【设计模式部分】

● 什么是设计模式

模式就是得到很好研究的范例。设计模式是软件开发过程中经验的积累，**特定问题**的经过实践检验的**特定解决方法。**

● 设计模式的原则（SOLID）

1 单一职责原则（SRP）：每一个引起类变化的原因就是一个职责，当类具有多职责时，应把多余职责分离出去，分别创建一些类来完成每一个职责。

2 “开-闭”原则（OCP）：软件对扩展是开放的，对修改是关闭的。开发一个软件时，应可以对其进行功能扩展(开放)，在进行扩展的时候，不需要对原来的程序进行修改(关闭)。

3 里氏代换原则（LSP）：把系统的所有可能的行为抽象成一个底层；由于可从抽象层导出一个或多个具体类来改变系统行为，因此对于可变部分，系统设计对扩展是开放的。

4 接口隔离原则（ISP）：“继承必须确保超类所拥有的性质在子类中仍然成立”，当一个子类的实例能够替换任何其超类的实例时，它们之间才具有 is-A 关系。

5 依赖倒置原则（DIP）：如果一个类有几个使用者，与其让这个类载入所有使用者需要使用的所有方法，还不如为每个使用者创建一个特定接口，并让该类分别实现这些接口。

● 模式的基本要素：模式名称、问题、解决方案、效果。

● 如何描述设计模式

模式名和分类、意图、动机、适用性、结构、参与者、协作、效果、实现、代码示例、相关模式。

● 创建型设计模式 5 个

用来创建对象的模式，抽象了实例化过程。**工厂方法**、抽象工厂、**单例**、生成器、原型模式。

● 结构型设计模式 7 个

结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（类结构型模式），或者通过组合一些对象来实现新的功能（对象结构型模式）。组合、**装饰者**、代理、享元、(外观)、桥梁、**适配器**。

● 行为型设计模式 11 个

着力解决的是类实体之间的通讯关系，希望以面向对象的方式描述一个控制流程。模版、**观察者**、迭代子、责任链、备忘录、命令、状态、访问者、(解释器、中介者)、策略模式。

● MVC

类的模型/视图/控制器（Model/View/Controller）三元组（MVC）。Model 是应用对象，View 是它在屏幕上的显示，Controller 定义用户界面对用户输入的响应模式。使用了模式：观察者、组合、策略、工厂方法、装饰。

● 框架（Framework）与结构（Architecture）

1 框架是可实例化的、部分完成的软件系统或子系统，它为一组系统或子系统定义了统一的体系结构，并提供了系统的基本构造块，还为实现具体功能定义了扩展点。2 框架实现了体系结构级别的复用。

● 框架（Framework）与模式（Pattern）

1 从应用领域上分，框架给出的是整个应用的体系结构；而设计模式则给出了单一设计问题的解决方案。2 从内容上分，设计模式仅是一个单纯的设计，这个设计可被不同语言以不同方式来实现；而框架则是设计和代码的混合体。3 设计模式比框架更容易移植：框架一旦设计成形，以其为基础进行应用的开发就要受制于框架的实现环境；而设计模式是与语言无关的，所以可以在更广泛的异构环境中进行应用。4 总之，框架是软件，而设计模式是软件的知识体，设计模式的合理利用可以提升框架的设计水平。

【代码实现】

● 单例模式的代码实现：

不会生成子类：

```
class MazeFactory {
public:
        static MazeFactory* Instance(){
                if (_instance == 0){
                        _instance = new MazeFactory;}
                return _instance;}
        //interfaces...
protected:
        MazeFactory();
private:
        static MazeFactory* _instance = 0;
};
```

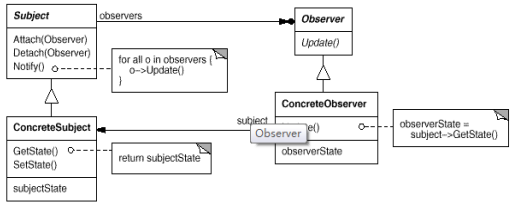
会生成子类：

```
class MazeFactory {
public:
        static MazeFactory* Instance(){
                if (_instance == 0){
                        const char* mazeStyle = getenv("MAZESTYLE");
                        if (strcmp(mazeStyle, "bombed")==0){
                                _instance = new BombedMazeFactory;}
                        else if (...){...}
                        else{_instance = new MazeFactory;}
                }
                return _instance;
        }
        //interfaces...
protected:
        MazeFactory();
private:
        static MazeFactory* _instance = 0;
};
```

