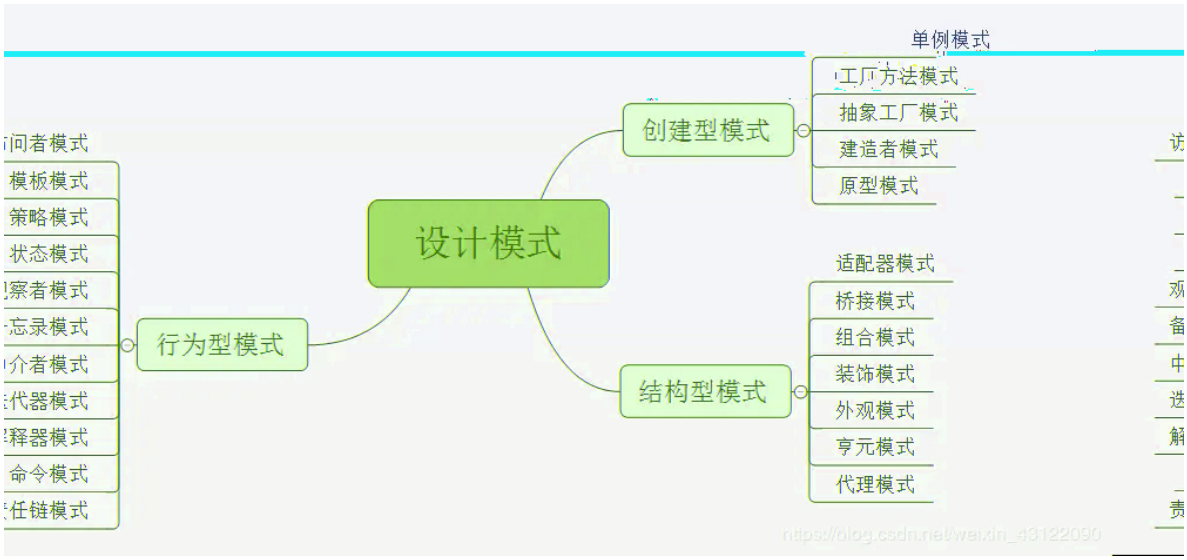
1.什么是设计模式

- ， 套 反复使 多 人 分 代 使
为了可 代 代 容 他人 保 代 可 序

2.为什么要学习设计模式

- 代 ：如 你不 去 Jdk Spring SpringMVC IO ，你会
，你会寸
- 前 代 ：你去个公司 你 ？ 可 ，前 发 不
？
- 写 己 中 好代 ： 个人反 ，对于 己 发 会 ， 对他
对 女 友 好， 己 儿子

3.设计模式分类



- 创 型 ，共五 ：**工厂方法模式 抽象工厂模式 单例模式 原型模式**
- 型 ，共七 ： 器 器 代 外 合 享元
- 为型 ，共十 ： 察 代子 任 命令 备 中介 器

4.设计模式的六大原则

开放封 原则（Open Close Principle）

- 原则 ：尽 展 件实体 决 变化， 不 修 已 代 完 变化
- ： 个 件产品在 命周 内， 会发 变化， 变化 个 定 事实， 们就应 在 候尽 应 些变化，以 定 和
- 优 ：单 原则告 们， 个 己 ， 原则不 坏 关 体

氏代换原则（Liskov Substitution Principle）

- 原则 ：使 基 可以在任何地 使 子 ，完 基
- 大 ：子 可以 展 功 ，但不 变 原 功 子 可以实 ，但不 ，子 中可以增加 己
- 优 ：增加 序 健 ，即使增加了子 ，原 子 可以 ，互不 响

依赖倒 原则（Dependence Inversion Principle）

- 依 倒 原则 向 口 .
- 依 倒 原则 们在 序代 中传 参 在关 关 中，尽 层 层 ，
- 个 封 原则 基 ，具体内容 ：对 口 ，依 于 不依 于具体

接口 离原则（Interface Segregation Principle）

- 个原则 ：使 多个 口， 使 单个 口 好 个 低 之 合度 ，从 儿 们 出，其实 就 个 件 ，从大型 件 出发，为了升和 便 以上 中多 出 ： 低依 ， 低 合
- 例如： 付 口和 单 口， 个 别 口变 个 口

米特法则（最少知 原则）（Demeter Principle）

- 原则 ： 个对 应 对其他对 尽可 少地了 ，
- 大 就 个 尽 减少 己对其他对 依 ，原则 低 合， 内 ，只 使各个 块之 合尽 低， 代 复
- 优 ：低 合， 内

