Java 应用与开发课程实验手册 [1]

王晓东
中国海洋大学
信息科学与工程学院计算机科学与技术系

2017 年 10 月 31 日

[1]配套中国海洋大学信息科学与工程学院计算机科学与技术系《Java 应用与开发》课程使用。

目　录

图目录

表目录

序　言

本文档为中国海洋大学信息科学与工程学院计算机科学与技术系《J2EE 应用与开发》实验课程参考资料。

本文档参考了多方书籍、资料和源代码而编写，在此对原始资料的作者和编者表示感谢。但为保证实验授课质量，并没有随本文档公开原始参考资料的出处，望相关作者和编者谅解！

第 1 章　Java 面向对象特性

实验编号 exp01

1.1　实验目的

1. 熟悉 Java 开发工具 Eclipse 的安装和使用。
2. 熟练使用 Maven 创建和管理 Java 工程。
3. 理解面向对象编程的封装、多态、继承等特点。
4. 掌握面向对象技术的编程方法。

1.2　实验要求

实验一（必做）

下载并安装 Java 开发环境 Eclispe，熟练应用 Eclipse。

- 熟悉 Eclipse 的基本菜单项，各类视图。
- 操作创建 Java 项目。
- 操作创建 Java Class、Interface 等。
- 掌握 Eclipse 的程序调试方法。

实验二（必做）

设计一个银行账户类，其中包括：

- 账户信息，如帐号、姓名、开户时间、身份证号码等。
- 存款方法。
- 取款方法。

- 其他方法如“查询余额”和“显示账户信息”等。

用字符界面模拟存款和取款过程。

实验三（必做）

计算三角形面积和周长，要求如下：

- 设计 Point 类（点类）。
- 设计 Triangle 类，至少包括其构造方法，获取面积和周长的方法。
- 通过输入三点坐标来确定一个三角形，并输出三角形的面积、周长等内容。

实验四（选做）

编写程序求解一元多次方程的解，要求如下：

- 至少包括一元一次、一元二次、一元三次方程。
- 至少设计两个接口。
- 必须用到内部类和包。
- 写出详细的运行过程。

1.3 问题分析

实验一问题分析

无。

实验二问题分析

1. 根据题意，设计类 UserInfo 实现对银行账户的封装，其中类的成员变量包含帐号、余额、姓名、开户时间、身份证号等信息，而成员方法主要实现存款、取款、显示等功能，为方便对账户信息进行初始化，需增加类构造方法。
2. 为模拟字符操作界面，另外设计类 BankAccount，以实现对 UserInfo 对象的调用。

实验三问题分析

1. 采用海伦公式求解三角形面积,假定三角形三边为a、b、c,其面积$S = \sqrt{p*(p-a)*(p-b)*(p-c)}$，其中 $p=(a+b+c)/2$。
2. 由于题目中只给出了三角形的三点坐标，因此在应用海伦公式前还需要设计方法求出每条边的长度，假定任意两点的坐标为 (x_1,y_1)、(x_2,y_2)，则其长度 $d=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$。

实验四问题分析

1. 对于一元一次和一元二次方程的求解算法很简单，这里不赘述。
2. 一元三次方程的一种求解算法简单分析如下：

   三次方程的一般形式为

   $$ax^3 + bx^2 + cx + d = 0$$

   两边除以 a 后设 $x = y - \frac{b}{3a}$，就可以转换成 $y^3 + py + q = 0$ 的形式，其中

   $$p = \frac{c}{a} - \frac{b^2}{3a^2}$$

   $$q = \frac{2b^3}{27a^3} - \frac{bc}{3a^2} + \frac{d}{a}$$

   令

   $$y = z - \frac{p}{3z}$$

   得

   $$z^3 - \frac{p^3}{27z^3} + q = 0$$

   即

   $$(z^3)^2 + qz^3 - \frac{p^2}{27} = 0$$

   解这个关于 z^3 的二次方程，得

   $$z^3 = -\frac{q}{2} + \sqrt{(\frac{p}{3})^3 + (\frac{q}{2})^2}$$

   这样就将求解一个一元三次方程转化成求解一个一元二次方程的问题。

1.4 实验过程、步骤及原始记录

实验二

实验过程和代码如下：

```
1  .
2  .
3  .
4  .
5  .
6  .
7  .
8  .
9  .
10 .
```

输出结果参考如下：

output

```
请选择要进行的操作：1. 存款 2. 取款 3. 查询 4. 显示账户 5. 退出
1
请输入要存入的金额（整数）:
1000
存款成功！已存入 1000 元可用余额为 2234 元
请选择要进行的操作：1. 存款 2. 取款 3. 查询 4. 显示账户 5. 退出
2
请输入要取款金额（整数）
400
取款成功！已取出 400 元可用余额为 1834 元
请选择要进行的操作：1. 存款 2. 取款 3. 查询 4. 显示账户 5. 退出
4
帐号：1
姓名：Java
开户时间：Sun Feb 01 00:00:00 CST 2009
身份证号:1111
请选择要进行的操作：1. 存款 2. 取款 3. 查询 4. 显示账户 5. 退出
5
程序退出
```

实验四

实验过程和代码如下：

```
1  .
2  .
3  .
4  .
5  .
6  .
7  .
8  .
9  .
10 .
```

输出结果参考如下：

output

```
求解几次方程？ 1：一次 2：二次 3：三次
3
你选择的是一元三次方程
请输入形如一元三次方程 mx^3+nx^2+tx+s=0 的四个系数
请输入系数 m:
2
请输入系数 n:
3
```

```
请输入系数 t：
4
请输入系数 s：
5
此方程有一个解：
x = -1.3711343313073632
你是否想继续：(y/n)
```