● 装饰者模式的代码实现：

```
class VisualComponent {
public:
        VisualComponent();
        virtual void Draw();
        virtual void Resize();
        ...
};
class Decorator : public VisualComponent {
public:
        Decorator(VisualComponent*);
        virtual void Draw(){
                _component->Draw();
        }
        virtual void Resize(){
                _component->Resize();
        }
        ...
private:
        VisualComponent* _component;
}
```

● 观察者模式的两种实现：推模型、拉模型



推模型：

```
class Subject {
        virtual ~Subject();
        virtual void Attach(Observer* o){
                _observers->Append(o);      }
        virtual void Detach(Observer* o){
                _observers->Remove(o);      }
        virtual void Notify(){
                ListIterator<Observer*> i(_observers);
                for (i.first(); !i.IsDone(); i.Next()){
        i.CurrentItem()->Update(this);}
        }
protected:
        Subject();
private:
        List<Observer*> *_observers;
};
```

拉模型：

```
class Subject;
class Observer {
public:
        virtual ~Observer();
        virtual void Update(Subject* theChanged) = 0;};
```

特点：推模型是假定主题对象知道观察者需要的数据；而拉模型是主题对象不知道观察者具体需要什么数据，没有办法的情况下，干脆把自身传递给观察者，让观察者自己去按需要取值。

优势：推模型可能会使得观察者对象难以复用，因为观察者的 update()方法是按需要定义的参数，可能无法兼顾没有考虑到的使用情况。

区别：推是目标向观察者发送关于改变的详细信息，不管需要与否；拉是目标除最小通知外什么也不送出，而在此之后观察者显式地向目标询问细节。

● 工厂方法模式代码实现：采用 new 的 if-else 设计

```
Pizza orderPizza(String type){
        Pizza pizza;
        if (type.equals("cheese")){
                pizza = new CheesePizza();
        } else if (type.equals("greek")) {
                pizza = new GreekPizza();}
        pizza.bake();
        pizza.cut();
        return pizza;
}
```

各种产品类继承产品类，各种创建类继承创建类。

客户端：

```
public class PizzaTestDrive {
        public static void main(String[] args){
                PizzaStore nyStore = new NYPizzaStore();
                PizzaStore chicagoStore = new ChicagoPizzaStor();
                Pizza pizza = nyStore.orderPizza("cheese");
                System.out.println("He ordered a " +
pizza.getname());
                pizza = chicagoStore.orderPizza("cheese");
                System.out.println("She ordered a " +
pizza.getname());    }
}
```

抽象工厂模式的实现即把每一种 make 部件均作为一个虚函数，返回部件对象。

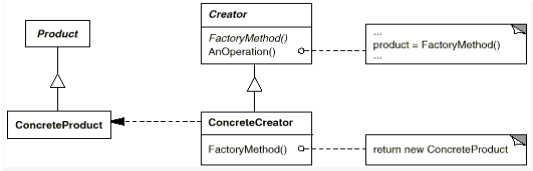
【意图、适用性、效果和类图】

【创建型】1 **工厂方法模式**：定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

适用：1 当一个类不知道它所必须创建的对象的类的时候。2 当一个类希望由它的子类来指定它所创建的对象的时候。3 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

协作：Creator 依赖于它的子类来定义工厂方法，所以它返回一个适当的 Concrete Product 实例。

效果：1 为子类提供挂钩（hook）；2 连接平行的各层次。



2 **单例模式**：保证一个类只有一个实例，并提供一个访问它的全局访问点。（提供一个允许用户看见这个实例的唯一接口；**可能是最有名/最流行的设计模式**）

适用：1 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时；2 当这个唯一实例应是通过子类化可扩展的，而且客户应该无需更改代码就能使用一个扩展的实例时。

