ООП

Кафедра ИВТ и ПМ

2018

План

Outline

Абстрактный тип данных

Абстрактный тип данных (АТД, Abstract Data Type - ADT) — это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

Абстрактный тип данных

АТД – это такой тип данных, который скрывает свою внутреннюю реализацию от клиентов.

Удивительно то, что путем применения абстракции АТД позволяет нам не задумываться над низкоуровневыми деталями реализации, а работать с высокоуровневой сущностью реального мира – Стив Макконнелл.

Абстрактный тип данных

АТД позволяет описать тип данных независимо от языка программирования.

Шаблон описание абстрактного типа данных

ADT НаименованиеАбстрактногоТипаДанных

- **Данные**

  . . . перечисление данных ...

- **Операции**
  - **Конструктор**

    Начальные значения:

    Процесс:
  - **Операция. . .**

    Вход:

    Предусловия:

    Процесс:

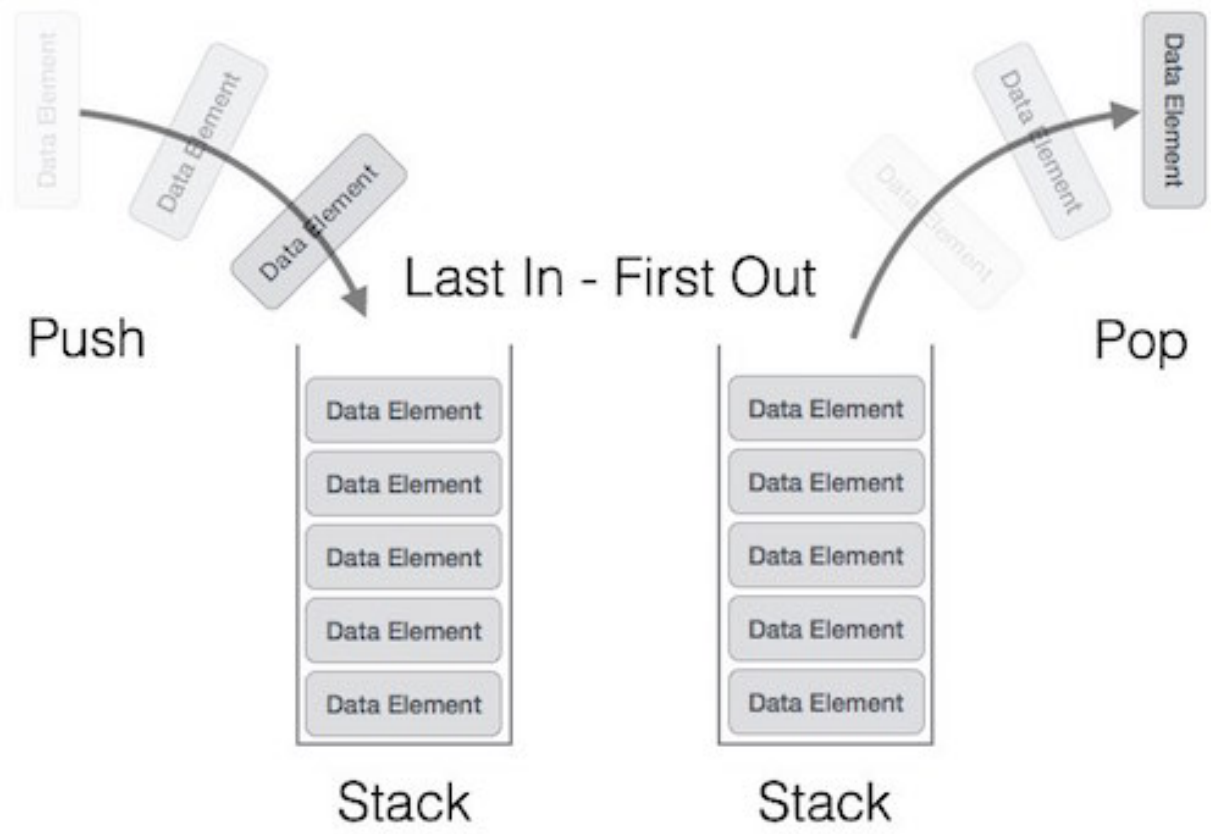
    Выход:

    Постусловия:
  - **Операция. . .** ...

Конец ADT НаименованиеАбстрактногоТипаДанных

Абстрактный тип данных. Пример - стек

Стек - абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»)



Пример ADT - Стек

ADT Stack

- **Данные**

  Список элементов с позицией top, указывающей на вершину стека.

- **Операции**

  - **Конструктор**
    Начальные значения:
    Нет
    Процесс: Инициализация вершины стека.

  - **StackEmpty**
    Вход: Нет
    Предусловия: Нет
    Процесс: Проверка, пустой ли стек
    Выход: Возвращать True, если стек пустой, иначе возвращать False.
    Постусловия: Нет

Пример ADT - Стэк

- Операции (продолжение)
  - **Pop**
    Вход: Нет
    Предусловия: Стек не пустой
    Процесс: Удаление элемента из вершины стека
    Выход: Возвращает элемент из вершины стека
    Постусловия: Элемент удаляется из вершины стека
  - **Push**
    Вход: Элемент для стека
    Предусловия: Нет
    Процесс: Сохранение элемента в вершине стека
    Выход: Нет
    Постусловия: Стек имеет новый элемент в вершине

Пример ADT - Стэк

- Операции (продолжение)
  - **Peek**
    Вход: Нет
    Предусловия: Стек не пустой
    Процесс: Нахождение значения элемента в вершине стека
    Выход: Возвращать значение элемента в вершине стека
    Постусловия: Стек неизменный
  - **ClearStack**
    Вход: Нет
    Предусловия: Нет
    Процесс: Удаление всех элементов из стека и переустановка вершины стека
    Выход: Нет
    Постусловия: Стек переустановлен в начальные условия

Конец ADT Stack

Чем полезен ADT?

- Инкапсуляция деталей реализации. Это означает, что единожды инкапсулировав детали реализации работы АТД мы предоставляем клиенту интерфейс, при помощи которого он может взаимодействовать с АТД. Изменив детали реализации, представление клиентов о работе АТД не изменится.
- Снижение сложности. Путем абстрагирования от деталей реализации, мы сосредотачиваемся на интерфейсе, т.е на том, что может делать АТД, а не на том как это делается. Более того, АТД позволяет нам работать с сущностью реального мира.
- Ограничение области использования данных. Используя АТД мы можем быть уверены, что данные, представляющие внутреннюю структуру АТД не будут зависеть от других участков кода. При этом реализуется “независимость” АТД.
- Высокая информативность интерфейса. АТД позволяет представить весь интерфейс в терминах сущностной предметной области, что, согласитесь, повышает удобочитаемость и информативность интерфейса.

Outline

Структурное программирование

- Что такое структурное программирование?
- Что такое процедурное программирование?
- Что такое модульное программирование?

Outline

Декомпозиция

- Большие программы создавать сложно
- Декомпозиция - разделение предметной области, задач и исходного кода на части - упрощает разработку.

Декомпозиция

- **Алгоритмическая декомпозиция** используется в структурном программировании.
  - Задачи разбиваются на подзадачи
  - Решение задачи и сама программа - *процесс*
  - Структуры данных - вторичны

Декомпозиция

Однако мир представляет собой совокупность взаимодействующих объектов...

- **Объектно-ориентированная декомпозиция.**
  - Предметная область представляется разбивается на объекты
  - Задача представляется как взаимодействие отдельных объектов.

