敏捷开发的必要技巧

——带你进入敏捷开发的世界

王伟杰(Wingel） 译

Email: seewingel@gmail.com

原著：Essential Skills for Agile Development
原作者：Tong Ka Iok, Kent
对原作有增改

目录

目录 2
第 1 章 移除重复代码 5
重复代码是怎么产生的？ 5
移除重复代码吧！ 6
章节练习 7
解决方法示例 16
第 2 章 将注释转换为代码 35
示例 35
将注释转换为代码,让代码足够清楚到可以表示注释 36
将注释转换为变量名 36
对参数的注释,转化为参数名 37
将注释转换为方法的一部分 38
删掉没用的注释 39
将一部分代码重构成方法,用方法名来表达注释的意思 40
抽取出方法,放于另一个类 42
用注释去命名一个已经存在的方法 43
为什么要删除额外的注释? 45
方法名太长 46
章节练习 47
解决方法示例 52
第 3 章 除去代码异味 60
示例 60
怎么判断代码的稳定性？ 62
消除代码异味：怎么去掉类别代码（type code） 64
消除代码异味：如何去掉一大串 if-then-else-if（或者 switch） 65
另一个例子 68
总结一下类别代码的移除 72
普遍的代码异味 72
章节练习 73
解决方法示例 88
第 4 章 保持代码简洁 118
示例 119
怎么判断一个类需要修整 121
章节练习 126
解决方法示例 137
第 5 章 慎用继承 167
示例 168
总结 176
章节练习 177
解决方法示例 181

第 6 章　处理不合适的依赖 193
示例 *194*
“不合适的依赖”，让代码很难被重用 *195*
怎么判断是“不合适的依赖” *195*
总结 *197*
章节练习 *201*
解决方法示例 *205*
第 7 章　将数据库访问，UI 和域逻辑分离 213
示例 *213*
层次混乱造成的问题 *216*
抽取出访问数据库代码后得到的灵活性 *219*
将域逻辑跟 UI 分离 *220*
给系统分层 *227*
很多东西都属于 UI 层 *231*
章节练习 *231*
解决方法示例 *236*
第 8 章 以用户例事管理项目 247
什么是用户例事(user story) *247*
用户例事只是描述系统的外在行为 *249*
评估发布时间 *249*
预计不能如期完成时怎么办？ *252*
发布计划编制，估算每个用户例事时要考虑哪些细节，忽略哪些细节？ *254*
迭代周期内的计划编制 *255*
章节练习（这章不用章节练习） *256*
第 9 章　用 CRC 卡协助设计 256
一个用户例事的例子 *256*
用户例事的设计 *256*
CRC 卡的典型应用 *260*
第 10 章　验收测试（ACCEPTANCE TEST） 261
我们是不是正确的实现了一个用户例事 *261*
怎么测试 *262*
自动验收测试 *264*
先写测试代码，作为需求文档 *275*
让客户看得懂测试用例 *276*
测试文件不一定是文本文件 *279*
用测试用例防止系统走下坡路 *279*
章节练习 *279*
解决方法示例 *280*
第 11 章 对 UI 进行验收测试 281
怎么操作 UI *281*
分开测试每个 UI 组件 *282*
如果要测试的对话框会弹出另一个对话框怎么办？ *285*
章节练习 *289*

解决方法示例 *290*
第 12 章　单元测试 294
单元测试 *294*
使用 JUnit *297*
我们需要对所有的类进行单元测试吗？ *301*
怎么运行所有的单元测试 *302*
什么时候运行单元测试 *303*
用单元测试来防止以后出现同样的 bug *304*
怎么测试是否有抛异常 *304*
章节练习 *306*
解决方法示例 *306*
第 13 章　测试驱动编程 308
实现一个跟学生选修课程的用户例事 *308*
怎么测试刚写的代码 *315*
测试先行的解决方法 *319*
测试->检查学生是否注册的代码 *322*
系统有没有考虑到父课程的选修情况 *331*
测试系统有没有考虑预订的情况 *331*
测试 add 方法 *334*
对 EnrollmentSet 的结构维持完整的概念 *350*
TDD 及它的优点 *353*
要做什么，不要做什么 *353*
章节练习 *354*
解决方法示例 *356*
第 14 章 结对编程 384
两个人怎么一起编程 *384*
结对编程的好处： *404*
结对编程需要的一些技能： *405*
需要把程序员的数量加倍吗？ *405*
什么情况结对编程行不通 *405*

第 1 章　移除重复代码

重复代码是怎么产生的？

请观察下面的代码，我们已经有一个根据出租记录的 id 取出租用客户姓名的方法:getCustomerName。

```
public class BookRental { //该类描述出租记录
    String id;
    String customerName;

    ...

}
public class BookRentals {
    private Vector rentals;
    public String getCustomerName(String rentalId) {  根据出租 id 取出客户姓名
        for (int i = 0; i < rentals.size(); i++) {
            BookRental rental = (BookRental) rentals.elementAt(i);
            if (rental.getId().equals(rentalId)) {
                return rental.getCustomerName();
            }
        }
        throw new RentalNotFoundException();
    }
}
public class RentalNotFoundException extends Exception {

    ...

}
```

假定现在你要增加一个新的方法，该方法是根据出租记录的 id 删除该记录,你把这方法命名为 deleteRental(String rentalId)。现在你已经考虑到，就像 getCustomerName 这个方法一样，也要一个一个遍历出租记录。所以你就将 getCustomerName 这个方法里面的一些代码拷出来，然后稍微修改一下：

```
public class BookRentals {
    private Vector rentals;
    public String getCustomerName(String rentalId) {
        for (int i = 0; i < rentals.size(); i++) {
            BookRental rental = (BookRental) rentals.elementAt(i);
```

```
                if (rental.getId().equals(rentalId)) {
                    return rental.getCustomerName();
                }
            }
            throw new RentalNotFoundException();
        }
        public void deleteRental(String rentalId) {
            for (int i = 0; i < rentals.size(); i++) {
                BookRental rental = (BookRental) rentals.elementAt(i);
                if (rental.getId().equals(rentalId)) {
                    rentals.remove(i);
                    return;
                }
            }
            throw new RentalNotFoundException();
        }
    }
```

现在这样的代码看起来怎么样？不怎么样，两个方法有大多的同样的代码了。

移除重复代码吧！

要移除所有的重复代码，你可以将 BookRentals 这个类修成如下的样子（也就是“重构”了）：

```
public class BookRentals {
    private Vector rentals;
    public String getCustomerName(String rentalId) {
        int rentalIdx = getRentalIdxById(rentalId);
        return ((BookRental) rentals.elementAt(rentalIdx)).getCustomerName();
    }
    public void deleteRental(String rentalId) {
        rentals.remove(getRentalIdxById(rentalId));
    }
    private int getRentalIdxById(String rentalId) { //新增加的一个方法
        for (int i = 0; i < rentals.size(); i++) {
            BookRental rental = (BookRental) rentals.elementAt(i);
            if (rental.getId().equals(rentalId)) {
                return i;
            }
```

```
            }
            throw new RentalNotFoundException();
        }
    }
```

为什么我们要移除重复代码?

我来向各位程序员同学稍微说一下，在 BookRentals 这个类中，rentals 这个属性的类型是 Ventor，如果我们需要将它改为数组，那我们就必须将所有的"rentals.size()"改为"rentals.length". 在重构以后的版本中，我们只需要在 getRentalIdxById 这个方法中修改一次，而在原来的版本，我们就得在 getCustomerName 跟 deleteRental 两个方法中都改一次。类似的，我们还要将所有的"rentals.elementAt(i)" 改为 "rentals[i]". 又是改一次跟改两次的比较！

大多数情况中，如果类似这样的代码在 10 个地方重复，当我们修改代码的时候，就要修改 10 个地方，我们并不能保证能把这 10 个地方都记住了，而一旦漏掉了几个地方，等待我们的，是一处一处的错误去修复。而最致命的是,当我们修改的是业务逻辑时,这时候，不管我们漏掉了几个地方，IDE 都不会报错，那么，等待我们的，将是一堆 Bug 去检查，而造成的一些 bug 中,很可能是短时间内还发现不了的. 惨-_-!!

章节练习

介绍

不用刻意管下面的代码到底是什么语言,也不用管是否有语法错误,我们谈的是设计思想,而不是在具体语言上的实现.

建议读者可以先自己找出并解决掉代码的质量问题,然后再比较一下答案(当然,答案不是唯一的),看大家的做法有什么不一样.

问题

1. 指出并修改下面的重复代码:

```
class Organization {
    String id;
    String eName; //英文名字
    String cName; //中文名字
    String telCountryCode;
    String telAreaCode;
    String telLocalNumber;
    String faxCountryCode;
    String faxAreaCode;
```

```
        String faxLocalNumber;
        String contactPersonEFirstName; //联系人的英文名
        String contactPersonELastName; //联系人的英文姓
        String contactPersonCFirstName; //联系人的中文名
        String contactPersonCLastName; //联系人的中文姓
        String contactPersonTelCountryCode;
        String contactPersonTelAreaCode;
        String contactPersonTelNumber;
        String contactPersonFaxCountryCode;
        String contactPersonFaxAreaCode;
        String contactPersonFaxLocalNumber;
        String contactPersonMobileCountryCode;
        String contactPersonMobileAreaCode;
        String contactPersonMobileLocalNumber;

        ...

    }
```

2. 指出并修改下面的重复代码:

```
    class ParticipantsInDB {
        Connection db;
        ParticipantsInDB(){
            Class.forName("org.postgresql.Driver");
            db =
                DriverManager.getConnection(
                    "jdbc:postgresql://myhost/ConferenceDB",
                    "PaulChan",
                    "myP@ssword");
        }
        void addParticipant(Participant part) {
            PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES (?,?,?,?,?,?,?)");
            try {
                st.setString(1, part.getId());
                st.setString(2, part.getEFirstName());
                st.setString(3, part.getELastName());

                ...

                st.executeUpdate();
            } finally {
                st.close();
```

```
            }
        }
    }
    class OrganizationsInDB {
        Connection db;
        OrganizationsInDB() {
            Class.forName("org.postgresql.Driver");
            db =
                DriverManager.getConnection(
                    "jdbc:postgresql://myhost/ConferenceDB",
                    "PaulChan",
                    "myP@ssword");
        }
        void addOrganization(Organization o) {
            PreparedStatement st =
                db.prepareStatement(
                    "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
            try {
                st.setString(1, o.getId());
                st.setString(2, o.getEName());
                st.setString(3, o.getCName());

                ...

                st.executeUpdate();
            } finally {
                st.close();
            }
        }
    }
```

3. 指出并修改下面的重复代码:

```
    class ParticipantsInDB {
        Connection db;
        void addParticipant(Participant part) {
            PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES (?,?,?,?,?,?,?)");
            try {
                st.setString(1, part.getId());

                ...

                st.executeUpdate();
```

```
            } finally {
                st.close();
            }
        }
        void deleteAllParticipants() {
            PreparedStatement st = db.prepareStatement("DELETE FROM participants");
            try {
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
        int getCount() {
            PreparedStatement st = db.prepareStatement("SELECT COUNT(*) FROM    participants");
            try {
                ResultSet rs = st.executeQuery();
                rs.next();
                return rs.getInt(1);
            } finally {
                st.close();
            }
        }
    }
```

4. 指出并修改下面的重复代码:

```
    class ParticipantsInDBTest extends TestCase {
        ParticipantsInDB p;
        void setUp() {
            p=ParticipantsInDB.getInstance();
        }
        void tearDown() {
            ParticipantsInDB.freeInstance();
        }
        void testAdd() {
            Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
            p.deleteAllParticipants();
            p.addParticipant(part1);
            assertEquals(p.getCount(),1);
        }
        void testAdd2() {
            Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
            Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
```

```
        p.deleteAllParticipants();
        p.addParticipant(part1);
        p.addParticipant(part2);
        assertEquals(p.getCount(),2);
    }
    void testEnum() {
        Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
        Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
        p.deleteAllParticipants();
        p.addParticipant(part2);
        p.addParticipant(part1);
        ParticipantEnumeratorById penum=new ParticipantEnumeratorById();
        assertTrue(penum.next());
        assertTrue(penum.get().equals(part1));
        assertTrue(penum.next());
        assertTrue(penum.get().equals(part2));
        assertTrue(!penum.next());
        penum.close();
    }
}
```

5. 指出并修改下面的重复代码:

```
class ReportCatalogueIndexCommandParser {
    final String NO_GROUPING = "orgNoGrouping";
    static final int ORG_CATALOG = 0;
    static final int PART_CATALOG = 1;
    int getGroupingType(String grouping)                {
        if (grouping.equals(NO_GROUPING)) {
            return ORG_CATALOG;
        } else if (grouping.equals("orgGroupByCountry")) {
            return ORG_CATALOG;
        } else if (grouping.equals("orgGroupByTypeOfOrgName")) {
            return ORG_CATALOG;
        } else if (grouping.equals("part")) {
            return PART_CATALOG;
        } else
            throw new IllegalArgumentException("Invalid grouping!");
    }

    ...

}
```

6. 指出并修改下面的重复代码:

```
class Restaurant extends Account {

    //字符串"Rest"是作为前缀来组成饭店的 id
    final static String RestaurantIDText = "Rest";
    String website;
    //中文地址
    String addr_cn;
    //英文地址
    String addr_en;
    //下面是最新的传真号码,饭店想要更新它的传真号,等确认下面的新传真号是正确的以后
    //新传真号就会存放在 Account 中,在这之前,都会存放在这个位置

    String numb_newfax;
    boolean has_NewFax = false;
    //存放假日的一个集合
    Vector Holiday; // a holiday
    //该饭店所属分类的 id
    String catId;
    //营业时段的集合,每个营业时段都是一个数组,
    //数组中包括两个时间,前面一个表示开始时间,后面一个表示结束时间
    //饭店在所有的时间段内营业
    Vector BsHour; //营业时段集

    ...

    //y: 年.
    //m: 月.
    //d: 日.
    void addHoliday(int y,int m,int d){
        if(y<1900) y+=1900;
        Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
        Holiday.add(aHoliday);
    }
    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
        int fMin = fromHr*60 + fromMin;    //以分表示的开始时间
        int tMin = toHr*60 + toMin;               //以分表示的结束时间
    //确定所有的时间都是有效的,而且结束时间比开始时间晚
        if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
            GregorianCalendar bs[] = {
                new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
```

```
                    new GregorianCalendar(1900,1,1, toHr, toMin,0)
                };
                BsHour.add(bs);
                return true;
            } else {
                return false;
            }
        }
    }
```

7. 指出并修改下面的重复代码:

```
class Order {

    ...

    boolean IsSameString(String s1, String s2){
        if(s1==s2) return true;
        if(s1==null) return false;
        return(s1.equals(s2));
    }
}
class Mail {

    ...

    static boolean IsSameString(String s1, String s2) {
        if (s1 == s2)
            return true;
        if (s1 == null)
            return false;
        return (s1.equals(s2));
    }
}
```

8. 指出并修改下面的重复代码:

```
class FoodDB {
    //找出所有名字都包括这两个关键字的食物,返回一个迭代器(Iterator)
    public Iterator srchFood(String cName, String eName){
        //it contains all the foods to be returned.
        TreeMap foodTree = new TreeMap();
        Iterator foodList;
```

```
            Food food;
            foodList = getAllRecords();
            while (foodList.hasNext()){
                food = (Food) foodList.next();
                //有没有包括两个关键字?
                if ((cName==null || //null or "" means no key is specified
                        cName.equals("") ||
                        food.getCName().indexOf(cName)!=-1)
                        &&
                        (eName==null || //null 或者 "" 表示没有关键字
                        eName.equals("") ||
                        food.getEName().indexOf(eName)!=-1)){
                        foodTree.put(food.getFoodKey(),food);
                }
            }
            return foodTree.values().iterator();
        }
    }
```

9. 指出并修改下面的重复代码:

```
    class UIDialogCustomerMain extends JDialog {
        JButton btnOrderDel;
        JButton btnCustChangePassword;
        void bindEvents() {

            ...

            btnOrderDel.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        Dialog d = new UIDialogCustomerDeleteOrder(UIDialogCustomerMain.this, "Del Order", true);
                        d.pack();
                        d.setLocation(400,200);
                        d.setVisible(true);
                }
            });
            btnCustChangePassword.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        Dialog d = new UIChangeAccountPW(UIDialogCustomerMain.this, "chg pw", true);
                        d.pack();
                        d.setLocation(400,200);
                        d.setVisible(true);
                }
```

```
            });
        }

        ...

    }
    class UIDialogRestaurantMain extends JDialog {
        JButton btnChangePassword;
        void bindEvents() {

            ...

        btnChangePassword.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    Dialog d = new UIChangeAccountPW(UIDialogRestaurantMain.this, "chgpw", true);
                    d.pack();
                    d.setLocation(400,200);
                    d.setVisible(true);
                }
            });
        }

        ...

    }
```

10.假定现在有两种出租种类:书和影碟. 指出并修改下面的重复代码.

```
    public class BookRental {
        private String bookTitle;
        private String author;
        private Date rentDate;
        private Date dueDate;
        private double rentalFee;
        public boolean isOverdue() {
            Date now=new Date();
            return dueDate.before(now);
        }
        public double getTotalFee() {
            return isOverdue() ? 1.2*rentalFee : rentalFee;
        }
    }
    public class MovieRental {
```

```
        private String movieTitle;
        private int classification;
        private Date rentDate;
        private Date dueDate;
        private double rentalFee;
        public boolean isOverdue() {
            Date now=new Date();
            return dueDate.before(now);
        }
        public double getTotalFee() {
            return isOverdue() ? 1.3*rentalFee : rentalFee;
        }
    }
```

11.Come up with some code that contains duplicate code/structure, point out the duplication and eliminate it.

提示

1. 这三个组合(国家区号, 地区号, 号码)出现太多次了. 它们应该被抽取出来,放到一个类,比如叫 TelNo 里面. "contactPerson"这个词也出现在太多属性中了, 这些属性也应该抽取出来,放到一个类,比如叫 ContactPerson.

2. 载入 Postgres 的 JDBC 驱动,取出一个连接,这样的代码在两个类中都出现了,它们应该抽取出来,放在一个叫 ConferenceDBConnection 的类里面.

3. 数据表名("participants") 在三个地方都出现了,它们应该转为一个 static final String 的属性,放在这个类中,或者抽取到一个类比如:ParticipantsTable.

4. 创建 Participant 的代码重复了,它们应该抽取出来,变成一个实例变量.

5. if-then 的每个分支都做了类似的事. 应该抽取所有的字符串放在数组或者 Map 里面,然后遍历就行了.

解决方法示例

1. 指出并修改下面的重复代码:

```
class Organization {
    String id;
    String eName; //English name
```

```
    String cName; //Chinese name
    String telCountryCode;
    String telAreaCode;
    String telLocalNumber;
    String faxCountryCode;
    String faxAreaCode;
    String faxLocalNumber;
    String contactPersonEFirstName; //First name and last name in English
    String contactPersonELastName;
    String contactPersonCFirstName; //First name and last name in Chinese
    String contactPersonCLastName;
    String contactPersonTelCountryCode;
    String contactPersonTelAreaCode;
    String contactPersonTelNumber;
    String contactPersonFaxCountryCode;
    String contactPersonFaxAreaCode;
    String contactPersonFaxLocalNumber;
    String contactPersonMobileCountryCode;
    String contactPersonMobileAreaCode;
    String contactPersonMobileLocalNumber;

    ...

}
```

代码修改成:

```
class Organization {
    String id;
    String eName;
    String cName;
    TelNo telNo;
    TelNo faxNo;
    ContactPerson contactPerson;

    ...

}
class ContactPerson{
    String eFirstName;
    String eLastName;
    String cFirstName;
    String cLastName;
```

```
    TelNo tel;
    TelNo fax;
    TelNo mobile;
}
class TelNo {
    String countryCode;
    String areaCode;
    String localNumber;
}
```

如果你觉得每次出现 FirstName, LastName 两次也不爽的话,你就将这两个属性也放在一个类中:

```
class ContactPerson{
    FullName eFullName;
    FullName cFullName;
    TelNo tel;
    TelNo fax;
    TelNo mobile;
}
class FullName {
    String firstName;
    String lastName;
}
```

2. 指出并修改下面的重复代码:

```
class ParticipantsInDB {
    Connection db;
    ParticipantsInDB(){
        Class.forName("org.postgresql.Driver");
        db =
            DriverManager.getConnection(
                "jdbc:postgresql://myhost/ConferenceDB",
                "PaulChan",
                "myP@ssword");
    }
    void addParticipant(Participant part) {
        PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES (?,?,?,?,?,?,?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());
```

```
                ...

                st.executeUpdate();
            } finally {
                st.close();
            }
        }
    }
    class OrganizationsInDB {
        Connection db;
        OrganizationsInDB() {
            Class.forName("org.postgresql.Driver");
            db =
                DriverManager.getConnection(
                    "jdbc:postgresql://myhost/ConferenceDB",
                    "PaulChan",
                    "myP@ssword");
        }
        void addOrganization(Organization o) {
            PreparedStatement st =
                db.prepareStatement(
                    "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
            try {
                st.setString(1, o.getId());
                st.setString(2, o.getEName());
                st.setString(3, o.getCName());

                ...

                st.executeUpdate();
            } finally {
                st.close();
            }
        }
    }
```

取得 connection 的代码重复,抽取在一个单独的类中:

```
class ConferenceDBConnection {
    static Connection makeConnection() {
        Class.forName("org.postgresql.Driver");
        return
            DriverManager.getConnection(
```

```
                "jdbc:postgresql://myhost/ConferenceDB",
                "PaulChan",
                "myP@ssword");
    }
}
class ParticipantsInDB {
    Connection db;
    ParticipantsInDB(){
        db = ConferenceDBConnection.makeConnection();
    }
    void addParticipant(Participant part) {
        PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES (?,?,?,?,?,?,?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());

            ...

            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
class OrganizationsInDB {
    Connection db;
    OrganizationsInDB() {
        db = ConferenceDBConnection.makeConnection();
    }
    void addOrganization(Organization o) {
        PreparedStatement st =
            db.prepareStatement(
                "INSERT INTO organizations VALUES (?,?,?,?,?,?,?,?,?,?,)");
        try {
            st.setString(1, o.getId());
            st.setString(2, o.getEName());
            st.setString(3, o.getCName());

            ...

            st.executeUpdate();
        } finally {
```

```
                st.close();
            }
        }
    }
```

