总体来说设计模式分为三大类：

创建型模式，共五种：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，共七种：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，共十一种：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。

其实还有两类：并发型模式和线程池模式。

原则1：使用多个隔离的接口，比使用单个接口要好；一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

单例模式：某个类只能有一个实例，提供一个全局的访问点。

简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

https://www.cnblogs.com/java-my-life/archive/2012/03/22/2412308.html

工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。http://www.cnblogs.com/java-my-life/archive/2012/03/28/2418836.html

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

http://www.cnblogs.com/java-my-life/archive/2012/04/07/2433939.html

原型模式：通过复制现有的实例来创建新的实例。

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

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。

装饰模式：动态的给对象添加新的功能。

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

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。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

观察者模式：对象间的一对多的依赖关系。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

中介者模式：用一个中介对象来封装一系列的对象交互。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

工厂方法模式分为三种：

1、普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建；

例：首先，创建二者的共同接口：

1. public interface Sender {
2. public void Send();
3. }
4. 其次，创建实现类：
5. public class MailSender implements Sender {
6. @Override
7. public void Send() {
8. System.out.println("this is mailsender!");
9. }
10. }
11. public class SmsSender implements Sender {
12. @Override
13. public void Send() {
14. System.out.println("this is sms sender!");
15. }
16. }
17. 最后，建工厂类：
18. public class SendFactory {
19. public Sender produce(String type) {
20. if ("mail".equals(type)) {
21. return new MailSender();
22. } else if ("sms".equals(type)) {
23. return new SmsSender();
24. } else {
25. System.out.println("请输入正确的类型!");
26. return null;
27. }
28. }
29. }
30. 测试：
31. public class FactoryTest {
32. public static void main(String[] args) {
33. SendFactory factory = new SendFactory();
34. Sender sender = factory.produce("sms");
35. sender.Send();
36. }
37. }

2、多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象；

例：将普通工厂模式的代码做下修改，改动下SendFactory类就行

1. public class SendFactory {
3. public Sender produceMail(){
4. return new MailSender();
5. }
7. public Sender produceSms(){
8. return new SmsSender();
9. }
10. }
11. 测试：
12. public class FactoryTest {
13. public static void main(String[] args) {
14. SendFactory factory = new SendFactory();
15. Sender sender = factory.produceMail();
16. sender.Send();
17. }
18. }

3、静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可；

1. public class SendFactory {
3. public static Sender produceMail(){
4. return new MailSender();
5. }
7. public static Sender produceSms(){
8. return new SmsSender();
9. }
10. }
11. 测试：
12. public class FactoryTest {
13. public static void main(String[] args) {
14. Sender sender = SendFactory.produceMail();
15. sender.Send();
16. }
17. }

工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。主要是方便灵活的创建对象的

**抽象工厂模式：**

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则（对扩展开放，对修改关闭），所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

**在什么情况下应当使用抽象工厂模式**

　　1.一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。

　　2.这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

　　3.同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。（比如：Intel主板必须使用Intel CPU、Intel芯片组）

4.系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

**抽象工厂模式的优点**

分离接口和实现

　　客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说，客户端从具体的产品实现中解耦。

使切换产品族变得容易

　　因为一个具体的工厂实现代表的是一个产品族，比如上面例子的从Intel系列到AMD系列只需要切换一下具体工厂。

抽象工厂模式的缺点

不太容易扩展新的产品

　　如果需要给整个产品族添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。

单例模式：

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

例：

1. public class Singleton {
3. /\* 私有构造方法，防止被实例化 \*/
4. private Singleton() {
5. }
7. /\* 此处使用一个内部类来维护单例 \*/
8. private static class SingletonFactory {
9. private static Singleton instance = new Singleton();
10. }
12. /\* 获取实例 \*/
13. public static Singleton getInstance() {
14. return SingletonFactory.instance;
15. }
17. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
18. public Object readResolve() {
19. return getInstance();
20. }
21. }

JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕。

**建造者模式：**

**建造模式是对象的创建模式。建造模式可以将一个产品的内部表象（internal representation）与产品的生产过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。**

**简**单的讲就是：将一个产品类通过建造者添加一些特殊功能到产品类中，之后通过调用构造方法来一部分一部分地建造出产品对象

建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。

例：和前面一样，一个Sender接口，两个实现类MailSender和SmsSender

1. public class Builder {
3. private List<Sender> list = new ArrayList<Sender>();
5. public void produceMailSender(int count){
6. for(int i=0; i<count; i++){
7. list.add(new MailSender());
8. }
9. }
11. public void produceSmsSender(int count){
12. for(int i=0; i<count; i++){
13. list.add(new SmsSender());
14. }
15. }
16. }
17. public class Test {
18. public static void main(String[] args) {
19. Builder builder = new Builder();
20. builder.produceMailSender(10);
21. }
22. }

# 在什么情况下使用建造模式

　　1. 需要生成的产品对象有复杂的内部结构，每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成部分。

　　2. 需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式是一个很好的设计思想。

　　3. 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。

适配器模式：

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式：

例：核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里：

1. public class Source {
2. **public** **void** method1() {
3. System.out.println("this is original method!");
4. }
5. }
6. **public** **interface** Targetable {
8. /\* 与原类中的方法相同 \*/
9. **public** **void** method1();
11. /\* 新类的方法 \*/
12. **public** **void** method2();
13. }
14. **public** **class** Adapter **extends** Source **implements** Targetable {
15. @Override
16. **public** **void** method2() {
17. System.out.println("this is the targetable method!");
18. }
19. }
20. Adapter类继承Source类，实现Targetable接口，下面是测试类：
21. **public** **class** AdapterTest {
22. **public** **static** **void** main(String[] args) {
23. Targetable target = **new** Adapter();
24. target.method1();
25. target.method2();
26. }
27. }
28. 这样Targetable接口的实现类就具有了Source类的功能。

对象的适配器模式：

例：基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题：

只需要修改Adapter类的源码即可：

1. public class Wrapper implements Targetable {
3. private Source source;
5. public Wrapper(Source source){
6. super();
7. this.source = source;
8. }
9. @Override
10. public void method2() {
11. System.out.println("this is the targetable method!");
12. }
14. @Override
15. public void method1() {
16. source.method1();
17. }
18. }
19. 测试类：
20. public class AdapterTest {
21. public static void main(String[] args) {
22. Source source = new Source();
23. Targetable target = new Wrapper(source);
24. target.method1();
25. target.method2();
26. }
27. }

接口的适配器模式：

有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

例：

1. public interface Sourceable {
2. public void method1();
3. public void method2();
4. }
5. 抽象类Wrapper2：
6. public abstract class Wrapper2 implements Sourceable{
7. public void method1(){}
8. public void method2(){}
9. }
10. public class SourceSub1 extends Wrapper2 {
11. public void method1(){
12. System.out.println("the sourceable interface's first Sub1!");
13. }
14. }
15. public class SourceSub2 extends Wrapper2 {
16. public void method2(){
17. System.out.println("the sourceable interface's second Sub2!");
18. }
19. }
20. 测试：
21. public class WrapperTest {
23. public static void main(String[] args) {
24. Sourceable source1 = new SourceSub1();
25. Sourceable source2 = new SourceSub2();
27. source1.method1();
28. source1.method2();
29. source2.method1();
30. source2.method2();
31. }
32. }
33. 结果：
34. the sourceable interface's first Sub1!
35. the sourceable interface's second Sub2!

比较：类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

装饰模式：

装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，

例：Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能：

1. public interface Sourceable {
2. public void method();
3. }
5. public class Source implements Sourceable {
6. @Override
7. public void method() {
8. System.out.println("the original method!");
9. }
10. }
11. public class Decorator implements Sourceable {
13. private Sourceable source;
15. public Decorator(Sourceable source){
16. super();
17. this.source = source;
18. }
19. @Override
20. public void method() {
21. System.out.println("before decorator!");
22. source.method();
23. System.out.println("after decorator!");
24. }
25. }
26. 测试：
27. public class DecoratorTest {
29. public static void main(String[] args) {
30. Sourceable source = new Source();
31. Sourceable obj = new Decorator(source);
32. obj.method();
33. }
34. }
35. 输出：
36. before decorator!  
    the original method!  
    after decorator!

