

宜宾学院

软件设计模式与体系结构设计报告书

学 院: 人工智能与大数据学部 班 级: 2018 级 9 班

学生姓名: 杨雪 学 号: 200109327

设计地点 (单位) 5305

设计题目: 软件设计模式与体系结构实验 3

完成日期: 2021 年 4 月 10 日

软件设计模式与体系结构实验 3

一、 为以下三个图，写出简单工厂，工厂方法，抽象工厂的实际代码

1、简单工厂

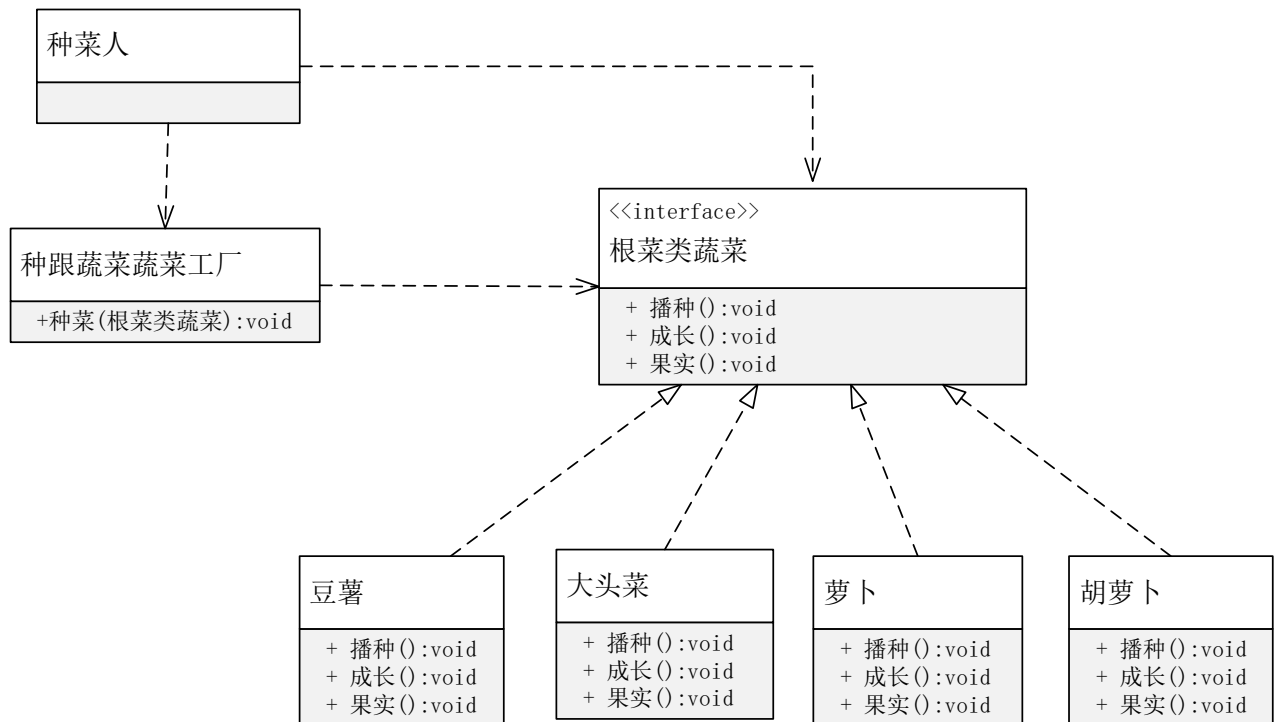


图 1 简单工厂 UML 图

代码框架：

```
/**
 * 任务一：为以下三个图，写出简单工厂，工厂方法，抽象工厂的实际代码
 * 1.简单工厂
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 */

public class Question1 {
    public static void main(String[] args) {
        RootVegetablesfactory rootVegetablesFactory = new
RootVegetablesfactory();
        RootVegetables potato = rootVegetablesFactory.getVegetable("potato");
        potato.sow();
    }
}
```

```

        potato.growUp();
        potato.fruit();
        System.out.println("");
        RootVegetables turnip = rootVegetablesFactory.getVegetable("turnip");
        turnip.sow();
        turnip.growUp();
        turnip.fruit();
        System.out.println("");
        RootVegetables carrot = rootVegetablesFactory.getVegetable("carrot");
        carrot.sow();
        carrot.growUp();
        carrot.fruit();
        System.out.println("");
        RootVegetables radish = rootVegetablesFactory.getVegetable("radish");
        radish.sow();
        radish.growUp();
        radish.fruit();
        System.out.println("");
    }
}
//根菜类蔬菜
interface RootVegetables{
    // 播种
    void sow();
    // 成长
    void growUp();
    // 果实
    void fruit();
}

//豆薯
class BeanPotato implements RootVegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了豆薯");
    }
    @Override
    public void growUp() {
        System.out.println("豆薯正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("豆薯已经长大了，已经有果实啦！！");
    }
}

```

```

}
//大头菜
class ATurnip implements RootVegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了大头菜");
    }
    @Override
    public void growUp() {
        System.out.println("大头菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("大头菜已经长大了，已经有果实啦！！");
    }
}
//胡萝卜
class Carrot implements RootVegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了胡萝卜");
    }
    @Override
    public void growUp() {
        System.out.println("胡萝卜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("胡萝卜已经长大了，已经有果实啦！！");
    }
}
//萝卜
class Radish implements RootVegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了萝卜");
    }
    @Override
    public void growUp() {
        System.out.println("萝卜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("萝卜已经长大了，已经有果实啦！！");
    }
}

```

```

    }
}
//种根菜类蔬菜工厂
class RootVegetablesfactory {
    public RootVegetables getVegetable(String type) {
        if(type.equalsIgnoreCase("potato")) {
            return new BeanPotato ();
        }
        if(type.equalsIgnoreCase("turnip")) {
            return new ATurnip();
        }
        if(type.equalsIgnoreCase("carrot")) {
            return new Carrot();
        }
        if(type.equalsIgnoreCase("radish")) {
            return new Radish();
        }
        return null;
    }
}

