依赖(dependency)关系：
对于两个相对独立的对象，当一个对象负责构造另一个对象的实例，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系。可以简单的理解，就是一个类 A 使用到了另一个类 B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是 B 类的变化会影响到 A。
例如：机器生产零件，充电电池通过充电器来充电，自行车通过打气筒来充气，人借助螺丝刀拧螺丝，人借用船过河，艺术家鉴赏艺术品，警察抓小偷，小猫钓鱼，学生读书，某人买车。
表现在代码层面，类 B 被类 A 在某个方法中使用，例如：局部变量，方法中的参数，对静态方法的调用等。

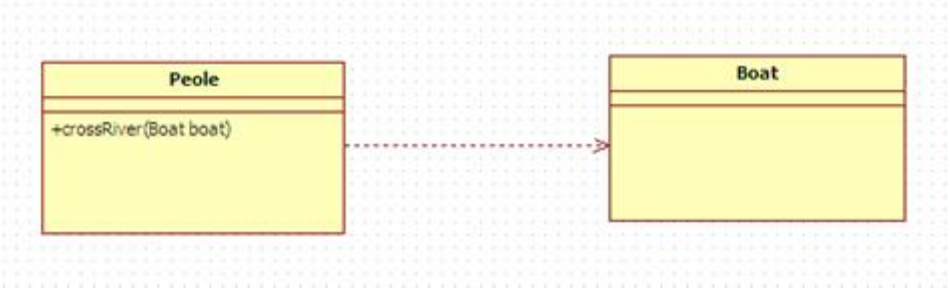
```
class B{...}
class A{...}
A::Function1(B &b) //或 A::Function1(B *b) //或 A::Function1(B b)
//或 B* A::Function1()    //或 B& A::Function1()
//或 int A::Function1()
      { B* pb = new B; /* ... */    delete pb; }
//或 int A::Function2()
      { B::sf(); }
```

```
// Car.h
class CCar
{
      // Do something
};
```

```
// Person.h
#include "Car.h"
class CPerson
{
      void MoveFast(CCar &pCar);
};
```

人的快速移动需要有车的协助, 但是这种依赖是比较弱的, 就是人也可以不用车而用其他工具, 与关联不同的是人不必拥有这辆车只要使用就行。

UML 表示方法：虚线箭头，类 A 指向类 B。



单向依赖例：一只老鼠在吃苹果，老鼠每吃一个苹果，就根据该苹果的所含能量增加一些体重。结合上边给出的模型，设计老鼠和苹果类。

```
class Apple
{
public:
        Apple(int e):power(e) {                         }
        int Energy( ) const      { return power;      }
private:
        int power;
};

class Mouse
{
public:
        Mouse(int w):weight(w) {           }
        int Weight( ) const        { return weight;}
        void Eat(Apple * one)    { weight += one->Energy() * 0.5; }
private:
        int weight;
};
```

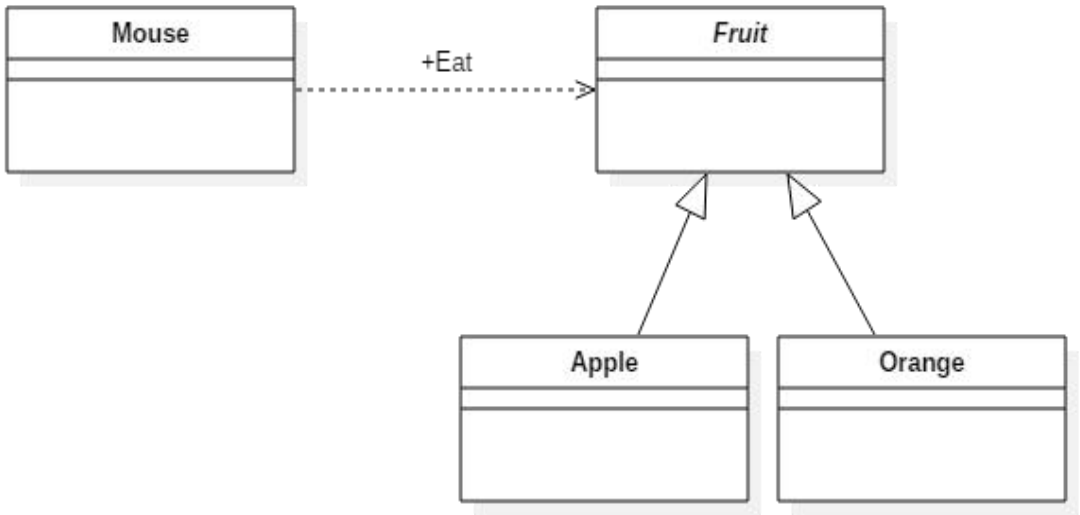
若老鼠不仅仅吃苹果，还吃香蕉、葡萄，怎么办？

```
void Eat(Apple * one);
void Eat(Banana* one);
void Eat(Grape* one);
```

又新增一种水果桃子，怎么办？

```
void Eat(Peach* one);
```

通常的面向对象设计，**通过子类型化适应变化**：




```
class Mouse {
public:
      void Eat(Fruit& fruit) ;
};
```