单 职责原则（Principle of single responsibility）

- 原则 ： 个 只 件事
- ：单 原则 单， 个 个 只 个 ，各个 序 动，不 响其它 序 常 ，几乎 序员 会 个原则
- 优 ： 低 和 合， 可 ，增加可 和可 展 ， 低可变

5.单例模式

1.什么是单例

- 保 个 只 个实例，并且 供 个 全局

2. 些地方用到了单例模式

1. 器， 也 单例 实 ，否则 以同
2. 应 序 应 ， 单例 实 ，只 个实例去 作 好，否则内容不好 加
3. 多 也 单例 ，因为 便对 中 制
4. Windows （任务 器）就 典型 单例 ，他不 个
5. windows （回 ）也 典型 单例应 在 个 中，回 只 个实例

3.单例优缺

优 ：

1. 在单例 中， 动 单例只 个实例，对单例 实例化 到 同 个实例 就 其它对 对 己 实例化， 保 对 个实例
2. 单例 具 定 伸 ， 己 制实例化 ， 就在 变实例化 上 应 伸
3. 供了对唯 实例 受
4. 于在 内存中只存在 个对 ，因 可以 ， 创 和 对 单例 可以
5. 允 可变 实例
6. 免对共享 多 占

缺 ：

1. 不 于变化 对 ，如 同 型 对 在不同 例场 发 变化，单例就会 ，不 保存
2. 于单利 中 层，因 单例 展 大 困
3. 单例 ，在 定 度上 了“单 原则”

4. 单例将带 些 ，如为了 将 库 对 为 单例 ，可 会导 共享 对 序 多 出 出；如 实例化 对 不 利 ， 会 为 垃圾 回 ， 将导 对 丢失

4.单例模式使用注意事 ：

1. 使 不 反射 创 单例，否则会实例化 个 对
2. 使 单例 安全
3. 单例 和 单例 ，因 不 ， 些单例 可以 （如 ）

5.单例 止反射漏洞攻击

```
private static boolean flag = false;

private Singleton() {

   if (flag == false) {
      flag = !flag;
   } else {
      throw new RuntimeException("             ");
   }
}

public static void main(String[] args) {

}
```

6.如何 择单例创建方式

- 如 不 延 加 单例，可以使 举 ， 对 举 好于 如 延 加 ，可以使 内 ， 对 内 好于 好使 尽

7.单例创建方式

（主要使用懒汉和懒汉式）

1. ： 初始化 ,会 即加 对 ， 天 安全,
2. ： 初始化 ,不会初始化 对 ， 使 候 会创 对 ,具备 加 功
3. 内 ： 合了 和 各 优 ， 对 候 会加 ，加 安全
4. 举单例: 使 举实 单例 优 :实 单 ， 举 就 单例， jvm从上 供保 ！ 免 反射和反序列化 ， 延 加
5. 双 (因为JVM 序 原因，可 会初始化多 ，不 使)

1. 汉式

1. ： 初始化 ,会 即加 对 ， 天 安全,

```
package com.lijie;

//
public class Demo1 {
```

```
    //          ,                       ,
  private static Demo1 demo1 = new Demo1();

    private Demo1() {
        System.out.println("    Demo1            ");
    }

    public static Demo1 getInstance() {
        return demo1;
    }

    public static void main(String[] args) {
        Demo1 s1 = Demo1.getInstance();
        Demo1 s2 = Demo1.getInstance();
        System.out.println(s1 == s2);
    }
}
```

2.懒汉式

1. : 初始化 ,不会初始化 对 , 使 候 会创 对 ,具备 加 功

```
package com.lijie;

//
public class Demo2 {

    //
  private static Demo2 demo2;

    private Demo2() {
        System.out.println("    Demo2            ");
    }

    public synchronized static Demo2 getInstance() {
        if (demo2 == null) {
            demo2 = new Demo2();
        }
        return demo2;
    }

    public static void main(String[] args) {
        Demo2 s1 = Demo2.getInstance();
        Demo2 s2 = Demo2.getInstance();
        System.out.println(s1 == s2);
    }
}
```

3. 内 类

1. 内 : 合了 和 各 优 , 对 候 会加 , 加 安全

```
package com.lijie;
```

```
//
public class Demo3 {

    private Demo3() {
        System.out.println("    Demo3            ");
    }

    public static class SingletonClassInstance {
        private static final Demo3 DEMO_3 = new Demo3();
    }

    //
  public static Demo3 getInstance() {
        return SingletonClassInstance.DEMO_3;
    }

    public static void main(String[] args) {
        Demo3 s1 = Demo3.getInstance();
        Demo3 s2 = Demo3.getInstance();
        System.out.println(s1 == s2);
    }
}
```

