1. 面向对象设计

> 面向对象设计主要就是使用UML的类图，类图用于描述系统中所包含的类以及它们之间的相互关系，帮助人们简化对系统的理解，它是**系统分析和设计阶段的重要产物，也是系统编码和测试的重要模型依据。**

2. 类的UML画法

> **类（Class）封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称。**在系统中，每个类都具有一定的职责，职责指的是类要完成什么样子的功能，要承担什么样子的义务。一个类可以有多种职责，但是设计得好的类一般只有一种职责。
>
> 假如我现在定义了这么一个类：

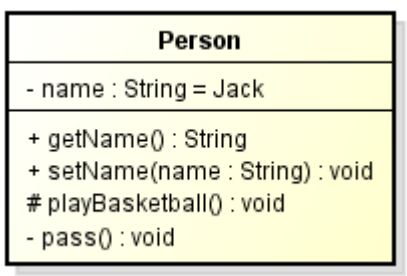
```
class Persion
{
public:
    string getName()
    {
        return name;
    }

    void setName(string name)
    {
        this->name = name;
    }

protected:
    void playBasketball()
    {
        pass();
    }

private:
    void pass()
    {
    }
private:
    string name = "Jack";
};
```

那么此类对应的UML为：



> 看到该图分为三层：最顶层的为类名，中间层的为属性，最底层的为方法。
>
> 属性的表示方式为：【可见性】【属性名称】：【类型】={缺省值，可选}
>
> 方法的表示方式为：【可见性】【方法名称】（【参数列表】）：【类型】
>
> 可见性都是一样的，**"-"表示private、"+"表示public、"#"表示protected**。

3. 继承关系

> 继承也叫作泛化（Generalization），用于描述父子类之间的关系，父类又称为基类或者超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的**实线**来表示。

3.1 普通继承关系

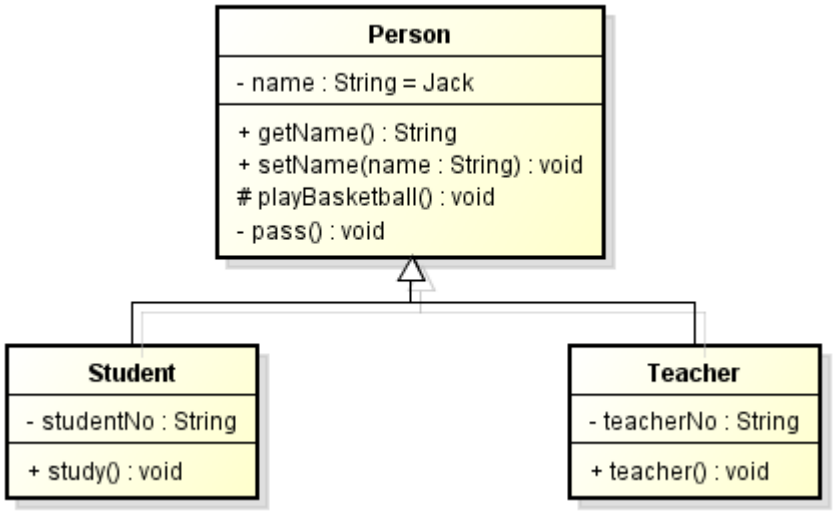
> 假如现在我又定义了一个Student和一个Teacher：

```
class Student : public Persion
{
public:
    void study()
    {
    }
private:
    string studentNo;
};
```

```
class Teacher : public Persion
{
public:
    void teach()
    {

    }
private:
    string teacherNo;
};
```

> 那么，用UML表示这种关系应当是：



3.2 抽象继承关系

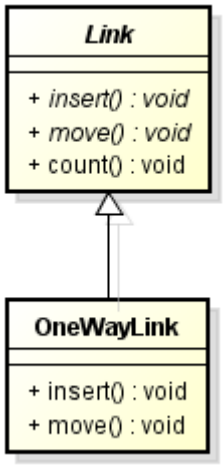
> 上面的继承是普通的继承，在C++中，除了普通的继承之外，众所周知的还有一种抽象的继承关系，因此就再讲讲抽象继承关系，作为上面的继承的补充。
>
> 比方说我想实现一个链表（Link），插入（insert）与删除（remove）动作我想让子类去实现，链表本身只实现统计链表中元素个数的动作（count），然后有一个子类单向链表（OneWayLink）去实现父类没有实现的动作，C++代码为：

```
// 抽象类(含有纯虚函数的类)
class Link
{
public:
    virtual void insert() = 0;
    virtual void remove() = 0;

    int count()
    {
        return 0;
    }
};
```

```
// 子类
class OneWayLink : public Link
{
public:
    void insert()
    {
    }
    void remove()
    {
    }
};
```

