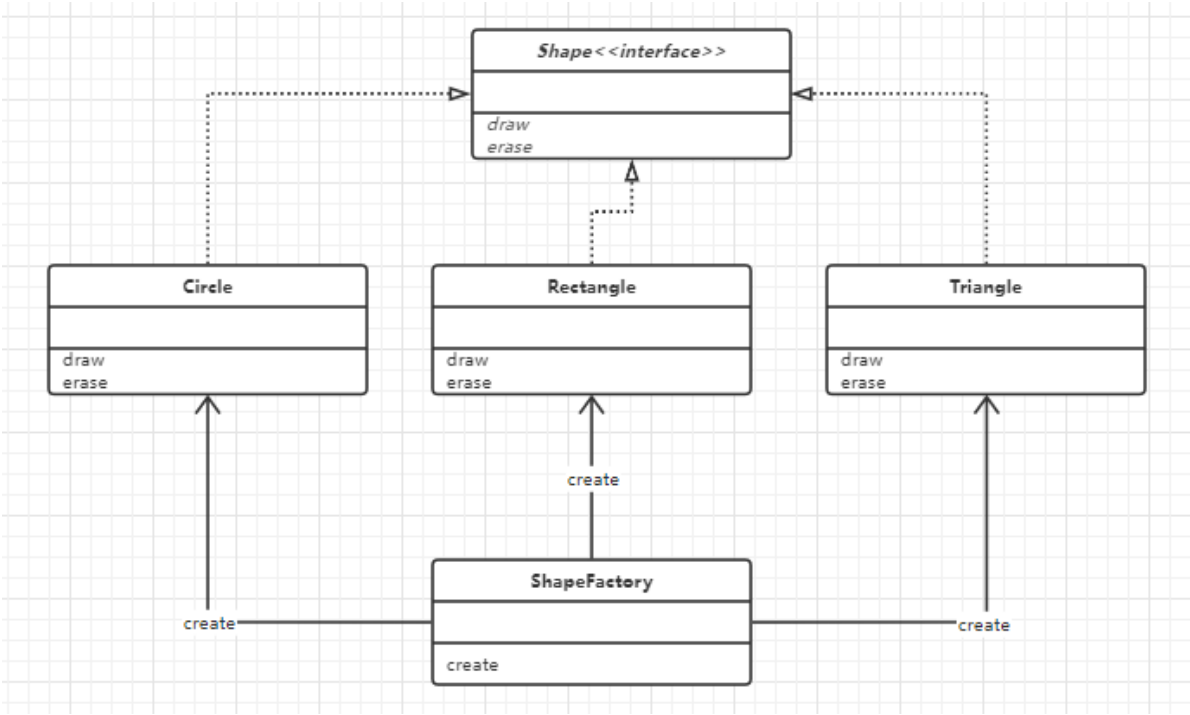
1. 简单工厂模式

简单工厂模式使用简单工厂模式设计一个可以创建不同几何形状（Shape）（例如圆形（Circle）、矩形（Rectangle）和三角形（Triangle）等）的绘图工具类，每个几何图形均具有绘制方法 draw()和擦除方法 erase()，要求在绘制不支持的几何图形时，抛出一个UnsupportedShapeException 异常。绘制类图并编程模拟实现。




```
package simple_factory;

public class ShapeFactory {
    public Shape create(String type) throws UnsupportedShapeException {
        switch (type){
            case "c":
                return new Circle();
            case "r":
                return new Rectangle();
            case "t":
                return new Triangle();
            default:
                throw new UnsupportedShapeException();
        }

    }
}
```

```
package simple_factory;

public interface Shape {
    public void draw();
    public void erase();
}
```

```java
package simple_factory;

public class Rectangle implements Shape{
    @Override
    public void draw() {
        System.out.println("画一个矩形");
    }

    @Override
    public void erase() {
        System.out.println("擦除一个矩形");
    }
}
```

```java
package simple_factory;

public class Triangle implements Shape{
    @Override
    public void draw() {
        System.out.println("画一个三角形");
    }

    @Override
    public void erase() {
        System.out.println("擦除一个三角形");
    }
}
```