4.枚举单例式

1. 举单例: 使 举实 单例 优 :实 单 , 举 就 单例, jvm从上 供保 ! 免 反射和反序列化 , 延 加

```
package com.lijie;

//                    :                          jvm             !

public class Demo4 {

    public static Demo4 getInstance() {
        return Demo.INSTANCE.getInstance();
    }

    public static void main(String[] args) {
        Demo4 s1 = Demo4.getInstance();
        Demo4 s2 = Demo4.getInstance();
        System.out.println(s1 == s2);
    }

    //
  private static enum Demo {
      INSTANCE;
      //
  private Demo4 demo4;

      private Demo() {
         System.out.println("    Demo          ");
         demo4 = new Demo4();
      }
```

```
    public Demo4 getInstance() {
        return demo4;
    }
  }
}
```

5.双　检测　方式

1. 双　　(因为JVM　　序　原因，可　会初始化多　，不　　使　)

```
package com.lijie;

//
public class Demo5 {

   private static Demo5 demo5;

   private Demo5() {
      System.out.println("   Demo4           ");
   }

   public static Demo5 getInstance() {
      if (demo5 == null) {
         synchronized (Demo5.class) {
            if (demo5 == null) {
               demo5 = new Demo5();
            }
         }
      }
      return demo5;
   }

   public static void main(String[] args) {
      Demo5 s1 = Demo5.getInstance();
      Demo5 s2 = Demo5.getInstance();
      System.out.println(s1 == s2);
   }
}
```

6.工厂模式

1.什么是工厂模式

- 它 供了 创 对 佳 在工厂 中， 们在创 对 不会对客 创 ，并且 使 个共同 口 向 创 对 实 了创 和 分 ，工厂分为 单工厂 工厂 工厂

2.工厂模式好处

- 工厂 们 常 实例化对 了， 工厂 代 new 作
- 利 工厂 可以 低 序 . 合 ，为后 修 供了 大 便利
- 将 实 创 对 和 制 从 将 们 实 .

3.为什么要学习工厂设计模式

- 不 你们 到 ，你 Spring 吗，MyBatis 吗， 如 你学习 多 ， 你 己 发 己 ，就 先 （工厂 常 常广 ）

4.Spring开发中的工厂设计模式

1.Spring IOC

- Spring 就 ，在Spring IOC容器创 bean 使 了工厂
- Spring中 xml 创 bean，大 分 单工厂 创
- 容器 到了beanName和class 型后，动 反射创 具体 个对 ， 后将创 对 到Map中

2.为什么Spring IOC要使用工厂设计模式创建Bean

- 在实 发中，如 们A对 B，B C，C D 们 序 . 合 就会变 （. 合大 分为 与 之 依 ， 与 之 依 ）
- 在 久以前 三层 ， 制层 业务层，业务层 层 ，new对 ，. 合 大大 升，代 复 ，对 天

- 为了 免 况，Spring使 工厂 ，写 个工厂， 工厂创 Bean，以后 们如 对 就 工厂 就可以，剩下 事 不 们 了 Spring IOC容器 工厂中 个 Map 合， 为了 工厂 合单例 ，即 个对 只 产 ， 产出对 后就存入到 Map 合中，保 了实例不会 复 响 序

5.工厂模式分类

- 工厂 分为 单工厂 工厂 工厂

5.1 简单工厂模式

什么是简单工厂模式

- 单工厂 于 个工厂中 各 产品，创 在 个 中，客 具体产品 名 ，只 产品 对应 参 即可 但 工厂 ， 且 型 多 不利于 展

代码演示：

1. 创 工厂

```
package com.lijie;

public interface Car {
   public void run();
}
```

1. 创 工厂 产品（宝 ）

```
package com.lijie;

public class Bmw implements Car {
   public void run() {
      System.out.println("          ...");
   }
}
```

1. 创 工另外 产品（奥 ）

```
package com.lijie;

public class AoDi implements Car {
   public void run() {
      System.out.println("          ..");
   }
}
```

1. 创　　工厂　，　他决定具体　　哪产品

```
package com.lijie;

public class CarFactory {

    public static Car createCar(String name) {
      if ("".equals(name)) {
            return null;
      }
      if(name.equals("   ")){
         return new AoDi();
      }
      if(name.equals("   ")){
         return new Bmw();
      }
      return null;
   }
}
```