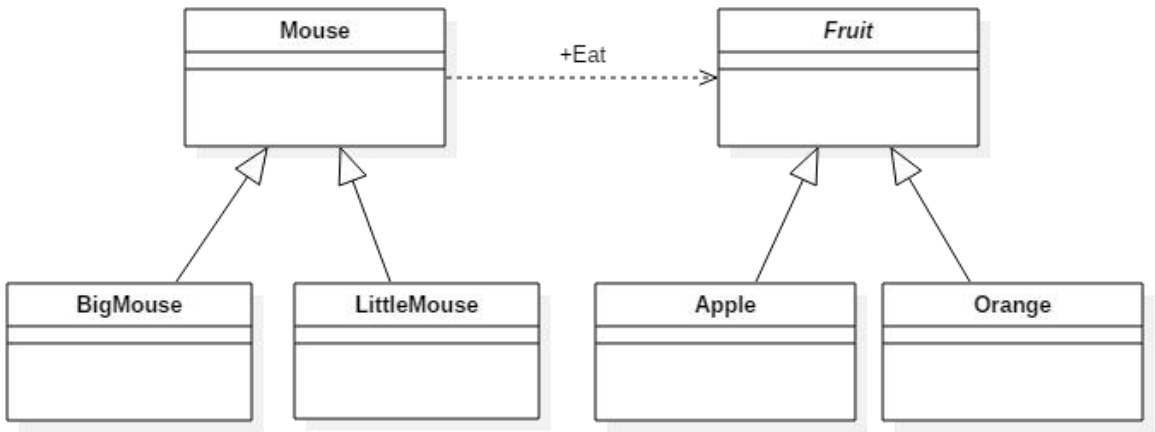
扩展 1：


```
class Parent;
class A {
public:
        virtual ~A( );
        void     Func(Parent *    p);
};
```

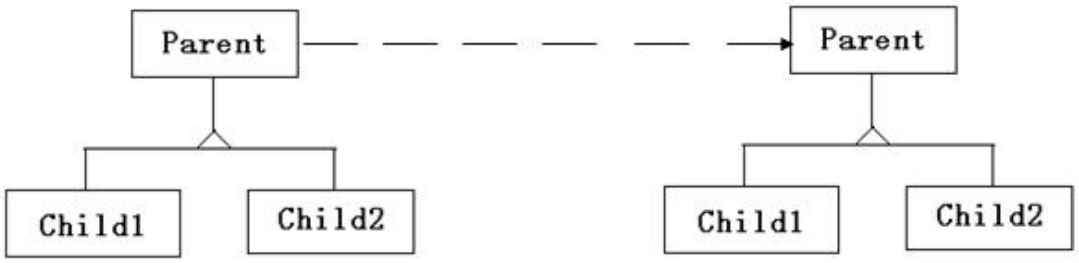
例：Mouse (eat) Fruit
例：Student (Do) Homework
例：Screen (draw) Shape's area
例：Printer (print) Date

面向对象设计需要尽可能地预知各种变化，而这离不开领域知识和领域经验。

进一步，若老鼠也有不同种类，有大老鼠、小老鼠等，怎么办？



扩展 2：



例：Human (read) Book

```
class Human {
public:
        virtual ~Human( ) {   }
        virtual void read( Book* );
};
class Student : public Human {   }
class Robot:   public Human {   }

class Book {
public:
        virtual ~Book( ) {   }
        // …..
};
class   Novel : public Book {   }
class   Cartoon:   public Book {    }
```

自身依赖例：一个游戏中有很多怪物(Monster)，怪物之间可能要发生战斗(fight)，每场战斗都是一个怪物与另一怪物之间的一对一战斗。每个怪物都有自己的速度(Speed)、生命值(hitpoint)、攻击力值(damage)和防御力值(defense)；战斗时，两个怪物依次**攻击**(attack)对方，即怪物 a 首先攻击怪物 b，然后轮到怪物 b 攻击怪物 a，之后，怪物 a 再次攻击怪物 b，…，直到一方生命值为 0；战斗时，由速度快的一方首先发起攻击；若速度一样，比较生命值，由高者首先攻击；若生命值也相等，比较攻击力，由高者首先攻击；若攻击力还相等，比较防御力，由高者首先攻击；若四项都相等，则选择任一方首先攻击；怪物 A 攻击怪物 B 时，会给怪物 B 造成伤害，使得怪物 B 的生命值降低，降低值为：2*A 的攻击力-B 的防御力，最小为 1。请根据你对上述描述的理解，定义并实现怪物类 Monster，成员的设计可以任意，但要求该类至少有一个成员函数 fight，用来描述与另外一个怪物进行战斗的过程。不必考虑怪物的生命值减少至 0 后如何处理。

