

```
enum A : int;
enum class B;
enum class C : short;
enum class Color : int {RED = 0x00FF0000, GREEN = 0x0000FF00, BLUE =
0x000000FF};
```

Объединения (union)- состоит из нескольких переменных, занимающих одну и ту же область памяти. Это конструкция обеспечивает низкоуровневую поддержку полиморфизма.

```
union Integer {
     int value;
     short half[2];
};
int main() {
     Integer integer;
     integer.value = 0xFFFF0000;
     cout << integer.half[0] << " " << integer.half[1] << endl;
};
```

Необходимость группировки данных

```
double lenght(double x1, double y1, double x2, double y2)
```

- функция, определяющая длину линии. Но пусть нужно вернуть точку. В таком случае работа с функцией становится неудобной. Чтобы этого избежать, можно использовать **структуры**:

```
typedef struct Point{
     double x;
     double y;
}Point;
typedef struct Segment{
     Point start;
     Point end;
}Segment;

double lenght(Segment);
Point intersects(Segment s1, Segment s1)
```

Структура - группа связанных переменных, составной тип данных.
Имя структуры - спецификатор пользовательского типа
Член структуры - переменная, являющаяся частью структуры

```
struct Point 2D{
     double x;
     double y;
}zero;
struct Point2{
     double x;
     double y;
};
```

Доступ к элементам структур осуществляется с помощью оператора '.', но если доступ осуществляется через указатель, то нужно использовать оператор '->'.

```
#include <cmath>
double length(Segment seg) {
     double dx = seg.start.x - seg.end.x;
     double dy = seg.start.y - seg.end.y;
     return sqrt(dx * dx + dy * dy);
}
double length(Segment *seg) {
     double dx = seg->start.x - seg->end.x;
     double dy = seg->start.y - seg->end.y;
```

```
    return sqrt(dx * dx + dy * dy);
}
```

При определении структур допускается перекрытие имен во вложенных областях видимости, но запрещается дублирование переменных с одинаковыми именами в одной области видимости. Структура может быть объявлена там же, где и обычная переменная.

Инициализация структур

Если не инициализировать структуру, все поля будут забиты мусором. Инициализировать можно сразу в определении, с помощью списка инициализации или с помощью конструктора

```
Point2D first;
int x = 0.5;
int y = 2.5;
Point2D second = {0.5, 3.0};
struct{
    int count;
    char message[13];
}single = {10, "Hello, world!"};
cout << single.message << endl;
```

Из структур можно создавать **массивы структур**.

```
struct Point2D {
    double x;
    double y;
};
struct Segment {
    Point2D points[2];
};
cout << sizeof(Segment) << endl;
```

Передача структур в функцию

Структуры в функцию передаются по значению. Так, такая функция ничего не изменит:

```
void increment(Point2D point){
    point.x++;
    point.y++;
}
```

Исправление 1:

```
void increment(Point2D *point){
    point->x++;
    point->y++;
}
```

Исправление 2:

```
void increment(Point2D& point){
    point.x++;
    point.y++;
}
```

Методы - функции, определенные внутри структуры. Отличие заключается

в том, что методы имеют прямой доступ к полям структуры. Для обращения к методам также используется оператор '.'.
Внутри метода всегда имеется неявный указатель **this**, указывающий на структуру, из которой вызывается метод. Это позволяет обращается к полям при перекрытии имен

```
struct Point2D{
     double x;
     double y;
     void Shift(){
          this->x++;
          this->y++;
     }
}
```

Как и обычные функции, для метода можно разделить объявления и определения. Так, если оператор Shift из предыдущего примера определять вне функции, нужно использовать оператор "::"

```
void Point2D::Shift{
     ...
}
```

Значения по умолчанию

В функциях можно задать значения по умолчанию

Пример

```
void func(int a = 1, int b = 2);
```

В таком случае можно вызвать как `func()`, так и `func(a)` или `func(a,b)`. Единственное правило - все переменные со значениями по умолчанию должны быть справа.

Встраиваемые (inline) функции

- небольшие по объему функции, код которых подставляется в место вызова

Абстракция и инкапсуляция

Абстракция позволяет выделить конкретный набор данных, важный для решения данной задачи. Инкапсуляция объединяет данные и код и защищает их от внешнего воздействия.

```
struct IntArray2D {
     int width;
     int height;
     int *data;
     int &get(int row, int column) {
          return data[row * width + column];
     }
};
int main() {
     IntArray2D array = createArray();
     for (int row = 0; row < array.width; ++row) {
          for (int column = 0; column < array.height; ++column) {
               cout << array.get(row, column) << " ";
          }
     cout << endl;
}
```

На этапе выполнения программы в C++ информация о типах используемых данных стираются, в отличие от C# или Java

Класс - пользовательский тип данных, который задает формат группы объектов. Отличие класса от структуры в модификаторах доступа. В структуре по умолчанию все поля - public, у класса - private.

```
class People{
     int age;
     string name;
public:
     int getAge();
     string getName();
}
int main(){
     People people;
     cout << sizeof(People) << endl;
     cout << sizeof(people) << endl;
}
```

Модификаторы доступа:

- Public - доступ открыт всем, кто видит определение класса
- Protected - доступ открыт классам, производным от данного
- Private - доступ открыт самому классу, друзьям-функциям и друзьям-классам.

По умолчанию все поля и методы класса объявлены закрытыми. Для доступа к ним следует использовать **геттеры** и **сеттеры** - методы, возвращающие значения и устанавливающие значения.

В итоге, отличие структуры от класса в C++ минимально. Когда объявляется структура, компилятор C++ делает класс с модификатором public. В C++ структуры оставлены для совместимости с C.

Публичный интерфейс - список методов, доступный внешним пользователям класса
Инвариант класса - набор утверждений, которые должны быть истинны применительно к любому объекту класса в любой момент времени, за исключением переходных процессов в методах объектов (например, вес человека должен быть положителен). Инварианты могут быть описаны в коде в виде проверок при записи, но могут быть описаны в комментариях или не быть описанными вообще.

Для сохранения инварианта необходимо, чтобы:

1. Все поля были закрытыми
2. Публичные методы должны сохранять инвариант класса.

Перегрузка функций - есть несколько методов с одинаковыми именами,

но разными параметрами.

```
class Point2D {
     int x;
     int y;
public:
     void move(int dx, int dy);
     void move(Point2D vector);
} zero;
int main() {
     Point2D point;
     point.move(10, 20);
     zero.move(point);
}
```

Константы

Определение констант. В C++ константы определяются ключевым словом const. Это значение, которое было инициализировано в начале программы и не меняется в ходе выполнения.
При попытке присвоения константе другого значения компилятор может отказаться компилировать программу, т.к. компилятор может положить константы в защищённую область памяти или подставить значение всюду, где константа использовалась.
Если объявляется константа кастомного типа (например, класс), то компилятор не будет ругаться на использование сеттеров.

Указатели на константу и константные указатели

Объявления `const int*` и `int const*` эквивалентны. Можно изменить значение указателя, но нельзя изменить саму константу.
Объявление `int* const` - *константный* указатель - позволяет изменять значение по адресу, но не сам указатель.
Объявление `int const* const` - *константный указатель на константу*, не позволяет изменить ничего.
Модификатор `const` применяется на тип слева от него.
Стандартами C++ разрешены преобразования вида `P* > T const*`, но запрещены `P** > T const**`

Константные ссылки и ссылки на константу

Ссылка сама по себе является неизменяемой, поэтому нет смысла объявлять const.
Тем не менее, можно делать ссылки на константу - `int const&`

Константные методы

Методы классов и структур могут быть помечены модификатором `const`. В таком случае значения полей не могут быть изменены. Указатель `this` является `Type const* this`.
Кроме того, у константных объектов можно вызывать только константные методы.

```
class IntArray{
```

```
    int size;
    int* data;
public:
    int get(int index) const{
        return data[index];
    }
    int &get(int index){
        return data[index];
    }
}
```

Синтаксическая и логическая константность

Компилятор следит лишь за соблюдением правил синтаксиса языка, поэтому модификатор const сам по себе не гарантирует неизменность полей. Так, с точки зрения компилятора следующий метод корректен:

```
void met() const{
    data[10] = "What is this";
}
```

Однако с логической точки зрения это некорректно.

Ключевое слово mutable

- позволяет определять поля, доступные для изменения внутри константных методов. Его можно использовать только с полями, не являющимися

```
class IntArray{
    int size;
    int *data;
    mutable int counter;
public:
    int size() const{
    ++counter;
        return size;
    }
};
```

IV. Создание, разрушение, копирование, перемещение

5 марта 2018 г. 13:50

Создание объектов

Конструкторы - специальные функции, объявляемые в классе. Имя такой функции совпадает с именем класса. Такие функции не могут возвращать значение и предназначены для инициализации создаваемых объектов.

```
class Date{
     int year;
     int month;
     int day;
public:
     void init(int day, int month, int year);
     void setYear(int year);
     void setMonth(int month);
     void setDay(int day)
}
```

В этом примере нет конструктора. Поэтому при объявлении класса его нужно инициализировать отдельно.

Конструкторы можно перегружать:

```
Date(int day, int month, int year);
Date(int day, int month);
Date(int day);
Date();
```

Но константные конструкторы создавать нельзя.

Конструкторы можно объявлять приватными - это нужно, чтобы запретить вызов определенного конструктора снаружи (например, конструктора копирования) или для контроля над временем жизни объекта с помощью внутренних методов

Для конструкторов можно создавать **списки инициализации**, выполняющиеся до входа в конструктор:

```
Date(int day, int month, int year) : year(year), month(month), day(day){ }
```

Важно, что инициализация проходит в порядке объявления полей в классе, а не в списке инициализации. Это может сыграть роль при выделении памяти под массив, где одно поле зависит от другого. Также при наследовании сначала инициализируются поля суперкласса, а уже потом - наследника.

Конструктор, как и любые другие функции, могут иметь **значения по умолчанию**. Эти значения указываются при объявлении функции.

```
Date(int day = 10, int month = 02, int year = 1970) : year(year),
month(month), day(day){ }
```

Конструкторы одного аргумента задают неявное преобразования от типа этого аргумента к типу класса.

```
class Point{
     int x;
     int y;
public:
     Point(int x = 0, int y = 0): x(x), y(y) { }
}
class Segment{
     Point first;
     Point second;
public:
     Segment() : first(0, 0), second (0, 0){}
     Segment(int lenght) : first(0, 0), second(length, 0) {}
}
int main{
     Segment second = 10;
}
```

Здесь возможно неявное преобразование от числа к сегменту, что может сбить с толку. Чтобы этого избежать, можно использовать ключевое слово **explicit.**

Конструктор по умолчанию - создается, если в классе не объявлен конструктор. Он вызывает конструкторы для объектов класса, но указатели и значения переменных остаются мусорными.

Однако если уже объявлен один явный конструктор, то default-конструктор без параметров не создается. Если есть необходимость явно объявить стандартный конструктор, то со стандарта C++11 есть ключевое слово **default**

```
Segment() = default;
```

Делегирующий конструктор - построение цепочки вызовов конструкторов в рамках одного и того же класса. Это сокращает дублирование кода

```
class Point{
     int x;
     int y;
public:
     explicit Point (int x = 0, int y = 0) : x(x), y(y){
          cout << x << "-" << y << endl;
     }
     explicit Point (double y) : Point(0, y)){
          cout << x << "-" << y << endl;
     }
}
```

Делегирующий конструктор отрабатывает в первую очередь.

Особенности синтаксиса C++

Если что-то похоже на объявление функции - это и есть объявление функции. Строка Point second() с точки зрения компилятора будет объявлением функции, а не переменной.

Разрушение объектов

Деструкторы - специальные функции, объявляемые в классе и предназначенные для разрушения объектов, т.е. для освобождения аллоцированных ресурсов. Объявляется значком ~, параметров и возвращаемого значения не имеет. Вызывается оператором delete или при выходе объекта из области видимости

```
class IntArray{
     int size;
     int* data;
public:
     explicit IntArray(int size): size(size), data(new int[size]){}
     ~int Array(){
          delete[] data;
     }
}
```

Важно, что если в данном примере поменять в объявлении size и data, код работать не будет, т.к. он инициализируется позже, чем data. Size будет взят из поля класса, а не из параметра конструктора. Поэтому сначала нужно инициализировать размер.

Время жизни объекта - интервал между корректным завершением конструктора и вызовом деструктора.
Деструкторы переменных на стеке вызываются в обратном порядке к объявлению переменных. Это сделано для корректного освобождения зависимых объектов.

Обращение к объекту до начала его жизни возможно с помощью арифметики указателей (попытаться вычислить его будущее местонахождение). В любом случае это ведет к неопределенному поведению.

Для динамически аллоцированных методов необходимо явно вызывать деструкторы, чтобы не допустить утечки памяти.

Чтобы обеспечить корректное освобождение памяти при наследовании деструктор суперкласса стоит объявить виртуальным. В этом случае всегда будет вызываться нужный деструктор.

```
class MyClass1{
     virtual ~MyClass1();
}
class MyClass2 : MyClass1{
     ~MyClass2();
}
MyClass1* ptr = new MyClass2;
delete ptr; //Будет вызван ~MyClass2()
```

Иначе в данном примере был бы вызван `~MyClass1()`

Работа с динамической памятью в C

- `void* malloc(size_t sizemem) //выделяет блок памяти и возвращает указатель на начало блока`
- `void* calloc(size_t nmemb, size_t size) //выделяет память для массива и`

обнуляет её
- void* realloc(void *ptr, size_t size) //изменяет размер блока памяти
- void free(void *ptr) //освобождает блок памяти, полученный ранее динамическим выделением
- realloc(ptr, 0) == free(ptr)

Работа с динамической памятью в C++

Для создания новых объектов используется оператор `new`. Помимо аллоцирования он вызывает конструктор для объектов. Освобождение происходит с помощью оператора `delete`.
Чтобы выделить память для массива, используется оператор `new[]` (Соответственно, чтобы освободить - `delete[]`)

```
class IntArray
{
     size_t size;
     int *data;
public:
     explicit IntArray(size_t size) : size(size), data(new int[size]) {}
          ~IntArray() { delete[] data; }
};
int main()
{
     // Только выделение памяти.
     IntArray *oldStyle = (IntArray *)malloc(sizeof(IntArray));
     // Выделение памяти и создание объекта.
     IntArray *newStyle = new IntArray(5);
}
```

Оператор new с размещением:

```
void* pointer = myalloc(sizeof(IntArray));
IntArray* array = new(pointer)IntArray(10);
```

Копирование объектов

```
class IntArray{
     int size;
     int* data;
public:
     explicit IntArray(int size): size(size), data(new int[size]){}
     ~int Array(){
          delete[] data;
     }
}
void* pointer = myalloc(sizeof(IntArray));
IntArray* array = new(pointer)IntArray(10);
int main(){
     IntArray array1(10);
     IntArray array2(20);
     IntArray array3 = array1; //Объявлена новая переменная, происходит
     побайтное копирование из array1
     array2 = array1; //Прозводится присваивание, скопированы все значения из
     области памяти array1
     return 0;
}
```

В данном случае после работы array2 = array1 потеряется явный доступ к array2 и утечёт память. Кроме того, для одной и той же области памяти

несколько раз вызовется деструктор, что приведет к ошибке

Конструктор копирования

Корректный конструктор копирования будет выглядеть так

```
IntArray(const IntArray& array) : size(array.size), data(new int[size]){
     for (int i = 0; i<size;i++){
     data[i] = array.data[i];
     }
}
```

Конструктор присваивания

Вызывается в том случае, когда присваивание производится в уже существующий объект. Принимает констатнтную ссылку на объект того же типа, возвращает ссылку на this

```
IntArray& operator=(IntArray const &other){
     if (this!=other){ //Чтобы избежать ошибки при присваивании самому себе
          delete[] data;
          size = other.size;
          data = new int[size];
          for (int i = 0; i<size;i++){
               data[i] = array.data[i];
          }
     }
}
```

Метод swap

```
void swap(IntArray& other)
{
     std::swap(size, other.size);
     std::swap(data, other.data);
}
```

С его помощью можно реализовать оператор присваивания следующим образом:

```
IntArray& operator=(IntArray const& other){
     if (this!=other){
          IntArray(other).swap(*this);
     }
     return *this;
}
```

Запрет копирования объектов

Некоторые классы с точки зрения семантики не подразумевают копирование - например, кэш или пул тредов. Запрет копирования можно сделать двумя способами:

1. Поместить конструктор копирования и присваивания в private
2. Присвоить конструктору копирования и присваивания спецификатор delete (с C++11).

Перемещение объектов

В C++11 добавлен конструктор перемещения. Используется в тех местах, где объявляется переменная и её тут же присваивается rvalue.

```
IntArray(IntArray&& tmp) : size(tmp.size), data(tmp.data){
```

```
    tmp.data = NULL;
}
```

Оператор присваивания перемещения используется тогда, когда справа от = стоит rvalue-ссылка.

```
IntArray& operator=(IntArray&& tmp){
this = ~IntArray();
size = tmp.size;
data = tmp.data;
tmp.data = NULL;
return *this;
}
```

Правило трех: если структура определяет конструктор копирования, оператор присваивания или деструктор, то всех их нужно явно определить.

Правило пяти (C++11): если структура определяет конструктор копирования, оператор присваивания, конструктор перемещения, оператор перемещения, деструктор, то всех их нужно явно определить.

Идиома программирования

- устойчивый способ выражения некоторой составной конструкции в языках программирования. Шаблоны проектирования не зависят от языка программирования; идиомы же зависят от особенностей языка.

Идиома RAII

Resource Acquisition Is Initialization - получение ресурса есть инициализация

Идея в том, что с помощью конкретных механизмов языка связывается создание объекта с инициализацией ресурса, разрушение объекта с освобождением ресурса. Типичный способ реализации - получение доступа в конструкторе, а освобождение - в деструкторе.

По истечении определенного времени любой выделенный ресурс нужно освободить.

Пример:

```
class File{
    const std::FILE *file;
public:
    file(const char* filename) : file(std::fopen(filename, "w+")){
        if (!file){
            trow std::runtime_error("File open failtur");
        }
    }
    ~file(){
        std::fclose(file);
    }
    void write(const char* data){
        if (std::fputs(data,file) == EOF){
```

```
                throw std::runtime_error("file write failture");
            }
        }
}
```

Здесь идиома заключается в том, что конструктор захватывает управление объектом, а деструктор его освобождает.
При использовании этого класса файл будет закрыт, как только File выйдет из области видимости. Поэтому нельзя выделять динамически память под объекты идиомы RAII.

V. Обработка исключительных ситуаций

12 марта 2018 г. 14:36

Традиционные варианты обработки ошибок:

1. Остановка программы
2. Возвращение определенного кода. Стандартное соглашение - 0 - всё хорошо, все остальное - коды ошибки.
   Проблема в том, что иногда методы должны возвращать что-то помимо кода ошибки. Кроме того, конструкторы ничего не возвращают.
3. Завести статическую глобальную переменную и использовать её как средство сигнализации об ошибках.
   Недостаток в загрязнении кода дополнительными проверками и перемешивании логики. Плохо работает в многопоточной среде.
4. Создание отдельной функции-обработчика ошибок.
   Минус в том, что эту функцию сложно написать в начале и сложно поддерживать. Нарушает принцип минимальной ответственности - она следит за всей программой.