1.　　创　工厂　具体实例

```
package com.lijie;

public class Client01 {

   public static void main(String[] args) {
      Car aodi  =CarFactory.createCar("   ");
      Car bmw  =CarFactory.createCar("   ");
      aodi.run();
      bmw.run();
   }
}
```

单工厂的优　/缺

- 优　：　单工厂　　够　外　定　信　，决定　应　创　哪个具体　对　　区分了　各　　和　力，　利于　个　件体　　优化
- 　　：　　工厂　中了　实例　创　　，容　反GRASPR　内　　任分　原则

5.2 工厂方法模式

什么是工厂方法模式

- 工厂　　Factory Method，又　多　工厂　　在工厂　　中，　　工厂　不再　　产品　创　，　将具体创　工作交　子　去做　　为　个　工厂　，仅　　出具体工厂子　　实　　口，　不　哪　个产品　应　实例化

代码演示：

1. 创　工厂

```
package com.lijie;

public interface Car {
   public void run();
}
```

1. 创 工厂   口 (产品   new出   他 实   )

```
package com.lijie;

public interface CarFactory {

   Car createCar();

}
```

1. 创 工厂 产品 (奥)

```
package com.lijie;

public class AoDi implements Car {
   public void run() {
      System.out.println("          ..");
   }
}
```

1. 创 工厂另外   产品 (宝)

```
package com.lijie;

public class Bmw implements Car {
   public void run() {
      System.out.println("          ...");
   }
}
```

1. 创 工厂   口 实例 (奥)

```
package com.lijie;

public class AoDiFactory implements CarFactory {

   public Car createCar() {

      return new AoDi();
   }
}
```

1. 创 工厂   口 实例 (宝)

```
package com.lijie;

public class BmwFactory implements CarFactory {

   public Car createCar() {

      return new Bmw();
   }

}
```

1. 创 工厂 具体实例

```
package com.lijie;

public class Client {

   public static void main(String[] args) {
      Car aodi = new AoDiFactory().createCar();
      Car jili = new BmwFactory().createCar();
      aodi.run();
      jili.run();
   }
}
```

5.3 抽象工厂模式

什么是抽象工厂模式

- 工厂 单地 工厂 工厂， 工厂可以创 具体工厂， 具体工厂 产 具体产品

入图

代码演示：

1. 创 个子工厂，及实

```
package com.lijie;

//
public interface Car {
      void run();
}

 class CarA implements Car{

   public void run() {
      System.out.println("   ");
   }

}
 class CarB implements Car{

   public void run() {
      System.out.println("   ");
   }
```

```
}
```

1. 创　　二个子工厂，及实

```
package com.lijie;

//
public interface Engine {

    void run();

}

class EngineA implements Engine {

    public void run() {
        System.out.println("     !");
    }

}

class EngineB implements Engine {

    public void run() {
        System.out.println("     !");
    }

}
```

1. 创　　个　工厂，及实　　（　　工厂　实　　决定　　个工厂　　个实例）

```
package com.lijie;

public interface TotalFactory {
   //
  Car createChair();
   //
  Engine createEngine();
}

//
class TotalFactoryReally implements TotalFactory {

   public Engine createEngine() {

      return new EngineA();
   }

   public Car createChair() {

      return new CarA();
   }
}
```

1. 中

代理模式

什么是代理模

代 制对 ，可以在 个对

代码演示：

- 代 ：（如何 在不修 UserDao 口 况下 事务和关 事务 ）

```
package com.lijie;

//
public class UserDao{
   public void save() {
      System.out.println("          ");
   }
}
```

```
package com.lijie;

//
public  class Test{
   public static void main(String[] args) {
      UserDao userDao = new UserDao();
      userDao.save();
   }
}
```

修改代码，添加代理类

```
package com.lijie;

//
public class UserDaoProxy extends UserDao {
   private UserDao userDao;

   public UserDaoProxy(UserDao userDao) {
      this.userDao = userDao;
   }

   public void save() {
      System.out.println("       ...");
      userDao.save();
      System.out.println("       ...");
   }

}
```

```
//
public class Test{
   public static void main(String[] args) {
      UserDao userDao = new UserDao();
      UserDaoProxy userDaoProxy = new UserDaoProxy(userDao);
      userDaoProxy.save();
   }
}
```

- : 个 代 对 己 复 写代 , 不 ,
- 优 : 但 可以 实 对 口 实 代

2.2.动 代理

什么是动 代理

- 动 代 也叫做，JDK代 口代
- 动 代 对 , 利 JDK API，动 在内存中 代 对 (代 口 动 代 class 件，并加), 就叫动 代

```
package com.lijie;

//
public interface UserDao {
    void save();
}
```

```
package com.lijie;

//
public class UserDaoImpl implements UserDao {
   public void save() {
      System.out.println("          ");
   }
}
```