"*Вместо процессоров, бесцеремонно расхватывающих структуры данных, мы имеем дело с благонравными объектами, вежливо просящими друг друга об услугах.*"

– Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений

Объекты

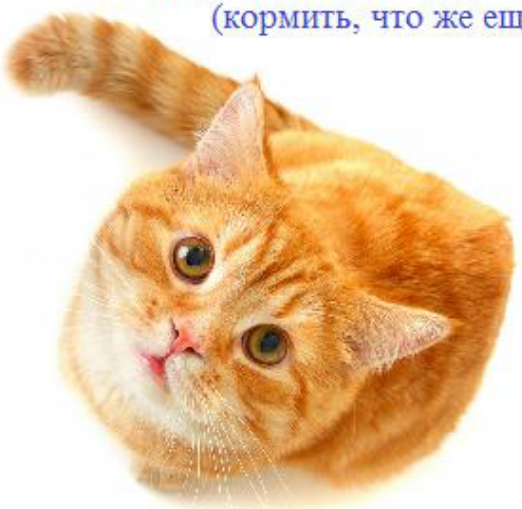
Примеры объектов?

Outline

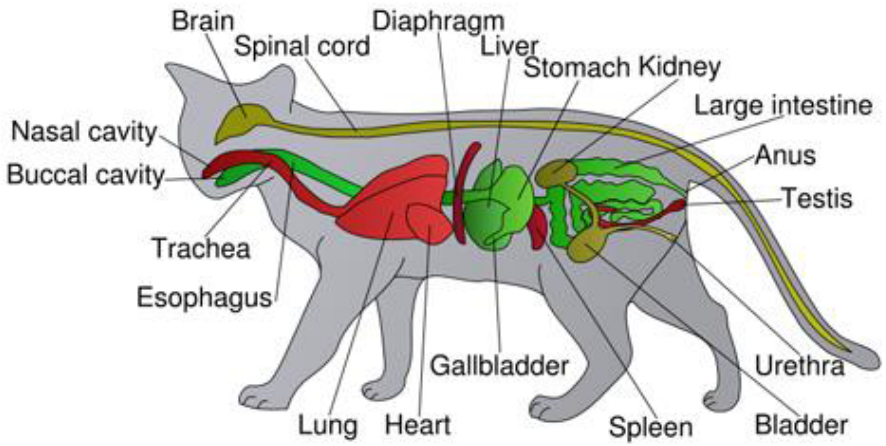
Принципы ООП

- **Абстрагирование (Abstraction)** означает выделение значимой информации и исключение из рассмотрения не значимой.

понятно, что делать с объектом
(кормить, что же еще)

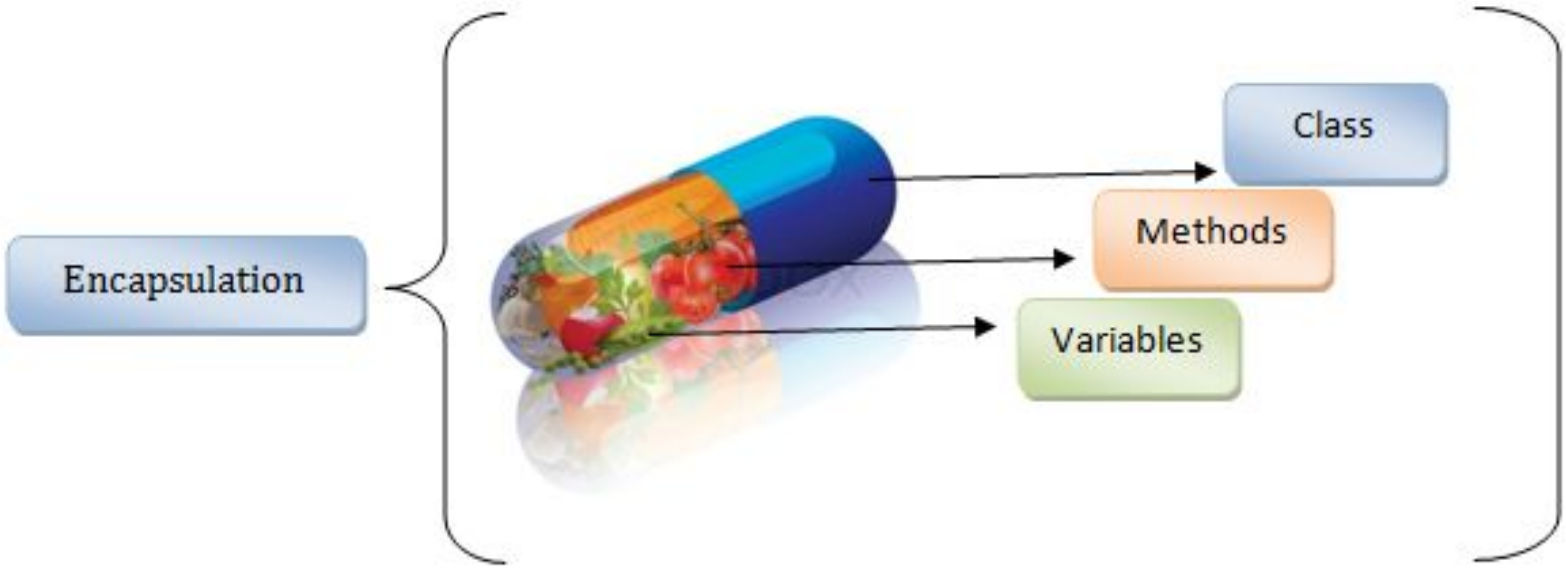


не понятно, как именно взаимодействовать с объектом
(слишком много деталей)