1.5 参考程序

实验二 File: BankAccount.java

```
1  package cn.edu.ouc.javase;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.util.Date;
7
8  class UserInfo {
9    private String id;
10   private String name;
11   private Date createDate;
12   private String idByCard;
13   private long money;
14
15   // 无参构造方法
16   public UserInfo() {
17   }
18
19   // 有参构造方法
20   public UserInfo(String _id, String _name, Date _createDate,
21       String _idByCard, long _money) {
22
23     this.id = _id;
24     this.name = _name;
25     this.createDate = _createDate;
26     this.idByCard = _idByCard;
27     this.money = _money;
28   }
29
30   public long AddMoney(long amount) {
31     money = amount + money;
32     return money;
33   }
34
35   public long DepositMoney(long amount) {
36     money = money - amount;
37     return money;
38   }
39
40   public long getMoney() {
41     return this.money;
42   }
43
44   // 打印人员信息
45   public void showUserInfo() {
```

```
46      System.out.println("帐号：" + id);
47      System.out.println("姓名：" + name);
48      System.out.println("开户时间：" + createDate);
49      System.out.println("身份证号:" + idByCard);
50    }
51  }

53  public class BankAccount {

55    public static void main(String[] args) {
56      UserInfo user = new UserInfo("1", "Java", new Date(109, 1, 1), "1111",
57          1234);
58      BankAccount b = new BankAccount();
59      b.operate(user);
60    }

62    public void operate(UserInfo user) {
63      long saveMoney = 0;
64      long takeMoney = 0;
65      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
66      String choose = "";

68      while (true) {
69        System.out.println("请选择要进行的操作：1. 存款 2. 取款 3. 查询"
70                           + " 4. 显示账户 5. 退出");

72        try {
73          choose = br.readLine();
74        } catch (IOException e) {
75          e.printStackTrace();
76        }

78        // 存款
79        if (choose.compareTo("1") == 0) {
80          while (true) {
81            System.out.println("请输入要存入的金额（整数）：");

83            try {
84              saveMoney = Long.parseLong(br.readLine());
85            } catch (NumberFormatException e) {
86              System.out.println("输入错误，请输入整数");
87              continue;
88            } catch (IOException e) {
89            }

91            long balance = user.AddMoney(saveMoney);
92            System.out.println("存款成功！已存入" + saveMoney + "元" + "可用余额为"
93                + balance + "元");
94            break;
95          }
96        }
```

```
98          // 取款
99          if (choose.compareTo("2") == 0) {
100           while (true) {
101             System.out.println("请输入要取款金额（整数）");

103             try {
104               takeMoney = Long.parseLong(br.readLine());

106             } catch (NumberFormatException e) {
107               System.out.println("输入错误，请输入整数");
108               continue;
109             } catch (IOException e) {

111             }

113             if (!(user.getMoney() < takeMoney)) {
114               long balance = user.DepositMoney(takeMoney);
115               System.out.println("取款成功！已取出" + takeMoney + "元"
116                   + "可用余额为" + balance + "元");
117               break;
118             } else {
119               System.out.println("余额不足，请重新输入");
120             }
121           }
122         }

124         // 查询
125         if (choose.compareTo("3") == 0) {
126           System.out.println("你的余额为" + user.getMoney() + "元");
127         }

129         // 显示账户
130         if (choose.compareTo("4") == 0) {
131           user.showUserInfo();
132         }
133         // 退出
134         if (choose.compareTo("5") == 0) {
135           System.out.println("程序退出");
136           System.exit(0);
137         }
138       }
139     }
140   }
```

实验四 File: SolveEquation.java

```
1  package cn.edu.ouc.javase;

3  import java.io.IOException;
```

```
4  import java.util.Scanner;
5
6  public class SolveEquation {
7
8    void print() throws IOException {
9      char ch = 'y';
10     System.out.println("求解几次方程？ 1: 一次 2: 二次 3: 三次");
11     Scanner sLine = new Scanner(System.in);
12     int pm = sLine.nextInt();
13
14     if (pm == 3) {
15       System.out.println("你选择的是一元三次方程");
16       SolveEquation.SolveCubicEquation fc = new SolveEquation().new SolveCubicEquation();
17       fc.SolveCubicEquation();
18     } else {
19       print();
20     }
21
22     System.out.println("你是否想继续：(y/n)");
23     ch = (char) System.in.read();
24     System.in.skip(2);
25
26     if (ch == 'y') {
27       print();
28     } else if (ch == 'n') {
29       System.out.println("Good luck!");
30     } else {
31       print();
32     }
33   }
34
35   public static void main(String[] args) throws IOException {
36     SolveEquation se = new SolveEquation();
37     se.print();
38   }
39
40   // 接口 规划求解一元三次方式必须实现的方法
41   interface ItCubicEquation {
42     void SolveCubicEquation();
43   }
44
45   class SolveCubicEquation implements ItCubicEquation {
46
47     @Override
48     public void SolveCubicEquation() {
49       System.out.println("请输入形如一元三次方程 mx^3+nx^2+tx+s=0 的四个系数");
50       Scanner sce = new Scanner(System.in);
51       System.out.println("请输入系数 m：");
52       double m = sce.nextDouble();
53       System.out.println("请输入系数 n：");
54       double n = sce.nextDouble();
```

```
55          System.out.println("请输入系数 t：");
56          double t = sce.nextDouble();
57          System.out.println("请输入系数 s：");
58          double s = sce.nextDouble();

60          if (m == 0) {
61            System.out.println("输入错误");
62          } else {
63            double a = n / m;
64            double b = t / m;
65            double c = s / m;
66            double q = (a * a - 3 * b) / 9;
67            double r = (2 * a * a * a - 9 * a * b + 27 * c) / 54;

69            if (r * r < q * q * q) {
70              System.out.println("此方程有三个解：");
71              t = Math.acos(r / Math.sqrt(q * q * q));
72              double x1 = -2 * Math.sqrt(q) + Math.cos(t / 3) - a / 3;
73              double x2 = -2 * Math.sqrt(q)
74                  + Math.cos((t + 2 * Math.PI) / 3) - a / 3;
75              double x3 = -2 * Math.sqrt(q)
76                  + Math.cos((t - 2 * Math.PI) / 3) - a / 3;

78              System.out.println("x1 = " + x1 + ", x2 = " + x2
79                  + ", x3 = " + x3);
80            } else {
81              System.out.println("此方程有一个解：");
82              int sgn = (r >= 0) ? 1 : -1;
83              double u = -sgn
84                  * Math.pow(
85                      (Math.abs(r) + Math.sqrt(r * r - q * q * q)),
86                      1.0 / 3);
87              double v = (u != 0) ? q / u : 0;
88              double x1 = u + v - a / 3;
89              System.out.println("x = " + x1);
90            }
91          }
92        }
93      }
94  }
```

第 2 章　模拟扑克牌游戏的洗牌

实验编号 exp02

2.1　实验要求

结合面向对象设计原则，分析和设计模拟扑克牌游戏的洗牌过程。

1. 编程定义一个表示扑克牌的类 Poke，用 suit 来表示扑克牌的花色，用 face 来表示扑克牌的牌面值，注意别忘了大小王，每副牌为 54 张。
2. 假设扑克牌游戏需要 2 付 (或 2 付以上）的牌，请编程实现模拟洗牌，以及将这些扑克牌分给参加游戏的所有人，并将参加扑克牌游戏的所有人的扑克牌输出。
3. 实现 Poke 类中包含的 4 个静态方法，所实现的 Poke 类在 PokeTest 的 main 方法中进行测试，请阅读 main 方法的代码，并根据其中逻辑猜测 Poke 类中静态方法的行为并编写方法实现代码。

