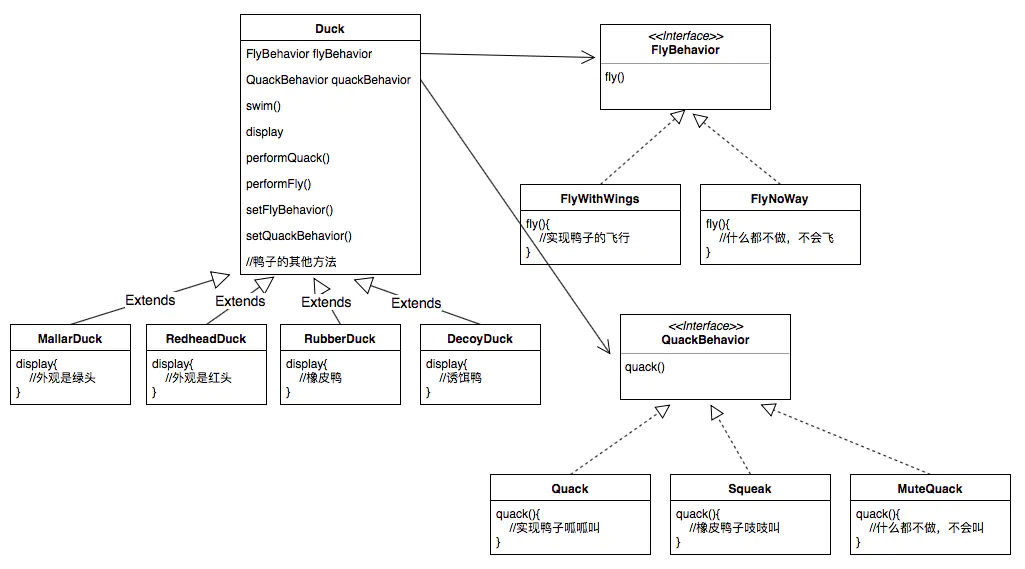
# 策略模式



public interface FlyBehavior {

    public void fly();

}

public class FlyWithWings implements FlyBehavior{

    @Override

    public void fly() {

        System.out.println("用翅膀飞");

    }

}

public class FlyNoWay implements FlyBehavior {

    @Override

    public void fly() {

        System.out.println("不会飞");

    }

}

public interface QuackBehaivor {

    public void quack();

}

public class Quack implements QuackBehaivor {

    @Override

    public void quack() {

        System.out.println("呱呱叫");

    }

}

public class MuteQuack implements QuackBehaivor {

    @Override

    public void quack() {

        System.out.println("不会叫");

    }

}

public abstract class Duck {

   FlyBehavior flyBehavior;

   QuackBehaivor quackBehavior;

   public abstract void display();

   public void performFly() {

       flyBehavior.fly();

   }

   public void performQuack() {

       quackBehavior.quack();

   }

   public void swim() {

       System.out.println("所有的鸭子都会漂浮");

   }

   // 动态设置飞行行为

   public void setFlyBehavior(FlyBehavior flyBehavior) {

       this.flyBehavior = flyBehavior;

   }

   //动态设置叫的行为

   public void setQuackBehavior(QuackBehaivor quackBehavior) {

       this.quackBehavior = quackBehavior;

   }

}

public class MallardDuck extends Duck {

    @Override

    public void display() {

        System.out.println("我是绿头鸭");

    }

}

public class MiniDuckSimulator {

    public static void main(String []args) {

        Duck mallard = new MallardDuck();

        mallard.setFlyBehavior(new FlyWithWings());//用翅膀飞

        mallard.setQuackBehavior(new Quack());//呱呱叫

        mallard.performFly();

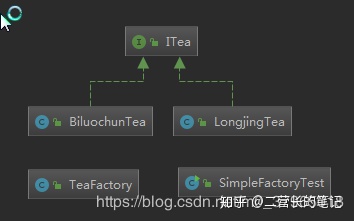
        mallard.performQuack();

    }

}

# 工厂方法模式

1. 简单工厂：



public interface ITea {

    // 沏茶

    public void makeTea();

}

public class LongjingTea implements ITea {

    @Override

    public void makeTea() {

        System.out.println("西湖龙井");

    }

}

public class BiluochunTea implements ITea{

    @Override

    public void makeTea() {

        System.out.println("碧螺春");

    }

}

/\*

准备沏一杯茶，根据传入的参数决定不同的茶叶类型

\*/

    public ITea prepareTea(String type){

        ITea tea = null;

        if(type.equals("longjing")){

            tea = new LongjingTea();

        }else if(type.equals("biluochun")){

            tea = new BiluochunTea();

        }

        if(tea != null){

            tea.makeTea();

        }

        return tea;

    }

public class TeaFactory {

    public ITea createTea(String type){

        ITea tea = null;

        if(type.equals("longjing")){

            tea = new LongjingTea();

