



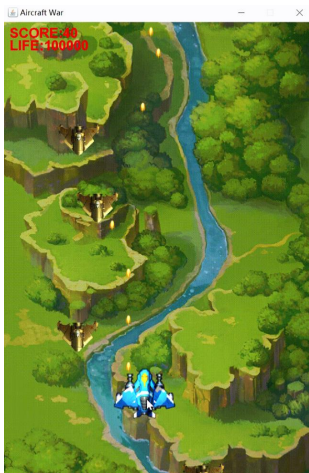
# 实验四：设计模式实验（2）

## 策略模式和数据访问对象模式

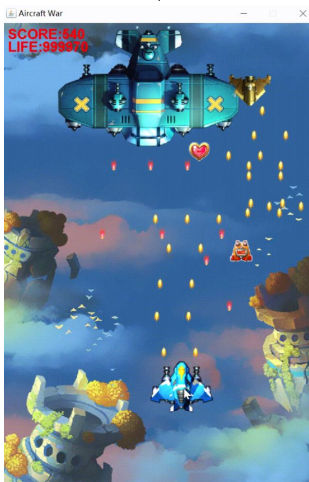
实验与创新实践教育中心 · 计算机与数据技术实验教学部

# 本学期实验总体安排

初始版本



最终版本



**游戏主界面**  
英雄机移动  
英雄机子弹直射  
碰撞检测  
统计得分和生命值

重构代码，采用**单例模式**  
创建英雄机  
重构代码，采用**工厂模式**  
创建敌机和道具

重构代码，采用**策略模式**  
实现不同弹道发射  
采用**数据访问对象模式**  
实现得分排行榜

采用**观察者模式**  
实现炸弹道具生效  
采用**模板模式**  
实现三种游戏难度

初始版本

01

**绘制UML类图**

创建精英敌机并直射子弹  
精英敌机随机掉落三种道具  
加血道具生效

02

03

**添加JUnit单元测试**

创建Boss和超级精英敌机

04

05

使用**Swing**添加游戏难度选择和  
排行榜界面  
使用**多线程**实现音效的开启/关闭、  
及火力道具

06

# 本学期实验总体安排

实验项目	一	二	三	四	五	六
学时数	2	2	2	2	4	4
实验内容	飞机大战 功能分析	单例模式 工厂模式	Junit 单元测试	策略模式 数据访问 对象模式	Swing 多线程	观察者模式 模板模式
分数	4	6	4	6	6	14 (6+8)
提交内容	UML类图、 代码	UML类图、 代码	测试报告、 代码	UML类图、 代码	代码	项目代码、实 验报告、展示 视频

实验课程共**16**个学时，**6**个实验项目，总成绩为**40**分。

# CONTENTS

## 目录

01 实验目的

02 实验任务

03 实验原理

04 实验步骤

# 实验目的

难度	知识点
理解	策略模式和数据访问对象模式的模式动机和意图
掌握	策略和数据访问对象模式UML结构图的绘制方法
熟练	使用Java语言，编码实现策略和数据访问对象模式



# 实验任务

绘制类图、重构代码，完成以下功能：

1. 采用策略模式实现不同机型的弹道发射及两种火力道具（需求变更）的加成效果；
2. 采用数据访问对象模式实现玩家的得分排行榜。

注意：先“设计”再编码！请结合飞机大战实例，完成模式UML类图设计后，再进行编码。



## 课前小测



请思考：

在面向对象编程中，如何实现代码复用？（多选）



☐ A . 封装

☐ B . 继承

☐ C . 组合

答案：B C


**继承：**在基类的基础上创建新类，新类可直接复用基类的属性和方法；

**组合：**在新类中创建已有类的对象，通过该对象来调用已有类中的属性和方法。

# 实验原理：场景分析（1）

发射  
弹道场景  
分析

飞机大战游戏中，不同类型飞机的发射弹道各不相同，且火力道具生效时英雄机切换弹道。

类型		弹道
普通敌机		不发射
精英敌机		直射
超级精英敌机		散射
Boss敌机		环射
英雄机	无道具	直射
	火力道具 	散射
	超级火力道具 	环射



## 实验原理：场景分析（2）





**请思考：**1. 目前各种飞机的子弹发射在哪个类实现？

2. 如何实现**火力道具**？


```
@Override
public List<BaseBullet> shoot() {
    List<BaseBullet> res = new LinkedList<>();
    int x = this.getLocationX();
    int y = this.getLocationY() + direction*2;
    int speedX = 0;
    int speedY = this.getSpeedY() + direction*5;
    BaseBullet bullet;
    for(int i=0; i<shootNum; i++){
        bullet = new HeroBullet( locationX: x + (i*2 - shootNum + 1)*10,
            y, speedX, speedY, power);
        res.add(bullet);
    }
    return res;
}
```

直射



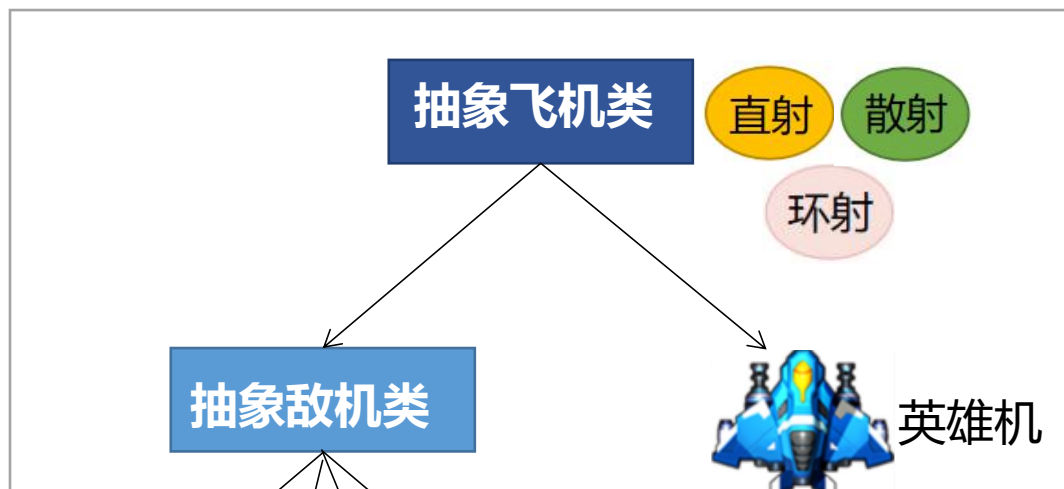
```
@Override
public List<BaseBullet> shoot() {
    List<BaseBullet> res = new LinkedList<>();
    int x = this.getLocationX();
    int y = this.getLocationY() + direction*2;
    int speedX = this.getSpeedX();
    int speedY = this.getSpeedY() + direction*5;
    BaseBullet bullet;
    for(int i=0; i<shootNum; i++){
        // 多个子弹横向分散, 呈扇形
        bullet = new EnemyBullet( locationX: x + (i * 2 - shootNum + 1) * 10, y,
            speedX: speedX + (i * 2 - shootNum + 1), speedY, power);
        res.add(bullet);
    }
    return res;
}
```