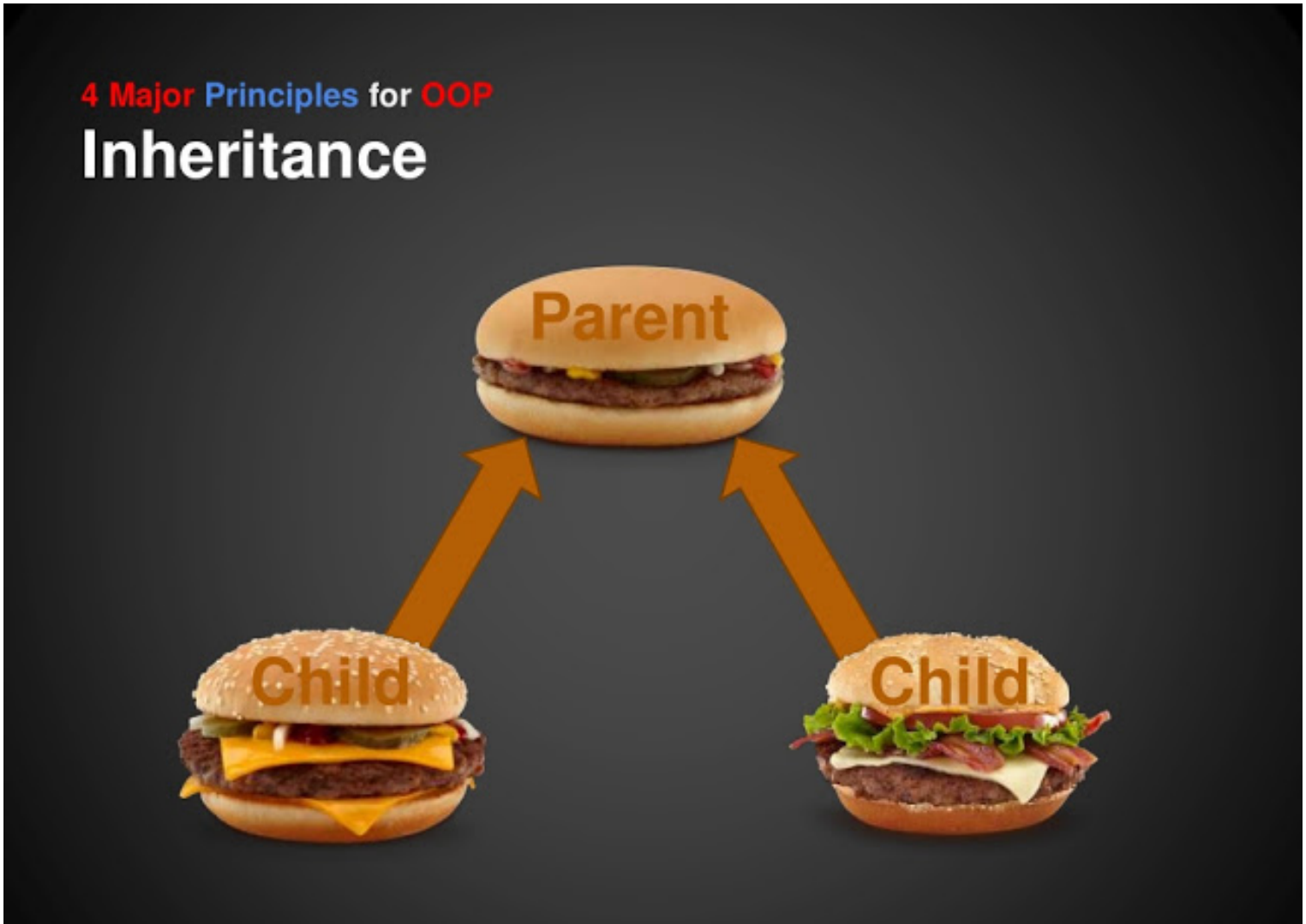
Принципы ООП

- **Инкапсуляция (Encapsulation)** - это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. Доступ к коду и данным жестко контролируется интерфейсом.



Принципы ООП

- **Наследование (Inheritance)** касается способности языка позволять строить новые определения классов на основе определений существующих классов.

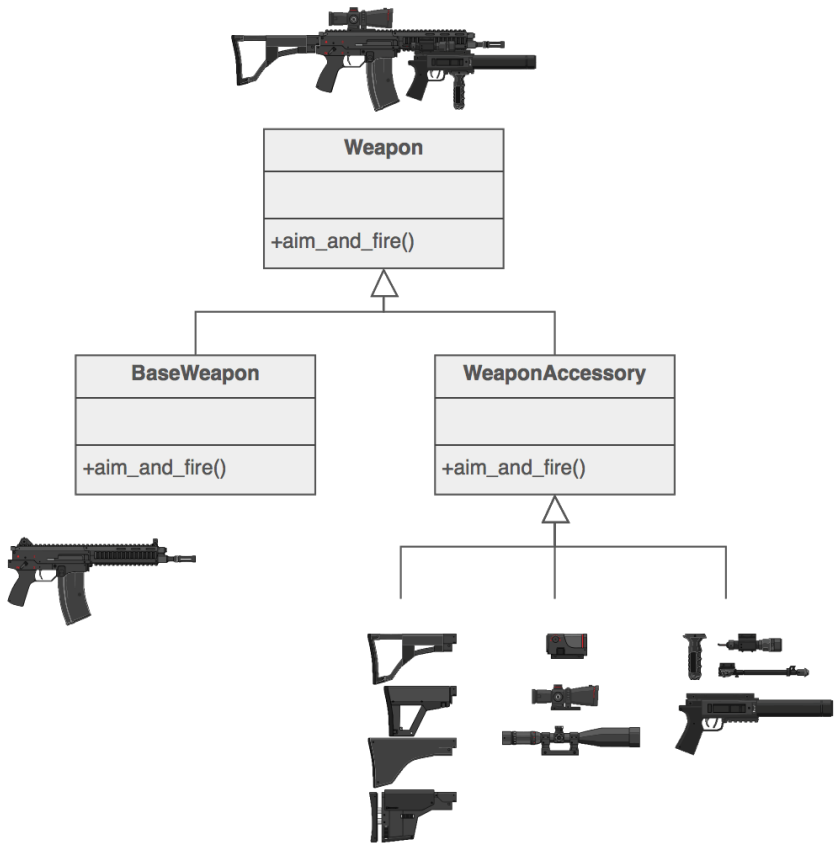


Принципы ООП

- **Полиморфизм (Polymorphism)** - свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Преимущества ООП

- Использование моделей из окружающего мира - объектов.
- Объектная декомпозиция.
- Повторное использование кода (наследование).
- Сокрытие сложности.



Классы и объекты

Класс — это элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию.