- //下 代 ，可 复使 ，不像 代 己 复 写代

```
package com.lijie;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

//                    ,     InvocationHandler
public class InvocationHandlerImpl implements InvocationHandler {

   //
  private Object target;

    //
  public InvocationHandlerImpl(Object target) {
        this.target = target;
    }

    //
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("          ");
        //   invoke()

  Object result = method.invoke(target, args);
        System.out.println("          ");
        return result;
```

```
    }
}
```

- //利 动 代 使 代

```
package com.lijie;

import java.lang.reflect.Proxy;

public class Test {
    public static void main(String[] args) {
        //
  UserDao userDaoImpl = new UserDaoImpl();
        InvocationHandlerImpl invocationHandlerImpl = new
InvocationHandlerImpl(userDaoImpl);

        //
  ClassLoader loader = userDaoImpl.getClass().getClassLoader();
        Class<?>[] interfaces = userDaoImpl.getClass().getInterfaces();

        //
  UserDao newProxyInstance = (UserDao) Proxy.newProxyInstance(loader,
interfaces, invocationHandlerImpl);
        newProxyInstance.save();
    }
}
```

- : 向 口, 业务 实 口
- 优 : 不 关 代 , 只 在 定代 哪 个对

5.3.CGLIB动 代理

CGLIB动 代理原理:

- 利 asm 包, 对代 对 class 件加 , 修 其字 子 处

什么是CGLIB动 代理

- CGLIB动 代 和jdk代 , 使 反射完 代 , 不同 他可以 代 (jdk动 代 不 , 他 业务 实 口) , CGLIB动 代 底层使 字 , CGLIB动 代 不 对 final (CGLIB动 代 导入jar包)

代码演示:

```
package com.lijie;

//
public interface UserDao {
    void save();
}
```

```
package com.lijie;

//
public class UserDaoImpl implements UserDao {
   public void save() {
      System.out.println("          ");
   }
}
```

```
package com.lijie;

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

//
public class CglibProxy implements MethodInterceptor {
   private Object targetObject;
   //              Object
  public Object getInstance(Object target) {
      //
  this.targetObject = target;
      Enhancer enhancer = new Enhancer();
      enhancer.setSuperclass(target.getClass());
      enhancer.setCallback(this);
      return enhancer.create();
   }

   //
  public Object intercept(Object obj, Method method, Object[] args, MethodProxy
proxy) throws Throwable {
      System.out.println("       ");
      Object result = proxy.invoke(targetObject, args);
      System.out.println("       ");
      //
  return result;
   }
}
```

```
package com.lijie;

//   CGLIB
public class Test {
    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
        UserDao userDao = (UserDao) cglibProxy.getInstance(new UserDaoImpl());
        userDao.save();
    }
}
```

8.建　者模式

1.什么是建　者模式

- 　　：　将　个复　　对　　与它　　分　，使　同　　可以创　不同　　创
- 工厂　　供　　创　单个　　产品
- 　　则　将各　产品　中　　，　　具　不同　属　　产品

建　者模式　常包括下　几个角色：

1. uilder：　出　个　　口，以　　产品对　　各个　　分　　个　口　定　实　复　对　　哪些　分　创　，并不　及具体　对　　件　创
2. ConcreteBuilder：实　Builder　口，　对不同　商业　　，具体化复　对　　各　分　创　在　　完　后，　供产品　实例
3. Director：　　具体　　创　复　对　　各个　分，在　导　中不　及具体产品　信　，只　　保　对　各　分完　创　　序创
4. Product：　创　　复　对

2.建　者模式的使用场景

使用场景：

1. 　　对　具　复　　内
2. 　　对　内　属　　互依

- 与工厂　　区别　：　　加关　与　件　　序
- JAVA 中　StringBuilder就　　创　　，他　　个单个字　　char　　合
- Spring不　　，它　供　　作应　　对于字　串　　些　作，　不　创　　变　个　字　串

3.代码案例

1. 　　个　备对　Arms

```
package com.lijie;

//
public class Arms {
   //
  private String helmet;
   //
  private String armor;
   //
  private String weapon;

   //   Git  Set   ...........
}
```

1. 创　Builder　口（　出　个　　口，以　　产品对　　各个　　分　　，　个　口只　　）

```
package com.lijie;

public interface PersonBuilder {

   void builderHelmetMurder();
```

```
    void builderArmorMurder();

    void builderWeaponMurder();

    void builderHelmetYanLong();

    void builderArmorYanLong();

    void builderWeaponYanLong();

    Arms BuilderArms(); //
}
```