散射



## 实验原理：场景分析（3）

💡 **请思考：** 2. 如何实现火力道具？ 3. 如何增加新的机型或弹道？



### 01 方案一

将散射和环射代码复制一份到英雄机类

重复代码多  
代码难维护

违反  
开闭原则

X

### 02 方案二

优先使用**对象组合**，而不是继承来实现代码复用！

违反开闭  
复用原则

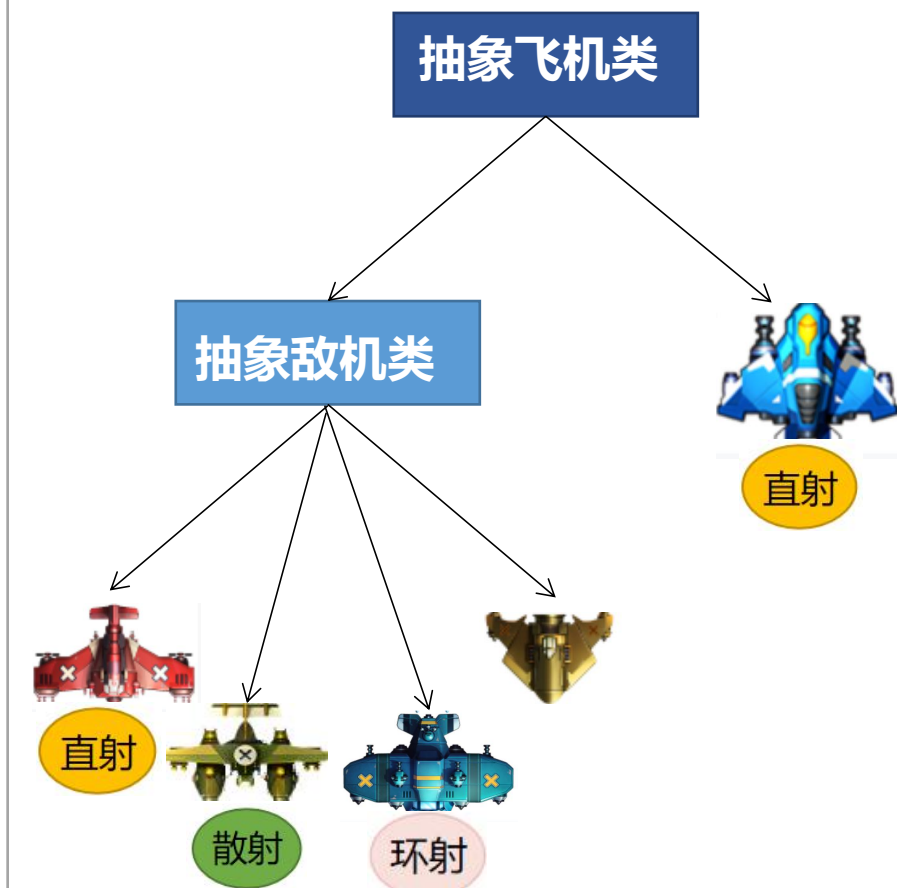
X

多重选择  
逻辑复杂  
代码可读性差

## 实验原理：场景分析（4）



**请思考：**如何使用对象组合而不是继承来实现代码复用？

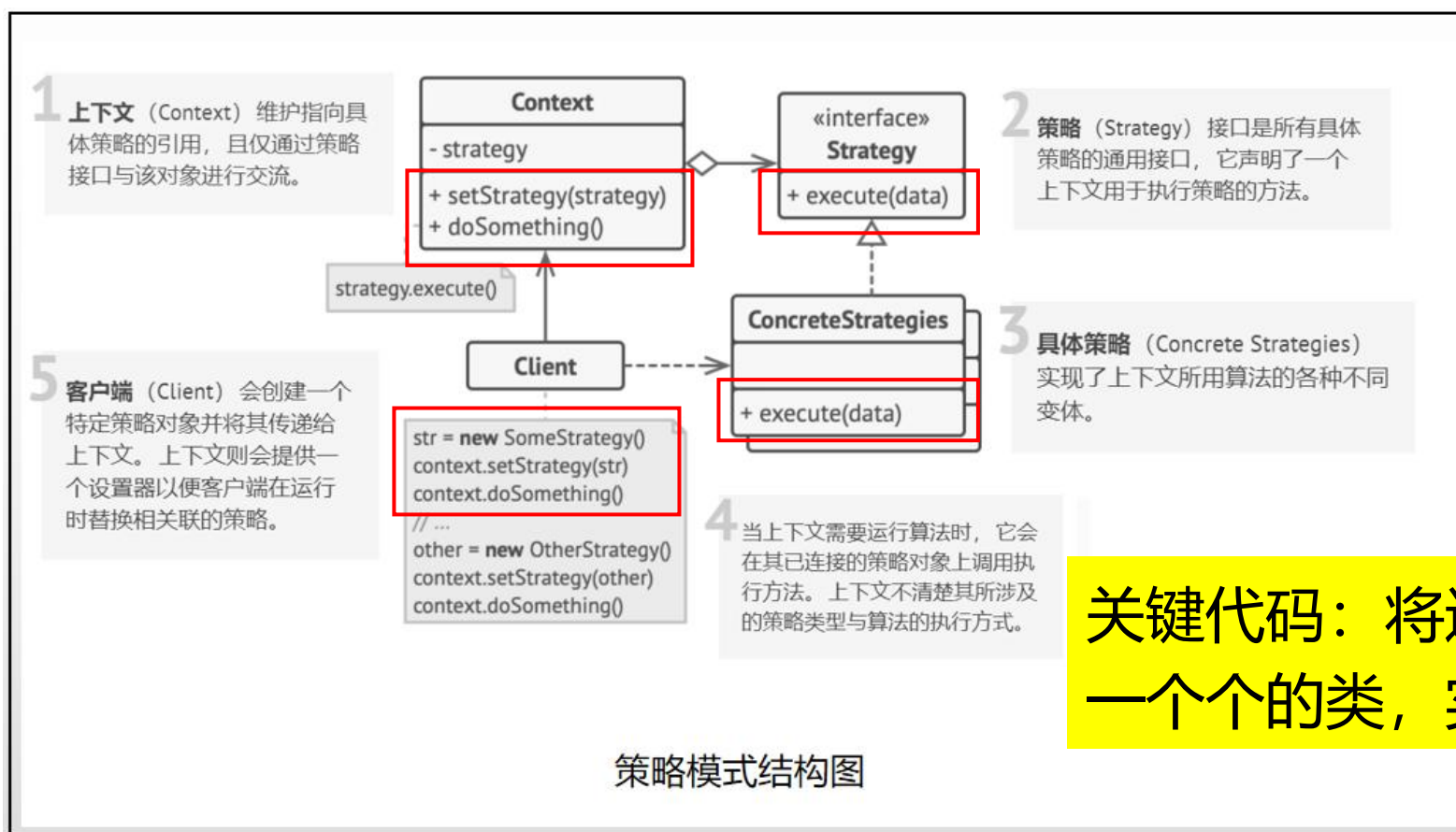


所有设计模式的实现都遵循一条原则：  
**找出程序中变化的地方，并将变化封装起来！**



# 实验原理：策略模式结构图

**策略模式** (Strategy Pattern) 是一种行为设计模式，它能让你定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够相互替换。



关键代码：将这些算法封装成一个个的类，实现同一个接口。

策略模式结构图

# 实验步骤：策略模式



## 实验步骤：计算器举例（策略模式）

假如我们要实现一个**计算器**，  
支持加法、减法、乘法、除法  
四种运算。我们该如何用策略  
模式实现呢？



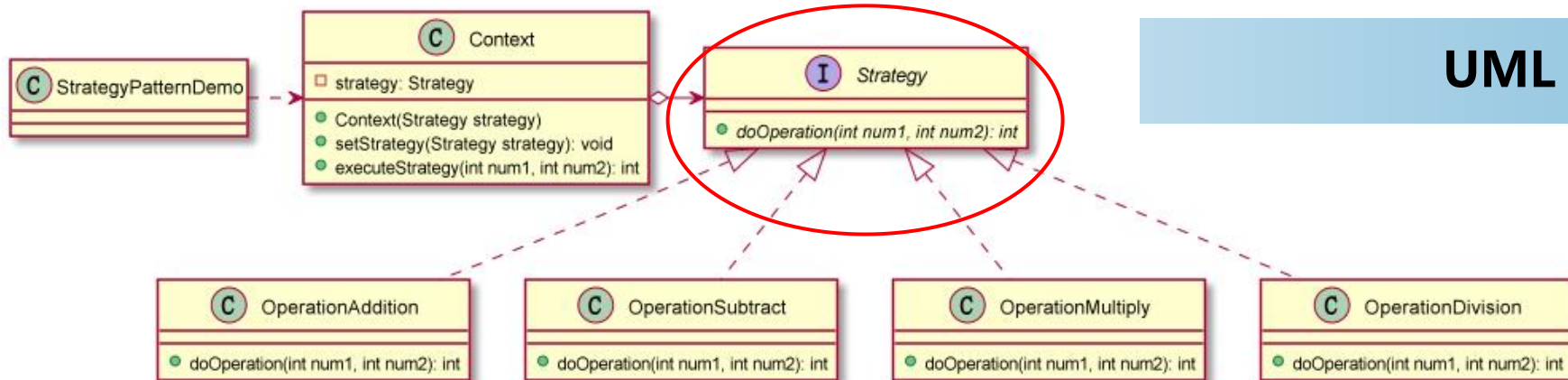
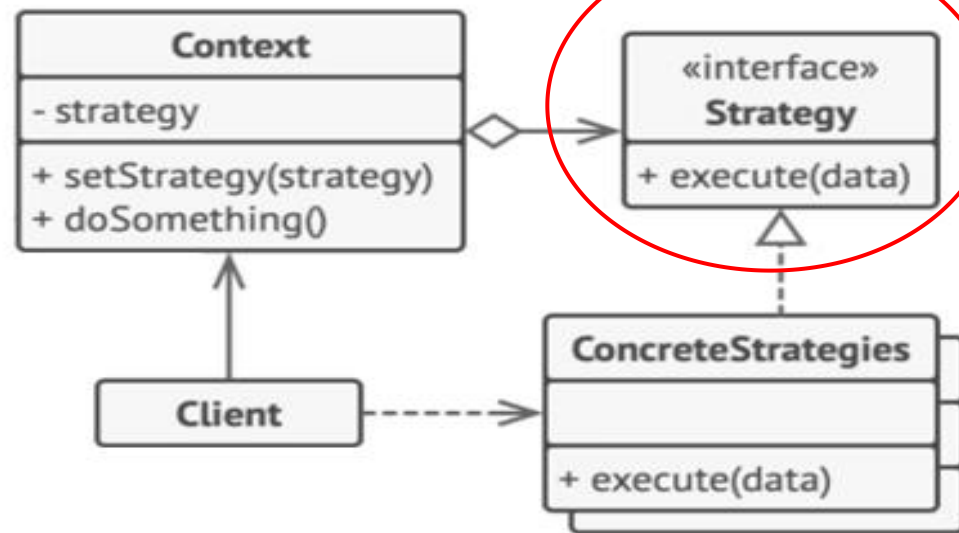