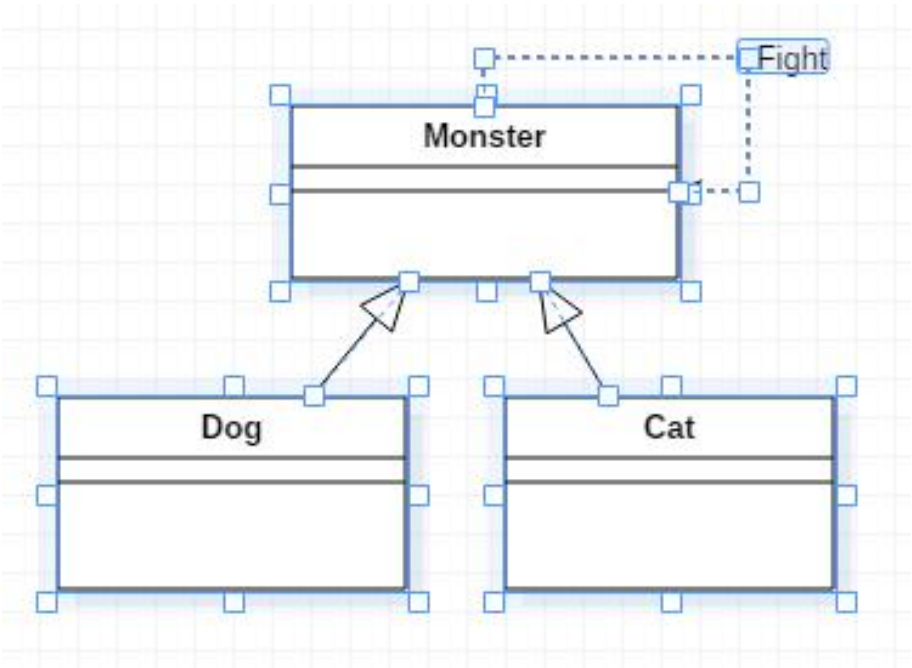
```
class Monster {
public:
    Monster(int spd, int hp, int dam, int def);
    bool Fight(Monster& other);
private:
    virtual int Attacked(Monster& other) const;
    virtual bool PriorTo(const Monster& other) const;
```

```
private:
    int speed;
    int hitpoint;
    int damage;
    int defense;
};

Monster::Monster(int spd, int hit, int dam, int def) :
    speed(spd), hitpoint(hit), damage(dam), defense(def)
{
}

bool Monster::Fight(Monster& other)
{
    if (PriorTo(other))
        if (Attacked(other) == 0)
            return true;

    while (true) {
        if (other.Attacked(*this) == 0)
            return false;
        if (Attacked(other) == 0)
            return true;
    }
}

int Monster::Attacked(Monster& other) const
{
    int harm = damage*2-other.defense;
    if (harm < 1)
        harm = 1;
    other.hitpoint -= harm;
    if (other.hitpoint < 0)
        other.hitpoint = 0;
    return other.hitpoint;
}

bool Monster::PriorTo(const Monster& other) const
{
    if (speed != other.speed)
        return speed>other.speed;

    if (hitpoint != other.hitpoint)
        return hitpoint > other.hitpoint;

    if (damage != other.damage)
```

```
        return damage > other.damage;

    if (defense != other.defense)
        return defense > other.defense;

    return true;
}

int main()
{
    Monster a(10,200,7,8);
    Monster b(10,150,8,7); //改成(10,180,8,7)则战斗失败

    if (a.Fight(b))
        cout<<"A Win!"<<endl;
    else
        cout<<"A    Lose!"<<endl;
    return 0;
}
```

若问题发生变化，出现了猫和狗，作为怪物的特例，它们的战斗方式不变，但攻击效果各自不同：
猫进攻导致对方的生命值减少量：
    (猫的攻击力值 * 2 - 对方的防御力值)   若上式小于 1，则取 1
狗进攻导致对方的生命值减少量：
    (狗的攻击力值 - 对方的防御力值 + 5) * 2   若上式小于 2，则取 2

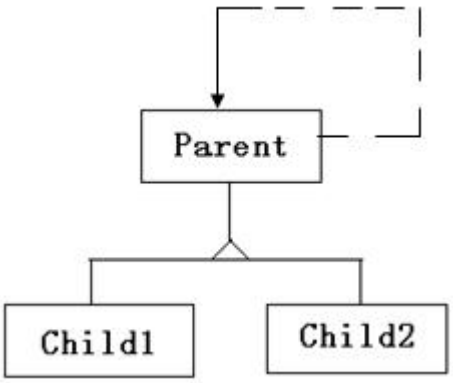
此时，怎么适应这种变化？子类型化。



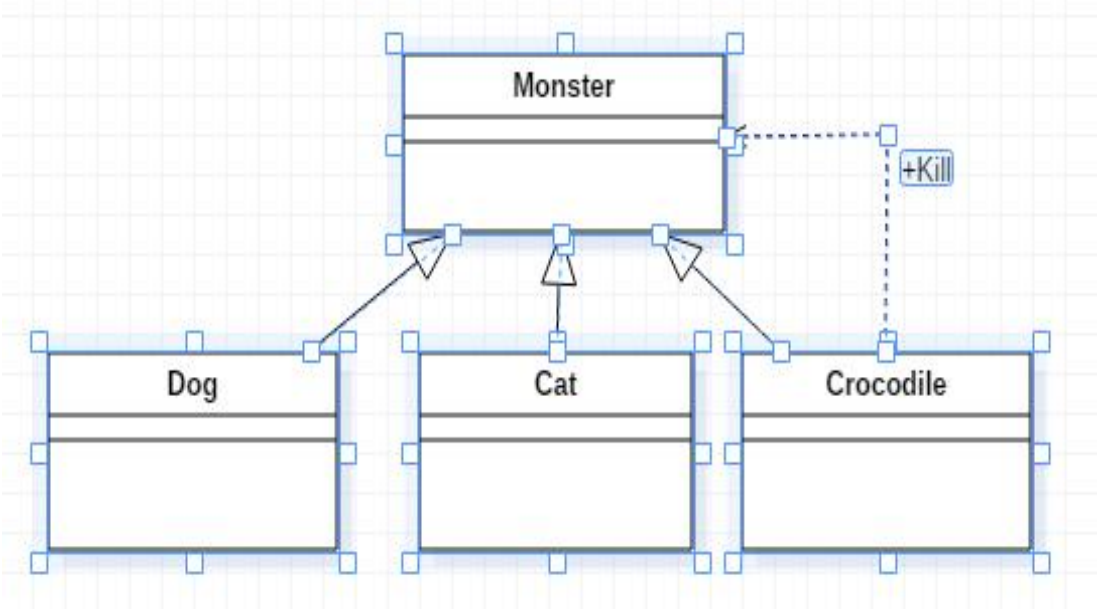

```
class Cat:public Monster {
private:
```

```
    virtual bool Attack(Monster& other) { //略  }
};

class Dog:public Monster {
private:
    virtual bool Attack(Monster& other) { //略  }
};
```