3. 指出并修改下面的重复代码:

```
    class ParticipantsInDB {
        Connection db;
        void addParticipant(Participant part) {
            PreparedStatement st = db.prepareStatement("INSERT INTO participants VALUES (?,?,?,?,?,?,?)");
            try {
                st.setString(1, part.getId());

                ...

                st.executeUpdate();
            } finally {
                st.close();
            }
        }
        void deleteAllParticipants() {
            PreparedStatement st = db.prepareStatement("DELETE FROM participants");
            try {
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
        int getCount() {
            PreparedStatement st = db.prepareStatement("SELECT COUNT(*) FROM participants");
            try {
                ResultSet rs = st.executeQuery();
                rs.next();
                return rs.getInt(1);
            } finally {
                st.close();
            }
        }
    }
```

表名重复了. 调用"db.preparedStatement"也重复了.

```
class ParticipantsInDB {
    static final String tableName="participants";
    Connection db;
    PreparedStatement makeStatement(String sql) {
        return   db.prepareStatement(sql);
    }
    void addParticipant(Participant part) {
        PreparedStatement st = makeStatement("INSERT INTO "+tableName+" VALUES (?,?,?,?,?,?,?)");
        try {
            st.setString(1, part.getId());

            ...

            st.executeUpdate();
        } finally {
            st.close();
        }
    }

    void deleteAllParticipants() {
        PreparedStatement st = makeStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    int getCount() {
        PreparedStatement st = makeStatement("SELECT COUNT(*) FROM "+tableName);
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        } finally {
            st.close();
        }
    }
}
```

4. 指出并修改下面的重复代码:

```
class ParticipantsInDBTest extends TestCase {
    ParticipantsInDB p;
```

```
    void setUp() {
        p = ParticipantsInDB.getInstance();
    }
    void tearDown() {
        ParticipantsInDB.freeInstance();
    }
    void testAdd() {
        Participant part1 =
            new Participant("ABC001", "Kent", "Tong", true, "Manager");
        p.deleteAllParticipants();
        p.addParticipant(part1);
        assertEquals(p.getCount(), 1);
    }
    void testAdd2() {
        Participant part1 =
            new Participant("ABC001", "Kent", "Tong", true, "Manager");
        Participant part2 =
            new Participant("ABC003", "Paul", "Chan", true, "Manager");
        p.deleteAllParticipants();
        p.addParticipant(part1);
        p.addParticipant(part2);
        assertEquals(p.getCount(), 2);
    }
    void testEnum() {
        Participant part1 =
            new Participant("ABC001", "Kent", "Tong", true, "Manager");
        Participant part2 =
            new Participant("ABC003", "Paul", "Chan", true, "Manager");
        p.deleteAllParticipants();
        p.addParticipant(part2);
        p.addParticipant(part1);
        ParticipantEnumeratorById penum = new ParticipantEnumeratorById();
        assertTrue(penum.next());
        assertTrue(penum.get().equals(part1));
        assertTrue(penum.get().equals(part2));
        assertTrue(!penum.next());
        penum.close();
    }
}
```

每次都要创建 part1 跟 part2,这两句代码可以拿出来了.

```
class ParticipantsInDBTest extends TestCase {
```

```
        ParticipantsInDB p;
        Participant part1=new Participant("ABC001","Kent","Tong",true,"Manager");
        Participant part2=new Participant("ABC003","Paul","Chan",true,"Manager");
        void setUp() {
            p=ParticipantsInDB.getInstance();
        }
        void tearDown() {
            ParticipantsInDB.freeInstance();
        }
        void testAdd() {
            p.deleteAllParticipants();
            p.addParticipant(part1);
            assertEquals(p.getCount(),1);
        }
        void testAdd2() {
            p.deleteAllParticipants();
            p.addParticipant(part1);
            p.addParticipant(part2);
            assertEquals(p.getCount(),2);
        }
        void testEnum() {
            p.deleteAllParticipants();
            p.addParticipant(part2);
            p.addParticipant(part1);
            ParticipantEnumeratorById penum=new ParticipantEnumeratorById();
            assertTrue(penum.next());
            assertTrue(penum.get().equals(part1));
            assertTrue(penum.next());
            assertTrue(penum.get().equals(part2));
            assertTrue(!penum.next());
            penum.close();
        }
    }
```

5. 指出并修改下面的重复代码:

```
    class ReportCatalogueIndexCommandParser {
        final String NO_GROUPING = "orgNoGrouping";
        static final int ORG_CATALOG = 0;
        static final int PART_CATALOG = 1;
        int getGroupingType(String grouping) {
            if (grouping.equals(NO_GROUPING)) {
                return ORG_CATALOG;
```

```
        } else if (grouping.equals("orgGroupByCountry")) {
            return ORG_CATALOG;
        } else if (grouping.equals("orgGroupByTypeOfOrgName")) {
            return ORG_CATALOG;
        } else if (grouping.equals("part")) {
            return PART_CATALOG;
        } else
            throw new IllegalArgumentException("Invalid grouping!");
    }

    ...

}
```

每个 if-then 的分支都做了类似的事,将字符串放在一个集合里面中.

```
class ReportCatalogueIndexCommandParser {
    final String NO_GROUPING = "orgNoGrouping";
    static final int ORG_CATALOG = 0;
    static final int PART_CATALOG = 1;

    int getGroupingType(String grouping) {
        Set orgGroupings = new TreeSet();
        orgGroupings.add(NO_GROUPING);
        orgGroupings.add("orgGroupByCountry");
        orgGroupings.add("orgGroupByTypeOfOrgName");
        if (orgGroupings.contains(grouping)) {
            return ORG_CATALOG;
        }
        if (grouping.equals("part")) {
            return PART_CATALOG;
        } else
            throw new IllegalArgumentException("Invalid grouping!");
    }
}
```

用 Set 可能会有些大材小用了,我们应该需要什么,用什么. 简单的一个方法,就是将返回结果的代码抽取出来.

```
class ReportCatalogueIndexCommandParser {
    final String NO_GROUPING = "orgNoGrouping";
    static final int ORG_CATALOG = 0;
    static final int PART_CATALOG = 1;
    int getGroupingType(String grouping)            {
```

```
        if (grouping.equals(NO_GROUPING) ||
            grouping.equals("orgGroupByCountry") ||
            grouping.equals("orgGroupByTypeOfOrgName")) {
            return ORG_CATALOG;
        } else if (grouping.equals("part")) {
            return PART_CATALOG;
        } else
            throw new IllegalArgumentException("Invalid grouping!");
    }

    ...

}
```

6. 指出并修改下面的重复代码:

```
class Restaurant extends Account {

    //字符串"Rest"是作为前缀来组成饭店的 id
    final static String RestaurantIDText = "Rest";
    String website;
    //中文地址
    String addr_cn;
    //英文地址
    String addr_en;
    //下面是最新的传真号码,饭店想要更新它的传真号,等确认下面的新传真号是正确的以后
    //新传真号就会存放在 Account 中,在这之前,都会存放在这个位置

    String numb_newfax;
    boolean has_NewFax = false;
    //存放假日的一个集合
    Vector Holiday; // a holiday
    //该饭店所属分类的 id
    String catId;
    //营业时段的集合,每个营业时段都是一个数组,
    //数组中包括两个时间,前面一个表示开始时间,后面一个表示结束时间
    //饭店在所有的时间段内营业
    Vector BsHour; //营业时段集

    ...

    //y: 年.
    //m: 月.
```

```
    //d: 日.
    void addHoliday(int y,int m,int d){
        if(y<1900) y+=1900;
        Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
        Holiday.add(aHoliday);
    }
    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
        int fMin = fromHr*60 + fromMin;   //以分表示的开始时间
        int tMin = toHr*60 + toMin;          //以分表示的结束时间
    //确定所有的时间都是有效的,而且结束时间比开始时间晚
        if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
            GregorianCalendar bs[] = {
                new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
                new GregorianCalendar(1900,1,1, toHr, toMin,0)
            };
            BsHour.add(bs);
            return true;
        } else {
            return false;
        }
    }
}
```

有三处代码重复:1. 将几时几分转化为总计几分的代码重复. 2. 验证分的总数是否合理的代码重复. 3. 创建一个从 1900.1.1 算起的 GregorianCalendar 的代码重复.

```
class Restaurant extends Account {

    ...

   int getMinutesFromMidNight(int hours, int minutes) {
       return hours*60+minutes;
    }
   boolean isMinutesWithinOneDay(int minutes) {
       return minutes>0 && minutes<=1440;
    }
   GregorianCalendar convertTimeToDate(int hours, int minutes) {
       return new GregorianCalendar(1900, 1, 1, hours, minutes, 0);
    }

    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
        int fMin = getMinutesFromMidNight(fromHr, fromMin);
        int tMin = getMinutesFromMidNight(toHr, toMin);
```

```
            if (isMinutesWithinOneDay(fMin) &&
                isMinutesWithinOneDay(tMin) &&
                fMin < tMin){
                GregorianCalendar bs[] = {
                    convertTimeToDate(fromHr, fromMin),
                    convertTimeToDate(toHr, toMin)
                };
                BsHour.add(bs);
                return true;
            } else {
                return false;
            }
        }
    }
```

7. 指出并修改下面的重复代码:

```
class Order {

    ...

    boolean IsSameString(String s1, String s2){
        if(s1==s2) return true;
        if(s1==null) return false;
        return(s1.equals(s2));
    }
}
class Mail {

    ...

    static boolean IsSameString(String s1, String s2) {
        if (s1 == s2)
            return true;
        if (s1 == null)
            return false;
        return (s1.equals(s2));
    }
}
```

方法 IsSameString 重复.

```
class StringComparer {
```

```
    static boolean IsSameString(String s1, String s2) {
        if (s1 == s2)
            return true;
        if (s1 == null)
            return false;
        return (s1.equals(s2));
    }
}
class Order {

    ...

    //如果可能的话,将这个方法移到一个 Util 的类里面作为一个静态方法,比如 StringComparer
    //然后直接调用 StringComparer.IsSameString.
    boolean IsSameString(String s1, String s2){
        return StringComparer.IsSameString(s1, s2);
    }
}

class Mail {

    ...

    //如果可能的话,将这个方法移到一个 Util 的类里面作为一个静态方法,比如 StringComparer
    //然后直接调用 StringComparer.IsSameString.
    static boolean IsSameString(String s1, String s2) {
        return StringComparer.IsSameString(s1, s2);
    }
}
```

8. 指出并修改下面的重复代码:

```
class FoodDB {
    //找出所有名字都包括这两个关键字的食物,返回一个迭代器(Iterator)
    public Iterator srchFood(String cName, String eName){
        //it contains all the foods to be returned.
        TreeMap foodTree = new TreeMap();
        Iterator foodList;
        Food food;
        foodList = getAllRecords();
        while (foodList.hasNext()){
            food = (Food) foodList.next();
            //有没有包括两个关键字?
```

```
            if ((cName==null || //null 或者 "" 表示没有关键字
                cName.equals("") ||
                food.getCName().indexOf(cName)!=-1)
                &&
                (eName==null || //null 或者 "" 表示没有关键字
                eName.equals("") ||
                food.getEName().indexOf(eName)!=-1)){
                foodTree.put(food.getFoodKey(),food);
            }
        }
        return foodTree.values().iterator();
    }
}
```

检查是否包括关键字的代码重复了.

```
class FoodDB {
    boolean nameMatchKeyword(String name, String keyword) {
        return keyword==null ||
                keyword.equals("") ||
                name.indexOf(keyword)!=-1;
    }
    public Iterator srchFood(String cName, String eName){
        TreeMap foodTree = new TreeMap();
        Iterator foodList;
        Food food;
        foodList = getAllRecords();
        while (foodList.hasNext()){
            food = (Food) foodList.next();
            if (nameMatchKeyword(food.getCName(), cName)
                &&
                nameMatchKeyword(food.getEName(), eName)) {
                foodTree.put(food.getFoodKey(),food);
            }
        }
        return foodTree.values().iterator();
    }
}
```

9. 指出并修改下面的重复代码:

```
class UIDialogCustomerMain extends JDialog {
    JButton btnOrderDel;
```

```
    JButton btnCustChangePassword;
    void bindEvents() {

        ...

        btnOrderDel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Dialog d = new UIDialogCustomerDeleteOrder(UIDialogCustomerMain.this, "Del Order", true);
                d.pack();
                d.setLocation(400,200);
                d.setVisible(true);
            }
        });
        btnCustChangePassword.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Dialog d = new UIChangeAccountPW(UIDialogCustomerMain.this, "chg pw", true);
                d.pack();
                d.setLocation(400,200);
                d.setVisible(true);
            }
        });
    }

    ...

}
class UIDialogRestaurantMain extends JDialog {
    JButton btnChangePassword;
    void bindEvents() {

        ...

    btnChangePassword.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Dialog d = new UIChangeAccountPW(UIDialogRestaurantMain.this, "chg pw", true);
                d.pack();
                d.setLocation(400,200);
                d.setVisible(true);
            }
        });
    }

    ...
```

```
 }
```

pack,设置位置跟显示对话框的代码重复了.改变密码的按钮也重复了.

```
 class UIDialogShower {
     public static void show(Dialog d, int left, int top) {
         d.pack();
         d.setLocation(left, top);
         d.setVisible(true);
     }

     public static void showAtDefaultPosition(Dialog d) {
         show(d, 400, 200);
     }
 }
class ChangePasswordButton extends JButton {
     public ChangePasswordButton(final JDialog enclosingDialog) {
        addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               Dialog d = new UIChangeAccountPW(enclosingDialog, "chg pw", true);
               UIDialogShower.showAtDefaultPosition(d);
            }
         });
     }
 }
class UIDialogCustomerMain extends JDialog {
     JButton btnOrderDel;
     ChangePasswordButton btnCustChangePassword = new ChangePasswordButton(this);
     void bindEvents() {

         ...

        btnOrderDel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Dialog d = new UIDialogCustomerDeleteOrder
                            (UIDialogCustomerMain.this, "Del Order", true);
               UIDialogShower.showAtDefaultPosition(d);
            }
         });
     }

     ...
```

```
  }
  class UIDialogRestaurantMain extends JDialog {
      ChangePasswordButton btnChangePassword = new ChangePasswordButton(this);
      void bindEvents() {

          ...

      }

      ...

  }
```

10.假定现在有两种出租种类:书和影碟. 指出并修改下面的重复代码:

```
  public class BookRental {
      private String bookTitle;
      private String author;
      private Date rentDate;
      private Date dueDate;
      private double rentalFee;
      public boolean isOverdue() {
          Date now=new Date();
           return dueDate.before(now);
      }
      public double getTotalFee() {
           return isOverdue() ? 1.2*rentalFee : rentalFee;
      }
  }
  public class MovieRental {
      private String movieTitle;
      private int classification;
      private Date rentDate;
      private Date dueDate;
      private double rentalFee;
      public boolean isOverdue() {
          Date now=new Date();
          return dueDate.before(now);
      }
      public double getTotalFee() {
          return isOverdue() ? 1.3*rentalFee : rentalFee;
      }
```

```
}
```

这边我们应该抽取一些共有的属性,比如 rentDate，dueDate，rentalFee 等等,还有，isOverdue()的这个方法重复了,getTotalFee()的方法也有重复的地方.

```
public abstract class Rental {
    private Date rentDate;
    private Date dueDate;
    private double rentalFee;
    protected abstract double getOverduePenaltyRate();
    public boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    public double getTotalFee() {
        return isOverdue() ? getOverduePenaltyRate()*rentalFee : rentalFee;
    }
}
public class BookRental extends Rental {
    private String bookTitle;
    private String author;
    protected double getOverduePenaltyRate() {
        return 1.2;
    }
}
public class MovieRental extends Rental {
    private String movieTitle;
    private int classification;
    protected double getOverduePenaltyRate() {
        return 1.3;
    }
}
```

第 2 章 将注释转换为代码

示例

这是一个会议管理系统. 在会议中,每个参会者都会戴一个牌子,这牌子上面有这个参会者的信息(比如姓名之类的).在这个系统中,Badge 这个类用来存放这个参会者的信息.请看一下下面的代码跟注释:

```
//存放参会者身上戴的牌子所显示的信息.
public class Badge {
    String pid;   //参会者 ID
    String engName; //英文全名
    String chiName; //中文全名
    String engOrgName; //所在部门英文名称
    String chiOrgName; //所在部门中文名称
    String engCountry; //部门所在国家的中文名称
    String chiCountry; //部门所在国家的英文名称

    //***********************
    //构造函数.
    //根据参会者的 id,去数据库取出该参与者的信息.
    //***********************
   Badge(String pid) {
       this.pid = pid;
       //***********************
       //取出参会者
       //***********************
       ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
       Participant part = partsInDB.locateParticipant(pid);
       if (part != null) {
          //取出参会者的英文全名
          engName = part.getELastName() + ", " + part.getEFirstName();
          //取出参会者的中文全名
          chiName = part.getCLastName()+part.getCFirstName();
          //***********************
          //取出所在部门跟国家.
          //***********************
          OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
          //取出所在部门的 id.
          String oid = orgsInDB.getOrganization(pid);
          if (oid != null) {
```

```
                Organization org = orgsInDB.locateOrganization(oid);
                engOrgName = org.getEName();
                chiOrgName = org.getCName();
                engCountry = org.getEAddress().getCountry();
                chiCountry = org.getCAddress().getCountry();
            }
        }
    }

    ...

}
```

将注释转换为代码,让代码足够清楚到可以表示注释

我们先看一下第一个注释:

```
//存放参会者身上戴的牌子所显示的信息.
public class Badge {

    ...

}
```

我们干嘛需要这个注释呢?因为程序员认为"Badge"这个类名不足以让读代码的人清楚这个类的作用,所以就写了这个注释. 那如果我们直接将注释所表达的一些信息放在类名里面的话,就没有单独写注释的必要了.比如::

```
public class ParticipantInfoOnBadge {

    ...

}
```

其实很多人肯定会问?难道写注释不是一个好的编程习惯吗?这问题很好,我也想知道.在解释之前,我们先把这个示例中所有的注释都转为代码先.

将注释转换为变量名

Consider:

```
public class ParticipantInfoOnBadge {
      String pid;   //参会者 ID
      String engName; //英文全名
      String chiName; //中文全名
      String engOrgName; //所在部门英文名称
      String chiOrgName; //所在部门中文名称
      String engCountry; //部门所在国家的中文名称
      String chiCountry; //部门所在国家的英文名称

      ...

}
```

这里,我们就像对属性的注释,转化为属性名, 比如:

```
public class ParticipantInfoOnBadge {
      String participantId;
      String participantEngFullName;
      String participantChiFullName;
      String engOrgName;
      String chiOrgName;
      String engOrgCountry;
      String chiOrgCountry;

      ...

  }
```

对参数的注释,转化为参数名

看看:

```
public class ParticipantInfoOnBadge {

      ...

      //***********************
      //构造函数.
```

```
        //根据参会者的 id,从数据库取出该参与者的信息.
        //***********************
        ParticipantInfoOnBadge(String pid) {
            this.pid = pid;

             ...

        }
    }
```

比如:

```
    public class ParticipantInfoOnBadge {

        ...

        //***********************
        //构造函数.
        //从数据库取出该参与者的信息.
        //***********************
        ParticipantInfoOnBadge(String participantId) {
            this.participantId = participantId;

             ...

        }
    }
```

将注释转换为方法的一部分

上面的构造函数中,有两句注释,第一句我们已经解决了,那么还有"从数据库取出该参与者的信息"? 这句注释描述了,这个构造函数是如何实现的(就是从数据库里面取出信息),我们将这句话转化:

```
    public class ParticipantInfoOnBadge {

        ...

        //***********************
        //构造函数.
        //***********************
```

```
    ParticipantInfoOnBadge(String participantId) {
        loadInfoFromDB(participantId);//现在,看一下这个构造函数内部,我们就能知道这个构造函数是做什么了吧.
    }
    void loadInfoFromDB(String participantId) {
        this.participantId = participantId;

        ...

    }
}
```

删掉没用的注释

有时候,我们会碰到一些注释,很明显没什么用处的,比如:

```
public class ParticipantInfoOnBadge {

    ...

    //***********************
    //构造函数.
    //***********************
    ParticipantInfoOnBadge(String participantId) {

        ...

    }
}
```

就算去掉这些注释,我们也能看得出来,这是个构造函数.这个注释并没什么用处.

什么样的类是看代码的人最喜欢的?那就是简单易看的类.一个设计得好的类,能够让人家一眼就能出你这个类都有些什么东西,明白你这个类都做了一些什么事.如果看这个类的时候,要不停的将屏幕滚来滚去,而思维还要随屏幕的滚动跳转,无形中,看懂这个类需要花的时间就多了.