В большинстве случаев **механизм обработки исключений** выигрывает у вышеперечисленных методов.
Этот механизм разделяет код на "естественное" выполнение программы и "исключительное" и предоставляет единый стиль обработки ошибок

Выброс исключения - если функция понимает, что она не может дальше корректно продолжать работать, она создает и выбрасывает объект - исключение.
Захват исключения - функция, которая может обработать исключение, ловит их в определенных блоках и обрабатывает, решая проблему.

```
void taskMaster(){
     try{
          int result = doTheTask();
     } catch(SomeError &error){
          //Обработка исключения
     }
}
int doTheTask(){
     if(/*Возможно выполнение функции*/){
          return result;
     }
     else{
          throw SomeError();
     }
}
```

При выбросе исключение стек раскручивается, пока не найдется обработчик исключения.

Обычно в исключении запакована какая-либо информация, позволяющая распознать и исправить ошибку. Исключение может быть любым объектом, который можно копировать. Но рекомендуется всё же создавать свои классы для исключений или использовать стандартные во избежание пересечений с используемыми библиотеками.

Исключение обычно используется, когда код, где может вылезти ошибка, расположен далеко от места проверки. Это хорошо для библиотек - они не обязаны обрабатывать ошибки, но предоставляют вызывающему коду возможность решить проблему.

Исключительная ситуация может вызываться достаточно часто и означает только то, что должен вызываться особый способ обработки. Но следует избегать "естественной" работы программы с только с исключениями. Так, не стоит возвращать какой-либо значение с помощью исключений.

Препятствия к использованию исключений

- Если система критическая к задержке. Например, системы реального времени, которые должны отвечать на запросы за строго определенное время. При исключении неизвестно, сколько будет раскручиваться стек до поиска блока catch.
- Если в программе реализован собственный механизм распределения ресурсов, несовместимый со стандартным.

Обработка ошибок без исключений

- В конструкторе: добавить в конструктор метод, проверяющий инвариант класса (int Valid()) и вызывать его после создания.
```
void function(int size) {
     MyVector vector(size);
     switch (vector.invalid()) {
          // Обработка ошибки ...
     }
     // Выполнение функции ...
}
```
- В функции: можно возвращать два значения - value и errorcode. Одно из них в любом случае будет заполнено
```
void function(int size) {
     Pair<MyVector, ErrorCode> result = createVector(size);
     switch (result.second) {
          // Обработка ошибки ...
     }
     MyVector value = result.first;
     // Выполнение функции
}
```

Многоуровневая обработка исключений

Систему можно разрезать на слои, каждый из которых может выбросить и обработать определенные исключения. Эти слои образует определенную

иерархию. Для каждого слоя нужно определить степень ответственности. Если слой не может обработать исключение, он делегирует обработку вышестоящему модулю.

Возбуждение исключения

Оператор throw создает объект-исключение и "пробрасывает" его вверх, чем запускает раскручивание стека в поисках обработчика.

```
class MyException{
     int errorCode;
public:
     MyException(int errorCode) : errorCode(errorCode){}
}
int main(){
     throw MyException(); //Некорректно
     throw MyException; //Тем более
     throw MyException(42); //Нормально
     throw new MyException(42); //Некорректно, не нужно выделять исключения в
     динамической памяти
}
```

Блок try

Внутрь этого блока помещается функция, которая может вызвать исключительная ситуация

Блок catch

Выполняется в случае данного исключения

```
try{
     func();
} catch(MyException1){
     //Обработка исключений типа MyException1
} catch(MyException2 ex){
     //Обработка исключений типа MyException2
} catch(MyException3 &ex){
     //Обработка исключений типа MyException3
}
```

Есть возможно перехвата исключений любых типов - например, для логирования. В этом случае можно использовать catch(…).

Раскрутка стека

В момент выброса исключения начинается раскрутка стека в поисках блока catch, способного обработать исключительную ситуацию. Если обработчик так и не найден, будет вызван стандартный обработчик - terminate - завершает программу без вызова деструкторов.
Можно использовать terminateHandler для переопределения стандартного обработчика.

Повторное возбуждение исключений

Может оказываться, что в одном блоке catch не удалось полностью обработать исключение. Тогда, выполнив некоторые действия, catch-блок

делегирует обработку исключения далее с помощью throw

```
try{
     func();
}
catch{
     //обработка
     throw;
}
```

Ресурсозатратность механизма исключений

Если исключение вылетает во время создания объекта, то это вызовет утечку памяти, т.к. время жизни объекта еще не началось. Поэтому в таком случае нужно вызвать деструкторы в правильном порядке.

Спецификатор исключений throw

Указывается после списка аргументов функции. Декларирует список исключений, которые может выбросить функция. Проблема в том, что функция может выбросить исключение, незадекларированные в throw. Поэтому этот спецификатор не рекомендован к использованию в стандарте c++11.

Спецификатор исключений noexcept

Также добавляется в конец описываемого метода после списка параметров. Если метод помечен модификатором noexcept, компилятор не будет ловить исключения в методе, что снижает накладные расходы. Установка этого модификатора подразумевает гарантию, что функция не выбросит исключения. Если исключение будет всё же выброшено, выполнение программы будет остановлено и стек не будет раскручен.

Гарантии механизма исключений

Exception-safe - после вызова этого метода программа остается в консистентном состоянии вне зависимости от успешности операции.
Консистентное состояние - соблюдаются инварианты для всех объектов.

Гарантии библиотечных функций:

- Basic - соблюдаются базовые инварианты для всех объектов, нет утечек ресурсов
- Strong - дополнительно к Basic операция либо выполняется полностью, либо не выполняются совсем. Таким операции называются **атомарными**
- Nothrow - гарантирует, что никакое исключение не может быть возбуждено при выполнении операции.

Классы исключений языковой поддержки

- bad_alloc – генерируется при неудачном выполнении оператора new
- bad_cast – генерируется оператором dynamic_cast , если преобразование типа завершается неудачей

- bad_typeid – генерируется оператором typeid , в случае если в качестве аргумента передан nullptr
- bad_exception – предназначено для обработки непредвиденных исключений

Классы исключений стандартной библиотеки

- invalid_argument – передано недопустимое значение аргумента
- length_error – осуществлена попытка выполнения операции, нарушающей ограничения на максимальный размер
- out_of_range – аргумент не входит в интервал допустимых значений
- domain_error – ошибка выхода за пределы области допустимых значений
- ios_base::failure – генерируется при изменении состояния потока вследствие ошибки или достижения конца файла

Классы исключений для внешних ошибок:

- Range_error - выход за пределы допустимого интервала
- Overflow_error - математическое переполнение
- Underflow_error - переполнение в нижнюю сторону

При выбрасывании исключения лучше создать наследника от одного из вышеперечисленных типов.

Исключения в конструкторе и деструкторе

В конструкторе исключения бросать можно. Но если исключение выброшено в конструкторе, объект ещё не полностью сконструирован, поэтому его не надо разрушать и деструктор вызван не будет. Поэтому перед выбросом исключения необходимо почистить.

Исключения в деструкторе очень сложно реализовать из-за сложностей при раскрутке стека и освобождения памяти, поэтому считается, что исключения в деструкторах недопустимы.

Дополнение к идиоме RAII:

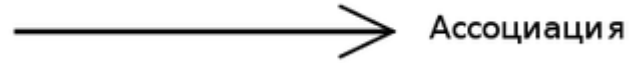
Если конструктор не является exception-safe, то при некорректном завершении конструктора будет утечка памяти. Например, при вылете в конструкторе, открывающем файл, мы не сможем его закрыть.

Exception-safe можно достичь, если в публичном конструкторе делегировать внутренний конструктор для объекта, после чего проверять на исключения. Как только отработал один конструктор до конца, время жизни объекта уже начинается и для него может быть корректно вызван деструктор.

VI. Виды отношений между классами

19 марта 2018 г. 14:34

Ассоциация - самое слабое отношение. Говорит о том, что между классами есть какое-то отношение. Например, это может быть вызов объекта одного класса объектом другого классом



Агрегация - объект одного класса может владеть объектом другого класса. Целое не является владельцем части не управляет временем жизни.

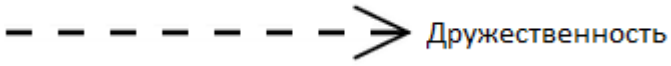

```
class Person {
     std::string name;
     std::string surname;
     int age;
     // Constructors
     // Getters
     // Setters
};
class Club {
     Person *members;
public:
     void addMember(Person *member) {
          // Добавление участника
     }
void removeMember(Person *member) {
     // Исключение участника
     }
};
```

(Обычно) удаление человека из клуба не прекращает существование человека во внешнем мире

Композиция - описывает целое и составные части, которые в него входят. Конкретный экземпляр части может принадлежать только одному владельцу. Целое управляет временем жизни входящих в него частей


```
class Point {
     int x;
     int y;
public:
     Point(int x, int y): x(x), y(y) {}
};
class Circle {
     Point *center;
     int radius;
public:
     Circle(int x, int y, int radius)
          : center(new Point(x, y)), radius(radius) {};
     ~Circle() { delete center; }
};
```

Дружественность - однонаправленная связь, в отличие от предыдущих. Это отношение не может быть предписано извне. Дружественность предоставляет внешнему классу доступ к приватным методам класса.

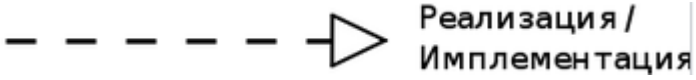



```
class Tv {
     int currentChanel;
     friend class RemoteControlTv;
};
class RemoteControlTv {
public:
     void changeChannel(Tv& tv, int channel) {
          tv.currentChannel = channel;
     }
};
```

Наследование - один класс наследуется от другого, перенимая его состояние и дополняет его. Функционал родительского класса полностью доступен в производном классе.


```
class Unit {
     int health;
public:
     Unit() : health(10) {}
     Unit(int health) : health(health) {}
     int getHealth() {
          return health;
     }
};
class Soldier : public Unit {
     int damage;
public:
     Soldier() : damage(20) {}
     Soldier(int damage) : damage(damage) {}
     Soldier(int health, int damage) : Unit(health), damage(damage) {}
     int getDamage() {
          return damage;
     }
};
```

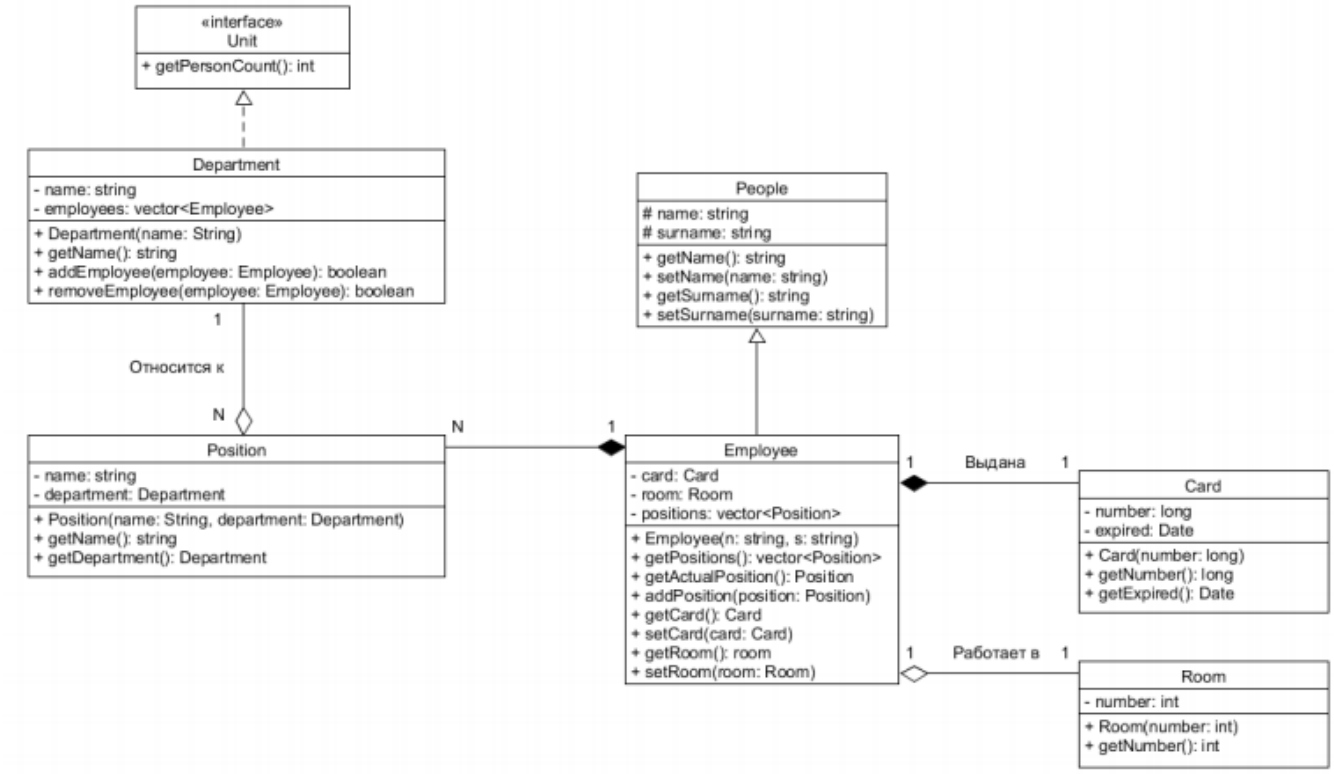
Реализация - наследование от интерфейса. В C++ нет отдельной сущности, называемой интерфейсом.
Интерфейс - класс, у которого все методы публичные, виртуальные и не обладающие телом




```
class Serializable {
public:
     virtual char* serialize() const = 0;
};
class MyClass : public Serializable {
public:
     char* serialize() const override {
```

```
        // Логика сериализации объекта
    }
};
```

Диаграмма классов



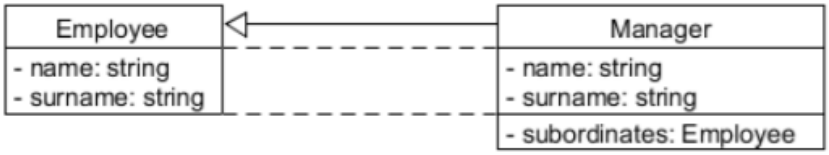
VII. Механизм наследования

2 апреля 2018 г. 13:44

Механизм наследования

Наследование - один из важнейших механизмов ООП. Он позволяет создавать новые классы на основе уже существующих. Новые классы изменяют или расширяют функционал тех классов, на основе которых они создаются.

В памяти подклассы располагаются следующим образом: сначала всегда располагаются поля и методы из суперкласса, а потом уже новые. Это позволяет везде, где есть ссылка или указатель на объект суперкласса использовать его наследников.




```
class Employee; // Только объявление без определения
class Manager : public Employee {
    // Определение класса
};
```

Для использования в качестве базового, класс должен быть определён перед этим.

Типы наследования

Базовый класс может быть объявлен со следующими модификаторами доступа:

- Public - ничего не меняется
- Protected - все объекты суперкласса, которые были public, становятся protected
- Private - все объекты суперкласса объявляются private

Ни при каком типе наследования в подклассе недоступны приватные члены суперкласса, а также его конструкторы и деструктор.

Порядок конструирования и разрушения объектов

Объекты создаются "снизу-вверх" - от базовых к производным. Порядок вызова конструкторов таков:

1. Конструкторы виртуальных базовых классов (в порядке объявления в списке наследования)

2. Конструкторы прямых базовых классов (в порядке объявления в списке наследования)
3. Конструкторы полей (в порядке объявления в классе)
4. Конструктор класса (вызванный)

Разрушаются же объекты в обратном порядке:

1. Деструктор класса (вызванный)
2. Деструкторы полей (в порядке обратном объявлению в классе)
3. Деструкторы прямых базовых классов (в порядке обратном объявлению в списке
наследования)
4. Деструкторы виртуальных базовых классов (в порядке обратном объявлению в списке
наследования)

Чтобы обеспечить корректное освобождение памяти при наследовании деструктор суперкласса стоит объявить виртуальным. В этом случае всегда будет вызываться нужный деструктор.

Определение типа объекта

В указателе на суперкласс может располагаться как суперкласс, так и какой-либо из подклассов. Чтобы определить тип объекта, хранящегося там, можно:

- Использовать в указателе модификатор final - в таком случае указатель сможет ссылаться только на объект базового класса
- Использовать специальное поле для хранения информации об объекте
- Использовать dynamic_cast
- Использовать механизм виртуальных функций

Пример использования поля:

```
class Employee {
public:
     enum EmployeeType {MANAGER, EMPLOYEE};
     Employee() : type(EMPLOYEE) {}
     EmployeeType getType() const {
          return type;
     }
protected:
     Employee(EmployeeType type) : type(type) {}
private:
     EmployeeType type;
};
class Manager : public Employee {
public:
     Manager() : Employee(MANAGER), level(0) {}
     int getLevel() const {
          return level;
     }
private:
     int level;
```

```
};

void printEmployee (const Employee *employee) {
     switch (employee->getType()) {
     case Employee::MANAGER:
          const Manager *manager = (const Manager*)employee;
          cout << manager->getLevel() << endl;
     case Employee::EMPLOYEE:
          cout << employee->getName() << endl;
     }
}
void printList(const vector<Employee*>& employees) {
     for (Employee* current : employees) {
          printEmployee(current);
     }
}
```

Виртуальные функции

Виртуальная функция - это такая функция, которую предполагается переопределить в производных классах. При использовании объекта подкласса через указатель или ссылку на суперкласс при вызове виртуальной функции будет вызвана соответствующая функция из подкласса (если она определена).

```
class Employee {
public:
     virtual void print() {
          cout << "I'm employee" << endl;
     }
};
class Manager : public Employee {
     void print() override {
          Employee::print();
          cout << "I'm also manager" << endl;
     }
};
void printList(const vector<Employee*>& employees) {
     for (Employee* current : employees) {
          current->print();
     }
};
```

Полиморфный класс - класс, имеющий хотя бы одну виртуальную функцию. Каждый объект такого класса содержит **таблицу виртуальных функций (vtable)**. При использовании ссылки или указателя разрешение методов происходит динамически в момент вызова.

```
struct Person {
     virtual ~Person() {}
     virtual string position() const = 0;
};
struct Teacher : Person {
     string position() const;
     virtual string course();
};
struct Professor : Teacher {
     string position() const;
     virtual string thesis();
};
```

Person

0	~Person	0xAB20
1	position	0x0000

Teacher

0	~Teacher	0xAB48
1	position	0xAB60
2	course	0xAB84

Professor

0	~Professor	0xABA8
1	position	0xABB4
2	course	0xAB84
3	thesis	0xABC8

Виртуальные функции в конструкторе и деструкторе

Для каждой функции используется версия, специфичная для класса, в конструкторе/деструкторе которого она вызвана

Приведение типов при наследовании

Как уже было сказано, везде, где есть ссылка или указатель на объект суперкласса использовать его наследников.

```
Manager manager(“Name”, “Surname”, “Sales”);
Employee &ref = manager; // Manager& -> Employee&
Employee *ptr = &manager; // Manager* -> Employee*

Manager manager(“Name”, “Surname”, “Sales”);
Employee employee = manager; // Employee(“Name”, “Surname”);
```

Наследование с модификаторами

```
class Class {};
class PublicChild : public Class{};
class ProtectedChild : protected Class{};
class PrivateChild : private Class{};
```

- При использовании публичного наследования использование ссылки на подкласс допустимо везде
- При protected-наследовании о том, что Class является суперклассом для ProtectedChild, знают только его наследники и сам класс
- При private-наследовании приведение ссылки к базовому доступно только внутри PrivateChild

Чистая виртуальная функция - такая функция, которая объявлена в базовом классе, но не имеет определения. В таком случае каждый подкласс обязан иметь собственную версию этой функции. Для того, чтобы создать чистую виртуальную функцию, после списка аргументов нужно указать "=0".