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

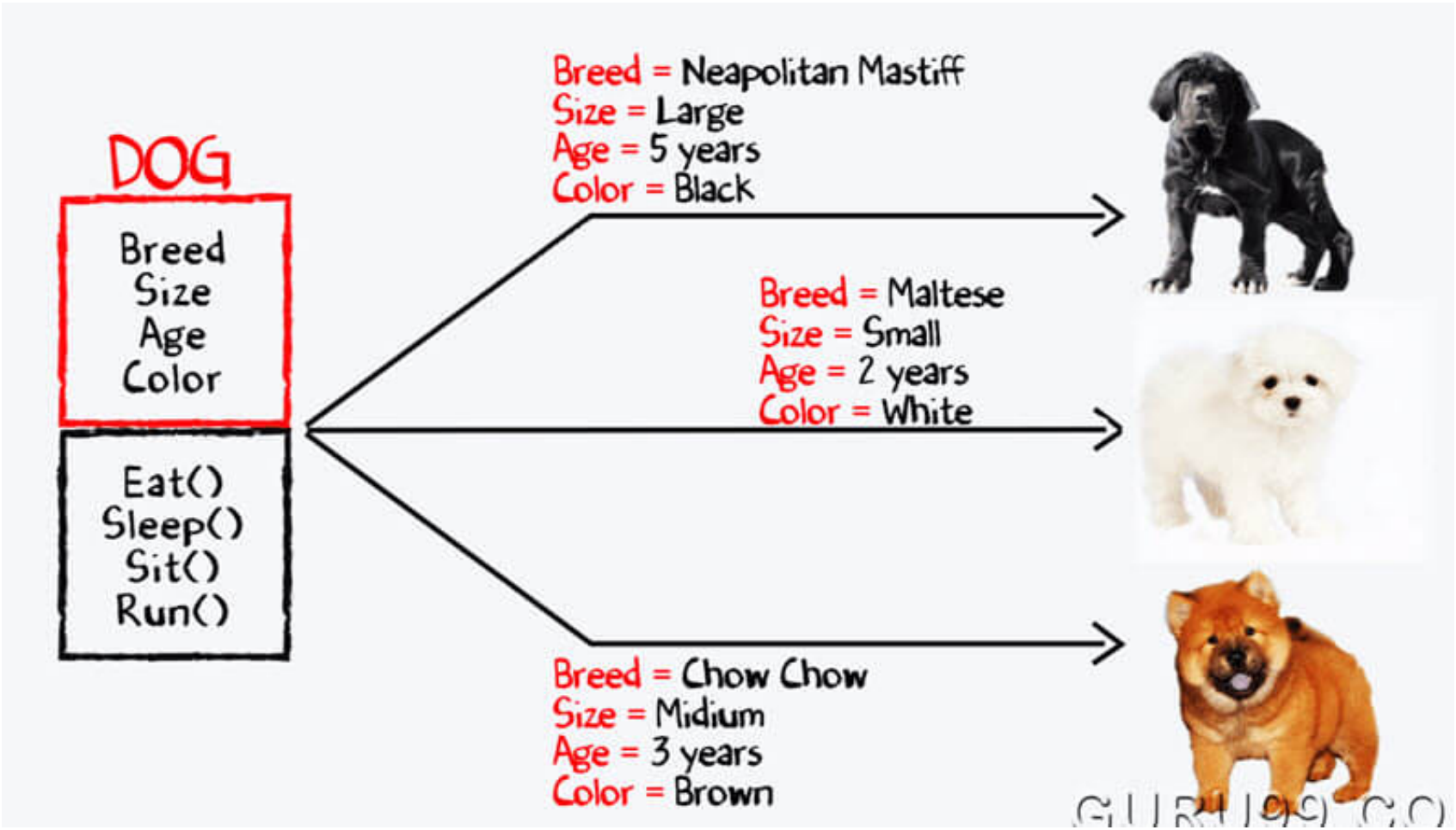
Классы и объекты

Объект - некоторая сущность в компьютерном пространстве, обладающая определённым состоянием и поведением, имеющая заданные значения свойств (атрибутов) и операций над ними (методов).

Объект - это экземпляр класса.

Если в классе может быть определён набор полей (свойств), то в объекте этим полям заданы значения.

Классы и объекты



Одному классу соответствуют много объектов.

Основные понятия

Методы класса — это его функции.

Свойства (атрибуты, поля, информационные члены класса) — его переменные.

Члены класса — методы и поля класса.

Основные понятия

Интерфейс совокупность средств, методов и правил взаимодействия (управления, контроля и т.д.) между элементами системы.

Интерфейс (ООП) - то, что доступно при использовании класса извне. Как правило это набор методов.

Основные понятия

Каждый объект характеризуется:

- Состояние - набор атрибутов, определяющих поведение объекта.
- Поведение - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.
- Идентичность (уникальность) - это такое свойство объекта, которое отличает его от всех других объектов. Два объекта идентичны если представлены одним и тем же участком памяти.
- Равенство (эквивалентность). Два объекта равны если содержат одинаковые данные.

Равенство и эквивалентность

```
string *s =  new string("ABC");
string *s1 = new string ("ABC");
string *s2 = s;
```

s и s1 равны

s и s2 эквивалентны (указывают на один и тот же участок памяти)

Объявление класса в C++

```
class ClassName {
private:
    // закрытые члены класса
    // рекомендуется для описания полей
public:
    // открытые (доступные из вне) члены класса
    // рекомендуется для описания интерфейса
protected:
    // защищенных члены класса
    // доступны только наследникам

// дружественные функции и классы
// модификатор доступа не важен
friend заголовок-функции;
friend имя_класса;
};
```

Классы и объекты. Пример. C++

```
class Book {
      public:
        string title;
        string author;
        unsigned pages;
    };

    // объекты (экземпляры класса Book)
    Book b1 = {"Code complite", "S. Macconell", 900};
    Book b2 = {"OOA and OOD", "Grady Booch", 897};
    Book b3 = {"Незнайка на Луне", "Носов. Н", 408};

    Book b4 = Book();
    Book b5();
    Book b6;
```

при создании объектов b1, b2 и b3 использованы списки инициализации. Такой способ инициализации подходит для иллюстрации создания класса, но нарушает инкапсуляцию.

Объекты b4, b5, b6 будут равны.

Классы и объекты. Пример. Python

```
class Book:
    title = ""
    author = ""
    pages = 0


b1 = Book()
b1.title = "Code complite"
b1.author = "S. Macconell"
b1.pages = 900
```

Описание полей и методов

- Поля и методы описываются также как и переменные и функции соответственно.
- Каждый класс неявно содержит специальный указатель на самого себя - `this`[1].
- Этот указатель неявно передаётся первым параметров в каждый метод[2].
- Если для члена класса не указан модификатор доступа, то считается что он private

[1] в python это `self`

[2] в python передаётся явно, например: `__init__(self, a, b)`

Доступ к членам класса

- доступ через объект или ссылку на объект - операция выбора члена класса "."

```
Book b1;
b1.author = "Станислав Лем";
```

- доступ с помощью указателя на объект - указатель на член класса "->"

```
Book *b2 = new Book();
b2->author = "Станислав Лем";
// аналогично
(*b2).author = "Станислав Лем";
```

Пример

```
class X {
private:
    int val1;
 public:
  float val2;
  int *vec;
  void foo() {
    this->val1 = 200;  // this - это указатель
    val1 = 200;  // this можно не указывать при обращении к членам клас
                }
};
...

X x;
X *xp = new X();
x.val1 = 10; // Ошибка! Поле val1 недоступно извне класса.
x.val2 = 10;
x.vec = NULL;
x.foo();
xp->val2 = 9000;
xp->foo();
```

Некоторые рекомендации.

Парадигма ООП

- Поля класса рекомендуется описывать в закрытой области класса (`private`).
- Для доступа к таким полям создавать методы для получения и задания значения (инкапсуляция).
- Эти методы должны быть доступны извне класса (`public`)
- Константность: методы не изменяющие состояние класса нужно помечать спецификатором const

```
void foo() const {
    ...
    }
```

Некоторые рекомендации

Стиль кодирования

- Классы рекомендуется называть используя верблюжью нотациию - `CamelCase`
- рекомендуемые имена методов для обращения к полю класса - см. пример
- Определение класса следует разделять на заголовочный (*.h) и cpp файл. В cpp файле должны приводится только определения (definition) методов.
- Имя заголовочного файла должно совпадать с именем класса.

Пример

Заголовочный файл MyClass.h

```
class MyClass{
    int _x;

  public:
    MyClass();

    int x() const;
    void setX(int x);

    void foo( int x, int y);
};
```

MyClass.cpp

```
MyClass::MyClass(){
  _x = 42;
  ...    }

int MyClass::x() const{
    return _x;}

void MyClass::setX(int x){
    // проверка входных данных
    // если всё OK:
    _x = x; }

void MyClass::foo(int x, int y){
    ... }
```