Poke.java

```
1  package ouc.cs.course.java.test.poke;
2  import java.util.Random;
3
4  public class Poke {
5    public static String[] createCard(int number) {
6    }
7
8    public static void display(String[] cards) {
9    }
10
11   public static void shuffle(String[] cards) {
12   }
13
14   public static void distribute(String[] cards,int player) {
15   }
16 }
```

PokeTest.java

```
package ouc.cs.course.java.test.poke;

import java.util.Scanner;
import ouc.cs.course.java.test.poke.Poke;

public class PokeTest {

  @SuppressWarnings("resource")
  public static void main(String[] args) {
    System.out.println("该扑克游戏需要几付扑克牌？");
    Scanner sc = new Scanner(System.in);
    int num = sc.nextInt();
    String[] cards = Poke.createCard(num);
    System.out.println("该扑克游戏有几个玩家？");
    Scanner sp = new Scanner(System.in);
    int per = sp.nextInt();
    System.out.println("\n显示所有的牌：");
    Poke.display(cards);
    Poke.shuffle(cards);
    System.out.println("\n显示分配给每个人的牌：");
    Poke.distribute(cards, per);

  }
}
```

4. 要随机产生某个范围内的整数，可以用 java.util.Random 类的 nextInt(int num)。例如，输出 0 - 53 范围内的一个整数如下：

```
Random rand = new Random();
int num = rand.nextInt(54);
```

2.2 实验过程、步骤及原始记录

实验过程和代码如下：

```
.
.
.
.
.
.
.
.
.
.
```

2.3 参考程序

File: Poke.java

```
1  package ouc.cs.course.java.test.poke;
2
3  import java.util.Random;
4
5  public class Poke {
6    public static String[] createCard(int number) {
7      if (number < 2)
8        number = 2;
9      int cardNumber = number * 52;
10     String[] cards = new String[cardNumber];
11     int suit;
12     int face;
13     for (int i = 0; i < cardNumber; i++) {
14       cards[i] = new String();
15       suit = i % 4;
16       switch (suit) {
17       case 0:
18         cards[i] = "红桃";
19         break;
20       case 1:
21         cards[i] = "黑桃";
22         break;
23       case 2:
24         cards[i] = "方块";
25         break;
26       case 3:
27         cards[i] = "梅花";
28         break;
29       }
30       face = i % 13;
31       switch (face) {
32       case 0:
33         cards[i] = cards[i] + "A";
34         break;
35       case 1:
36         cards[i] = cards[i] + "2";
37         break;
38       case 2:
39         cards[i] = cards[i] + "3";
40         break;
41       case 3:
42         cards[i] = cards[i] + "4";
43         break;
44       case 4:
45         cards[i] = cards[i] + "5";
46         break;
```

```
47          case 5:
48            cards[i] = cards[i] + "6";
49            break;
50          case 6:
51            cards[i] = cards[i] + "7";
52            break;
53          case 7:
54            cards[i] = cards[i] + "8";
55            break;
56          case 8:
57            cards[i] = cards[i] + "9";
58            break;
59          case 9:
60            cards[i] = cards[i] + "10";
61            break;
62          case 10:
63            cards[i] = cards[i] + "J";
64            break;
65          case 11:
66            cards[i] = cards[i] + "Q";
67            break;
68          case 12:
69            cards[i] = cards[i] + "K";
70            break;
71          }
72        }
73        return cards;
74      }

76      public static void display(String[] cards) {
77        for (int i = 0; i < cards.length; i++) {
78          System.out.printf("%-7s", cards[i]);
79          if ((i + 1) % 13 == 0)
80            System.out.println();
81        }
82      }

84      public static void shuffle(String[] cards) {
85        Random r = new Random();
86        String tmp = null;
87        for (int i = 0; i < cards.length; i++) {
88          int k1 = r.nextInt(cards.length);
89          int k2 = r.nextInt(cards.length);
90          tmp = cards[k1];
91          cards[k1] = cards[k2];
92          cards[k2] = tmp;
93        }
94      }

96      public static void distribute(String[] cards, int player) {
97        int num = cards.length / player;
```

```
98      for (int j = 0; j < cards.length; j++) {
99        System.out.printf("%-7s", cards[j]);
100       if ((j + 1) % 13 == 0)
101         System.out.println();
102       if ((j + 1) % num == 0) {
103         System.out.println("第" + (j + 1) / num + "个人的牌如上");
104         System.out.println();
105       }
106     }
107   }
108 }
```

第 3 章　设计模式: 观察者（Observer）模式

实验编号 exp03

3.1　实验目的

1. 理解设计模式之观察者（Observer）模式。[1]
2. 理解模式中所包含的 Java 设计原则。
3. 基于观察者（Observer）模式编写 Java 程序，加深对观察者模式的理解。
4. 学习使用 Java 自带的观察者模式实现。

3.2　需求分析

假设我们团队获得了一项合约，是需求方 Weather-O-Rama 公司委托我们开发一个气象站应用。首先来看一下需求方 Weather-O-Rama 公司对气象站应用的需求概况（如图 3.1）。其中，WeatherData 对象用于从气象站获取温度、湿度和气压数据，并驱动多个显示装置显示。合约中要求至少包括目前状况、气象统计、天气预报的显示布告板。同时，需求方所给的 WeatherData 类参考设计如下。

Class: WeatherData

```
1  public class WeatherData {
2    public float getTemperature() {
3      // 从气象站获取温度数据，由需求方提供方法
4    }
5    public float getHumidity() {
6      // 从气象站获取湿度数据，由需求方提供方法
7    }
8    public float getPressure() {
9      // 从气象站获取气压数据，由需求方提供方法
10   }
11   public void measurementsChanged() {
```

[1]接下来两章分别讲解并示范两个 Java 经典的设计模式。设计这两个章节的目的是：对 Java 中接口、抽象类、继承、多态等概念的更进一步理解，它们有什么用？搞得这么复杂干什么？有些东西不写代码是学不来的，也体会不到。

```
    // 当气象数据值发生改变时，该方法被期望调用
  }
}
```