扩展 1：

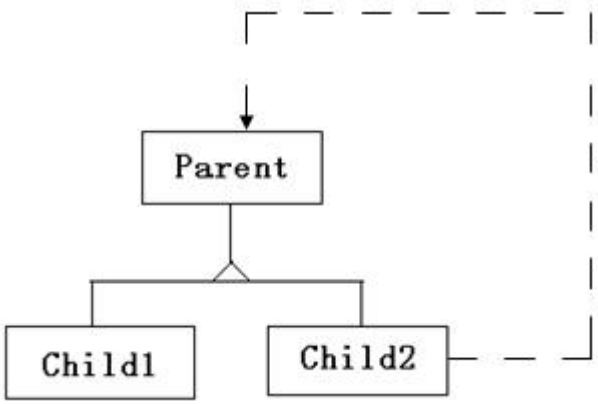


若新增一种怪物，如鳄鱼，它不但和其它动物战斗，还会吃掉其它动物。




```
class Crocodile:public Monster {
private:
    virtual bool Attack(Monster& other) {   //略  }
    kill(Monster & other) {   //略   }
};
// Monster、Dog、Cat 没有 kill 接口
// Crocodile 新增 Kill 接口
```

扩展 2：



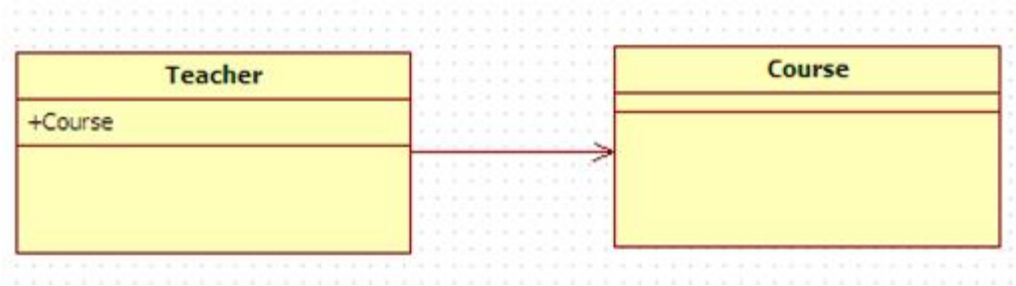
双向依赖例：使用面向对象的方法设计一个警察抓人的游戏，游戏中有小偷和行人，每个小偷有不同的经验分。警察每抓住一个小偷，得到小偷的经验分的一半再加 50 分奖励；但若警察抓住一个行人，则要扣掉 500 分。考虑到游戏的趣味性，将来可能还要增加其他类型的人物，其对应的奖励方法或惩罚方法也各不相同，如增加海盗，抓住他的奖励分可能是直接奖励 10000 分，如增加议员，抓住议员，可能的惩罚是所有已获得的分数减半等。

```
class Police
{
public:
      Police() {totalAward = 0;}
      void     Catch(Person * p);
      int      GetAward( ) const { return totalAward;}
private:
      int       totalAward;
};

class Person
{
public:
     virtual ~Person() {}
     virtual void BeCatched(Police * cop){}
};

class Thief: public Person {
     virtual void BeCatched(Police * cop){ //略  }
}

class Walker: public Person {
     virtual void BeCatched(Police * cop){ //略  }
}
```

关联(association)关系：
对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。关联关系体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。
例如：客户和订单(1:N)，公司和员工(1:N)，主人和汽车(1:N)，师傅和徒弟(1:N)，丈夫和妻子(1:1)，飞机和航班(1:N)，学生和课程(N:N)。

```
class B {...};
class A { B* b; .....};
A::afun1() { b->bfun1(); }
A::afun2() { b->bfun2(); }
```

关联关系可以分为单向关联，自身关联和双向关联。
UML 表示方法：实线箭头，类 A 指向类 B，表示单向关联。如果使用双箭头或不使用箭头表示双向关联。



单向关联是指只有某一方拥有另一方的引用，这样只有拥有对方者可以调用对方的公共属性和方法。 如下面代码：

```
// Husband.h
class CHusband
{
public:
	int nMoney;
	void GoShopping();
};

// Wife.h
#include "Husband.h"
class CWife
{
public:
	CHusband* pHuband;
};
```

上面代码中妻子拥有丈夫，可以使用对方的属性，比如钱，可以让对方做能做的事，比如去买东西。

单向关联例 1：游戏中的英雄有各自的魅力值、声望值、攻击力、防御力、法力等，每个英

雄可以最多带 5 个宝物，每种宝物有特有提升英雄某种能力的效果。游戏中假设共有 6 种宝物（暂时用 1,2,3,....6 代表，1 提升魅力 2 点，2 提升声望 3 点，3 提升攻击力 1 点，...），英雄这个类需要有功能：取得当前状态下的各种能力值，在指定位置中携带指定宝物，丢弃指定位置中的宝物等。

