Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) - основная методология программирования.

Она является продуктом 30 летней практики и включает ряд языков: Simula 67, SmallTalk, Eiffel, Objective C, C++, Object Pascal, Java, C##.

Это стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты.

ООП – решение кризиса

ООП является лишь последним звеном в длиной цепи решений, которые были предложены для разрешения "кризиса программного обеспечения".

Кризис программного обеспечения означает, что те задачи, которые мы хотим решить, опережают наши возможности.

Сложные системы реального мира

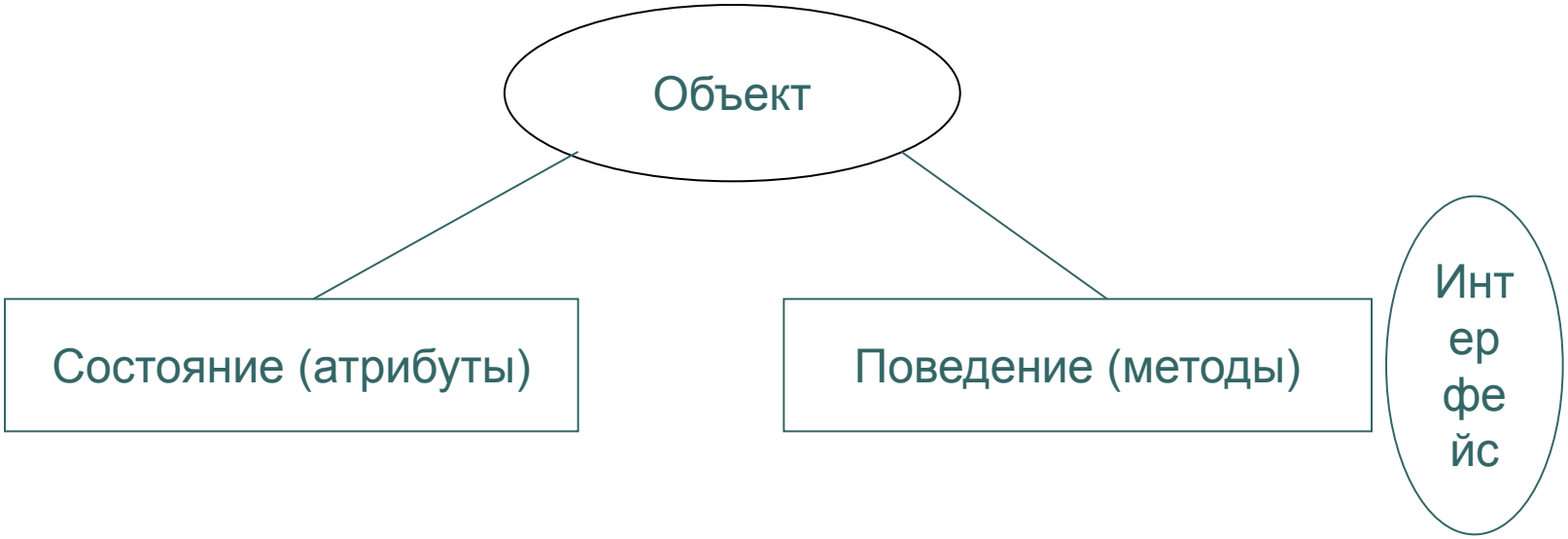
- Персональный компьютер
- Дерево
- Человек
- Предприятие
- Государство

Признаки сложных систем

- Являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы и т.д. – до самого низкого уровня.
- Выбор элементарных компонентов произволен и зависит от исследователя.
- Каждая часть системы имеет свою функцию и может рассматриваться независимо от других.
- Системы состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
- Система сложнее, чем совокупность ее частей.

Объекты

В предметной области выделяются объекты – некоторые целостные сущности, обладающие определенным поведением.



Классы

Классы – множества однотипных объектов с одинаковым поведением и набором атрибутов.

У различных объектов одного класса различаются значения атрибутов, однако методы совпадают.

Объекты называются экземплярами класса.

Реализация действий в ООП

Действие в ООП инициируется посредством передачи сообщений объекту, ответственному за действия.

Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией (аргументами), необходимой для его выполнения. Если объект принимает сообщение, то на него автоматически возлагается **ответственность** за выполнение указанного действия.