```

测试截图：

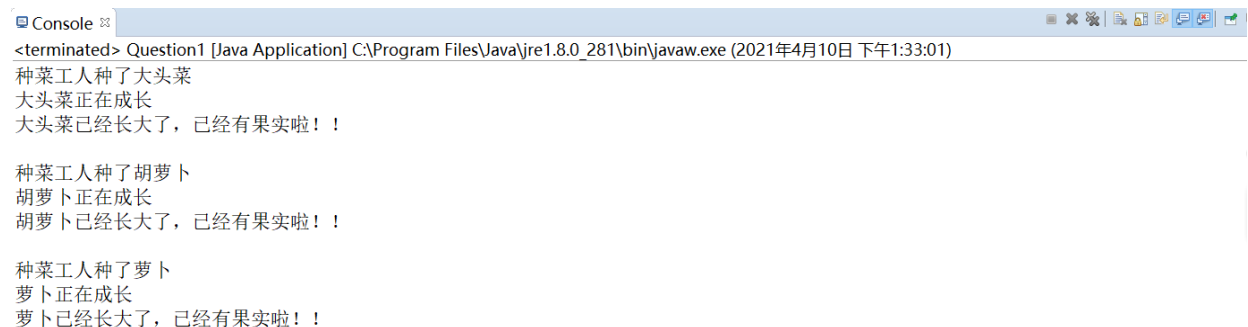


图 2 简单工厂截图

2、工厂方法

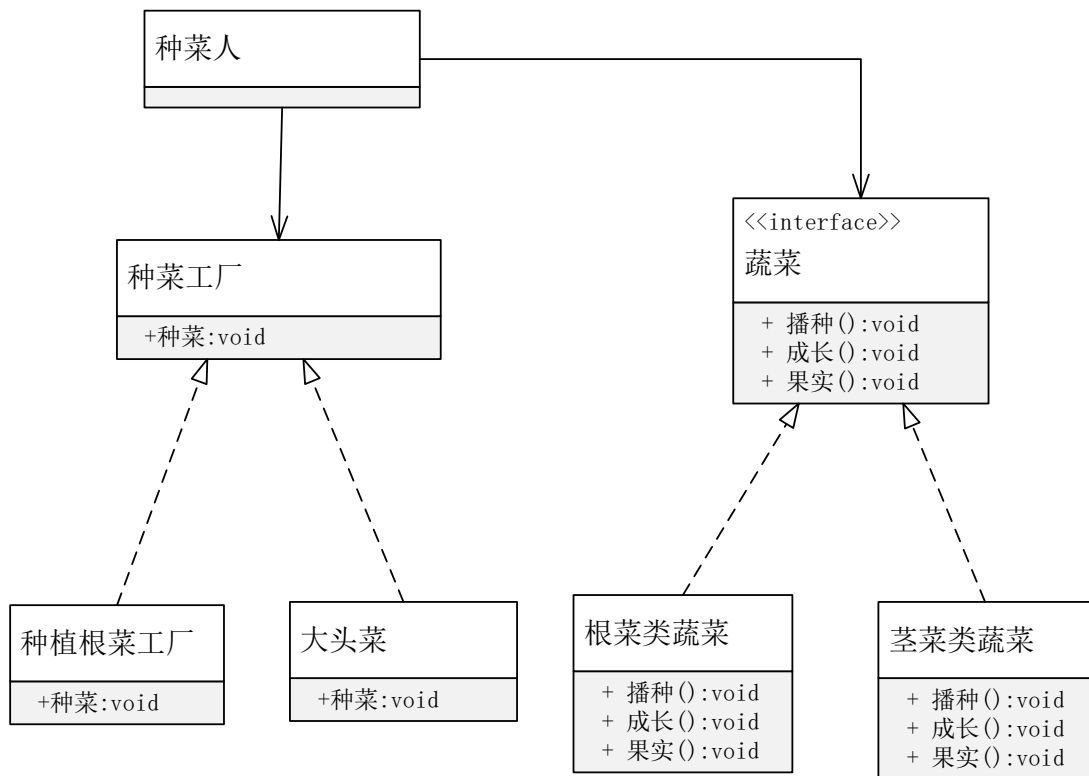


图 3 工厂方法 UML 图

代码框架:

```

/**
 * 任务一: 为以下三个图, 写出简单工厂, 工厂方法, 抽象工厂的实际代码
 *      2.工厂方法
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 */

public class Question2 {
    public static void main(String[] args) {
        PlantRootVegetableFactory plantRootVegetableFactory = new
PlantRootVegetableFactory();
        Vegetables plantRootVegetable =
plantRootVegetableFactory.growingVegetables();
        plantRootVegetable.sow();
        plantRootVegetable.growUp();
        plantRootVegetable.fruit();
        System.out.println("");
        PlantStemVegetableFactory plantStemVegetableFactory = new
PlantStemVegetableFactory();
        Vegetables plantStemVegetable =
plantStemVegetableFactory.growingVegetables();
    }
}
  
```

```

        plantStemVegetable.sow();
        plantStemVegetable.growUp();
        plantStemVegetable.fruit();
    }
}
//根菜类蔬菜
interface Vegetables{
    // 播种
    void sow();
    // 成长
    void growUp();
    // 果实
    void fruit();
}

//根菜类蔬菜
class RootVegetable implements Vegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了根菜类蔬菜");
    }
    @Override
    public void growUp() {
        System.out.println("根菜类蔬菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("根菜类蔬菜已经长大了，已经有果实啦！！");
    }
}

//茎菜类蔬菜
class StemVegetables implements Vegetables{
    @Override
    public void sow() {
        System.out.println("种菜工人种了茎菜类蔬菜");
    }
    @Override
    public void growUp() {
        System.out.println("茎菜类蔬菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("茎菜类蔬菜已经长大了，已经有果实啦！！");
    }
}

```

```

}

//蔬菜工厂
interface Vegetablesfactory {
    Vegetables growingVegetables();
}
//种植根菜工厂
class PlantRootVegetableFactory implements Vegetablesfactory{
    @Override
    public Vegetables growingVegetables() {
        return new RootVegetable();
    }
}
//种植茎菜工厂
class PlantStemVegetableFactory implements Vegetablesfactory{
    @Override
    public Vegetables growingVegetables() {
        return new StemVegetables();
    }
}