1. 创 Builder实   （ 个 主 实 复 对 创   哪些 分   什么属 ）

```
package com.lijie;

public class ArmsBuilder implements PersonBuilder {
    private Arms arms;

    //       Arms    ,       set
  public ArmsBuilder() {
        arms = new Arms();
    }

    public void builderHelmetMurder() {
        arms.setHelmet("       ");
    }

    public void builderArmorMurder() {
        arms.setArmor("       ");
    }

    public void builderWeaponMurder() {
        arms.setWeapon("       ");
    }

    public void builderHelmetYanLong() {
        arms.setHelmet("       ");
    }

    public void builderArmorYanLong() {
        arms.setArmor("       ");
    }

    public void builderWeaponYanLong() {
        arms.setWeapon("       ");
    }

    public Arms BuilderArms() {
        return arms;
    }
}
```

1. Director（ 具体 创 复 对 各个 分，在 导 中不 及具体产品 信 ，只 保 对 各 分完 创 序创 ）

```
package com.lijie;

public class PersonDirector {

   //
  public Arms constructPerson(PersonBuilder pb) {
      pb.builderHelmetYanLong();
      pb.builderArmorMurder();
      pb.builderWeaponMurder();
      return pb.BuilderArms();
   }

   //
  public static void main(String[] args) {
      PersonDirector pb = new PersonDirector();
      Arms arms = pb.constructPerson(new ArmsBuilder());
      System.out.println(arms.getHelmet());
      System.out.println(arms.getArmor());
      System.out.println(arms.getWeapon());
   }
}
```

9.模板方法模式

1.什么是模板方法

- ：定义 个 作中 （ ），将 些 延 到子 中 使 子 可以不 变 个 定义

2.什么时候使用模板方法

- 实 些 作 ， 体 固定，但 就 其中 小 分 变， 候可以使 ，将容 变 分 出 ，供子 实

3.实 开发中应用场景哪 用到了模板方法

- 其实 多 中 到了
- 例如： 库 封 Junit单元 servlet中关于doGet/doPost

4.现实生活中的模板方法

例如：

1. 去 厅吃 ， 厅 们 供了 个 就 ： 单， ，吃 ，付 ， 人（ “ **菜和付款**” 不 定 子 完 ，其他 则 个 ）

5.代码实现模板方法模式

1. 先定义 个 中 和付 ， 子 实

```
package com.lijie;
```

```
//
public abstract class RestaurantTemplate {

   // 1.
  public void menu() {
      System.out.println("     ");
   }

   // 2.
  abstract void spotMenu();

   // 3.
  public void havingDinner(){ System.out.println("    "); }

   // 3.
  abstract void payment();

   // 3.
  public void GoR() { System.out.println("    "); }

   //
  public void process(){
      menu();
      spotMenu();
      havingDinner();
      payment();
      GoR();
   }
}
```

1. 具体　　子　1

```
package com.lijie;

public class RestaurantGinsengImpl extends RestaurantTemplate {

    void spotMenu() {
        System.out.println("    ");
    }

    void payment() {
        System.out.println("5  ");
    }
}
```

1. 具体　　子　2

```
package com.lijie;

public class RestaurantLobsterImpl  extends RestaurantTemplate  {

    void spotMenu() {
        System.out.println("    ");
    }

    void payment() {
        System.out.println("50  ");
    }
}
```

1. 客

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //
  RestaurantTemplate restaurantTemplate = new RestaurantGinsengImpl();
        restaurantTemplate.process();
    }
}
```

10.外观模式

1.什么是外观模式

- 外　　：也叫　　，　　复　　，并向客　　供了　个客　　可以　　口
- 它向　　加　个　口，　　个　口　　实　　复
- 使　外　　，他外　　就　　个　口，其实他　内　　多复　　口已　　实

2.外观模式例子

- 　　册完之后，　　信　口　　件　口　　信　　口

1. 创　　信　口

```
package com.lijie;

//
public interface AliSmsService {
   void sendSms();
}
```

```
package com.lijie;

public class AliSmsServiceImpl implements AliSmsService {

    public void sendSms() {
        System.out.println("          ");
    }

}
```

1. 创　　件　口

```
package com.lijie;

//
public interface EamilSmsService {
   void sendSms();
}
```

```
package com.lijie;

public class EamilSmsServiceImpl implements   EamilSmsService{
   public void sendSms() {
      System.out.println("          ");
   }
}
```

1. 创　　信　　口

