



Объектно-ориентированное программирование - 1

Технологии и языки программирования

Юдинцев В. В.

Кафедра теоретической механики



САМАРСКИЙ УНИВЕРСИТЕТ
SAMARA UNIVERSITY

Сложность программного обеспечения

Простые программные системы

- Задумываются, разрабатываются, сопровождаются одним человеком
- Ограниченная область применения
- Короткое время жизни
- Не требуется подробная документация

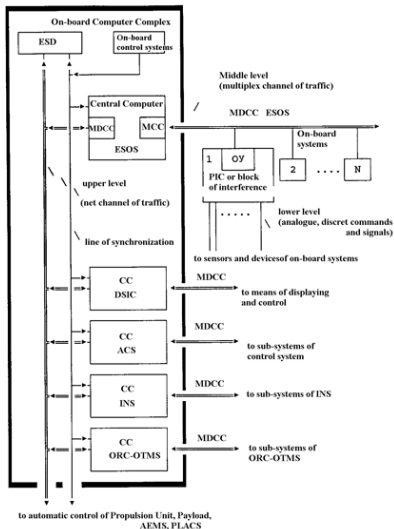
Сложные программные системы

- В разработку сложных программных систем вовлечено значительное количество людей (более 5-ти человек)
- Сложную программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку
- Имеют продолжительный период эксплуатации
- Должны иметь возможность эволюционировать с изменением требований

Количество строк кода

ОС	Количество строк
Windows XP	45 000 000
Debian 7.0 (Linux)	419 000 000
Mac OS X 10.4	86 000 000

Причины сложности



- Сложность предметной области
- Сложность управления процессом разработки
- Гибкость программного обеспечения
- Сложность описания поведения больших дискретных систем

Причины сложности



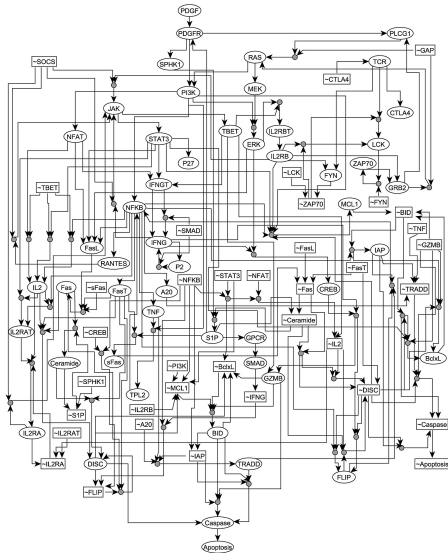
- Сложность предметной области
- Сложность управления процессом разработки
- Гибкость программного обеспечения
- Сложность описания поведения больших дискретных систем

Причины сложности



- Сложность предметной области
- Сложность управления процессом разработки
- **Гибкость программного обеспечения**
- Сложность описания поведения больших дискретных систем

Причины сложности



- Сложность предметной области
- Сложность управления процессом разработки
- Гибкость программного обеспечения
- Сложность описания поведения больших дискретных систем

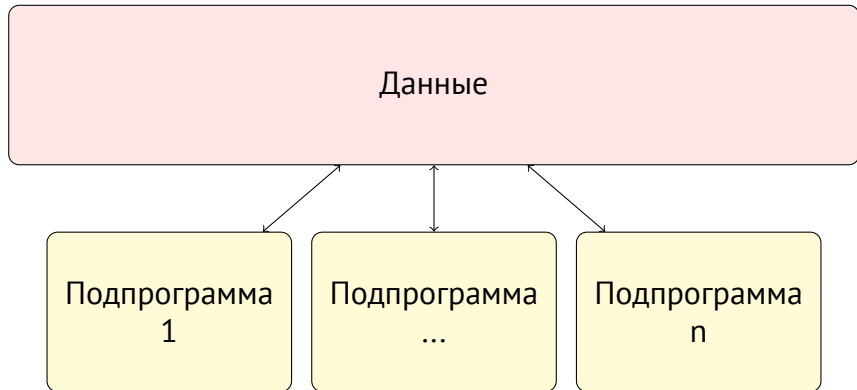
Декомпозиция

Виды декомпозиции

При проектировании сложной программной системы её необходимо разделять на на меньшие подсистемы, каждую из которых можно совершенствовать независимо.