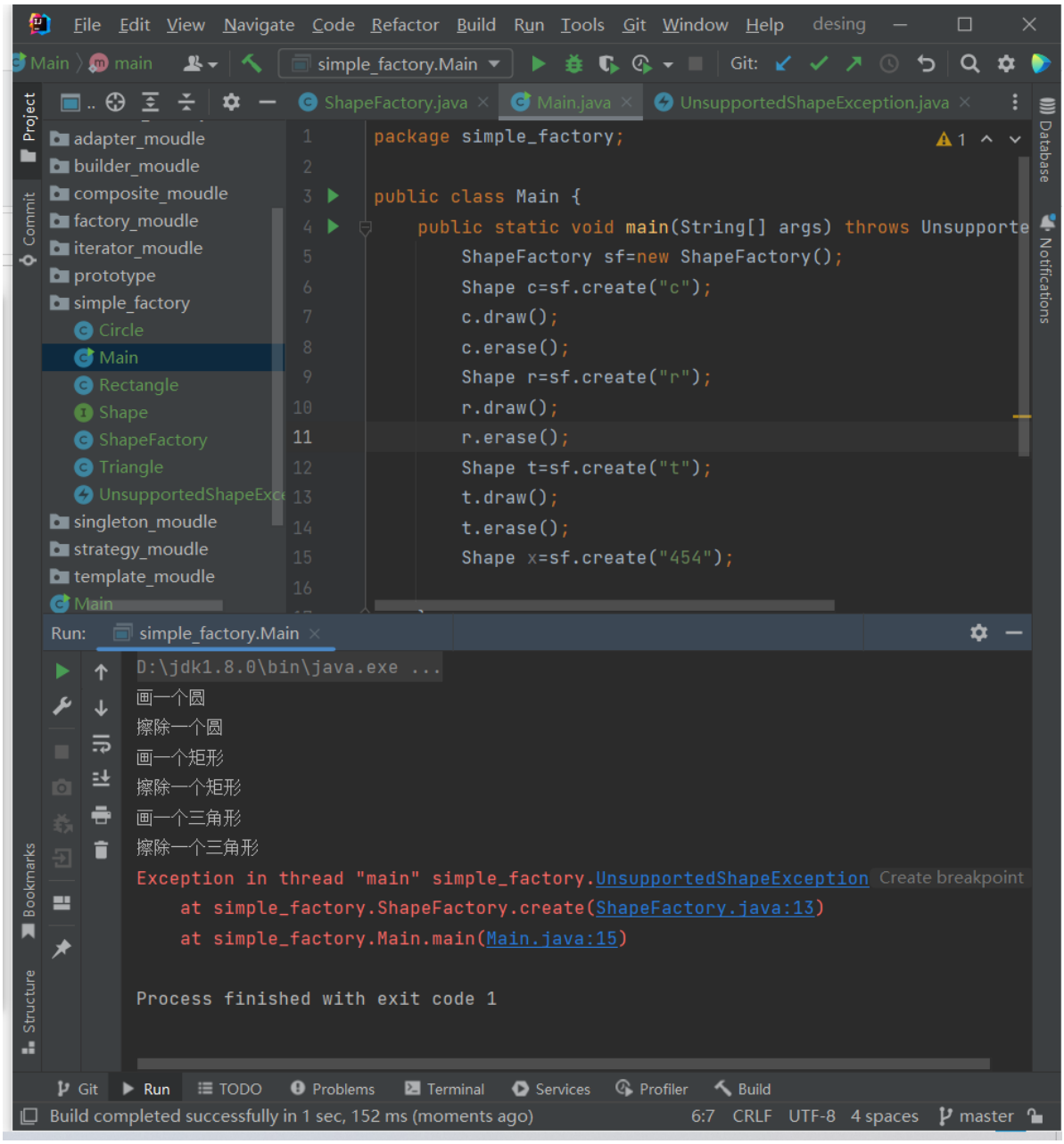
```java
package simple_factory;

public class Circle implements Shape{
    @Override
    public void draw() {
        System.out.println("画一个圆");
    }

    @Override
    public void erase() {
        System.out.println("擦除一个圆");
    }
}
```

```java
package simple_factory;

public class UnsupportedShapeException extends Exception{

}
```

```java
package simple_factory;

public class Main {
    public static void main(String[] args) throws UnsupportedShapeException {
```

```
        ShapeFactory sf=new ShapeFactory();
        Shape c=sf.create("c");
        c.draw();
        c.erase();
        Shape r=sf.create("r");
        r.draw();
        r.erase();
        Shape t=sf.create("t");
        t.draw();
        t.erase();
        Shape x=sf.create("454");

    }
}
```

