策略模式 strategy pattern

Encapsulates interchangeable behaviors and use delegation to decide which one to use.
定义了算法族，分别封装起来，让它们之间可以互相替换。
此模式让算法的变化独立于使用算法的客户，由客户决定什么情况下使用具体策略角色。
体现的设计原则：
1）封装变化
2）针对接口编程而不是实现
3）多用组合少用继承
使用场景：
1）许多相关类只是在行为上不同，可以动态地在许多行为中选择一种行为。
2）动态地在几种算法中选择一种。
3）避免暴露复杂的、特定用于算法的数据结构。
4）类定义了很多表现为多个条件语句的行为，应将分支移到各自策略类中。

状态模式 state pattern

Encapsulates state-based behaviors and use delegation to switch between behaviors.
允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。
体现的设计原则：
1）单一职责原则——专门用一个状态类来表示状态。
使用场景：
1）行为随状态改变而改变。
2）条件、分支语句的替代者。

模板方法模式 template method pattern

Subclasses decide how to implement steps in an algorithm.
在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。
使子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。
工厂模式+模板方法模式：在工厂中使用模板方法模式。
模板方法设为 final，防止被修改。
体现的设计原则：开闭原则、单一职责原则、好莱坞原则。
hook()钩子：默认不做事的方法，子类视情况要不要覆盖它，可减轻抽象类的子类负荷。
若算法该部分可选就用钩子，必须实现就用抽象方法。
1）让子类实现算法中的可选部分，或者直接置之不理抛出异常。
2）让子类有机会对模板方法中即将发生的步骤作出反应。
优点：
1）由子类实现细节处理，父类实现算法骨架。
2）反向控制结构——钩子。子类可通过重写钩子方法控制父类的执行。
缺点：
1）算法中的步骤若切分的太细就没起到作用，但步骤太少的话就比较没有弹性。
2）子类必须实现抽象类中所有方法。

观察者模式 observer pattern

Allows objects to be notified when state changes.
定义了对象之间一对多依赖，当一个对象改变状态，其所有依赖者都会收到通知并自动更新。
MVC——模型：随最新状态改变而更新。
内部观察者和外部观察者的区别
推拉策略
1）推 push：subject 维护一份观察者列表，每当有更新发生，会主动把更新消息推到各个 observer 去。
高效 – 每次消息推送都是有更新之后，是有意义的。
实时 – 第一时间出发更新操作。
可由 subject 确定通知时间，避开繁忙时间。
2）拉 pull：各个 observer 维护各自关心的 subject 列表，自行决定在合适的时间从 subject 拉数据。
若 observer 众多，把订阅关系解脱到 observer 完成。
observer 可以不理会不关心的变更事件，只获取自己感兴趣的。
observer 自行决定获取更新事件的时间。
通知部分观察者时，可以使用**组合+迭代器**来通知观察者。
涉及到容器时，用观察者+命令模式。

装饰者模式 decorator pattern

Wraps an object to provide new behavior.
动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。
利用继承达到类型匹配，组合对象产生新行为。
以客户端透明的方式扩展对象功能。
体现的设计原则：开闭原则——对扩展开放，对修改关闭。
优点：灵活、动态、解耦
缺点：产生很多小对象，装饰比继承更容易出错且排错困难。
应用：扩展一个类的功能、动态增加或撤销功能。

外观模式 façade pattern

Simplifies the interface of a set of classes.
提供一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。
体现的设计原则：
1）单一职责原则——使子系统之间的依赖关系达到最小。
2）迪米特法则——不要和陌生人说话。降低客户与子系统的耦合度。
优点：
1）实现子系统与客户之间的松耦合，子系统变化不会影响客户类。
2）客户处理对象数目变少，使子系统更易使用。
3）可附加“聪明的”功能，一个子系统可有多个外观。
缺点：
1）违背开闭原则，不引入抽象外观类的情况下，添加子系统可能要修改外观类。
2）不能很好地限制客户使用子系统类。