# 实验步骤：计算器举例（1）

## 代码实现

- ① 创建一个Strategy接口，充当抽象策略角色；

```
public interface Strategy {  
    int doOperation(int num1, int num2);  
}
```



## UML 类图设计

## 实验步骤：计算器举例（2）

### 代码实现

② 创建四个运算实体类，充当具体策略角色；

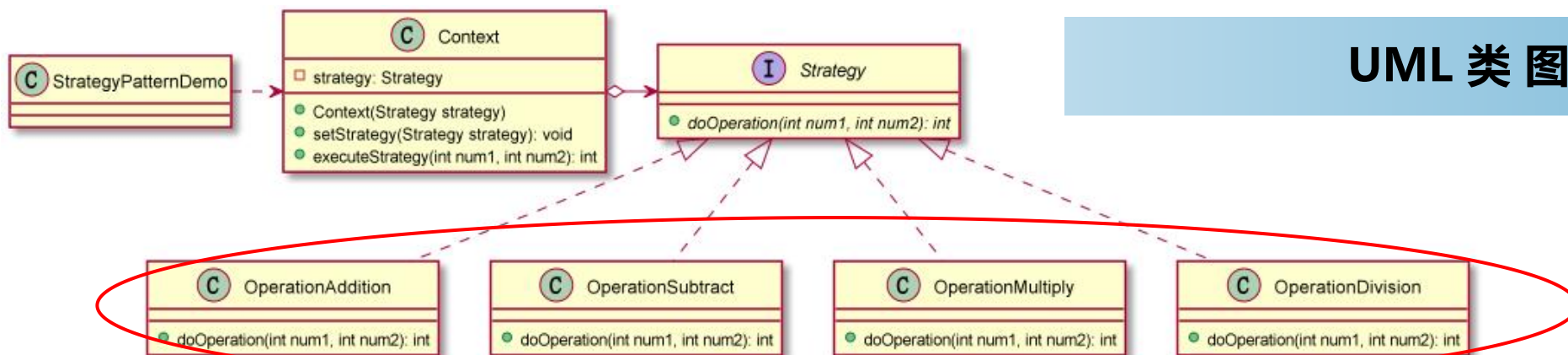
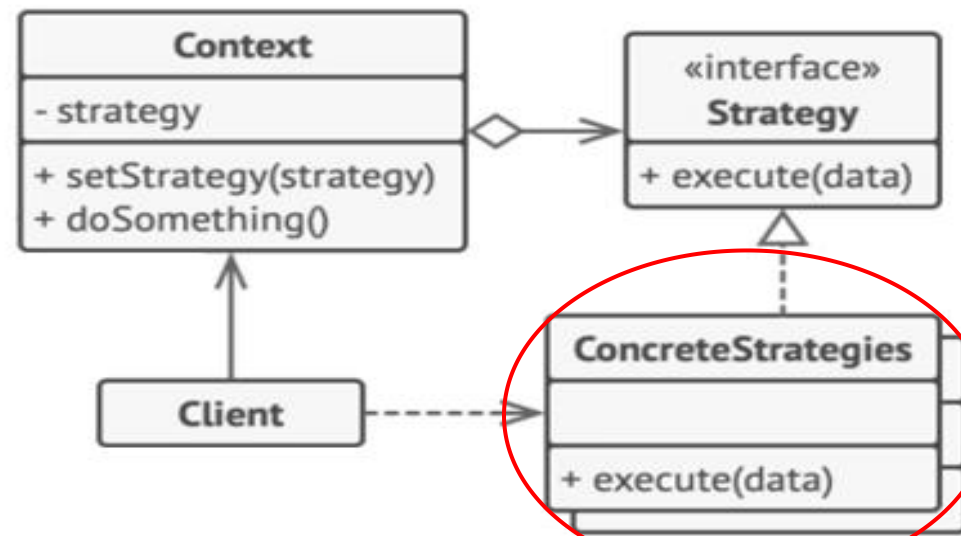
```
public class OperationAddition implements Strategy {  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

加法

```
public class OperationSubtract implements Strategy {  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
}
```

减法

...



### UML 类图设计

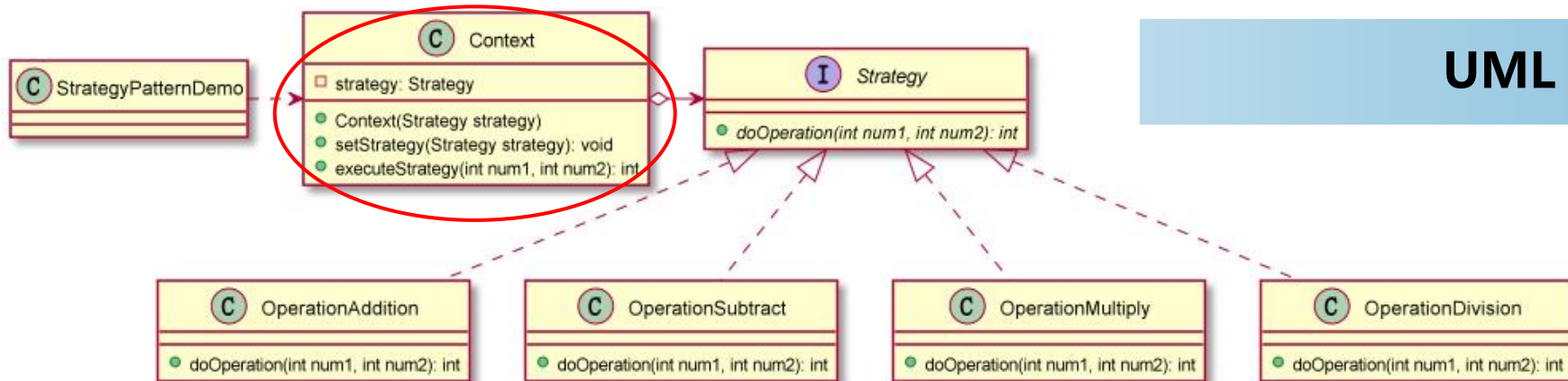
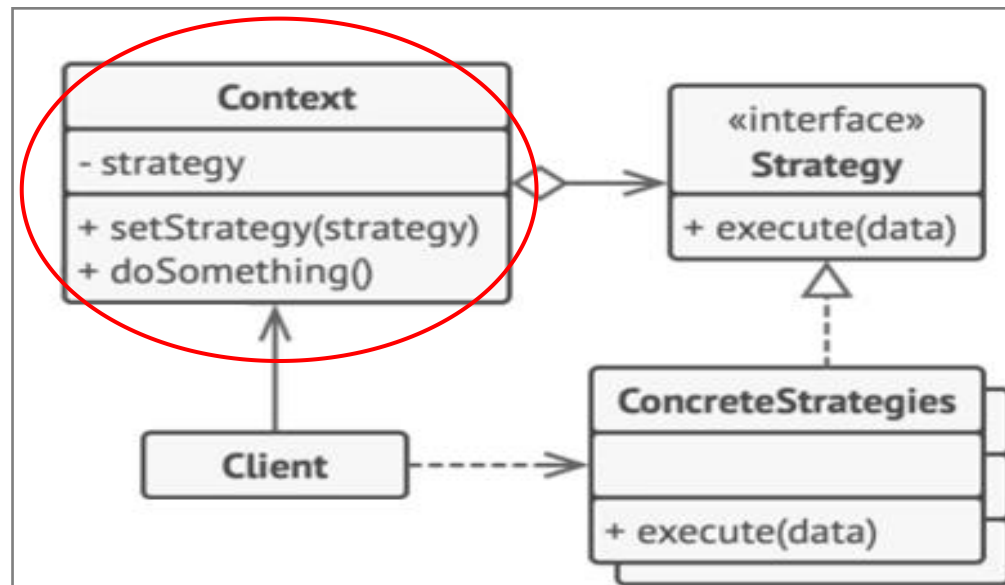


## 实验步骤：计算器举例（3）

### 代码实现

#### ③ 创建Context类，充当上下文角色；