Абстрактный класс - класс, который обладает хотя бы одной чисто виртуальной функцией. Обычно располагается на вершине иерархии классов. Объекты такого класса создать нельзя.

Интерфейс - класс, у которого все методы публичные, чисто виртуальные и нет полей. Если абстрактный класс *наследуется*, то интерфейс *реализуется.* В C++ отдельного понятия "интерфейс" нет, в отличие, например, от Java.

Переопределение метода - возможность подклассу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов.

Перегрузка методов - создание нескольких методов с одинаковым именем, но с разным набором параметров.
Если есть несколько перегруженных методов в базовом классе и несколько таких же методов в дочернем классе, методы базового класса не участвуют в перегрузке. Чтобы они участвовали, нужно объявить namespace с именем класса

```
struct File {
     void write(std::string string);
     ...
};
struct FormattedFile : File {
     void write(int value);
     void write(double value);
     using File::write;
     ...
};
```

Разрешение перегрузок при наследовании

В общем случае разрешение перегрузок происходит по следующему порядку:

- При наличии точного совпадения сигнатуры - используется найденная функция
- Если нет, то ищутся функции, которые могут подойти с учётом преобразований:
  - Расширение типов:
    ```
    char, signed char, short -> int
    unsigned char, unsigned short -> int / unsigned int
    float -> double
    ```
  - Стандартные преобразования (числа, указатели)
  - Пользовательские преобразования

  Если найдена единственная подходящая функция, которая строго лучше остальных по каждому из параметров, то перегрузка разрешается, в противном случае выдается ошибка.

При разрешении перегрузок функций используется статический

полиморфизм и **раннее связывание,** а при вызове виртуального метода - динамический и **позднее связывание**

VIII. Множественное наследование. Приведение типов

2 апреля 2018 г. 13:44

Переопределение private виртуальных методов

```
struct NetworkDevice{
     void send(char* data, int size){
          log("Start sending");
          sendImpl(data, size);
          log("Start sending");
     }
private:
     virtual void sendImpl(char* data, int size){
          //Обобщенная реализация передачи данных
     }
}

struct Router : NetworkDevice{
private:
     void sendImpl(char* data, int size){
          //Конкретная реализация передачи данных
     }
}
```

В данном примере при вызове метода *send* из *Router* будет вызвана конкретная реализация.

Множественное наследование

Если порожденный класс наследует элементы одного базового класса, то такое наследование называется **одиночным**. **Множественное** наследование позволяется порожденному классу наследовать элементы более чем от одного базового класса. В настоящий момент практически не используется и оставлено для обратной совместимости.

```
class X{};
class Y{};
class Z{};
class A: public X, protected Y, private Z {};
```

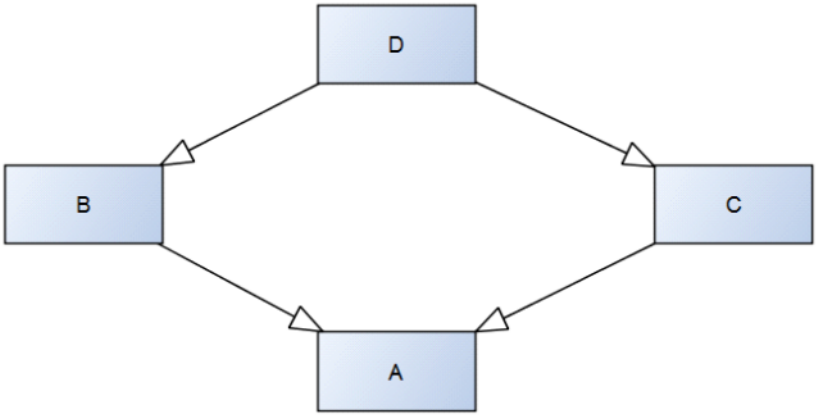
Доступ к членам

Для доступа к членам порожденного класса, унаследованного от нескольких базовых, может быть необходимо явно указать имя базового класса, к которому идет обращение.

Конструирование объектов

Объект порожденного класса состоит из нескольких частей, унаследовано от базовых классов. При конструировании дочернего класса нужно вызвать конструкторы для всех базовых классов. Конструкторы выполняются в порядке объявления конструкторов в списке наследования

Ромбовидное наследование



Если класс D наследуется и от B, и от C, то возникают две проблемы:

1. Дублирование состояния. В классе D будут содержаться поля классов B и C, которые могут дублироваться.
2. Дублирование поведения. Если на объекте класса D будет вызван метод, то компилятор не сможет разрешить, какой метод вызвать - из класса B или C. В классе D будут два указателя на разные таблицы виртуальных функций.

Для решения первой проблемы нужно использовать **виртуальное наследование**. При наследовании B от A и C от A нужно будет добавить ключевое слово virtual. При использовании этого модификатора в дочерних классах их собственные поля будут идти вначале и в классе D будет только одна копия класса A.

Для решения второй проблемы нужно использовать виртуальные методы. В таком случае для устранения неоднозначности в классе D нужно переопределить метод, возможно, используя какие-то из базовых классов. В классе D будет два указателя на одну таблицу виртуальных функций.

Рефакторинг - переписывание кода без изменения функциональности, т.е. повышение его читаемости, производительности без изменения публичного интерфейса.
Юнит-тест - кусок кода, который тестирует другой код. Их использование сокращает время поиска ошибок при больших проектах. Тесты рекомендуется разбивать на классы эквивалентности.
Чистый метод - метод, который не взаимодействует с внешним состоянием, т.е. работает только с теми параметрами, которые получает на вход. Использование таких методов значительно облегчает написание юнит-тестов.
Гипотеза разбитых окон - если "разбито одно окно" и не починено вовремя, то остальные окна будут разбиты в короткие сроки. Аналогичная гипотеза работает с тестами - если какой-то тест не работает, ошибку стоит исправить сразу.

Приведение типов

Приведение типов в C

Неявное приведение типов производится автоматически при компиляции. Некоторые возможные приведения:

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

Явное приведение вызывается следующим образом:

(type) expression

Статическое приведение типов (static_cast)

Появилось в C++. Используется для преобразования одного типа в другой, но не может быть использовано для выполнения недопустимого преобразования. Выполняет следующие преобразования:

- Приведение указателя базового класса к указателю производного класса (без проверок типа во время выполнения)
- Приведение численных типов
- Вызов метода класса, осуществляющего преобразование

```
class A{};
class B : public A {};

int main(){
     double result = static_cast<double>(13) / 7;
     A *ptr = static_cast<A*>(new B());
     ptr = static_cast<A*>(&result); //Ошибка
}
```

Динамическое приведение типов (dynamic_cast)

Имеет смысл только для объектов класса, входящего в иерархию полиморфных типов. Используется для безопасного преобразования указателя (ссылки) на суперкласс в указатель (ссылку) на подкласс.
В отличие от static_cast, проверка корректности преобразования происходит на этапе выполнения программы, и в случае невозможность каста dynamic_cast вернёт nullptr.

```
class A{
protected:
     virtual ~A() {}
}
class B : public A {}
class C1 : public B {}
class C2 : public B {}

int main(){
     A* parent = new C1();
     C2* derived = dynamic_cast<C2*>(parent);
}
```

Константное приведение типов (const_cast)

Используется для приведения константного указателя к неконстантному.

Проверка совместимости производится на этапе компиляции.

```
void funct(char *); // прототип функции с неконстантным параметром
const char *string = "Sevastopol"; // константная строка
func(const_cast<char *>(string));
```

Приведение reinterpet_cast

Является наименее безопасным приведением в C++.

```
class A{
     int a;
};
class B{
     int b;
};
struct C: A, B {};
int main(){
     C c;
     c.a = 1;
     c.b = 2;
     cout << reinterpret_cast<B*>(&c)->b << endl; //Будет выведено 1
     cout << static_cast<B*>(&c)->b << endl; //Будет выведено 2
}
```

Приведение типов в Си-стиле на C++

Если использовать приведение в стиле Си, то компилятор будет пытаться использовать приведения в следующем порядке:

1. const_cast
2. static_cast
3. static_cast + const_cast
4. reinterpret_cast
5. reinterpret_cast + const_cast

Порядок идет от наиболее безопасных к наиболее опасным.

RunTime Type Identification (RTTI)

- механизм, позволяющий определять тип данных во время выполнения программы. В C++ для реализации такого механизма есть два пути:

Использование typeid

Для использования этой функции необходимо добавить в компилятор опцию, включающую сохранение типов.

```
#include <iostream>
#include <typeinfo.h>
class Base {
public:
     virtual void vvfunc() {}
};
class Derived : public Base {};
using namespace std;
int main() {
     Derived* pd = new Derived;
     Base* pb = pd;
     cout << typeid( pb ).name() << endl; //prints "class Base *"
     cout << typeid( *pb ).name() << endl; //prints "class Derived"
     cout << typeid( pd ).name() << endl; //prints "class Derived *"
     cout << typeid( *pd ).name() << endl; //prints "class Derived"
```

```
    delete pd;
}
```

Использование dynamic_cast

dynamic_cast выполняет приведение к указанному типу с проверкой. Если dynamic_cast возвращает nullptr, то приведение не состоялось и объект не того типа, которого предполагалось.

IX. Перегрузка операторов

Monday, April 16, 2018 13:57

Перегрузка операторов

Необходимость перегрузке может возникнуть при написании собственных методов - логично было бы реализовать уже существующие методы, а не определять аналогичные свои.
Перегрузка - лишь более удобный способ вызова функций. Компилятор будет искать нужную функцию в перегрузках.

Основные операторы:

	Унарные	Бинарные
Арифметические	Префиксные: + - ++ -- Постфиксные ++ --	+ - * / += -= *= /= %=
Битовые	~	& | ^ &= |= ^= >> <<
Логические	!	&& || == != <> >= <=

- Оператор присваивания =
- Специальные:
  - Префиксные: *, &
  - Постфиксные: ->, ->*

    Отличие -> от ->*

```
class A{
     int* ptr = new int(0);
}
int main(){
     new A()->ptr //Получим int*
     new A()->* //Получим int
}
```

  - Особые: , . ;

    Запятая может применятся, например, в перечислении в третьем пункте for
- Скобки [] ()
- Оператор приведения (type)
- Тернарный оператор x ? Y : z
- Работа с памятью: new new[] delete delete[]

Все эти методы можно переопределить, кроме . и оператора определения области видимости

Синтаксис перегрузки

Операторы необходимо перегружать, используя имя operator [OP]
Для операторов (type) [] () -> ->* = обязательно определение в классе.
Пример:

```
A operator+(A const& left, A const& right){
     return A(left.x + right.x, left.y + right.y);
}
```

Для перегрузки данных операторов необходимо посмотреть спецификацию и узнать, какие параметры оператор принимает на вход и на выход.

Перегрузка инкремента и декремента:
Постфиксная версия принимает один неиспользуемый int-параметр для различения. В остальном перегрузка не отличается.

```
struct BigNum {
     /* prefix */
     BigNum & operator++() {
          // increment current state
          ...
          return *this;
     }
     /* postfix */
     BigNum operator++(int) {
          BigNum tmp(*this);
          ++(*this);
          return tmp;
     }
}
```

Перегрузка операторов ввода-ввода
Эти операторы нужно объявлять снаружи класса и объявлять дружественными.

```
struct Point2D { . . . };
std::istream& operator>>(std::istream& input, Point2D& point) {
     input >> point.x >> point.y;
     return input;
}
std::ostream& operator>>(std::ostream& output, Point2D const& point) {
     return output << point.x << ' ' << point.y;
}
```

Перегрузка оператора привидения
Можно переопределять двумя способами - разрешать приведения (и приводить) или запрещать с помощью explicit. Например, перегрузка оператора приведения к int будет выглядеть как operator int.

```
struct String {
     operator bool() const {
          return size;
     }
     explicit operator char const*();
private:
     char *data;
     unsigned int size;
```

```
};
```

Операторы с особым порядком вычисления
Компилятор автоматически оптимизирует вычисления - если результат уже получен и не может быть изменен ни в каком случае, дальше вычисления не идут (например, в логических операторах && и ||). Если возникает необходимость вычисления всего, нужно использовать & и |. Но если такие методы переопределяются, то оптимизации не происходит и для && вычисляются обе части.

Важные аспекты перегрузки

- Для читаемости кода следует придерживаться стандартной семантики
- Необходимо учитывать приоритет операторов. Порядок операторов задан стандартом языка и не может быть переопределен.
- Хотя бы один из параметров должен быть пользовательским.

Шаблоны

Шаблоны (template) в C++ - средство, предназначенное для написания алгоритмов без привязки к типам данных.
Шаблон функции выглядит следующим образом:

```
template <typename T>
void func (T* ptr, T& link, T obj);
int a = 42;
func<int>(a*, a, a);
char c = 'h';
func<char>(c*, c, c)
```

В этом случае func принимает на вход переменную не какого-то конкретного одного типа, а того, что указан в шаблоне. Использование шаблонов полезно для реализации некоторых обобщенных алгоритмов - например, сортировки.

Может возникнуть необходимость и в создании **шаблонов класса**. Это полезно, например, при создании контейнеров:

```
template <class Type>
struct node
{
     Type value;
     node* next;
     node* prev;
     /*...*/
};
template <class Type>
class list
{
public:
     typedef Type value_type;
     typedef list_iterator<Type> iterator;
     iterator insert(iterator pos, const Type& value){/*...*/}
     iterator erase(iterator pos) {/*...*/}
     /*...*/
private:
```

```
    node<Type>* m_head;
    node<Type>* m_tail;
};
```

Вывод (deduction) аргументов шаблона

В некоторых случаях компилятор способен "догадаться", какого типа объекты используются в функции и без явного указания шаблона. Если в функции один параметр, что компилятор способен провести дедукцию для одного аргумента. Если их несколько, то дедукция проводится для каждого аргумента отдельно.
Например, при вызове функции

```
int m = std::max(22, 54);
```

Очевидно, что вызывается именно `std::max<int>`.

В процессе дедукции компилятор будет производить возможные преобразования:

- Преобразования для lvalue: перевод в rvalue, элемент массива - в указатель на элемент, функцию - в указатель на функцию
- Добавление const
- Преобразования в рамках иерархии классов

Специализация шаблона

В C++ возможно создавать функции, специализирующиеся на работе с каким-то конкретным шаблоном.
Например, если нужно отсортировать массив, то можно использовать алгоритм быстрой сортировки. Но для типа char существует более эффективный алгоритм.

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
// code to implement Quick Sort
}
// Template Specialization: A function
// specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
// code to implement counting sort
}
```

X. Умные указатели

Monday, April 16, 2018 14:34

Умные указатели

Умный указатель - объект, с которым можно работать также, как с обычным указателем, но предоставляющий дополнительный функционал - например, осуществлять контроль за освобождением памяти.
Умные указатели удобны для борьбы с утечками памяти при возникновении исключений. Умный указатель в своём деструкторе освобождает выделенную память.

Простейший умный указатель:

```
template <typename T>
class SmartPointer {
     T* object
public:
     SmartPointer(T *obj) : object(obj);
     ~SmartPointer(){
          delete object;
     }
     T* get() {
          return object;
     }
}
```

Хотя с этим классом нельзя работать, как с указателем, он годится для того, чтобы удалять объект при выходе из локальной области видимости.

Примечание
Библиотека **Boost** предоставляет продвинутые возможности и базовые инструменты, отсутствующие в std. Понравившиеся сообществу методы переезжают в std.

boost::scoped_ptr

В Boost был метод boost::scoped_ptr, по функционалу схожий с вышеописанным примером. Этот указатель нельзя было присвоить, и это гарантирует корректное освобождение памяти. Единственное его предназначение - очистка ресурсов. Также в нём был operator* для работы с классом как с указателем.

```
template <typename T>
class ScopedPtr {
public:
     ScopedPtr(T *object) : object(object){}
     ~ScopedPtr() {
          delete object;
     }
     T* get() {
```

```
        return object;
    }
    T& operator*() {
        return *object;
    }
private:
    T *object;
    ScopedPtr(const ScopedPtr& p) = delete;
    const ScopedPtr& operator=(const ScopedPtr& p) = delete;
};
```

std::auto_ptr

Введен в C++98, помечен deprecated в C++11.
Имеет операторы "копирования" и присваивания. При копировании копируемый объект освобождается. Очевидный минус - нельзя использовать в контейнерах, т.к. они построены на копировании.

std::unique_ptr

Пришел на смену auto_ptr. В отличие от последнего, запрещает копирование, но имеет метод передачи прав на владение через std::move.

```
std::unique_ptr<int> ptr1(new int(42));
std::unique_ptr<int> ptr2;
ptr1 = ptr2; // Ошибка компиляции
std::unique_ptr<int> ptr3(ptr1); // Ошибка компиляции
ptr2 = std::move(ptr1); //Скомпилируется
```

std::shared_ptr

Самый популярный на данный момент.
Это умный указатель с подсчётом ссылок на ресурс. Так, если создать 2 unique_ptr и связать их с одним ресурсом, при освобождении будет ошибка. Shared_ptr позволяет избежать этого - ресурс освобождается, когда счётчик достигает 0.
Некоторые методы:

- explicit shared_ptr(Y* ptr) - берёт неуправляемый указатель ptr под автоматическое управление. Тип Y должен быть полностью определён и неявно преобразовываться в T.
- ~shared_ptr() – уничтожает объект, если им не владеет больше ни один shared_ptr, иначе ничего не делает
- shared_ptr(const shared_ptr& r)
  template< class Y > shared_ptr(const shared_ptr<Y>& r)
  - создает shared_ptr, который разделяет право собственности на объект, управляемый r. Если r не управляет объектом, *this тоже не управляет объектом.
- shared_ptr& operator=(const shared_ptr& r)
  template< class Y > shared_ptr& operator=(const shared_ptr<Y>& r)
  - разделяет права собственности на объект, управляемый r. Если r ничем не управляет, то *this тоже.
- operator bool() const – проверят, управляет ли *this объектом.
- T* get() const – возвращает указатель на управляемый объект.

- long use_count() const – возвращает количество shared_ptr, которые управляют объектом

Пример

```
std::shared_ptr<int> ptr1(new int(10));
std::shared_ptr<int> ptr2(new int(12));
ptr2 = ptr1;
```

Здесь после выполнения ptr1 будет пуст, а в ptr2 будет счётчик, равный 2. При вызове метода ptr2.reset() счётчик снова станет равным 1, и хранимое значение вернется к 10. При вызове его ещё раз обе переменные int будут очищены.

```
std::shared_ptr<MyClass> ptr(new MyClass());
MyClass* rawPtr = ptr.get();
ptr.reset();
ptr.get()->method();
```

Здесь последний метод вызовет ошибку, т.к. после reset() со счётчиком 0 get() вернет nullptr;

При создании умного указателя "на лету" может быть ошибка:

```
method(std::shared_ptr<MyClass>(new MyClass), getRandomKey());
```

При таком вызове память под shared_ptr будет очищена раньше, чем нужно.

std::weak_ptr

- умный указатель, который содержит "слабую" ссылку на объект, управляемый std::shared_ptr. При этом, чтобы получить доступ к управляемому объекту, нужно привести std::weak_ptr к std::shared_ptr. std::weak_ptr используется в следующем случае: к объекту нужно получить доступ, если он существует. Метод lock() создает временный std::shared_ptr, владеющий объектом. Это может быть использовано для разрешения циклических зависимостей из std::shared_ptr.