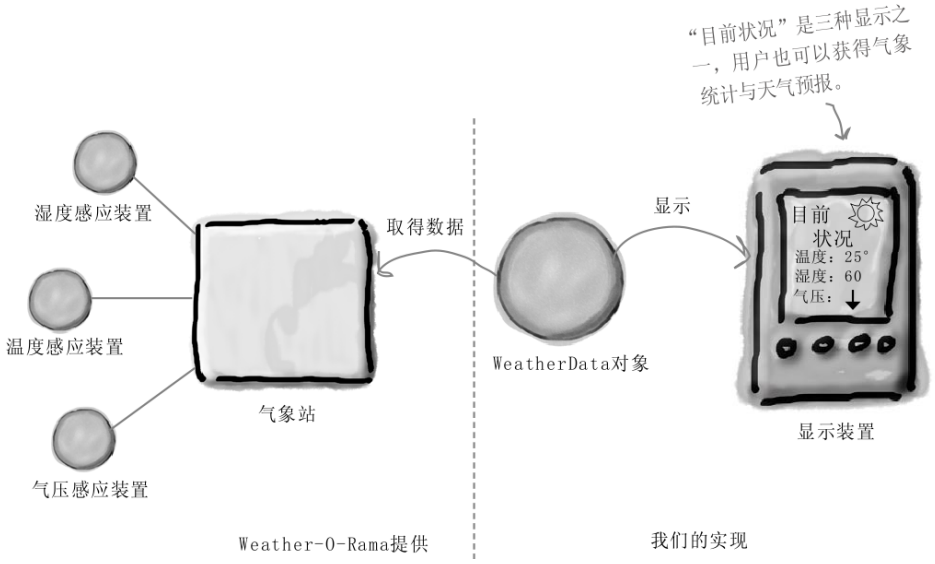

图 3.1: 气象站应用的架构

根据任务合约中的内容和需求方提供的 WeatherData 类参考，你一般会这样设计这个气象观测站：在 measurementsChanged 方法中添加了功能代码，实现了对三个布告板的数据更新，如下所示。

```
public class WeatherData {
  // 实例变量声明

  public void measurementsChanged() {
    float temp = getTemperature(); // 调用取得三种数据的方法
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
  }
  // 这里是其他的 WeatherData 方法
}
```

上述实现方案非常不佳，我们来分析一下并从如下选项中找出上述实现所存在的问题：

- 针对具体实现编程，而非针对接口编程 ❒
- 对于每添加一个布告板，都得修改代码 ❒
- 无法在运行时动态增加（删除）布告板 ❒

- 布告板没有实现一个共同的接口 ❐
- 尚未封装改变的部分 ❐
- 侵犯了 WeatherData 类的封装 ❐

所以，本实验将采取观察者（Observer）模式来实现气象站应用。根据 UML 类图 3.2，实现气象站，要求能够避免上述分析中旧设计所存在的问题。

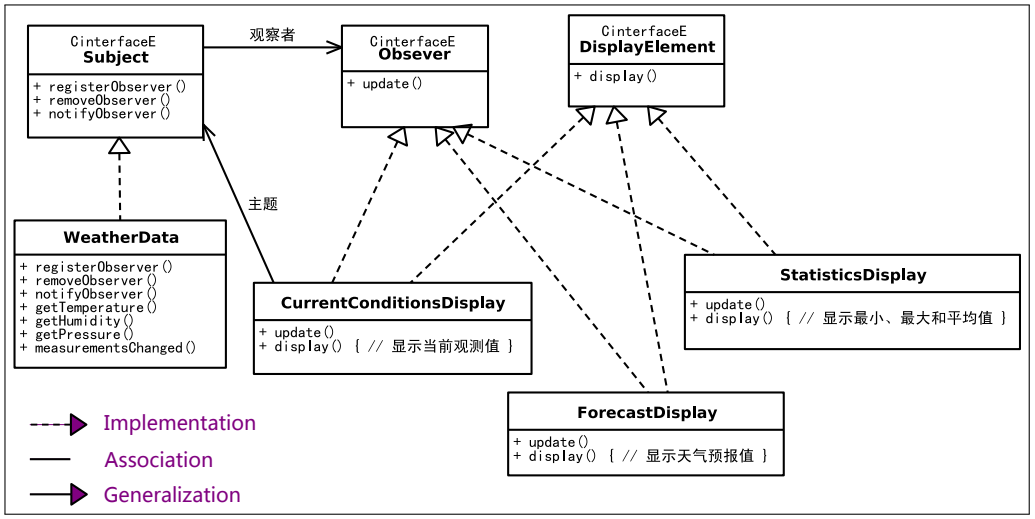

图 3.2: 气象站 UML 类图

3.3 问题分析

3.3.1 观察者模式定义

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新，如图 3.3。

在现实世界中，我们可以用报纸订阅服务来类比观察者模式。

3.3.2 松耦合

设计原则

为了交互对象之间的松耦合设计而努力。

两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

- 关于观察者的一切，主题只知道观察者实现了某个接口（也就是 Observer 接口）。主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。

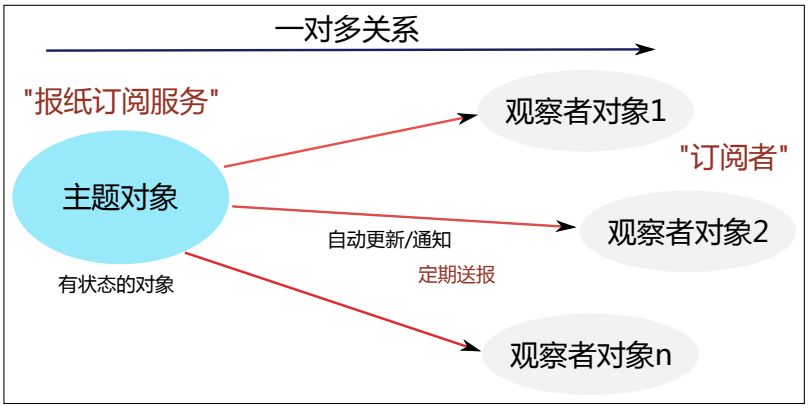

图 3.3: 观察者模式

- 任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现 Observer 接口的对象列表，所以我们可以随时增加观察者。事实上，在运行时我们可以用新的观察者取代现有的观察者，主题不会受到任何影响。同样的，也可以在任何时候删除某些观察者。
- 有新类型的观察者出现时，主题的代码不需要修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所有要做的就是在新的类里实现此观察者接口，然后注册为观察者即可。主题不在乎别的，它只会发送通知给所有实现了观察者接口的对象。
- 我们可以独立地复用主题或观察者。如果我们在其他地方需要使用主题或观察者，可以轻易地复用，因为二者并非紧耦合。
- 改变主题或观察者其中一方，并不会影响另一方。因为两者是松耦合的，所以只要他们之间的接口仍被遵守，我们就可以自由地改变他们。

