**模式：**

指在一定环境中解决某一问题的方案，包括三个基本元素问题、环境、解决方案

**设计模式（Design pattern）：**

是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己于他人于系统都是多赢的;设计模式使代码编制真正工程化。

设计模式共有23种可以分为以下**三大类型**：

**创建型模式 :** 通常和对象的创建有关，涉及到对象实例化的方式。(共 5 种模式：工厂方法模式 、抽象工厂模式 、单例模式 、建造者模式 、原型模式)

**结构型模式:** 描述的是如何组合类和对象以获得更大的结构。(共 7 种模式：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式)

**行为型模式:** 用来对类或对象怎样交互和怎样分配职责进行描述。(共 11 种模式：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式)

**列举四种模式的例子：**

1. **建造者模式**

**1.1角色和职责：**

**Builder：**为创建产品各个部分，统一抽象接口。

**ConcreteBuilder：**具体的创建产品的各个部分，部分 A， 部分 B，部分 C。

**Director：**构造一个使用 Builder 接口的对象。

**Product：**表示被构造的复杂对象。

**ConcreteBuilder：**创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类， 包括将这些部件装配成最终产品的接口

**1.2案例：**

工程队的基本操作已经被抽象成方法，所以工厂师在操作的时候，只需要调用对应接口的方法，从而达到给工程师某个类型的实例的时候，工程师可以按照对应类型的要求去实例化

**关键字:**建公寓工程队 FlatBuild 别墅工程队 VillaBuild 设计者Director

1. class Builder {
2. public:
3. virtual void makeFloor() = 0; virtual void makeWall() = 0;
4. virtual void makeDoor() = 0; virtual House \*GetHouse() = 0;
5. };
6. *//公寓*
7. class FlatBuild : public Builder {
8. *//别墅*
9. class VillaBuild : public Builder {
10. class Director {
11. public:
12. void Construct(Builder \*builder) {
13. builder->makeFloor();
14. builder->makeWall();
15. builder->makeDoor();
16. }
17. *//最终实现效果*
18. *//客户直接造房子*
19. House \*pHose = new House;
20. pHose->setDoor("wbm 门");
21. pHose->setFloor("wbmFloor");
22. pHose->setWall("wbmWall");
23. delete pHose;
24. *//工程队直接造房子*
25. Builder \*builder = new FlatBuild;
26. builder->makeFloor();
27. builder->makeWall();
28. builder->makeDoor();
29. *//指挥者(设计师)指挥 工程队 和 建房子*
30. Director \*director = new Director;
31. *//建公寓*
32. Builder \*builder = new FlatBuild;
33. director->Construct(builder); *//设计师 指挥 工程队干活*
34. House \*house = builder->GetHouse();
35. cout << house->getFloor() << endl;
36. delete house;
37. delete builder;
38. builder = new VillaBuild;
39. director->Construct(builder); *//设计师 指挥 工程队干活*
40. house = builder->GetHouse();
41. cout << house->getFloor() << endl;
42. delete house;
43. delete builder;
44. delete director;
45. system("pause");

**特点：**

是一种对象创建型模式之一，用来隐藏复合对象的创建过程，它把复合对象的 创建过程加以抽象，通过子类继承和重载的方式，动态地创建具有复合属性的对象。

builder 模式是为对象的创建而设计的模式，创建的是一个复合对象，被创建的对象为一个具有简单复合属性的复合对象，关注对象创建的各部分，不同工厂对产品属性有不同的创建方法。

1. **代理模式**

**2.1角色和职责**

**subject**(抽象主题角色)：真实主题与代理主题的共同接口。

**RealSubject**(真实主题角色)：定义了代理角色所代表的真实对象。

**Proxy**(代理主题角色)：含有对真实主题角色的引用，代理角色通常在将客户端调用传递给真是主题对象之前或者之后执行某些操作，而不是单纯返回真实的对象。

**2.2案例：**

1. using namespace std;
2. class Interface{
3. public:
4. virtual void Request()=0;
5. };
6. class RealClass:public Interface{
7. public:
8. virtual void Request() {
9. cout<<"真是的请求"<<endl;
10. }
11. };
12. class ProxyClass:public Interface{
13. PointerPropertyBuilderByName(RealClass,realClass,private)
14. *//在该代理类中代理其他对象*
15. virtual void Request() {
16. m\_realClass=new RealClass();
17. cout<<"before call"<<endl;
18. m\_realClass->Request();
19. cout<<"after call"<<endl;
20. delete m\_realClass;
21. }
22. };

**特点：**

Proxy 模式又叫做代理模式，是构造型的设计模式之一，它可以为其他对象提供一 种代理(Proxy)以控制对这个对象的访问。

所谓代理，是指具有与代理元(被代理的对象)具有相同的接口的类，客户端必须 通过代理与被代理的目标类交互，而代理一般在交互的过程中(交互前后)，进行某些 特别的处理。