```

测试截图：

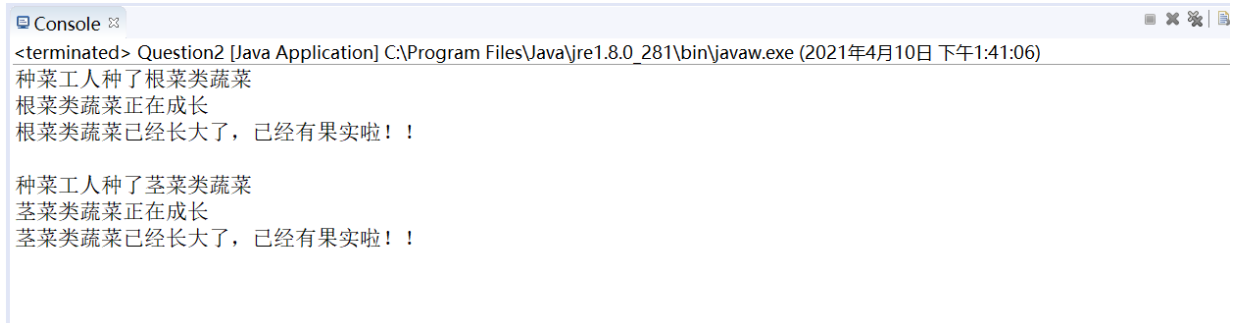


图 4 工厂方法截图

3、抽象工厂

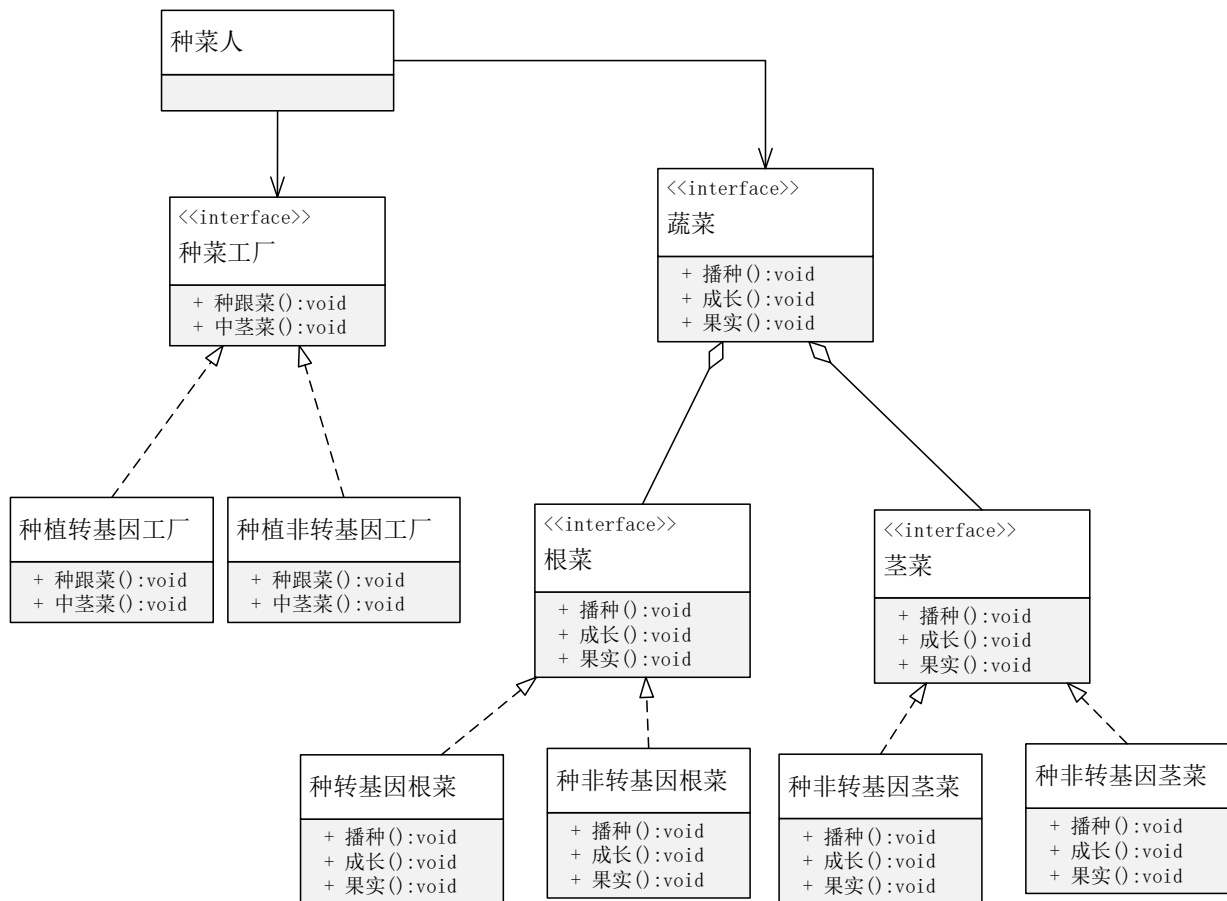


图 5 抽象工厂 UML 图

代码框架:

```

/**
 * 任务一: 为以下三个图, 写出简单工厂, 工厂方法, 抽象工厂的实际代码
 * 3.抽象工厂
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 */

public class Question3 {
    public static void main(String[] args) {
        Vegetablesfactory1 plantTransgeniPlants = new PlantTransgeniPlants();
        Vegetablesfactory1 plantNoTransgeniPlants = new
PlantNoTransgeniPlants();

        Vegetables1 transgeneRootVegetables =
plantTransgeniPlants.growingRootVegetables();
        Vegetables1 transgeneStemVegetables =
plantTransgeniPlants.growingStemVegetables();
    }
}
  
```

```

        Vegetables1 noTransgeneRootVegetables =
plantNoTransgeniPlants.growingRootVegetables();
        Vegetables1 noTransgeneStemVegetables =
plantNoTransgeniPlants.growingStemVegetables();

        transgeneRootVegetables.sow();
        transgeneRootVegetables.growUp();
        transgeneRootVegetables.fruit();
        System.out.println();
        transgeneStemVegetables.sow();
        transgeneStemVegetables.growUp();
        transgeneStemVegetables.fruit();
        System.out.println();
        noTransgeneRootVegetables.sow();
        noTransgeneRootVegetables.growUp();
        noTransgeneRootVegetables.fruit();
        System.out.println();
        noTransgeneStemVegetables.sow();
        noTransgeneStemVegetables.growUp();