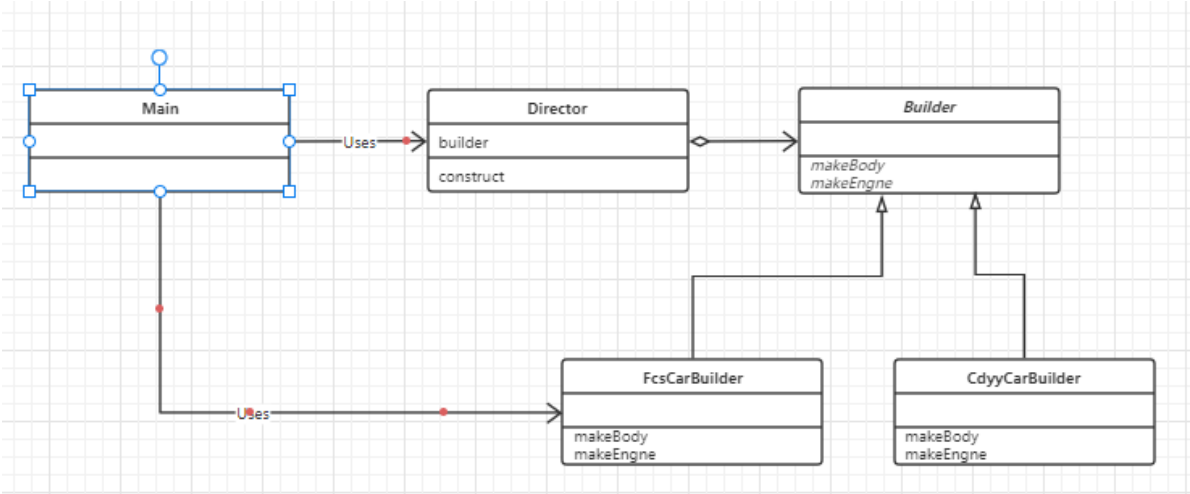
运行main结果:



2. 建造者模式

在某赛车游戏中，赛车包括方程式赛车、场地越野赛车、运动汽车、卡车等类型，不同类型的赛车的车身、发动机、轮胎、变速箱等部件有所区别。玩家可以自行选择赛车类型，系统将根据玩家的选择创建出一辆完整的赛车。现采用建造者模式实现赛车的构建，绘制对应的类图并编程模拟实现。

以方程式赛车、场地越野赛车为例，可以为赛车构建车身和发动机




```
package builder_moudle;

public class Main {
    public static void main(String[] args) {
        Builder fcs=new FcsCarBuilder();
        Director d=new Director(fcs);
        d.construct();

        System.out.println("--------------------");

        Builder cdyy=new CdyyCarBuilder();
        Director d1=new Director(cdyy);
        d1.construct();

    }
}
```