一个屏幕,差不多只能显示 20 行左右的代码,而这个没用的注释,一下子就占用了 3 行的代码,一些有用的信息反而被挤掉了(比如说代码),得不偿失啊!我看还是赶紧移除这个注释:

```
public class ParticipantInfoOnBadge {

    ...
```

```
    ParticipantInfoOnBadge(String participantId) {

        ...

    }
}
```

将一部分代码重构成方法,用方法名来表达注释的意思

先看看下面这个注释:

```
void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    //************************
    //取得参会者的全名.
    //************************
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
    Participant part = partsInDB.locateParticipant(participantId);
    if (part != null) {
        //取得参会者的英文全名.
        engFullName = part.getELastName() + ", " + part.getEFirstName();
        //取得参会者的中文全名.
        chiFullName = part.getCLastName()+part.getCFirstName();
        //************************
        //取得参会者所在部门和国家.
        //************************
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        //取得参会者被雇佣部门的 id.
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}
```

现在我们已经看清这段注释要表达一些什么信息,如果要使代码跟注释一样清楚,我们可以将注释所解释的那部分代码抽取出来,做成一个方法,然后让方法名来表达注释的意思.如果可以的话,我们就不需要额外的注释了:

```
void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    getParticipantFullNames(); //(取得参会者的全名,注意,我们已经将注释去掉了.)
     //***********************
     //取得参会者所在部门和国家.
     //***********************
     //取得参会者被雇佣部门的 id.
    OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
    String oid = orgsInDB.getOrganization(participantId);
    if (oid != null) {
        Organization org = orgsInDB.locateOrganization(oid);
        engOrgName = org.getEName();
        chiOrgName = org.getCName();
        engOrgCountry = org.getEAddress().getCountry();
        chiOrgCountry = org.getCAddress().getCountry();
    }
}
void getParticipantFullNames() {
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
    Participant part = partsInDB.locateParticipant(participantId);
    if (part != null) {
        //取得参会者的英文全名.
        engFullName = part.getELastName() + ", " + part.getEFirstName();
        //取得参会者的中文全名.
        chiFullName = part.getCLastName()+part.getCFirstName();
    }
}
```

此外,还有一个注释:"取得参会者所在部门和国家",也是可以重构在方法名里面的:

```
void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    getParticipantFullNames();
    getOrgNameAndCountry(); //又抽取掉了一个注释
}
void getParticipantFullNames() {
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
    Participant part = partsInDB.locateParticipant(participantId);
    if (part != null) {
        //取得参会者的英文全名.
```

```
            engFullName = part.getELastName() + ", " + part.getEFirstName();
            //取得参会者的中文全名.
            chiFullName = part.getCLastName()+part.getCFirstName();
        }
    }
    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        //取得参会者被雇佣部门的 id.
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
```

抽取出方法,放于另一个类

请看一下下面这两个注释:

```
public class ParticipantInfoOnBadge {

    ...

    void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
            //取得参会者的英文全名.
            engFullName = part.getELastName() + ", " + part.getEFirstName();
            //取得参会者的中文全名.
            chiFullName = part.getCLastName()+part.getCFirstName();
        }
    }
}
```

因为程序员觉得,这些代码片段还是不够清楚,所以还是要用注释还解释它们. 但这回移除注释时,我们会将抽取出来的方法,放到 Participant 这个类里面,而不是 ParticipantInfoOnBadge 了:

```
public class ParticipantInfoOnBadge {

    ...

    void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
            engFullName = part.getEFullName(); //将职责交给 domain 自己,也就是 Participant.
            chiFullName = part.getCFullName();
        }
    }
}
public class Participant {
    String getEFullName() {
        return getELastName() + ", " + getEFirstName();
    }
    String getCFullName() {
        return getCLastName() + getCFirstName();
    }
}
```

用注释去命名一个已经存在的方法

请看下面的注释,也是这个例子中的最后一个注释了:

```
public class ParticipantInfoOnBadge {

    ...

    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        //取得参会者被雇佣部门的 id.
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
```

```
                chiOrgCountry = org.getCAddress().getCountry();
            }
        }
    }
```

我们之所以要用这个注释"取得参会者被雇佣部门的 id",是因为这个方法名"getOrganization"取得不够清楚.所以,我们将注释表达的信息,放在这个方法名里面:

```
public class ParticipantInfoOnBadge {

    ...

    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        String oid = orgsInDB.findOrganizationEmploying(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}

public class OrganizationsInDB {

    ...

    void findOrganizationEmploying(String participantId) {

        ...

    }
}
```

改进完的代码

看看下面这个改进完的代码,如果让一个人来读懂我们的的代码,现在要看懂多长时间?而重构之前,那个人又需要花多长时间?

```
public class ParticipantInfoOnBadge {
    String participantId;
```

```
    String participantEngFullName;
    String participantChiFullName;
    String engOrgName;
    String chiOrgName;
    String engOrgCountry;
    String chiOrgCountry;

    ParticipantInfoOnBadge(String participantId) {
        loadInfoFromDB(participantId);
    }
    void loadInfoFromDB(String participantId) {
        this.participantId = participantId;
        getParticipantFullNames();
        getOrgNameAndCountry();
    }
    void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
            participantEngFullName = part.getEFullName();
            participantChiFullName = part.getCFullName();
        }
    }
    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        String oid = orgsInDB.findOrganizationEmploying(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}
```

为什么要删除额外的注释?

为什么要删除额外的注释?其实,加注释本身是一个很正确的事情,因为注释可以让人更容易的理解代码.因为程序员觉得光光代码,可能还不能让人完全理解.

而问题就在于,因为常常没有把代码写清楚,所以我们就找了一个捷径,那就是写上注释.注释不够清楚,再写上文档.好,这样子,程序可以让人看得懂了吧.

而这样造成的结果就是,没有人愿意去好好的组织代码,让代码清楚起来,因为他们觉得加上注释就好了.

之后,代码更新了,可是程序员却常常忘了去更新注释(我们不得不承认,这种情况经常发生,这点,再好的软件工程,都会有这种问题的存在).

过了一段时间,这些过时的注释不仅不能让代码更容易懂,反而会误导了读代码的人.

到了最后,我们剩下的东西就是:本身就不清晰的代码,混上一些不正确的注释.

因此,不管任何时候,当我们要加注释的时候,我们应该再三的想想:

我的注释能不能转化在代码里面,让代码跟注释一样的清晰?

而大多数情况下,我们得到的答案都是:能!

每一个注释都是一个改进代码的好机会!

我们不能说,注释少的代码就是好代码.

但我们绝对可以说,**包含太多注释的代码,绝对不是高质量的代码.**

方法名太长

请参看下面的例子:

```
class StockItemsInDB {
//找出所有数目少于 10 的海外存货
    StockItem[] findStockItems() {

        ...

    }
}
```

如果要将这段注释转为代码的话,原则上,我们会改成这样:

```
class StockItemsInDB {
    StockItem[] findStockItemsFromOverseasWithInventoryLessThan10() {

        ...

    }
}
```

可惜的是,这个方法名实在是太长了.别急,这并不是表明我们不能去掉注释,而相反,这是在警告我们,代码有问题了.

有什么问题呢?

有什么问题我们先不说,我们先看一下,在这种情况下,要怎么做?现在我们可以判断的是:这个系统的用户确实只对海外的那些数目少于 10 的存货有兴趣吗?它有没有可能对那些数目少于 20 的存货有兴趣?然后他对本地的存货就没兴趣了?或者超过 25 的存货有没有兴趣呢?如果你回答说,管他的,反正他现在只要海外的数目少于 10 的,我们不要那么多事----*抽死你,这程序员是谁招进来的*?!
这里的注释,我们可以转化到参数里面:

```
class StockItemsInDB {
    StockItem[] findStockItemsWithFeatures(
        boolean isFromOverseas,
         InventoryRange inventoryRange) {
         ...

    }
 }
class InventoryRange {
    int minimumInventory;
    int maximumInventory;
 }
```

好,话说得太独断了.如果客户他真是只对海外的少于 10 的存货有兴趣呢? 那他肯定有一些特殊的理由(为什么只是这些条件,而不是其他的呢?).所以聊了一下,我们发现,因为海运花太长时间了,所以他需要补充这些存货.此时,我们发现,客户真正有兴趣的,是那些需要补充的存货,而不仅仅是那些海外的数目少于 10 的.因此,我们可以将它真正的意思转在方法名跟方法的实现里面:

```
class StockItemsInDB {
    StockItem[] findStockItemsToReplenish() {
        StockItem stockItems[];
        stockItems = findStockItemsFromOverseas();
        stockItems = findStockItemsInventoryLessThan10(stockItems);
        return stockItems;
    }
 }
```

章节练习

问题

1. 将注释转化为代码:

```
class InchToPointConvertor {
    //将数量由英寸转为磅
    static float parseInch(float inch) {
        return inch * 72;   //一英寸包括 72 磅.
    }
}
```

2. 将注释转化为代码:

```
class Restaurant extends Account {

    //字符串"Rest"是作为前缀来组成饭店的 id
    final static String RestaurantIDText = "Rest";
    String website;
    //中文地址
    String addr_cn;
    //英文地址
    String addr_en;
    //下面是最新的传真号码,饭店想要更新它的传真号,等确认下面的新传真号是正确的以后
    //新传真号就会存放在 Account 中,在这之前,都会存放在这个位置

    String numb_newfax;
    boolean has_NewFax = false;
    //存放假日的一个集合
    Vector Holiday; // a holiday
    //该饭店所属分类的 id
    String catId;
    //营业时段的集合,每个营业时段都是一个数组,
    //该数组包括两个时间,前面一个表示开始时间,后面一个表示结束时间
    //饭店在所有的时间段内营业
    Vector BsHour; //营业时段集

    ...

    //y:  年.
    //m:  月.
    //d:  日.
    void addHoliday(int y,int m,int d){
        if(y<1900) y+=1900;
        Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
        Holiday.add(aHoliday);
    }
    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
```

```
            int fMin = fromHr*60 + fromMin;    //以分表示的开始时间
            int tMin = toHr*60 + toMin;              //以分表示的结束时间
        //确定所有的时间都是有效的,而且结束时间比开始时间晚
            if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
                GregorianCalendar bs[] = {
                    new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
                    new GregorianCalendar(1900,1,1, toHr, toMin,0)
                };
                BsHour.add(bs);
                return true;
            } else {
                return false;
            }
        }
    }
```

3. 将注释转化为代码:

```
    class Account {

        ...

        //检查这个密码是否足够复杂,比如:
        //,包括字母,数字或特殊字符
        boolean isComplexPassword(String password){
            //包含一个数字或者特殊字符
            boolean dg_sym_found=false;
            //包含一个字母
            boolean letter_found=false;
            for(int i=0; i<password.length(); i++){
                char c=password.charAt(i);
                if(Character.isLowerCase(c)||Character.isUpperCase(c))
                    letter_found=true;
                else dg_sym_found=true;
            }
            return (letter_found) && (dg_sym_found);
        }
    }
```

4. 将注释转化为代码:

```
    class TokenStream {
        Vector v;    //由 br 解析出来的词符集(Vector).
```

```
    int index; //当前的词符序号.
    BufferedReader br; //从这里,读取字符串,解析出所有词符.
    int currentChar;   //从 br 解析出来的前一个字符.

    //从 reader 中读取字符,解析出所有词符
    TokenStream(Reader read) {
        br = new BufferedReader(read);
        takeChar();
        v = parseFile();
        index=0;
    }
    //从 br 中读取字符,解析出所有词符并存放在一个 venctor 中.
    Vector parseFile() {
       Vector v; //堆积所有已经解析出的词符

        ...

        return v;
    }

    ...

}
```

5. 将注释转化为代码:

```
class FoodDB {
    //找出所有名字都包括这两个关键字的食物,返回一个迭代器(Iterator)
    public Iterator srchFood(String cName, String eName){
        //it contains all the foods to be returned.
        TreeMap foodTree = new TreeMap();
        Iterator foodList;
        Food food;
        foodList = getAllRecords();
        while (foodList.hasNext()){
            food = (Food) foodList.next();
            //有没有包括两个关键字?
            if ((cName==null || //null 或者 "" 表示没有关键字
                cName.equals("") ||
                food.getCName().indexOf(cName)!=-1)
                &&
                (eName==null || //null 或者 "" 表示没有关键字
```

```
                    eName.equals("") ||
                    food.getEName().indexOf(eName)!=-1)){
                    foodTree.put(food.getFoodKey(),food);
                }
            }
            return foodTree.values().iterator();
        }
    }
```

6. 将注释转化为代码:

```
    //一个订单.
    class Order {
        String orderId; //订单 id.
        Restaurant r1;   //预订哪个饭店.
        Customer c1;   //预订客户.
        //"H":  送到客户家庭住址.
        //"W":  送到客户办公地址.
        //"O":  送到客户特定的地址.
        String addressType;
        String otherAddress; //如果 addressType 是"O"的话,这里存放特定的地址.
        HashMap orderItems;   //预订的东西.

        //取得订单总额.
        public double getTotalAmt() {
            //总计.
            BigDecimal amt= new BigDecimal("0.00");
            //遍历每个预订的东西
            Iterator iter=orderItems.values().iterator();
            while(iter.hasNext()){
                //增加下一样东西的金额
                OrderItem oi=(OrderItem)iter.next();
                amt = amt.add(new BigDecimal(String.valueOf(oi.getAmount())));
            }
            return amt.doubleValue();
        }
    }
```

7. 找出一些包括注释的代码,然后将它们转化为代码.

解决方法示例

1. 将注释转化为代码:

```
class InchToPointConvertor {
    //将数量由英寸转为磅
    static float parseInch(float inch) {
        return inch * 72;   //一英寸包括 72 磅.
    }
}
```

为什么不把这个方法名叫做 "convertToPoints"呢?而且,我们也可以把 72 这个值放在一个变量中.

将第一行注释转为方法名,将第二行注释放在一个变量名中:

```
class InchToPointConvertor {
    final static int POINTS_PER_INCH=72;
    static float convertToPoints(float inch) {
        return inch * POINTS_PER_INCH;
    }
}
```

2. 将注释转化为代码:

```
class Restaurant extends Account {

    //字符串"Rest"是作为前缀来组成饭店的 id
    final static String RestaurantIDText = "Rest";
    String website;
    //中文地址
    String addr_cn;
    //英文地址
    String addr_en;
    //下面是最新的传真号码,饭店想要更新它的传真号,等确认下面的新传真号是正确的以后
    //新传真号就会存放在 Account 中,在这之前,都会存放在这个位置

    String numb_newfax;
    boolean has_NewFax = false;
    //存放假日的一个集合
    Vector Holiday; // a holiday
    //该饭店所属分类的 id
```

```
        String catId;
        //营业时段的集合,每个营业时段都是一个数组,
        //该数组包括两个时间,前面一个表示开始时间,后面一个表示结束时间
        //饭店在所有的时间段内营业
        Vector BsHour; //营业时段集

        ...

        //y: 年.
        //m: 月.
        //d: 日.
        void addHoliday(int y,int m,int d){
            if(y<1900) y+=1900;
            Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
            Holiday.add(aHoliday);
        }
        public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
            int fMin = fromHr*60 + fromMin;   //以分表示的开始时间
            int tMin = toHr*60 + toMin;           //以分表示的结束时间
        //确定所有的时间都是有效的,而且结束时间比开始时间晚
          if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
              GregorianCalendar bs[] = {
                  new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
                  new GregorianCalendar(1900,1,1, toHr, toMin,0)
              };
             BsHour.add(bs);
              return true;
          } else {
              return false;
          }
        }
    }
```

这里最麻烦的问题,就是对命名为"BsHour"的这个 Vector 的注释,这个注释很长,要表达的信息也很多.

注释的信息多,证明营业时段拥有的职责也多.而在这里,营业时段的职责,只完全可以分离出来的.我们要将它分离出来,放在另一个类里面定义,类名叫做 BusinessSession:

```
class BusinessSession { //(现在,结构又清楚很多了)
    int minStart;
    int minEnd;
    BusinessSession(int fromHour, int fromMinute, int toHour, int toMinute) {
        minStart=getMinutesFromMidNight(fromHour, fromMinute);
        minEnd=getMinutesFromMidNight(toHour, toMinute);
```

```
    }
    int getMinutesFromMidNight(int hours, int minutes) {
        return hours*60+minutes;
    }
    boolean isMinutesWithinOneDay(int minutes) {
        return minutes>0 && minutes<=1440;
    }
    boolean isValid() {
        return isMinutesWithinOneDay(minStart) &&
                isMinutesWithinOneDay(minEnd) &&
                minStart<minEnd;
    }
}
class Restaurant extends Account {
    final static String keyPrefix="Rest";
    String website;
    String chineseAddr;
    String englishAddr;
    String faxNoUnderConfirmation;
    boolean hasFaxNoUnderConfirmation = false;
    Vector holidayList;
    String catIdForThisRestaurant;
    Vector businessSessionList;

    ...

    void addHoliday(int year, int month, int day){
        if(year<1900) year+=1900;
        Calendar aHoliday = (new GregorianCalendar(year,month,day,0,0,0));
        holidayList.add(aHoliday);
    }
    public boolean addBusinessSession(int fromHr, int fromMin, int toHr, int toMin){
        BusinessSession bs=new BusinessSession(fromHr, fromMin, toHr, toMin);
        if(bs.isValid()){
            businessSessionList.add(bs);
            return true;
        } else {
            return false;
        }
    }
}
```

3. 将注释转化为代码:

```
class Account {

    ...

    //检查这个密码是否足够复杂,比如:
    //包括字母,数字或特殊字符
    boolean isComplexPassword(String password){
        //包含一个数字或者特殊字符
        boolean dg_sym_found=false;
        //包含一个字母
        boolean letter_found=false;
        for(int i=0; i<password.length(); i++){
            char c=password.charAt(i);
            if(Character.isLowerCase(c)||Character.isUpperCase(c))
                letter_found=true;
            else dg_sym_found=true;
        }
        return (letter_found) && (dg_sym_found);
    }
}
```

先看前一个注释,包含两句.第一句("检查这个密码是否足够复杂")描述了这个方法的作用,应该作为方法名. 而原来的方法名(isComplexPassword)刚刚好表达了它的意思,所以,我们可以将这句注释移除.第二句("包括字母,数字或特殊字符") 描述的则是这个方法所做的一些事情,所以它应该在方法的实现里面,而不是方法名上.

```
class Account {

    ...

    boolean isComplexPassword(String password){
        return containsLetter(password) &&
                (containsDigit(password) || containsSymbol(password));
    }
    boolean containsLetter(String password) {

        ...

    }
    boolean containsDigit(String password) {

        ...
```

```
    }
    boolean containsSymbol(String password) {

        ...

    }
}
```

4. 将注释转化为代码:

```
class TokenStream {
    Vector v;   //由 br 解析出来的词符集(Vector).
    int index; //当前的词符序号.
    BufferedReader br; //从这里,读取字符串,解析出所有词符.
    int currentChar;   //从 br 解析出来的前一个字符.

    //从 reader 中读取字符,解析出所有词符
    TokenStream(Reader read) {
        br = new BufferedReader(read);
        takeChar();
        v = parseFile();
        index=0;
    }
    //从 br 中读取字符,解析出所有词符并存放在一个 venctor 中.
    Vector parseFile() {
        Vector v; //堆积所有已经解析出的词符

        ...

        return v;
    }

    ...

}
```

这些注释可以转化为变量名,参数名,跟方法. 对于描述构造函数所做的事的这个注释,可以转化到构造函数的实现里面.

```
class TokenStream {
    Vector parsedTokenList;
    int currentTokenIdxInList;
    BufferedReader charInputSourceForParsing;
```

```
        int previousCharReadFromSource;

        TokenStream(Reader reader) {
            charInputSourceForParsing = new BufferedReader(reader);
            takeChar();
            parsedTokenList = parseTokensFromInputSource();
            currentTokenIdxInList = 0;
        }
        Vector parseTokensFromInputSource() {
           Vector tokensParsedSoFar;

            ...

            return tokensParsedSoFar;
        }

        ...

    }
```

5. 将注释转化为代码:

```
    class FoodDB {
        //找出所有名字都包括这两个关键字的食物,返回一个迭代器(Iterator)
        public Iterator srchFood(String cName, String eName){
            //it contains all the foods to be returned.
            TreeMap foodTree = new TreeMap();
            Iterator foodList;
            Food food;
            foodList = getAllRecords();
            while (foodList.hasNext()){
               food = (Food) foodList.next();
               //有没有包括两个关键字?
               if ((cName==null || //null 或者 "" 表示没有关键字
                   cName.equals("") ||
                   food.getCName().indexOf(cName)!=-1)
                   &&
                   (eName==null || //null 或者 "" 表示没有关键字
                   eName.equals("") ||
                   food.getEName().indexOf(eName)!=-1)){
                   foodTree.put(food.getFoodKey(),food);
               }
            }
```

```
            return foodTree.values().iterator();
        }
    }
```

将这些注释转化为变量名,参数名和方法

```
    class FoodDB {
        public Iterator findFoodsWithKeywordsInNames(String cKeyword, String
eKeyword){
            TreeMap foodsFound = new TreeMap();
            for (Iterator foodIter=getAllRecords(); foodIter.hasNext(); ){
                Food food = (Food) foodIter.next();
                if (foodContainsKeyInNames(food, cKeyword, eKeyword)) {
                    foodsFound.put(food.getFoodKey(),food);
                }
            }
            return foodsFound.values().iterator();
        }
        boolean foodContainsKeyInNames(
            Food food,
            String cKeyword,
            String eKeyword) {
            return nameContainsKeyword(food.getCName(), cKeyword) &&
                    nameContainsKeyword(food.getEName(), eKeyword);
        }
        boolean nameContainsKeyword(String name, String keyword) {
            return keywordIsNotSpecified(keyword) || name.indexOf(keyword)!=-1;
        }
        boolean keywordIsNotSpecified(String keyword) {
            return keyword==null || keyword.equals("");
        }
    }
```

6. 将注释转化为代码:

```
    //一个订单.
    class Order {
        String orderId; //订单 id.
        Restaurant r1;  //预订哪个饭店.
        Customer c1;  //预订客户.
        //"H": 送到客户家庭住址.
        //"W": 送到客户办公地址.
        //"O": 送到客户特定的地址.
```

```
        String addressType;
        String otherAddress; //如果 addressType 是"O"的话,这里存放特定的地址.
        HashMap orderItems;   //预订的东西.

        //取得订单总额.
        public double getTotalAmt() {
            //总计.
            BigDecimal amt= new BigDecimal("0.00");
            //遍历每个预订的东西
            Iterator iter=orderItems.values().iterator();
            while(iter.hasNext()){
                //增加下一样东西的金额
                OrderItem oi=(OrderItem)iter.next();
                amt = amt.add(new BigDecimal(String.valueOf(oi.getAmount())));
            }
            return amt.doubleValue();
        }
    }
```

将这些注释转化为变量名,参数名和方法

```
class Order {
        String orderId;
      Restaurant restToReceiveOrder;
      Customer customerPlacingOrder;
      static final String DELIVER_ADDRESS_TYPE_HOME="H";
      static final String DELIVER_ADDRESS_TYPE_WORK="W";
      static final String DELIVER_ADDRESS_TYPE_OTHER="O";
        String addressType;
        String  otherAddress; //如果 addressType 是 DELIVER_ADDRESS_TYPE_OTHER=的话,这里存放特定的地址. .
      HashMap orderItems;

      Iterator getItemsIterator() {
          return orderItems.values().iterator();
        }
      public double getTotalAmount() {
          BigDecimal totalAmt= new BigDecimal("0.00");
          for (Iterator iter=getItemsIterator(); iter.hasNext(); ){
              OrderItem nextOrderItem=(OrderItem)iter.next();
              totalAmt =
                  totalAmt.add(
                      new BigDecimal(String.valueOf(nextOrderItem.getAmount())));
          }
```

```
            return totalAmt.doubleValue();
        }
    }
```

现在我们注意到,还有一个注释留着.除非我们移除 type code,否则这个注释非常难以移除.
如何移除 type codes, 可以看第 3 章:去除代码的异味。

第 3 章 除去代码异味

示例

异味这个词,可能有点抽象,我们先看一下下面的例子

这是一个 CAD 系统. 现在,它已经可以画三种形状了:线条,长方形,跟圆.
先认真的看一下下面的代码:

```
class Shape {
    final static int TYPELINE = 0;
    final static int TYPERECTANGLE = 1;
    final static int TYPECIRCLE = 2;
    int shapeType;
    //线条的开始点
    //长方形左下角的点
    //圆心
    Point p1;
    //线条的结束点
    //长方形的右上角的点
    //如果是圆的话,这个属性不用
    Point p2;
    int radius;
}
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            switch (shapes[i].getType()) {
                case Shape.TYPELINE:
                    graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
                    break;
                case Shape.TYPERECTANGLE:
```

```
                    //画四条边
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    break;
                case Shape.TYPECIRCLE:
                    graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
                    break;
            }
        }
    }
}
```

代码都是一直在改变的,而这也是上面的代码会碰到的一个问题.