3.4 实验过程、步骤及原始记录

实验过程和代码如下：

```
1 .
2 .
3 .
4 .
5 .
6 .
```

输出结果参考如下（仅实现了当前气象状况布告板，另外的布告板请自行添加并完成实验报告）：

output

```
Current conditions: 80.0F degrees and 65.0% humidity
Current conditions: 78.0F degrees and 90.0% humidity
```

3.5 参考程序

本实验为学习验证性实验，直接给出参考程序，希望大家认真学习理解观察者模式的用法以及其中包含的设计哲学，并尝试增加其他类型的气象布告板，数据人工模拟即可。

File: Subject.java

```
package ouc.cs.java.desingpattern.observer;

public interface Subject {
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}
```

File: Observer.java

```
package ouc.cs.java.desingpattern.observer;

public interface Observer {
  public void update(float temperature, float humidity, float pressure);
}
```

File: DisplayElement.java

```
package ouc.cs.java.desingpattern.observer;

public interface DisplayElement {
  public void display();
}
```

File: WeatherData.java

```
package ouc.cs.java.desingpattern.observer;

import java.util.ArrayList;

public class WeatherData implements Subject {

  private ArrayList observers;

  private float temperature;
  private float humidity;
  private float pressure;

  public WeatherData() {
    observers = new ArrayList();
  }

  @Override
  public void registerObserver(Observer o) {
    observers.add(o);
```

```
20    }
22    @Override
23    public void removeObserver(Observer o) {
24      int i = observers.indexOf(o);
25      if (i >= 0) {
26        observers.remove(i);
27      }
28    }
30    @Override
31    public void notifyObservers() {
32      for (int i = 0; i < observers.size(); i++) {
33        Observer observer = (Observer) observers.get(i);
34        observer.update(temperature, humidity, pressure);
35      }
36    }
38    public void measurementsChanged() {
39      notifyObservers();
40    }
42    public void setMeasurements(float temperature, float humidity, float pressure) {
43      this.temperature = temperature;
44      this.humidity = humidity;
45      this.pressure = pressure;
46      measurementsChanged();
47    }
48  }
```

File: CurrentConditionsDisplay.java

```
1   package ouc.cs.java.desingpattern.observer;
3   public class CurrentConditionsDisplay implements Observer, DisplayElement {
5     private float temperature;
6     private float humidity;
7     private Subject weatherData;
9     public CurrentConditionsDisplay(Subject weatherData) {
10      this.weatherData = weatherData;
11      weatherData.registerObserver(this);
12    }
14    @Override
15    public void update(float temperature, float humidity, float pressure) {
16      this.temperature = temperature;
17      this.humidity = humidity;
18      display();
19    }
```

```
21    @Override
22    public void display() {
23      System.out.println("Current conditions: " + temperature
24      + "F degrees and " + humidity + "% humidity");
25    }
26  }
```

File: WeatherStation.java

```
1   package ouc.cs.java.desingpattern.observer;
2
3   public class WeatherStation {
4
5     public static void main(String[] args) {
6       WeatherData weatherData = new WeatherData();
7       CurrentConditionsDisplay cd = new CurrentConditionsDisplay(weatherData);
8
9       weatherData.setMeasurements(80, 65, 30.4f);
10      weatherData.setMeasurements(78, 90, 29.2f);
11    }
12  }
```

3.6 Java 自带的观察者模式

由于观察者模式的广泛应用性，Java API 自带了观察者模式的实现。其中，Observable 对象是被观察者，Observer 对象是观察者。使用 Java 自带的 API 实现观察者模式非常简单：

- 创建被观察者类（即 Subject），它继承自 java.util.Observable 类；
- 创建观察者类，它实现 java.util.Observer 接口；

3.6.1 被观察者类

使用被观察者类的 addObserver() 方法把观察者对象添加到观察者对象列表中。当被观察者中的事件发生变化时执行 setChanged() 和 notifyObservers() 方法。

- setChange() 方法用来设置一个内部标志位注明数据发生了变化；
- notifyObservers() 方法会去调用观察者对象列表中所有的 Observer 的 update() 方法，通知它们数据发生了变化。

注意：只有在 setChange() 被调用后，notifyObservers() 才会去调用 update()。

3.6.2 观察者类

观察者类需要实现 Observer 接口的唯一方法 update()。

```
1 void update(Observable o, Object arg);
```

形参 Object arg 对应一个由 notifyObservers(Object arg) 传递来的参数，当执行的是 notifyObservers() 时，arg 为 null。

3.6.3 示例代码

被观察者 ServerManager

```
1  package ouc.cs.java.test.observer;
2
3  import java.util.Observable;
4
5  /**
6  * 被观察者
7  *
8  */
9  public class ServerManager extends Observable {
10
11   private int data = 0;
12
13   public int getData() {
14     return data;
15   }
16
17   public void setData(int i) {
18
19     /*
20     * 只有在setChanged()被调用后，notifyObservers()才会去调用update()，否则什么都不干
21     */
22     if (this.data != i) {
23       this.data = i;
24       setChanged();
25     }
26     notifyObservers();
27   }
28 }
```

观察者 ObserverA

```
1  package ouc.cs.java.test.observer;
2
3  import java.util.Observable;
4  import java.util.Observer;
5
6  /**
7  * 观察者A
8  *
9  */
10 public class ObserverA implements Observer {
```

```
12      public ObserverA(ServerManager sm) {
13        super();
14        sm.addObserver(this);
15      }
16
17      public void update(Observable arg0, Object arg1) {
18        System.out.println("ObserverA receive: Data has changed to " + ((ServerManager) arg0).getData());
19      }
20    }
```

观察者 ObserverB

```
1   package ouc.cs.java.test.observer;
2
3   import java.util.Observable;
4   import java.util.Observer;
5
6   /**
7   * 观察者B
8   *
9   */
10  public class ObserverB implements Observer {
11
12    public ObserverB(ServerManager sm) {
13      super();
14      sm.addObserver(this);
15    }
16
17    public void update(Observable arg0, Object arg1) {
18      System.out.println("ObserverB receive: Data has changed to " + ((ServerManager) arg0).getData());
19    }
20  }
```