        }else if(type.equals("biluochun")){

            tea = new BiluochunTea();

        }

        if(tea != null){

            tea.makeTea();

        }

        return tea;

    }

}

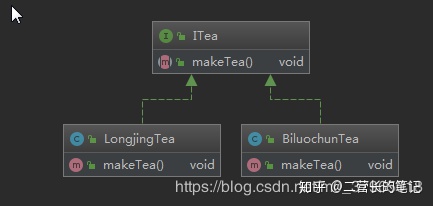
public static void main(String[] args) {

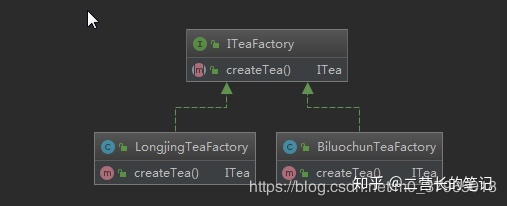
        TeaFactory teaFactory = new TeaFactory();

        ITea tea = teaFactory.createTea("longjing");

    }

1. 工厂方法：





/\*\*

 \* 生产茶叶的统一接口

 \*/

public interface ITeaFactory {

    // 生产茶叶

    public ITea createTea();

}

public class LongjingTeaFactory implements ITeaFactory{

    @Override

    public ITea createTea() {

        return new LongjingTea();

    }

}

public class BiluochunTeaFactory implements ITeaFactory{

    @Override

    public ITea createTea() {

        return new BiluochunTea();

    }

}

public class FactoryMethodTest {

    public static void main(String[] args) {

        ITeaFactory factory = new LongjingTeaFactory();

        factory.createTea();

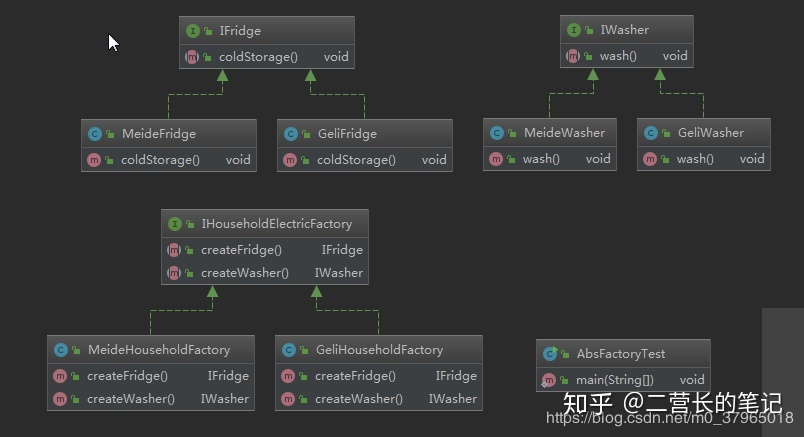
        factory = new BiluochunTeaFactory();

        factory.createTea();

    }

}

1. 抽象工厂：



/\*\*

 \* 冰箱

 \*/

public interface IFridge {

    // 冷藏

    void coldStorage();

}

/\*\*

 \* 洗衣机

 \*/

public interface IWasher {

    void wash();

}

public class MeideFridge implements IFridge{

    @Override

    public void coldStorage() {

        System.out.println("美的冰箱");

    }

}

public class MeideWasher implements IWasher {

    @Override

    public void wash() {

        System.out.println("美的洗衣机");

    }

}

public class GeliFridge implements IFridge {

    @Override

    public void coldStorage() {

        System.out.println("格力冰箱");

    }

}

public class GeliWasher implements IWasher{

    @Override

    public void wash() {

        System.out.println("格力洗衣机");

    }

}

/\*\*

 \* 抽象工厂接口，家用电器工厂，生产冰箱和洗衣机

 \*/

public interface IHouseholdElectricFactory {

    IFridge createFridge();

    IWasher createWasher();

}

public class MeideHouseholdFactory implements IHouseholdElectricFactory{

    @Override

    public IFridge createFridge() {

        return new MeideFridge();

    }

    @Override

    public IWasher createWasher() {

        return new MeideWasher();

    }

}

public class GeliHouseholdFactory implements IHouseholdElectricFactory{

    @Override

    public IFridge createFridge() {

        return new GeliFridge();

    }

    @Override

    public IWasher createWasher() {

        return new GeliWasher();

    }

}

public class AbsFactoryTest {

    public static void main(String[] args) {

        IHouseholdElectricFactory factory = new MeideHouseholdFactory();