В качестве реакции на сообщение получатель запускает некоторый метод (алгоритм), чтобы удовлетворить принятый запрос. Детали метода известны только получателю сообщения.

Инкапсуляция

Таким образом внутреннее состояние и поведение объекта скрыто от других объектов. Изменить его можно извне только с помощью передачи сообщения (вызова метода). В этом состоит принцип **инкапсуляции**.

Метод, выполняемый в ответ на сообщение, определяется классом, которому принадлежит объект. Все экземпляры одного и того же класса используют одинаковые методы.

Позднее связывание. Полиморфизм.

- Имеется определенный объект, выполняющий метод
- Интерпретация сообщения (вызываемый метод) зависит от получателя и для разных объектов может быть разной.

  Обычно конкретный получатель неизвестен до выполнения программы, следовательно неизвестен метод, который будет вызван. В этом случае решение, какой метод вызывать, должно быть принято во время выполнения программы. Такой способ связи сообщения и метода называется **поздним связыванием**. Возможность объектов по-разному реагировать на одинаковые сообщения называется **полиморфизм**.

Иерархия наследования

Классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс (или подкласс) наследует атрибуты родительского класса (или надкласса), расположенного выше в иерархическом дереве.

Переопределение метода

Поиск метода, который вызывается в ответ на определенное сообщение, начинается с методов, принадлежащих классу получателя.

Если подходящий метод не найден, то поиск продолжается для родительского класса. Поиск продвигается вверх по цепочке родительских классов до тех пор, пока не будет найден нужный метод или пока не будет исчерпана последовательность родительских классов.

В первом случае выполняется найденный метод, во втором - выдается сообщение об ошибке.

Если выше в иерархии классов существуют методы с тем же именем, что и текущий, то говорят, что данный метод переопределяет наследуемое поведение.

Возможность переопределения методов есть реализация полиморфизма.

Повторное использование

Наследование позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности. Полиморфизм обеспечивает, чтобы общий код удовлетворял конкретным особенностям отдельных типов данных.

В итоге становится возможным максимально использовать существующий код.

ОО программа

Программа – совокупность взаимодействующих объектов. Каждый объект выполняет конкретную функцию. Объект соединяет вместе состояние (данные) и поведение (методы). Объекты одного класса имеют одни и те же методы.

Объект проявляет свое поведение путем вызова метода в ответ на сообщение. Интерпретация сообщения зависит от объекта и может быть различной для различных классов объектов .

Для удобства создания нового класса из уже существующих используется механизм наследования. Наследование обеспечивает повторное использование.

Принципы ООП

- Инкапсуляция (объединение данных с методами в сочетании со скрытием данных)
- Абстракция (расширяемость системы классов)
- Наследование (повторное использование методов и атрибутов)
- Полиморфизм (переопределение методов и гибкий выбор метода во время выполнения программы)

ООП в C++

Класс – тип данных, содержащий поля (переменные) и методы (функции) для их обработки.

Объект – переменная типа класс.

Отличия ООП в C++ от Delphi

- статическое размещение объектов в памяти
- неявный вызов конструкторов и деструкторов
- перегрузка операций
- множественное наследование
- отсутствие свойств (property)
- вложенность классов
- отсутствие единого предка

Описание класса (без наследования)

```
class <имя> {
   [private:]
    <описание скрытых элементов>
  public:
    <описание доступных элементов>
};
```

Элементы класса – поля и методы.

Пример класса

```
#include <iostream.h>
#include <string.h>

class student
{       char fam[20],name[20];
        int age;
public:
         void setdata (char *f, char *n, int a = 20)
         {     age = a; strcpy(fam, f);
               strcpy(name, n);
         }
         void show ()
        { cout << fam << ' ' << name << ' ' <<  age <<
       endl; }
};
```

Поля

Встраиваемые методы

Использование объектов

```
int main ()
{  student a, b ;
   a.setdata("Ivanov", "Boris");
   b.setdata("Sokolova", "Olga", 19);
   a.show();
   b.show();
  return 0;
}
```

Описание объектов

Вызов методов (передача сообщений)

Глобальные и локальные классы

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или класса).

Локальные классы не могут иметь статических элементов, все методы должны быть описаны внутри класса.