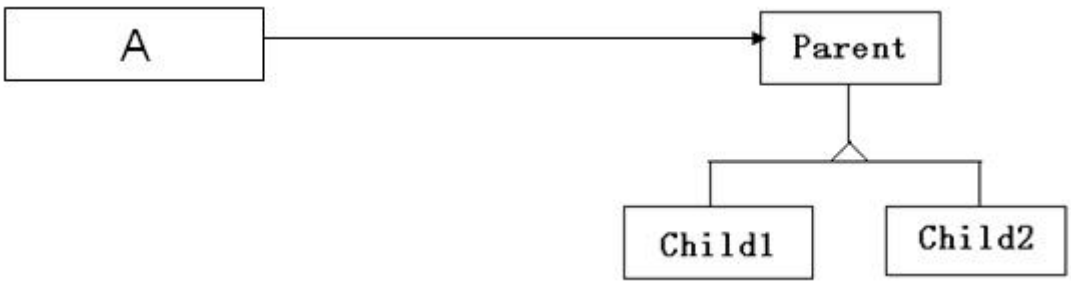
```
const int    ABLITTYCOUNT =5;        //5 种属性值
enum ABLITY {CHARM = 0,REPUTE,ATTACK,DEFENSE,POWER };
enum GOODS {NONE=0,G1,G2,G3,G4,G5,G6};   //6 种宝物
const int    BAGCOUNT = 5;        //5 个宝物袋

class Hero
{
public:
    Hero(int cha,int rep,int att,int def,int pow);
    void AddGood(int bagID,GOODS goods);
    void RemoveGood(int bagID);
    int    CurAblity(ABLITY which) const { return curAblities[which]; }
private:
    void RecaculateAblities();
private:
    //魅力值、声望值、攻击力、防御力、法力
    int rawAblities[ABLITTYCOUNT];
    int curAblities[ABLITTYCOUNT];
    //宝物袋
    GOODS* bags[BAGCOUNT];
};
```

单向关联例 2：**学生管理程序**中学生和宿舍。每个学生的信息除了包括姓名、学号等之外，还要有宿舍信息。宿舍信息包括几号楼，第几层，几号房间，以及住了哪几个学生等信息。

```
class Student
{
public:
    Student(Dorm * aDorm): mpDorm(aDorm) {    }
    ~Student() { }
    int     DormFloor( ) const    { return mpDorm->Floor( ); }
    // ….
private:
    Dorm * mpDorm;
};
```

若例子 1 中的宝物是具体的，如头盔、盾牌、铠甲、戒指、护身符...
若例子 2 中的宿舍存在不同类型，如四人间、双人间、单人间...

扩展：




```
class Parent;
class A {
public:
        virtual ~A( );
        void     Func( );
protected:
         Parent *    p[MAXCOUNT];
};
```

再进一步:
若例子 1 中不仅仅有英雄，还有魔法师、骑士、精灵、黑武士...
若例子 2 中学生还分为中学生、专科生、本科生、研究生...

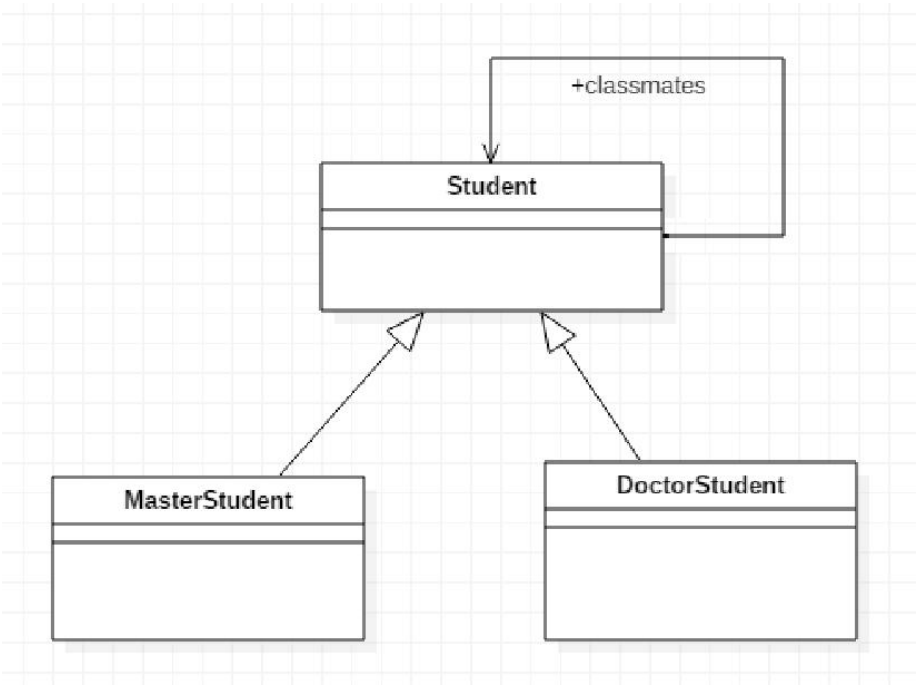
自身关联是指拥有一个自身的引用。
例如下面代码：

```
class CSingle
{
public:
     CSingle *pSingle;
};
```

再例如链表中的节点类：

```
template<class T>
class Node
{
public:
    T data;                                    //存放数据
    Node(const T& item, Node<T>* next = 0);
    void insertAfter(Node<T>* p);              //往指定节点后插入一个新的节点
    Node<T>* deleteAfter();                    //删除当前节点后的节点
    Node<T>* nextNode();
    const Node<T>* nextNode() const;           //返回下一个节点的指针和地址
private:
    Node<T>* next;
};
```

再例如学生类中的同学关系：



双向关联是指双方都拥有对方的引用，都可以调用对方的公共属性和方法。
例如下面代码：

```
class CWife;
class CHusband
{
public:
    CWife* pWife;
};

class CWife
{
public:
    CHusband* pHuband;
};
```