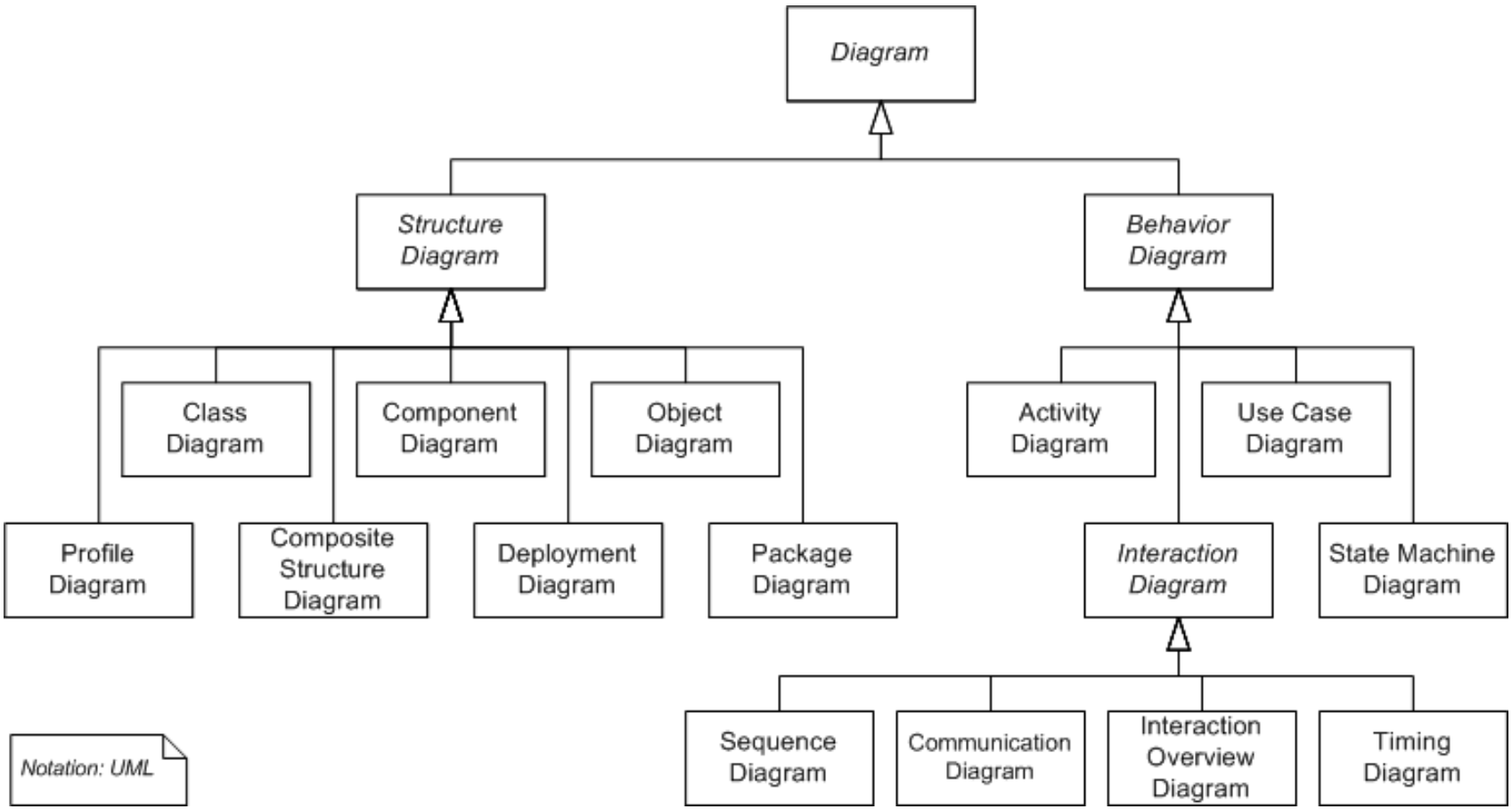
Outline

UML

UML (Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

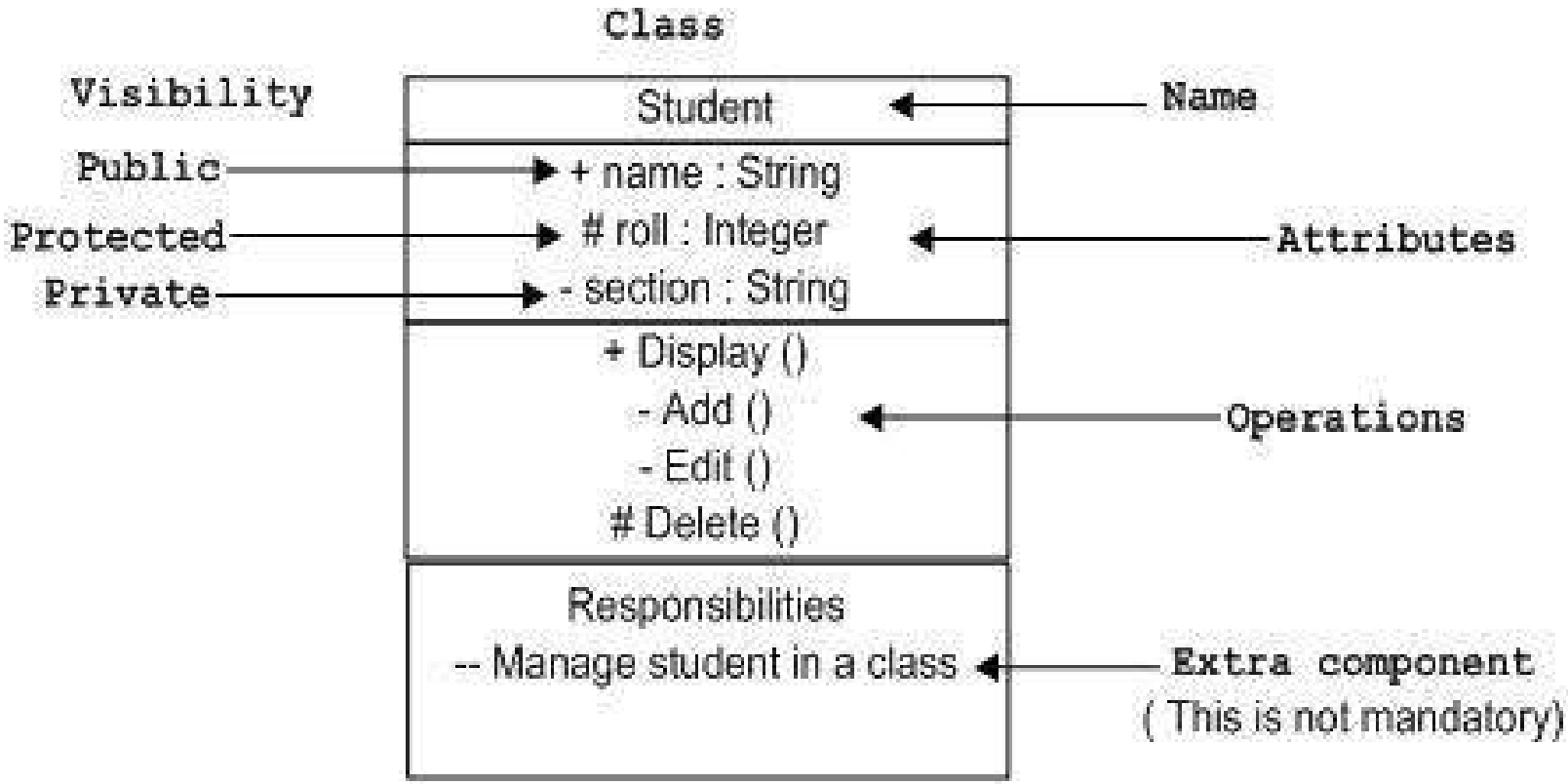
UML

Виды UML диаграмм представленные в виде UML диаграммы.