适配器模式 adapter pattern

Wraps an object and provides a different interface to it.
将一个类的接口，转换成客户期望的另一个接口。适配器让原本不兼容的类可以合作无间。
适配器实现目标接口 target，并持有被适配者的实例。
体现的设计原则：合成复用原则
优点：
1）客户从实现的接口中解耦。
2）提高了类的复用性、灵活性、透明性。
3）可以让任何两个没有关联的类在一起运行。
缺点：
1）过多使用适配器，会让系统非常凌乱，不易整体进行把握。
2）Java 不支持多重继承。

工厂模式 factory pattern

Subclasses decide which concrete classes to create.
定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。让类把实例化推迟到子类。
封装具体类型的实例化，将产品的实现从使用中解耦。
体现的设计原则：开闭原则——工厂方法模式符合，简单工厂不符合。
策略模式与工厂模式一起使用，用工厂来创建算法类。
使用场景：明确地计划不同条件创建不同实例时。
优点：
1）一个调用者想创建一个对象，只要知道其名称就可以了。
2）扩展性高，可通过扩展一个工厂类来增加产品。
3）屏蔽产品的具体实现，调用者只关心接口，形成解耦。
缺点：每增加一个产品，都需要增加一个具体类和对象实现工厂，使系统中类数量成倍增加。

抽象工厂模式 abstract factory pattern

Allows a client to create families of objects without specifying their concrete classes.
提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。
客户从具体产品中被解耦。
优点：
1）一个产品系列多个对象使用时，保证使用的是同系列的对象。
2）降低具体产品和具体类及客户端之间的耦合性，通过抽象类相关联而不是具体类。
缺点：在系列产品中新增产品困难并且违反了开闭原则。
使用场景：多套系列相互依赖的产品。

单例模式 singleton

Ensures one and only one object is created.
确保一个类只有一个实例，并提供一个全局访问点。
构成：一个公有的静态工厂方法、一个静态私有成员变量、一个私有的构造函数。

Multithreading 多线程

处理多线程：将 getInstance()变成同步(synchronized)方法，即可避免多线程灾难。

改善多线程：只有第一次执行方法时才需要同步，之后每次调用都是一种累赘。

1）性能不关键的话，什么都不做。

2）使用急切（eagerly）创建实例，而不用延迟实例化的做法。

```
private static Singleton uniqueInstance = new Singleton();
```

3）用“双重检查加锁 double-checked locking”，在 getInstance()减少同步。<Java5 版本才可>

```
private volatile static Singleton uniqueInstance;
private Singleton() {}
private static Singleton getInstance() {
        if(uniqueInstance == NULL) {
                synchronized (Singleton.class) {
                        if(uniqueInstance == NULL) {
                                uniqueInstance = new Singleton(); }}}
        return uniqueInstance; }
```

优点：

1）内存中只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。

2）避免对资源的多重占用。

缺点：没有接口，不能继承，与单一职责原则冲突。不适用于变化频繁的对象。

饿汉：类定义的时候就实例化了（不存在多线程问题）。

懒汉：等到第一次用的时候实例化。

二阶段锁：double-checked locking

命令模式 command pattern

Encapsulate a request as an object.

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。

角色职责：

1）client – 发送者

2）receiver – 接收者 – 真正的命令执行对象

3）invoker – 调用者 – 使用命令对象的入口，决定何时执行

4）command – 命令

使 client 和 receiver 解耦，使 invoker 和 receiver 解耦。

undo() & redo() 撤销与重做：

1）记录前一个命令：加实例变量，用来追踪最后被调用的命令。（状态实现）

2）用堆栈记录。

MacroCommand 宏命令（组合命令）：

用命令数组存储一系列命令，当这个宏命令被执行时，就一次性执行数组里的每个命令。

用**组合模式**来代替数组，用迭代器模式来执行数组里的命令。

体现的设计原则：开闭原则——松耦合，新命令很容易加入。

更多用途：队列请求（工作队列类与计算的对象解耦）、日志请求（死机后加载动作恢复之前状态）

模式应用：Java 中的 AWT/Swing GUI 的委派事件模型、Unix 下的 shell 宏命令功能。

迭代器模式 iterator pattern

provides a way to traverse a collection of objects without exposing its implementation.