        noTransgeneStemVegetables.fruit();
    }
}
//根菜类蔬菜
interface Vegetables1{
    // 播种
    void sow();
    // 成长
    void growUp();
    // 果实
    void fruit();
}

//根菜
interface RootVegetable1 extends Vegetables1{}
//茎菜
interface StemVegetables1 extends Vegetables1{}

//转基因根菜
class TransgeneRootVegetables implements RootVegetable1{
    @Override
    public void sow() {
        System.out.println("种菜工人种了转基因根菜");
    }
    @Override

```

```

    public void growUp() {
        System.out.println("转基因根菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("转基因根菜已经长大了，已经有果实啦！！");
    }
}
//非转基因根菜
class NoTransgeneRootVegetables implements RootVegetable1{
    @Override
    public void sow() {
        System.out.println("种菜工人种了非转基因根菜");
    }
    @Override
    public void growUp() {
        System.out.println("非转基因根菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("非转基因根菜已经长大了，已经有果实啦！！");
    }
}
//转基因茎菜
class TransgeneStemVegetables implements RootVegetable1{
    @Override
    public void sow() {
        System.out.println("种菜工人种了转基因茎菜");
    }
    @Override
    public void growUp() {
        System.out.println("转基因茎菜正在成长");
    }
    @Override
    public void fruit() {
        System.out.println("转基因茎菜已经长大了，已经有果实啦！！");
    }
}
//非转基因茎菜
class NoTransgeneStemVegetables implements RootVegetable1{
    @Override
    public void sow() {
        System.out.println("种菜工人种了非转基因茎菜");
    }
}