- **Алгоритмическая декомпозиция**
каждый модуль выполняет один из этапов процесса, задачи
- **Объектно-ориентированная декомпозиция**
декомпозиция с точки зрения абстракций предметной области

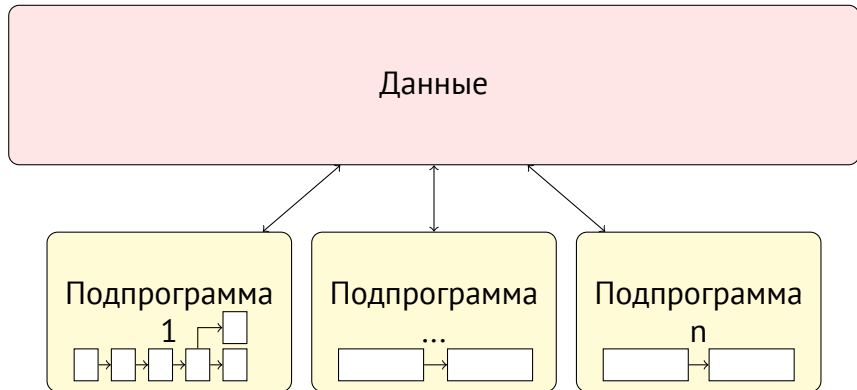
Языки первого поколения



Языки первого поколения

- Программы состоят из подпрограмм
- Используются глобальные переменные — данные открытые для всех подпрограмм
- ✗ Ошибка, допущенная в одной части программы, может оказать разрушительное влияние на остальную часть системы
- ✗ Низкая надежность, запутанность большой программы

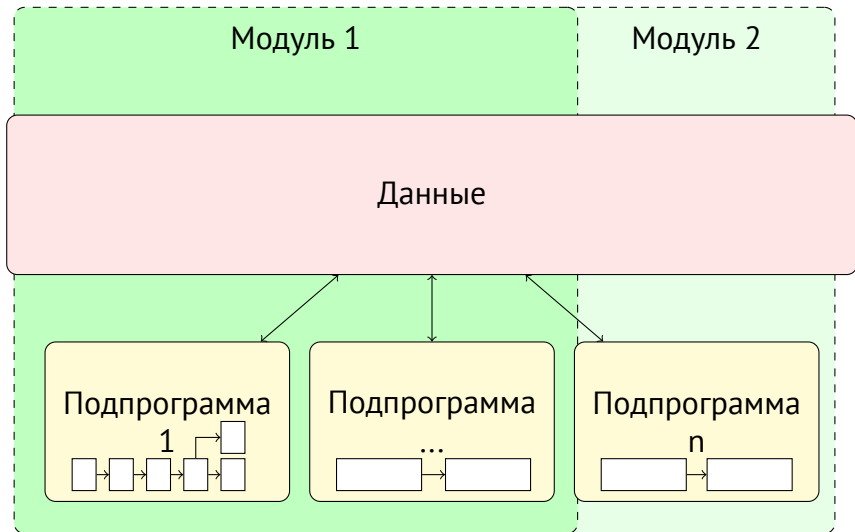
Структурное программирование



Структурное программирование

- Программа разбивается на составляющие элементы
- В программе четко обозначены управляющие структуры, программные блоки, автономные **подпрограммы**.
- Базовые управляющие структуры
 - последовательность
 - ветвление
 - цикл
- Используются **локальные переменные**
- Разработка программы ведётся пошагово, методом “сверху вниз”

Топология языков третьего поколения



Модульное программирование

- Программа разбивается на модули изолированные программные сегменты с четко определённым **интерфейсом**, описывающим как подготовить данные и как вызвать функции модуля преобразующие эти данные.
- Отладка модулей может проводиться независимо
- Модули одной программы могут разрабатываться на различных языках программирования
- Модули могут использоваться повторно

Объектно-ориентированное программирование

Принципы ООП

- Основной элемент конструкции — модуль
- Модуль состоит из связанных классов и объектов
- Классы образуют иерархию (наследование, включение)
- Программа представляет собой набор объектов, имеющих состояние и поведение.

