1. 观察者(Observer)模式
2. 装饰者(Decorator)模式：

+ 保证接口不变的情况下加强肋的功能， 它保证的是被修饰的对象功能比原来对象丰富，但不做准入条件判断和准入参数过滤，如是否可以执行类的功能，过滤输入参数是否合法等，这不是装饰模式关心的。

+通常在构造函数里用参数的方式传进来装饰者对象。

+星巴兹 例子， 调制一杯咖啡，根据添加的材料累加价格，算出最终价格。

class CIRunner

{

public:

CIRunner() {};

~CIRunner() {};

virtual void mvRun() = 0;

};

class CRunnerWithJet : public CIRunner

{

public:

CRunnerWithJet(CIRunner\* opRunner) { mopRunner = opRunner; };

~CRunnerWithJet() {};

void mvRun()

{

cout << "加快运动员的速度： 为运动员增加喷气装置" << endl;

mopRunner->mvRun();

}

private:

CIRunner\* mopRunner;

};

int main()

{

//定义运动员

CIRunner\* op\_runner = new CRunner();

//对其功能加强

op\_runner = new CRunnerWithJet(op\_runner);

//看看它的跑步情况如何

cout << "===增强后的运动员的功能===" << endl;

op\_runner->mvRun();

return 0;

}

1. 简单工厂

+ 一个工厂用case 语句 创建某个商品

1. 工厂方法模式

+工厂会独立出来， 每个工厂单独生产一种商品

SFJFactory factory1;

soapBase\* pSoap1 = factory1.creatSoap();

pSoap1->show();

XSLFactory factory2;

soapBase\* pSoap2 = factory2.creatSoap();

pSoap2->show();

NASFactory factory3;

soapBase\* pSoap3 = factory3.creatSoap();

1. 抽象工厂(Abstract Factory)模式

+ 工厂会独立出来， 每个工厂单独生产多种商品

FactoryA factory1;

FactoryB factory2;

SoapBase\* pSoap1 = NULL;

ToothBase\* pToothpaste1 = NULL;

pSoap1 = factory1.creatSoap();

pToothpaste1 = factory1.creatToothpaste();

pSoap1->show();

pToothpaste1->show();

SoapBase\* pSoap2 = NULL;

ToothBase\* pToothpaste2 = NULL;

pSoap2 = factory2.creatSoap();

pToothpaste2 = factory2.creatToothpaste();

pSoap2->show();

pToothpaste2->show();

1. 单件(Singleton)模式
2. 命令(Command)模式

+请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

+将请求封装成对象，以便使用不同的请求，队列或者日志来参数化其他对象。

+应用：线程池里的工作队列

Receiver\* pReceiver = new Receiver();

ConcreteCommand\* pCommand = new ConcreteCommand(pReceiver);

Invoker\* pInvoker = new Invoker(pCommand);

pInvoker->call();

1. 适配器(Adapter)模式

+8.1 对象适配器：（松耦合，推荐）

class Adapter : public Target

{

public:

private:

Adaptee\* m\_Adaptee;

};

+8.2 类适配器

class Adapter : public Target, Adaptee

{

public:

void Request() // Implements ITarget interface

{

// Possibly do some data manipulation

// and then call SpecificRequest

this->Request();

this->SpecificRequest();

}

};

1. 外观(Facade)模式

+提供统一接口，用来访问子系统的一群接口。让子系统更容易使用。类似：Player类

+最少知识原则(Least knowledge): 减少对象之间的交互，只留下几个密友。

1. 模板方法(Template)模式

+好莱坞原则：不要调用我，让我来调用你。 低层类结构不要调用高层类结构

1. 迭代器(Iterator)模式

+提供一种方法顺序访问一个容器中的各个元素，而又不暴露其内部的实现。

+单一责任：尽量让每个类保持单一的责任。

1. 组合(Composite)模式

+树形结构来表现“整体/部分”层次结构。

1. 状态(State)模式

+允许对象在内部改变时改变它的行为，对象看起来好像修改了它的类。

+ 状态机

1. 策略(Strategy)模式

+定义一系列算法的方法，所有这些算法完成的都是相同的工作，只是实现不同。

1. 代理(Proxy)模式

+控制对象的访问。

+一般隐藏对象信息， 直接在构造函数创建。

class CIRunner

{

public:

CIRunner() {};

~CIRunner() {};

virtual void mvRun() = 0;

};

class CRunner :public CIRunner

{

public:

CRunner() {};

~CRunner() {};

void mvRun() { cout << "运动员跑步： 动作很潇洒" << endl; };

};

class CRunnerAgent : public CIRunner

{

public:

CRunnerAgent() { mopRunner = new CRunner (); }

~CRunnerAgent() {}

void mvRun() { mopRunner->mvRun(); }

private:

CIRunner\* mopRunner;

};

int main()

{

//定义一个短跑运动员

CIRunner\* op\_runner = new CRunner();

//定义的代理人

CIRunner\* op\_agent = new CRunnerAgent(op\_runner);

//要求运动员跑步

cout << "====客人找到运动员的代理要求其去跑步===" << endl;

op\_agent->mvRun();

return 0;

}

1. 复合(Compound)模式

+MVC

+控制器是视图的策略，视图使用不同的控制器实现，得到不同的行为

+模型使用观察者模式

+视图使用组合模式实现用户界面

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

//不经常使用的模式

//////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////

1. 桥接(Bridge)模式

+ 基类之间调用逻辑

1. 生成器(Builder)模式

+ 封装过长的初始化过程, 并且初始化函数是有先后顺序的

1. 责任链(Responsibility)模式

+ 一层一层询问 handler

1. 享元(Flyweight)模式

+ map管理 对象

1. 解释器(Interpreter)模式

+ 字符串解析

1. 中介者(Mediator)模式

+ A和B不能直接调用，必须通过中介者调用。中介者包含A，B的指针。

1. 备忘录(Memento)模式

+ 类似数据库

1. 原型(Prototype)模式

+ 赋值构造函数

1. 访问者(Visitor)模式

+ Visitor类， Element类 来管理访问