协作：客户只能通过单例的 Instance 操作访问一个单例的实例。

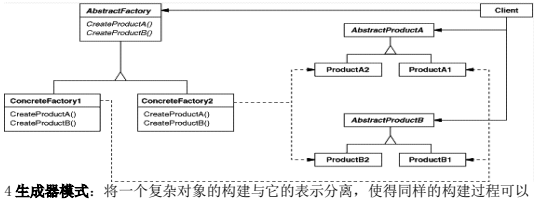
效果：优点：1 对唯一实例的受控访问；2 缩小名空间；3 允许对操作和表示的精化；4 允许可变数目的实例；5 比类操作更灵活。缺点：1 缺少弹性；2 不可扩展。

3 **抽象工厂模式**：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

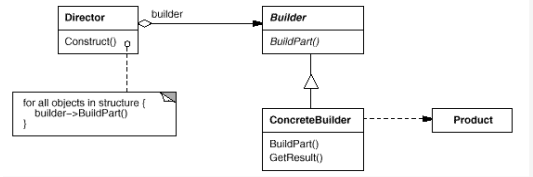
适用：1 一个系统要独立于它的产品的创建组合和表示时；2 一个系统要由多个产品系列中的一个来配置时；3 当你强调一系列相关产品对象的设计以便进行联合使用时；4 当你提供一个产品类库而只想显示它们的接口而不是实现时。

协作：1 在运行时刻创建一个 ConcreteFactory 类的实例，这一具体的工厂创建具有特定实现的产品对象。为创建不同产品对象客户应使用不同的具体工厂。2 抽象工厂将产品对象的创建延迟到它的 ConcreteFactory 子类。

效果：优缺点：1 它分离了具体的类；2 它使得易于交换产品系列；3 它有利于产品的一致性；4 难以支持新种类的产品。



4 **生成器模式**：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

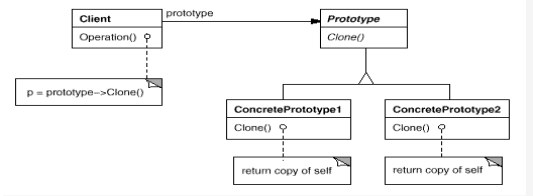


适用性：1 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时；2 当构造过程必须允许被构造的对象有不同表示时。

协作：1 客户创建 Director 对象并用它所想要的生成器对象进行配置。2 一旦产品部件被生成，导向器就会通知生成器。3 生成器处理导向器的请求，并将部件添加到该产品中。4 客户从生成器中检索产品。

效果：1 它使你可以改变一个产品的内部表示；2 它将构造代码和表示代码分开；3 它使你可对构造过程进行更精细的控制

5 **原型模式**：用原型实例制定创建对象的种类，并且通过拷贝这些原型创建新的对象。

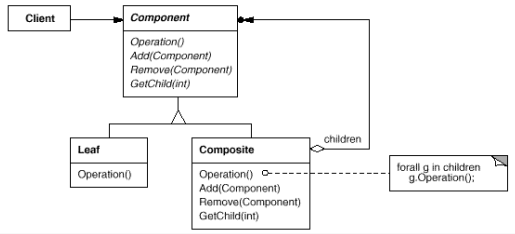


适用性：1 当一个系统应该独立于它的产品创建、构成和表示时；2 当要实例化的类是在运行时刻指定时；3 为了避免创建一个与产品类层次平行的工厂类层次时；4 当一个类的实例只能有几个不同状态组合中的一种时。

协作：客户请求一个原型克隆自身。

效果：优点：1 运行时刻增加和删除产品；2 改变值以指定新对象；3 改变结构以指定新对象；4 减少子类的构造；5 用类动态配置应用。缺陷：每一个 Prototype 的子类都必须实现 Clone 操作，可能很困难。

【结构型】6 **组合模式**：将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。



适用：1 你想表示对象的部分-整体层次结构。2 你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

协作：用户使用 Component 类接口与组合结构中的对象进行交互。若接收者是一个叶节点，则直接处理请求。如果接收者是 Composite，它通常请求发送给它的子部件，在转发请求之前/之后可能执行一些辅助操作。

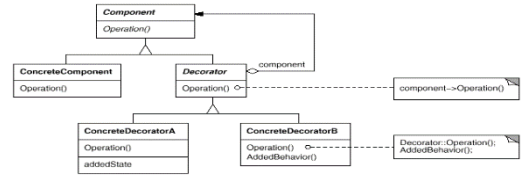
效果：1 定义了包含基本对象和组合对象的类层次结构；2 简化客户代码；3 使得更容易增加新类型的组件；4 使你的设计变得更加一般化。