Класс на UML диаграмме

Класс на UML диаграмме

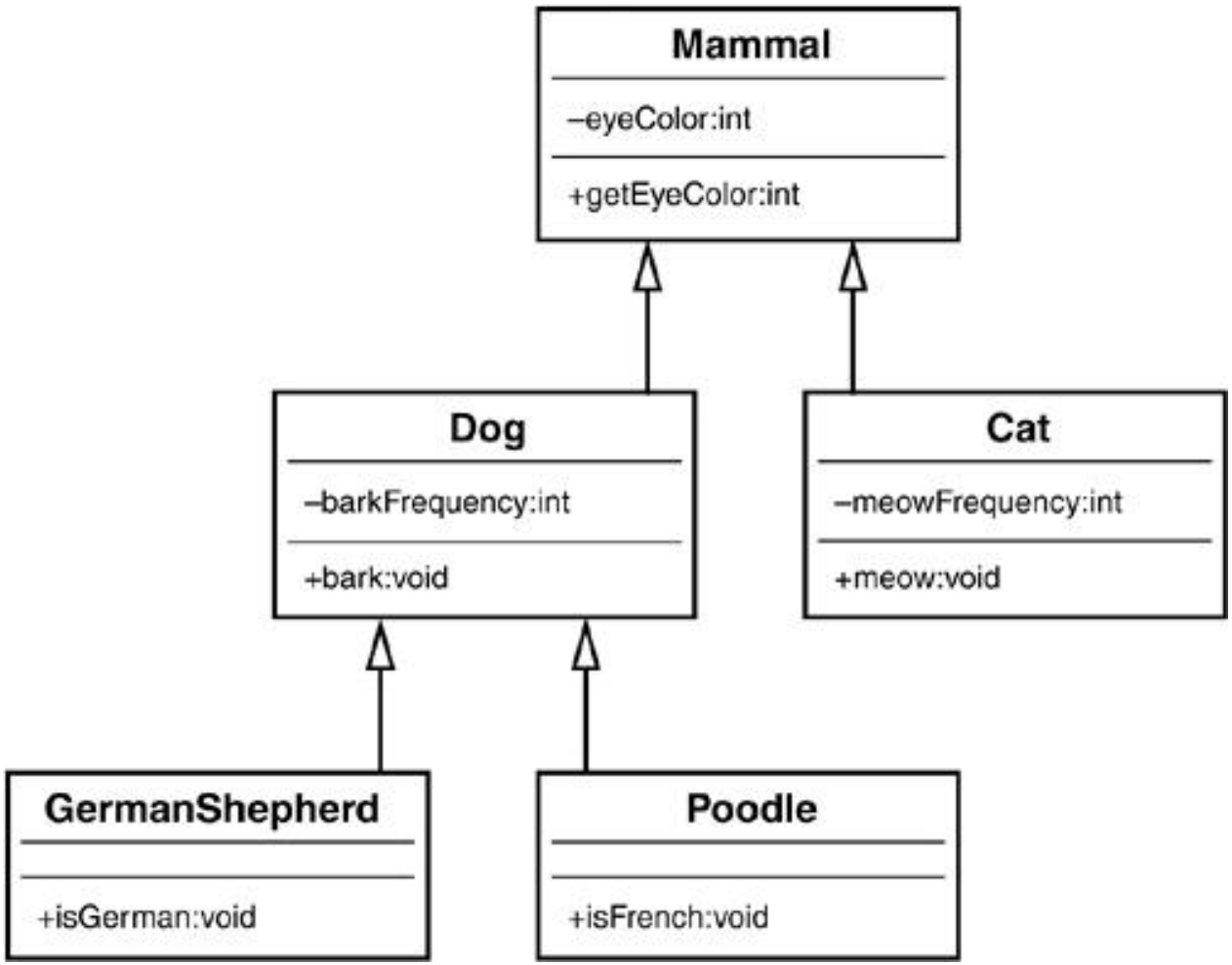


Отношения между классами

- обобщение/специализация (generalization/specialization)
  кошки - это животные
- целое/часть (whole/part)
  двигатель - часть автомобиля
- ассоциация (семантическая зависимость)
  художник - кисть