Встроенные и обычные методы

Встроенный метод определяется внутри класса. Метод может быть определен вне класса, тогда внутри класса указывается только его заголовок.

```
class student
{      char fam[20],name[20];
       int age;
public:
      void setdata (char *f, char *n, int a = 20);
      void show ();
};
```

Определение метода вне класса

При определении метода вне класса в его заголовке указывается имя класса с использованием операции доступа к области видимости (::).

```
void student :: setdata (char *f, char *n, int a)
{     age = a; strcpy(fam, f);
      strcpy(name, n);
 }
```

Примеры использования объектов

```
student group [25];
student *s = new student;
   …
for (int i = 0; i < 25; i++)
  group[i].show();
s -> setdata ( “Sidorov”, “Ivan”);
```

Указатель this

Для того, чтобы метод работал с полями того объекта, для которого он вызван, передается скрытый параметр this, содержащий константный указатель на объект. В явном виде this используется для возврата ссылки или указателя на объект.

```
student& student :: old (student &a)
{ if ( age > a.age ) return *this;
                     return a;
}
 …
student c = a.old(b);
c.show();
```

Конструкторы

Конструктор – метод предназначенный для инициализации объекта, выполняющийся автоматически в момент его создания.

Конструктор имеет имя, совпадающее с именем класса.

Конструктор не возвращает значение.

Класс может иметь несколько перегруженных конструкторов с разными параметрами.

Конструктор по умолчанию – не имеет параметров. Для любого класса автоматически создается такой конструктор.

Конструктор может иметь параметры со значениями по умолчанию.

Пример использования конструктора

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
class _time
{   int h,m;
    public:
          _time ( int hours = 0, int minutes = 0)
           { h = hours; m = minutes; }
       void show()
      { cout << setfill('0')<< setw(2) << h
                 << ':' << setw(2) << m << endl;}
};
int main()
{ _time t(12,30), p = 12;
  t.show(); p.show(); return 0;}
```

Конструктор

Создание объектов с инициализацией

Список инициализации

Служит для инициализации полей в конструкторе. Список позволяет инициализировать константные поля и поля-ссылки

```
class _time
{  int h,m;
   public:
          _time ( int hours = 0, int minutes = 0)
           : h( hours ), m ( minutes ) { }
    …
};
```

Конструктор копирования

Конструктор копирования получает в качестве параметра константную ссылку на объект того же класса.

T (const T&)

Этот конструктор вызывается

- при описании объекта с инициализацией другим объектом
- при передаче объекта в функцию по значению
- при возврате объекта из функции

Если конструктор копирования не указан, он будет создан автоматически.

Пример использования конструктора копирования

```
class student
{      char *fam,*name;
       int age;
public:
        student (char *f, char *n, int a = 20) :  age (a)
    { fam = new char [20];   name = new char [20];
          strcpy (fam, f);   strcpy (name, n);   }
        student (student &s) : age (s.age)
        { fam = new char [20];   name = new char [20];
          strcpy (fam, s.fam);   strcpy (name, s.name); }
        void show ();
};
int main ()
{ student a("Ivanov", "Boris"), b("Sokolova", "Olga", 19);
  student c = a;
  a.show();   b.show();   c.show(); return 0; }
```

Объекты как параметры функций

Рассмотрим метод для сложения времени `class _time`

```
{ …
    void add (const _time &time1,
          const _time &time2)
            { h = time1.h + time2.h;
              m = time1.m + time2.m;
              if (m >= 60) { h++; m-=60;}
              h = h % 24;}
};
Вызов: c.add( a , b);
```

Объекты как результаты функций

```
_time inc ( _time interv)
          { _time t;
            t.h = h + interv.h;
            t.m = m + interv.m;
            if (t.m>=60) { t.h++; t.m-=60;}
            t.h = t.h % 24;
            return t;
          }
Вызов: _time c = a.inc(b);
```

Деструкторы

~ <имя класса>

```
class student
{      char *fam,*name;
       int age;
public:
        …
        ~student()    { delete [] name; delete [] fam;}

};
int main ()
{ student a("Ivanov", "Boris"), b("Sokolova", "Olga",
  19);
  student c = a;
  a.show();   b.show();   c.show(); return 0; }
```