测试主程序 TestJavaObserver

```
1   package ouc.cs.java.test.observer;
2
3   public class TestJavaObserver {
4
5     public static void main(String[] args) {
6
7       ServerManager sm = new ServerManager();
8       ObserverA a = new ObserverA(sm);
9       ObserverB b = new ObserverB(sm);
10      sm.setData(5);
11      sm.setData(8);
12
13      /*
14      * 注销观察者，以后被观察者有数据变化就不再通知这个已注销的观察者
15      */
```

```
16          sm.deleteObserver(a);
17          sm.setData(10);
18      }
19  }
```

第 4 章　设计模式: 工厂（Factory）模式

实验编号 exp04

4.1　知识拓展：设计模式概述

设计模式是一套被反复使用、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

4.1.1　设计模式体现的六大原则

1. 开闭原则（Open Close Principle）
   开闭原则即是对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

2. 里氏代换原则（Liskov Substitution Principle）
   里氏代换原则（LSP）面向对象设计的基本原则之一，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充，实现开闭原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3. 依赖倒转原则（Dependence Inversion Principle）
   依赖倒转原则是开闭原则的基础，具体内容包括：针对接口编程，依赖于抽象而不依赖于具体。

4. 接口隔离原则（Interface Segregation Principle）
   接口隔离原则是使用多个隔离的接口，比使用单个接口要好。

5. 迪米特法则（最少知道原则）（Demeter Principle）
   迪米特法则（最少知道原则）就是说一个实体应当尽量少的与其他实体之间发生相互作用，

使得系统功能模块相对独立。

6. 合成复用原则（Composite Reuse Principle）
   合成复用原则是尽量使用合成/聚合的方式，而不是使用继承。

4.1.2 设计模式的分类

Java 设计模式可以分为以下几个大类：

创建型模式 共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式 共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式 共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其他 并发型模式和线程池模式。

4.2 实验目的

通过本实验章节的学习，需要完成：

1. 理解设计模式之工厂（Factory）模式。
2. 理解工厂模式中所包含的 Java 设计原则。
3. 学习使用工厂（Factory）模式编写程序。
4. 为后续持续学习掌握 Java 其他设计模式打下一定的基础。

4.3 问题分析

我们首先来看一个不够合理的披萨店设计：

```
1  public class OldStore {
2
3    Pizza orderPizza() {
4      Pizza pizza = new Pizza();
5
6      pizza.prepare();
7      pizza.bake();
8      pizza.cut();
9      pizza.box();
10     return pizza;
11   }
12 }
```

在上述代码中，为了让系统有弹性，我们希望 Pizza 类是个抽象类或者接口。但如果是这样，这个类或接口就无法直接实例化。

当我们需要更多种类的 Pizza 时，我们必须增加代码来确定适合的 Pizza 类型，然后再制造 Pizza。所以，披萨店设计可能变成了如下的样子：

```
public class OldStore {

  Pizza OrderPizza(String type) {
    Pizza pizza;
    // ---------  不断变化的部分 ----------
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {
      pizza = new GreekPizza();
    } else if (type.equals("other")) {
      // other type.
    }
    //     ---------------------------------
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

但是，如果需要不断的增加 Pizza 的种类，我们需要不断的修改 orderPizza() 方法的代码，这使得orderPizza() 无法对修改关闭，所以我们需要进一步使用封装。

接下来，我们将分析三种类型的工厂模式，严格的说第一种称不上是设计模式，而可以理解为一种编程习惯。

4.3.1 简单工厂

❶ 封装创建对象的代码 将创建 Pizza 对象的代码移到 orderPizza() 方法之外的另一个对象 SimplePizzaFactory 中，由这个新对象专职创建披萨。我们称这个新对象为“工厂”。这样，orderPizza() 只需要关心从工厂得到一个披萨，然后进行后续的操作。

```
public class OldStore {

  Pizza OrderPizza() {
    Pizza pizza;

    // 披萨工厂，应该如何实现？

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
```

```
13    }
14  }
```

❷ 建立简单的披萨工厂 建立工厂类，为所有披萨封装创建对象的代码。

```
1   public class SimplePizzaFactroy {
2
3     public Pizza createPizza(String type) {
4       Pizza pizza = null;
5
6       if (type.equals("cheese")) {
7         pizza = new CheesePizza();
8       } else if (type.equals("greek")) {
9         pizza = new GreekPizza();
10      } else if (type.equals("other")) {
11        // other type.
12      }
13
14      return pizza;
15    }
16  }
```

当然，上述实现并不完美。

❸ 重做 PizzaStore 类 代码如下。

```
1   public class PizzaStore {
2     SimplePizzaFactory factory; // 加入一个对 SimplePizzaFactory 的引用
3
4     // 构造器需要一个工厂作为参数
5     public PizzaStore(SimplePizzaFactory factory) {
6       this.factory = factory;
7     }
8
9     public Pizza orderPizza(String type) {
10      Pizza pizza;
11
12      // 该方法通过简单传入订单类型来使用工厂创建披萨
13      pizza = factory.createPizza(type); // new 操作符被替换为工厂对象创建方法
14
15      pizza.prepare();
16      pizza.bake();
17      pizza.cut();
18      pizza.box();
19      return pizza;
20    }
21  }
```

❸ 简单工厂小结 简单工厂类图如图 4.1所示。

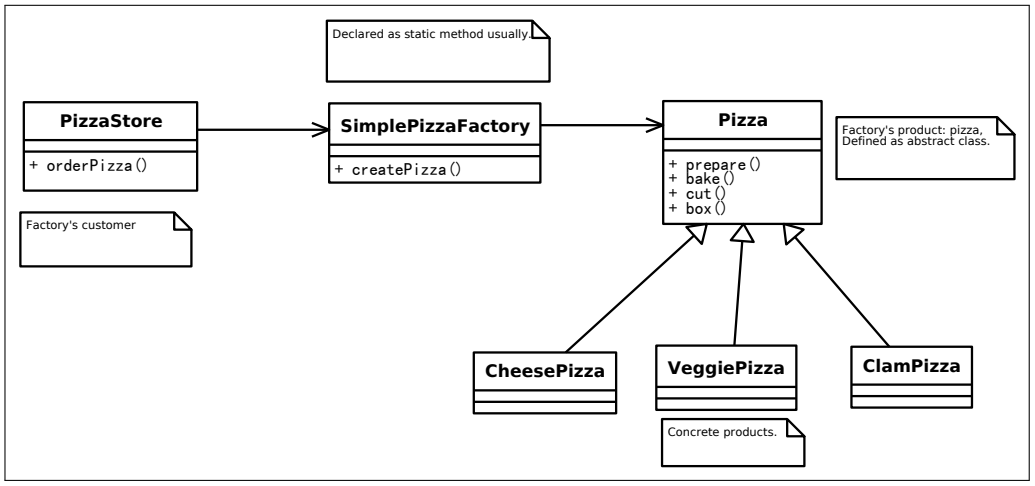

图 4.1: 简单工厂类图

4.3.2 工厂方法

❶ 加盟披萨店 要求披萨店的各个加盟店能够提供不同风味的披萨，并复用代码，以使得披萨的流程能够一致不变。利用 SimplePizzaFactory 的一般实现如下：

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie") ;

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore nyStore = new PizzaStore(chicagoFactory);
nyStore.orderPizza("Veggie") ;
```