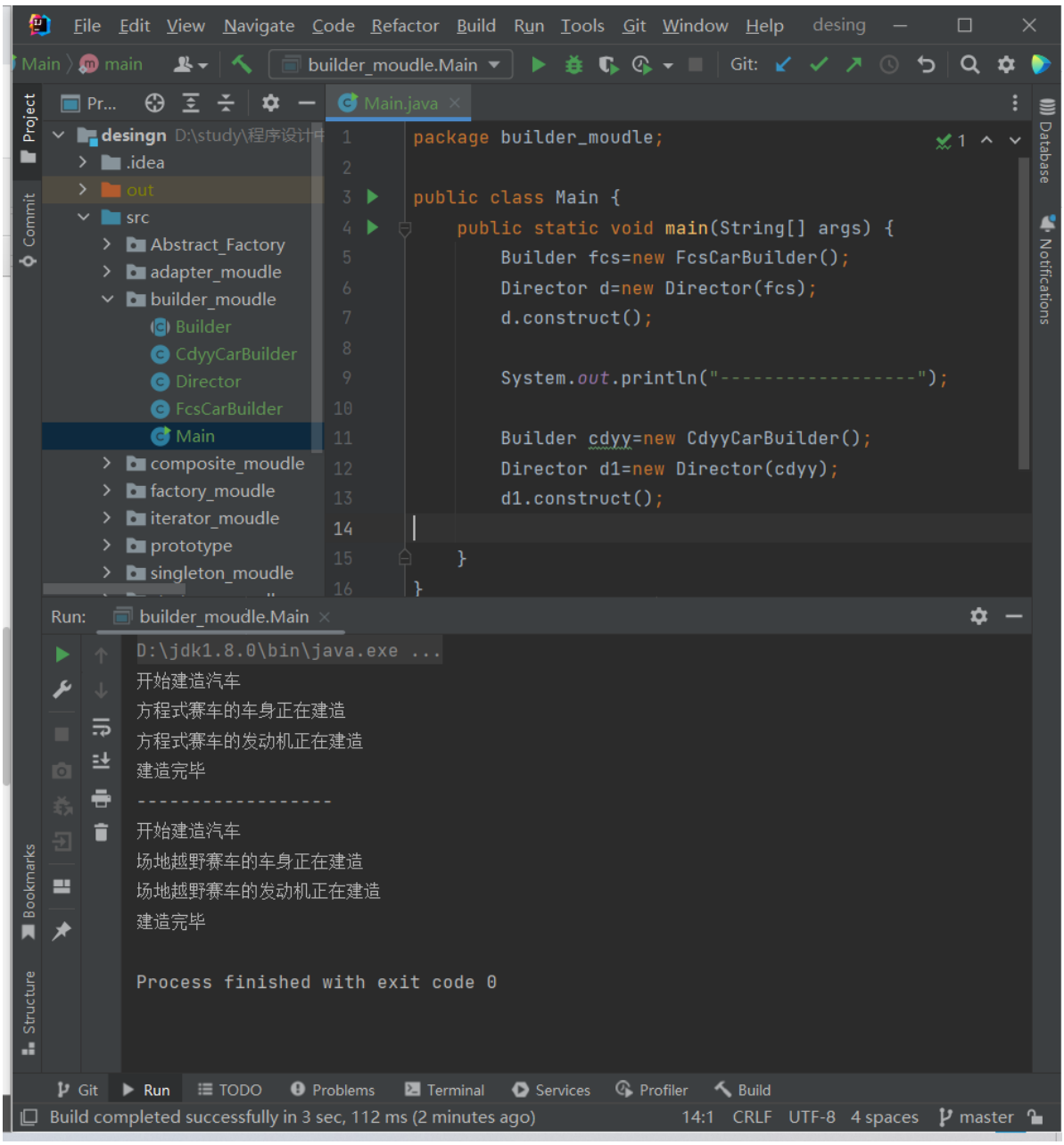
```
package builder_moudle;

public abstract class Builder {
    public abstract void makeBody();
    public abstract void makeEngne();
}
```

```java
package builder_moudle;

public class CdyyCarBuilder extends Builder{
    @Override
    public void makeBody() {
        System.out.println("场地越野赛车的车身正在建造");
    }
    @Override
    public void makeEngne() {
        System.out.println("场地越野赛车的发动机正在建造");
    }
}
```

```java
package builder_moudle;

public class FcsCarBuilder extends Builder{
    @Override
    public void makeBody() {
        System.out.println("方程式赛车的车身正在建造");
    }

    @Override
    public void makeEngne() {
        System.out.println("方程式赛车的发动机正在建造");
    }
}
```