现在我们有一个问题: 如果我们需要支持更多的形状(比如三角形), 那么肯定要改动 Shape 这个类, CADApp 里面的 drawShapes 这个方法也要改.

好,改为如下的样子:

```
class Shape {
    final static int TYPELINE = 0;
    final static int TYPERECTANGLE = 1;
    final static int TYPECIRCLE = 2;
    final static int TYPETRIANGLE = 3;
    int shapeType;
    Point p1;
    Point p2;
    //三角形的第三个点.
    Point p3;
    int radius;
}
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            switch (shapes[i].getType()) {
                case Shape.TYPELINE:
                    graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
                    break;
                case Shape.TYPERECTANGLE:
                    //画四条边.
                    graphics.drawLine(...);
                    graphics.drawLine(...);
```

```
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    break;
                case Shape.TYPECIRCLE:
                    graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
                    break;
                case Shape.TYPETRIANGLE:
                    graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
                    graphics.drawLine(shapes[i].getP2(), shapes[i].getP3());
                    graphics.drawLine(shapes[i].getP3(), shapes[i].getP1());
                    break;
            }
        }
    }
}
```

如果以后要支持更多的形状,这些类又要改动……，这可不是什么好事情！
理想情况下，我们希望当一个类，一个方法或其他的代码设计完以后，就不用再做修改了。它们应该稳定到不用修改就可以重用。

现在的情况恰好相反！

每当我们增加新的形状，都得修改 Shape 这个类，跟 CADApp 里面的 drawShapes 方法。

怎么让代码稳定（也就是无需修改）？这个问题是个好问题！不过老规矩，先不说，我们以行动回答。

我们先看看另外一个方法： 当给你一段代码，你怎么知道它是稳定的？

怎么判断代码的稳定性？

要判断代码的稳定性，我们可能会这样来判定：先假设一些具体的情况或者需求变动了，然后来看一看，要满足这些新的需求，代码是否需要被修改？

可惜，这也是一件很麻烦的事，因为有那么多的可能性！我们怎么知道哪个可能性要考虑，哪些不用考虑？

有个更简单的方法，如果发现说，我们已经第三次修改这些代码了，那我们就认定这些代码是不稳定的。

这样的方法很“懒惰”，而且“被动”！因为这是在我们被伤到了以后，才开始处理问题。虽然这种方法还算是一个很有效的方法。

此外，还有一个简单，而且“主动”的方法：如果这段代码是不稳定或者有一些潜在问题的，那么代码往往会包含一些明显的痕迹。正如食物要腐坏之前，经常会发出一些异味一样（当然，食物如果有异味了，再怎么处理我们都不想吃了。但是代码可不行。）。我们管这些痕迹叫做“代码异味”。并不是所有的食物有异味都不能吃了，但大多数情况下，确实是不能吃了。并不是所有的代码异味都是坏事，但大多数情况下，它们确实是坏事情！

因此，当我们感觉出有代码异味时，我们必须小心谨慎的检查了。

现在，我们来看看上面例子中的代码异味吧。

示例代码中的代码异味：

第一种异味：代码用了类别代码（type code）。

```
class Shape {
    final int TYPELINE = 0;
    final int TYPERECTANGLE = 1;
    final int TYPECIRCLE = 2;
    int shapeType;
    ...
}
```

这样的异味，是一种严肃的警告：我们的代码可能有许多问题。

第二种异味：Shape 这个类有很多属性有时候是不用的。例如，radius 这个属性只有在这个 Shape 是个圆的时候才用到：

```
class Shape {
    ...
    Point p1;
    Point p2;
    int radius; //有时候不用
}
```

第三种异味：我们想给 p1,p2 取个好一点的变量名都做不到，因为不同的情况下，它们有不同的含义：

```
class Shape {
    ...
    Point p1; //要取作“起始点”，“左下点”，还是“圆心”？
    Point p2;
}
```

第四种异味：drawShapes 这个方法里面，有个 switch 表达式。当我们用到 switch（或者一大串的 if-then-else-if）时，小心了。switch 表达式经常是跟类别代码（type code）同时出现的。

现在，让我们将这个示例中的代码异味消除吧。

消除代码异味：怎么去掉类别代码（type code）

大多数情况下，要想去掉一个类别代码，我们会为每一种类别建立一个子类，比如：
（当然，并不是每次要去掉一个类别代码都要增加一个新类，我们下面的另一个例子里面会讲另一种解决方法）

```
class Shape {
}
class Line extends Shape {
    Point startPoint;
    Point endPoint;
}
class Rectangle extends Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
}
class Circle extends Shape {
    Point center;
    int radius;
}
```

因为现在没有类别代码了，drawShapes 这个方法里面，就要用 instanceof 来判断对象是哪一种形状了。因此，我们不能用 switch 了，而要改用 if-then-else：

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                Line line = (Line)shapes[i];
                graphics.drawLine(line.getStartPoint(),line.getEndPoint());
            } else if (shapes[i] instanceof Rectangle) {
                Rectangle rect = (Rectangle)shapes[i];
                graphics.drawLine(...);
                graphics.drawLine(...);
                graphics.drawLine(...);
                graphics.drawLine(...);
            } else if (shapes[i] instanceof Circle) {
                Circle circle = (Circle)shapes[i];
                graphics.drawCircle(circle.getCenter(), circle.getRadius());
            }
        }
    }
```

```
}
```

因为没有类别代码了，现在每个类（Shape,Line,Rectangle,Circle）里面的所有属性就可以保证任何情况都是必需的了。现在我们也可以给它们取一些好听点的名字了（比如在 Line 里面，p1 这个属性可以改名为 startPoint 了）。现在四种异味只剩一种了，那就是，在 drawShapes 里面还是有一大串 if-then-else-if。我们下一步，就是要去掉这长长的一串。

消除代码异味：如何去掉一大串 if-then-else-if（或者 switch）

经常地，为了去掉 if-then-else-if 或者 switch，我们需要先保证在每个条件分支下的要写的代码是一样的。在 drawShapes 这个方法里面，我们先以一个较抽象的方法（伪码）来写吧：

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                画线条;
            } else if (shapes[i] instanceof Rectangle) {
                画长方形;
            } else if (shapes[i] instanceof Circle) {
                画圆;
            }
        }
    }
}
```

条件分支下的代码还是不怎么一样，不如再抽象一点：

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                画出形状;
            } else if (shapes[i] instanceof Rectangle) {
                画出形状;
            } else if (shapes[i] instanceof Circle) {
                画出形状;
            }
        }
    }
```

```
}
```

好，现在三个分支下的代码都一样了。我们也就不需要条件分支了：

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            画出形状;
        }
    }
}
```

最后，将“画出形状”这个伪码写成代码吧:

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            shapes[i].draw(graphics);
        }
    }
}
```

当然，我们需要在每种 Shape 的类里面提供 draw 这个方法：

```
abstract class Shape {
    abstract void draw(Graphics graphics);
}
class Line extends Shape {
    Point startPoint;
    Point endPoint;
    void draw(Graphics graphics) {
        graphics.drawLine(getStartPoint(), getEndPoint());
    }
}
class Rectangle extends Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
    void draw(Graphics graphics) {
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
    }
}
```

```
class Circle extends Shape {
    Point center;
    int radius;
    void draw(Graphics graphics) {
        graphics.drawCircle(getCenter(), getRadius());
    }
}
```

将抽象类变成接口

现在，看一下 Shape 这个类，它本身没有实际的方法。所以，它更应该是一个接口：

```
interface Shape {
    void draw(Graphics graphics);
}
class Line implements Shape {
    ...
}
class Rectangle implements Shape {
    ...
}
class Circle implements Shape {
    ...
}
```

改进后的代码

改进后的代码就像下面这样：

```
interface Shape {
    void draw(Graphics graphics);
}
class Line implements Shape {
    Point startPoint;
    Point endPoint;
    void draw(Graphics graphics) {
        graphics.drawLine(getStartPoint(), getEndPoint());
    }
}
class Rectangle implements Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
```

```
    void draw(Graphics graphics) {
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
    }
}
class Circle implements Shape {
    Point center;
    int radius;
    void draw(Graphics graphics) {
        graphics.drawCircle(getCenter(), getRadius());
    }
}
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            shapes[i].draw(graphics);
        }
    }
}
```

现在如果我们想要支持更多的图形（比如：三角形），上面的所有类都不用修改。我们只需要创建一个新的类 Triangle 就行了。

另一个例子

让我们来看一下另外一个例子。在当前的系统中，有三种用户：常规用户，管理员和游客。

常规用户必须每隔 90 天修改一次密码（更频繁也行）。管理员必须每 30 天修改一次密码。游客就不需要修改了。

常规用户跟管理员可以打印报表。

先看一下当前的代码：

```
class UserAccount {
    final static int USERTYPE_NORMAL = 0;
    final static int USERTYPE_ADMIN = 1;
    final static int USERTYPE_GUEST = 2;
    int userType;
    String id;
```

```
        String name;
        String password;
        Date dateOfLastPasswdChange;
        public boolean checkPassword(String password) {
            ...
        }
    }
    class InventoryApp {
        void login(UserAccount userLoggingIn, String password) {
            if (userLoggingIn.checkPassword(password)) {
                GregorianCalendar today = new GregorianCalendar();
                GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
                if (today.after(expiryDate)) {
                    //提示用户修改密码
                    ...
                }
            }
        }
        GregorianCalendar getAccountExpiryDate(UserAccount account) {
            int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
            GregorianCalendar expiryDate = new GregorianCalendar();
            expiryDate.setTime(account.dateOfLastPasswdChange);
            expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
            return expiryDate;
        }
        int getPasswordMaxAgeInDays(UserAccount account) {
            switch (account.getType()) {
                case UserAccount.USERTYPE_NORMAL:
                    return 90;
                case UserAccount.USERTYPE_ADMIN:
                    return 30;
                case UserAccount.USERTYPE_GUEST:
                    return Integer.MAX_VALUE;
            }
        }
        void printReport(UserAccount currentUser) {
            boolean canPrint;
            switch (currentUser.getType()) {
                case UserAccount.USERTYPE_NORMAL:
                    canPrint = true;
                    break;
                case UserAccount.USERTYPE_ADMIN:
                    canPrint = true;
```

```
                break;
            case UserAccount.USERTYPE_GUEST:
                canPrint = false;
        }
        if (!canPrint) {
            throw new SecurityException("You have no right");
        }
        //打印报表
    }
}
```

用一个对象代替一种类别（注意，之前是一个类代替一种类别）

根据之前讲的解决方法，要去掉类别代码，我们只需要为每种类别创建一个子类，比如：

```
abstract class UserAccount {
    String id;
    String name;
    String password;
    Date dateOfLastPasswdChange;
    abstract int getPasswordMaxAgeInDays();
    abstract boolean canPrintReport();
}
class NormalUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return 90;
    }
    boolean canPrintReport() {
        return true;
    }
}
class AdminUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return 30;
    }
    boolean canPrintReport() {
        return true;
    }
}
class GuestUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return Integer.MAX_VALUE;
    }
```

```
        boolean canPrintReport() {
            return false;
        }
    }
```

但问题是，三种子类的行为（里面的代码）都差不多一样，getPasswordMaxAgeInDays 这个方法就一个数值不同（30,90 或者 Integer.MAX_VALUE)。canPrintReport 这个方法也不同在一个数值（true 或 false)。这三种用户类型只需要用三个对象代替就行了，无须特地新建三个子类了：

```
    class UserAccount {
        UserType userType;
        String id;
        String name;
        String password;
        Date dateOfLastPasswdChange;
        UserType getType() {
            return userType;
        }
    }
    class UserType {
        int passwordMaxAgeInDays;
        boolean allowedToPrintReport;
        UserType(int passwordMaxAgeInDays, boolean allowedToPrintReport) {
            this.passwordMaxAgeInDays = passwordMaxAgeInDays;
            this.allowedToPrintReport = allowedToPrintReport;
        }
        int getPasswordMaxAgeInDays() {
            return passwordMaxAgeInDays;
        }
        boolean canPrintReport() {
            return allowedToPrintReport;
        }
        static UserType normalUserType = new UserType(90, true);
        static UserType adminUserType = new UserType(30, true);
        static UserType guestUserType = new UserType(Integer.MAX_VALUE, false);
    }
    class InventoryApp {
        void login(UserAccount userLoggingIn, String password) {
            if (userLoggingIn.checkPassword(password)) {
                GregorianCalendar today = new GregorianCalendar();
                GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
                if (today.after(expiryDate)) {
                    //提示用户修改密码
```

```
                ...
            }
        }
    }
    GregorianCalendar getAccountExpiryDate(UserAccount account) {
        int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
        GregorianCalendar expiryDate = new GregorianCalendar();
        expiryDate.setTime(account.dateOfLastPasswdChange);
        expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
        return expiryDate;
    }
    int getPasswordMaxAgeInDays(UserAccount account) {
        return account.getType().getPasswordMaxAgeInDays();
    }
    void printReport(UserAccount currentUser) {
        boolean canPrint;
        canPrint = currentUser.getType().canPrintReport();
        if (!canPrint) {
            throw new SecurityException("You have no right");
        }
        //打印报表.
    }
}
```

注意到了吧，用一个对象代替类别，同样可以移除 switch 或者 if-then-else-if。

总结一下类别代码的移除

要移动一些类别代码和 switch 表达式，有两种方法：

1.用基于同一父类的不同子类来代替不同的类别。

2.用一个类的不同对象来代替不同的类别。

当不同的类别具有比较多不同的行为时，用第一种方法。当这些类别的行为非常相似，或者只是差别在一些值上面的时候，用第二个方法。

普遍的代码异味

类别代码和 switch 表达式是比较普遍的代码异味。此外，还有其他的代码异味也很普遍。

下面是大概的异味列表：

代码重复

太多的注释
类别代码（type code）
switch 或者一大串 if-then-else-if
想给一个变量，方法或者类名取个好名字时，也怎么也取不好
用类似 XXXUtil, XXXManager, XXXController 和其他的一些命名
在变量，方法或类名中使用这些单词“And”,“Or”等等
一些实例中的变量有时有用，有时没用
一个方法的代码太多，或者说方法太长
一个类的代码太多，或者说类太长
一个方法有太多参数
两个类都引用了彼此（依赖于彼此）

引用：
http://c2.com/cgi/wiki?CodeSmell.

开闭原则（Open Closed Principle）：如果我们需要增加新的功能，我们只需要增加新的代码，而不是改变原有的。移除 switch 和类别代码是达到开闭原则的普遍方法。对开闭原则有兴趣的话，可以看：

http://www.objectmentor.com/publications/ocp.pdf 。

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.

Bertrand Meyer, Object-Oriented Software Construction, Pearson Higher Education, 1988.

Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring:

Improving the Design of Existing Code, Addison-Wesley, 1999. This book lists many code smells and the refactoring methods. Martin Fowler's web site about refactoring: http://www.refactoring.com/catalog/index.html.

章节练习

介绍

有一些问题，刚好也可以测试一些在前面两章的观点。

问题

1.指出并消除下面代码里的异味：

```
class FoodSalesReport {
    int q0; //卖了多少米？
    int q1; //卖了多少面？
    int q2; //卖了多少饮料？
```

```
        int q3; //卖了多少甜点？
        void LoadData(Connection conn) {
            PreparedStatement st = conn.prepareStatement("select "+
                "sum(case when foodType=0 then qty else 0 end) as totalQty0,"+
                "sum(case when foodType=1 then qty else 0 end) as totalQty1,"+
                "sum(case when foodType=2 then qty else 0 end) as totalQty2,"+
                "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
                "from foodSalesTable group by foodType");
            try {
                ResultSet rs = st.executeQuery();
                try {
                    rs.next();
                    q0 = rs.getInt("totalQty0");
                    q1 = rs.getInt("totalQty1");
                    q2 = rs.getInt("totalQty2");
                    q3 = rs.getInt("totalQty3");
                } finally {
                    rs.close();
                }
            } finally {
                st.close();
            }
        }
    }
```

2.指出并消除下面代码里的异味

```
    class SurveyData {
        String path; //将数据存到这个路径的文件
        boolean hidden; //这个文件是否要隐藏
        //根据 t 的类型，设置存放数据的文件的路径和隐藏属性
        void setSavePath(int t) {
            if (t==0) { //raw data.
                path = "c:/application/data/raw.dat";
                hidden = true;
            } else if (t==1) {    //清空数据
                path = "c:/application/data/cleanedUp.dat";
                hidden = true;
            } else if (t==2) {    //处理数据
                path = "c:/application/data/processed.dat";
                hidden = true;
            } else if (t==3) {    //数据可以公布
                path = "c:/application/data/publication.dat";
```

```
                hidden = false;
            }
        }
    }
```

3.指出并消除下面代码里的异味

```
    class CustomersInDB {
        Connection conn;
        Customer getCustomer(String IDNumber) {
            PreparedStatement st = conn.prepareStatement(
                "select * from customer where ID=?");
            try {
                st.setString(1,
                        IDNumber.replace('-', ' ').
                            replace('(', ' ').
                            replace(')', ' ').
                            replace('/', ' '));
                ResultSet rs = st.executeQuery();
                ...
            } finally {
                st.close();
            }
        }
        void addCustomer(Customer customer) {
            PreparedStatement st = conn.prepareStatement(
                "insert into customer values(?,?,?,?)");
            try {
                st.setString(1,
                        customer.getIDNumber().replace('-', ' ').
                                replace('(', ' ').
                                replace(')', ' ').
                                replace('/', ' '));
                st.setString(2, customer.getName());
                ...
                st.executeUpdate();
                ...
            } finally {
                st.close();
            }
        }
    }
```

4.下面的代码包括一些重复代码：方法 printOverdueRentals 和 countOverdueRentals 里面的循环。如果你需要不惜代价的移掉这些重复，你要怎么做？

```
class BookRentals {
    Vector rentals;
    int countRentals() {
        return rentals.size();
    }
    BookRental getRentalAt(int i) {
        return (BookRental)rentals.elementAt(i);
    }
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                System.out.println(rental.toString());
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                count++;
        return count;
    }
}
```

5.指出并消除下面代码里的异味:

```
class Currency {
    final public int USD=0;
    final public int RMB=1;
    final public int ESCUDO=2;  //葡萄牙币的汇率
    private int currencyCode;
    public Currency(int currencyCode) {
        this.currencyCode=currencyCode;
    }
    public String format(int amount) {
        switch (currencyCode) {
            case USD:
                //返回$1,200 之类的
            case RMB:
```

```
                    //返回 RMB1,200 之类的
                case ESCUDO:
                    //返回$1.200 之类的
            }
        }
    }
```

6.指出并消除下面代码里的异味:

```
    class Payment {
        final static String FOC = "FOC";   //免费的
        final static String TT = "TT";   //电汇付账
        final static String CHEQUE = "Cheque";   //支票付账
        final static String CREDIT_CARD = "CreditCard"; //信用卡付账.
        final static String CASH = "Cash";   //现金付账
        //type of payment. Must be one of the above constant.
        String paymentType;
        Date paymentDate; //如果是免费的话，付款日期就不用
        int actualPayment; //如果是免费的话，实际付款金额就不用
        int discount;   //如果是免费的话，折扣就不用
        String bankName; //如果是用电汇，支票或信用卡的话
        String chequeNumber; //如果是用支票的话
        //如果是用信用卡的话.
        String creditCardType;
        String creditCardHolderName;
        String creditCardNumber;
        Date creditCardExpiryDate;
        int getNominalPayment() {
            return actualPayment+discount;
        }
        String getBankName() {
            if (paymentType.equals(TT) ||
                paymentType.equals(CHEQUE) ||
                paymentType.equals(CREDIT_CARD)) {
                return bankName;
            }
            else {
                throw new Exception("bank name is undefined for this payment type");
            }
        }
    }
```

7.在上面的程序中，有一个窗口让用户编辑表示付款情况的对象。用户可以在一个下拉框中选择付账类型。然后

该付账类型相关的输入组件都会显示出来。这个功能是用 Java 下的 CardLayout 排版。下面就是这个窗口的代码。请修改相应的代码。

```
class EditPaymentDialog extends JDialog {
    Payment newPayment;  //返回一个新的表示付款情况的对象
    JPanel sharedPaymentDetails;
    JPanel uniquePaymentDetails;
    JTextField paymentDate;
    JComboBox paymentType;
    JTextField discountForFOC;
    JTextField bankNameForTT;
    JTextField actualAmountForTT;
    JTextField discountForTT;
    JTextField bankNameForCheque;
    JTextField chequeNumberForCheque;
    JTextField actualAmountForCheque;
    JTextField discountForCheque;
    ...
    EditPaymentDialog() {
        //创建所有组件
        Container contentPane = getContentPane();
        String comboBoxItems[] = { //可用的付款类型
            Payment.FOC,
            Payment.TT,
            Payment.CHEQUE,
            Payment.CREDIT_CARD,
            Payment.CASH
        };
        //创建所有付款类型共用的一些组件
        sharedPaymentDetails = new JPanel();
        paymentDate = new JTextField();
        paymentType = new JComboBox(comboBoxItems);
        sharedPaymentDetails.add(paymentDate);
        sharedPaymentDetails.add(paymentType);
        contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
        //创建每种付款类型各自需要用到的组件
        uniquePaymentDetails = new JPanel();
        uniquePaymentDetails.setLayout(new CardLayout());
        //为免费创建一个面板
        JPanel panelForFOC = new JPanel();
        discountForFOC = new JTextField();
        panelForFOC.add(discountForFOC);
        uniquePaymentDetails.add(panelForFOC, Payment.FOC);
```

```
        //为电汇创建一个面板
        JPanel panelForTT = new JPanel();
        bankNameForTT = new JTextField();
        actualAmountForTT = new JTextField();
        discountForTT = new JTextField();
        panelForTT.add(bankNameForTT);
        panelForTT.add(actualAmountForTT);
        panelForTT.add(discountForTT);
        uniquePaymentDetails.add(panelForTT, Payment.TT);
        //为支票付账创建一个面板
        JPanel panelForCheque = new JPanel();
        bankNameForCheque = new JTextField();
        chequeNumberForCheque = new JTextField();
        actualAmountForCheque = new JTextField();
        discountForCheque = new JTextField();
        panelForCheque.add(bankNameForCheque);
        panelForCheque.add(chequeNumberForCheque);
        panelForCheque.add(actualAmountForCheque);
        panelForCheque.add(discountForCheque);
        uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
        //为信用卡付账创建一个面板
        ...
        //为现金付账创建一个面板
        ...
        contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
    }
Payment editPayment(Payment payment) {
        displayPayment(payment);
        setVisible(true);
        return newPayment;
    }
    void displayPayment(Payment payment) {
        paymentDate.setText(payment.getDateAsString());
        paymentType.setSelectedItem(payment.getType());
        if (payment.getType().equals(Payment.FOC)) {
            discountForFOC.setText(Integer.toString(payment.getDiscount()));
        }
        else if (payment.getType().equals(Payment.TT)) {
            bankNameForTT.setText(payment.getBankName());
            actualAmountForTT.setText(
                Integer.toString(payment.getActualAmount()));
            discountForTT.setText(Integer.toString(payment.getDiscount()));
        }
```