```
struct Bar;
struct Foo {
    Foo() { cout << "Foo()" << endl; }
    ~Foo() { cout << "~Foo()" << endl; }
    std::shared_ptr<Bar> bar; //Умный указатель на Bar
};
struct Bar {
    Bar() { cout << "Bar()" << endl; }
    ~Bar() { cout << "~Bar()" << endl; }
    std::shared_ptr<Foo> foo; //Умный указатель на Foo
};
int main() {
    //Создается объект Foo и помещается в shared_ptr
    std::shared_ptr<Foo> fooPtr = std::make_shared<Foo>();
    //Создается объект Bar, оборачивается и помещается в Foo
    foo->bar = std::make_shared<Bar>();
    //А в этот Bar помещается указатель на уже существующий Foo
    foo->bar->foo = fooPtr;
}
```

В вышеприведённом коде получится циклическая зависимость двух std::shared_ptr друг от друга. Чтобы этого избежать, можно использовать std::weak_ptr

```
std::shared_ptr<Foo> fooSharedPtr = std::make_shared<Foo>();
std::weak_ptr<Foo> fooWeakPtr(fooSharedPtr);
if (std::shared_ptr<Foo> ptr = fooWeakPtr.lock()) {
     ptr->method();
}
```

Умные указатели на массивы

Вышеописанные указатели будут некорректно работать с массивами - в указатель обернется лишь первый элемент и будет вызван delete вместо delete[]. Впрочем, массивы поддерживает unique_ptr и указатель в библиотеке boost (boost::shared_array). Последние не введены в std, так как существует vector

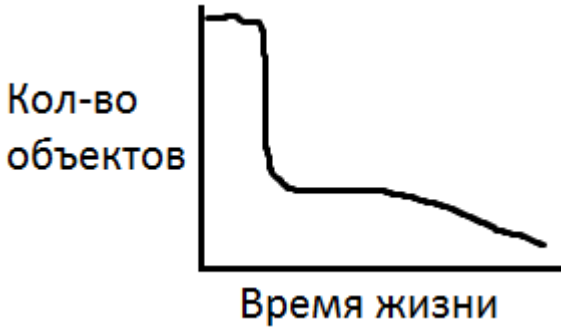
Сборка мусора

Мусор - ненужные объекты, которые можно почистить.
В Qt реализован такой механизм: QObject'ы образуют иерархию. Удаление высшего элемента запускает деструкторы для нижних.

Сборка мусора в Java/C#

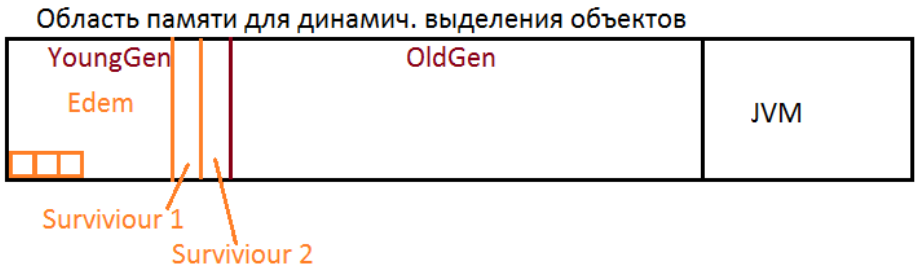
В Java, как и в C++, выделение памяти происходит с помощью оператора new. Но если в C++ необходимо следить за всем вручную, то в Java оператор new *запрашивает* память у JVM и освобождает, когда нужно. При запуске JVM запрашивает у ОС какое-то количество памяти, и из этого количества дает выполняемой программе.

Гипотеза о поколениях



Большинство объектов использует малое количество времени - обычно внутри одного метода - и лишь немногие живут долгое время. Таким образом, предпочтительно эффективнее следить за короткоживущими объектами в левой части графика, так как их гораздо больше.

Принцип работы



Когда приложению не хватает памяти, программа останавливается.
Запускается *MinorGC (Garbage Collector)*
Сборщик проходит по объектам в YoungGen и проверяет, есть ли ссылки на них в программе. Выжившие объекты помещаются в Survivior 1, а Edem очищается.
При следующей итерации проверяются и Edem, и Survivor 1, и выжившие оттуда помещаются в Survivor 2. Survivor 1 и Edem очищаются. Таким образом, при каждой итерации для избежания фрагментации Survivor 1 и 2 меняются местами.
Долго выживающие элементы при нехватке памяти переезжают в OldGen.
Когда OldGen заполняется, запускается *MajorGC*, очищающий OldGen и проводящий дефрагментацию.
Проблема в том, что программист не знает время отклика программы - сборщик может в любой момент прекратить процесс

XI. Системы контроля версий. Кодировка

Monday, April 23, 2018 13:53

Системы контроля версий

- Централизованные (CVS, Subversion) - есть центральный сервер, к которому подключаются клиентские приложения.
  - Первое обращение к серверу - установка клиента и запрос **checkout**'а.
  - После работы делается закрепление (**commit**). Клиент анализирует состояние рабочей директории и отправляет на сервер. Сервер анализирует изменения и сохраняет изменение.
  - После этого остальные разработчики делают checkout и синхронизируются.

  Недостаток в том, что для работы с этой системой необходимо подключение к серверу даже для простейших операций. Кроме того, центральный сервер является уязвимым местом.
- Децентрализованные (Git) - каждая копия является полноценным репозиторием, не нуждающимся в других копиях.
  Git был разработан Линусом Торвальдсом при разработке Linux. На данный момент является стандартным для большинства проектов.
  GitHub - сервис, представляющий возможность создания репозитория на Git или Mercurial.

Работа с Git

- `Git init` - создание пустого репозитория. В папке создается директория .git, которая и будет является репозиторием. Все, что находится на том же уровне, является **working directory.**
  Создается начальная ветка **master**
  - GitHub создаст файл `readme.md` (html-подобный формат)
  - `.gitignore` - содержит маски файлов, которые не отслеживаются Git'ом.
    В репозитории стоит хранить:
    - Исходные коды проекта

    И не стоит:
    - Бинарные файлы и прочие файлы, генерируемые системой из исходных кодов.
    - Файлы конфигурации IDE
- `Git status` - показывает статус репозитория и текущую ветку
  - Красный - изменённые файлы по сравнению предыдущего
  - Зеленый - добавленные файлы (индексируемые)
- `Git add <files>` - добавляет файлы в репозиторий
- `Git rm <file>` - удаление файла из репозитория. В следующем коммите файл будет удален.
  - `Git rm <file> --cached-`

- `Git commit -m <message>` - закрепление.
  В репозитории есть история коммитов, каждый коммит имеет ссылку на родительского. **Ветка** является лишь указателем на коммит.
  - `Git commit --amend` - добавить изменения к предыдущему коммиту
- `Git log` - лог
- `Git checkout <file>` - вернет файл к версии из предыдущего коммита
- `Git revert ^-<number>` - создает новый коммит, отменяющий все изменения указанного числа коммитов
  Возможно указать `Git revert HEAD-<number>`
- `Git reset ^-<number>` - перемещает указатель на указанный коммит. Если установить флаг -h, то вся информация о следующих коммитах пропадет.
- `Git branch` - вывод всех веток репозитория.
  - `Git branch <branchname>` - создание ветки. При переходе на другую ветку содержимое директории синхронизируется с веткой. Незафиксированные изменения могут войти в конфликт.
  - `Git branch rem <branchname>` - удаление ветки.
- `Git checkout -b <branchname>` - создание ветки и переход на неё
- `Git stash` - сохраняет изменения в текущей ветке без создания коммита в специальном стеке. Это полезно, если есть необходимость переключиться на другую ветку, не создавая коммит для текущей.
  - `Git stash apply` - вернуть ветку в состояние, сохранённое в стеке. Можно взять индекс.
  - `Git stash drop` - снять последний элемент со стека
  - `Git stash pop` - две последних команды вместе.
- `Git merge <branchname>` - соединяет текущую ветку и указанную. При этом создается новый коммит, содержащий все коммиты из указанной ветки. После успешного выполнения указанную ветку можно удалить.
- `Git rebase <branchname>` - вычисляются и применяются все изменения относительно указанной ветки
- Если в двух ветках появляется пересечение изменений, происходит **конфликт**. Для разрешения конфликта используется:
  `Git merge conflicts` - создает файл, показывающий конфликты между ветками. После этого можно будет указать, как будет выглядеть итоговый файл.

Работа с удалёнными репозиторием

- `Git clone` - забрать полную копию удалённого репозитория. Связывает удаленный репозиторий с текущим.
- `Git remote -v` - показывает, сколько Git знает удалённых

репозиториев
- `Git remote add <rep_name> <adress>` - создает ссылку на удалённый репозиторий, обозначая её указанным именем. По умолчанию имя - origin
- `Git fetch` - вытаскивает информацию у всех удалённых репозиториев. Можно указать и конкретное имя и ветку через / После применения команды создает в локальном репозитории ветки, хранящую все изменения, находящиеся в новом репозитории. Затем можно слить ветки с помощью `merge`.
- `Git pull <rep_name>/<remote_branch>` - делает `fetch` и `merge`
- `Git push <rep_name> <branch_name>` - загружает ветку на удалённый репозиторий, если имеются права доступа

Работа с GitHub
После создания репозитория на github он будет выглядеть также, как и после git init. После этого нужно связать удаленный репозиторий с локальным - либо склонировать себе на машину с помощью clone, либо соединить с помощью git remote add.
Для того, чтобы делать push на GitHub, нужны права на изменение. Если такой возможности нет, можно сделать **fork**. Тогда будет создана копия репозитория, прикрепленная к текущему аккаунту. В этот репозиторий будет гарантирована возможность внесения изменений.
Чтобы связать fork с основным репозиторием, нужно создать **pull request**.

Continuous integration (CI) - система, которая при внесении изменений в код на сервер его автоматически собирает и запускает ряд тестов. Для того, чтобы реализовать это с помощью GitHub, можно установить посылку запросов на нужный сервер при принятии pull request'a. На данный момент, одна из самых распространённых подобных систем - **Jenkins**
Continuous Delivery (CD) - автоматическая доставка результатов CI в конечное окружение.

Issue - "задачи" в GitHub. Задачей может быть исполнение ошибок, добавление новых функций и т.д. Обычно ветки разработчиков именуются согласно нужной issue.

Кодировка

Кодировка - способ представления информации в компьютере. Состоит из двух частей:

- Таблица символов - все символы, которые поддерживает данная кодировка. Каждому символу сопоставляется число в таблице. Например, в таблице Unicode за последние 10 лет появилось очень много символов, и 4-х байт для символа стало не хватать. Поэтому

были придуманы различные обходные пути - например, некоторые символы в Unicode можно кодировать парой.

- Двоичное представление числа.
  Если используется расширенная таблица Unicode, то для одного символа нужно выделить 8 байт - это очень много. Однако, если известно, что все используемые символы будут лежать в первых строках таблицы, можно ограничить число используемых байт.

Семейство кодировок **UTF** - кодировки с динамической длиной кодируемого символа. В начале байта лежит маркер, указывающий, сколько байт отведено под символ. Это позволяет экономить память - так, для кодирования первых 127 символов нужен всего 1 байт.