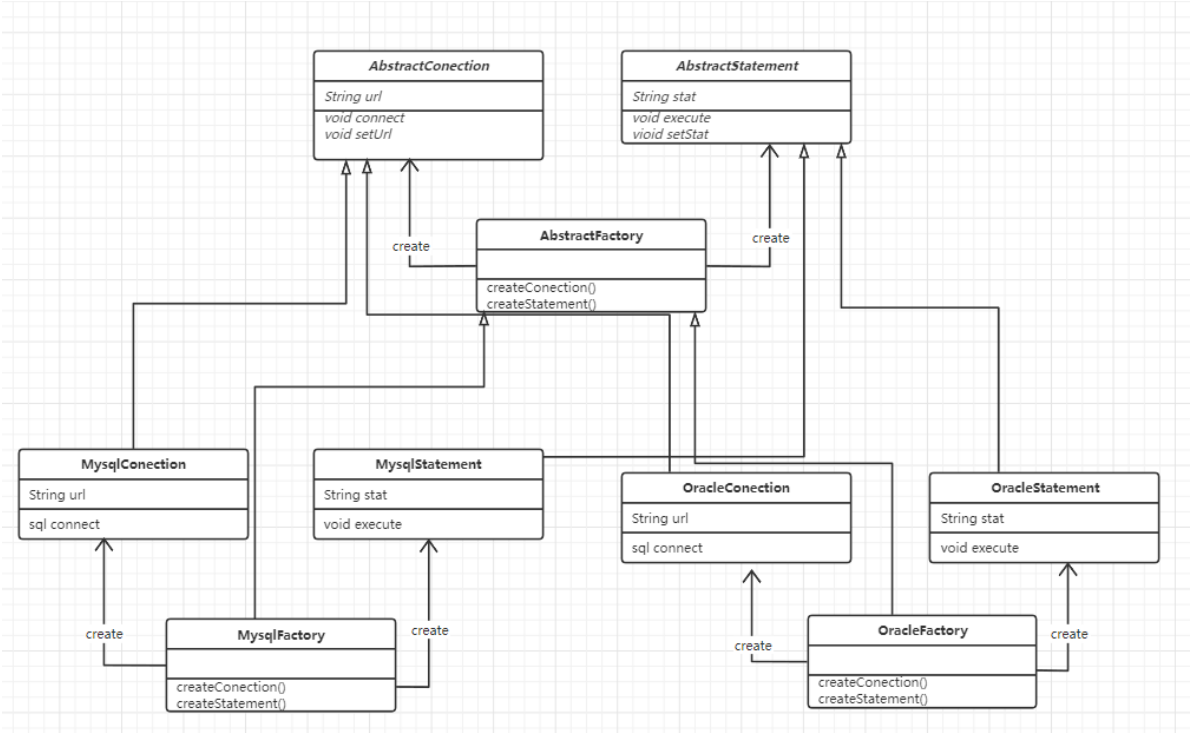
```java
package builder_moudle;

public class Director {
    Builder builder;
    public Director(Builder builder){
        this.builder=builder;
    }
    public void construct(){
        System.out.println("开始建造汽车");
        builder.makeBody();
        builder.makeEngne();
        System.out.println("建造完毕");
    }
}
```