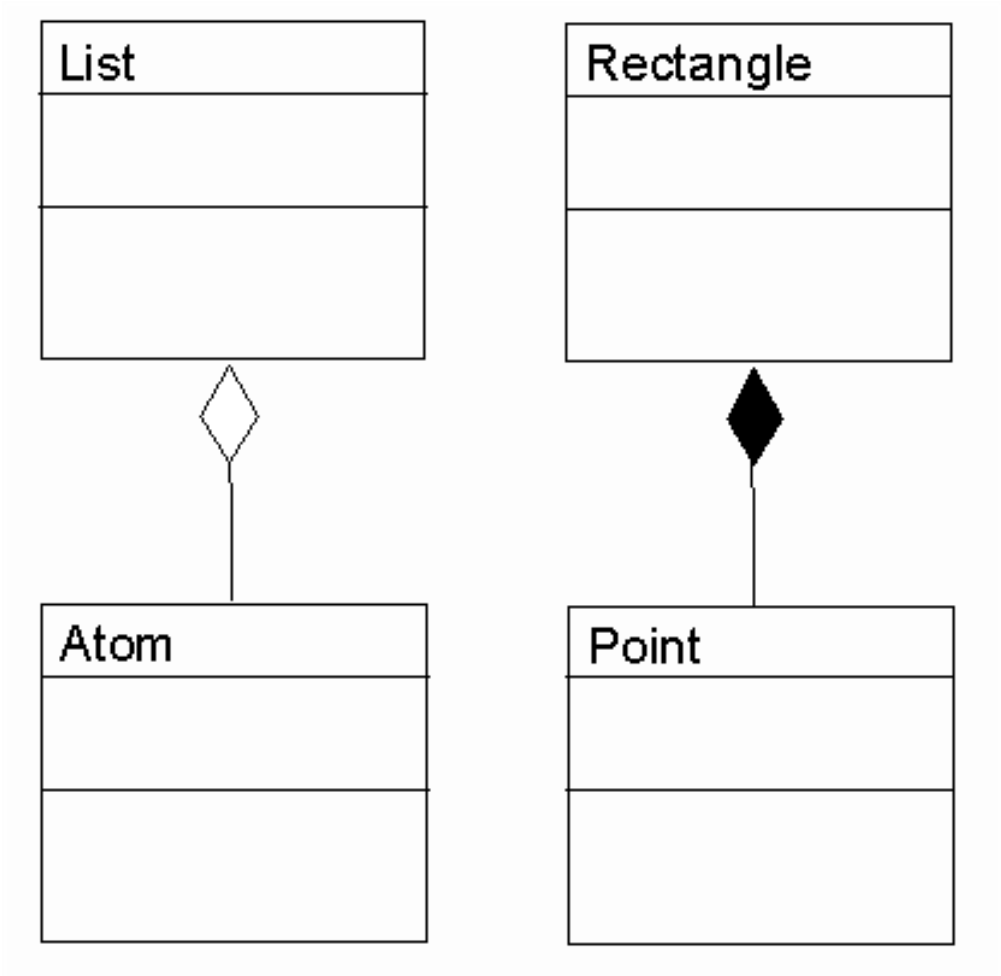
Обобщение\специализация

Наследование

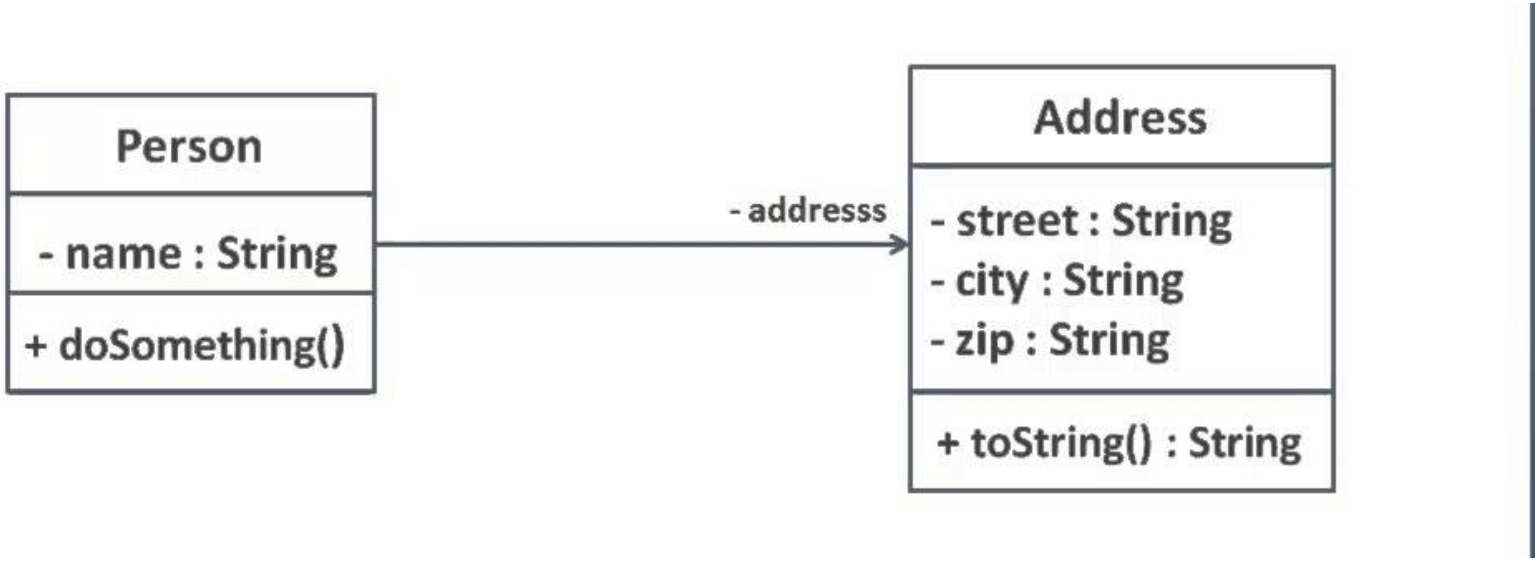


Часть\целое

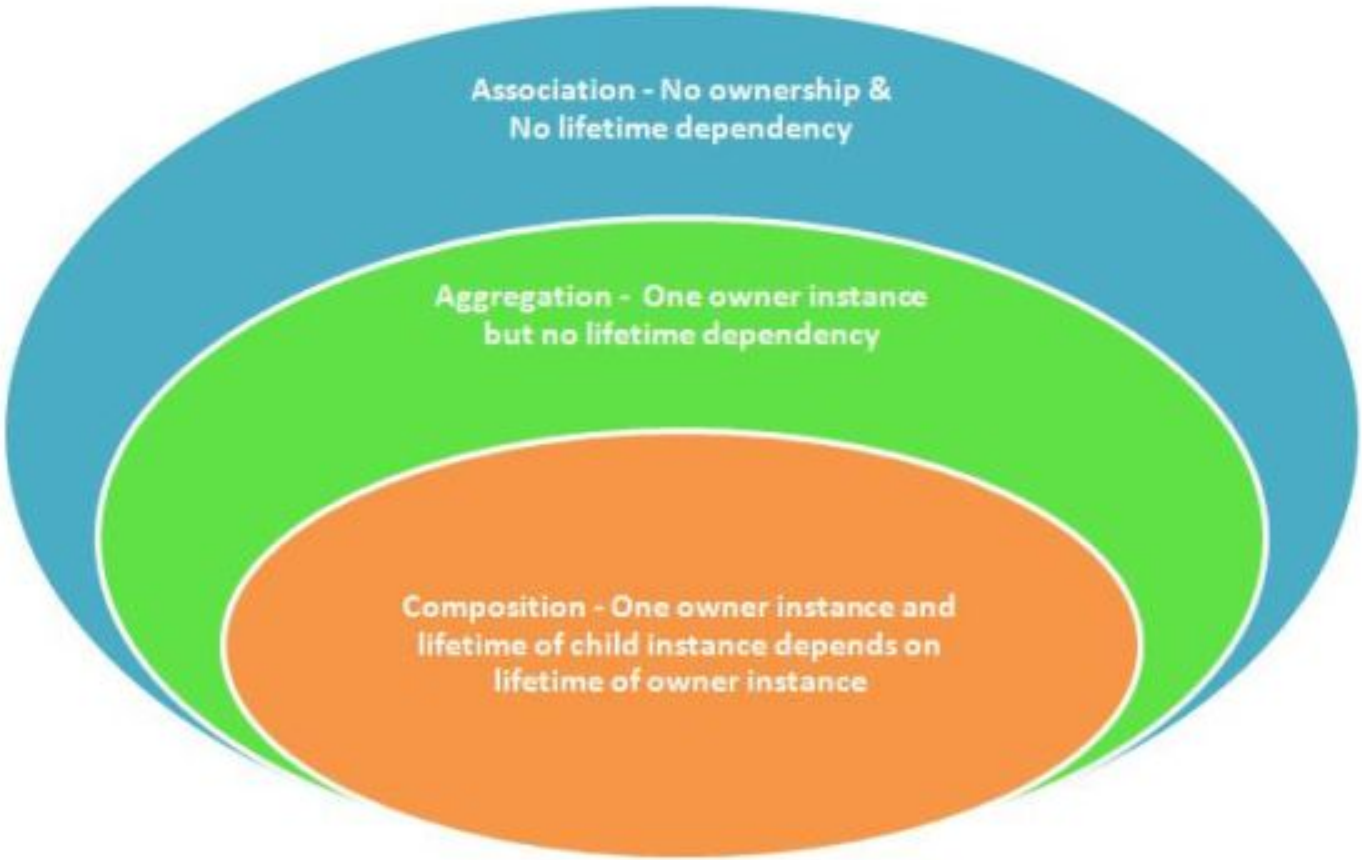
Агрегация и композиция



Ассоциация


Person
- name : String
+ doSomething()
- addresss
Address
- street : String
- city : String
- zip : String
+ toString() : String

Ассоциация - Агрегация - Композиция



Outline

Описание класса в C++

```
class ClassName {
private:
  // закрытые члены класса
  // рекомендуется для описания полей
public:
 // открытые (доступные из вне) члены класса
 // рекомендуется для описания интерфейса
protected:
  // защищенных члены класса
  // доступны только наследникам

// дружественные функции и классы
// модификатор доступа не важен
friend заголовок-функции;
friend имя_класса;
};
```

Outline

Конструктор

Конструктор — это особый метод, инициализирующий экземпляр своего класса.

- Имя конструктора совпадает с именем класса.
- У конструктора может быть любое число параметров.
- У класса может быть любое число конструкторов.
- Конструкторы могут доступными (public), защищенными (protected) или закрытыми (private).
- Если не определено ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров (а также некоторые другие к. и оператор присваивания)

Конструктор по умолчанию (Default constructor)

`MyClass()`

- Не имеет параметров.
- Может быть только один.
- Может отсутствовать.
- Может быть реализован компилятором.

Когда вызывается

```
class MyClass {...};
...

MyClass c0 = MyClass();
MyClass c1;
MyClass cv[16];       // к. будет вызван 16 раз
list<MyClass> cl(10) // к. будет вызван 10 раз
```

Конструктор преобразования (Conversion constructor)

`MyClass(T t)`

конструкторы с двумя и более параметрами

Конструктор копирования (copy constructor)

```
MyClass(MyClass &c)
```

Конструктор перемещений (move constructor)

```
MyClass(MyClass &&c)
```

Конструкторы

- Конструктор по умолчанию (default constructor)
- Конструкторы преобразования (conversion constructors)
  - Конструкторы с параметрами (parameterized constructor)
- Конструктор копирования (copy constructor)
- Конструктор перемещения (move constructor)

Оператор присваивания копированием (assignment operator)

```
MyClass& operator=(MyClass& data)
```

- используется для присваивания одного объекта текущему (существующему)
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Оператор присваивания перемещением (move assignment operator)

```
MyClass& operator = (const MyClass &c)
```

- используется для присваивания *временного* объекта существующему
- "забирает"временный объект "в себя"; временный объект перестаёт существовать
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Когда вызывается?
Когда существующему объекты присваиваю значение временного объекта.