希望能够建立一个框架，把加盟店和创建披萨捆绑在一起，保持披萨的质量控制，同时使得代码具有一定的弹性。

❷ 给披萨店使用的框架 采用如下框架可以使得披萨制作活动局限于 PizzaStore 类，而同时又能让这些加盟店依然可以自由的制作该区域的风味。

```
public abstract class PizzaStore {

  public Pizza OrderPizza(String type) {
    Pizza pizza;

    // createPizza() 方法从工厂对象中回到 PizzaStore
    pizza = createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
```

```
14    }
16    // PizzaStore 里的工厂方法是抽象的
17    abstract Pizza createPizza(String type);
18  }
```

现在，我们将 PizzaStore 作为超类，让每个区域类型（NYPizzaStore 等）都继承这个 PizzaStore，每个子类各自决定如何制作披萨。

❸ 允许子类做决定 orderPizza() 方法负责处理订单，使得所有加盟店对于订单的处理保持一致，如图 4.2所示。

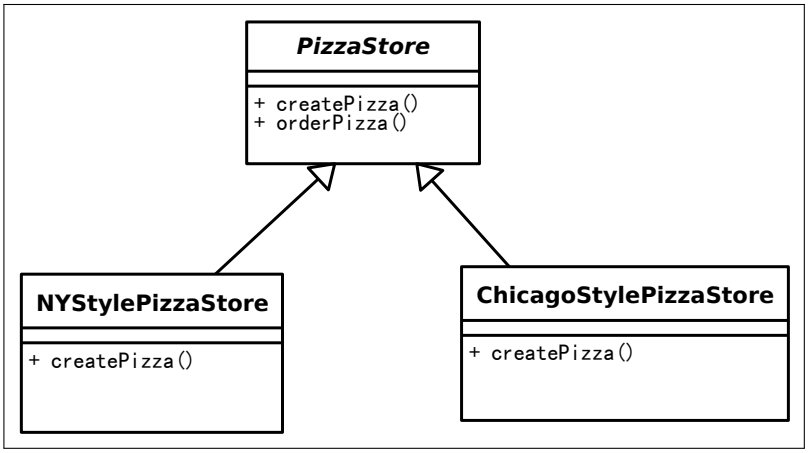

图 4.2: 加盟店实现超类抽象方法

createPizza() 方法是个抽象方法，所有任何具体的加盟店必须实现这个方法，从而个性化加盟店的披萨风味。例如，我们创建 NYPizzaStore。

```
1   public class NYPizzaStore extends PizzaStore {
3     @Override
4     Pizza createPizza(String type) {
6       if (type.equals("cheese")) {
7         return new NYStyleCheesePizza();
8       } else if (type.equals("veggie")) {
9         return new NYStyleVeggiePizza();
10      } else if (type.equals("other")) {
11        // other type.
12      } else
13        return null;
14    }
15  }
```

❹ 声明一个工厂方法 原来是在简单工厂中是由一个对象负责所有具体类的实例化，现在通过对 PizzaStore 作改变，使得由一群子类来负责实例化。

工厂方法用来处理对象的创建，并将这样的行为封装在子类中，这样客户程序中关于超类的代码就和子类对象创建代码解耦。

- 工厂方法是抽象；
- 工厂方法必须返回一个产品，超类中定义的方法，通常会用到工厂方法的返回值；
- 工厂方法将客户（即超类中的代码，如 orderPizza()）和实际创建具体产品的代码分隔开。

```
abstract Product factoryMethod(String type);
```

❺ 制作一些披萨用于出售 声明为抽象类，所有的具体披萨都必须派生自这个类。

```
public abstract class Pizza {
  String name; // 披萨名字
  String dough; // 面团类型
  String sauce;  // 酱料类型
  ArrayList toppings = new ArrayList(); // 一套佐料
  public void prepare() {
    // 具体实现
  }
  public void bake() {
    // 具体实现
  }
  public void cut() {
    // 具体实现
  }
  public void box() {
    // 具体实现
  }
}
```

❻ 工厂方法订购和生产披萨的过程（main() 方法）

1. 需要一个纽约披萨店。

```
    PizzaStore nyPizzaStore = new NYPizzaStore();
```

2. 下订单。

```
    nyPizzaStore.orderPizza("cheese"); // 该方法在 PizzaStore 中定义
```

3. orderPizza() 方法调用 createPizza() 方法

```
1    Pizza pizza = createPizza("chesse"); // 工厂方法在具体子类中实现
2    pizza.prepare();      // orderPizza() 方法并不知道披萨的具体类
3    pizza.bake();
4    pizza.cut();
5    pizza.box();
```

❼ 工厂方法模式小结 工厂方法模式（Factory Method）包含的组成元素：

- 创建者类，定义一个抽象工厂方法，让子类实现此方法制造产品。
- 产品类，抽象产品类，子类实现具体的产品。
- 工厂生产产品。

☛ 工厂方法模式定义

工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是那一个，工厂方法让类把实例化推迟到子类。

4.3.3 抽象工厂

工厂方法模式从设计角度考虑存在一定的问题：类的创建依赖工厂类。也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则。

所以，可以使用抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。参考示例的4.3类图，该示例包含一个消息发送者接口(Sender) 和一个抽象工厂接口（Provider）。