```
public class Context {  
    private Strategy strategy;  
  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
    public void setStrategy(Strategy strategy) {  
        this.strategy = strategy;  
    }  
    public int executeStrategy(int num1, int num2) {  
        return strategy.doOperation(num1, num2);  
    }  
}
```



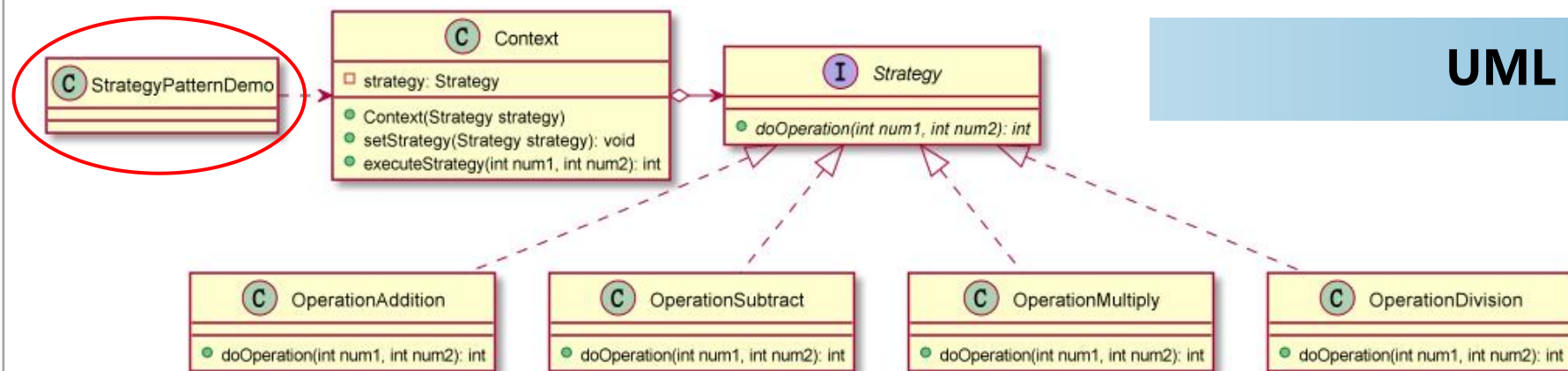
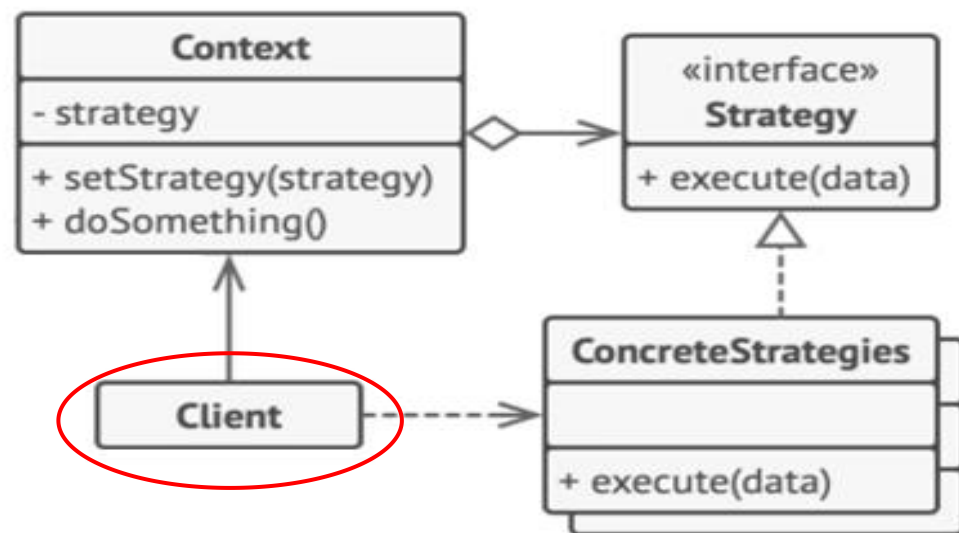
### UML 类图设计

# 实验步骤：计算器举例（4）

## 代码实现

### ④ 客户端使用 Context 来设置不同的策略。

```
public class StrategyPatternDemo {  
    public static void main(String[] args) {  
  
        Context context = new Context(new OperationAddition());  
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));  
  
        context.setStrategy(new OperationSubtract());  
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));  
  
        context.setStrategy(new OperationMultiply());  
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));  
  
        context.setStrategy(new OperationDivision());  
        System.out.println("10 / 5 = " + context.executeStrategy(10, 5));  
    }  
}
```



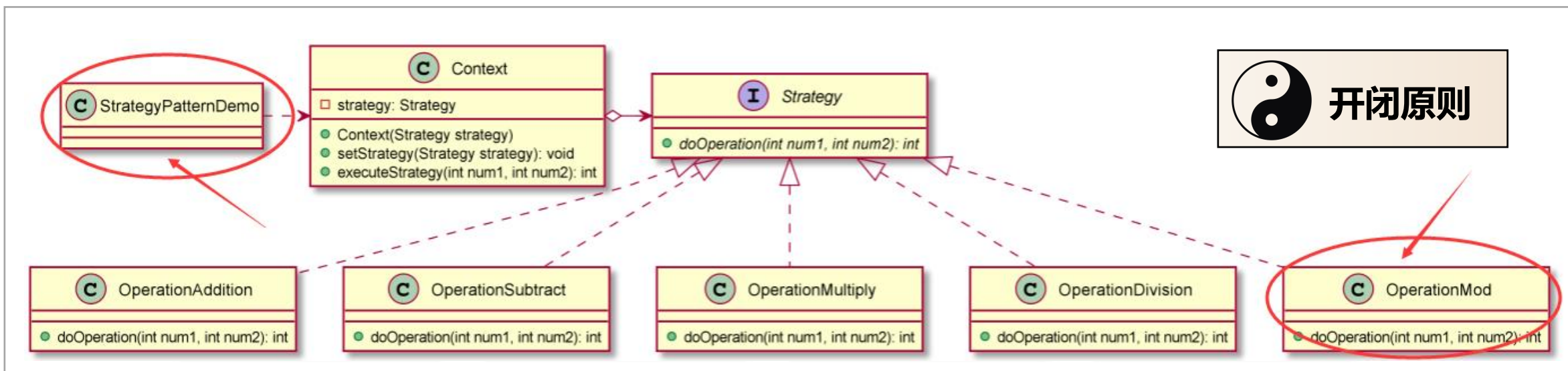
## UML 类图设计

```
10 + 5 = 15  
10 - 5 = 5  
10 * 5 = 50  
10 / 5 = 2
```

## 实验步骤：计算器举例（5）



**请思考：**如何添加一个取模运算？