运行main结果



3. 抽象工厂模式

某系统为了改进数据库操作的性能，用户可以自定义数据库连接对象Connection和语句对象 Statement，针对不同类型的数据库提供不同的连接对象和语句对象，例如提供Oracle 或 MySQL 专用连接类和语句类，而且用户可以通过配置文件等方式根据实际需要




```java
package Abstract_Factory.factory;

public abstract class AbstractConection {
    protected String url;
    public abstract void connect();

    public void setUrl(String url) {
        this.url = url;
    }
}
```

```java
package Abstract_Factory.factory;

public abstract class AbstractFactory {
    public abstract AbstractConection createConection(String url);
    public abstract AbstractStatement createStatement(String stat);
}
```

```java
package Abstract_Factory.factory;

public abstract class AbstractStatement {
    protected String stat;
    public abstract void execute();
    public void setStat(String stat){
        this.stat=stat;
    }
}
```

```java
package Abstract_Factory.mysqlfactory;

import Abstract_Factory.factory.AbstractConection;

public class MysqlConection extends AbstractConection {
    @Override
    public void connect() {
        System.out.println("mysql连接"+this.url);
    }
}
```

```java
package Abstract_Factory.mysqlfactory;

import Abstract_Factory.factory.AbstractConection;
import Abstract_Factory.factory.AbstractFactory;
import Abstract_Factory.factory.AbstractStatement;

public class MysqlFactory extends AbstractFactory {
    @Override
    public AbstractConection createConection(String url) {
        System.out.println("mysql工厂创建连接");
        AbstractConection con=new MysqlConection();
        con.setUrl(url);
        return con;
    }

    @Override
    public AbstractStatement createStatement(String stat) {
        System.out.println("mysql工厂创建语句");
        AbstractStatement sta=new MysqlStatement();
        sta.setStat(stat);
        return sta;
    }
}
```

```java
package Abstract_Factory.mysqlfactory;

import Abstract_Factory.factory.AbstractStatement;

public class MysqlStatement extends AbstractStatement {
    @Override
    public void execute() {
        System.out.println("mysql执行"+this.stat);
    }
}
```

```java
package Abstract_Factory.oraclefactory;

import Abstract_Factory.factory.AbstractConection;

public class OracleConection extends AbstractConection {
    @Override
    public void connect() {
        System.out.println("oracle连接"+this.url);
    }
}
```

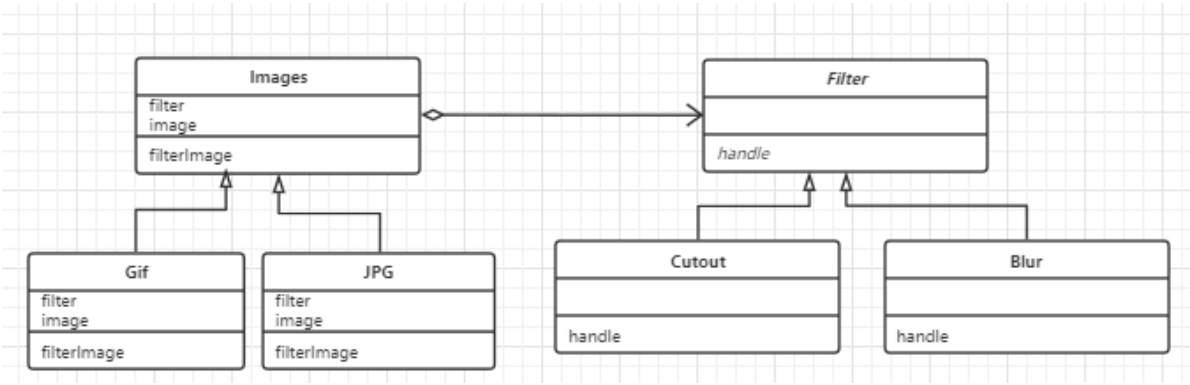
```java
package Abstract_Factory.oraclefactory;

import Abstract_Factory.factory.AbstractConection;
import Abstract_Factory.factory.AbstractFactory;
import Abstract_Factory.factory.AbstractStatement;
import Abstract_Factory.mysqlfactory.MysqlConection;
import Abstract_Factory.mysqlfactory.MysqlStatement;

public class OracleFactory extends AbstractFactory {
    @Override
    public AbstractConection createConection(String url) {
        System.out.println("oracle工厂创建连接");
        AbstractConection con=new OracleConection();
        con.setUrl(url);
        return con;
    }

    @Override
    public AbstractStatement createStatement(String stat) {
        System.out.println("oracle工厂创建语句");
        AbstractStatement sta=new OracleStatement();
        sta.setStat(stat);
        return sta;
    }
}
```

```java
package Abstract_Factory.oraclefactory;

import Abstract_Factory.factory.AbstractStatement;

public class OracleStatement extends AbstractStatement {
    @Override
    public void execute() {
        System.out.println("oracle执行"+this.stat);
    }
}
```