Правило пяти

Если класс или структура определяет один из следующих методов, то они должны явным образом определить все методы:

- Конструктор копирования
- Конструктор перемещения
- Оператор присваивания копированием
- Оператор присваивания перемещением
- Деструктор

Спецификаторы default и delete

Спецификаторы **default** и **delete** заменяют тело метода.

Спецификатор **default** означает реализацию по умолчанию (компилятором). Может быть применён только к конструкторам, деструктору и операторам присваивания.

Спецификатором **delete** помечают те методы, работать с которыми нельзя.

Спецификаторы default и delete

```
class Foo{
public:
    Foo() = default;
    Foo(const Foo&) = delete;
    Foo operator = (const Foo& f) = delete;
};

...

Foo o1, o2; // вызов констр. созданного компилятором
o1 = o2; // Ошибка компиляции! Оп-р присваивания запрещён.
Foo o3(01); // Ошибка компиляции! Констр. копирования запре
```

Вопросы

- Зачем нужны конструкторы?
- Как запретить создание объекта на основе уже существующего?
- Как запретить любой другой способ создания объекта?
- Зачем нужны конструкторы перемещения? В чём их отличие от к. копирования?
- Когда вызывается конструктор, а когда оператор присваивания?
- Что если не описать ни одного конструктора?
- Что если не описать ни одного оператора присваивания?

Outline

Перегрузка операторов

```
Type operator opr ( parameters );
```

type - тип возвращаемого значения
opr - обозначение оператора, например * или =
parameters - параметры, описываются также как и для метода

Число параметров функции должно соответствовать арности оператора. Например для бинарных операторов параметра два.

Когда перегруженный оператор является методом класса, тип первого операнда должен быть указателем на данный класс (всегда *this), а второй должен быть объявлен в списке параметров.
Операторы в C и C++

Перегрузка операторов overloading

```
class T{
...
    public:
        T operator+ (const T&) const { ... }
};


...

T a,b,c;

// этот код будет транслирован компилятором в
c = a + b;

// этот
c = a.operator+(b);
```

Перегрузка операторов (overloading)

Когда оператор делать методом, а когда дружественной функцией?

Унарные операторы и бинарные операторы типа “X=” рекомендуется реализовывать в виде методов класса, а прочие бинарные операторы — в виде дружественных функций.
Так стоит делать потому, что оператор-метод всегда вызывается для левого операнда.

Outline

Outline

Наследование (Inheritance)

Наследование - построение новых классов на основе уже существующих.

Базовый класс (предок) — класс на основе которого строится определение нового класса - **производного класса (потомка)**.

Наследование. Пример

```
class B{
    int x_;
  protected:
    int y;
  public:
    B() { cout "Base constructor";
      x_ = 42; y = 9000;}
    void setX(int x_) {x = x_;}
    int x() const {return x_;}
    int getY() {return y;}
    void foo() const {cout << "Base";}
};
```

```
class D : public B{
// поле x_ унаследовано,
// но к нему нет прямого доступа
// к полю y есть прямой доступ
// только внутри этого класса
public:
// в списке инициализации возможен
// вызов конструктора базового класса
  D() : B() { setX(1729); }
  void bar() const
    {cout << "Delivered";}
};
```

```
B b;
D d;
a.foo();   // Base
// вызов унаследованного метода
b.foo();   // Base
b.bar();   // Delivered
b.getX();  // 1729
b.getY();  // 9000
```

Outline

Перекрытие имен (overriding)

Перекрытие имён в классе = переопределение имен

```
    class B {
          public:
                void foo(){cout << "base";}
        };

class D: public B{
  public:
        void foo(){cout << "delivered";}
};

B b;
D d;
b.foo(); // base
d.foo(); // delivered
// Если нужно вызывать метод из базового класса,
// то явно указывается имя этого класса
d.B::foo(); // base
```

Множественное наследование

Множественное наследование - наследование от нескольких классов одновременно.

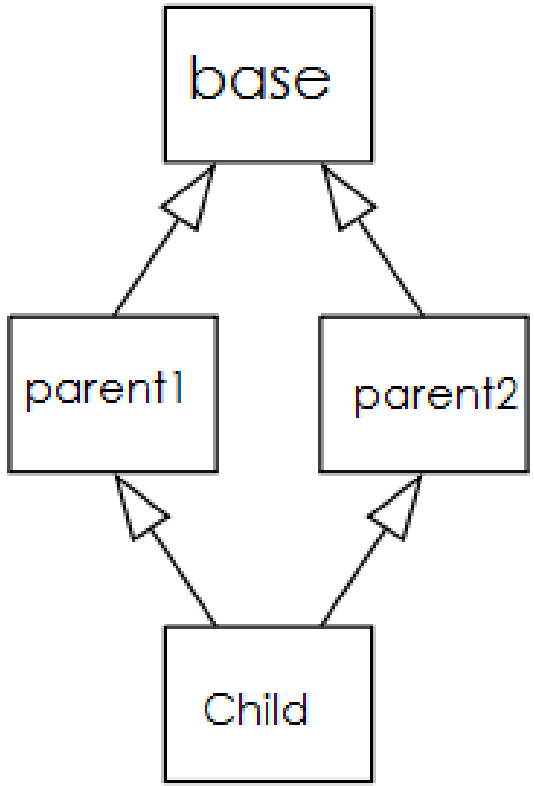
```
class Z: public X, public Y { . . . };
```

При множественном наследовании возникает проблема неоднозначности из-за совпадающих имен в базовых классах.

Поэтому лучше наследоваться от интерфейсов и классов-контейнеров.

Deadly Diamond of Death

Проблема ромба [wiki]



если метод класса Child вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C ?

Outline

Outline

Виртуальные методы

Виртуальный метод - метод, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Чистый виртуальный (абстрактный) метод - виртуальный метод для которого не приведена реализация.

Виртуальные методы

Зачем нужны?

- Реализует динамический полиморфизм
- Упрощает интерфейс

У целого набора классов может быть метод с одним именем и набором параметров (или несколько таких), который решает одну и ту же задачу, но специфичным для каждого класса способом. Какая конкретно реализация метода должна быть вызвана определяется во время выполнения программы.

Ранее и позднее связывание

Статическая типизация (раннее связывание) — определение типа на этапе компиляции.

Динамическая типизация (позднее связывание) — определения типа во время выполнения программы.

Динамический полиморфизм

Реализуется с помощью виртуальных методов.

Вопросы

- Чем отличается переопределение виртуальных методов от переопределения виртуальных?
- Для чего нужен динамический полиморфизм? Приведите примеры.
- Как задействовать позднее связывание?

Ссылки и литература

1. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений. 720 с. 2010 г. 700 страниц. Теория. Примеры на C++. Картинки! Вторая половина книги - примеры OOA и OOD с UML диаграммами.
2. MSDN - Microsoft Developer Network
3. Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 и более поздние издания г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
4. www.stackowerflow.com - система вопросов и ответов
5. draw.io — создание диаграмм.

Материалы курса

Слайды, вопросы к экзамену, задания, примеры

github.com/VetrovSV/OOP