```
public class StrategyPatternDemo {
    public static void main(String[] args) {

        Context context = new Context(new OperationAddition());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
        ...
        context.setStrategy(new OperationDivision());
        System.out.println("10 / 5 = " + context.executeStrategy(10, 5));

        //切换策略为取模运算
        context.setStrategy(new OperationMod());
        System.out.println("10 % 5 = " + context.executeStrategy(10, 5));
    }
}
```

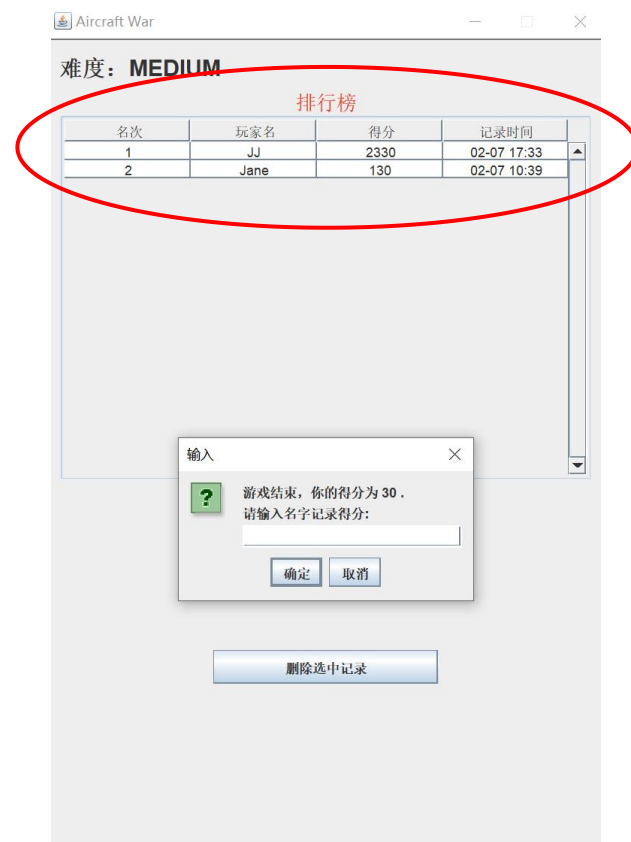
10	+	5	=	15
10	-	5	=	5
10	*	5	=	50
10	/	5	=	2
10	%	5	=	0

## 实验原理：场景分析（2）

### 排行榜 场景 分析

每局游戏记录英雄机得分，游戏结束后，显示该难度的玩家**得分排行榜**。玩家可以删除某条选中记录。