7 **装饰模式**：动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

适用：1 不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。2 处理那些可以撤消的职责。3 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

协作：Decorator 将请求转发给他的 Component 对象，并有可能在转发前后执行一些附加的动作。

效果：1 比静态继承更加灵活；2 避免在层次结构高层的类有太多的特征；3Decorator 和它的 Component 不一样；4 有许多小对象。

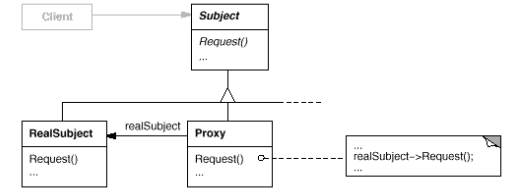


8 **代理模式**：为其他对象提供一种代理以控制对这个对象的访问。

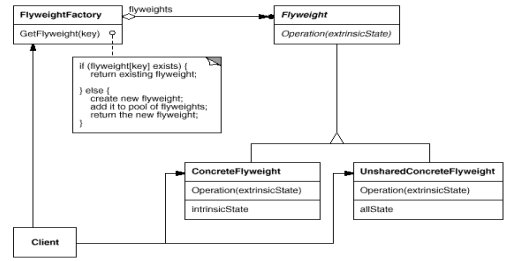
适用：在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用代理模式：远程代理、虚代理、保护代理、智能指引。

协作：代理根据其种类，在适当的时候向 RealSubject 转发请求。

效果：远程代理可以隐藏一个对象存在于不同地址空间的事实；虚代理可以进行最优化；保护代理和智能指引都允许在访问一个对象时有一些附加的内务处理。



9 **享元模式**：运用共享技术有效地支持大量细粒度的对象。



适用：1 一个应用程序使用了大量的对象。2 完全由于使用大量的对象，造成很大的存储开销。3 对象的大多数状态都可变为外部状态。4 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。5 应用程序不依赖于对象标识。由于享元对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

协作：1 享元执行时所需的状态必定是内部的或外部的，内部状态存储于 Con.Fly.对象之中；而外部对象则由 client 对象存储或计算。当用户调用享元对象的操作时，将该状态传递给它。2 用户不应该直接对 Con.Fly.类实例化，而只能从 Fly.Fac.对象得到 Con.Fly.对象，这可以保证对它们适当地进行共享。

（10 **外观模式**）：为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

11 **适配器模式**：将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。（类图见区别与联系）

适用：1 你想使用一个已经存在的类，而它的接口不符合你的需求。2 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。3（仅适用于对象适配器）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

协作：客户在适配器实例上调用一些操作。接着适配器调用 Adaptee 的操作实现这个请求。

12 **桥梁模式**：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

适用：1 你不希望在抽象和它的实现部分之间有一个固定的绑定关系。2 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。3 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。4（C + +）你想对客户完全隐藏抽象的实现部分。5 有许多类要生成。6 你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

协作：Abstraction 将 client 的请求转发给它的 Implementor 对象。

效果：优点：1 分离接口及其实现部分；2 提高可扩充性；3 实现细节对客户透明

【行为型】13 **观察者模式**：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。（类图见代码实现）

适用：1 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。2 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变。3 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

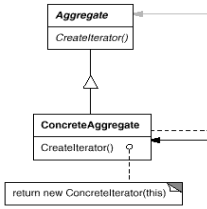
效果：1 目标和观察者间的抽象耦合。2 支持广播通信。3 意外的更新。

14 **迭代子模式**：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

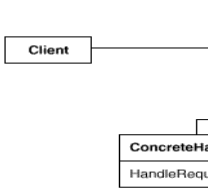
适用：1 访问一个聚合对象的内容而无需暴露它的内部表示。2 支持对聚合对象的多种遍历。3 为遍历不同的聚合结构提供一个统一的接口(即，支持多态迭代)。