Определение ООП

Технология создания **сложного программного обеспечения**, основанная на представлении **программы** в виде **совокупности объектов**, каждый из которых является **экземпляром определённого типа** (класса), а классы образуют **иерархию** с наследованием свойств.

Принципы ООП

Принципы ООП, сформулированные разработчиком языка Smalltalk Аланом Кеем.

- Всё является объектом.
- Вычисления осуществляются путём взаимодействия (обмена данными) между объектами: один объект требует, чтобы другой объект выполнил некоторое действие.
- Каждый объект имеет независимую память, которая состоит из других объектов.
- Каждый объект является представителем класса, который выражает общие свойства объектов
- В классе задаётся поведение (функциональность) объекта.
- Классы организованы в единую древовидную структуру

Класс = данные + методы

Класс – универсальный, комплексный тип данных, состоящий из набора **полей** (переменных более элементарных типов) и **методов** (функций для работы с этими полями).

Точка на плоскости

Свойства

Координата x

Координата y

Методы

Переместить на dx , dy

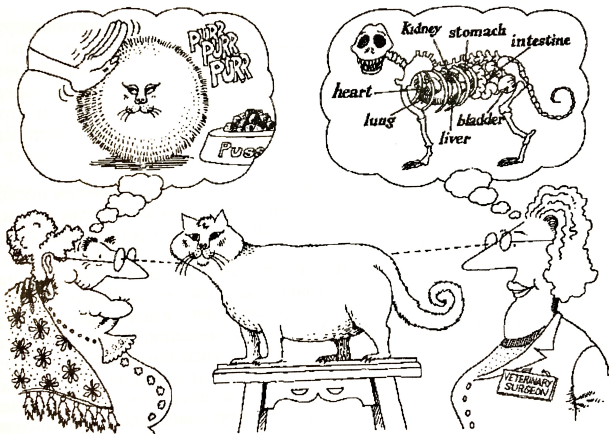
Объявление класса в Python

```
1 class Point :
2     # Конструктор
3     def __init__(self , coordinates):
4         self.x = coordinates[0]
5         self.y = coordinates[1]
6     # Переместить
7     def move(self , delta):
8         self.x = self.x + delta[0]
9         self.y = self.y + delta[1]
```

Объявлен класс `Point`, описывающий точку на плоскости.

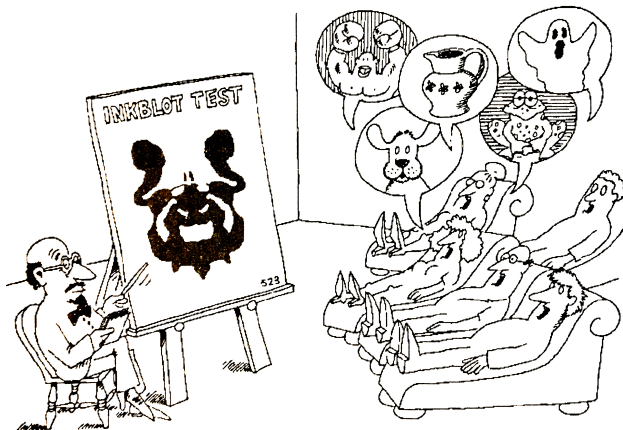
Абстракция данных

Абстрагирование означает выделение значимых характеристик объекта, которые отличают его от остальных объектов, четко определяя его границы с точки зрения наблюдателя.



Абстракция сущности

Выбор правильного набора абстракций для заданной предметной области — главная задача объектно-ориентированного программирования.