提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

游走的任务在迭代器上，而不是聚合上，简化了聚合的接口和实现，也让责任各得其所。

内部迭代器：自行在元素间游走。

外部迭代器：客户通过调用 next()取得下一个元素。
向前移动的迭代器：Java 的 collection framework 提供 ListIterator 的 previous()方法。
工厂模式的应用：aggregate 聚合类→工厂类，iterator 迭代类→产品类。
体现的设计模式：单一职责原则

组合模式 composite pattern

Clients treat collections of objects and individual objects uniformly.
允许你将对象组合成树形结构来表现“整体/部分”层次结构。
让客户以一致的方式处理个别对象以及对象的组合。
组合 composite 包括组件 component，组件包括组合 composite 和叶节点 leaf。
透明性 transparency 和安全性 safety：
1）组合和叶节点具有相同的方法，其中包括 add()方法。
→叶节点和组合之间是透明的，客户根本不用管究竟是组合还是叶节点，调用两者的同一方法。
2）但把组合维护孩子的方法和叶节点分开，只让 flock 具有 add()方法，给 duck 添加无意义。
→安全性。不用调用无意义的方法，但透明性差，若用户想用 add()方法，得确定对象是 flock 才行。
相对安全性，比较强调透明性，不需要的方法可以用空处理或异常报告方式解决。
客户希望代码是透明的，更注重容器内组合结构的时候，可牺牲透明，实现安全组合模式。
观察者模式、命令模式。

代理模式 proxy pattern

Wraps an object to control access to it.
以另一个对象提供一个替身或占位符以控制对这个对象的访问。
使用代理模式创建代表 representative 对象，让代表对象控制对某对象的访问。
提供一个工厂，实例化并返回主题，使客户解耦。
变体：都是将客户对主题施加的方法调用拦截下来。间接→可以做许多事。
远程代理：远程对象
虚拟代理：创建开销大的资源对象
保护代理：安全控制（权限控制）的对象

复合模式 compound pattern

Q1 对比适配器模式、外观模式、代理模式、装饰者模式的目的。【结构型】
相同点：都作用于客户与真实被使用的类或系统之间，作为一个中间层，起到了解耦的作用。
适配器模式：转换行为
外观模式：简化行为
代理模式：控制访问行为

装饰者模式：新增行为

Q2 对比策略模式、状态模式。

1）状态模式要对客户透明。策略模式不能对客户透明。

2）状态模式的行为封装在状态对象中，当前状态在状态对象中游走改变，以反映出 context 内部的状态，客户不关心具体状态以及状态转换过程。策略模式常常是客户根据环境或条件的不同主动指定 context 所要组合的策略对象是哪一个，可在运行时改变策略方案，具有弹性。

3）状态模式可作为条件判断语句的替代。策略模式可作为继承之外的弹性替代方案。

4）状态模式是某个类的对象由多种状态且不同状态下行为有差异。策略模式是某类的某行为有多种实现方式。

Q3 类适配器和对象适配器（结构、目的、优缺点）。

结构上：类适配器使通过继承 adaptee，并实现 target 方法，是静态的方式。对象适配器是通过委托与 adaptee 衔接，即持有 adaptee 对象，是动态的方式。

目的上：类适配器的重点在于类，通过构造一个继承 adaptee 类来实现适配器功能。对象适配器的重点在于对象，通过在直接包含 adaptee 类来实现的，需要调用特殊功能时直接使用 adapter 包含的 adaptee 对象来调用方法。