适合于为其他对象提供一种代理以控制对这个对象的访问。

1. **组合模式**

**3.1角色和职责**

**Component** (树形结构的节点抽象)

为所有的对象定义统一的接口(公共属性，行为等的定义) - 提供管理子节点对象的接口方法

[可选]以及提供管理父节点对象的接口方法

**Leaf** (树形结构的叶节点) Component 的实现子类

**Composite**(树形结构的枝节点) Component 的实现子类

**3.2案例：**

1. class IFile{
2. public:
3. virtual ~IFile()=0;
4. virtual void display()=0;
5. virtual int add(IFile\* file)=0;
6. virtual int remove(IFile\* file)=0;
7. virtual list<IFile\*>\* getChild()=0;
8. };
9. class File:public IFile{
10. PointerPropertyBuilderByName(list<IFile\*>,list,private)
11. PropertyBuilderByName(string,name,private)
12. public:
13. File(string name){
14. *//文件是不允许有list的*
15. this->m\_list=NULL;
16. this->m\_name=name;
17. }
18. ~File(){
19. if(this->m\_list!=NULL){
20. delete m\_list;
21. }
22. }
23. virtual void display() {
24. cout<<this->m\_name<<endl;
25. }
26. virtual int add(IFile \*file){
27. return -1;
28. }
29. int remove(IFile \*file){
30. return -1;
31. }
32. list<IFile \*> \*getChild(){
33. return NULL;
34. }
35. };
36. class Directory:public IFile{
37. PointerPropertyBuilderByName(list<IFile\*>,list,private)
38. PropertyBuilderByName(string,name,private)
39. public:
40. Directory(string name){
41. this->m\_name=name;
42. this->m\_list=new list<IFile\*>;
43. }
44. ~Directory(){
45. if(m\_list!=NULL){
46. delete this->m\_list;
47. this->m\_list=NULL;
48. }
49. }
50. void display(){
51. cout<<this->m\_name;
52. }
53. int add(IFile \*file){
54. this->m\_list->push\_back(file);
55. return 0;
56. }
57. int remove(IFile \*file){
58. this->m\_list->remove(file);
59. return 0;
60. }
61. list<IFile \*> \*getChild(){
62. return this->m\_list;
63. }
64. };

**特点：**

Composite 模式也叫组合模式，是构造型的设计模式之一。通过递归手段来构造树形的 对象结构，并可以通过一个对象来访问整个对象树。

1. **桥接模式**

**4.1角色和职责**

**Client：**Bridge 模式的使用者

**Abstraction：**

抽象类接口(接口或抽象类)维护对行为实现(Implementor)的引用

**Refined Abstraction：**Abstraction 子类

**Implementor：**行为实现类接口 (Abstraction 接口定义了基于Implementor 接口的更高层次的操作: 如：ConcreteImplementor Implementor 子类)

**4.2案例：**

1. *//引擎*
2. class Engine{
3. public:
4. virtual ~Engine();
5. virtual void install()=0;
6. };
7. Engine::~Engine(){
8. }
9. *//引擎*
10. *//4000引擎*
11. class Engine4000:public Engine{
12. public:
13. void install(){
14. cout<<"发动机4000安装"<<endl;
15. }
16. };
17. *//4000引擎*
18. *//3500 引擎*
19. class Engine3500:public Engine{
20. public:
21. void install(){
22. cout<<"发动机4000安装"<<endl;
23. }
24. };
25. *//3500引擎*
26. *//抽象汽车，维护抽象引擎*
27. class Car{
28. PointerPropertyBuilderByName(Engine,engine,protected)
29. public:
30. Car(Engine\* engine);
31. virtual ~Car();
32. virtual void installEngine()=0;
33. };
34. Car::Car(Engine \*engine){
35. this->m\_engine=engine;
36. }
37. Car::~Car(){
38. }
39. *//抽象汽车*
40. *//实际汽车，维护抽象引擎，实际引擎由外部引入*
41. class BMW7:public Car{
42. public:
43. BMW7(Engine \*e);
44. public:
45. void installEngine();
46. };
47. BMW7::BMW7(Engine \*e):Car(e){
48. }
49. void BMW7::installEngine(){
50. cout<<"Bmw7:";
51. m\_engine->install();
52. }

**特点：**

Bridge 模式又叫做桥接模式，是构造型的设计模式之一。Bridge 模式基于类的最小设计 原则，通过使用封装，聚合以及继承等行为来让不同的类承担不同的责任。它的主要特点是 把抽象(abstraction)与行为实现(implementation)分离开来，从而可以保持各部分的独 立性以及应对它们的功能扩展。