装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

代理模式：

代理模式就是多一个代理类出来，替原对象进行一些操作.

例：

1. public interface Sourceable {
2. public void method();
3. }
4. public class Source implements Sourceable {
5. @Override
6. public void method() {
7. System.out.println("the original method!");
8. }
9. }
10. public class Proxy implements Sourceable {
12. private Source source;
13. public Proxy(){
14. super();
15. this.source = new Source();
16. }
17. @Override
18. public void method() {
19. before();
20. source.method();
21. atfer();
22. }
23. private void atfer() {
24. System.out.println("after proxy!");
25. }
26. private void before() {
27. System.out.println("before proxy!");
28. }
29. }
30. 测试：
31. public class ProxyTest {
32. public static void main(String[] args) {
33. Sourceable source = new Proxy();
34. source.method();
35. }
36. }
37. 输出：
38. before proxy!  
    the original method!  
    after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

外观模式：

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口：

1. 类cpu
2. public class CPU {
4. public void startup(){
5. System.out.println("cpu startup!");
6. }
8. public void shutdown(){
9. System.out.println("cpu shutdown!");
10. }
11. }
12. 类memory
13. public class Memory {
15. public void startup(){
16. System.out.println("memory startup!");
17. }
19. public void shutdown(){
20. System.out.println("memory shutdown!");
21. }
22. }
23. 类disk
24. public class Disk {
26. public void startup(){
27. System.out.println("disk startup!");
28. }
30. public void shutdown(){
31. System.out.println("disk shutdown!");
32. }
33. }
34. 类computer
35. public class Computer {
36. private CPU cpu;
37. private Memory memory;
38. private Disk disk;
40. public Computer(){
41. cpu = new CPU();
42. memory = new Memory();
43. disk = new Disk();
44. }
46. public void startup(){
47. System.out.println("start the computer!");
48. cpu.startup();
49. memory.startup();
50. disk.startup();
51. System.out.println("start computer finished!");
52. }
54. public void shutdown(){
55. System.out.println("begin to close the computer!");
56. cpu.shutdown();
57. memory.shutdown();
58. disk.shutdown();
59. System.out.println("computer closed!");
60. }
61. }
62. user类
63. public class User {
65. public static void main(String[] args) {
66. Computer computer = new Computer();
67. computer.startup();
68. computer.shutdown();
69. }
70. }
71. 输出
72. start the computer!
73. cpu startup!
74. memory startup!
75. disk startup!
76. start computer finished!
77. begin to close the computer!
78. cpu shutdown!
79. memory shutdown!
80. disk shutdown!
81. computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观模式！

桥接模式：

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了：

例：

1. public interface Sourceable {
2. public void method();
3. }
4. 两个实现类
5. public class SourceSub1 implements Sourceable {
7. @Override
8. public void method() {
9. System.out.println("this is the first sub!");
10. }
11. }
12. public class SourceSub2 implements Sourceable {
14. @Override
15. public void method() {
16. System.out.println("this is the second sub!");
17. }
18. }
19. 定义一个桥，持有Sourceable的一个实例：
20. public abstract class Bridge {
21. private Sourceable source;
23. public void method(){
24. source.method();
25. }
27. public Sourceable getSource() {
28. return source;
29. }
31. public void setSource(Sourceable source) {
32. this.source = source;
33. }
34. }
35. public class MyBridge extends Bridge {
36. public void method(){
37. getSource().method();
38. }
39. }
40. 测试：
41. public class BridgeTest {
43. public static void main(String[] args) {
45. Bridge bridge = new MyBridge();
47. /\*调用第一个对象\*/
48. Sourceable source1 = new SourceSub1();
49. bridge.setSource(source1);
50. bridge.method();
52. /\*调用第二个对象\*/
53. Sourceable source2 = new SourceSub2();
54. bridge.setSource(source2);
55. bridge.method();
56. }
57. }
58. 输出：
59. this is the first sub!
60. this is the second sub!