Кодировки **UTF-16 LE** и **UTF-16 BE** различаются тем, какой байт старший. Для **UTF-8** это не имеет смысла, так как байт один.

XII. Порождающие паттерны

Monday, May 7, 2018 14:51

SOLID - набор из 5 принципов:

- **Single Responsibility Principle** - каждая сущность должна отвечать за одну задачу
- **Open/Close Principle** - сущности должны быть закрыты для изменения, но открыты для расширения
- **Liskov Substitution Principle** - функции, которые используют базовый тип, должны быть способны использовать подтипы базового типа, не зная об этом. Поэтому нельзя создавать дочерний класс, который изменит поведение методов интерфейса базового класса.
- **Interface Segregation Principle** - разделение интерфейсов по мере возможностей. Должно быть много интерфейсов, описывающих различные функции, а не один, реализующий всё.
- **Dependence Inversion Principle** - если система разбита на уровни, то модули верхних уровней не должны зависеть от нижних уровней.

Эти пять принципы позволяют создавать красивый код, разбитый на маленькие куски, слабо связанные между собой

DRY - Don't Repeat Yourself. Каждая часть системы должна иметь единственное непротиворечивое осуществление в рамках системы.
WET - We Enjoy Typing - в системе много дублирующегося кода.

KISS - Keep It Simple Stupid - системы работают лучше, если они остаются простыми. Внесение изменений в сложную систему означает риск цепной реакции по всей системе. Не нужно использовать более сложные средства, чем необходимо

YAGNI - You Aren't Gonna Need It - принцип, декларирующий отказ от избыточной функциональности.

Шаблоны проектирования (Design Patterns)

Шаблоны проектирования - часто встречающееся решение определённой проблемы при проектировании архитектуры. Если алгоритм - чётки набор действий, то паттерн - это высокоуровневое описание, описывающее только общий подход.
Паттерн - наиболее часто встречающееся решение, признанное типовым.

Обычно паттерн состоит из:

- Мотивации к решению проблемы данным способом
- Структуры классов (обычно в виде UML-диаграммы)
- Примера на одном из ЯП или на псевдокоде

- Особенностей реализации в различных контекстах
- Связи с другими паттернами

GOF book - первая книга по программированию, содержащая паттерны.

Преимущества паттернов:

- Паттерн - проверенное решение. Его реализация занимает меньше времени и более эффективна.
- Стандартизация кода
- Общий программистский словарь

Критика паттернов:

- Нужда в паттернах появляется тогда, когда используется язык с недостаточным уровнем абстракции. В современных языках многие паттерны доступны "из коробки".
  Тем не менее, зачастую переход на более высокоуровневый язык невозможен.
- Стандартизация подхода может привести к неэффективному решению
- Паттерны не всегда стоит использовать

Классификация паттернов

Паттерны отличаются по уровню сложности, детализации и по охвату используемой системы

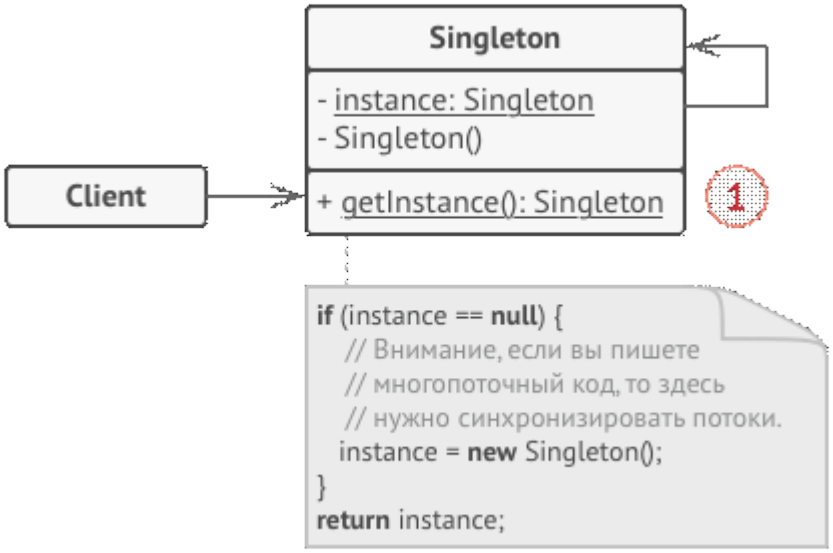
- **Идиомы** - самые низкоуровневые и простые программы
- **Архитектурные** паттерны - самые универсальные
  - **Порождающие** паттерны - отвечают за получение новых объектов
  - **Структурные** - описывают, на какие компоненты можно разбить структуры и как обеспечить связь между ними
  - **Поведенческие** - описывают архитектуру в динамике - коммуникацию между объектами

Порождающие паттерны

Singleton

Решает две **проблемы**: гарантирует наличие единственного экземпляра класса и предоставляет глобальную точку доступа к объекту.

Решение такое - скрыть конструктор и сделать его приватным, при том создать публичный статический метод



Т.е. все способы создания класса скрыты, но доступен метод, который при первом запуске инициализирует объект, а затем - возвращает его

Применимость:

- В программе должен быть единственный экземпляр какого-то класса, доступного всем клиентам
- Необходимо иметь больше контроля над глобальными переменными

Преимущества:

- Гарантирует единственность экземпляр класса
- Предоставляет глобальную точку доступа
- Реализует отложенную инициализацию объекта-singleton

Недостатки:

- Нарушает принцип единственной ответственности
- Маскирует плохой дизайн
- Проблема многопоточности
- Сложно подменить объект > Сложная реализация юнит-тестов
- Требует постоянного создания Mock-объектов

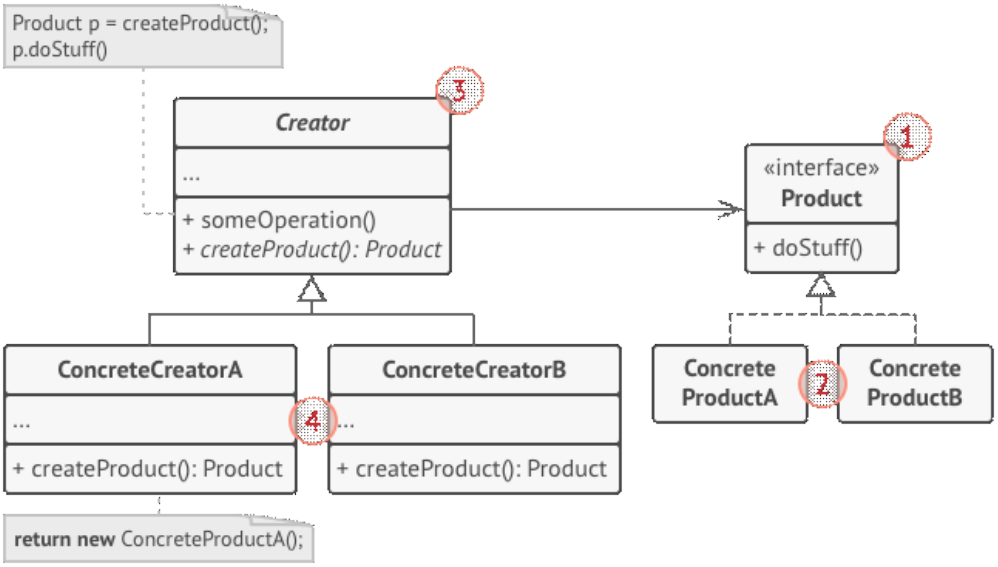
Шаги реализации:

1. Добавить в класс приватное статическое поле, которое будет содержать одиночный объект
2. Объявить статический создающий метод, который будет использоваться для получения одиночки
3. Добавить ленивую инициализацию
4. Сделать конструктор класса приватным
5. Заменить в клиентском коде вызовы конструктора вызовами создающего метода

Factory method

- порождающий паттерн, который определяет общий интерфейс создания объектов в суперклассе. Решает **проблему** завязанности приложения на конкретный класс.

Решение - создавать объекты не напрямую, с использованием оператора new, а завести новый "фабричный" метод, который этим занимается.



Чтобы изменить тип создаваемого продукта, можно переопределить фабричный метод в подклассе. При этом все возвращаемые объекты должны иметь общий интерфейс.
Клиент метода будет работать через один абстрактный интерфейс.

Реализация:

1. Привести все создаваемые продукты к общему интерфейсу
2. В классе, который производит продукты, создать фабричный метод, возвращающий этот интерфейс
3. Везде, где явно вызывался метод new, вызвать новый фабричный метод. В фабричный метод, возможно, придется добавить параметры, отвечающие за класс создаваемого объекта
4. Для каждого конкретного типа продукта завести подкласс, переопределить в них фабричные методы, которые будут возвращать именно этот класс
5. Если создаваемых продуктов много, можно подумать о введении параметров для выбора типа

Применимость:

- Когда заранее неизвестны типы и зависимости объектов, с которым должен работать код
- Есть необходимость расширения создаваемого фреймворка или библиотеки
- Экономия системных ресурсов

Преимущества:

- Избавляет класс от привязки к конкретным типам продуктов
- Выделяет код производства продуктов в одно место, упрощая поддержание кода
- Упрощает добавление новых продуктов программу

- Реализует принцип открытости/закрытости

Недостатки:

- Может привести к созданию больших параллельных иерархий классов

Abstract factory

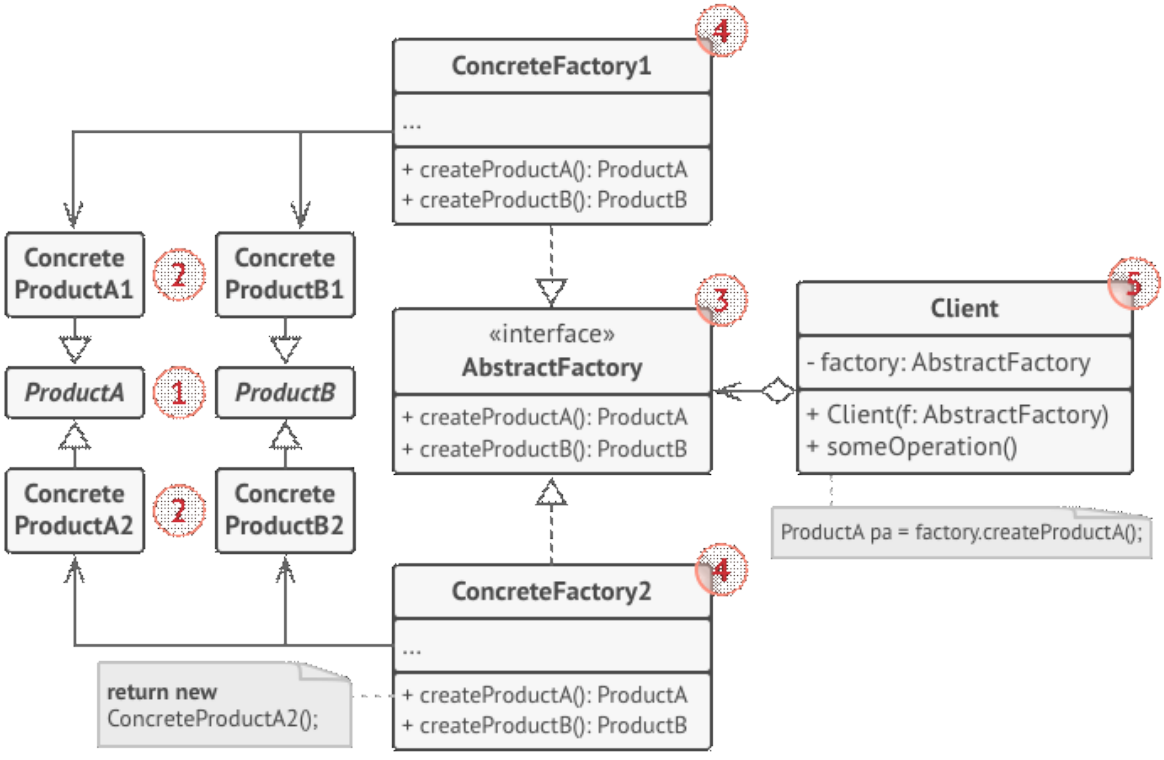
- порождающий паттерн, который позволяет создавать семейство зависимых классов.

Проблема - необходимость создания групп объектов в семействе классов, отличающихся по разным признакам

Решение - выделить общий интерфейс, после чего использовать абстрактную фабрику для создания семейства объектов.

После выделения интерфейсов для всех типов объектов создается абстрактная фабрика. Методы внутри фабрики будут возвращать эти интерфейсы.

Унаследованные фабрики будут создавать конкретные объекты



Применимость:

- Логика программы должна работать с видами связанных друг с другом продуктов, не завися от конкретных классов продуктов
- Когда в программе уже есть фабричный метод, но очередные изменения предполагают введение новых типов продуктов

Реализация

1. Создать таблицу соотношения типов продуктов
2. Свести все вариации продуктов в общий интерфейс
3. Определить интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов

4. Создать классы конкретных фабрик
5. Изменить код инициализации программы так, чтобы она создавала определённую фабрику и передавала в клиентский код
6. Заменить в клиентском коде участки создания продуктов через конструктор соответствующими методами фабрики

Преимущества:

- Избавляет класс от привязанности к конкретным классам продуктов
- Гарантирует сочетаемость создаваемых продуктов
- Избавляет клиентский код от привязанности к конкретным классам
- Выделяет код производства в одно место, упрощая код
- Упрощает добавление новые объектов в программу
- Реализует принцип открытости/закрытости

Недостатки:

- Для корректной работы нужно добавить конструктор нового объекта в каждую фабрику, что сложно, если объектов много
- Усложняет код программы

Различия фабрик

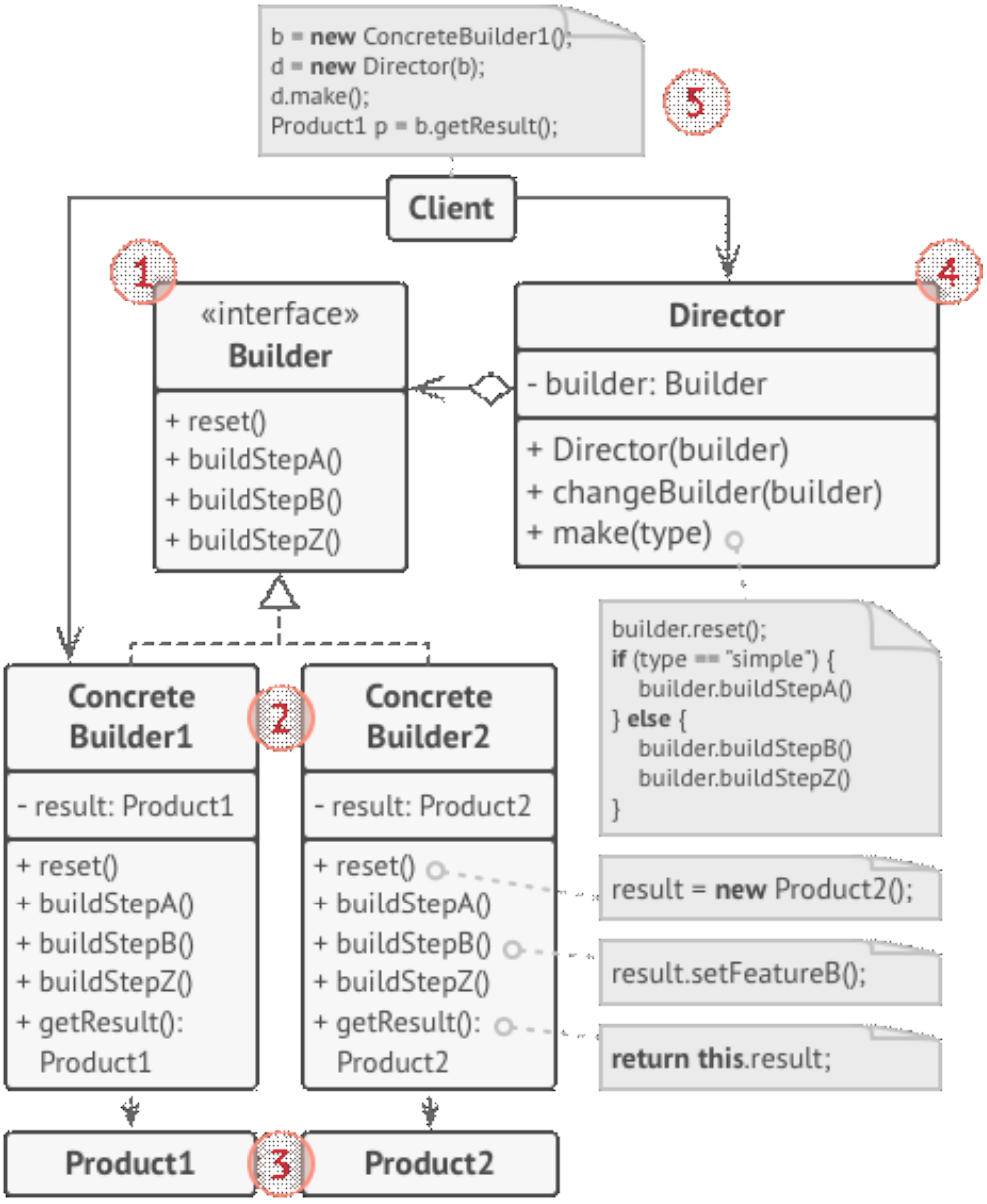
Фабрика - общая концепция проектирования, когда какая-то одна часть программы отвечает за создание объектов в другой части программы. Это может быть класс, создающий методы или статический метод
Создающий метод - метод-обёртка над вызовом конструктора. Это изолирует любые изменения в конструировании от прочего кода.
Статический фабричный метод - вариация создающего метода, когда сам метод объявлен статичным. Это может быть полезно, если нужно использовать готовые объекты вместо создания новых.
Простая фабрика - класс, в котором есть один метод с большим оператором, выбирающий создаваемый продукт**.** У простой фабрики подклассов нет.
Фабричный метод - устройство классов, при котором подклассы могут переопределять тип создаваемых в классе продуктов
Абстрактная фабрика - устройство классов, облегчающее создание семейства продуктов.

Builder

- порождающий паттерн, позволяющий создавать сложные методы пошагово.
Проблема - сложные объект, требующий кропотливой пошаговой реализации со множеством вложенных полей. Наследование классов может вырасти в экспоненциальное число операций, передавать множество параметров неудобно с точки зрения клиентского кода.

Решение - создать отдельный **класс-строитель**, который будет знать, как создавать объекты. Чтобы создать объект, нужно создать строителя, затем

вызвать нужные методы строителя, чтобы получить нужный класс. Это удобно, так как один шаг строительства может различаться для различных объектов.
Значит, нужно выделить интерфейс строителя и определить конкретные реализации.
Директор - задает порядок шагов строительства. Может быть полезен, если есть несколько способов конструирования объектов, различающихся последовательностью шагов.