```

```

@Override
public void growUp() {
    System.out.println("非转基因茎菜正在成长");
}
@Override
public void fruit() {
    System.out.println("非转基因茎菜已经长大了，已经有果实啦！！");
}
}

```

//蔬菜工厂

```

interface Vegetablesfactory1{
    Vegetables1 growingRootVegetables();
    Vegetables1 growingStemVegetables();
}

```

//种植转基因工厂

```

class PlantTransgeniPlants implements Vegetablesfactory1{
    @Override
    public Vegetables1 growingRootVegetables() {
        return new TransgeneRootVegetables();
    }

    @Override
    public Vegetables1 growingStemVegetables() {
        return new TransgeneStemVegetables();
    }
}

```

//种植非转基因工厂

```

class PlantNoTransgeniPlants implements Vegetablesfactory1{
    @Override
    public Vegetables1 growingRootVegetables() {
        return new NoTransgeneRootVegetables();
    }

    @Override
    public Vegetables1 growingStemVegetables() {
        return new NoTransgeneStemVegetables();
    }
}

```

测试截图：

```
Console
<terminated> Question3 (1) [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (2021年4月10日 下午1:50:42)

种菜工人种了转基因茎菜
转基因茎菜正在成长
转基因茎菜已经长大了，已经有果实啦！！

种菜工人种了非转基因根菜
非转基因根菜正在成长
非转基因根菜已经长大了，已经有果实啦！！

种菜工人种了非转基因茎菜
非转基因茎菜正在成长
非转基因茎菜已经长大了，已经有果实啦！！
```