```
package com.lijie;

//
public interface WeiXinSmsService {
   void sendSms();
}
```

```
package com.lijie;

public class WeiXinSmsServiceImpl implements  WeiXinSmsService {
    public void sendSms() {
        System.out.println("             ");
    }
}
```

1. 创　　(　　单使　，复　东　以及　　封　好了)

```
package com.lijie;

public class Computer {
```

```
    AliSmsService aliSmsService;
    EamilSmsService eamilSmsService;
    WeiXinSmsService weiXinSmsService;

    public Computer() {
        aliSmsService = new AliSmsServiceImpl();
        eamilSmsService = new EamilSmsServiceImpl();
        weiXinSmsService = new WeiXinSmsServiceImpl();
    }

    //
   public void sendMsg() {
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();
    }
}
```

1. 启动

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //
   AliSmsService aliSmsService = new AliSmsServiceImpl();
        EamilSmsService eamilSmsService = new EamilSmsServiceImpl();
        WeiXinSmsService weiXinSmsService = new WeiXinSmsServiceImpl();
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();

        //
   new Computer().sendMsg();
    }
}
```

11.原型模式

1.什么是原型模式

- 原型　　单　就 克
- 原型　了　个　实例，　个原型 可定制　原型　多 于创 复　　实例，因为　　况下，复制 个已 存在 实例可使 序

2.原型模式的应用场景

1. 初始化　　化 常多　　，　个　包　　件　　们就可以　原型　　免 些
2. 　new产　　个对　　常　　准备　　，　　可以使 原型
3. 个对　　供 其他对　　，　且各个　　可　　修 其值 ，可以　使 原型　　多个对 供　　使 ，即保

```
Spring
```

3.原型模式的使用方式

1. 实 Cloneable 口 在java 个Cloneable 口，它 作 只 个，就 在
可以安全地在实 了 口 上使 clone 在java

```
        user.setName("    ");
        user.setPassword("123456");
        ArrayList<String> phones = new ArrayList<>();
        phones.add("17674553302");
        user.setPhones(phones);

        //copy    user    ,
  User user2 = user.clone();
        user2.setPassword("654321");

        //
  System.out.println(user == user2);

        //
  System.out.println(user.getName() + " | " + user2.getName());
        System.out.println(user.getPassword() + " | " + user2.getPassword());
        //
  System.out.println(user.getPhones() == user2.getPhones());
    }
}
```

1. 如 不   复制,   删 User 中

```
//
user.phones = (ArrayList<String>) this.phones.clone();
```

12.策略模式

1.什么是策略模式

- 定义了   列   同 义   作,并将   个   作封   , 且 使它们 可以 互   (其实   Java中   常 常广)
-       主   为了 化 if...else 带   复 和 以

2.策略模式应用场景

-                对   ,将   个   封 到具 共同 口   中, 从 使 它们之 可以 互

1. 例如: 做 个不同会员   力度不同 三   ,初 会员,中 会员,   会员(三 不同)
2. 例如: 个 付 块,   信 付   付宝 付   付

3.策略模式的优 和缺

- 优 : 1   可以   切   2   免使 多   件判   3   展   常 好
-     : 1   会增多 2   对外

4.代码演示

-       付 块   信 付   付宝 付   付

1. 定义   公共

```
package com.lijie;

//
abstract class PayStrategy {

   //
  abstract void algorithmInterface();

}
```

1. 定义实　信　付

```
package com.lijie;

class PayStrategyA extends PayStrategy {

   void algorithmInterface() {
      System.out.println("       ");
   }
}
```

1. 定义实　付宝　付

```
package com.lijie;

class PayStrategyB extends PayStrategy {

   void algorithmInterface() {
      System.out.println("         ");
   }
}
```

1. 定义实　　付

```
package com.lijie;

class PayStrategyC extends PayStrategy {

   void algorithmInterface() {
      System.out.println("       ");
   }
}
```

1. 定义下

```
package com.lijie;//

class Context {

   PayStrategy strategy;
```

```
   public Context(PayStrategy strategy) {
      this.strategy = strategy;
   }

   public void algorithmInterface() {
      strategy.algorithmInterface();
   }

}
```

1.

```
package com.lijie;

class ClientTestStrategy {
   public static void main(String[] args) {
      Context context;
      //          A
      context = new Context(new PayStrategyA());
      context.algorithmInterface();
      //          B
      context = new Context(new PayStrategyB());
      context.algorithmInterface();
      //          C
      context = new Context(new PayStrategyC());
      context.algorithmInterface();
   }
}
```