> 其UML的画法为：



> 在UML中，抽象类无论类名还是抽象方法名，都以**斜体**的方式表示，因为这也是一种继承关系，所以子类与父类通过带空心三角形的实线来联系。

4. 关联关系

> 关联（Assocition）关系是类与类之间最常见的一种关系，它是一种结构化的关系，表示一类对象与另一类对象之间有联系，如汽车和轮胎、师傅和徒弟、班级和学生等。在UML类图中，用实线连接有关联关系的对象所对应的类，**在C++中通常将一个类的对象作为另一个类的成员变量。**关联关系分单向关联、双向关联、自关联，逐一看一下。

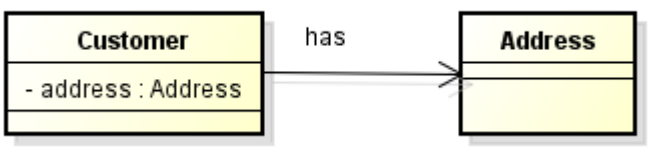
4.1 单向关联关系

> 单向关联指的是关联只有一个方向，比如顾客（Customer）拥有地址（Address），其代码实现为：

```
// 地址类
class Address
{
};
```

```
// 顾客类
class Customer
{
private:
    Address address;    // 作为成员变量
};
```

UML的画法为：



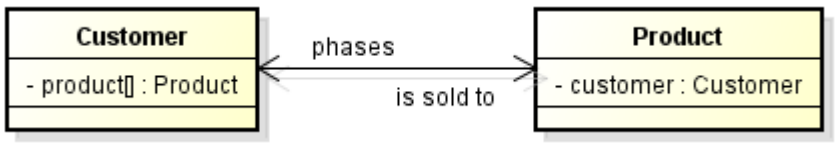
4.2 双向关联关系

> 默认情况下的关联都是双向的，比如顾客（Customer）购买商品（Product），反之，卖出去的商品总是与某个顾客与之相关联，这就是双向关联。c++ 类的写法为：

```
// 商品类
class Product
{
private:
    Customer customer;  // 该商品属于哪一位顾客，作为成员变量
};
```

```
// 顾客类
class Customer
{
private:
    Product product[64];    // 给顾客购买了哪些商品，作为成员变量
};
```

对应的UML类图应当是：

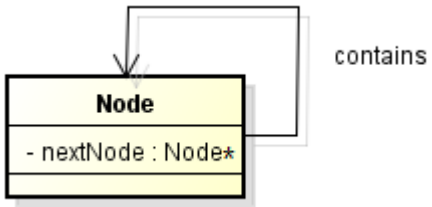


4.3 自关联关系

> 自关联，指的就是对象中的属性为对象本身，这在链表中非常常见，单向链表Node中会维护一个它的前驱Node，双向链表Node中会维护一个它的前驱Node和一个它的后继Node。就以单向链表为例，它的C++写法为：

```
// 链表节点
class Node
{
private:
    Node* nextNode;     // 指向后继节点的指针，作为成员变量
};
```

对应的UML类图应当是：



5. 聚合关系

> 聚合（Aggregation）关系表示**整体**与**部分**的关系。在聚合关系中，**成员对象是整体的一部分，但是成员对象可以脱离整体对象独立存在**。在UML中，聚合关系用带空心菱形的直线表示，如汽车（Car）与引擎（Engine）、轮胎（Wheel）、车灯（Light），C++ 表示为：

```
class Engine
{
};
```

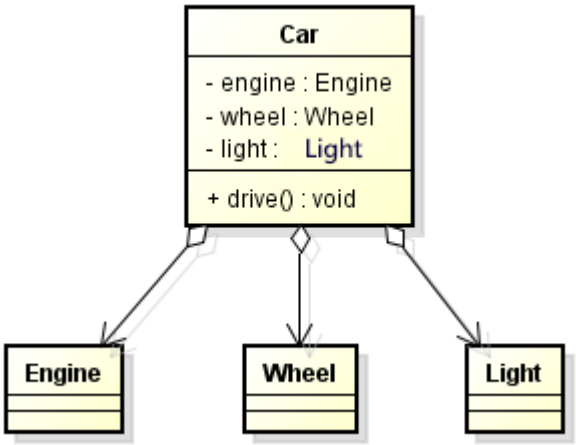
```
class Wheel
{
};
```

```
class Light
{
};
```

```
class Car
{
public:
    Car(Engine engine, Light light, Wheel wheel)
    {
        this->engine = engine;
        this->light = light;
        this->wheel = wheel;
    }
    void drive()
    {

    }
private:
    Engine engine;
    Light light;
    Wheel wheel;
};
```