Применимость:

- Избавление от "телескопического конструктора"
- Код должен создавать различные представления одного объекта
- Нужно собирать сложные составные объекты

Реализация:

1. Убедиться, что создание разных представлений объекта можно свести к общим шагам
2. Описать эти шаги в общем интерфейсе строителя

3. Для каждого представления объекта-продукта создать по одному классу-строителю и реализовать его методы
4. Возможно, создать класс-директор

Преимущества:

- Вызов метод по имени позволяет определить свойства создаваемого объект
- Позволяет создавать продукты пошагово
- Позволяет использовать один и тот же код для разных продуктов
- Использует сложный код сборки продуктов

Недостатки:

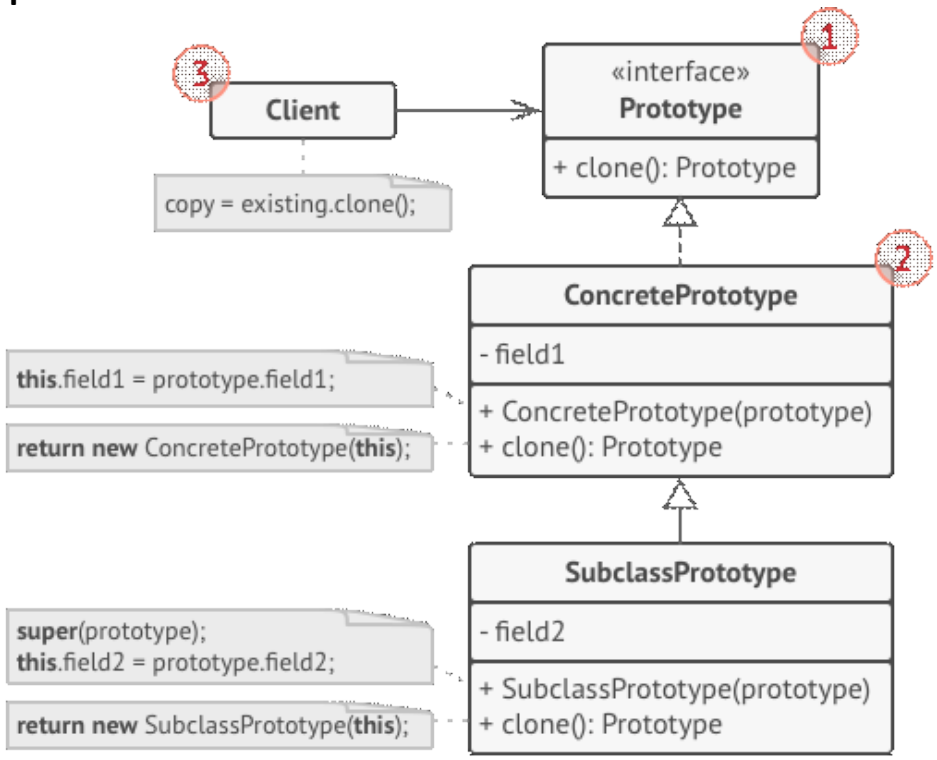
- Усложняет код программы из-за введения дополнительных классов
- Привязка к конкретным классам строителей

Prototype

- порождающий паттерн, который позволяет копировать объекты, не вдаваясь в подробности их реализации

Проблема - наличие объекта, который нужно скопировать. При копировании возможны проблемы с приватными полями объекта или с незнанием конкретной реализации класса.

Решение - поручение создания копий самим копируемым объектам. Метод копирования - обычно называется clone - создает новый объект текущего класса и копирует в него значения всех полей собственного объекта, в том числе и приватные. Копируемый объект называется **прототипом.**



Применимость:

- Когда код не должен зависеть от классов копируемых объектов
- Если есть множество подклассов, которые отличаются начальными

значениями полей

Реализация:

- Создать интерфейс прототипа с методом clone
- Добавить в классы прототипов конструктор копирования
- Создать метод клонирования, возвращающий новый объект, созданный из текущего
- Возможно создать центральное хранилище прототипов

Преимущества:

- Позволяет клонировать объекты, не привязываясь к конкретным классам
- Меньше повторяющегося кода инициализации объектов
- Ускоряет создание объектов
- Альтернатива созданию подклассов для конструирования особо сложных объектов

Недостатки:

- Сложности с копированием объектов, имеющими ссылки на другие объекты

XIII. Структурные паттерны

Wednesday, May 30, 2018 23:01

Структурные паттерны

- отвечают за построение удобных в поддержке иерархий классов

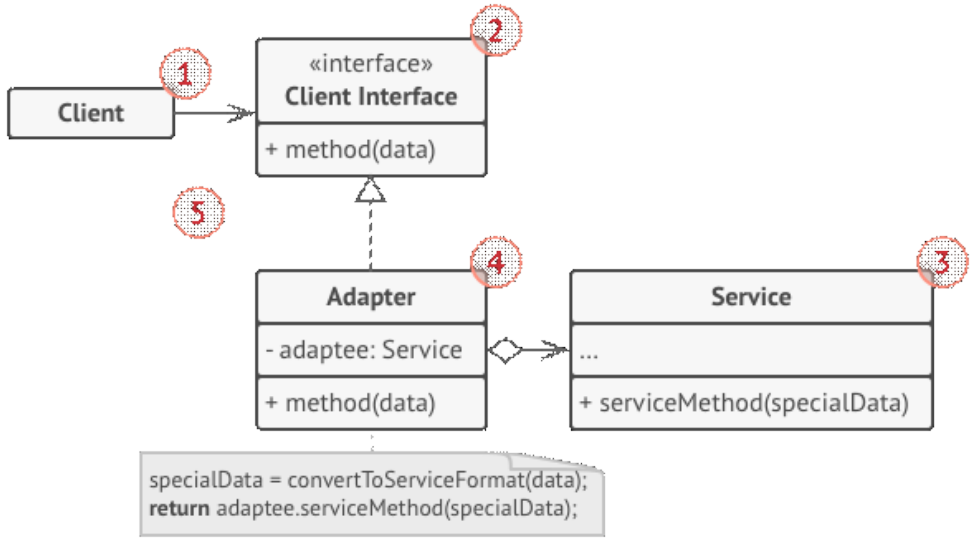
Adapter

- структурный паттерн, позволяющий объектам с несовместимыми интерфейсами работать вместе

Проблема - необходимость заставить программу или библиотеку работать с объектами, сходными по назначению, но различающимися по интерфейсу

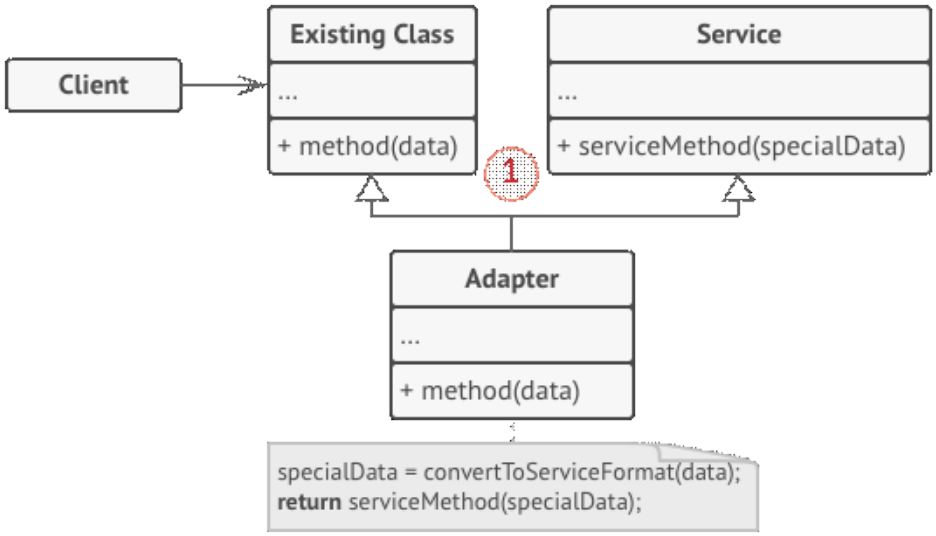
Решение - создать **адаптер** - объект-переводчик, который трансформирует вид нужного объекта в понятный для программы. При этом клиент может даже не знать о существовании адаптера.

Иногда бывает необходимо создание **двустороннего адаптера.**



Адаптер реализует клиентский интерфейс и содержит ссылку на адаптируемый объект.

Если язык поддерживает множественное наследование, то адаптер может унаследовать оба интерфейса одновременно



Применимость:

- Если есть необходимость в использовании стороннего класса, интерфейс которого не соответствует остальному коду приложения
- Если нужно использовать несколько подклассов, в которых не хватает общей функциональности, и расширить суперкласс невозможно.

Реализация:

1. Описать клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса
2. Создать класс адаптера, реализующий этот интерфейс
3. Поместить в адаптер поле, которое будет хранить ссылку на объект сервиса
4. Реализовать методы клиентского интерфейса в адаптере

Преимущества:

- Отделяет и скрывает от клиента подробности использования различных интерфейсов

Недостатки:

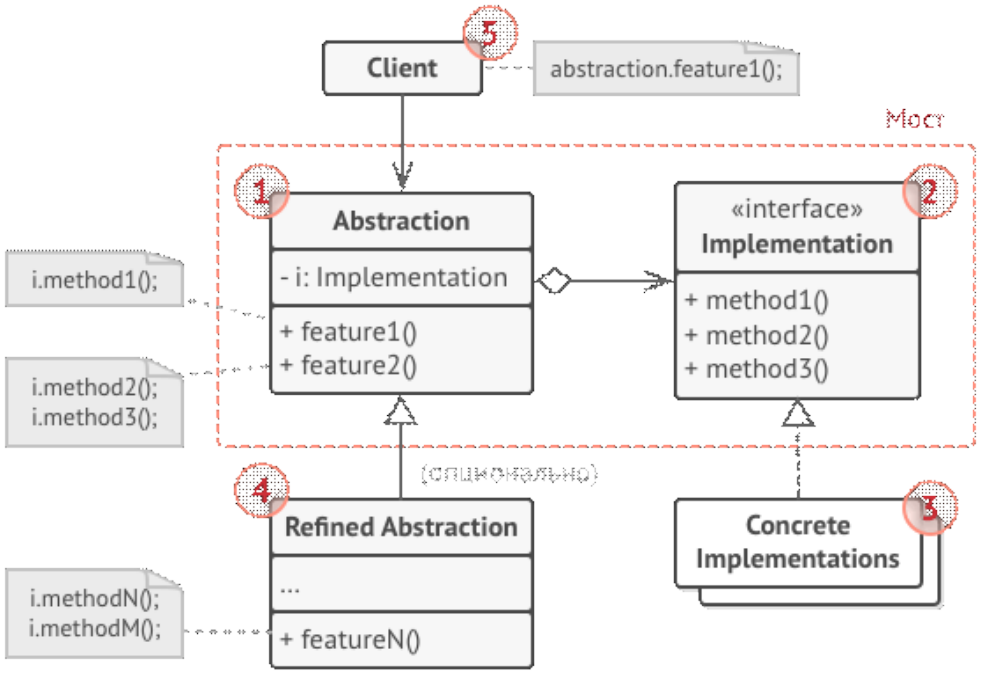
- Усложняет код программы из-за введения дополнительных классов

Bridge

- структурный паттерн, который разделяет один или несколько классов на две отдельные иерархии - абстракцию и реализацию, позволяя их изменять независимо друг от друга

Проблема - наличие иерархии по нескольким признакам. Введение каждого нового признака будет экспоненциально увеличивать потенциальное количество подклассов

Решение - заменить наследование делегированием. Нужно выделить одну из плоскостей иерархии в отдельный объект и ссылаться на него.



Абстракция содержит управляющую логику, делегирующую работу связанному объекту реализации

Применимость:

- Необходимость разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности
- Необходимость расширять класс в двух независимых плоскостях
- Необходимость изменять реализацию во время выполнения программы

Реализация:

1. Определить, какие операции будут нужны клиентам, и описать их в базовом классе абстракции
2. Определить поведения, доступные на всех платформах, и выделить из них ту часть, которая будет нужна абстракции. На основании этого описать общий интерфейс реализации
3. Для каждой платформы создать класс конкретной реализации
4. Добавить в класс абстракции ссылку на объект реализации

Преимущества:

1. Позволяет строить платформо-независимые программы
2. Скрывает лишние или опасные детали реализации от клиентского кода
3. Реализует принцип открытости/закрытости

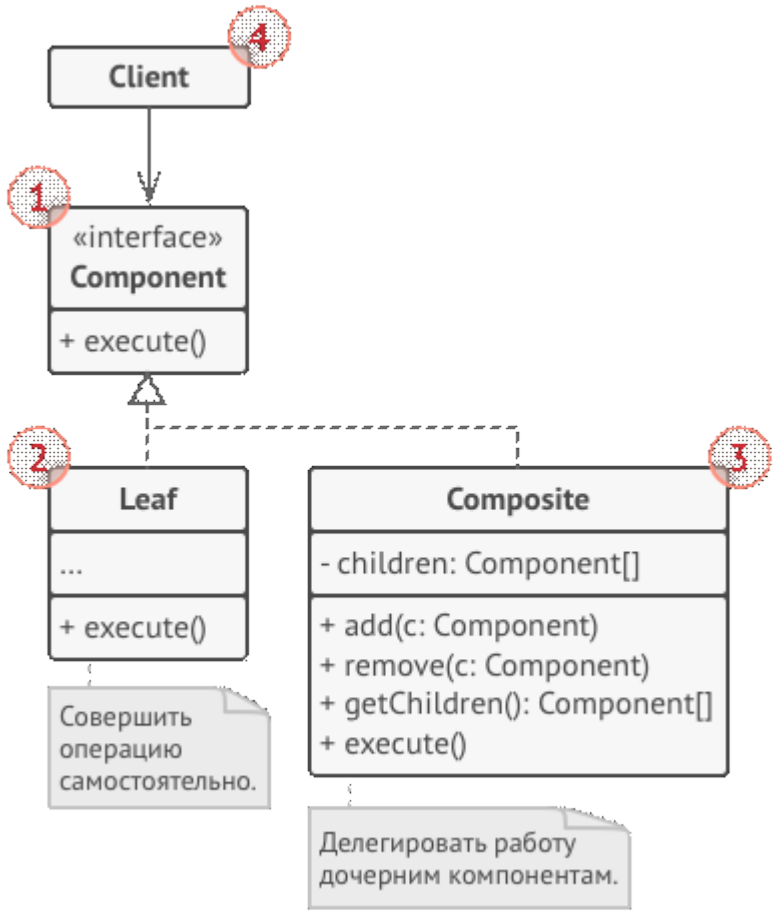
Недостатки

1. Усложняет код программы

Composite (Компоновщик)

- структурный паттерн, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней, как с

единичным объектом
Проблема - программа со сложной древовидной иерархией, состоящей из контейнеров и компонентов. Контейнер может хранить в себе как контейнеры, так и компоненты. Задача - произвести какое-то действие со всеми компонентами.
Решение - рассматривать контейнер и компонент как объект с единым интерфейсом с общим методом, производящим действие. Компонент применит его на себя, контейнер - применит на все объекты, хранящиеся в нём



Таким образом процедурой execute() будет запущено рекурсивное выполнение операции по всему дереву.
Составной элемент такого дерева и есть **компоновщик**. Он ничего не знает о типах дочерних компонентов, но это не проблема, так как все компоненты сведены к одному интерфейсу

Применимость:

- Необходимость представить древовидную структуру объектов
- Необходимость единообразной работы простых и составных компонентов

Реализация:

1. Создать общий интерфейс компонентов, который объединит операции контейнеров и простых компонентов
2. Создать класс компонентов-листьев

3. Создать класс компонентов-контейнеров
4. Добавить в класс компонентов-контейнеров операции добавления и удаления элементов

Преимущества:

- Упрощение архитектуры клиента при работе со сложным деревом компонентов
- Облегчение добавления новых видов компонентов

Недостатки:

- Слишком общий дизайн классов

Decorator

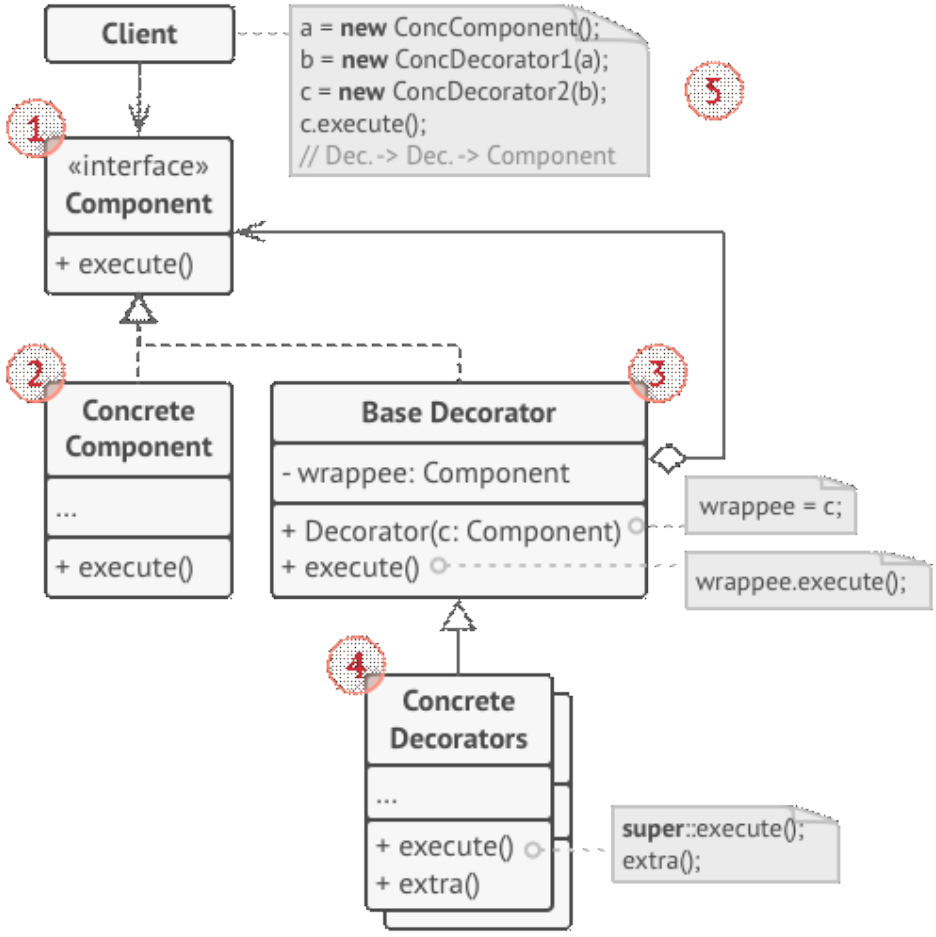
- структурный паттерн, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные обёртки

Проблема - наличие основного компонента и нескольких настроек над ним. Использование наследования при расширении функциональности сопряжено с двумя неприятностями:

- Невозможности изменить поведение уже существующего объекта
- Невозможность унаследовать поведение нескольких классов одновременно

Использование наследования приведет к необходимости создания большого количества подклассов - по одному для каждой комбинации компонент

Решение - добавить в один объект ссылку на другой, чтобы первый делегировал работу второму. Целевой объект помещается в объект-обёртку, который запускает базовое поведение объекта и добавляет что-то своё. Пользователю нет разницы, работает он с чистым объектом или с обёрнутым.



Применимость:

- Необходимость добавлять объектам новые обязанности на лету, незаметно для использующего их кода
- Невозможность расширить обязанности объекта с помощью наследования

Реализация:

1. Создать интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений
2. Создать класс конкретного компонента и поместить в него основную логику
3. Создать базовый класс декораторов, методы которого делегируют действие вложенному объекту
4. Создать классы конкретных декораторов, наследуя их от базового

Преимущества:

- Большая гибкость, чем у наследования
- Позволяет добавлять обязанности на лету
- Позволяет добавить несколько новых обязанностей сразу
- Позволяет иметь несколько мелких объектов вместо одного на все случаи жизни

Недостатки:

- Трудности с конфигурацией многократно обернутых классов

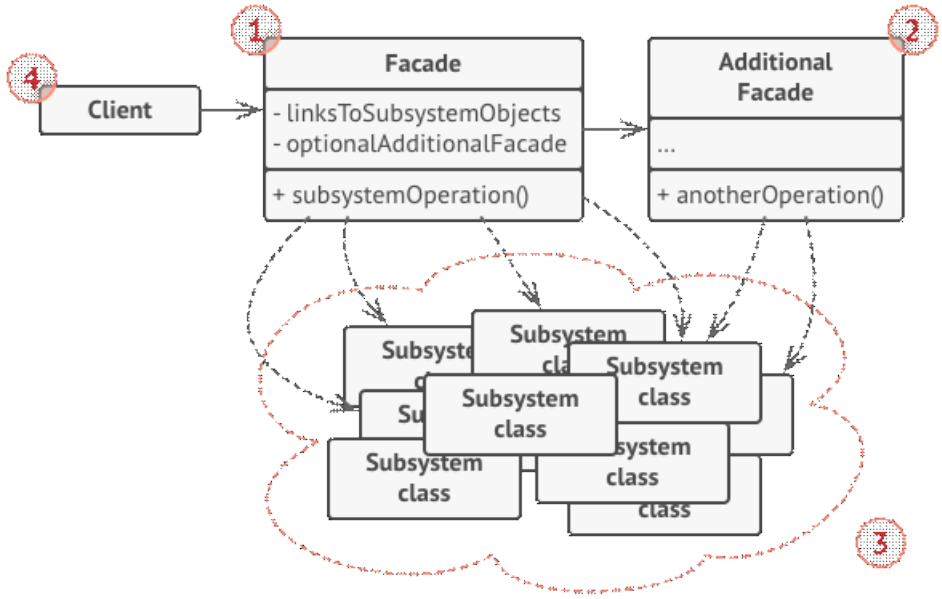
- Обилие крошечных классов

Façade (Фасад)

- структурный паттерн, предоставляющий простой интерфейс к сложной системе классов

Проблема - коду нужно работать с большим количеством объектов некой сложной библиотеки или фреймворка - нужно следить за инициализацией, правильным порядком зависимостей и т.д, в результате чего логика программы оказывается сильно связанной со сторонними классами.

Решение - создать простой интерфейс **- фасад** - для работы со сложной подсистемой. Это полезно, в частности, если нужна только часть функционала библиотеки.



Применимость:

- Необходимость предоставить простой или урезанный интерфейс к сложной подсистеме
- Необходимость разложить подсистему на отдельные слои

Реализация:

1. Создать класс фасада, реализующий более простой интерфейс, чем тот, что предоставляет система.
2. Возможно введение дополнительных фасадов при размытии ответственности

Преимущества:

- Изоляция клиентов от компонентов сложной подсистемы

Недостатки:

- Фасад рискует стать **божественным объектом** - объектом, который делает слишком много различных действий или хранит слишком много данных. Это противоречит Single Responsibility Principle

Flyweight (Легковес)

- паттерн, позволяющий вместить большее количество объектов в отведённую оперативную память