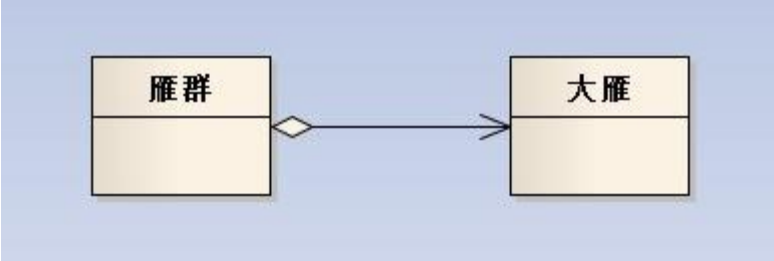
上面代码中丈夫和妻子是比较公平的关系，都可以使用对方公共的属性和方法。

聚合/聚集(aggregation)关系：
当对象 B 被加入到对象 A 中，成为对象 A 的组成部分时，对象 A 和对象 B 之间为聚合关系。聚合是关联关系的一种特例。聚合指的是整体与部分之间的关系，体现的是整体与部分、拥有的关系，即 has-a 的关系，此时整体与部分之间是可分离的，可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；
例如：自行车和车把、响铃、轮胎，汽车和引擎、轮胎、刹车装置，计算机和主板、CPU、内存、硬盘，航母编队和航母、驱护舰艇、舰载机、潜艇，课题组和科研人员。
在代码层面，聚合和关联关系是一致的，只能从语义级别来区分；
关联关系中两个类是处于相同的层次，而聚合关系中两个类是处于不平等的层次，一个表示整体，一个表示部分。

```
class B {...}
class A { B* b; .....}
```

```
class CCar
{
public:
	CTyre *cTyre[4];
};

class CTyre
{
	// Do something
};
```

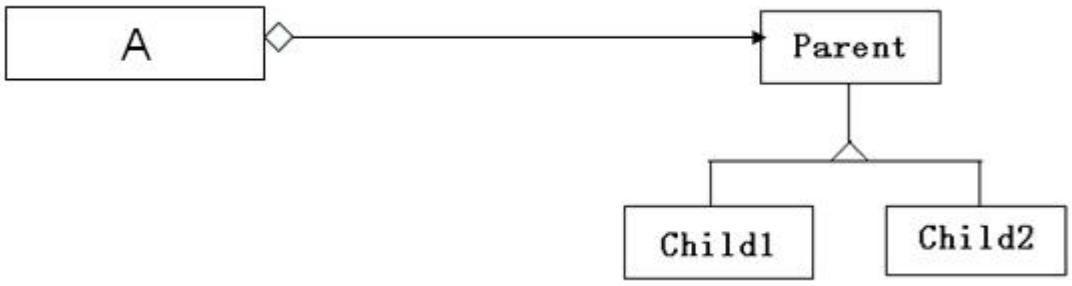
UML 表示方法：尾部为空心菱形的实线箭头（也可以没箭头），类 A 指向类 B。



聚合例：宿舍管理程序中宿舍和学生。宿舍有多个学生，宿舍建立时，可以指定，也可不指定对应的多个学生，并且以后可以随时增删学生。

```
class Dorm {
public:
	Dorm( Student * s[ ],int count):maxCount(count) {
		mStudents = new Student*[maxCount];
		for(int i=0;i<count;i++)
			mStudents[i] = s[i];
	}
	~Dorm( ) {    delete[] mStudents; }

	void AddStudent(Student * s, int index){
		if ( mStudents[index] == NULL))
			mStudents[index] = s;
    }

	void RemoveStudent(int index) {
		mStudents[index] = NULL;
	    }
private:
	int    maxCount;
	Student * * mStudents;
};
```

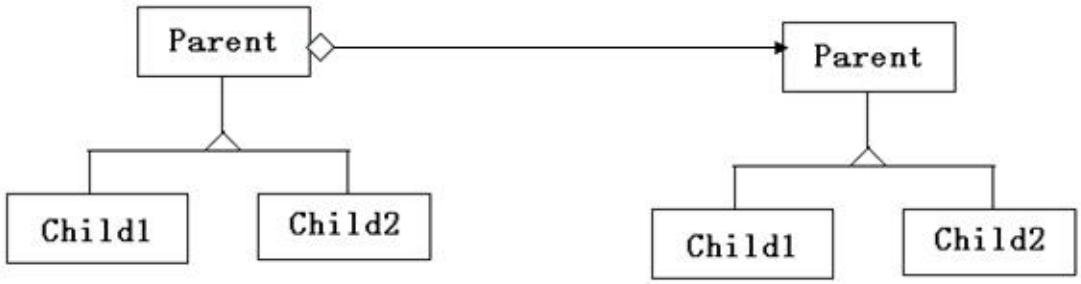
扩展 1：




```
class Parent;
class A {
public:
        virtual ~A( );
        void    Func( );
protected:
        Parent *    p[5];
};
```