4. 桥接模式

某手机美图 APP 软件支持多种不同的图像格式，例如 JPG、GIF、BMP 等常用图像格式，同时提供了多种不同的滤镜对图像进行处理，例如木刻滤镜（Cutout）、模糊滤镜(Blur）、锐化滤镜（Sharpen）、纹理滤镜（Texture）等。现采用桥接模式设计该APP 软件，使得该软件能够为多种图像格式提供一系列图像处理滤镜，同时还能够很方便地增加新的图像格式和滤镜，绘制对应的类图并编程模拟实现。

Images为抽象化的图片处理，Filter为滤镜的实现，增加Filter的新功能直接去实现Filer接口即可，增加新的图片格式直接继承Images即可。




```
package bridge_moudle;

public class Images {
    Filter filter;
    public Images(Filter filter){
        this.filter=filter;
    }
    public void filterImage() {
        this.filter.handle();
    }
}
```

```
package bridge_moudle;

public class Gif extends Images{

    public Gif(Filter filter) {
        super(filter);
    }
    public void process(){
        System.out.println("打开Gif");
        this.filterImage();
        System.out.println("对Gif滤镜处理完毕");
    }
}
```

```java
package bridge_moudle;

public class JPG extends Images{
    public JPG(Filter filter){
        super(filter);
    }
    public void process(){
        System.out.println("打开jpg");
        this.filterImage();
        System.out.println("对jpg滤镜处理完毕");
    }
}
```

```java
package bridge_moudle;

public abstract class Filter {
    public abstract void handle();
}
```

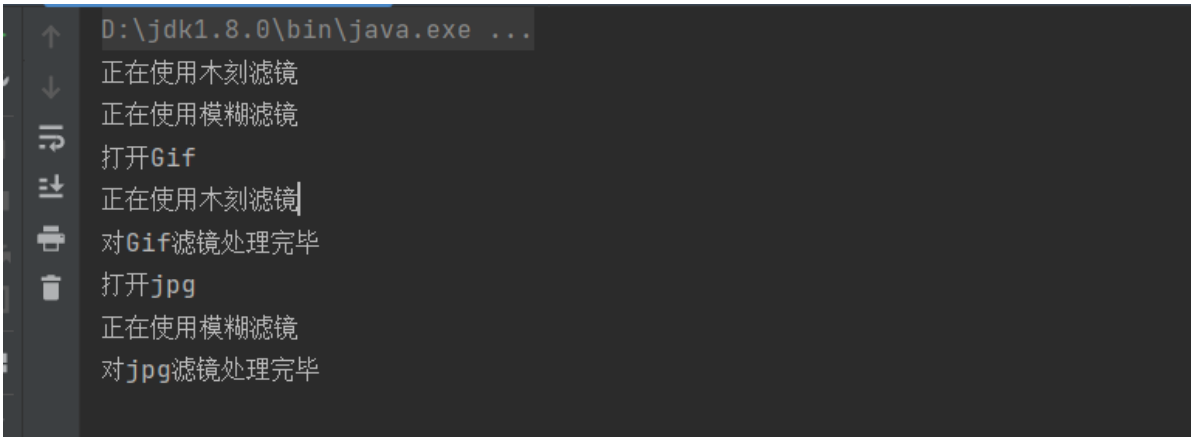
```java
package bridge_moudle;

public class Cutout extends Filter{
    @Override
    public void handle() {
        System.out.println("正在使用木刻滤镜");
    }
}
```

```java
package bridge_moudle;

public class Blur extends Filter{
    @Override
    public void handle() {
        System.out.println("正在使用模糊滤镜");
    }
}
```

```java
package bridge_moudle;

public class Main {
    public static void main(String[] args) {
        Images i1=new Images(new Cutout());
        Images i2=new Images(new Blur());
        i1.filterImage();
        i2.filterImage();

        Gif f1=new Gif(new Cutout());
        JPG J1=new JPG(new Blur());
        f1.process();
        J1.process();

    }
```