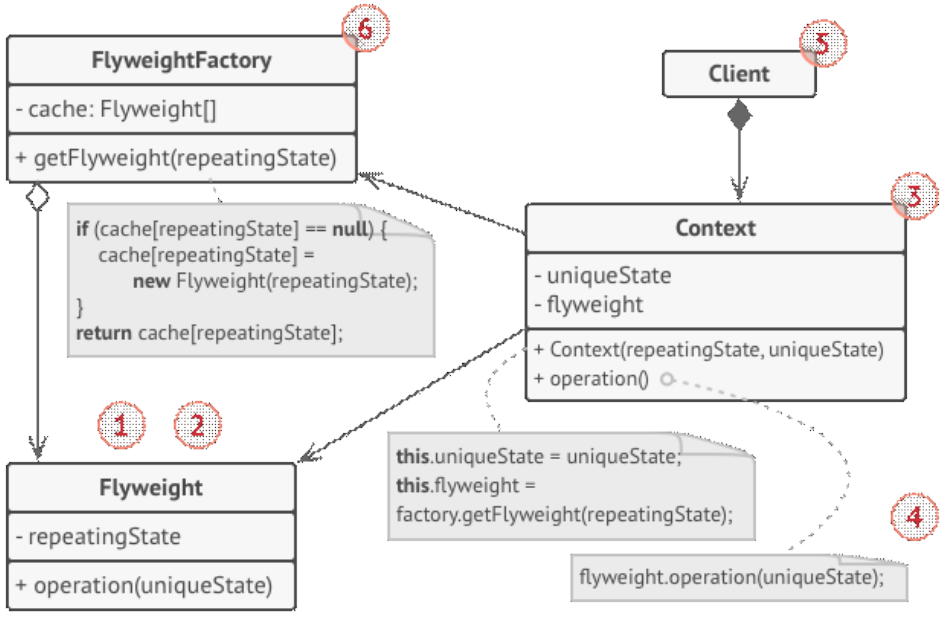
Проблема - наличие в оперативной памяти слишком большого одинаковых объектов

Решение - выделить неизменяемые данные объекта как **внутреннее состояние**, а остальные - как **внешнее состояние (контекст)**. Внешнее состояние стоит хранить не в классе, а передавать в методы через параметры. Таким образом, одни и те же объекты можно будет использовать в различных контекстах. Такой облегчённый объект и будет называться **легковесом**.

Внешние состояния при этом можно поместить в какой-либо контейнер или создать дополнительный класс-контекст, который бы связывал внешнее состояние с тем или иным легковесом.

Так как легковесы хранят только внутреннее состояние, они не должны иметь геттеров или сеттеров, и все параметры должны получаться через конструкторы.

Для удобства работы с контекстами и легковесами можно создать фабричный метод, принимающий на вход внутренние (можно и внешние) параметры. В таком случае, если нужный легковес существует - можно повторное его использовать, иначе создать новый.



Применимость:

- Нехватка оперативной памяти для поддержки всех нужных объектов

Реализация:

1. Разделить поля класса, который станет легковесом на две части: внутреннее состояние и внешнее состояние
2. Оставить поля внутреннего состояния в классе. Они должны инициализироваться только через конструктор.
3. Превратить поля внешнего состояния в параметры методов, где эти

поля использовались. Удалить поля из класса

4. Создать фабрику, которая будет хешировать и повторно отдавать уже созданные объекты. Клиент должен запрашивать из этой фабрики объект с определённым состоянием
5. Клиент должен вычислять или хранить значения внешнего состояния и передавать его в методы объекта легковеса.

Преимущества:

- Экономия оперативной памяти

Недостатки:

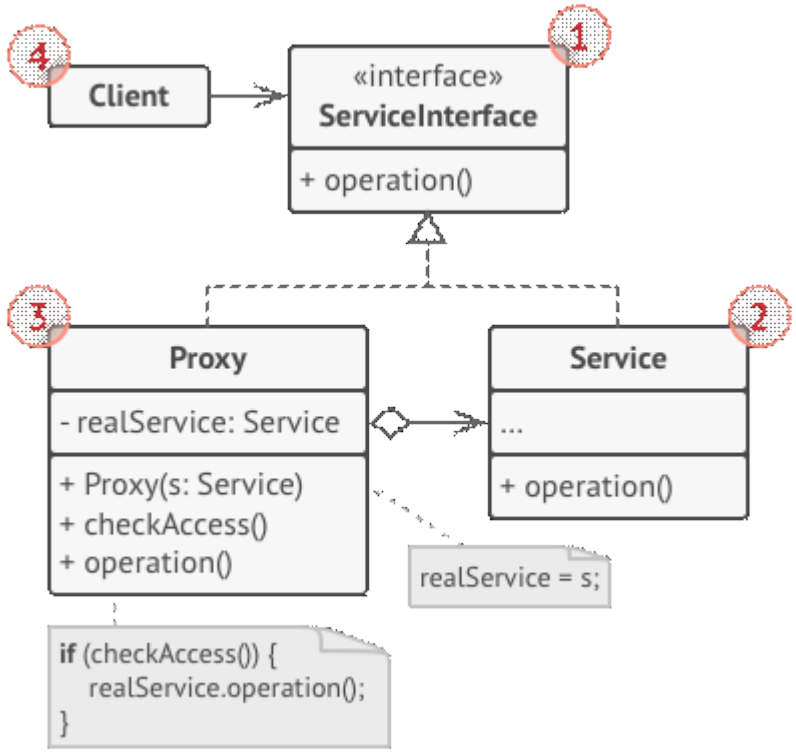
- Расход процессорного времени на поиск/вычисление контекста
- Усложнение кода программы

Proxy (Заместитель)

- структурный паттерн, который позволяет подставлять вместо реальных объектов объекты-заместители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то

Проблема - наличие внешнего ресурсоёмкого объекта, который не нужно постоянно использовать. Инициализировать объект в начале программы - затратно, а отложенная инициализация не всегда возможна и может привести к дублированию кода.

Решение - создать класс-**заместитель**, имеющий тот же интерфейс, что и оригинальный объект. При получении запроса от клиента заместитель бы сам создавал объект и переадресовывал ему всю работу.



Применимость:

- **Виртуальный прокси -** необходимость ленивой инициализации тяжелых объектов

- **Защищающий прокси -** защита доступа
- **Удалённый прокси -** "локальный" запуск сервиса, находящегося на удалённом сервисе
- **Логирующий прокси** - логирование запросов
- **Умная ссылка -** кеширование объектов - необходимость кешировать запросы клиентов и управлять их жизненным циклом

Реализация:

1. Определить интерфейс, который бы сделал оригинальный объект и заместитель взаимозаменяемыми.
2. Создать класс заместителя, содержащий ссылку на объект
3. Реализовать методы заместителя
4. Возможно введение фабрики для решения вопроса - создавать заместителя или настоящий объект
5. Возможно введение ленивой инициализации сервисного объекта при первом обращении к методам заместителя

Преимущества:

- Позволяет контролировать сервисный объект независимо от клиента
- Может работать, даже если сервисный объект ещё не создан
- Может контролировать жизненный цикл сервисного объекта

Недостатки:

- Усложнение кода программы
- Увеличение времени отклика от сервиса

XIV. Поведенческие паттерны

Monday, June 11, 2018 21:33

Поведенческие паттерны

- паттерны, решающие задачу эффективного и безопасного взаимодействия между объектами программы

Chain Of Responsibility

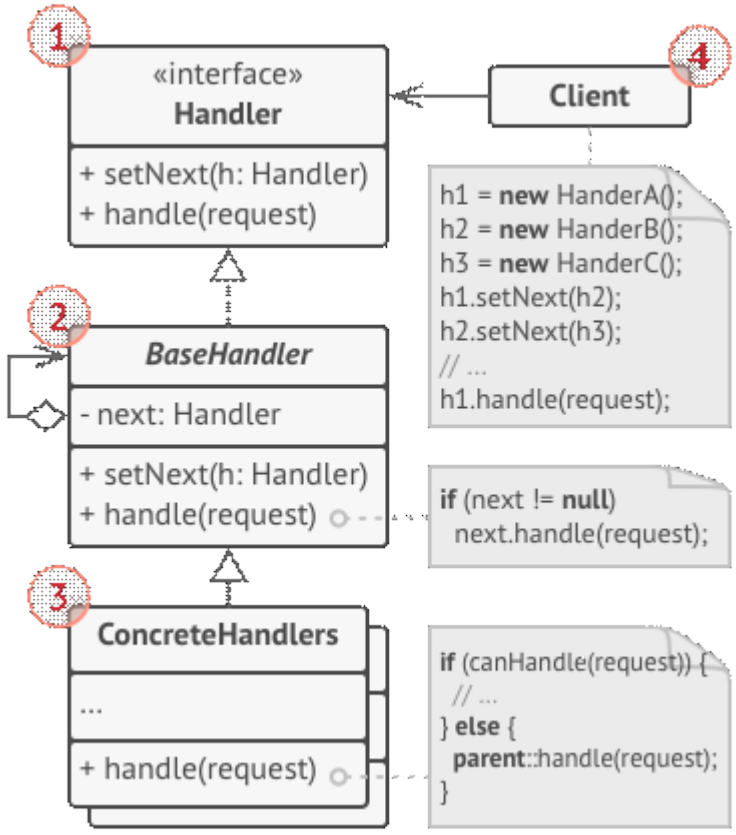
- поведенческий паттерн, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи

Проблема - необходимость обработки неизвестных заранее разнообразных запросов разными способами. Со временем код обработки становится более сложным и запутанным - при изменении одного правила необходимо менять код всех остальных проверок

Решение - превратить отдельные поведения в объекты-обработчики и связать их одну цепь. Каждый из объектов будет содержать ссылку на следующий обработчик в цепи. Чтобы обработать запрос, достаточно передать его первому обработчику. Есть два варианта передачи запроса между обработчиками:

- Обработчик прерывает дальнейшую обработку, если не прошла текущая проверка
- Обработчики прерывают цепь, только когда они могут обработать запрос

Такие цепочки можно легко выстраивать в древовидной структуре, если все объекты имеют какой-то общий интерфейс.



Применимость:

- Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие запросы будут приходить и какие обработчики понадобятся.
- Когда важно, чтобы обработчики выполнялись в строгом порядке
- Когда набор обработчиков должен задаваться динамически.

Реализация:

1. Создать интерфейс обработчика и описать в нём основной метод обработки
2. Создать абстрактный класс обработчика, чтобы не дублировать код получения метода следующего обработчика
3. Создать классы конкретных обработчиков и реализовать в них методы обработки классов

Преимущества:

- Уменьшает зависимость между клиентом и обработчиком
- Реализует Single Responsibility Principle
- Реализует Open/Close Principle

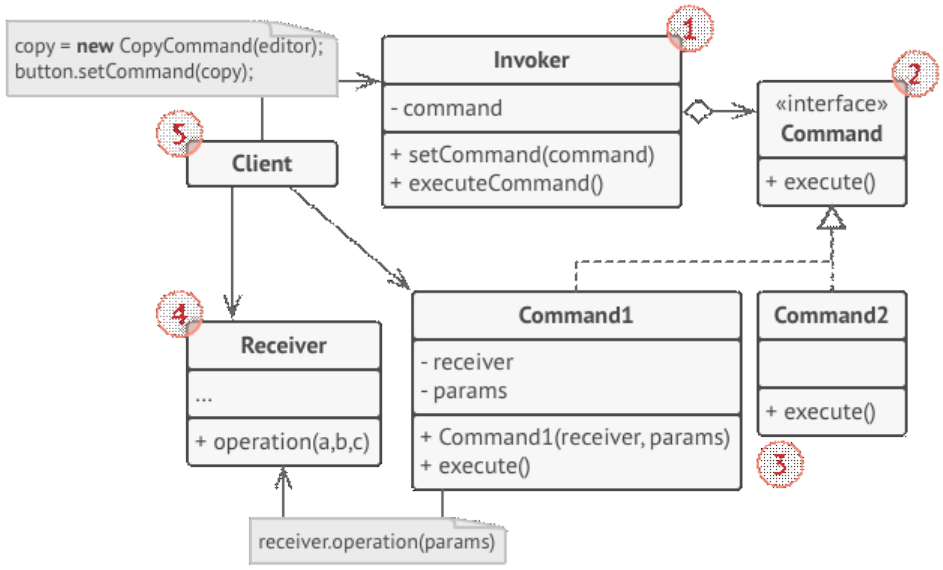
Недостатки

- Запрос может остаться никем не обработанным

<u>Команда</u>

- поведенческий паттерн, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их и поддерживать отмену операций

Проблема - необходимость выполнять одну операцию в разных объектах
Решение - разделить программу на несколько слоёв: один посылает команды, другой - принимает и выполняет. Каждый вызов нужно обернуть в свой класс **- команду -** с единственным методом, который и будет осуществлять вызов. Классы команд стоит объединить в один интерфейс с единственным методом запуска. В таком случае отправители смогут работать с различными командами, не привязываясь к конкретному классу.



Применимость:

- Необходимость параметризовать объекты выполняемым действием
- Необходимость ставить операции в очередь и выполнять их по расписанию или передавать по сети
- Необходимость в операции отмены

Реализация:

1. Создать общий интерфейс и определить в нём метод запуска
2. Создать классы конкретных команд
3. Добавить в классы отправителей поля для хранения команд
4. Изменить основной код отправителей так, чтобы они делегировали выполнение действия команде

Преимущества:

- Убирает прямую зависимость между объектами, вызывающими операции и объектами, которые непосредственно их выполняют
- Позволяет реализовать простую отмену и повтор операций
- Позволяет реализовать отложенный запуск операций
- Позволяет собирать сложные команды из простых
- Реализует Open/Close Principle

Недостатки

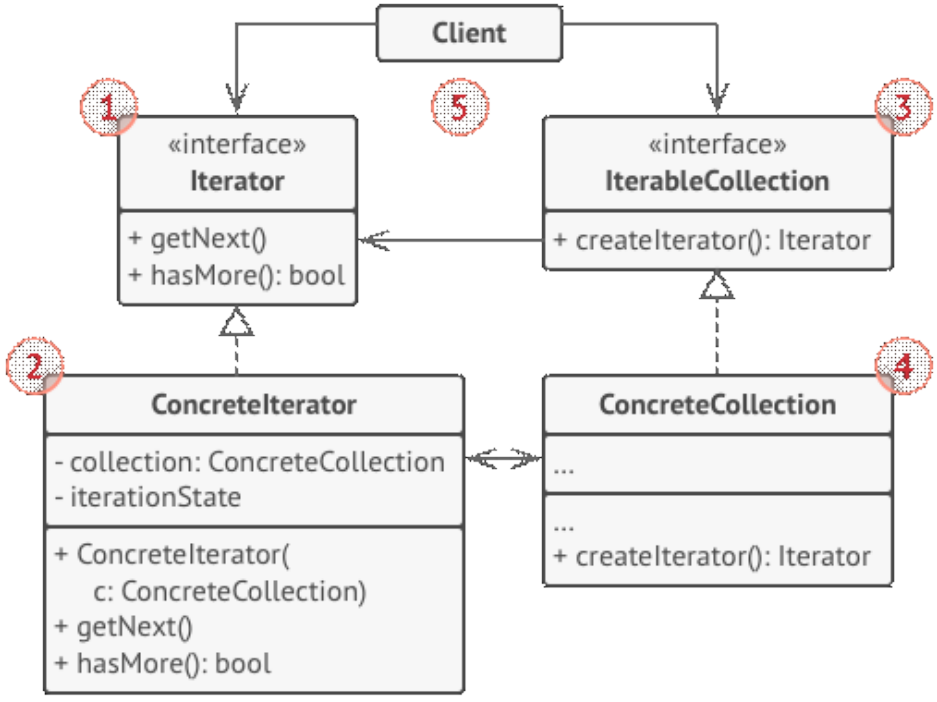
- Усложнение кода программы

Iterator

- поведенческий паттерн, позволяющий последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления

Проблема - необходимость обойти все элементы какой-либо коллекции. Коллекция может иметь любую форму, и иногда для одной коллекции могут потребовать разные методы обхода. Всё это увеличит код класса-коллекции

Решение - вывести поведение обхода коллекции в отдельный класс - **итератор.** В каждом объекте-итераторе будет находится состояние обхода и метод перехода на следующий элемент



Применимость

- Необходимость скрыть от клиента детали реализации сложной структуры
- Необходимость иметь несколько вариантов обхода одной и той же структуры
- Необходимость иметь единый интерфейс для обхода различных структур данных

Реализация:

1. Создать общий интерфейс итераторов
2. Создать интерфейс коллекции и описать в нём метод получения итераторов.
3. Создать нужные классы конкретных итераторов

Преимущества:

- Упрощение классов хранения данных
- Возможность реализовать различные способы обхода структуры данных

- Возможность одновременно перемещаться по структуре данных в разные стороны

Недостатки

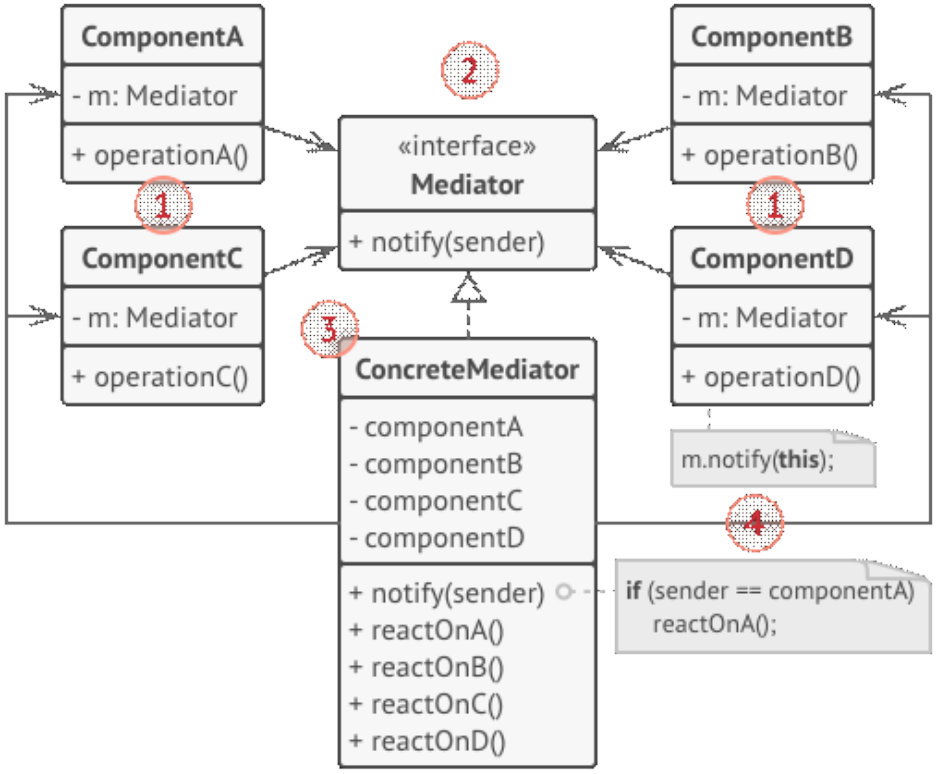
- Не оправдан, если можно обойтись простым циклом

Mediator (Посредник)

- поведенческий паттерн, который позволяет уменьшить связанность классов между собой, благодаря перемещению этих связей в один класс-посредник

Проблема - наличие в программе сильно связанных сущностей, которые невозможно повторно использовать

Решение - заставить классы общаться не напрямую, а через объект-**посредник**, который знает, кому перенаправить тот или иной запрос. В таком случае компоненты системы будут зависеть не друг от друга, а только от посредника. Посредник скроет в себе все сложные связи и зависимости между отдельными компонентами



Благодаря тому, что компоненты общаются с посредником через абстрактный интерфейс, их можно использовать повторно с другим посредником. Если в компоненте происходит событие, он оповещает об этом посредника, который принимает решение.

Применимость:

- Наличие хаотично взаимосвязанных классов
- Невозможность из-за зависимостей повторно использовать класс
- Необходимость создавать подклассы компонентов, чтобы использовать компоненты в разных контекстах

Реализация:

1. Найти группу сильно связанных компонентов, создать общий интерфейс посредников и описать в нём методы для взаимодействия между компонентами
2. Реализовать интерфейс посредника в классе конкретного посредника
3. Можно перевести код создания посредника в фабрику
4. Установить связь компонентов с посредником, например, передачи параметра посредника в конструктор компонента
5. Изменить код компонентов, так, чтобы компоненты вызывали метод посредника вместо методов других компонентов

Преимущества:

- Устранение зависимости между компонентами
- Упрощение взаимодействия между компонентами
- Централизация управления

Недостатки

- Посредник может получится очень большим

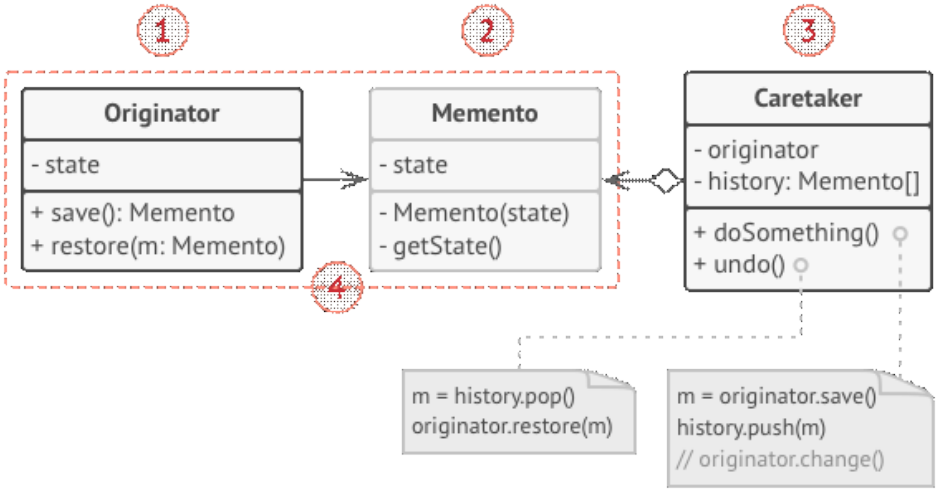
Memento (Снимок)

- поведенческий паттерн, который позволяет сохранять и восстанавливать предыдущие состояния объектов, не раскрывая подробностей их реализации

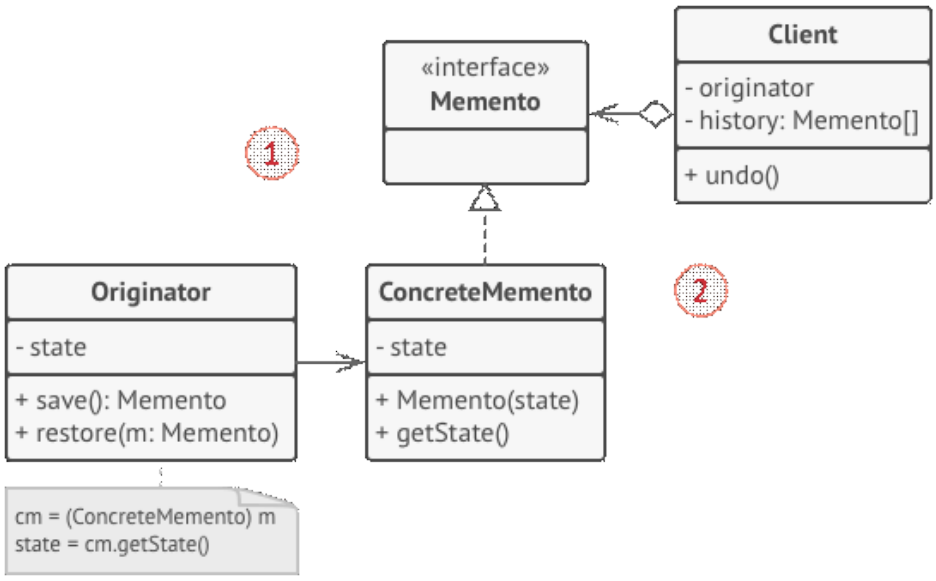
Проблема - необходимость отмены действий или работы с предыдущими состояниями класса. Копирование "в лоб" сопряжено с трудностями - так, приватные поля класса не скопировать, а при копировании всего класса будет занято очень много памяти. Кроме того, изменение класса приведёт к необходимости изменять код копирования везде, где можно.

Решение - предоставить создание копий самому объекту, который владеет этим состоянием и держать копию в специальном объекте-**снимке** с ограниченным интерфейсом.

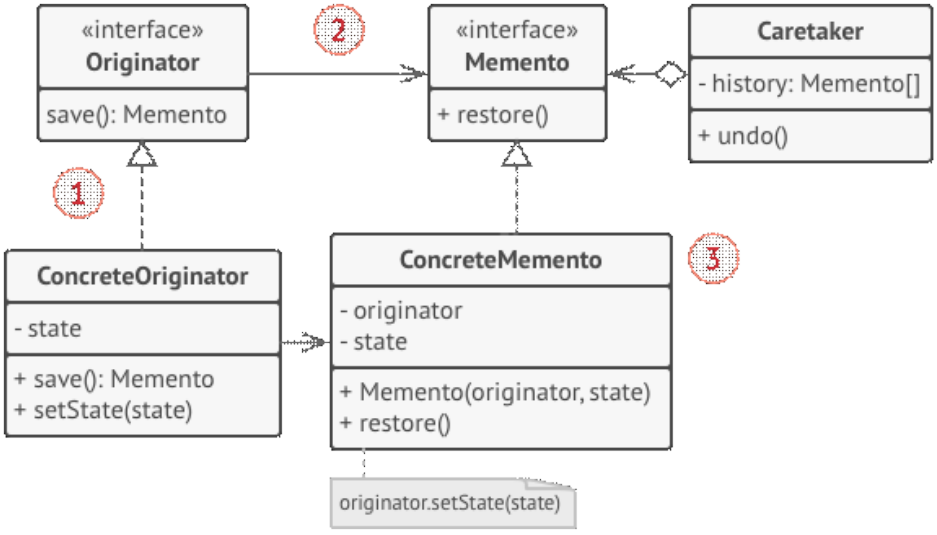
Снимок должен быть полностью открыт для создателя, чтобы он мог восстановить своё состояние по нему, но в то же время снимки могут хранится и в других классах-**опекунах**.



Для реализации снимка в таком виде необходима возможность создания вложенных классов, доступная только в некоторых языках (C++, Java, C#). Если такой возможности нет, то можно разделить снимок на интерфейс, с которым будет работать класс-опекун и реализацию, с которой будет работать создатель. Создатель будет выполнять операцию привидения типов.



Если же нужно полностью исключить доступ к состоянию создателей и снимков, можно выделить интерфейс и у создателя:



Применимость:

- Необходимость создавать любое количество снимков объекта или его части, чтобы можно было восстановить его в том же состоянии
- Когда прямое получение объекта раскрывает приватные детали его реализации

Реализация:

1. Создать класс снимка и описать в нём все поля состояния, которые имеются в создателе
2. Сделать объект снимка неизменяемым, и, если возможно, вложенным в класс создателя.

3. Добавить в класс создателя метод получения снимка и метод восстановления из снимка. Можно метод восстановления создателя добавить и в снимок.

Преимущества

- Не нарушает инкапсуляции исходного объекта
- Упрощает структуру исходного объекта

Недостатки

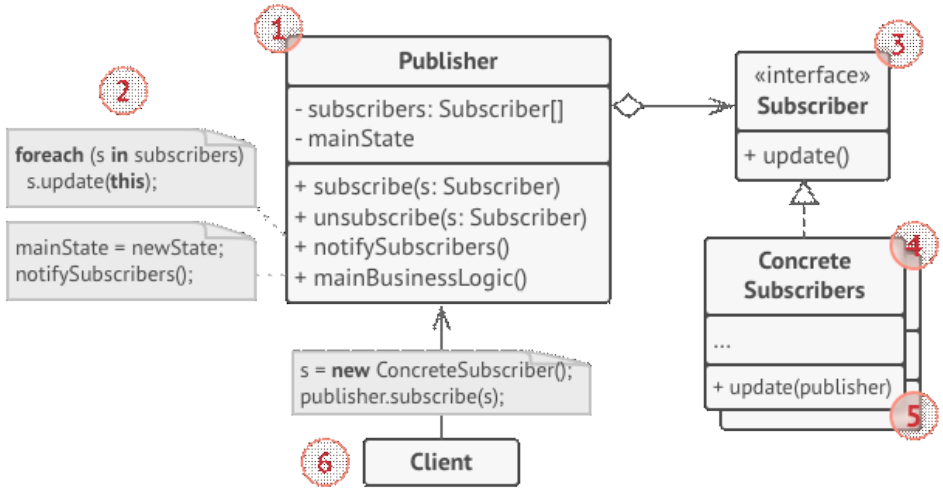
- Требует много памяти и может повлечь за собой издержки
- Иногда сложно гарантировать, то, что только исходный объект имеет доступ к состоянию снимка

Observer (Наблюдатель)

- поведенческий паттерн, который создает механизм подписки, позволяющий одним объектам следить за другими и реагировать на события, произошедшие в других объектах

Проблема - необходимость получения одними объектами (**подписчиками**) уведомления о изменении состояния других объектов (**издателей**). Оповещение всех подписчиков издателем или периодическая проверка подписчиками издателя - лишняя трата ресурсов

Решение - хранить внутри объекта-издателя список ссылок на объекты-подписчики. Издатель предоставляет подписчикам методы, с помощью которых они могут добавлять или убирать себя из списка. Все подписчики должны следовать общему интерфейсу и иметь один метод оповещения.



Применимость:

- Когда после изменения одного объекта нужно сделать что-то в других, но что - наперёд не известно
- Когда одни объекты должны наблюдать за другими только в определённых случаях

Реализация

1. Разбить функциональность программы на две части: независимое ядро и опциональные зависимые части - подписчики

2. Создать интерфейс подписчиков
3. Создать интерфейс издателей, и описать в нём операции управления подпиской.
4. Решить, куда поместить код ведения подписки. Можно вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться издатели или делегировать работу издателей вспомогательному объекту
5. Создать классы конкретных издателей
6. Реализовать метод оповещения в подписчиках

Преимущества:

- Издатели не взаимозависимы с подписчиками
- Можно динамически подписывать и отписывать подписчиков
- Реализует Open/Close Principle

Недостатки:

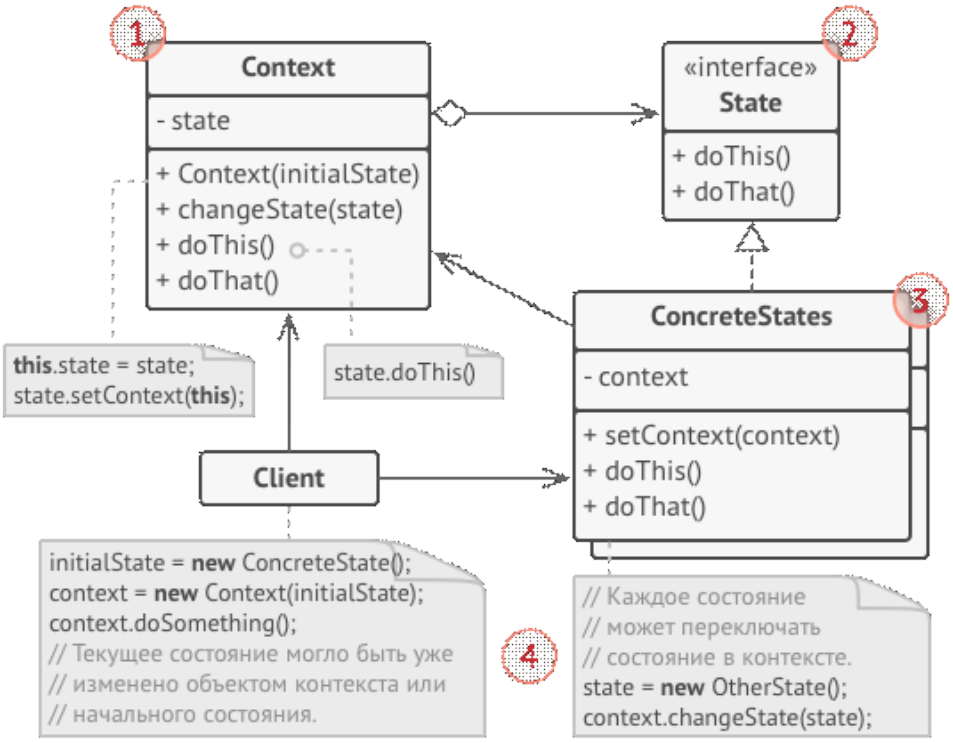
- Случайный порядок оповещения подписчиков

State (Состояние)

- поведенческий паттерн, который позволяет объектам менять поведение в зависимости от своего состояния. Снаружи кажется, что изменился класс объект

Проблема - необходимость реализовать конечный автомат, поведение которого в различных состояниях различно. Реализация автомата с помощью условных операторов чревата разрастанием кода и сложностью поддержки.

Решение - создать отдельный класс для каждого состояния и выносить туда нужные состояния. Первоначальный объект **- контекст** - будет содержать ссылку на объект-состояние и делегировать ему работу. Объекты-состояния должны иметь общий интерфейс. Состояния должны знать друг о друге и иметь возможность инициировать переходы друг от друга.



Применимость:

- Наличие объекта, поведение которого кардинально варьируется в зависимости от внутреннего состояния
- Наличие большого количество условных операторов
- Построение табличной машины состояний с дублирующимся кода

Реализация:

1. Определиться с классом, который будет играть роль контекста.
2. Создать общий интерфейс состояний
3. Создать конкретные классы состояний
4. Создать в контексте поля для хранения классов состояний

Преимущества:

- Избавление от множества условных операторов внутри автомата
- Концентрация кода, связанного с одним состоянием, в одном месте
- Упрощение кода контекста

Недостатки

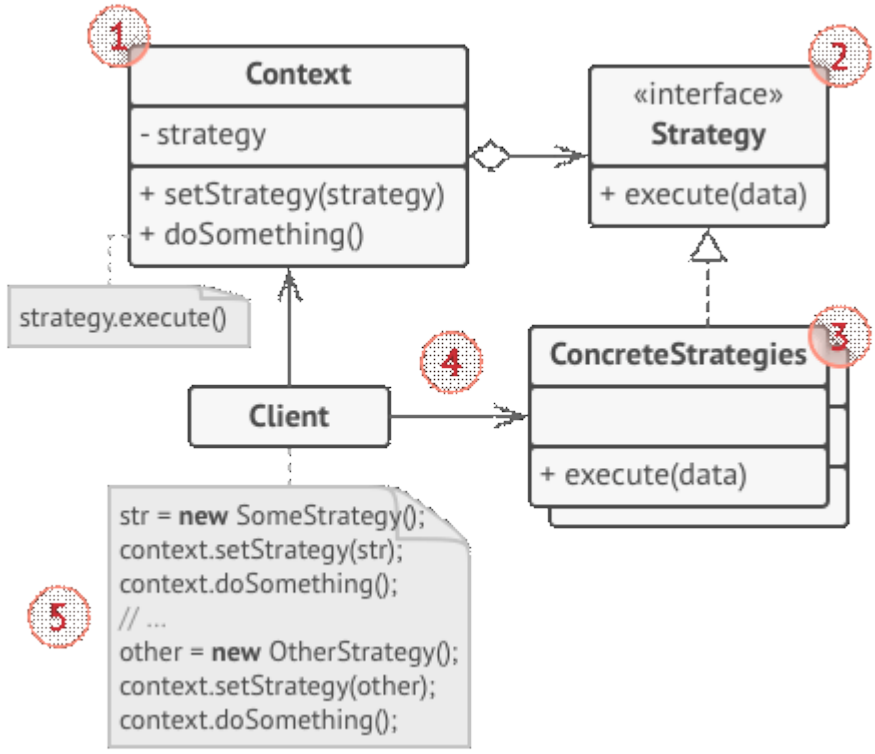
- Возможное неоправданное усложнение кода

Strategy

- поведенческий паттерн, который определяет семейство похожих алгоритмов и помещает их в собственный класс, после чего алгоритмы можно взаимозаменять во время выполнения программы.

Проблема - наличие нескольких алгоритмов, решающих сходные задачи

Решение - вынести эти алгоритмы в классы с общим интерфейсом - **стратегии**. Чтобы изменить работу программу, достаточно будет изменить класс-стратегию. Это облегчает добавление новых алгоритмов



Применимость

- Необходимость использовать разные вариации алгоритма внутри одного и того же объекта
- Наличие множества классов, отличающихся только поведением
- Необходимость скрыть детали реализации алгоритмов
- Реализация выбора алгоритма через большой условный оператор

Реализация:

1. Создать общий интерфейс стратегии, описывающей нужный часто изменяющийся алгоритм или варьирующийся в ходе выполнения программы
2. Поместить вариации объекта в собственные классы, которые реализуют этот интерфейс
3. В классе контекста создать поле для хранения ссылки на текущий объект-стратегию и способ её изменения

Преимущества:

- Возможность замены алгоритмов на лету
- Изоляция кода и данных алгоритмов от остальных классов
- Уход от наследования к делегированию
- Реализация Open/Close Principle

Недостатки:

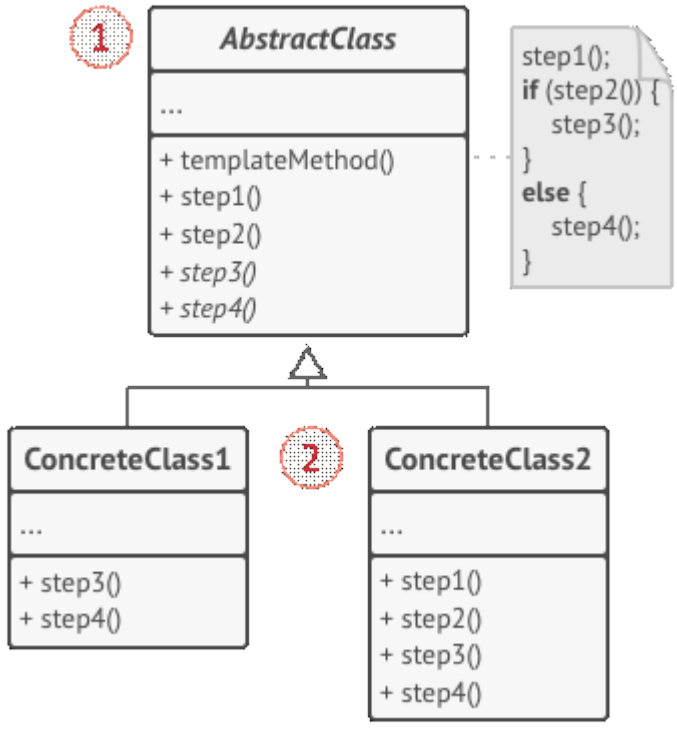
- Усложнение кода
- Клиент должен знать разницу между стратегиями

Template Method (Шаблонный Метод)

- поведенческий паттерн, который определяет скелет алгоритма,

перекладывая ответственность за некоторые его шаги на подклассы.
Проблема - наличие разных алгоритмов, содержащих похожие куски кода.
Решение - разбить алгоритм на последовательность шагов, описать шаги в отдельных методах и вызвать эти шаги в одном шаблонном методе друг за другом. Это позволит подклассам переопределять некоторых шаги алгоритма, оставляя структуру без изменений. Шаги делятся на 3 категории:

- Абстрактные шаги - каждый подкласс должен их реализовывать
- Шаги с реализацией по умолчанию
- Хуки - пустые методы, благодаря которым подклассы могут вклиниться в нужный участок кода



Применимость:

- Необходимость подклассам расширять базовый алгоритм, не меняя его структуры
- Наличие нескольких классов, делающих одно и то же с незначительными отличиями

Реализация:

1. Разбить алгоритм на шаги
2. Создать абстрактный базовый класс и определить в нём шаблонный метод, состоящий из вызовов шагов алгоритма
3. Добавить в абстрактный класс методы для каждого из шагов
4. Возможно, добавить хуки
5. Создать конкретные классы, унаследовав их от абстрактного

Преимущества:

- Облегчает повторное использование кода

Недостатки:

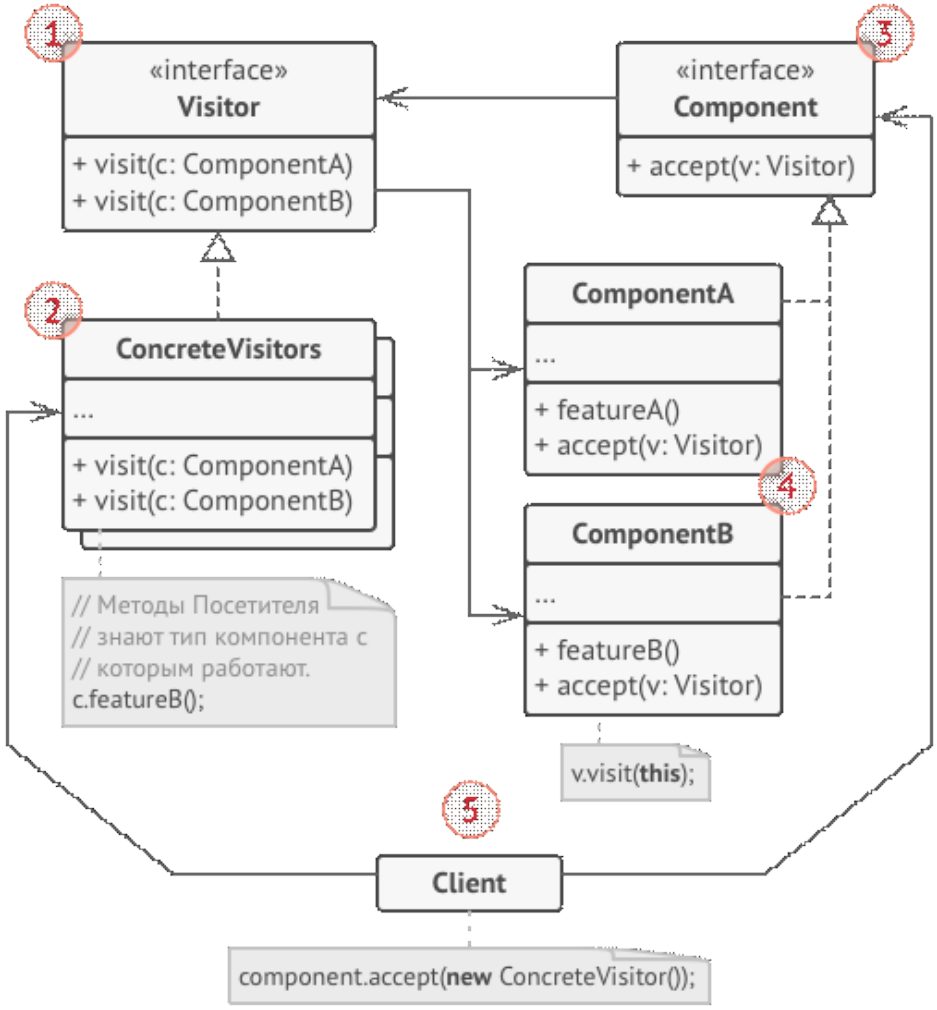
- Жесткое ограничение скелетом алгоритма
- Возможность нарушения Liskov Substitution Principle изменением базового поведения одного из шагов алгоритма через подкласс
- Сложность поддержки шаблонного метода с ростом количества шагов

Visitor (Посетитель)

- поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Проблема - необходимость перебрать разные элементы коллекции, причем каждый тип элемента по-своему.

Решение - вынести логику перебора в отдельный класс-**посетитель**. В нём должны быть методы, принимающие на вход все типы объектов. Внутри элементов нужно создать метод, принимающий посетителя. Таким образом, каждый элемент будет вызывать нужный метод посетителя сам.



Применимость

- Необходимость выполнения операции над элементами сложной структуры
- Нежелание засорять классы элементов не связанными друг с другом операциями
- Когда новое поведение имеет смысл только для некоторых классов из

иерархии

Реализация:

1. Создать интерфейс посетителя и объявить в нём методы посещения для каждого класса
2. Описать интерфейс компонентов
3. Реализовать методы принятия во всех конкретных компонентах
4. Для каждого нового поведения создать свой конкретный класс. Реализовать методы интерфейса посетителей

Преимущества:

- Упрощает добавление операций, работающих со сложными структурами объектов
- Объединяет родственные операции в одном классе
- Посетитель может накапливать состояние при обходе структуры компонентов

Недостатки

- Сложно реализовать часто меняющуюся иерархию
- Может привести к нарушению инкапсуляции.