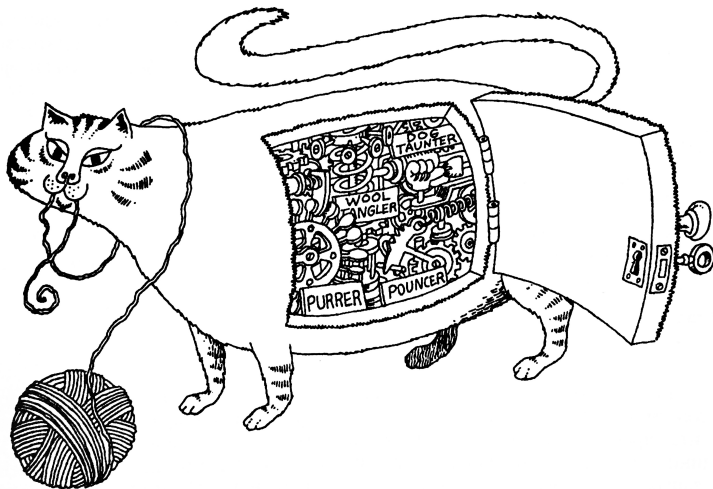


Инкапсуляция

- Инкапсуляция локализует проектные решения, которые могут измениться в результате эволюции системы.
- Инкапсуляция позволяет менять детали реализации без ведома клиентов – пользователей класса.

Инкапсуляция

Инкапсуляция позволяет скрывать, держать в секрете детали реализации необходимого поведения объекта.



Python

- В языке Python всё от чисел до модулей представляют собой объекты.
- Python скрывает большую часть принципов функционирования объектов (инкапсуляция).

```
1 a = list([1, 2, 3, 4, 5])  
2 a.append(6)  
3 print(a.count())
```

Объект

Объект – это сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса.

Создание объекта – экземпляра класса

```
1 # p1 -- объект типа Point
2 # или экземпляр класса Point
3
4 p1 = Point([1, 3])
5
6 print(p1.x, p1.y)
```

1 3

Вызов метода класса:

```
1 p1.move([2, 3])
2
3 print(p1.x, p1.y)
```

3 6

Пользовательские типы данных

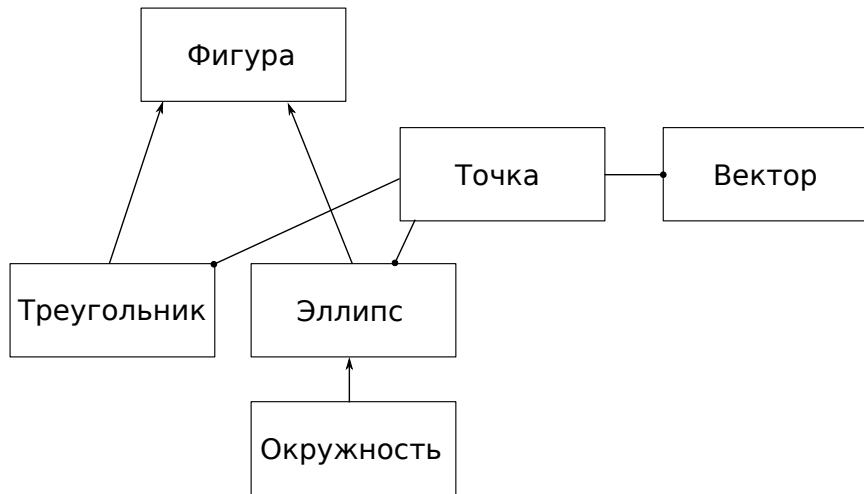
Программирование только с использованием классов без выстраивания иерархии классов, не является ООП. Это программирование на основе абстрактных типов данных.

Методы построения классов

Отношения между классами

- Классы и объекты не существуют изолированно
- В любой проблемной области абстракции взаимодействуют различными способами