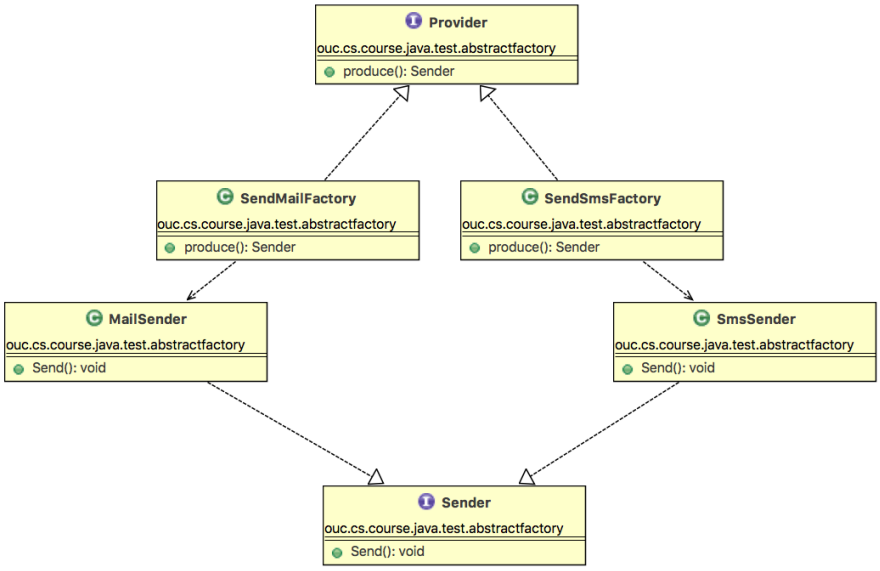

图 4.3: 抽象工厂示例类图

相关代码如下：

File: Sender.java

```
1 package ouc.cs.course.java.test.abstractfactory;
2
3 public interface Sender {
4   public void Send();
5 }
```

File: Provider.java

```
1 package ouc.cs.course.java.test.abstractfactory;
2
3 public interface Provider {
4   public Sender produce();
5 }
```

File: MailSender.java

```
1 package ouc.cs.course.java.test.abstractfactory;
2
3 public class MailSender implements Sender {
4
5   public void Send() {
6     System.out.println("this is mailsender!");
7   }
8 }
```

File: SmsSender.java

```
1 package ouc.cs.course.java.test.abstractfactory;
2
3 public class SmsSender implements Sender {
4
5   public void Send() {
6     System.out.println("this is smssender!");
7   }
8 }
```

File: SendMailFactory.java

```
1 package ouc.cs.course.java.test.abstractfactory;
2
3 public class SendMailFactory implements Provider {
4
5   @Override
6   public Sender produce() {
7     return new MailSender();
8   }
9 }
```

File: SendSmsFactory.java

```
package ouc.cs.course.java.test.abstractfactory;

public class SendSmsFactory implements Provider {

  @Override
  public Sender produce() {
    return new SmsSender();
  }
}
```

File: Test.java

```
package test.abstractfactory;

import ouc.cs.course.java.test.abstractfactory.Provider;
import ouc.cs.course.java.test.abstractfactory.SendMailFactory;
import ouc.cs.course.java.test.abstractfactory.Sender;

public class Test {
  public static void main(String[] args) {
    Provider provider = new SendMailFactory();
    Sender sender = provider.produce();
    sender.Send();
  }
}
```

4.4 实验要求

学习本章内容并参考问题分析章节 4.3，编写工厂模式的程序实例。

4.5 实验过程、步骤及原始记录

实验过程和代码如下：

```
.
.
.
.
.
.
.
.
.
```

4.6 参考程序

本章节参考前述代码示例即可。

第 5 章　Java 命令行文件操作程序设计

实验编号 exp05

5.1　实验目的

1. 掌握 Java 终端命令行程序设计的基本方法。
2. 掌握 Java 基本文件操作类 File 及相关方法，包括 getName()、getPath()、mkdir()、exist() 等，这些方法的具体使用请参考手册。
3. 掌握 Java 程序归档的方法。

5.2　实验要求

设计实现一个 Java 命令行文件操作程序 **fsops.jar**。命令行操作模式，程序的基本使用方法如下：

Windows 平台

```
C:\> java.exe -jar fsops.jar [CMD] [ARGS]
```

其中 [CMD] 是该文件操作工具所支持的文件操作命令；[ARGS] 是 [CMD] 命令所要求的参数，该参数根据 [CMD] 的不同，参数个数存在差异。本实验要求该文件操作工具至少支持以下操作：

- 创建目录，[CMD] 为 mkdir，[ARGS] 为要创建的目录路径（可以为多个参数，即一次可同时创建多个目录），如：

```
C:\> java.exe -jar fsops.jar mkdir D:\testdir
C:\> java.exe -jar fsops.jar mkdir D:\testdir01 D:\testdir02
```

- 新建空文件，[CMD] 为 nfile，[ARGS] 为要新建的文件名称，如：

```
C:\> java.exe -jar fsops.jar mfile D:\testdir\testfile.txt
```

- 删除文件或目录，[CMD] 为 rm，[ARGS] 为要删除的文件或目录，如：

```
C:\> java.exe -jar fsops.jar rm D:\testdir\testfile.txt
C:\> java.exe -jar fsops.jar rm D:\testdir
```

- 列表列出目录下所有文件和子目录，不需要递归列出子目录中包含的文件，[CMD] 为 ls，[ARGS] 为要列出的目录路径，要求能够列出文件或目录的名称、创建时间或最后修改时间、类型是否为目录或文件，如：

```
C:\> java.exe -jar fsops.jar ls D:\testdir

Directory D:\testdir
        ------------------------------------------------------------
Name        Time created        Time modified       Type
test01      2017-10-30 00:10    2017-10-30 00:20    d
test02      2017-10-30 00:30    2017-10-30 00:35    d
test01.txt  2017-10-30 00:32    2017-10-30 00:34    f
test02.txt  2017-10-30 00:40    2017-10-30 00:50    f
```

- 工具帮助，[CMD] 为 help，显示该文件操作程序的简要帮助手册。
- 软件基本信息，[CMD] 为 self，显示本软件的软件名称、版本、开发者和编写日期。

☛ Linux 平台或其他 按照上述要求开发，注意文件路径分隔符为“/”。
命令行使用方式参考下图所示：

5.3 实验过程、步骤及原始记录

1. 使用 UML 建模工具绘制或自动生成该软件的类图；
2. 完成软件程序代码编写；
3. 将代码归档为 fsops.jar；

4. 运行软件并测试功能，给出软件运行使用过程截图；

5. 按规定模板要求撰写实验报告。