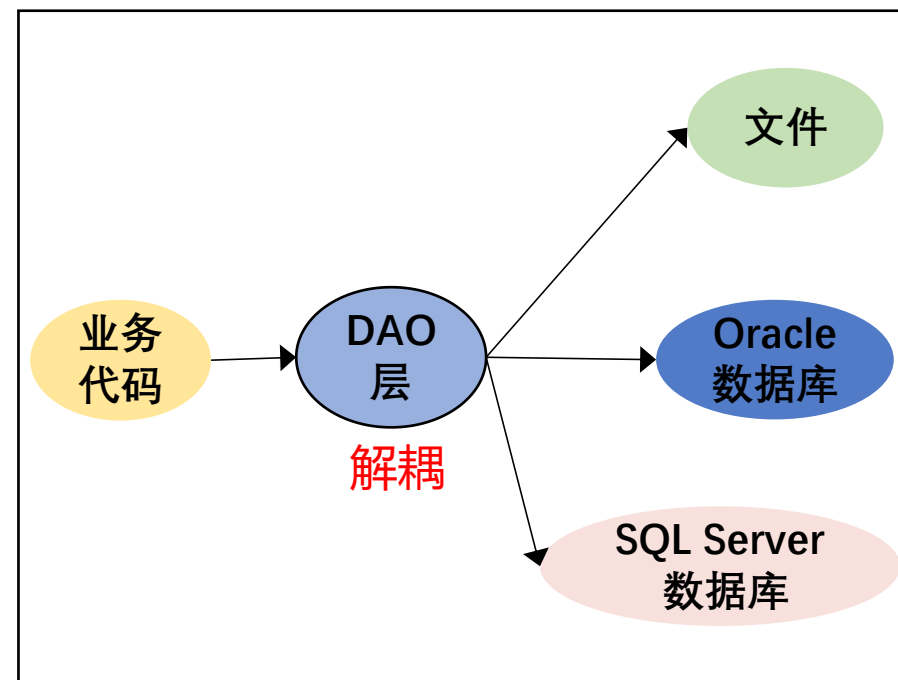
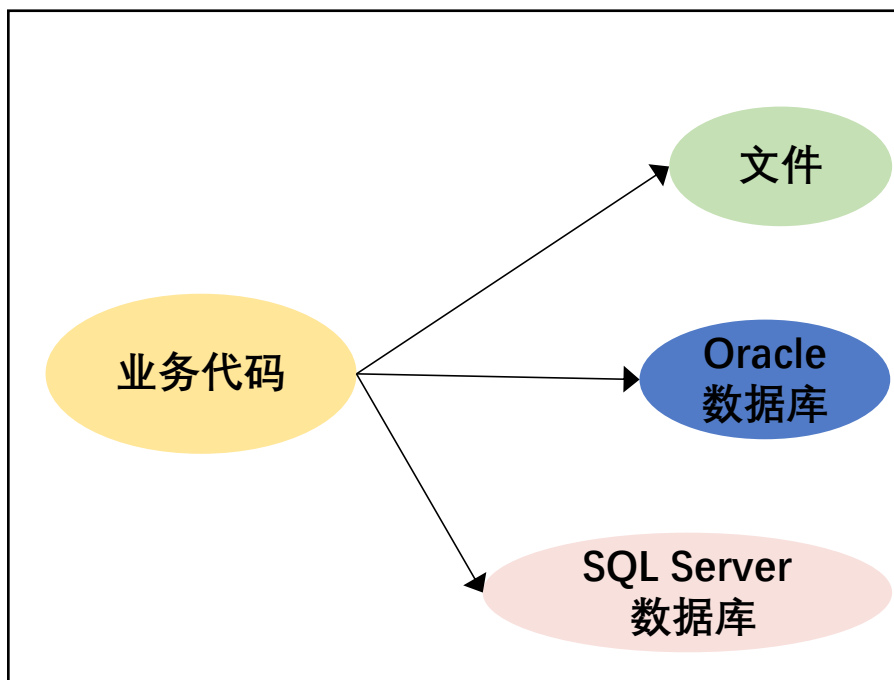
**内容包括：**名次、玩家名、得分和记录时间。



## 实验原理：场景分析（2）



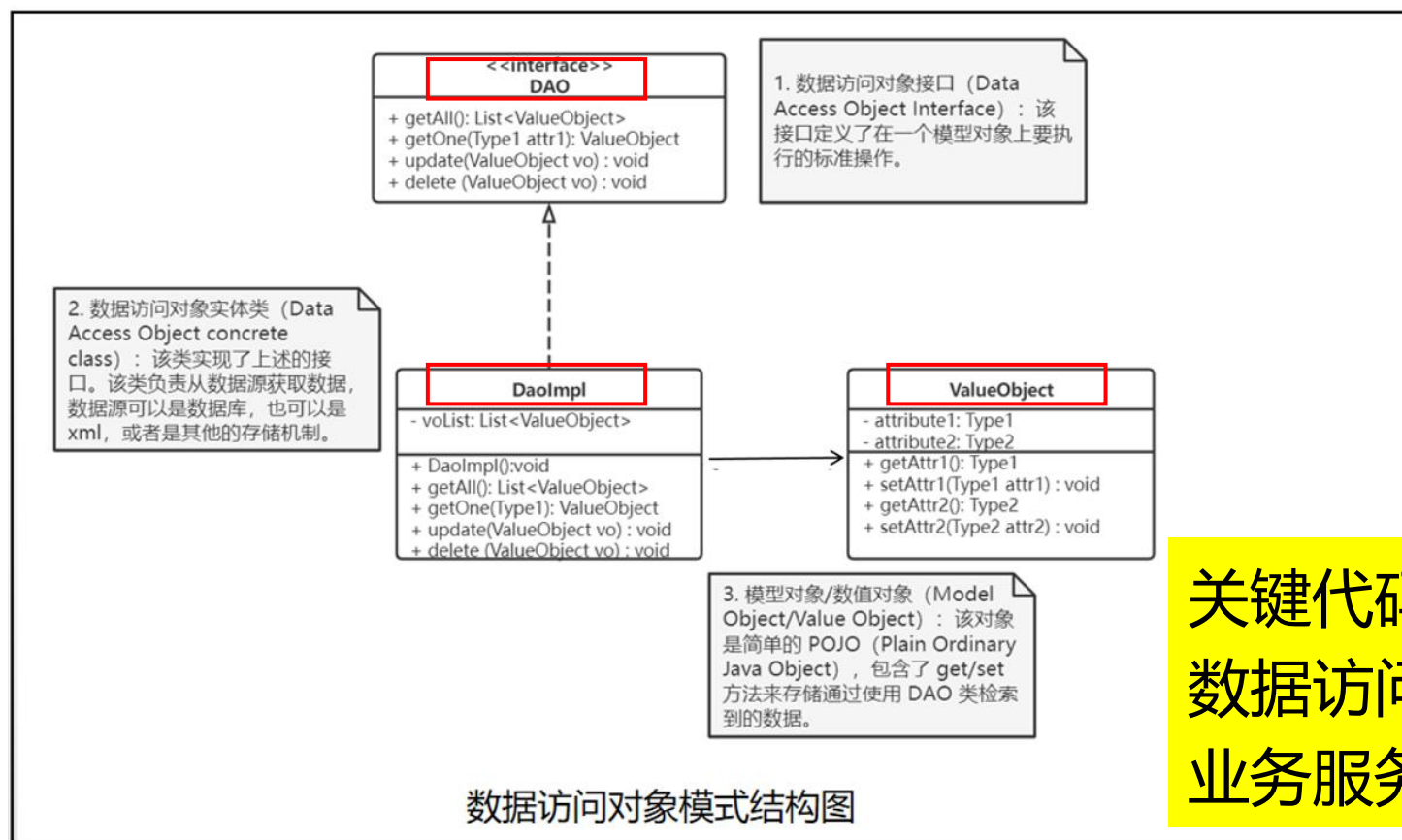
**请思考：**1. 玩家得分数据存储在哪里？如何获取和修改玩家的得分？  
2. 若更换数据源，需要改动哪些类的代码？





# 实验原理：数据访问对象模式结构图

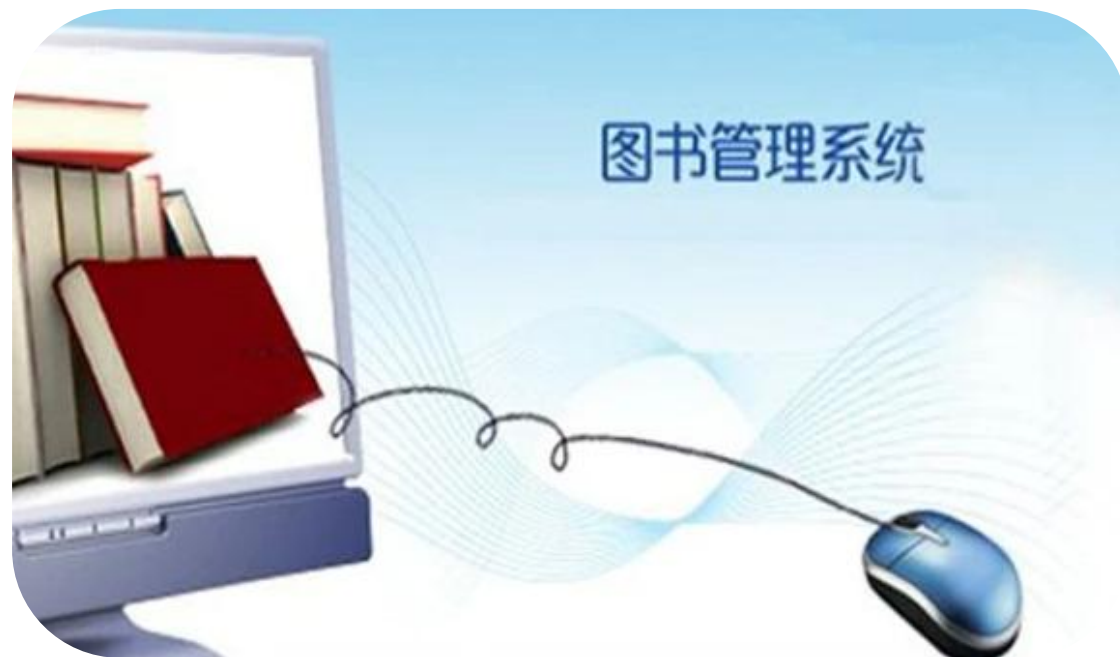
**数据访问对象模式**（Data Access Object Pattern）也叫做 DAO 模式，用于把低级的数据访问 API 或操作从高级的业务服务中分离出来。



关键代码：分离低级的数据访问操作和高级的业务服务

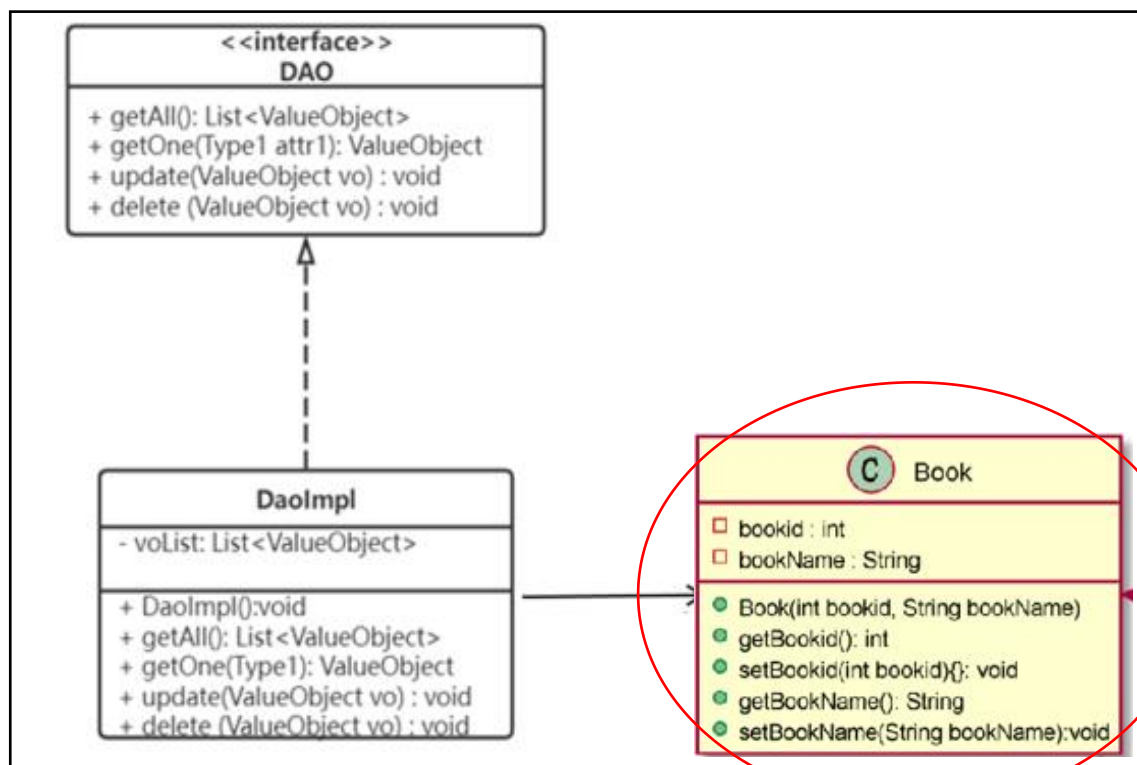
## 实验步骤：图书管理系统举例（数据访问对象模式）

假如我们实现一个**图书管理系统**，实现查询所有图书、按编号查询、增加、删除图书的功能。我们该如何绘制UML类图？



# 实验步骤：图书管理系统举例（1）

## UML 类图设计



### ① 创建一个作为数值对象的实体类Book。

```
public class Book {
    private int bookid;
    private String bookName;