```
    else if (payment.getType().equals(Payment.CHEQUE)) {
        bankNameForCheque.setText(payment.getBankName());
        chequeNumberForCheque.setText(payment.getChequeNumber());
        actualAmountForCheque.setText(
            Integer.toString(payment.getActualAmount()));
        discountForCheque.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.CREDIT_CARD)) {
        //...
    }
    else if (payment.getType().equals(Payment.CASH)) {
        //...
    }
}
//当用户点击“OK“时
void onOK() {
    newPayment = makePayment();
    dispose();
}
//从所有的组件中组合出一个付款情况
Payment makePayment() {
    String paymentTypeString = (String) paymentType.getSelectedItem();
    Payment payment = new Payment(paymentTypeString);
    payment.setDateAsText(paymentDate.getText());
    if (paymentTypeString.equals(Payment.FOC)) {
        payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
    }
    else if (paymentTypeString.equals(Payment.TT)) {
        payment.setBankName(bankNameForTT.getText());
        payment.setActualAmount(
            Integer.parseInt(actualAmountForTT.getText()));
        payment.setDiscount(
            Integer.parseInt(discountForTT.getText()));
    }
    else if (paymentTypeString.equals(Payment.CHEQUE)) {
        payment.setBankName(bankNameForCheque.getText());
        payment.setChequeNumber(chequeNumberForCheque.getText());
        payment.setActualAmount(
            Integer.parseInt(actualAmountForCheque.getText()));
        payment.setDiscount(
            Integer.parseInt(discountForCheque.getText()));
    }
    else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
```

```
            //...
        }
        else if (paymentTypeString.equals(Payment.CASH)) {
            //...
        }
        return payment;
    }
}
```

8.这是个控制电饭煲的嵌入式系统。该系统每秒钟都要做这些事：检查一下这个蒸机是否过热（像短路的话，就会过热）。如果过热的话，该系统就是自动断开电源，并用内置的发声器发出警报；会检查里面的湿度是否低于一定的值（像饭煮好了，湿度就低于该值）。如果湿度低于该值的话，自动转换到一个内置的 50 度的加热器，用来为里面的米饭保温。

将来，你还期望可以让这个嵌入式系统每秒钟多做一些其他的事情。

指出并修正下面代码的问题

```
class Scheduler extends Thread {
    Alarm alarm;
    HeatSensor heatSensor;
    PowerSupply powerSupply;
    MoistureSensor moistureSensor;
    Heater heater;
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            //检查是否过热
            if (heatSensor.isOverHeated()) {
                powerSupply.turnOff();
                alarm.turnOn();
            }
            //检查饭熟了没
            if (moistureSensor.getMoisture()<60) {
                heater.setTemperature(50);
            }
        }
    }
}
```

9.这个系统用来管理培训课程。课程的时间排列有三种方式：按每周排，按一个时间段排，和按一个日期列表排。按每周排就比如说“10 月 22 日以后五周内的每个星期二”。按一个时间段排就比较如“10 月 22 日到 11 月 3 日里每天上”。按一个日期列表排就像“10 月 22 日,10 月 25 日,11 月 3 日,11 月 10 日”。这回的练习，我们先忽略时间，就假定每天上课都是从晚上 7 点到晚上 10 点。以后要增加新的时间表排列方式。

指出并消除下面代码里的异味:

```
class Course {
    static final int WEEKLY=0;
    static final int RANGE=1;
    static final int LIST=2;
    String courseTitle;
    int scheduleType;   // 按每周排，按一个时间段排，或者按一个日期列表排
    int noWeeks;        // 按每周排
    Date fromDate;      // 按每周排或者按一个时间段排需要用到
    Date toDate;        // 按一个时间段排要用到
    Date dateList[];    // 按一个日期列表排要用到

    int getDurationInDays() {
        switch (scheduleType) {
            case WEEKLY:
                return noWeeks;
            case RANGE:
                int msInOneDay = 24*60*60*1000;
                return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
            case LIST:
                return dateList.length;
            default:
                return 0;   // 未知的排列方式
        }
    }
    void printSchedule() {
        switch (scheduleType) {
            case WEEKLY:
                //...
            case RANGE:
                //...
            case LIST:
                //...
        }
    }
}
```

10.这个系统用来管理培训课程。一个培训课程有名称，费用和时间段列表。不过，有时候一个课程是有几个模块组成的，每个模块又是一个课程。比如，现在有个混合的课程叫“快速成为网页开发员”，它有三个模块组成：“HTML”的课程，“FrontPage”的课程，还有“Flash”的课程。一个模块也有可能是有其他模块组成的。如果一个课程是由几个模块组成的话，那么该课程的费用跟时间表是由它的模块合在一起算的，因为这个课程自己没有费用跟时间表。

指出并消除下面代码里的异味:

```
class Session {
    Date date;
    int startHour;
    int endHour;
    int getDuration() {
        return endHour-startHour;
    }
}
class Course {
    String courseTitle;
    Session sessions[];
    double fee;
    Course modules[];
    Course(String courseTitle, double fee, Session sessions[]) {
        //...
    }
    Course(String courseTitle, Course modules[]) {
        //...
    }
    String getTitle() {
        return courseTitle;
    }
    double getDuration() {
        int duration=0;
        if (modules==null)
            for (int i=0; i<sessions.length; i++)
                duration += sessions[i].getDuration();
        else
            for (int i=0; i<modules.length; i++)
                duration += modules[i].getDuration();
        return duration;
    }
    double getFee() {
        if (modules==null)
            return fee;
        else {
            double totalFee = 0;
            for (int i=0; i<modules.length; i++)
                totalFee += modules[i].getFee();
            return totalFee;
        }
```

```
        }
        void setFee(int fee) throws Exception {
            if (modules==null)
                this.fee = fee;
            else
                throw new Exception("Please set the fee of each module one by one");
        }
    }
```

11.指出并消除下面代码里的异味:

```
    class BookRental {
        String bookTitle;
        String author;
        Date rentDate;
        Date dueDate;
        double rentalFee;
        boolean isOverdue() {
            Date now=new Date();
            return dueDate.before(now);
        }
        double getTotalFee() {
            return isOverdue() ? 1.2*rentalFee : rentalFee;
        }
    }
    class MovieRental {
        String movieTitle;
        int classification;
        Date rentDate;
        Date dueDate;
        double rentalFee;
        boolean isOverdue() {
            Date now=new Date();
            return dueDate.before(now);
        }
        double getTotalFee() {
            return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
        }
    }
```

12.指出并消除下面代码里的异味:

```
    class Customer {
        String homeAddress;
        String workAddress;
```

```
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    //"H": 送到客户家庭住址
    //"W": 送到客户办公地址
    //"O": 送到客户指定的其他地址
    String addressType;
    String otherAddress; //如果是指定的其他地址的话
    HashMap orderItems;

    public String getDeliveryAddress() {
        if (addressType.equals("H")) {
            return customerPlacingOrder.getHomeAddress();
        } else if (addressType.equals("W")) {
            return customerPlacingOrder.getWorkAddress();
        } else if (addressType.equals("O")) {
            return otherAddress;
        } else {
            return null;
        }
    }
}
```

13.找出一些含有类别代码的代码，然后移掉类别代码。

提示

1.更改 q0-q3 的变量名，LoadData 的方法名和其他。这个 SQL 语句中包含重复代码。域的命名重复了。

2.存储文件所在的文件夹路径这串字符串重复了。更改隐藏属性的代码重复。考虑一下移掉 if-then-else-if。考虑一下更改一些命名。

3.替换特定的字符和空格的代码重复了。再看一下替换字符的实现代码，连续 replace 的这些代码也重复了。

4.你要先让两个循环里面的代码一样（在比较抽象的水平上一样）：

```
class BookRentals {
    ...
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
```

```
                if (rental.isOverdue())
                    处理租贷情况;
            }
        }
        int countOverdueRentals() {
            int i, count;
            count=0;
            for (i=0; i<countRentals(); i++)
                if (getRentalAt(i).isOverdue())
                    处理租贷情况;
            return count;
        }
}
```

将这些共同的代码抽取到一个独立的方法。然后让原来的两个方法调用这个方法。

```
class BookRentals {
    ...

    void yyy() {
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                处理租贷情况;
        }
    }
    void printOverdueRentals() {
        yyy();
    }
    int countOverdueRentals() {
        int count;
        yyy();
        return count;
    }
}
```

因为“处理租贷情况”的具体代码实现有两种，一种是用来 printOverdueRentals 的，还有一种是用来 countOverdueRentals 的，你应该先创建一个接口，让它可以允许两种实现。

```
interface XXX {
    void process(Rental rental);
}
class BookRentals {
```

```
    ...
    void yyy(XXX obj) {
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                obj.process(rental);
        }
    }
    ...
}
```

然后，printOverdueRentals 和 countOverdueRentals 要提供它们各自的代码实现：

```
class XXX1 implements XXX {
    ...
}
class XXX2 implements XXX {
    ...
}
class BookRentals {
    ...
    void yyy(XXX obj) {
        ...
    }
    void printOverdueRentals() {
        XXX1 obj=new XXX1();
        yyy(obj);
    }
    int countOverdueRentals() {
        int count;
        XXX2 obj=new XXX2();
        yyy(obj);
        return count;
    }
}
```

好，重复的循环代码移移掉了。现在根据 XXX, XXX1, XXX2 和 yyy 所做的事，为什么它们想个好名字吧。如果可以的话，我们可以将 XXX1 和 XXX2 做成匿名类（Java 才行，Delphi 和 C++不支持）。

5.创建一些子类，比如：USDCurrency, RMBCurrency 和 ESCUDOCurrency. 每个子类提供一个格式化的方法。

6.创建一些子类，比如 FOCPayment, TTPayment 等等. 每个子类都应该有 getNominalPayment 这个方法。部分的这些子类提供 getBankName 这个方法。

7. 有一些注释可以转化到代码里面的，另外：

你应该为每一个 payment 创建各自的实现 UI 的类，比如 FOCPaymentUI,TTPaymentUI 等等。它们都应该继续/实现这个父类 PaymentUI。父类 PaymentUI 应该有两个类似于这样的方法：tryToDisplayPayment(Payment payment)和 makePayment()，然后让所有的子类各自实现这两个方法。 为了实现 tryToDisplayPayment(Payment payment)这个方法，每个子 UI 类都要自己增加一些输入组件（Text field 之类的输入组件）作为属性。

tryToDisplayPayment 就是用自身的这些属性组件来显示 Payment 对象的信息。makePayment 则是根据这些属性组件的值，创建一个 Payment 对象返回。

比如，TTPaymentUI 的 tryToDisplayPayment 方法检查 Payment 对象是不是属于 TTPayment 类。如果是的话，它就显示这个 TTPayment 的信息，然后返回 true。否则返回 false。TTPaymentUI 还要自己初始化这些属性组件（包括托起这些组件的面板）。当然，也有一些输入组件是共有的，比如像表示付款日期的这个组件，就是每个 UI 类都有。所以这个组件就不应该由 TTPaymentUI 自己来创建，而应该将这个组件放在 EditPaymentDialog 这个类里面，然后在 EditPaymentDialog 这个类的代码中，将这个组件传递给 TTPaymentUI 的构造函数。

每个 UI 都应该实现这个方法 toString，这样的话，我们就可以将所有的 UI 对象都存放在一个表示 paymentType 的下拉框里面（用过 java 的 Combobox 的话就明白这句的意思，在 ComboBox 里面，每个选项可以不仅仅是 String 类型，也可以是其他类型的对象，Java 会调用这个对象的 toString 方法，用来显示选项的文本）

用了这些类的话，displayPayment 方法和 EditPaymentDialog 里面的 makePayment 这个方法里面的一大串 if-then-else-if，就可以移除了。

看了这么长的一段描述，可能你会有些混淆，没事，看一下后面的解决办法后，再回来看这里就会明白了。

8. 如果按照需求所说的话，这个 run 方法肯定会越来越长。要避免这种情况的话，你首先要让检查是否过热的代码跟检查饭熟了没的代码一样（当然，是在一定的抽象程度一样）。然后以后每个增加到 run 里面的方法都要一样。这些，看一下 Java 里面的 JButton 跟 ActionListener 的机制就知道了。

解决方法示例

1.指出并消除下面代码里的异味：

```
class FoodSalesReport {
    int q0; //卖了多少米？
    int q1; //卖了多少面？
    int q2; //卖了多少饮料？
    int q3; //卖了多少甜点？？
    void LoadData(Connection conn) {
        PreparedStatement st = conn.prepareStatement("select "+
            "sum(case when foodType=0 then qty else 0 end) as totalQty0,"+
            "sum(case when foodType=1 then qty else 0 end) as totalQty1,"+
            "sum(case when foodType=2 then qty else 0 end) as totalQty2,"+
            "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
            "from foodSalesTable group by foodType");
        try {
```

```
                ResultSet rs = st.executeQuery();
                try {
                    rs.next();
                    q0 = rs.getInt("totalQty0");
                    q1 = rs.getInt("totalQty1");
                    q2 = rs.getInt("totalQty2");
                    q3 = rs.getInt("totalQty3");
                } finally {
                    rs.close();
                }
            } finally {
                st.close();
            }
        }
}
```

q0-q3 这四个变量的名字可以改得好一点，就省得注释了。SQL 语句里面的四个 sum(...) 也重复了。

```
class FoodSalesReport {
    int qtyRiceSold;
    int qtyNoodleSold;
    int qtyDrinkSold;
    int qtyDessertSold;
    void LoadData(Connection conn) {
        String sqlExprList = "";
        for (int i = 0; i <= 3; i++) {
            String separator = (i==0) ? "" : ",";
            sqlExprList = sqlExprList+
                        separator+
                        getSQLForSoldQtyForFoodType(i);
        }
        PreparedStatement st = conn.prepareStatement("select "+
            sqlExprList+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                qtyRiceSold = rs.getInt("totalQty0");
                qtyNoodleSold = rs.getInt("totalQty1");
                qtyDrinkSold = rs.getInt("totalQty2");
                qtyDessertSold = rs.getInt("totalQty3");
            } finally {
```

```
                rs.close();
            }
        } finally {
            st.close();
        }
    }
    String getSQLForSoldQtyForFoodType(int foodType) {
        return "sum(case when foodType="+foodType+
                " then qty else 0 end) as totalQty"+foodType;
    }
}
```

如果你还是担心类别代码还有这些类别代码之间太相近的话，这样做吧：

```
class FoodType {
    int typeCode;
    FoodType(int typeCode) {
        ...
    }
    static FoodType foodTypeRice = new FoodType(0);
    static FoodType foodTypeNoodle = new FoodType(1);
    static FoodType foodTypeDrink = new FoodType(2);
    static FoodType foodTypeDessert = new FoodType(3);
    static FoodType knownFoodTypes[] =
        { foodTypeRice, foodTypeNoodle, foodTypeDrink, foodTypeDessert };
}
class FoodSalesReport {
    HashMap foodTypeToQtySold;
    void LoadData(Connection conn) {
        FoodType knownFoodTypes[] = FoodType.knownFoodTypes;
        String sqlExprList = "";
        for (int i = 0; i < knownFoodTypes.length; i++) {
            String separator = (i==0) ? "" : ",";
            sqlExprList = sqlExprList+
                        separator+
                        getSQLForSoldQtyForFoodType(knownFoodTypes[i]);
        }
        PreparedStatement st = conn.prepareStatement("select "+
            sqlExprList+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
```

```
                    for (int i = 0; i < knownFoodTypes.length; i++) {
                        FoodType foodType = knownFoodTypes[i];
                        int qty = rs.getInt(getQtyFieldNameForFoodType(foodType));
                        foodTypeToQtySold.put(foodType, new Integer(qty));
                    }
                } finally {
                    rs.close();
                }
            } finally {
                st.close();
            }
        }
        static String getQtyFieldNameForFoodType(FoodType foodType) {
            return "totalQty"+foodType.getCode();
        }
        String getSQLForSoldQtyForFoodType(FoodType foodType) {
            return "sum(case when foodType="+foodType.getCode()+
                    " then qty else 0 end) as "+
                    getQtyFieldNameForFoodType(foodType);
        }
    }
```

2.指出并消除下面代码里的异味

```
    class SurveyData {
        String path; //将数据存到这个路径的文件
        boolean hidden; //这个文件是否要隐藏
        //根据 t 的类型，设置存放数据的文件的路径和隐藏属性
        void setSavePath(int t) {
            if (t==0) { //raw data.
                path = "c:/application/data/raw.dat";
                hidden = true;
            } else if (t==1) {    //清空数据
                path = "c:/application/data/cleanedUp.dat";
                hidden = true;
            } else if (t==2) {    //处理数据
                path = "c:/application/data/processed.dat";
                hidden = true;
            } else if (t==3) {    //数据可以公布
                path = "c:/application/data/publication.dat";
                hidden = false;
            }
        }
    }
```

这个文件夹的路径"c:/application/data"一直重复出现。".dat"这个后缀名也重复出现。设置 path 跟 hidden 的代码都是重复的。注释也应该转为代码。类别代码（就是那个变量 t）也避免被剔除。每种类别都应该用一个对象代替：

```
class SurveyDataType {
    String baseFileName;
    boolean hideDataFile;
    SurveyDataType(String baseFileName, boolean hideDataFile) {
        this.baseFileName = baseFileName;
        this.hideDataFile = hideDataFile;
    }
    String getSavePath() {
        return "c:/application/data/"+baseFileName+".dat";
    }

    static SurveyDataType rawDataType =
        new SurveyDataType("raw", true);
    static SurveyDataType cleanedUpDataType =
        new SurveyDataType("cleanedUp", true);
    static SurveyDataType processedDataType =
        new SurveyDataType("processed", true);
    static SurveyDataType publicationDataType =
        new SurveyDataType("publication", false);
}
```

3.指出并消除下面代码里的异味 ：

```
class CustomersInDB {
    Connection conn;
    Customer getCustomer(String IDNumber) {
        PreparedStatement st = conn.prepareStatement(
            "select * from customer where ID=?");
        try {
            st.setString(1,
                    IDNumber.replace('-', ' ').
                        replace('(', ' ').
                        replace(')', ' ').
                        replace('/', ' '));
            ResultSet rs = st.executeQuery();
            ...
        } finally {
            st.close();
```

```
            }
        }
        void addCustomer(Customer customer) {
            PreparedStatement st = conn.prepareStatement(
                "insert into customer values(?,?,?,?)");
            try {
                st.setString(1,
                        customer.getIDNumber().replace('-', ' ').
                                replace('(', ' ').
                                replace(')', ' ').
                                replace('/', ' '));
                st.setString(2, customer.getName());
                ...
                st.executeUpdate();
                ...
            } finally {
                st.close();
            }
        }
}
```

处理 ID 的代码重复了，在处理 ID 里面的代码中，屡次的调用 replace 也是重复代码。

```
class CustomersInDB {
    Connection conn;
    String replaceSymbolsInID(String idNumber) {
        String symbolsToReplace = "-()/";
        for (int i = 0; i < symbolsToReplace.length(); i++) {
            idNumber = idNumber.replace(symbolsToReplace.charAt(i), ' ');
        }
        return idNumber;
    }
    Customer getCustomer(String IDNumber) {
        PreparedStatement st = conn.prepareStatement(
            "select * from customer where ID=?");
        try {
            st.setString(1, replaceSymbolsInID(IDNumber));
            ResultSet rs = st.executeQuery();
            ...
        } finally {
            st.close();
        }
    }
```

```
        void addCustomer(Customer customer) {
            PreparedStatement st = conn.prepareStatement(
                    "insert into customer values(?,?,?,?)");
            try {
                    st.setString(1, replaceSymbolsInID(customer.getIDNumber()));
                    st.setString(2, customer.getName());
                    ...
                    st.executeUpdate();
                    ...
            } finally {
                    st.close();
            }
        }
    }
```

4.下面的代码包括一些重复代码：方法 printOverdueRentals 和 countOverdueRentals 里面的循环。如果你需要不惜代价的移动这些重复，你要怎么做？

```
    class BookRentals {
        Vector rentals;
        int countRentals() {
            return rentals.size();
        }
        BookRental getRentalAt(int i) {
            return (BookRental)rentals.elementAt(i);
        }
        void printOverdueRentals() {
            int i;
            for (i=0; i<countRentals(); i++) {
                    BookRental rental = getRentalAt(i);
                    if (rental.isOverdue())
                        System.out.println(rental.toString());
            }
        }
        int countOverdueRentals() {
            int i, count;
            count=0;
            for (i=0; i<countRentals(); i++)
                    if (getRentalAt(i).isOverdue())
                        count++;
            return count;
        }
    }
```

首先，先让这两个循环里面的代码一样：

```
class BookRentals {
    ...
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                处理租贷情况;
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                处理租贷情况;
        return count;
    }
}
```

现在这两个循环的代码是一样的了。好，现在我们将这两个一样的循环抽取出来，放在一个独立的方法里面，叫做 yyy，然后 printOverdueRentals()和 countOverdueRentals()都调用了这个新方法：

```
class BookRentals {
    ...
    void yyy() {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
    void printOverdueRentals() {
        yyy();
    }
    int countOverdueRentals() {
        int count;
        yyy();
        return count;
    }
```

```
}
```

要为这个方法取个好名字，我们先想想，这个方法都做了些什么事。从代码中我们可以看来了，这个代码是遍历所有的 rental，然后处理每个 overdue 的 rental。所以，我们将这个方法取作“processOverdueRentals”：

```
class BookRentals {
    ...
    void processOverdueRentals() {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
    void printOverdueRentals() {
        processOverdueRentals();
    }
    int countOverdueRentals() {
        int count;
        processOverdueRentals();
        return count;
    }
}
```

因为“处理租贷情况”这样抽象的行为实现出来以后，会有两个具体的行为。我们就创建一个接口，让它可以派生不同的实现：

```
interface XXX {
    void process(BookRental rental);
}
class BookRentals {
    ...
    void processOverdueRentals(XXX obj) {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                obj.process(rental);
        }
    }
    ...
}
```

现在想想要为这个接口取作什么名字。好，看看这接口要做什么事？很明显的，这个接口只有一个方法，这

个方法处理租赁情况。所以“RentalProcessor”会是个好名字：

```
interface RentalProcessor {
    void process(BookRental rental);
}
class BookRentals {
    ...
    void processOverdueRentals(RentalProcessor processor) {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                processor.process(rental);
        }
    }
    ...
}
```

printOverdueRentals 和 countOverdueRentals 现在需要提供它们各自的实现代码：