协作：Con.Ite.跟踪聚合中的当前对象，并能够计算出待遍历的后继对象。

效果：1 它支持以不同的方式遍历一个集合。2 迭代器简化了聚合的接口。3 在同一个聚合上可以有多个遍历。



15 **责任链模式**：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。



适用：1 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。2 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。3 可处理一个请求的对象集合应被动态指定。

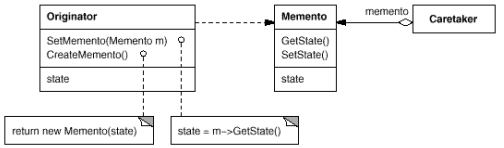
协作：当客户提交一个请求时，请求沿链传递直至一个有 ConcreteHandler 对象负责处理它。

16 **备忘录模式**：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

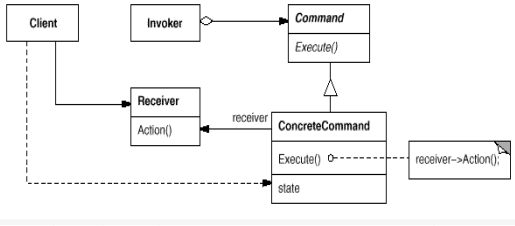
适用：1 必须保存一个对象在某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。2 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

效果：1 保持封装边界；2 它简化了原发器；3 使用备忘录可能代价很高；4 定义窄接口和宽接口；5 维护备忘录的潜在代价。

协作：1 管理器向原发器请求一个备忘录，保留一段时间后，将其送回给原发器。2 备忘录是被动的，只有创建备忘录的原发器会对它的状态进行赋值和检索。



17 **命令模式**：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。



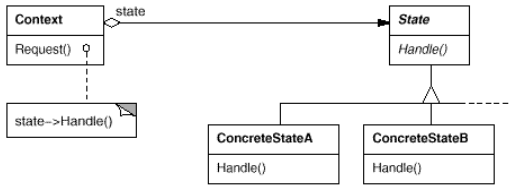
适用：1 你可用过程语言中的回调（callback）函数表达这种参数化机制。2 在不同的时刻指定、排列和执行请求。3 支持取消操作。

效果：1 命令模式将调用操作的对象与知道如何实现该操作的对象解耦。2 命令是头等的对象。它们可像其他的对象一样被操纵和扩展。3 你可以将多个命令装配成一个复合命令。4 增加新的命令很容易，因为这无需改变已有的类。

18 **状态模式**：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

适用：1 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。2 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。

效果：1 它将与特定状态相关的行为局部化。2 它使得状态转换显式化。3State 对象可被共享。

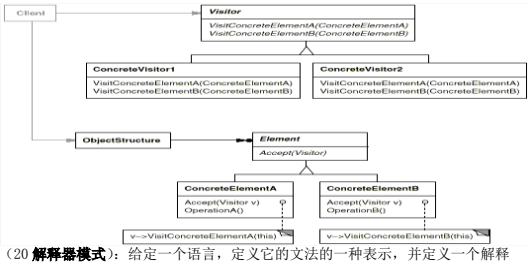


19 **访问者模式**：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

适用：1 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。2 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。

协作：1 一个使用访问者模式的客户必须创建一个 Con.Visitor 对象，然后遍历该对象结构，并用该访问者访问每一个元素。2 当一个元素被访问时，它调用对应于它的类的访问者操作。

效果：优缺点：1 访问者模式使得易于增加新的操作。2 访问者集中相关的操作而分离无关的操作。3 增加新的 ConcreteElement 类很困难。



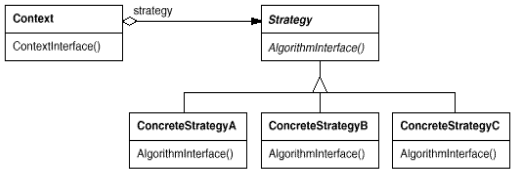
（20 **解释器模式**）：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

（21 **中介者模式**）：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

22 **策略模式**：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

适用：1 许多相关的类仅仅是行为有异。2 需要使用一个算法的不同变体。3 算法使用客户不应该知道的数据。4 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。