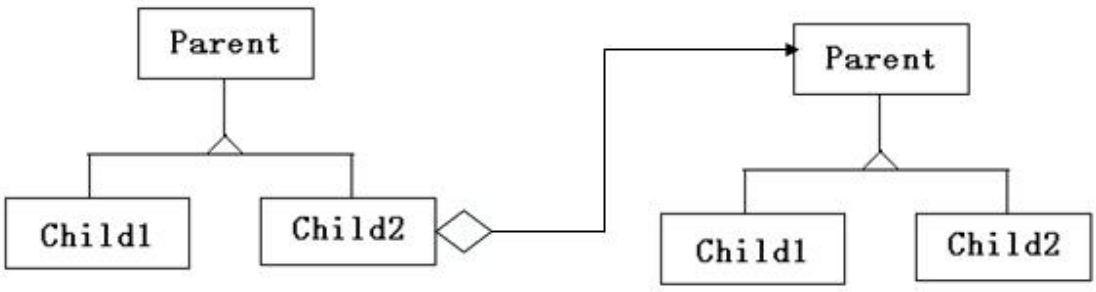
例：果篮与水果

扩展 2：



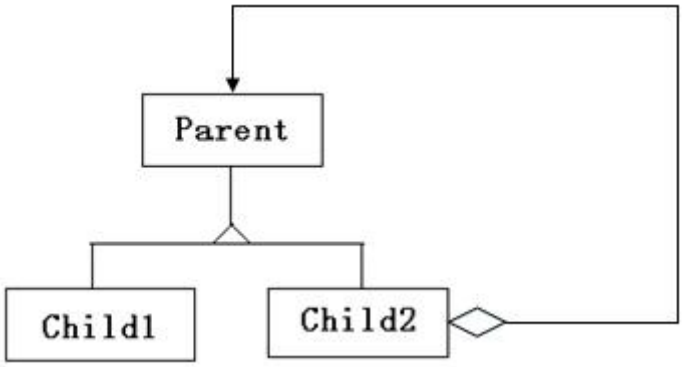
例：防盗门和锁

扩展 3：

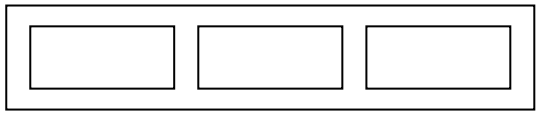


例：防盗门和报警器

扩展 4：



例：在制作绘图程序时，需要绘制各种图形元素，包括矩形、椭圆等。现需要增加一种图形元素—Grid，每个 Grid 本身就是一个矩形，但其文本内容一定为空，同时一个 Grid 还包含多个矩形，矩形个数由创建 Grid 时的参数指定，示例如图。请定义并实现类 Grid。

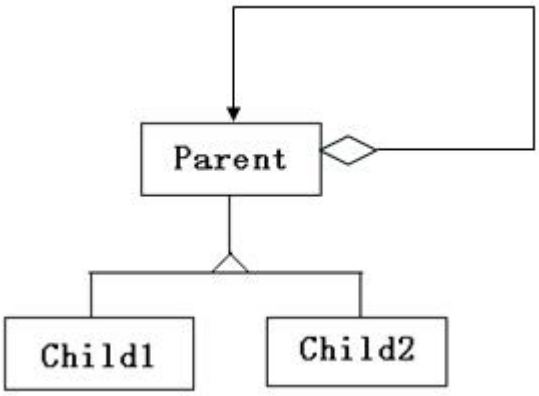


Grid 示例

```
class    Shape {
public: virtual ~Shape( ){ }
   virtual void Draw( )=0;
};
Class Ellipse:public Shape{
public:      virtual ~ Ellipse ( ) { }
              virtual void Draw ( ) { /*略  */ }
};
class Rect:public Shape {
public:
   Rect ( const char * text) { /*略  */}
   virtual ~ Rect ( ) { }
   virtual void Draw ( )    { /*略  */ }
};
class Grid:public Rect {
public:
     Grid (int n):Rect(NULL),count(n)
     {
          prects=new Shape*[count];
          for (int i=0; i<count; i++)
          {
               prects[i]=new Rect(NULL);
          }
     }
     void Draw ( )
     {
          for (int i=0; i<count; i++)
               prects[i]->Draw();
```

```
    }
    ~Grid()
    {
        for (int i=0; i<count; i++)
        {
            delete prects[i];
        }
        delete[] prects;
    }
private:
    int count;
    Shape ** prects;
};
```

扩展 5：



例：套娃，按主题，情侣类、风光类、美少女类；按用途，迎宾类、收纳类等

组合/合成(composition)关系：

组合也是关联关系的一种特例，体现的是一种 contains-a 的关系，这种关系比聚合更强，也称为强聚合；同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束。**整体类负责部分类对象的生存与消亡。**

例如：公司和部门，人和大脑、四肢，窗口和标题栏、菜单栏、状态栏。

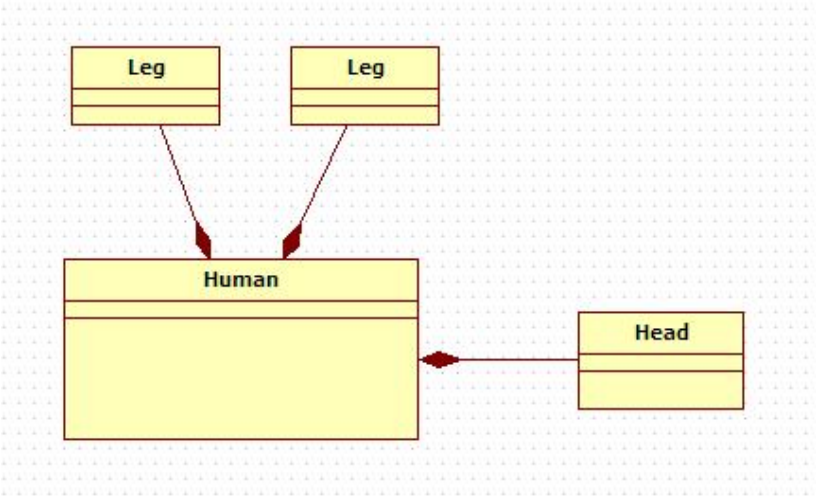
在代码层面，组合和关联关系是一致的，只能从语义级别来区分。

组合跟聚合几乎相同，唯一的区别就是“部分”不能脱离“整体”单独存在，就是说， “部分”的生命期不能比“整体”还要长。

```
class B{...}
class A{ B b; ...}
或
class A{
public: A():pb(new B){}
        ~A(){ delete pb; }
private: B *pb; ...}

// Company.h
#include "Department.h"
```

```
class CCompany
{
public:
	CDepartment cDepartment[N];
};

// Department.h
class CDepartment
{
	// Do something
};
```