```
class RentalPrinter implements RentalProcessor {
    void process(BookRental rental) {
        System.out.println(rental.toString());
    }
}
class RentalCounter implements RentalProcessor {
    int count = 0;
    void process(BookRental rental) {
        count++;
    }
}
class BookRentals {
    ...
    void processOverdueRentals(RentalProcessor processor) {
        ...
    }
    void printOverdueRentals() {
        RentalPrinter rentalPrinter=new RentalPrinter();
        processOverdueRentals(rentalPrinter);
    }
    int countOverdueRentals() {
        RentalCounter rentalCounter=new RentalCounter();
        processOverdueRentals(rentalCounter);
        return rentalCounter.count;
```

```
        }
    }
```

如果可能的话，用内类或者匿名类：

```
class BookRentals {
    ...
    void printOverdueRentals() {
        processOverdueRentals(new RentalProcessor() {
            void process(BookRental rental) {
                System.out.println(rental.toString());
            }
        });
    }
    int countOverdueRentals() {
        class RentalCounter implements RentalProcessor {
            int count = 0;
            void process(BookRental rental) {
                count++;
            }
        }
        RentalCounter rentalCounter = new RentalCounter();
        processOverdueRentals(rentalCounter);
        return rentalCounter.count;
    }
}
```

5.指出并消除下面代码里的异味:

```
class Currency {
    final public int USD=0;
    final public int RMB=1;
    final public int ESCUDO=2;   //葡萄牙币的汇率
    private int currencyCode;
    public Currency(int currencyCode) {
        this.currencyCode=currencyCode;
    }
    public String format(int amount) {
        switch (currencyCode) {
            case USD:
                //返回$1,200 之类的
            case RMB:
                //返回 RMB1,200 之类的
```

```
            case ESCUDO:
                //返回$1.200 之类的
        }
    }
}
```

用到类别代码真是坏事。将每个类别代码转为一个类吧。

```
interface Currency {
    public String format(int amount);
}
class USDCurrency implements Currency {
    public String format(int amount) {
        //返回$1,200 之类的
    }
}
class RMBCurrency implements Currency {
    public String format(int amount) {
        //返回 RMB1,200 之类的
    }
}
class ESCUDOCurrency implements Currency {
    public String format(int amount) {
        //返回$1.200 之类的
    }
}
```

6.指出并消除下面代码里的异味:

```
class Payment {
    final static String FOC = "FOC";   //免费的
    final static String TT = "TT";   //电汇付账
    final static String CHEQUE = "Cheque";   //支票付账
    final static String CREDIT_CARD = "CreditCard"; //信用卡付账.
    final static String CASH = "Cash";   //现金付账
    //type of payment. Must be one of the above constant.
    String paymentType;
    Date paymentDate; //如果是免费的话，付款日期就不用
    int actualPayment; //如果是免费的话，实际付款金额就不用
    int discount;   //如果是免费的话，折扣就不用
    String bankName; //如果是用电汇，支票或信用卡的话
    String chequeNumber; //如果是用支票的话
    //如果是用信用卡的话.
    String creditCardType;
    String creditCardHolderName;
```

```
        String creditCardNumber;
        Date creditCardExpiryDate;
        int getNominalPayment() {
            return actualPayment+discount;
        }
        String getBankName() {
            if (paymentType.equals(TT) ||
                paymentType.equals(CHEQUE) ||
                paymentType.equals(CREDIT_CARD)) {
                return bankName;
            }
            else {
                throw new Exception("bank name is undefined for this payment type");
            }
        }
    }
```

用到类别代码了。而且大多数的属性都是时用时不用的。我们现在要把每种类别代码转为一个类。actualAmount 和 discount 这两个属性在不同的付款类型都有出现，所以我们将这两个属性放在父类里面。

有些注释可以顺带转为代码。

```
abstract class Payment {
    Date paymentDate;
    abstract int getNominalPayment();
}
class FOCPayment extends Payment {
    int amountWaived;
    int getNominalPayment() {
        return amountWaived;
    }
}
class RealPayment extends Payment {
    int actualPayment;
    int discount;
    int getNominalPayment() {
        return actualPayment+discount;
    }
}
class TTPayment extends RealPayment {
    String bankName;
    String getBankName() {
        return bankName;
    }
```

```
}
class ChequePayment extends RealPayment {
    String bankName;
    String chequeNumber;
    String getBankName() {
        return bankName;
    }
}
class CreditCardPayment extends RealPayment {
    String bankName;
    String cardType;
    String cardHolderName;
    String cardNumber;
    Date expiryDate;
    String getBankName() {
        return bankName;
    }
}
class CashPayment extends RealPayment {
}
```

注意到 bankName 这个属性还不同的 payment 类型里面还是会重复。可能我们要将这个属性抽取到父类的，但问题就是，我们很难为这样的属性取一个名字，因为不同的情况它表达大不同的东西（BankPayment? PaymentThroughBank?）。因为这只是一个变量，将这个变量抽取出来的动机不怎么强烈。所以看你自己抽不抽了。不过你要注意的是，如果抽取出来的话，你可能要加很多注释了。

7.在上面的程序中，有一个窗口让用户编辑表示付款情况的对象。用户可以在一个下拉框中选择付账类型。然后该付账类型相关的组件都会显示出来。这个功能是用 Java 下的 CardLayout 排版。下面就是这个窗口的代码。请修改相应的代码。

```
class EditPaymentDialog extends JDialog {
    Payment newPayment;   //返回一个新的表示付款情况的对象
    JPanel sharedPaymentDetails;
    JPanel uniquePaymentDetails;
    JTextField paymentDate;
    JComboBox paymentType;
    JTextField discountForFOC;
    JTextField bankNameForTT;
    JTextField actualAmountForTT;
    JTextField discountForTT;
    JTextField bankNameForCheque;
    JTextField chequeNumberForCheque;
```

```
JTextField actualAmountForCheque;
JTextField discountForCheque;
...
EditPaymentDialog() {
    //创建所有组件
    Container contentPane = getContentPane();
    String comboBoxItems[] = { //可用的付款类型
        Payment.FOC,
        Payment.TT,
        Payment.CHEQUE,
        Payment.CREDIT_CARD,
        Payment.CASH
    };
    //创建所有付款类型共用的一些组件
    sharedPaymentDetails = new JPanel();
    paymentDate = new JTextField();
    paymentType = new JComboBox(comboBoxItems);
    sharedPaymentDetails.add(paymentDate);
    sharedPaymentDetails.add(paymentType);
    contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
    //创建每种付款类型各自需要用到的组件
    uniquePaymentDetails = new JPanel();
    uniquePaymentDetails.setLayout(new CardLayout());
    //为免费创建一个面板
    JPanel panelForFOC = new JPanel();
    discountForFOC = new JTextField();
    panelForFOC.add(discountForFOC);
    uniquePaymentDetails.add(panelForFOC, Payment.FOC);
    //为电汇创建一个面板
    JPanel panelForTT = new JPanel();
    bankNameForTT = new JTextField();
    actualAmountForTT = new JTextField();
    discountForTT = new JTextField();
    panelForTT.add(bankNameForTT);
    panelForTT.add(actualAmountForTT);
    panelForTT.add(discountForTT);
    uniquePaymentDetails.add(panelForTT, Payment.TT);
    //为支票付账创建一个面板
    JPanel panelForCheque = new JPanel();
    bankNameForCheque = new JTextField();
    chequeNumberForCheque = new JTextField();
    actualAmountForCheque = new JTextField();
    discountForCheque = new JTextField();
```

```
        panelForCheque.add(bankNameForCheque);
        panelForCheque.add(chequeNumberForCheque);
        panelForCheque.add(actualAmountForCheque);
        panelForCheque.add(discountForCheque);
        uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
        //setup a panel for credit card payment.
        ...
        //为现金付账创建一个面板
        ...
        contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
    }
Payment editPayment(Payment payment) {
        displayPayment(payment);
        setVisible(true);
        return newPayment;
    }
    void displayPayment(Payment payment) {
        paymentDate.setText(payment.getDateAsString());
        paymentType.setSelectedItem(payment.getType());
        if (payment.getType().equals(Payment.FOC)) {
            discountForFOC.setText(Integer.toString(payment.getDiscount()));
        }
        else if (payment.getType().equals(Payment.TT)) {
            bankNameForTT.setText(payment.getBankName());
            actualAmountForTT.setText(
                Integer.toString(payment.getActualAmount()));
            discountForTT.setText(Integer.toString(payment.getDiscount()));
        }
        else if (payment.getType().equals(Payment.CHEQUE)) {
            bankNameForCheque.setText(payment.getBankName());
            chequeNumberForCheque.setText(payment.getChequeNumber());
            actualAmountForCheque.setText(
                Integer.toString(payment.getActualAmount()));
            discountForCheque.setText(Integer.toString(payment.getDiscount()));
        }
        else if (payment.getType().equals(Payment.CREDIT_CARD)) {
            //...
        }
        else if (payment.getType().equals(Payment.CASH)) {
            //...
        }
    }
    //当用户点击“OK“时
```

```
    void onOK() {
        newPayment = makePayment();
        dispose();
    }
    //从所有的组件中组合出一个付款情况
    Payment makePayment() {
        String paymentTypeString = (String) paymentType.getSelectedItem();
        Payment payment = new Payment(paymentTypeString);
        payment.setDateAsText(paymentDate.getText());
        if (paymentTypeString.equals(Payment.FOC)) {
            payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
        }
        else if (paymentTypeString.equals(Payment.TT)) {
            payment.setBankName(bankNameForTT.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForTT.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForTT.getText()));
        }
        else if (paymentTypeString.equals(Payment.CHEQUE)) {
            payment.setBankName(bankNameForCheque.getText());
            payment.setChequeNumber(chequeNumberForCheque.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForCheque.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForCheque.getText()));
        }
        else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
            //...
        }
        else if (paymentTypeString.equals(Payment.CASH)) {
           //...
        }
        return payment;
    }
}
```

有些注释可以转为代码。displayPayment 和 EditPaymentDialog 的 makePayment 里面的这一大串 if-then-else-if 也不是好事。

要去掉这一大串 if-then-else-if，我们就要为每种 payment 都创建一个 UI 类，比如 FOCPaymentUI, TTPaymentUI 等等。它们应该都继承/实现同一个父类如 PaymentUI。PaymentUI 这个类里面有两个方法 tryToDisplayPayment (Payment payment) 和 makePayment(). PaymentUI 的构造函数应该将 paymentDate JTextField 这个作为一个参数，因为每个子类都需要它，而我们没必要在每个子类都有处理 paymentDate JTextField 的重复代

码。

好，现在为了在 EditPaymentDialog 里面可以用下拉框选择使用哪个具体的 PaymentUI，我们就要让所有的 PaymentUI 子类都实现这个方法 toString。这样我们就可以将各个类型的 PaymentUI 组成一个对象数组，扔给下拉框，而下拉框每次选择的，就是一个具体的 PaymentUI，而不仅仅是个字符串 String 而已了。

```
abstract class PaymentUI {
    JTextField paymentDate;
    PaymentUI(JTextField paymentDate) {
        ...
    }
    void displayPaymentDate(Payment payment) {
        paymentDate.setText(
            new SimpleDateFormat().format(payment.getPaymentDate()));
    }
    Date makePaymentDate() {
        //解析 paymentDate 的值，然后返回一个 Date 对象；
    }
    abstract boolean tryToDisplayPayment(Payment payment);
    abstract Payment makePayment();
    abstract JPanel getPanel();
}
abstract class RealPaymentUI extends PaymentUI {
    JTextField actualPayment;
    JTextField discount;
    RealPaymentUI(JTextField paymentDate) {
        super(paymentDate);
        actualPayment = new JTextField();
        discount = new JTextField();
    }
    void displayActualPayment(RealPayment payment) {
        actualPayment.setText(Integer.toString(payment.getActualPayment()));
    }
    void displayDiscount(RealPayment payment) {
        discount.setText(Integer.toString(payment.getDiscount()));
    }
    int makeActualPayment() {
        //解析 actualPayment 的文本，返回一个 int 型
    }
    int makeDiscount() {
        //解析 discount 的文本，返回一个 int 型
    }
}
class TTPaymentUI extends RealPaymentUI {
```

```
    JPanel panel;
    JTextField bankName;
    TTPaymentUI(JTextField paymentDate) {
        super(paymentDate);
        panel = ...;
        bankName = ...;
        //增加这些组件：bankName, actualPayment, discount  到面板里面.
    }
    boolean tryToDisplayPayment(Payment payment) {
        if (payment instanceof TTPayment) {
            TTPayment ttpayment = (TTPayment)payment;
            displayPaymentDate(payment);
            displayActualPayment(ttpayment);
            displayDiscount(ttpayment);
            bankName.setText(ttpayment.getBankName());
            return true;
        }
        return false;
    }
    Payment makePayment() {
        return new TTPayment(makePaymentDate(),
                            makeActualPayment(),
                            makeDiscount(),
                            bankName.getText());
    }
    String toString() {
        return "TT";
    }
    JPanel getPanel() {
        return panel;
    }
}
class FOCPaymentUI extends PaymentUI {
    ...
}
class ChequePaymentUI extends RealPaymentUI {
    ...
}
class EditPaymentDialog extends JDialog {
    Payment newPaymentToReturn;
    JPanel sharedPaymentDetails;
    JTextField paymentDate;
    JComboBox paymentType;
```

```
PaymentUI paymentUIs[];
EditPaymentDialog() {
    setupComponents();
}
void setupComponents() {
    setupComponentsSharedByAllPaymentTypes();
    setupPaymentUIs();
    setupPaymentTypeIndicator();
    setupComponentsUniqueToEachPaymentType();
}
void setupComponentsSharedByAllPaymentTypes() {
    paymentDate = new JTextField();
    sharedPaymentDetails = new JPanel();
    sharedPaymentDetails.add(paymentDate);
    Container contentPane = getContentPane();
    contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
}
void setupPaymentUIs() {
    paymentUIs[0] = new TTPaymentUI(paymentDate);
    paymentUIs[1] = new FOCPaymentUI(paymentDate);
    paymentUIs[2] = new ChequeaymentUI(paymentDate);
    ...
}
void setupPaymentTypeIndicator() {
    paymentType = new JComboBox(paymentUIs);
    sharedPaymentDetails.add(paymentType);
}
void setupComponentsUniqueToEachPaymentType() {
    JPanel uniquePaymentDetails = new JPanel();
    uniquePaymentDetails.setLayout(new CardLayout());
    for (int i = 0; i < paymentUIs.length; i++) {
        PaymentUI UI = paymentUIs[i];
        uniquePaymentDetails.add(UI.getPanel(), UI.toString());
    }
    Container contentPane = getContentPane();
    contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
}
Payment editPayment(Payment payment) {
    displayPayment(payment);
    setVisible(true);
    return newPaymentToReturn;
 }
 void displayPayment(Payment payment) {
```

```
            for (int i = 0; i < paymentUIs.length; i++) {
                PaymentUI UI = paymentUIs[i];
                if (UI.tryToDisplayPayment(payment)) {
                    paymentType.setSelectedItem(UI);
                }
            }
        }
        void onOK() {
            newPaymentToReturn = makePayment();
            dispose();
        }
        Payment makePayment() {
            PaymentUI UI = (PaymentUI)paymentType.getSelectedItem();
            return UI.makePayment();
        }
    }
```

8.这是个控制电饭煲的嵌入式系统。该系统每秒钟都要做这些事：检查一下这个蒸机是否过热（像短路的话，就会过热）。如果过热的话，该系统就是自动断开电源，并用内置的发声器发出警报；会检查里面的湿度是否低于一定的值（像饭煮好了，湿度就低于该值）。如果湿度低于该值的话，自动转换到一个内置的 50 度的加热器，用来为里面的米饭保温。

将来，你还期望可以让这个嵌入式系统每秒钟多做一些其他的事情。

指出并修正下面代码的问题

```
class Scheduler extends Thread {
    Alarm alarm;
    HeatSensor heatSensor;
    PowerSupply powerSupply;
    MoistureSensor moistureSensor;
    Heater heater;
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            //检查是否过热
            if (heatSensor.isOverHeated()) {
                powerSupply.turnOff();
                alarm.turnOn();
            }
            //检查饭熟了没
            if (moistureSensor.getMoisture()<60) {
                heater.setTemperature(50);
            }
```

```
            }
        }
    }
```

如果按照需求所说的话，这个 run 方法肯定会越来越长。要避免这种情况的话，你首先要让检查是否过热的代码跟检查饭熟了没的代码一样（当然，是在一定的抽象程度一样）。

```
class Scheduler extends Thread {
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            //检查是否过热
            做些事情;
            //检查饭熟了没
            做些事情;
        }
    }
}
```

定义一个接口，比如 Task，然后派生两个实现类，分别做不同的事情：

```
interface Task {
    void doIt();
}
class Scheduler extends Thread {
    Task tasks[];
    void registerTask(Task task) {
        //增加任务;
    }
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            for (int i = 0; i < tasks.length; i++) {
                tasks[i].doIt();
            }
        }
    }
}
class OverHeatCheckTask implements Task {
    Alarm alarm;
    PowerSupply powerSupply;
    HeatSensor heatSensor;
    void doIt() {
```

```
            if (heatSensor.isOverHeated()) {
                powerSupply.turnOff();
                alarm.turnOn();
            }
        }
    }
class RickCookedCheckTask implements Task {
    Heater heater;
    MoistureSensor moistureSensor;
    void doIt() {
        if (moistureSensor.getMoisture()<60) {
            heater.setTemperature(50);
        }
    }
}
class Cooker {
    public static void main(String args[]) {
        Scheduler scheduler = new Scheduler();
        scheduler.registerTask(new OverHeatCheckTask());
        scheduler.registerTask(new RickCookedCheckTask());
    }
}
```

9.这个系统用来管理培训课程。课程的时间排列有三种方式：按每周排，按一个时间段排，和按一个日期列表排。按每周排就比如说“10 月 22 日以后五周内的每个星期二”。按一个时间段排就比较如“10 月 22 日到 11 月 3 日里每天上”。按一个日期列表排就像“10 月 22 日,10 月 25 日,11 月 3 日,11 月 10 日”。这回的练习，我们先忽略时间，就假定每天上课都是从晚上 7 点到晚上 10 点。以后要增加新的时间表排列方式。

指出并消除下面代码里的异味:

```
class Course {
    static final int WEEKLY=0;
    static final int RANGE=1;
    static final int LIST=2;
    String courseTitle;
    int scheduleType;  // 按每周排，按一个时间段排，或者按一个日期列表排
    int noWeeks;       // 按每周排
    Date fromDate;     // 按每周排或者按一个时间段排需要用到
    Date toDate;       // 按一个时间段排要用到
    Date dateList[];   // 按一个日期列表排要用到

    int getDurationInDays() {
        switch (scheduleType) {
```

```
                case WEEKLY:
                    return noWeeks;
                case RANGE:
                    int msInOneDay = 24*60*60*1000;
                    return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
                case LIST:
                    return dateList.length;
                default:
                    return 0;   // 未知的排列方式
            }
        }
        void printSchedule() {
            switch (scheduleType) {
                case WEEKLY:
                    //...
                case RANGE:
                    //...
                case LIST:
                    //...
            }
        }
    }
```

用到类别代码了。而且并不是所有的变量在各个情况下都要用到。我们应该将每种类别代码转为一个子类。注意，我们说的是时间排列的不同方式，并不是说不同的课程。

```
    interface Schedule {
        int getDurationInDays();
        void print();
    }
    class WeeklySchedule implements Schedule {
        int noWeeks;
        Date fromDate;
        Date toDate;
        int getDurationInDays() {
            return noWeeks;
        }
        void print() {
            ...
        }
    }
    class RangeSchedule implements Schedule {
        Date fromDate;
```

```
        Date toDate;
        int getDurationInDays() {
            int msInOneDay = 24*60*60*1000;
            return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
        }
        void print() {
            ...
        }
    }
    class ListSchedule implements Schedule {
        Date dateList[];
        int getDurationInDays() {
            return dateList.length;
        }
        void print() {
            ...
        }
    }
    class Course {
        String courseTitle;
        Schedule schedule;
        int getDurationInDays() {
            return schedule.getDurationInDays();
        }
        void printSchedule() {
            schedule.print();
        }
    }
```

10.这个系统用来管理培训课程。一个培训课程有名称，费用和时间段列表。不过，有时候一个课程是有几个模块组成的，每个模块又是一个课程。比如，现在有个混合的课程叫“快速成为网页开发员”，它有三个模块组成：“HTML”的课程，“FrontPage”的课程，还有“Flash”的课程。一个模块也有可能是有其他模块组成的。如果一个课程是由几个模块组成的话，那么该课程的费用跟时间表是由它的模块合在一起算的，因为这个课程自己没有费用跟时间表。

指出并消除下面代码里的异味:

```
    class Session {
        Date date;
        int startHour;
        int endHour;
        int getDuration() {
            return endHour-startHour;
```

```
    }
}
class Course {
    String courseTitle;
    Session sessions[];
    double fee;
    Course modules[];

    Course(String courseTitle, double fee, Session sessions[]) {
        //...
    }
    Course(String courseTitle, Course modules[]) {
        //...
    }
    String getTitle() {
        return courseTitle;
    }
    double getDuration() {
        int duration=0;
        if (modules==null)
            for (int i=0; i<sessions.length; i++)
                duration += sessions[i].getDuration();
        else
            for (int i=0; i<modules.length; i++)
                duration += modules[i].getDuration();
        return duration;
    }
    double getFee() {
        if (modules==null)
            return fee;
        else {
            double totalFee = 0;
            for (int i=0; i<modules.length; i++)
                totalFee += modules[i].getFee();
            return totalFee;
        }
    }
    void setFee(int fee) throws Exception {
        if (modules==null)
            this.fee = fee;
        else
            throw new Exception("Please set the fee of each module one by one");
    }
```

```
}
```

有些变量不是每种情况都需要的。事实上，这里面有两类课程：一些包含子模块（CompoundCourse），还有一类没有（SimpleCourse）。

if-then-else 这种表达式被不同的方法重复了很多次。所以我们应该将这两类课程划到两个子类里面。有些方法比如 setFee，就只被一个子类用到，所以不应该放在父类中。

```
class Session {
    Date date;
    int startHour;
    int endHour;
    int getDuration() {
        return endHour-startHour;
    }
}
abstract class Course {
    String courseTitle;
    Course(String courseTitle) {
        ...
    }
    String getTitle() {
        return courseTitle;
    }
    abstract double getFee();
    abstract double getDuration();
}
class SimpleCourse extends Course {
    Session sessions[];
    double fee;
    SimpleCourse(String courseTitle, double fee, Session sessions[]) {
        ...
    }
    double getFee() {
        return fee;
    }
    double getDuration() {
        int duration=0;
        for (int i=0; i<sessions.length; i++) {
            duration += sessions[i].getDuration();
        }
        return duration;
    }
    void setFee(int fee) {
```