图 6 抽象工厂截图

二、完成书上 P70 页 第 4 题，提供系统的 UML 图和代码框架

UML 图：

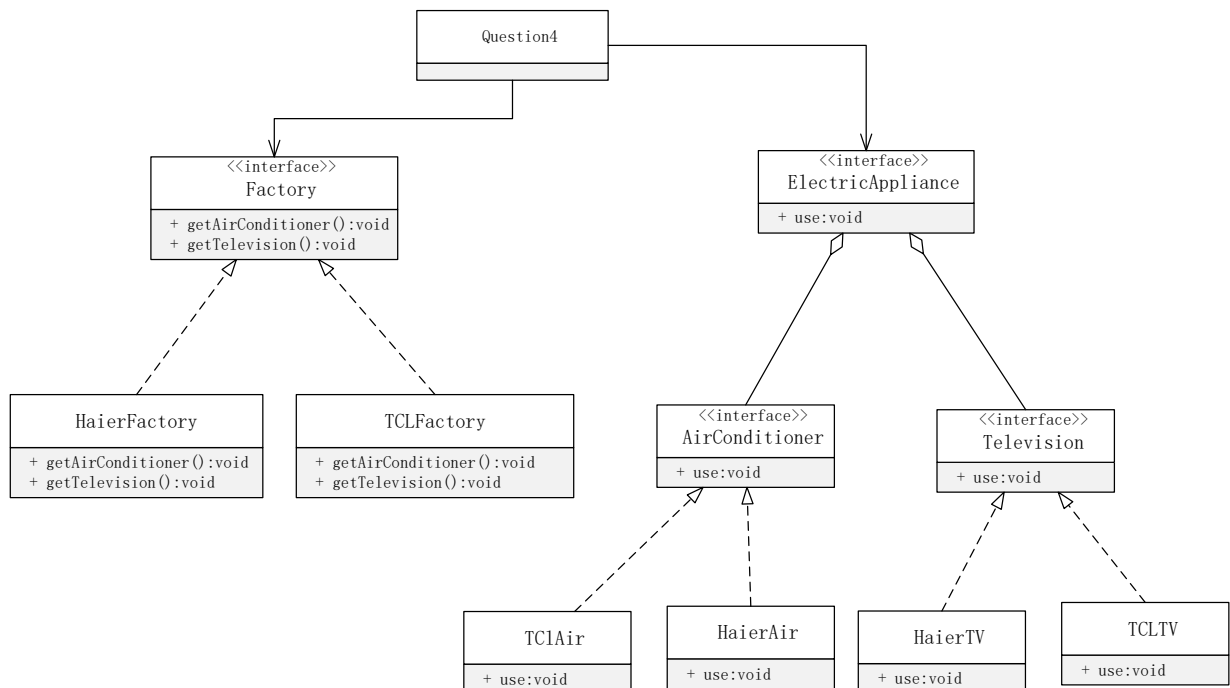


图 7 P70 的 4 题目 UML 图

代码框架：

```
/**
 * 完成书上P70页 第4题，提供系统的UML图和代码框架
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 */
public class Question4 {
    public static void main(String[] args) {
```

```

        System.out.println("Haier工厂测试");
        //创建Haier工厂对象
        Factory haierFactort = new HaierFactory();
        //通过Haier工厂创建Haier空调
        AirConditioner haierAir = haierFactort.getAirConditioner();
        haierAir.use();
        //通过Haier工厂创建Haier电视
        Television haierTV = haierFactort.getTelevision();
        haierTV.use();
        System.out.println();
        System.out.println("TCL工厂测试");
        //创建TCL工厂对象
        Factory tclFactort = new TCLFactory();
        //通过TCL工厂创建TCL空调
        AirConditioner tclAir = tclFactort.getAirConditioner();
        tclAir.use();
        //通过TCL工厂创建TCL电视
        Television tclTV = tclFactort.getTelevision();
        tclTV.use();

    }
}
//电器
interface ElectricAppliance {
    void use();
}
//空调
interface AirConditioner extends ElectricAppliance {}
//电视剧
interface Television extends ElectricAppliance {}
//Haier空调
class HaierAir implements AirConditioner {
    @Override
    public void use() {
        System.out.println("海尔空调正在使用...");
    }
}
//TCL空调
class TCLAir implements AirConditioner {
    @Override
    public void use() {
        System.out.println("TCL空调正在使用...");
    }
}
}

```

```

//Haier电视
class HaierTV implements Television {