13.观察者模式

1.什么是观察者模式

- 先 什么 为 型， 为型 关 中对 之 互交互， 决 在 对 之 互 信和协作， 对
- 察 ， 为 型，又叫发布- ，他定义对 之 对多 依 关 ， 使 个对 变 ，则 依 于它 对 会 到 并 动

2.模式的职责

- 察 主 于1对N 个对 变化 ，他 及 告 列对 ，令他们做出 应

实现有两种方式：

1. ： 会 以广 发 察 ， 察 只 动
2. ： 察 只 况即可， 于什么 候 取内容， 取什么内容， 可以 主决定

3.观察者模式应用场景

1. 关 为场 ， ，关 为 可 分 ， 不 “ 合”关 事件多 发场
2. 交 场 ，如 列 事件 处 制

4.代码实现观察者模式

1. 定义　　察　，　个实　　口　实　　具体　察

```
package com.lijie;

//
public interface Observer {
    //
  void update(int state);
}
```

1. 定义具体　察

```
package com.lijie;

//
public class ObserverImpl implements Observer {

    //
  private int myState;

    public void update(int state) {
        myState=state;
        System.out.println("       ,myState       "+state);
    }

    public int getMyState() {
        return myState;
    }
}
```

1. 定义主　　主　定义　察　　，并实　增　删及　　作

```
package com.lijie;

import java.util.Vector;

//
public class Subjecct {
   //                    ArrayList
  private Vector<Observer> list = new Vector<>();

   //
  public void registerObserver(Observer obs) {
      list.add(obs);
   }
    //
  public void removeObserver(Observer obs) {
      list.remove(obs);
   }

   //
  public void notifyAllObserver(int state) {
```

```
      for (Observer observer : list) {
         observer.update(state);
      }
   }
}
```

1. 定义具体 ，他 Subject ，在 实 具体业务，在具体 中， 会 多

```
package com.lijie;

//
public class RealObserver extends Subjecct {
    //
  private int state;
    public int getState(){
       return state;
    }
    public void  setState(int state){
       this.state=state;
       //       (       )
  this.notifyAllObserver(state);
    }
}
```

1.

```
package com.lijie;

public class Client {

   public static void main(String[] args) {
      //
  RealObserver subject = new RealObserver();
      //
  ObserverImpl obs1 = new ObserverImpl();
      ObserverImpl obs2 = new ObserverImpl();
      ObserverImpl obs3 = new ObserverImpl();
      //
  subject.registerObserver(obs1);
      subject.registerObserver(obs2);
      subject.registerObserver(obs3);
      //    State
  subject.setState(300);
      System.out.println("obs1      MyState        "+obs1.getMyState());
      System.out.println("obs2      MyState        "+obs2.getMyState());
      System.out.println("obs3      MyState        "+obs3.getMyState());
      //    State
  subject.setState(400);
      System.out.println("obs1      MyState        "+obs1.getMyState());
      System.out.println("obs2      MyState        "+obs2.getMyState());
      System.out.println("obs3      MyState        "+obs3.getMyState());
   }
}
```

14.文章就到　了，没　，没了

察　　public void removeObserver(Observer obs) { list.remove(obs); }

```
//
public void notifyAllObserver(int state) {
   for (Observer observer : list) {
      observer.update(state);
   }
}
```

}

```
4\.                   Subject
```java
package com.lijie;

//
public class RealObserver extends Subjecct {
    //
  private int state;
    public int getState(){
       return state;
    }
    public void  setState(int state){
       this.state=state;
       //       (       )
  this.notifyAllObserver(state);
    }
}
```

1.

```
package com.lijie;

public class Client {

   public static void main(String[] args) {
      //
  RealObserver subject = new RealObserver();
      //
  ObserverImpl obs1 = new ObserverImpl();
      ObserverImpl obs2 = new ObserverImpl();
      ObserverImpl obs3 = new ObserverImpl();
      //
  subject.registerObserver(obs1);
      subject.registerObserver(obs2);
      subject.registerObserver(obs3);
      //    State
  subject.setState(300);
      System.out.println("obs1      MyState        "+obs1.getMyState());
      System.out.println("obs2      MyState        "+obs2.getMyState());
      System.out.println("obs3      MyState        "+obs3.getMyState());
```
```

```
        //    State
    subject.setState(400);
        System.out.println("obs1       MyState        "+obs1.getMyState());
        System.out.println("obs2       MyState        "+obs2.getMyState());
        System.out.println("obs3       MyState        "+obs3.getMyState());
     }
}
```