通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。

组合模式：

组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便：

1. public class TreeNode {
3. private String name;
4. private TreeNode parent;
5. private Vector<TreeNode> children = new Vector<TreeNode>();
7. public TreeNode(String name){
8. this.name = name;
9. }
11. public String getName() {
12. return name;
13. }
15. public void setName(String name) {
16. this.name = name;
17. }
19. public TreeNode getParent() {
20. return parent;
21. }
23. public void setParent(TreeNode parent) {
24. this.parent = parent;
25. }
27. //添加孩子节点
28. public void add(TreeNode node){
29. children.add(node);
30. }
32. //删除孩子节点
33. public void remove(TreeNode node){
34. children.remove(node);
35. }
37. //取得孩子节点
38. public Enumeration<TreeNode> getChildren(){
39. return children.elements();
40. }
41. }
43. public class Tree {
45. TreeNode root = null;
47. public Tree(String name) {
48. root = new TreeNode(name);
49. }
51. public static void main(String[] args) {
52. Tree tree = new Tree("A");
53. TreeNode nodeB = new TreeNode("B");
54. TreeNode nodeC = new TreeNode("C");
56. nodeB.add(nodeC);
57. tree.root.add(nodeB);
58. System.out.println("build the tree finished!");
59. }
60. }

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

享元模式：

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。类似Java里面的JDBC连接池：

1. **public** **class** ConnectionPool {
3. **private** Vector<Connection> pool;
5. /\*公有属性\*/
6. **private** String url = "jdbc:mysql://localhost:3306/test";
7. **private** String username = "root";
8. **private** String password = "root";
9. **private** String driverClassName = "com.mysql.jdbc.Driver";
11. **private** **int** poolSize = 100;
12. **private** **static** ConnectionPool instance = **null**;
13. Connection conn = **null**;
15. /\*构造方法，做一些初始化工作\*/
16. **private** ConnectionPool() {
17. pool = **new** Vector<Connection>(poolSize);
19. **for** (**int** i = 0; i < poolSize; i++) {
20. **try** {
21. Class.forName(driverClassName);
22. conn = DriverManager.getConnection(url, username, password);
23. pool.add(conn);
24. } **catch** (ClassNotFoundException e) {
25. e.printStackTrace();
26. } **catch** (SQLException e) {
27. e.printStackTrace();
28. }
29. }
30. }
32. /\* 返回连接到连接池 \*/
33. **public** **synchronized** **void** release() {
34. pool.add(conn);
35. }
37. /\* 返回连接池中的一个数据库连接 \*/
38. **public** **synchronized** Connection getConnection() {
39. **if** (pool.size() > 0) {
40. Connection conn = pool.get(0);
41. pool.remove(conn);
42. **return** conn;
43. } **else** {
44. **return** **null**;
45. }
46. }
47. }

策略模式：

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数：

例：统一接口：

1. **public** **interface** ICalculator {
2. **public** **int** calculate(String exp);
3. }
4. 辅助类：
5. **public** **abstract** **class** AbstractCalculator {
7. **public** **int**[] split(String exp,String opt){
8. String array[] = exp.split(opt);
9. **int** arrayInt[] = **new** **int**[2];
10. arrayInt[0] = Integer.parseInt(array[0]);
11. arrayInt[1] = Integer.parseInt(array[1]);
12. **return** arrayInt;
13. }
14. }
15. 三个实现类：
16. **public** **class** Plus **extends** AbstractCalculator **implements** ICalculator {
17. @Override
18. **public** **int** calculate(String exp) {
19. **int** arrayInt[] = split(exp,"\\+");
20. **return** arrayInt[0]+arrayInt[1];
21. }
22. }
23. **public** **class** Minus **extends** AbstractCalculator **implements** ICalculator {
24. @Override
25. **public** **int** calculate(String exp) {
26. **int** arrayInt[] = split(exp,"-");
27. **return** arrayInt[0]-arrayInt[1];
28. }
29. }
31. **public** **class** Multiply **extends** AbstractCalculator **implements** ICalculator {
32. @Override
33. **public** **int** calculate(String exp) {
34. **int** arrayInt[] = split(exp,"\\\*");
35. **return** arrayInt[0]\*arrayInt[1];
36. }
37. }
38. 测试类：
39. **public** **class** StrategyTest {
41. **public** **static** **void** main(String[] args) {
42. String exp = "2+8";
43. ICalculator cal = **new** Plus();
44. **int** result = cal.calculate(exp);
45. System.out.println(result);
46. }
47. }
48. 输出：10
49. 策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