```
            this.fee = fee;
        }
    }
    class CompoundCourse extends Course {
        Course modules[];
        CompoundCourse(String courseTitle, Course modules[]) {
            ...
        }
        double getFee() {
            double totalFee = 0;
            for (int i=0; i<modules.length; i++) {
                totalFee += modules[i].getFee();
            }
            return totalFee;
        }
        double getDuration() {
            int duration=0;
            for (int i=0; i<modules.length; i++) {
                duration += modules[i].getDuration();
            }
            return duration;
        }
    }
```

11.指出并消除下面代码里的异味:

```
class BookRental {
    String bookTitle;
    String author;
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? 1.2*rentalFee : rentalFee;
    }
}
class MovieRental {
    String movieTitle;
    int classification;
```

```
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
    }
}
```

有一些变量跟方法在两个类里面重复了。它们应该被抽取到父类 Rental 中。两个类里面的 getTotalFee 方法的结构其实是一样的。把它们抽象成一样的代码，抽取到父类中吧。

```
abstract class Rental {
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? getFeeWhenOverdue() : rentalFee;
    }
    abstract double getFeeWhenOverdue();
}
class BookRental extends Rental {
    String bookTitle;
    String author;
    double getFeeWhenOverdue() {
        return 1.2*rentalFee;
    }
}
class MovieRental extends Rental {
    String movieTitle;
    int classification;
    double getFeeWhenOverdue() {
        return Math.max(1.3*rentalFee, rentalFee+20);
    }
}
```

12.指出并消除下面代码里的异味:

```
class Customer {
    String homeAddress;
    String workAddress;
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    //"H": 送到客户家庭住址
    //"W": 送到客户办公地址
    //"O": 送到客户指定的其他地址
    String addressType;
    String otherAddress; //如果是指定的其他地址的话
    HashMap orderItems;
    public String getDeliveryAddress() {
        if (addressType.equals("H")) {
            return customerPlacingOrder.getHomeAddress();
        } else if (addressType.equals("W")) {
            return customerPlacingOrder.getWorkAddress();
        } else if (addressType.equals("O")) {
            return otherAddress;
        } else {
            return null;
        }
    }
}
```

Order 这个类中有类别代码。otherAddress 这个变量并不是任何情况都需要的。我们现在将每种类别代码转为类：

```
class Customer {
    String homeAddress;
    String workAddress;
}
interface DeliveryAddress {
}
class HomeAddress implements DeliveryAddress {
    Customer customer;
    HomeAddress(Customer customer) {
        ...
    }
    String toString() {
```

```
            return customer.getHomeAddress();
        }
}
class WorkAddress implements DeliveryAddress {
        Customer customer;
        WorkAddress(Customer customer) {
            ...
        }

    String toString() {
            return customer.getWorkAddress();
    }
}
class SpecifiedAddress implements DeliveryAddress {
    String addressSpecified;
    SpecifiedAddress(String addressSpecified) {
            ...
    }
    String toString() {
            return addressSpecified;
    }
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    DeliveryAddress deliveryAddress;
    HashMap orderItems;
    public String getDeliveryAddress() {
            return deliveryAddress.toString();
    }
}
```

第 4 章　保持代码简洁

示例

这是一个会议管理系统。它用来管理所有参会者的信息。刚开始的时候，我们只需要记录每个参会者的 ID（这是会议组织者分配的），姓名，电话和地址就行。于是，我们写了如下的代码：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}

class ConferenceSystem {
    Participant participants[];
}
```

接着，新的需求来了：现在每个参会者都可以让组织者帮忙预订酒店，所以我们要记录下他想预订的酒店名，入住日期，离开日期，房间类型（单人房或者双人房）。于是我们又扩充成如下的代码：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
    boolean bookHotelForHim;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
    void setHotelBooking(String hotelName, Date checkInDate, ...) {
        ...
    }
}
```

接着，又有一个新的需求来了：参会者可以参加不同的研讨会，所以我们要记录下参会者参加的研讨会。对于他要参加的每一场研讨会，我们还要记录下他的登记时间，同时他还需要什么翻译设备。于是代码又扩充成：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
    boolean bookHotelForHim;
```

```
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
    String idOfSeminarsRegistered[];
    Date seminarRegistrationDates[];
    boolean needSIDeviceForEachSeminar[];
    void setHotelBooking(String hotelName, Date checkInDate, ...) {
        ...
    }
    void registerForSeminar(String seminarId, Date regDate, boolean needSIDevice) {
        //将 seminarId 加到 idOfSeminarsRegistered
        //将 regDate 加到 seminarRegistrationDates
        //将 needSIDevice 加到 needSIDeviceForEachSeminar.
    }
    boolean isRegisteredForSeminar(String seminarId) {
        ...
    }
    Date getSeminarRegistrationDate(String seminarId) {
        ...
    }
    boolean needSIDeviceForSeminar(String seminarId) {
        ...
    }
    String [] getAllSeminarsRegistered() {
        return idOfSeminarsRegistered;
    }
}
```

代码开始肿胀起来了

请注意，这已经是我们第二次扩充 Participant 这个类了。每扩充一次，它就包含了更多的代码（实例变量和方法）及更多的功能。本来它只有 4 个属性。现在已经是 12 个了！此外，这个类要处理的业务逻辑也极大的增加了。本来它只需要处理参会者的基本信息（姓名，地址等等），现在它还要包含酒店，酒店预订，研讨会和翻译设备等等的逻辑。如果以后新的需求又来了，我们又要扩充 Participant 这个类，到时候，这个类要复杂庞大成什么样子！

所以我们得修整这个类了！

那怎么修整 Participant 这个类呢？怎么让它一直保持在第一天那样的简洁度？在回答这两个问题之前，我们先来考虑一下另一个需要优先回答的问题：给你一个类，你怎么认定它需要修整？

怎么判断一个类需要修整

要判断一个类是否需要修整，一个比较主观的方法是：当在读一个类的代码时，看看我们会不会觉得这个类“太长了”，“太复杂了”，或者讲的概念“太多了”？如果会这样觉得的话，我们就认定，这个类需要修整。

另外一个比较简单而且客观的方法是：当发现我们已经在第二次或者第三次扩充这个类的时候，我们认定这个类要修整了。这是一个比较”懒惰，被动”的方法，但却很有效。

现在让我们看一下怎么修整 Participant 这个类吧。

抽取出有关酒店预订的功能

首先，先来考虑一下怎么抽取出酒店预订的功能。一个可行的方案是：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}

class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}

class HotelBookings {
    HotelBooking hotelBookings[];
    void addBooking(HotelBooking booking) {
        ...
    }
}

class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
}
```

现在，Participant 这个类就一点都不知道酒店预订的存在。当然，我们不一定要用数组来存放酒店预订情况。

比如，我们可以用 Map：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}

class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}

class HotelBookings {
    HashMap mapFromPartIdToHotelBooking;
    //必须提供参会者 id
    void addBooking(String participantId, HotelBooking booking) {
        ...
    }
}

class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
}
```

这样的方案优点是 Participant 一点都不知道 HotelBooking 的存在，Participant 不依赖于 HotelBooking。还有另一个可行的方案是：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
    HotelBooking hotelBooking;
}

class HotelBooking {
    String hotelName;
```

```
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}

class ConferenceSystem {
    Participant participants[];
}
```

注意到，在这种方案里面，Participant 这个类还是要知道 HotelBooking 的存在，也就是说，Participant 还是要知道有酒店预订这回事。只是具体酒店预订是怎么做的，这些真正的功能是放在 HotelBooking 这个里面实现的。因为每个 Participant 都直接引用了本人的酒店预订情况，所以可以直接找到他的酒店预订情况。而代价就是，Pariticipant 还是要知道酒店预订的概念。从类的关系来讲，Pariticipant 还要依赖 HotelBooking 这个类。

当然，除了以上几种情况，还有许多其他的可行方案。

抽取研讨会的相关功能

现在我们来考虑一下怎么抽取出研讨会的功能。一个可行的方案：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}

class SeminarRegistration {
    String participantId;
    String seminarId;
    Date registrationDate;
    boolean needSIDevice;
}

class SeminarRegistry {
    SeminarRegistration registrations[];
    void registerForSeminar(SeminarRegistration registration) {
        //将 registration 加到 registrations.
    }
    boolean isRegisteredForSeminar(String participantId, String seminarId) {
        ...
    }
    Date getSeminarRegistrationDate(String participantId, String seminarId) {
```

```
            ...
        }
        boolean needSIDeviceForSeminar(String participantId, String seminarId) {
            ...
        }
        SeminarRegistration[] getAllRegistrations(String participantId) {
            ...
        }
    }

    class ConferenceSystem {
        Participant participants[];
        SeminarRegistry seminarRegistry;
    }
```

当然，除了以上的方案以外，还有其他可行的方案，这里就先不讲。

改进后的代码

下面就是改进后的代码：

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}

class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}

class HotelBookings {
    HotelBooking hotelBookings[];
    void addBooking(HotelBooking booking) {
        ...
    }
}
```

```
class SeminarRegistration {
    String participantId;
    String seminarId;
    Date registrationDate;
    boolean needSIDevice;
}

class SeminarRegistry {
    SeminarRegistration registrations[];
    void registerForSeminar(SeminarRegistration registration) {
        //将 registration 加到 registrations.
    }
    boolean isRegistered (String participantId, String seminarId) {
        ...
    }
    Date getRegistrationDate(String participantId, String seminarId) {
        ...
    }
    boolean needSIDevice(String participantId, String seminarId) {
        ...
    }
    SeminarRegistration[] getAllRegistrations(String participantId) {
        ...
    }
}

class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
    SeminarRegistry seminarRegistry;
}
```

引述

单一职责原则（The Single Responsibility Principle）认为:每个类都应该只为一个理由而修改。当一个类包含许多其他的功能时，很明显违反了单一职责原则。要看更多细节的话，可以看：

http://www.objectmentor.com/resources/articles/srp.

http://c2.com/cgi/wiki?OneResponsibilityRule .

章节练习

介绍

有些问题可以测试前面几章讲的观点

问题

（译者认为：1－9 随便看一下就行了，注意，是随便看，不是不看，从 10 开始讲的是本章的观点）

1.下面的代码有一些重复代码：在许多地方，我们都要创建, 设置，执行，然后再关闭一个 prepared statement。如果要你不惜代价的移除这样的重复代码，怎么做？

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?,?,?,?,?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());
            //...
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteParticipant(String participantId){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
        try {
            st.setString(1, participantId);
            st.executeUpdate();
```

```
            } finally {
                st.close();
            }
        }
    }
```

2.这个系统可以让网络管理员记录下他们的应用服务器设置。根据下面的显示，有很多不同类型的应用服务器。用 Swing 设计一个 GUI，让用户可以编辑一个描述服务器信息的对象。用 CardLayout 的布局，用户在左边选择服务器类型时，右边要显示该类型对应的一些详细信息的编辑框。

```
class Server {
    String id;
    String CPUModel;
}

class DNSServer extends Server {
    String domainName;
}

class WINSServer extends Server {
    String replicationPartner;
    int replicationInterval;
}

class DomainController extends Server {
    boolean remainNT4Compatible;
}
```

3.指出并消除下面代码里的异味:

```
public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
    // mode 的值有：ADD, MOD, DEL
    String outResult = "";
    final String connName = "tmp";
    DBConnectionManager connMgr = DBConnectionManager.getInstance();
    Connection conP    = connMgr.getConnection(connName);
    Statement stmt = conP.createStatement();
    int upd = stmt.executeUpdate(sqlStmt);
    switch(upd) {
    case 1:
        if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
        else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
        else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
        break;
```

```
        case 0:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
        default:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
        }
        stmt.close();
        conP.close();
      return outResult;
  }
```

4.指出并消除下面代码里的异味 ：

```
class Account {
    final public int SAVING=0;
    final public int CHEQUE=1;
    final public int FIXED=2;    //葡萄牙币的汇率
    private int accountType;
    private double balance;
    public double getInterestRate(...) { // 一些方法;
        ...
    }
    public Account(int accountType) {
        this.accountType=accountType;
    }
    public double calcInterest() {
        switch (accountType) {
            case SAVING:
                return balance*getInterestRate();
            case CHEQUE:
                return 0;
            case FIXED:
                return balance*(getInterestRate()+0.02);
        }
    }
}
```

5.指出并消除下面代码里的异味：

```
class Department{
    final public int Account =0;
    final public int Marketing = 1;
    final public int CustomerServices = 2;
    protected int departmentCode;
    public Department(int departmentCode){
        this.departmentCode = departmentCode;
    }
    public String getDepartmentName(){
        switch (departmentCode){
            case Account:
                return "Account";
            case Marketing:
                return "Marketing";
            case CustomerServices:
                return "Customer Services";
        }
    }
}
```

6.指出并消除下面代码里的异味 ：

```
class NormalPayment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate;
        double tax = baseAmt * TAX_RATE;
        return baseAmt + tax;
    }
}

class PaymentForSeniorCitizen {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate * 0.8;
        double tax = baseAmt * (TAX_RATE - 0.5) ;
        return baseAmt + tax;
    }
```

```
}
```

7.指出并消除下面代码里的异味：

```
class PianoKey {
    final static int key0 = 0;
    final static int key1 = 1;
    final static int key2 = 2;
    int keyNumber;
    public void playSound() {
        if (keyNumber == 0) {
            //播放 key0 的频率
        }
        else if (keyNumber == 1) {
            //播放 key1 的频率
        }
        else if (keyNumber == 2) {
            //播放 key2 的频率
        }
    }
}

class Piano {
    Vector rythmn;
    public void play() {
        for(int i=0;i<rythmn.size();i++) {
            rythmn.getElementAt(i).playSound();
        }
    }
}
```

8.指出并消除下面代码里的异味：

```
class Account {
    final static int LEVEL_USER = 1;
    final static int LEVEL_ADMIN = 2;
    int accountLevel;
    Date expiredDate; // 用户账号要用到
    boolean hasLogin; // 管理员账号要用到
}

class ERPApp {
    public boolean checkLoginIssue(Account account) {
```

```
            if (account.getLevel() == Account.LEVEL_USER) {
                // 检查账号的使用截止日期
                Date now = new Date();
                if (account.getExpiredDate().before(now))
                    return false;
                return true;
            }
            else if (account.getLevel() == Account.LEVEL_ADMIN) {
                // 管理员账号永不过期
                // 检查是否重复登录
                if (account.hasLogin())
                    return false;
                return true;
            }
            return false;
        }
    }
```

9.指出并消除下面代码里的异味：

```
    class Form1 extends JDialog {
        JComboBox comboBoxReportType;
        Form1() {
            comboBoxReportType = new JComboBox();
            comboBoxReportType.addItem("r1");
            comboBoxReportType.addItem("r2");
            ...
            comboBoxReportType.addItem("r31c");
        }
        void processReport1() {
            //打印一些特定的报表...
        }
        void processReport2() {
            //打印另外的完全不同的报表...
        }
        ...
        void processReport31c() {
            //打印第三种完全不同的报表...
        }
        void printReport(String repNo) {
            if (repNo.equals("r1"))
                processReport1();
            else if (repNo.equals("r2"))
```

```
                processReport2();
            ...
            else if (repNo.equals("r31c"))
                processReport31c();
        }
        void onPrintClick() {
            printReport((String) comboBoxReportType.getSelectedItem());
        }
    }
```

10.这个系统跟餐厅有关。每个餐厅在我们的系统上都有一个账号。初始的时候，我们创建一个 Restaurant 类，用来表示餐厅的账号，里面包含名称，访问系统的密码，电话，传真，地址等跟该餐厅有关的信息：

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
}
```

然后，下面新的需求依次到来：

1.初始注册后，系统给每个餐厅账号分配一个激活码。只有用户输入激活码以后，账号才会被激活。激活后的账号才开始有效，并可以登录。

2.用户要修改传真的时候，新输入的传真号码并不会马上生效（仍然是旧的传真号码在生效）。系统将再分配一个激活码给这个账号，当用户输入正确的激活码以后，新的传真号码才生效。

3.一个餐厅属于一个特定的分类（比如，中餐厅，葡萄牙餐厅等等）。每种分类都有一个分类 ID。

4.用户可以给餐厅输入节假日。

5.用户可以给餐厅输入营业时段。

以上 5 个需求实现以后。这个类就变得下面这样复杂了：

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
    String verificationCode;
    boolean isActivated;
    String faxNoToBeConfirmed;
```

```
    boolean isThereFaxNoToBeConfirmed = false;
    String catId;
    Vector holidays;
    Vector businessSessions;
    void activate(String verificationCode) {
        isActivated=(this.verificationCode.equals(verificationCode));
        if (isActivated && isThereFaxNoToBeConfirmed){
            faxNo = faxNoToBeConfirmed;
            isThereFaxNoToBeConfirmed = false;
        }
    }
    void setFaxNo(String newFaxNo) {
        faxNoToBeConfirmed = newFaxNo;
        isThereFaxNoToBeConfirmed = true;
        isActivated = false;
    }
    boolean isInCategory(String catId) {
        return this.catId.equals(catId);
    }
    void addHoliday(int year, int month, int day) {
        ...
    }

    void removeHoliday(int year, int month, int day) {
        ...
    }
    boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin){
        ...
    }
    boolean isInBusinessHour(Calendar time) {
        ...
    }
    Vector getAllHolidays() {
        return holidays;
    }
    Vector getAllBusinessSessions() {
        return businessSessions;
    }
}
```

你的任务就是在分开的类里面将上面的 5 个需求实现，原来的那个简单的 Restaurant 这个类不要作修改。

11.这个系统跟学生有关。原来我们有一个简单的 Student 类：

```
class StudentManagementSystem {
    Student students[];
}

class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}
```

接着，为了记录学生参加了哪些课程，参加日期和交费的方式，我们将代码修改成：

```
class StudentManagementSystem {
    Student students[];
}

class Student {
    String studentId;
    String name;
    Date dateOfBirth;
    String courseCodes[]; //学生参加了这些课程
    Date enrollDates[];   //每个课程的开始时间
    Payment payments[];   //每个课程的交费方式
    void enroll(String courseCode, Date enrollDate, Payment payment) {
        //将 courseCode  加到  courseCodes
        //将 enrollDate 加到  enrollDates
        //将  payment 加到  Payments
    }
    void unenroll(String courseCode) {
        ...
    }
}
```

你的任务就是在不修改 Student 这个类的前提下，实现后面的这些需求。

12.这个系统可以让网络管理员记录他们的服务器配置。原来我们只有下面这个简单的 Server 类：

```
class Server {
    String name;
    String CPUModel;
    int RAMSizeInMB;
    int diskSizeInMB;
```

```
        InetAddress ipAddress;
    }

    class ServerConfigSystem {
        Server servers[];
    }
```

现在，有了四条新的需求。要求你在不修改 Server 类的前提下实现这些需求：

1.每个管理员（各自有个管理员 ID）分配去专门管理一台服务器。

2.确认一台服务器是否是 DHCP 服务器。如果是的话，记下这台服务器管理的 IP 段（比如：192.168.0.21 到 192.168.0.254）。

3.确认一台服务器是不是文件服务器。如果是的话，要求可以为每个用户（以用户 ID 来区分）分配磁盘空间，也可以确认每个用户已分配的磁盘空间。

4.一台服务器可以同时是文件服务器跟 DHCP 服务器。

13.目前某个系统的代码如下所示:

```
    class Customer {
        String id;
        String name;
        String address;
    }

    class Supplier {
        String id;
        String name;
        String telNo;
        String address;
    }

    class SalesSystem {
        Vector customers;
        Vector suppliers;
    }
```

在保持代码简洁的前提下，实现下面的新需求：

1.客户可以订货。每个订单有一个 ID 区分。订单要记录预订日期，客户的 ID，预订的每项商品的名字跟数量。

2.系统可以根据客户列出所有的订单。

3.系统可以根据供应商列出所有的订单。

4.供应商可以为它指定的一些客户打折（只对一些指定的商品有效）。不同的客户、不同的商品都可以有不可的折扣。

14.找出一些本来很简洁，但因为实现了新需求以后，变得很臃肿的类。然后将新增的代码抽取到不同的类去，让结构变得跟原来一样简洁。

提示

1.这个跟 RentalProcessor 的例子很相似。首先，在一度的程度上抽象出相同的伪码：

```
class ParticipantsInDB
    ...
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

2.这个可以参考 EditPaymentDialog。

3.不同的"mode"值，并没有造成不同的行为。因此，你没必要为"mode"的每个值创建一个子类（如果你已经创建了的话，你会发现，这些类会非常相似）。“upd“同样如此。

4.没有提示。

5.在"departmentCode"不同的值的情况下，并没有造成什么行为的不同。因此没必要为这些类别创建子类。

6.没有提示。

7.key 之间的不同，其实也就是频率（frequency）这样一个值的不同。因此，用包含一个 int 型属性（描述频率的值）的对象来描述 key 的不同就行了。没必要每种 key 都创建一个子类。
8.没有提示。

9.没有提示。

10.系统应该保存一个所有已激活餐厅的列表。当一个新的餐厅账号对象被创建，但还没激活时，先不要将这个餐厅对象存到已激活列表里面，而是先将这个新的餐厅对象的信息存放到一个临时对象 A 里面。对象 A 应该有这样一个激活的方法，如：activate(餐厅 id,激活号码)。当用户为这个新的餐厅账号输入激活码时，系统调用 A 的这个方法。验证通过的话，系统就将 A 里面所存的餐厅对象加到已激活餐厅列表里面，然后把 A 销毁，因为 A 已经完成它的职责了。

要实现激活新的传真号码这个需求，做一些跟上面类似的事。创建一个临时对象 B。当用户激活新传真号码成功时，将新的传真号码存到餐厅对象中，然后销毁 B。

注意到，我们要实现上面这两个需求的代码中，会有一些重复代码，因为这两段代码做了一些同样的事，所以记得移除重复代码。

有关分类，节假日和营业时段这些需求，应该很好实现，不多讲。

11.没有提示。

12.做第 4 条需求时，你不能用继承。尝试用一个 DHCPConfiguration 的类代替 DHCPServer。

13.创建 Orders 和 Discounts 这样的类.

解决方法示例

1.下面的代码有一些重复代码：在许多地方，我们都要创建一个 prepared statement, 设置它，执行它，然后再关闭它。如果要你不惜代价的移除这些重复代码，怎么做？

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?,?,?,?,?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());
            //...
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
```

```
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteParticipant(String participantId){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
        try {
            st.setString(1, participantId);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

首先，在一定的抽象高度上让伪码一样：

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteParticipant(String participantId){
```

```
            PreparedStatement st = conn.prepareStatement(一些 SQL);
            try {
                设置 st 的参数
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
    }
```