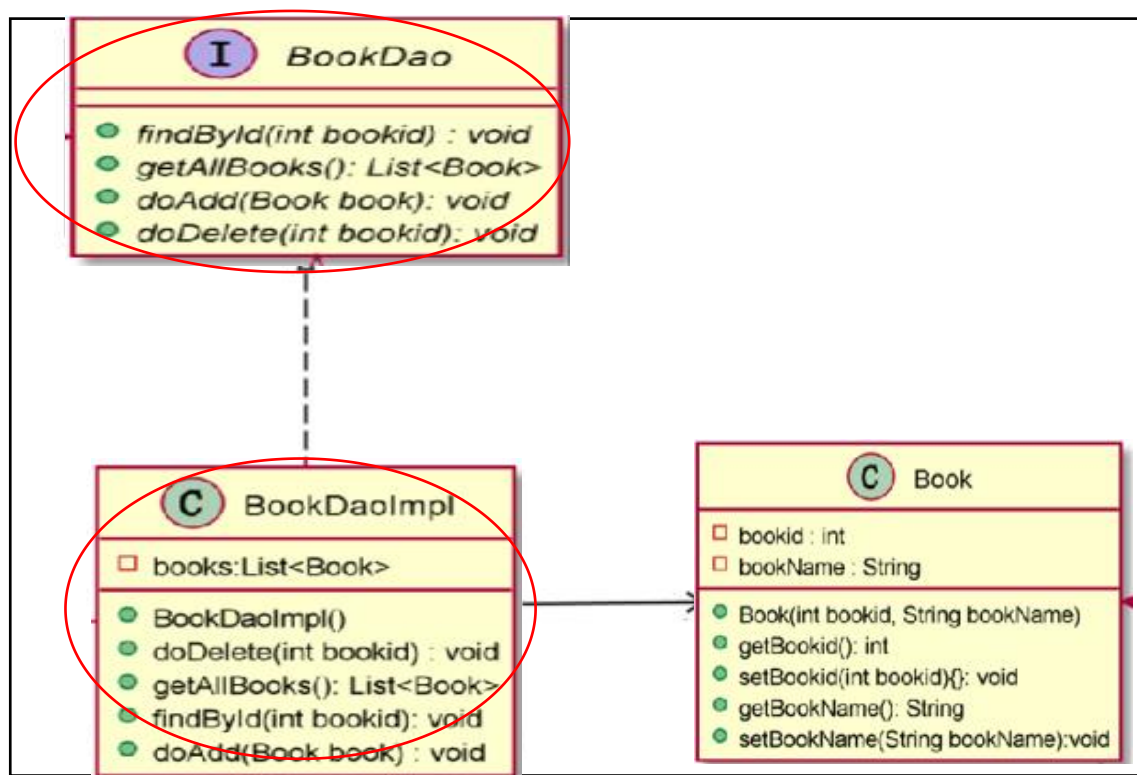
    Book(int bookid, String bookName) {
        this.bookid = bookid;
        this.bookName = bookName;
    }

    public int getBookid() {
        return bookid;
    }
    public void setBookid(int bookid) {
        this.bookid = bookid;
    }
    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
}
```



## 实验步骤：图书管理系统举例（2）

### UML 类图设计



### ② 创建数据访问对象DAO接口;

```
public interface BookDao {  
    void findById(int bookid);  
    List<Book> getAllBooks();  
    void doAdd(Book book);  
    void doDelete(int bookid);  
}
```

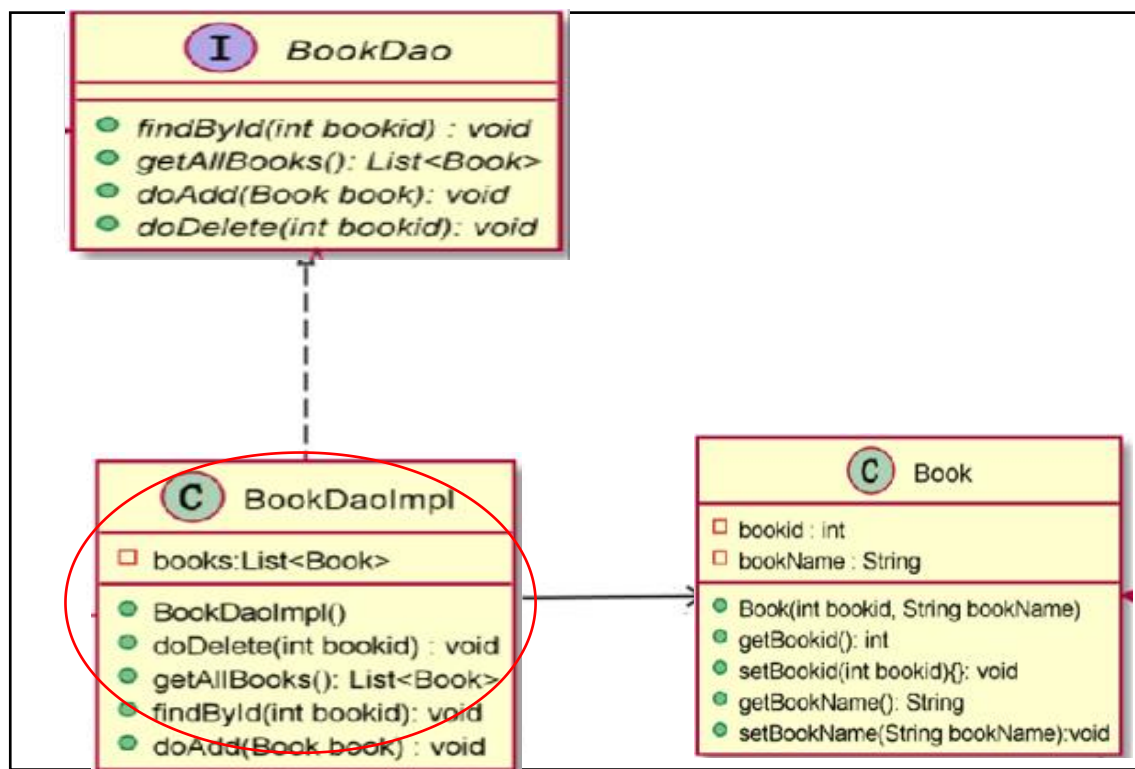
### ③ 创建实现了上述接口的DAO实现类。

```
public class BookDaoImpl implements BookDao {  
    //模拟数据库数据  
    private List<Book> books;  
  
    public BookDaoImpl() {  
        books = new ArrayList<Book>();  
        books.add(new Book(1001, "Clean Code"));  
        books.add(new Book(1002, "Design Patterns"));  
        books.add(new Book(1003, "Effective Java"));  
    }  
  
    //获取所有图书  
    @Override  
    public List<Book> getAllBooks() {  
        return books;  
    }  
}
```

1

## 实验步骤：图书管理系统举例 (3)

### UML 类图设计



### ③ 创建实现了上述接口的DAO实现类。

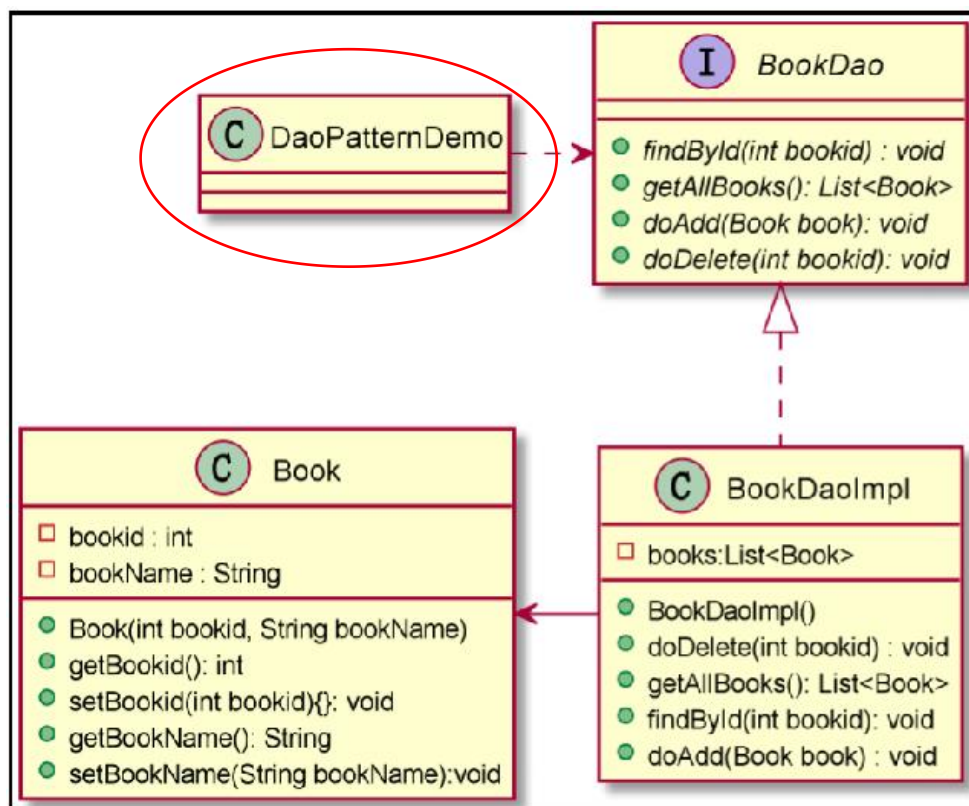
```
//查找图书
@Override
public void findById(int bookid) {
    for (Book item : books) {
        if (item.getBookid() == bookid) {
            System.out.println("Find Book: ID [" + bookid +
                               "], Book Name [" + item.getBookName() + "]);
            return;
        }
    }
    System.out.println("Can not find this book!");
}