Пример иерархии классов



Типы отношений между классами

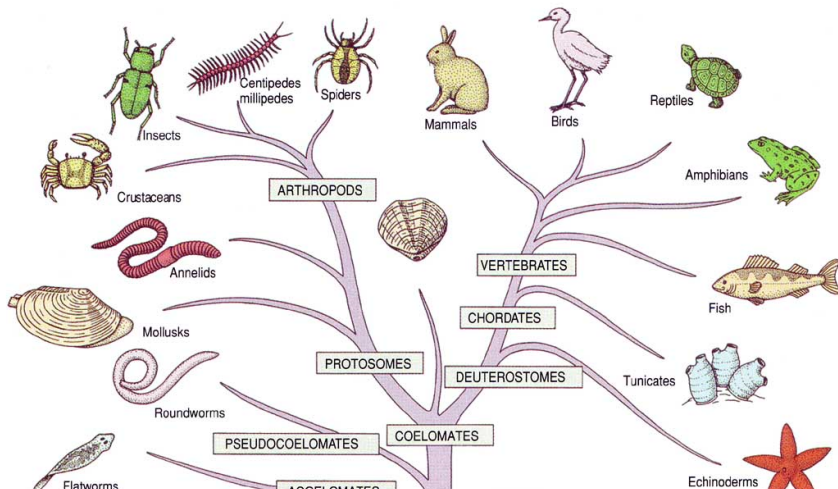
- Общее и частное
растение – цветок
- Часть и целое
двигатель – автомобиль
- Семантические, смысловые отношения и ассоциации
клиент – счёт

Основные отношения между классами

- Наследование
- Композиция
- Агрегация

Наследование

Класс **наследник** создается на основе родительского (базового) класса, **заимствуя** его **свойства и методы** и добавляя новые свойства и методы.



Наследование

```
1 class Figure :
2     # Конструктор
3     def __init__(self , point) :
4         self.position = point
5     def area(self) :
6         pass
```

```
1 class Ellipse (Figure) :
2     # Конструктор
3     def __init__(self , position , a , b) :
4         super().__init__( position )
5         self.a = a
6         self.b = b
7
8     def area(self) :
9         return math.pi*self.a*self.b
```

Наследование

```
1 class Circle(Ellipse):  
2     # Конструктор  
3     def __init__(self, position, r):  
4         super().__init__(position, r, r)  
5         self.r = r  
6  
7     def area(self):  
8         return math.pi*self.r**2
```

Агрегация

Один класс является частью другого класса.

```
1 class Student:
2     def __init__(self, name):
3         self.name = name
4
5 class Group:
6     def __init__(self, group_id):
7         self.group_id = group_id
8         self.students = list()
9
10    def add_student(self, student):
11        self.students.append(student)
12
13    def remove_student(self, student):
14        self.students.remove(student)
```

Агрегация

```
1 s1 = Student( 'Петров' )  
2 s2 = Student( 'Сидоров' )  
3  
4 group = Group()  
5 group.add_student(s1)  
6 group.add_student(s2)  
7  
8 group.remove_student(s2)
```

Композиция

Один класс является **частью** другого класса.

```
1 class Vector :  
2     # Конструктор  
3     def __init__(self , A, B) :  
4         self.A = Point(A)  
5         self.B = Point(B)
```

Поля **A** и **B** класса **Vector** являются объектами типа **Point**:

```
1 vec_AB = Vector(Point(1 , 2) , Point(3 , 4))
```

Точки A и B не существуют без объекта типа **Vector**.

Пример

Уравнение. Абстрактный класс

```
1 class equation :  
2     def get_roots () :  
3         pass
```

Линейное уравнение $bx + c = 0$

```
1 class linear_equation(equation):
2
3     def __init__(self, b, c):
4         self.b = b
5         self.c = c
6
7     def get_roots(self):
8         return -c/b
```

Создаём объект типа `linear_equation`

```
eq1 = linear_equation(5,10)
eq1.get_roots()
```

-2.0

Квадратное уравнение $ax^2 + bx + c = 0$

```
1 class quadratic_equation(equation):
2     def __init__(self, a, b, c):
3         self.b, self.c, self.a = b, c, a;
4     def get_det(self):
5         return self.b**2 - 4*self.a*self.c
6     def roots_exist(self):
7         return self.get_det() >= 0
8     def get_roots(self):
9         det = self.get_det()
10        if self.roots_exist():
11            x1 = (- self.b + det**0.5)*0.5/self.a
12            x2 = (- self.b - det**0.5)*0.5/self.a
13            res = (x1, x2)
14        else:
15            res = None
16        return res
```

Квадратное уравнение $ax^2 + bx + c = 0$

Создаём объект типа `quadratic_equation`

```
eq1 = quadratic_equation(5,10,2)  
eq1.get_det()
```

60

```
eq1.roots_exist()
```

True

```
eq1.get_roots()
```

(-0.2254033307585166, -1.7745966692414832)