    @Override
    public void use() {
        System.out.println("海尔电视正在使用...");
    }
}

//TCL电视
class TCLTV implements Television {
    @Override
    public void use() {
        System.out.println("TCL电视正在使用...");
    }
}

//工厂
interface Factory {
    AirConditioner getAirConditioner();
    Television getTelevision();
}

//Haier工厂
class HaierFactory implements Factory {
    @Override
    public AirConditioner getAirConditioner() {
        return new HaierAir();
    }
    @Override
    public Television getTelevision() {
        return new HaierTV();
    }
}

//TCL工厂
class TCLFactory implements Factory {
    @Override
    public AirConditioner getAirConditioner() {
        return new TCLAir();
    }
    @Override
    public Television getTelevision() {
        return new TCLTV();
    }
}

```

测试截图：

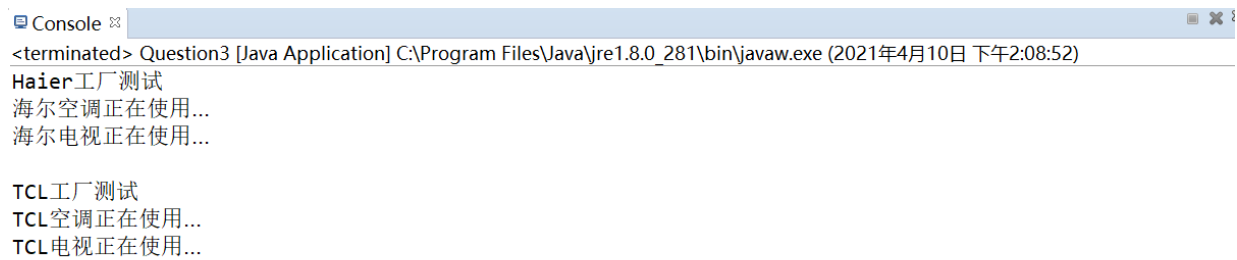


图 8 P70 的 4 题目测试截图

二、 参考课件，自己编写并实现单例模式的两种样式（静态单例版本和普通单例版本）

代码框架：

```
/**
 * 参考课件，自己编写并实现单例模式的两种样式（静态单例版本和普通单例版本）
 * 1.静态单例版本
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 */
public class StaticSingleton {
    //静态内部类
    private static class SingletonHolder {
        //静态初始化器机制初始化本数据（保证了同步控制，线程安全）
        private static StaticSingleton instance = new StaticSingleton();
    }
    //私有构造方法
    private StaticSingleton() {}
    //获取对象实例
    public static StaticSingleton getInstance() {
        return SingletonHolder.instance;
    }
    //测试
    public static void main(String[] args) {
        System.out.println("测试"+SingletonHolder.instance);
    }
}
```