UML 表示方法：尾部为实心菱形的实线箭头（也可以没箭头），类 A 指向类 B



泛化(generalization)关系

泛化是一种一般与特殊、一般与具体之间关系的描述，具体描述建立在一般描述的基础之上，并对其进行了扩展。

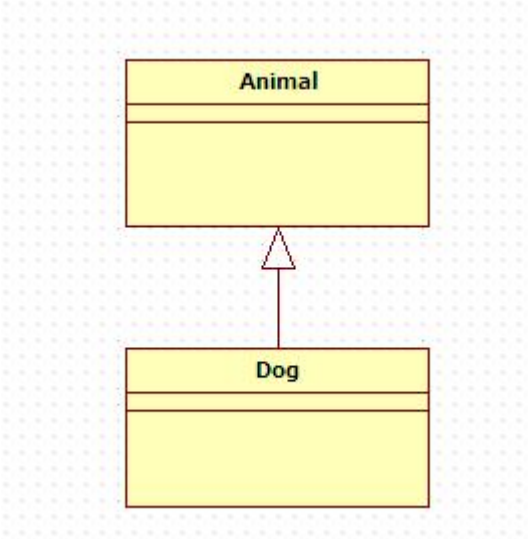
比如狗是对动物的具体描述，一般把狗设计为动物的子类。

在代码层面，泛化是通过继承实现的。

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

// Animal.h
class CAnimal
{
public:
	// implement
	virtual void EatSomething()
	{
		// Do something
	}
};
```

```
// Tiger.h
#include "Animal.h"
class CTiger : public CAnimal
{
	// Do something
};
```

UML 表示方法：空心三角形箭头的实线，子类指向父类



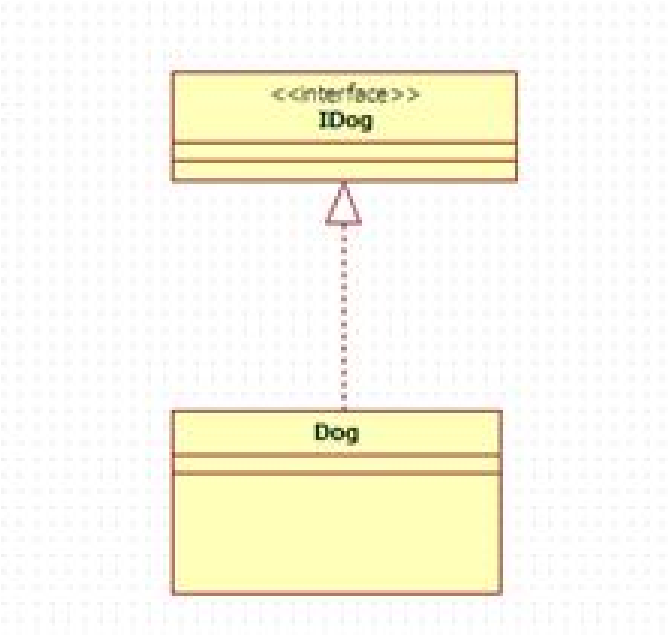
实现(realization)关系

实现是一种类与接口的关系，表示类是接口所有特征和行为的实现。从广义上来说，类模板和模板类也是一种实现关系。

在代码层面，实现一般通过类实现接口来描述的。

```
// Animal.h
class IAnimal
{
public:
	// interface
	virtual void EatSomething() = 0;
};

class Animal : public IAnimal
{
	// Do something
};
```

UML 表示方法：空心三角形箭头的虚线，实现类指向接口。



泛化和实现的区别就在于子类是否继承了父类的实现，如有继承则关系为泛化，反之为实现。

几种关系所表现的强弱程度依次为：泛化/实现>组合>聚合>关联>依赖。

如何适应变化实现复用？
基于关系模型通过扩展既有代码来适应变化实现复用。

假设存在下面的 Some 类：

```
class Some {
public:
      void OperA( );
      void OperB( );
      //…
private:
      int  myData;
};
```

Some 类的使用者可能是另一个类或函数：

```
Some * p = new Some;
p->OperA( );
delete p;
```

Some 类可能存在两类变化：
1. 职责的变化（接口、功能的变化）
2. 实现的变化（数据表示、行为的变化）

针对第 1 类变化，再展开讨论：
a. 增加新功能（如增加 OperC）
① 使用继承，派生出子类 SomeChild，在其中增加 OperC()；
② 使用组合，构造一个 SomeNew 类，新增 OperC()，OperA，OperB 的功能委托给 Some。

b. 需要删掉 OperA( )
① 使用继承
② 使用组合

c. 需要将 OperA 改名为 MyOperA
① 使用继承
② 使用组合

d. 需要给 OperB()增加新的形式参数
① 使用继承
② 使用组合

e. 需要改变 OperB()的返回类型
① 使用继承
② 使用组合

注意：职责的变化，一般需要同步修改 Some 类的使用者。

针对第 2 类变化，再展开讨论：
a. 修改数据表示
① 使用继承，派生出子类 SomeChild，在其中给出新的数据表示；
② 使用组合，构造 SomeNew 类，在其中给出新的数据表示，OperA，OperB 的功能委托给 Some。

b. 修改行为的实现
① 使用继承，派生出子类 SomeChild，在其中给出 OperA 的新实现（应使用虚机制）；
② 使用组合，构造 SomeNew 类，在其中给出 OperA 的新实现，OperB 的功能委托给 Some。

进一步，若同时存在多种变化，例如：OperA 有多种实现变化、OperB 有多种实现变化、还需要增加新功能 OperC，解决方法：
1. 综合使用组合和继承，参考上面关系模型的各种扩展型
2. 组合优先