对应的UML类图为：



> 代码实现聚合关系，**成员对象通常以构造方法、Setter方法的方式注入到整体对象之中。**

6. 组合关系

> **组合（Composition）关系也表示的是一种整体和部分的关系**，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也不存在，**整体对象和成员对象之间具有同生共死的关系。**在UML中组合关系用带实心菱形的直线表示。
>
> 比如人的头（Head）和嘴巴（Mouth）、鼻子（Nose），嘴巴和鼻子是头的组成部分之一，一旦头没了，嘴巴也没了，因此头和嘴巴、鼻子是组合关系，C++ 表示为：

```
class Mouth
{

};
```

```
class Nose
{

};
```

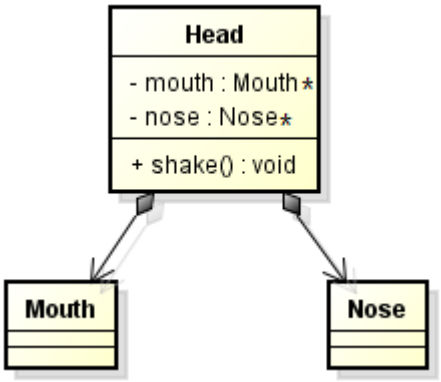
```
class Head
{
public:
    Head()
    {
        mouth = new Mouth();
        nose = new Nose();
    }

    void shake()
    {

    }
private:
    Mouth *mouth;
    Nose  *nose;
```

```
};
```

其UML的表示方法为：



> 代码实现组合关系，通常在**整体类的构造方法中直接实例化成员类**，因为组合关系的整体和部分是共生关系，如果通过外部注入，那么即使整体不存在，那么部分还是存在的，这就相当于变成了一种聚合关系了。

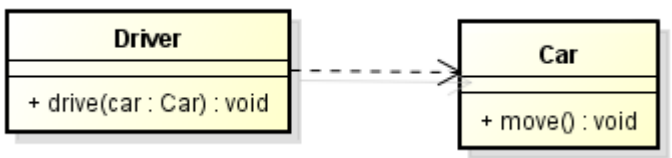
7. 依赖关系

> **依赖（Dependency）关系是一种使用关系**，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系，大多数情况下依赖关系体现在某个类的方法使用另一个类的对象作为参数。在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。
>
> 比如，驾驶员（Driver）开车，Driver类的drive()方法将车（Car）的对象作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的drive()方法依赖车的move()方法，因此也可以说Driver依赖Car，C++代码为：

```
class Car
{
public:
    void move();
};
```

```
class Driver
{
public:
    void drive(Car car)
    {
        car.move();
    }
};
```

其UML的画法为：



> 依赖关系通常通过三种方式来实现：
>
> 1. **将一个类的对象作为另一个类中方法的参数**
> 2. **在一个类的方法中将另一个类的对象作为其对象的局部变量**
> 3. **在一个类的方法中调用另一个类的静态方法**

8. 关联关系、聚合关系、组合关系之间的区别

> 从上文可以看出，关联关系、聚合关系和组合关系三者之间比较相似，本文的最后就来总结一下这三者之间的区别。
>
> 关联和聚合的区别主要在于语义上：**关联的两个对象之间一般是平等的，聚合则一般是不平等的。**
>
> 聚合和组合的区别则在语义和实现上都有差别：**组合的两个对象之间生命周期有很大的关联，被组合的对象在组合对象创建的同时或者创建之后创建，在组合对象销毁之前销毁，一般来说被组合对象不能脱离组合对象独立存在，而且也只能属于一个组合对象；聚合则不一样，被聚合的对象可以属于多个聚合对象。**
>
> 再举例子来说：
>
> - 你和你的朋友属于关联关系，因为你和你的朋友之间的关系是平等的，关联关系只是表示一下两个对象之间的一种简单的联系而已，就像我有一个朋友
> - 你和你借的书属于聚合关系，第一是因为书可以独立存在，第二是因为书不仅仅属于你，也可以属于别人，只是暂时你拥有
> - 你和你的心脏属于组合关系，因为你的心脏只是属于你的，不能脱离与你而存在
>
> 不过，实际应用中，我个人感觉三种关系其实没有区分得这么清楚，有些架构师甚至会说"组合和聚合没什么区别"，所以，有时候不需要把细节扣得这么细，合理利用对象之间的关系给出设计方案即可。