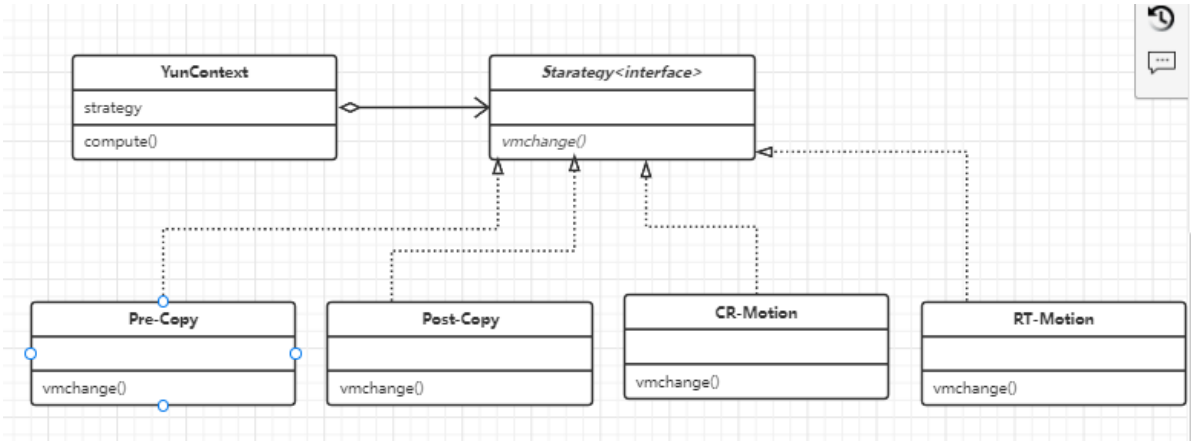
```
}
```

运行main结果：



5. 策略模式

在某云计算模拟平台中提供了多种虚拟机迁移算法，例如动态迁移算法中的Pre- Copy （预拷贝）算法、 Post - Copy （后拷贝）算法、 CR / RT - Motion 算法等，用户可以灵活地选择所需的虚拟机迁移算法，也可以方便地增加新算法。现采用策略模式进行设计，绘制对应的类图并编程模拟实现。




```java
package strategy_moudle;

public class YunContext {
    Starategy strategy;
    public YunContext(Starategy strategy){
        this.strategy=strategy;
    }
    public void compute(){
        this.strategy.vmchange();
        System.out.println("完成虚拟化，计算1+1=2,计算完成");
    }
}
```

```java
package strategy_moudle;

public interface Starategy {
    public void vmchange();
}
```

```java
package strategy_moudle;

public class CRMotion implements Starategy{

    @Override
    public void vmchange() {
        System.out.println("正在使用CRMotion方法虚拟化");
    }
}
```

```java
package strategy_moudle;

public class PostCopy implements Starategy{
    @Override
    public void vmchange() {
        System.out.println("正在使用PostCopy方法虚拟化");
    }
}
```

```java
package strategy_moudle;

public class PreCopy implements Starategy{
    @Override
    public void vmchange() {
        System.out.println("正在使用PreCopy方法虚拟化");
    }
}
```

```java
package strategy_moudle;

public class RTMotion implements Starategy{

    @Override
    public void vmchange() {
        System.out.println("正在使用RTMotion方法虚拟化");
    }
}
```