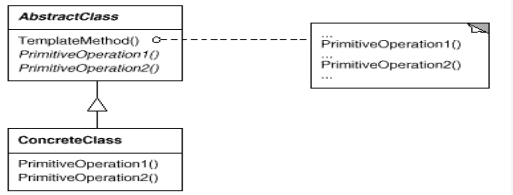
效果：优缺点：1 相关算法系列。2 一个替代继承的方法。3 消除了一些条件语句。4 实现的选择。5 客户必须了解不同的 Strategy。6Strategy 和 Context 之间的通信开销。7 增加了对象的数目。



23 **模板方法模式**：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

协作：ConcreteClass 靠 AbstractClass 来实现算法中不变的步骤。

适用：1 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。2 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。3 控制子类扩展。



【区别与联系】

● 迭代器模式和组合模式

1Composite 模式常用于组织树型结构数据，故同样会面临数据遍历的问题。树型数据的遍历主要有两种不同的算法：深度优先和广度优先。利用 Iterator，可以将具体的遍历算法屏蔽，客户端也就不再受算法的困扰。2 此外，Iterator 还可用于每个节点所包含子节点的遍历。

● 命令模式和责任链模式

1Command 模式和响应链都起到了隔离处理代码和使响应端对客户端透明的作用。2Command 模式简化和优化了 request 到处理代码的传递，为 request 的扩展提供良好支持。3 责任链只是将传递流程以链式或树型结构管理起来，为处理代码的激发顺序提供了很好的灵活性。

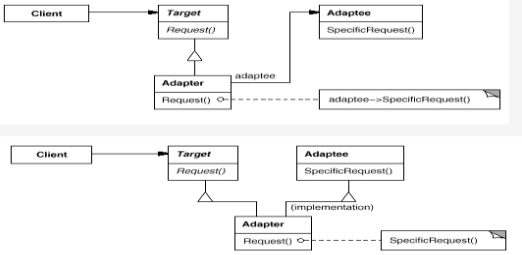
● 访问者模式和迭代器模式

1Visitor 和 Iterator 都是用于数据遍历的，其差异主要是由所采取的数据读取模式不同而引起。Visitor 使用的推(Push)模型，而 Iterator 用的则是拉(Pull)模型。2 拉模型更加简单直观，但是推模型能充分利用 OOP 的多态来简化代码，优化体系结构。

● Observer 模式与 MVC

VC(Model-View-Controller)的核心思想和 Observer 模式是完全类似的。数据模型（业务逻辑）和表现逻辑相互隔离，Controller 则负责与客户交互，根据请求调用 M 和 V 的功能。通常情况下 Controller 并不单独存在，而是和 M 或 V 结合（JFC 中便是如此）。但在分布式系统中，客户端不可能直接与 M 或 V 进行交互，此时 Controller 将独立出来并发挥重要作用。访问协议、请求分发、安全认证、日志记录和异常处理等都由其承担。总体上看，Observer 和 MVC 在本质上并没有太多的区别，只是应用范围的不同引起差异。MVC 通常会用在分布式系统中，而交互式的应用程序一般使用 Observer。

● 两种适配器模式



上半图为对象适配器，依赖于对象组合。下半图为类适配器，使用多重继承对一个接口与另一个接口进行匹配。

类适配器：用一个具体的 Adapter 类对 Adaptee 和 Target 进行匹配（但当我们想要匹配一个类以及它的子类时，Adapter 将不能胜任工作）；使得 Adapter 可以重定义 Adaptee 的部分行为；仅仅引入了一个对象，并不需要额外的指针以间接得到 Adaptee。

对象适配器：允许一个 Adapter 与多个 Adaptee 同时工作；使得重定义 Adaptee 的行为比较困难。

● 策略模式和状态模式

区别：状态模式将各个状态所对应的操作分离开来，即对于不同的状态，由不同的子类实现具体操作，不同状态的切换由子类实现，当发现传入参数不是自己这个状态所对应的参数，则自己给语境类切换状态；而策略模式是直接依赖注入到语境类的参数进行选择策略，不存在切换状态的操作。策略类对象封装一个算法；状态类对象封装一个与状态相关的行为；迭代器类对象封装访问和遍历一个聚集对象中的各个构件的方法。

● 模板方法模式和工厂方法模式的区别

工厂方法模式是类的创建模式，又叫做虚拟构造子模式或者多态性工厂模式。工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

而模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的构造子类可以以不同的方法实现这些抽象方法，从而对剩余的逻辑有不同的实现。