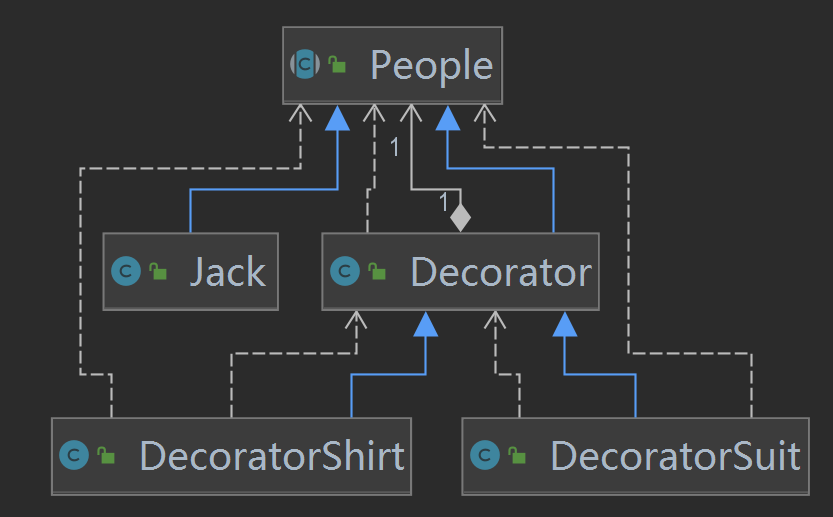
        factory.createFridge().coldStorage();

        factory.createWasher().wash();

    }

}

# 装饰着模式



/\*\*

 \* 模拟一个人穿衣服的过程

 \* 抽象类，用来规范将要被附加一些操作的对象

 \*/

public abstract class People {

    public abstract void wear();

}

/\*\*

 \* 具体的对象，该对象将被附加一些额外的操作

 \*/

public class Jack extends People {

    @Override

    public void wear() {

        System.out.println("该穿什么衣服呢");

    }

}

————————————————

/\*\*

 \* 装饰者类，持有一个将要被装饰者的接口对象实例

 \*/

public class Decorator extends People {

    public People people;

    //记得生成带参构造，用来给被装饰者赋值

    public Decorator(People people) {

        this.people = people;

    }

    @Override

    public void wear() {

        people.wear();

    }

}

/\*\*

 \* 具体的装饰者类，负责给增加附加的操作：穿衬衫

 \*/

public class DecoratorShirt extends Decorator {

    public DecoratorShirt(People people) {

        super(people);

    }

    public void  wear(){

        super.wear();

        System.out.println("穿衬衫");

    }

}

/\*\*

 \* 具体的装饰者类，负责给增加附加的操作：穿西服

 \*/

public class DecoratorSuit extends Decorator {

    public DecoratorSuit(People people) {

        super(people);

    }

    public void wear() {

        super.wear();

        System.out.println("穿个西服");

    }

}

public class Client {

    public static void main(String[] args) {

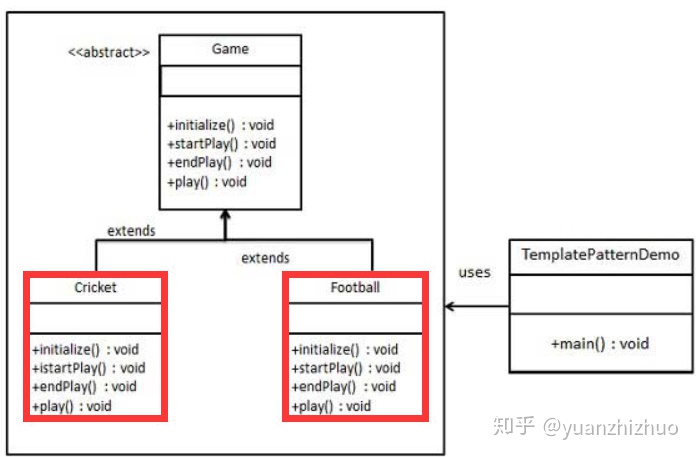
        People people =new DecoratorShirt(new DecoratorSuit(new Jack()));

        people.wear();

    }

}

# 模板方法模式



public abstract class Game {

    abstract void initialize();

    abstract void startPlay();

    abstract void endPlay();

    //模板规定了方法的顺序

    public final void play() {

        initialize();

        startPlay();

        endPlay();

    }

}

public class Cricket extends Game {

    @Override

    void initialize() {

        System.out.println("Cricket Game Initialized! Start playing.");

    }

    @Override

    void startPlay() {

        System.out.println("Cricket Game Started. Enjoy the game!");

    }

    @Override

    void endPlay() {

        System.out.println("Cricket Game Finished!");

    }

}

public class Football extends Game {

    @Override

    void initialize() {

        System.out.println("Football Game Initialized! Start playing.");

    }

    @Override

    void startPlay() {

        System.out.println("Football Game Started. Enjoy the game!");

    }

    @Override

    void endPlay() {

        System.out.println("Football Game Finished!");

    }

}

public class TemplatePatternDemo {

    public static void main(String[] args) {

        Game cricket = new Cricket();

        cricket.play();

    }

}