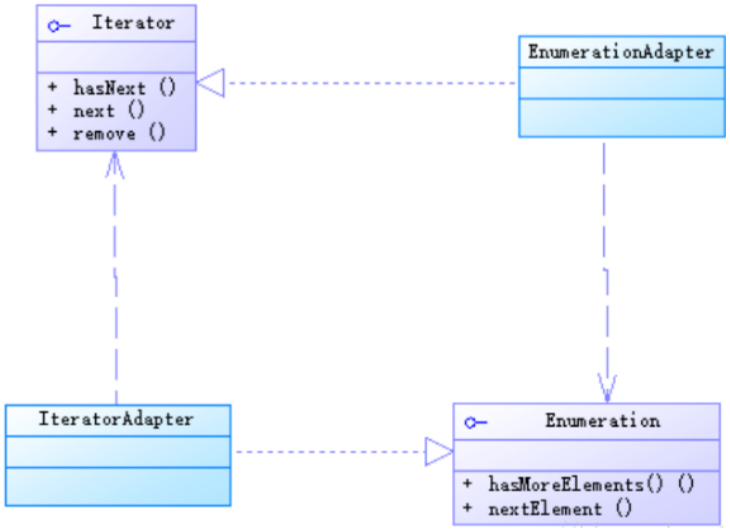
类适配器：优点是可在适配器类中置换一些适配者的方法，增强灵活性。缺点是对于不支持多重继承的语言，一次只能适配一个适配者类，而且目标抽象类只能为接口，使用上有一定局限性。

对象适配器：优点是可把多个不同的适配者适配到同一个目标，缺点是置换适配者类的方法不容易。

Q4 双向适配器。

适配器中同时包含对目标类和适配者类的引用，适配者可以通过它调用目标类中的方法，目标类也可以通过它调用适配者类中的方法，那么该适配器就是一个双向适配器。

Iterator 与 Enumeration 的双向适配



Enumeration 接口兼容 Iterator 接口

```
public class IteratorAdapter<E> implements Enumeration<E>
    Iterator<E> iterator;
    public IteratorAdapter(Iterator<E> iterator) {
        this.iterator = iterator;
    }
    @Override
    public boolean hasMoreElements() {
        return iterator.hasNext();
    }
    @Override
    public E nextElement() {
        return iterator.next();
    }
}
```

Iterator 兼容 Enumeration 接口

```
public class EnumerationAdapter<E> implements Iterator<E> {
    Enumeration<E> enumeration;
    public EnumerationAdapter(Enumeration<E> enumeration) {
        this.enumeration = enumeration;
    }
    @Override
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }
    @Override
    public E next() {
        return enumeration.nextElement();
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```

Q5 工厂方法模式、抽象工厂模式、简单工厂。

抽象工厂的退化: 当抽象工厂模式中每一个具体工厂类只创建一个产品对象, 也就是只存在一个产品等级结构时, 抽象工厂模式就退化成工厂方法模式。当工厂方法模式中抽象工厂与具体工厂合并，提供一个统一的工厂来创建产品对象，并将创建对象地工厂方法设计为静态方法时，工厂方法模式退化成简单工厂模式。

共同点：都封装了对象的创建。通过减少客户与具体类之间的依赖促进松耦合。

工厂方法模式：继承；通过子类创建对象；抽象创建者中实现的代码会用到子类创建的具体类型。
抽象工厂模式：对象的组合；创建产品家族的抽象类型；具体工厂经常实现工厂方法来创建产品。

工厂方法模式：创建一个框架，让子类决定要如何实现。
简单工厂：把全部事情在一个地方处理完了，将创建封装起来，不能变更正在创建的产品。

Q6 模板方法模式与策略模式

共同点：都封装算法。
模板方法模式：定义算法大纲，子类定义其中某些步骤的内容。通过继承。对算法有更多控制权，重复代码少。
策略模式：定义算法家族。通过组合。更有弹性，可以在运行时改变算法。

Q7 MVC 模式

model – 模型 – 观察者模式
view – 视图 – 组合模式
control – 控制 – 策略模式