【体系结构部分（System Architecture）**】**

● 软件体系结构的本意：

对于大规模的，分布的，需要协作的，需要交互的，需要监测的，需要扩展的，需要演化的复杂软件系统的规划。

● SA 要素

组件 (component)、连接件(connector)、配置(configuration)、约束(constraints)

● 软件体系结构的 4+1 View

主要特色：多视图共同表达不同涉众的观点

逻辑视图(Logical View):表示系统功能。考虑功能性需求——系统需要在给用户的服务方面应该提供的。

开发视图(Development View):表示开发分工和任务管理。考虑软件模块组织——层次分析，软件管理，重用，工具约束。

进程视图(Process View):表示系统进程，线程，分布等信息。考虑非功能性需求——并发性，性能，可扩展性。

物理视图(Physical View):表示系统物理部署情况。考虑非功能性需求——关于底层硬件（拓扑，通信）。

场景(Scenarios):用一些场景、用例来描述系统各个部分之间，以及与环境之间的交互。考虑系统一致性，有效性。

● 一个风格

1 描述一类体系结构；2 在实践中被多次设计、应用；3 是若干设计思想的综合；4 具有已经被熟知的特性，并且可以复用。

● 风格由什么决定？

1 一组组件类型；2 一组连接件类型/交互机制；3 这些组件的拓扑分布；4 一组对拓扑和行为的约束；5 一些对风格的成本和益处的非正式的描述。

● 管道-过滤器风格

在管道-过滤器风格下，每个功能模块都有一组输入和输出。功能模块称作过滤器（filters）；功能模块间的连接可以看作输入、输出数据流之间的通路，所以称作管道（pipes）。

管道-过滤器风格的特性之一在于过滤器的相对独立性，即过滤器独立完成自身功能，相互之间无需进行状态交互。

特性：1 过滤器是独立运行的构件；2 过滤器对其处理上下连接的过滤器“无知”；3 结果的正确性不依赖于各个过滤器运行的先后次序。

优点：设计者可以将整个系统的输入、输出特性简单的理解为各个过滤器功能的合成。1 潜在的并行支持；2 支持死锁检测等分析；3 可维护性和可扩展性；4 支持功能模型的复用。

不足：1 交互式处理能力弱；2 管道-过滤器风格往往导致系统处理过程的成批操作；3 设计者也许不得不花费精力协调两个相对独立但又存在某种关系的数据流之间的关系；4 根据实际设计的需要，设计者也需要对数据传输进行特定的处理，增加了过滤器具体实现的复杂性。

实例：1 数字通信系统；2 Unix 系统中的管道过滤器结构；3 通讯协议的信息封装

● ADL 体系结构描述语言

ADL 是在底层语义模型的支持下，为软件系统的概念体系结构建模提供了具体语法和概念框架。基于底层语义的工具为体系结构的表示、分析、演化、细化、设计过程等提供支持。其三个基本元素是：构件、连接件、体系结构配置。

主流语言： Aesop、MetaH、C2、Rapide、SADL、Unicon 和 Wright 等；

这些 ADL 强调了体系结构不同的侧面，对体系结构的研究和应用起到了重要的作用，但也有负面的影响。每一种 ADL 都以独立的形式存在，描述语法不同且互不兼容，同时又有许多共同的特征，这使设计人员很难选择一种合适的 ADL，若设计特定领域的软件体系结构又需要从头开始描述。

● ADL 与其他语言的比较

构造能力：ADL 能够使用较小的独立体系结构元素来建造大型软件系统；

抽象能力：ADL 使得软件体系结构中的构件和连接件描述可以只关注它们的抽象特性，而不管其具体的实现细节；

重用能力：ADL 使得组成软件系统的构件、连接件甚至是软件体系结构都成为软件系统开发和设计的可重用部件；

组合能力：ADL 使得其描述的每一系统元素都有其自己的局部结构，这种描述局部结构的特点使得 ADL 支持软件系统的动态变化组合；

异构能力：ADL 允许多个不同的体系结构描述关联存在；

分析和推理能力：ADL 允许对其描述的体系结构进行多种不同的性能和功能上的多种推理分析。

● ADL 实例

1 一个常规的 Client-Server 架构；2 可容错的 C/S 架构。