测试截图：

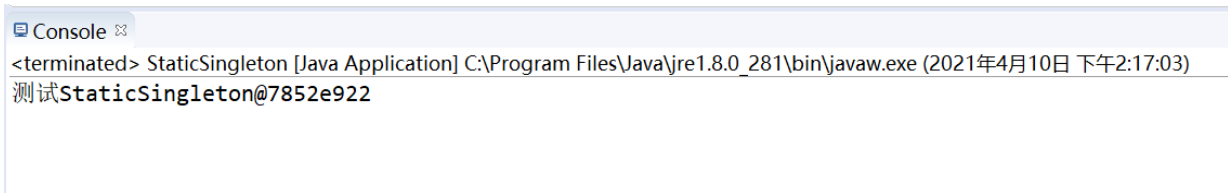


图 9 静态单例版本测试截图

```
/**
 * 参考课件，自己编写并实现单例模式的两种样式（静态单例版本和普通单例版本）
 * 2. 普通单例版本---饿汉式
 * @author Moppet
 * @since JDK 1.8.0
 * @data 2021/4/8
 * @version V1.0.0
 * 饿汉式：
 *     优点：简单方便
 *     缺点：不管程序中是否使用到了单例对象，都会生成单例对象，并且由于静态对象是否在类加载时就需要生成，会降低应用的启动速度
 *     使用场景：类对象功能简单，占用内存较小，使用频繁
 *     不适用：类对象功能赋值，占用内存大，使用概率较低
 */
public class HungrySingleton {
    //设立静态变量，直接创建实例
    private static HungrySingleton hungrySingleton = new HungrySingleton();

    private HungrySingleton(){
        //私有化构造函数
        System.out.println("-->饿汉式单例模式开始调用构造函数");
    }

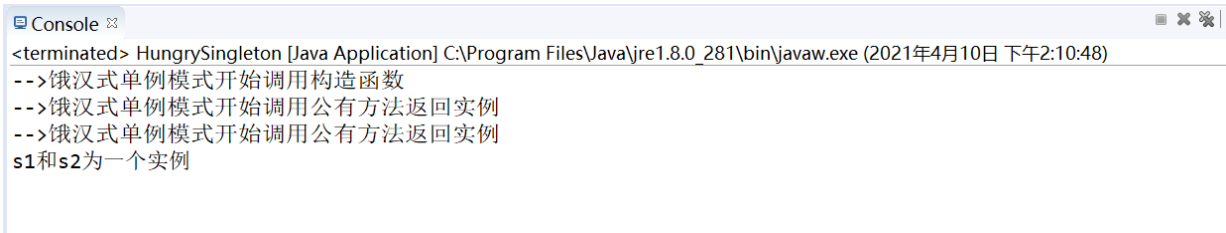
    //开放一个公有方法，判断是否已经存在实例，有返回，没有新建一个在返回
    public static HungrySingleton getInstance(){
        System.out.println("-->饿汉式单例模式开始调用公有方法返回实例");
        return hungrySingleton;
    }

    //测试
    public static void main(String[] args) {
        HungrySingleton s1 = HungrySingleton.getInstance();
        HungrySingleton s2 = HungrySingleton.getInstance();

        if(s1==s2) {
            System.out.println("s1和s2为一个实例");
        }
        System.out.println();
    }
}
```

}

测试截图：



```
Console
<terminated> HungrySingleton [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe (2021年4月10日 下午2:10:48)
-->饿汉式单例模式开始调用构造函数
-->饿汉式单例模式开始调用公有方法返回实例
-->饿汉式单例模式开始调用公有方法返回实例
s1和s2为一个实例
```

图 10 普通单例版本---饿汉式测试截图