//删除图书
@Override
public void doDelete(int bookid) {
    for (Book item : books) {
        if (item.getBookid() == bookid) {
            books.remove(item);
            System.out.println("Delete Book: ID [" + bookid + "]);
            return;
        }
    }
    System.out.println("Can not find this book!");
}

//新增图书
@Override
public void doAdd(Book book) {
    books.add(book);
    System.out.println("Add new Book: ID [" + book.getBookid() +
                       "], Book Name [" + book.getBookName() + "]);
}
```

## 实验步骤：图书管理系统举例（4）

### UML 类图设计



### ④ 使用DaoPatternDemo来演示数据访问对象模式的用法。

```
public class DaoPatternDemo {
    public static void main(String[] args) {

        BookDao bookDao = new BookDaoImpl();

        //输出所有图书
        for (Book book : bookDao.getAllBooks()) {
            System.out.println("Book ID [" + book.getBookid() +
                "], Book Name : [" + book.getBookName() + "]);
        }

        //查找图书
        bookDao.findById(1002);

        //删除图书
        bookDao.doDelete(1002);

        //新增图书
        Book newBook = new Book(1004, "Thinking In java");
        bookDao.doAdd(newBook);

        //输出所有图书
        for (Book book : bookDao.getAllBooks()) {
            System.out.println("Book ID [" + book.getBookid() +
                "], Book Name : [" + book.getBookName() + "]);
        }
    }
}
```

```
Book ID [1001], Book Name : [Clean Code]
Book ID [1002], Book Name : [Design Patterns]
Book ID [1003], Book Name : [Effective Java]



Find Book: ID [1002], Book Name [Design Patterns]

Delete Book: ID [1002]

Add new Book: ID [1004], Book Name [Thinking In java]

Book ID [1001], Book Name : [Clean Code]
Book ID [1003], Book Name : [Effective Java]
Book ID [1004], Book Name : [Thinking In java]
```

# 本次迭代开发的目标 (1)

- ✓ 采用策略模式重构代码，实现直射、散射和环射三种子弹发射弹道；
- ✓ 火力道具  生效后，英雄机弹道由直射切换为散射；
- ✓ 超级火力道具  生效后，英雄机弹道由直射切换为环射（需求变更）；

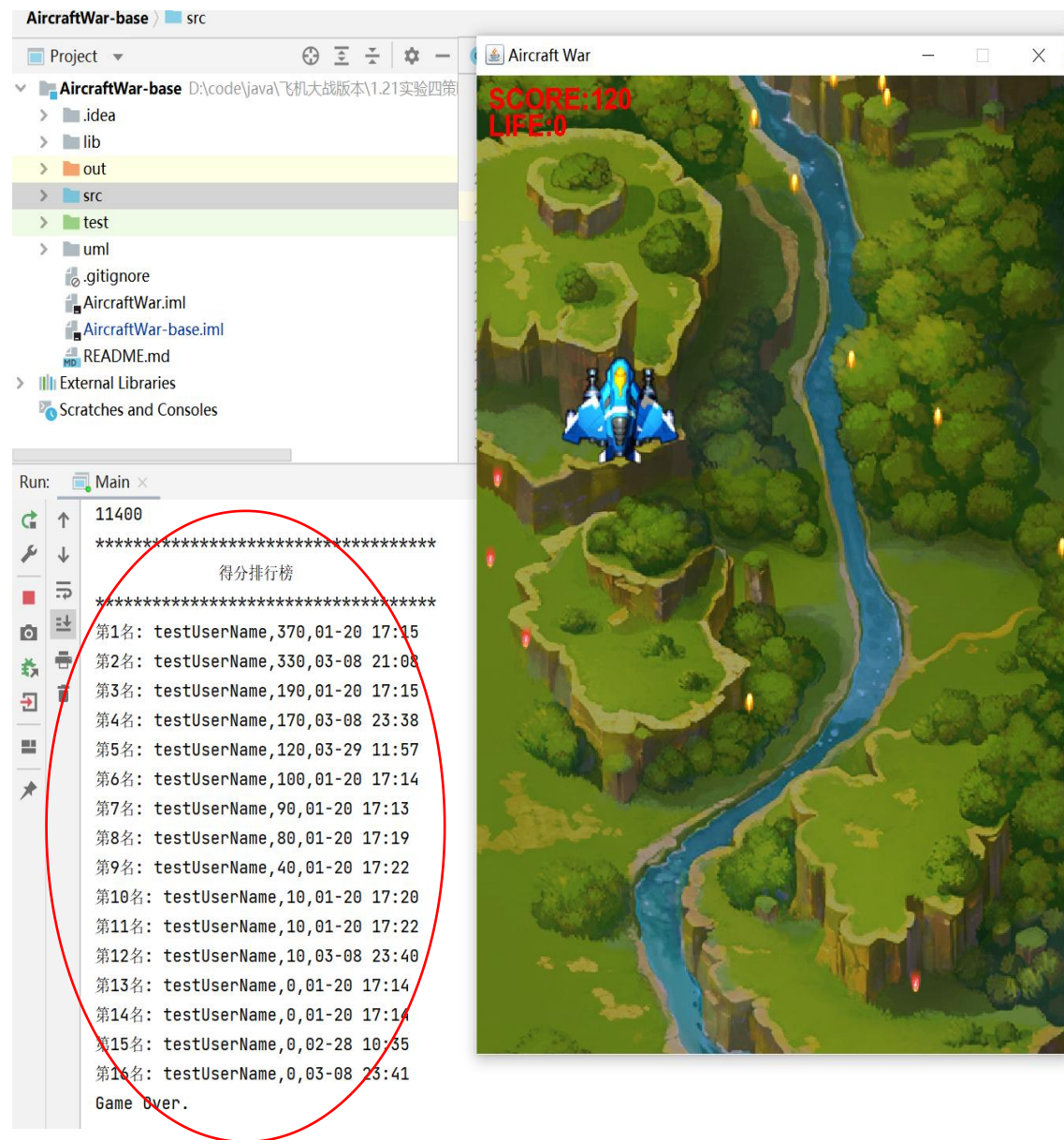
注意：本次实验未涉及多线程，故弹道改变后无法恢复，实验五继续完善即可。





## 本次迭代开发的目标 (2)

- ✓ 每局游戏结束后在控制台打印输出**得分排行榜**，无需实现界面和玩家交互；
- ✓ 得分数据存储在**文件**中。



# 作业提交

- 提交内容

- ① 项目压缩包（整个项目压缩成zip包提交，包含代码、uml图等）
- ② 实验截图报告（设计模式类图和说明，请使用报告模板）

- 截止时间

实验课后一周内提交至HITsz Grader 作业提交平台，具体截止日期参考平台发布。登录网址：： <http://grader.tery.top:8000/#/login>



# 同学们，请开始实验吧！

## THANK YOU