模板方法模式：

一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用：

例：就是在AbstractCalculator类中定义一个主方法calculate，calculate()调用spilt()等，Plus和Minus分别继承AbstractCalculator类，通过对AbstractCalculator的调用实现对子类的调用：

1. public abstract class AbstractCalculator {
3. /\*主方法，实现对本类其它方法的调用\*/
4. public final int calculate(String exp,String opt){
5. int array[] = split(exp,opt);
6. return calculate(array[0],array[1]);
7. }
9. /\*被子类重写的方法\*/
10. abstract public int calculate(int num1,int num2);
12. public int[] split(String exp,String opt){
13. String array[] = exp.split(opt);
14. int arrayInt[] = new int[2];
15. arrayInt[0] = Integer.parseInt(array[0]);
16. arrayInt[1] = Integer.parseInt(array[1]);
17. return arrayInt;
18. }
19. }
20. public class Plus extends AbstractCalculator {
21. @Override
22. public int calculate(int num1,int num2) {
23. return num1 + num2;
24. }
25. }
26. 测试：
27. public class StrategyTest {
29. public static void main(String[] args) {
30. String exp = "8+8";
31. AbstractCalculator cal = new Plus();
32. int result = cal.calculate(exp, "\\+");
33. System.out.println(result);
34. }
35. }

首先将exp和"\\+"做参数，调用AbstractCalculator类里的calculate(String,String)方法，在calculate(String,String)里调用同类的split()，之后再调用calculate(int ,int)方法，从这个方法进入到子类中，执行完return num1 + num2后，将值返回到AbstractCalculator类，赋给result，打印出来。

观察者模式：

简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

一个Observer接口：

1. public interface Observer {
2. public void update();
3. }
4. 两个实现类：
5. public class Observer1 implements Observer {
6. @Override
7. public void update() {
8. System.out.println("observer1 has received!");
9. }
10. }
11. public class Observer2 implements Observer {
12. @Override
13. public void update() {
14. System.out.println("observer2 has received!");
15. }
16. }
17. Subject接口及实现类：
18. public interface Subject {
20. /\*增加观察者\*/
21. public void add(Observer observer);
23. /\*删除观察者\*/
24. public void del(Observer observer);
26. /\*通知所有的观察者\*/
27. public void notifyObservers();
29. /\*自身的操作\*/
30. public void operation();
31. }
32. 实现类
33. public abstract class AbstractSubject implements Subject {
35. private Vector<Observer> vector = new Vector<Observer>();
36. @Override
37. public void add(Observer observer) {
38. vector.add(observer);
39. }
41. @Override
42. public void del(Observer observer) {
43. vector.remove(observer);
44. }
46. @Override
47. public void notifyObservers() {
48. Enumeration<Observer> enumo = vector.elements();
49. while(enumo.hasMoreElements()){
50. enumo.nextElement().update();
51. }
52. }
53. }
55. ublic class MySubject extends AbstractSubject {
56. @Override
57. public void operation() {
58. System.out.println("update self!");
59. notifyObservers();
60. }
61. }
62. 测试类：
63. public class ObserverTest {
64. public static void main(String[] args) {
65. Subject sub = new MySubject();
66. sub.add(new Observer1());
67. sub.add(new Observer2());
69. sub.operation();
70. }
71. }
72. 输出：
73. update self!  
    observer1 has received!  
    observer2 has received!