将重复的代码抽取到一个方法里面：

```
class ParticipantsInDB {
    ...
    void ???() {
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

现在为这个方法取一个好名字，我们看看这个方法做了什么事。在这边，它创建，执行和关闭了一个 SQL 语句。所以我们可以用“executeSQL”：

```
class ParticipantsInDB {
    ...
    void executeSQL() {
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            设置 st 的参数
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

因为"设置 st 的参数"这个伪码其实多种具体的实现，我们现在要创建一个接口，让它可以有多个实现：

```
interface ??? {
```

```
    void setParametersOf(PreparedStatement st);
}

class ParticipantsInDB {
    ...
    void executeSQL(??? someObject) {
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            someObject.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

现在要为这个接口取个名字，看看它做了些什么事。它只有一个方法(setParametersOf)，用来设置 prepared statement 的参数。所以取作"StatementParamsSetter"：

```
interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}

class ParticipantsInDB {
    ...
    void executeSQL(StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(一些 SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

"一些 SQL"在不同的情况有不同的值，不过其实这边并不是一个类别代码，而仅仅是值的不同而已，我们把它放在参数里面传递进来即可：

```
interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}
class ParticipantsInDB {
    ...
```

```
    void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

最后，提供不同的实现类。可以的话，将这些实现类做成内类或匿名类：

```
interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}

class ParticipantsInDB {
    ...
    void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void addParticipant(final Participant part){
        executeSQL("INSERT INTO "+tableName+" VALUES (?,?,?,?,?)",
            new StatementParamsSetter() {
                void setParametersOf(PreparedStatement st) {
                    st.setString(1, part.getId());
                    st.setString(2, part.getEFirstName());
                    st.setString(3, part.getELastName());
                }
            });
    }
    void deleteAllParticipants(){
        executeSQL("DELETE FROM "+tableName,
            new StatementParamsSetter() {
                void setParametersOf(PreparedStatement st) {
                }
```

```
                });
        }
        void deleteParticipant(final String participantId){
            executeSQL("DELETE FROM "+tableName+" WHERE id=?",
                new StatementParamsSetter() {
                    void setParametersOf(PreparedStatement st) {
                        st.setString(1, participantId);
                    }
                });
        }
    }
```

2.这个系统可以让网络管理员记录下他们的应用服务器设置。根据下面的显示，有很多不同类型的应用服务器。用 Swing 设计一个 GUI，让用户可以编辑一个描述服务器信息的对象。用 CardLayout 的布局，用户在左边选择服务器类型时，右边要显示该类型对应的一些详细信息的编辑框。

```
class Server {
    String id;
    String CPUModel;
}

class DNSServer extends Server {
    String domainName;
}

class WINSServer extends Server {
    String replicationPartner;
    int replicationInterval;
}

class DomainController extends Server {
    boolean remainNT4Compatible;
}
```

为每个 server 类创建一个 UI 类：

```
abstract class ServerUI {
    JPanel panelForThisServerType;
    JTextField id;
    JTextField CPUModel;
    ServerUI(JTextField id, JTextField CPUModel) {
        this.id = id;
```

```
        this.CPUModel = CPUModel;
        this.panelForThisServerType = new JPanel(...);
    }
    void showServerCommonInfo(Server server) {
        id.setText(server.getId());
        CPUModel.setText(server.getCPUModel());
    }
    String getIdFromUI() {
        return id.getText();
    }
    String getCPUModelFromUI() {
        return CPUModel.getText();
    }
    JPanel getPanel() {
        return panelForThisServerType;
    }
    abstract boolean tryToDisplayServer(Server server);
    abstract Server makeServer();
}

class DNSServerUI extends ServerUI {
    JTextField domainName;
    DNSServerUI(JTextField id, JTextField CPUModel) {
        super(id, CPUModel);
        domainName = new JTextField(...);
        panelForThisServerType.add(domainName);
    }
    String toString() {
        return "DNS";
    }
    boolean tryToDisplayServer(Server server) {
        if (server instanceof DNSServer) {
            DNSServer dnsServer = (DNSServer)server;
            showServerCommonInfo(server);
            domainName.setText(dnsServer.getDomainName());
            return true;
        }
        return false;
    }
    Server makeServer() {
        return new DNSServer(
                getIdFromUI(),
                getCPUModelFromUI(),
```

```
                domainName.getText());
    }
}

class WINSServerUI extends ServerUI {
    JTextField replicationPartner;
    JTextField replicationInterval;
    WINSServerUI(JTextField id, JTextField CPUModel) {
        super(id, CPUModel);
        replicationPartner = new JTextField(...);
        replicationInterval = new JTextField(...);
        panelForThisServerType.add(replicationPartner);
        panelForThisServerType.add(replicationInterval);
    }
    String toString() {
        return "WINS";
    }
    boolean tryToDisplayServer(Server server) {
        ...
    }
    Server makeServer() {
        ...
    }
}

class DomainControllerUI extends ServerUI {
    ...
}

class ServerDialog extends JDialog {
    Server newServerToReturn;
    JPanel panelForCommonComponents;
    JPanel panelForServerTypeDependentComponents;
    JTextField id;
    JTextField CPUModel;
    JComboBox serverType;
    ServerUI serverUIs[];
    ServerDialog() {
        setupCommonComponents();
        setupServerUIs();
        setupServerTypeCombo();
        setupServerTypeDependentComponents();
    }
```

```
void setupCommonComponents() {
    panelForCommonComponents = new JPanel(...);
    id = new JTextField(...);
    CPUModel = new JTextField(...);
    panelForCommonComponents.add(id);
    panelForCommonComponents.add(CPUModel);
    Container contentPane = getContentPane();
    contentPane.add(panelForCommonComponents, BorderLayout.NORTH);
}
void setupServerUIs() {
    ServerUI serverUIs[] = {
        new DNSServerUI(id, CPUModel),
        new WINSServerUI(id, CPUModel),
        new DomainControllerUI(id, CPUModel)
    };
    this.serverUIs = serverUIs;
}
void setupServerTypeCombo() {
    serverType = new JComboBox(serverUIs);
    panelForCommonComponents.add(serverType);
 }
 void setupServerTypeDependentComponents() {
    panelForServerTypeDependentComponents = new JPanel(...);
    panelForServerTypeDependentComponents.setLayout(new CardLayout());
    for (int i = 0; i < serverUIs.length; i++) {
        ServerUI UI = serverUIs[i];
         panelForServerTypeDependentComponents.add(
            UI.getPanel(),
            UI.toString());
    }
    Container contentPane = getContentPane();
    contentPane.add(
        panelForServerTypeDependentComponents,
        BorderLayout.CENTER);
 }
 Server editServer(Server server) {
    displayServer(server);
    setVisible(true);
    return newServerToReturn;
 }
 void displayServer(Server server) {
    for (int i = 0; i < serverUIs.length; i++) {
        ServerUI serverUI = serverUIs[i];
```

```
                if (serverUI.tryToDisplayServer(server)) {
                    serverType.setSelectedItem(serverUI);
                }
            }
        }
        void onOK() {
            newServerToReturn = makeServer();
            dispose();
        }
        Server makeServer() {
            ServerUI serverUI = (ServerUI)serverType.getSelectedItem();
            return serverUI.makeServer();
        }
    }
```

3.指出并消除下面代码里的异味：

```
    public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
        // mode 的值有：ADD, MOD, DEL
        String outResult = "";
        final String connName = "tmp";
        DBConnectionManager connMgr = DBConnectionManager.getInstance();
        Connection conP   = connMgr.getConnection(connName);
        Statement stmt = conP.createStatement();
        int upd = stmt.executeUpdate(sqlStmt);
        switch(upd) {
        case 1:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
            break;
        case 0:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
        default:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
        }
        stmt.close();
        conP.close();
```

```
        return outResult;
    }
```

这三个分支里面有太多的重复代码了。而且这些重复代码之间的区别不会很大。对于这里的类别代码，只需要用同一个类的不同的实例对象，来表示不同的类别的值就行了。（没必要创建多个类了）

```
    public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
        // mode 的值有：ADD, MOD, DEL
        String outResult = "";
        final String connName = "tmp";
        DBConnectionManager connMgr = DBConnectionManager.getInstance();
        Connection conP   = connMgr.getConnection(connName);
        Statement stmt = conP.createStatement();
        int upd = stmt.executeUpdate(sqlStmt);
        outResult = (upd==0 ? "SUC" : "Err")+
                    " "+
                    capitalizeString(mode);
        stmt.close();
        conP.close();
      return outResult;
    }
    public static String capitalizeString(String string) {
        ...
    }
```

4.指出并消除下面代码里的异味 ：

```
    class Account {
        final public int SAVING=0;
        final public int CHEQUE=1;
        final public int FIXED=2;   //葡萄牙币的汇率
        private int accountType;
        private double balance;
        public double getInterestRate(...) { // 一些方法；
            ...
        }
        public Account(int accountType) {
            this.accountType=accountType;
        }
        public double calcInterest() {
            switch (accountType) {
                case SAVING:
                    return balance*getInterestRate();
                case CHEQUE:
                    return 0;
                case FIXED:
```

```
                return balance*(getInterestRate()+0.02);
            }
        }
}
```

将这里面的类别代码SAVING,CHEQUE,FIXED转为不同的类。这里面还有个错误的注释("葡萄牙币的汇率")，直接移掉。

```
abstract class Account {
    private double balance;
    abstract public double calcInterest();
    public double getInterestRate(...) { //  一些方法；
        ...
    }
}

class SavingAccount extends Account {
    public double calcInterest() {
        return getBalance()*getInterestRate();
    }
}

class ChequeAccount extends Account {
    public double calcInterest() {
        return 0;
    }
}

class FixedAccount extends Account {
    public double calcInterest() {
        return getBalance()*(getInterestRate()+0.02);
    }
}
```

5.指出并消除下面代码里的异味 ：

```
class Department{
    final public int Account =0;
    final public int Marketing = 1;
    final public int CustomerServices = 2;
    protected int departmentCode;
    public Department(int departmentCode){
        this.departmentCode = departmentCode;
    }
    public String getDepartmentName(){
        switch (departmentCode){
```

```
            case Account:
                return "Account";
            case Marketing:
                return "Marketing";
            case CustomerServices:
                return "Customer Services";
        }
    }
}
```

用同一个类的不同实例代表不同的类别（无需用不同的类）。

```
class Department {
    final public Department Account = new Department("Account");
    final public Department Marketing = new Department("Marketing");
    final public Department CustomerServices =
        new Department("Customer Services ");
    private String departmentName;
    private Department(String departmentName){
        this.departmentName = departmentName;
    }
    public String getDepartmentName(){
        return departmentName;
    }
}
```

6.指出并消除下面代码里的异味：

```
class NormalPayment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate;
        double tax = baseAmt * TAX_RATE;
        return baseAmt + tax;
    }
}

class PaymentForSeniorCitizen {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate * 0.8;
```

```
            double tax = baseAmt * (TAX_RATE - 0.5) ;
            return baseAmt + tax;
        }
    }
```

这两个类里面的重复代码太多了。将这些重复代码抽取到父类中吧：

```
abstract class Payment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    abstract double getPreTaxedAmount();
    abstract double getTaxRate();
    double getBillableAmount() {
        return getPreTaxedAmount() * (1 + getTaxRate());
    }
    double getNormalAmount() {
        return units * rate;
    }
}

class NormalPayment extends Payment {
    double getPreTaxedAmount() {
        return getNormalAmount();
    }
    double getTaxRate() {
        return TAX_RATE;
    }
}

class PaymentForSeniorCitizen extends Payment {
    double getPreTaxedAmount() {
        return getNormalAmount()*0.8;
    }
    double getTaxRate() {
        return TAX_RATE - 0.5;
    }
}
```

7.指出并消除下面代码里的异味：

```
class PianoKey {
    final static int key0 = 0;
```

```
        final static int key1 = 1;
        final static int key2 = 2;
        int keyNumber;
        public void playSound() {
            if (keyNumber == 0) {
                //播放 key0 的频率
            }
            else if (keyNumber == 1) {
                //播放 key1 的频率
            }
            else if (keyNumber == 2) {
                //播放 key2 的频率
            }
        }
    }

    class Piano {
        Vector rythmn;
        public void play() {
            for(int i=0;i<rythmn.size();i++) {
                ((PianoKey) rythmn.elementAt(i)).playSound();
            }
        }
    }
```

将类别代码跟这一大串的 if-then-else-if 移除。这里面的类别可以用同一类的不同实例来描述（无需用不同的类）。

```
    class PianoKey {
        final static PianoKey key0 = new PianoKey(frequency for key0);
        final static PianoKey key1 = new PianoKey(frequency for key1);
        final static PianoKey key2 = new PianoKey(frequency for key2);
        ...
        int frequency;
        private PianoKey(int frequency) {

            this.frequency = frequency;
        }
        public void playSound() {
            //播放频率
        }
    }
```

```
class Piano {
    Vector rythmn;
    public void play() {
        for(int i=0;i<rythmn.size();i++) {
            ((PianoKey) rythmn.elementAt(i)).playSound();
        }
    }
}
```

8.指出并消除下面代码里的异味：

```
class Account {
    final static int LEVEL_USER = 1;
    final static int LEVEL_ADMIN = 2;
    int accountLevel;
    Date expiredDate; // 用户账号要用到
    boolean hasLogin; // 管理员账号要用到
}

class ERPApp {
    public boolean checkLoginIssue(Account account) {
        if (account.getLevel() == Account.LEVEL_USER) {
            // 检查账号的使用截止日期
            Date now = new Date();
            if (account.getExpiredDate().before(now))
                return false;
            return true;
        }
        else if (account.getLevel() == Account.LEVEL_ADMIN) {
            // 管理员账号永不过期
            // 检查是否重复登录
            if (account.hasLogin())
                return false;
            return true;
        }
        return false;
    }
}
```

移除类别代码，用不同的类代替这些类别。将这些注释("检查账号的使用截止日期", "检查是否重复登录" 等等)转为代码：

```
interface Account {
    boolean canLogin();
```

```
}

class UserAccount implements Account {
    Date expiredDate;
    boolean canLogin() {
        return isAccountExpired();
    }
    boolean isAccountExpired() {
        Date now = new Date();
        return !getExpiredDate().before(now);
    }
}

class AdminAccount implements Account {
    boolean hasLogin;
    boolean canLogin() {
        return !isTryingMultiLogin();
    }
    boolean isTryingMultiLogin() {
        return hasLogin;
    }
}

class ERPApp {
    public boolean checkLoginIssue(Account account) {
        return account.canLogin();
    }
}
```

9.指出并消除下面代码里的异味：

```
class Form1 extends JDialog {
    JComboBox comboBoxReportType;
    Form1() {
        comboBoxReportType = new JComboBox();
        comboBoxReportType.addItem("r1");
        comboBoxReportType.addItem("r2");
        ...
        comboBoxReportType.addItem("r31c");
    }
    void processReport1() {
        //打印一些特定的报表...
    }
    void processReport2() {
```

```
        //打印另外的完全不同的报表...
    }
    ...
    void processReport31c() {
        //打印第三种完全不同的报表...
    }
    void printReport(String repNo) {
        if (repNo.equals("r1"))
            processReport1();
        else if (repNo.equals("r2"))
            processReport2();
        ...
        else if (repNo.equals("r31c"))
            processReport31c();
    }
    void onPrintClick() {
        printReport((String) comboBoxReportType.getSelectedItem());
    }
}
```

用不同的类代替不同的类别代码：

```
interface Report {
    void print();
}

class Report1 implements Report {
    String toString() {
        return "r1";
    }
    void print() {
        //打印一些报表
    }
}

class Report2 implements Report {
    String toString() {
        return "r2";
    }
    void print() {
        //打印另外的完全不同的报表...
    }
}
```

```
...
class Report31c implements Report {
    String toString() {
        return "31c";
    }
    void print() {
        //打印第三种完全不同的报表...
    }
}

class Form1 extends JDialog {
    JComboBox comboBoxReportType;
    Report reports[] = {
        new Report1(),
        new Report2(),
        ...
        new Report31c()
    };
    Form1() {
        comboBoxReportType = new JComboBox();
        for (int i = 0; i < reports.length; i++) {
            comboBoxReportType.addItem(reports[i]);
        }
    }
    void onPrintClick() {
        Report report = (Report)comboBoxReportType.getSelectedItem();
        report.print();
    }
}
```

10.这个系统跟餐厅有关。每个餐厅在我们的系统上都有一个账号。初始的时候，我们创建一个 Restaurant 类，用来表示餐厅的账号，里面包含名称，访问系统的密码，电话，传真，地址等跟该餐厅有关的信息：

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
}
```

然后，下面新的需求依次到来：

1.初始注册后，系统给每个餐厅账号分配一个激活码。只有用户输入激活码以后，账号才会被激活。激

活后的账号才开始有效，并可以登录。

2.用户要修改传真的时候，新输入的传真号码并不会马上生效（仍然是旧的传真号码在生效）。系统将再分配一个激活码给这个账号，当用户输入正确的激活码以后，新的传真号码才生效。

3.一个餐厅属于一个特定的分类（比如，中餐厅，葡萄牙餐厅等等）。每种分类都有一个分类 ID。

4.用户可以给餐厅输入节假日。

5.用户可以给餐厅输入营业时段。

以上 5 个需求实现以后。这个类就变得下面这样复杂了：

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
    String verificationCode;
    boolean isActivated;
    String faxNoToBeConfirmed;
    boolean isThereFaxNoToBeConfirmed = false;
    String catId;
    Vector holidays;
    Vector businessSessions;
    void activate(String verificationCode) {
        isActivated=(this.verificationCode.equals(verificationCode));
        if (isActivated && isThereFaxNoToBeConfirmed){
            faxNo = faxNoToBeConfirmed;
            isThereFaxNoToBeConfirmed = false;
        }
    }
    void setFaxNo(String newFaxNo) {
        faxNoToBeConfirmed = newFaxNo;
        isThereFaxNoToBeConfirmed = true;
        isActivated = false;
    }
    boolean isInCategory(String catId) {
        return this.catId.equals(catId);
    }
    void addHoliday(int year, int month, int day) {
        ...
    }
    void removeHoliday(int year, int month, int day) {
        ...
```

```
    }
    boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin) {
        ...
    }
    boolean isInBusinessHour(Calendar time) {
        ...
    }
    Vector getAllHolidays() {
        return holidays;
    }
    Vector getAllBusinessSessions() {
        return businessSessions;
    }
}
```

你的任务就是在分开的类里面将上面的 5 个需求实现，原来的那个简单的 Restaurant 这个类不要作修改。

首先，我们先将 Restaurant 这个类还原到原来的身形：

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
}
```

实现账号初始的激活：

```
class RestaurantActivator {
    String verificationCode;
    Restaurant restaurantToAdd;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurantToAdd.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //将 restaurantToAdd 这个对象加到已激活餐厅列表中
            return true;
        }
        return false;
    }
}

class RestaurantActivators {
    RestaurantActivator activators[];
```

```
    void activate(String restName, String verificationCode) {
        for (int i = 0; i < activators.length; i++) {
            if (activators[i].tryToActivate(restName, verificationCode)) {
                //将 activator[i]这个对象 从 activators 里面移走
                return;
            }
        }
    }
}

class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantActivators restaurantActivators;
}
```

实现设置传真号码的激活：

```
class FaxNoActivator {
    String verificationCode;
    String newFaxNo;
    Restaurant restaurantToEdit;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurantToEdit.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            restaurantToEdit.setFaxNo(newFaxNo);
            return true;
        }
        return false;
    }
}

class FaxNoActivators {
    FaxNoActivator activators[];
    void activate(String restName, String verificationCode) {
        for (int i = 0; i < activators.length; i++) {
            if (activators[i].tryToActivate(restName, verificationCode)) {
                //将 activator[i]这个对象 从 activators 里面移走
                return;
            }
        }
    }
}

class RestaurantSystem {
```

```
    Restaurant restaurants[];
    RestaurantActivators restaurantActivators;
    FaxNoActivators faxNoActivators;
}
```

不过，RestaurantActivator 跟 FaxNoActivator 这个类之间很多代码重复了。而且这两个类之间只有很小的差别。我们先让它们看起来一样吧：

```
class RestaurantActivator {
    String verificationCode;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //对 restaurant 这个对象做点事情
            return true;
        }
        return false;
    }
}

class FaxNoActivator {
    String verificationCode;
    String newFaxNo;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //对 restaurant 这个对象做点事情
            return true;
        }
        return false;
    }
}
```

现在，将同样的代码抽取到一个父类中，然后让 RestaurantActivator 和 FaxNoActivator 继承自它：

```
abstract class RestaurantTaskActivator {
    String verificationCode;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
```

```
                doSomethingToRestaurant();
                return true;
            }
            return false;
        }
        abstract void doSomethingToRestaurant();
}

class RestaurantActivator extends RestaurantTaskActivator {
        void doSomethingToRestaurant() {
            //将 restaurant 这个对象加到已激活列表中
        }
}

class FaxNoActivator extends RestaurantTaskActivator {
        String newFaxNo;
        void doSomethingToRestaurant() {
            restaurant.setFaxNo(newFaxNo);
        }
}

class RestaurantTaskActivators {
        RestaurantTaskActivator activators[];
        void activate(String restName, String verificationCode) {
            for (int i = 0; i < activators.length; i++) {
                if (activators[i].tryToActivate(restName, verificationCode)) {
                    //将 activator[i]这个对象 从 activators 里面移走
                    return;
                }
            }
        }
}

class RestaurantSystem {
        Restaurant restaurants[];
        RestaurantTaskActivators restaurantTaskActivators;
}
```

分类 ID 的实现：

```
class Category {
        String catId;
        String IdOfRestaurantsInThisCat[];
        boolean isInCategory(String restName) {
```

```
        ...
    }
}

class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
    Category categories[];
}
```

节假日的实现：

```
class Holidays {
    Vector holidays;
    void addHoliday(int year, int month, int day) {
        ...
    }
    void removeHoliday(int year, int month, int day) {
        ...
    }
    Vector getAllHolidays() {
        return holidays;
    }
}

class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
    Category categories[];
    HashMap mapRestIdToHolidays;
}
```

营业时段的实现：

```
class BusinessSessions {
    Vector businessSessions;
    boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin)
{
        ...
    }
    boolean isInBusinessHour(Calendar time) {
        ...
    }
    Vector getAllBusinessSessions() {
        return businessSessions;
```

```
        }
    }

    class RestaurantSystem {
        Restaurant restaurants[];
        RestaurantTaskActivators restaurantTaskActivators;
        Category categories[];
        HashMap mapRestIdToHolidays;
        HashMap mapRestIdToBusinessSessions;
    }
```

11.这个系统跟学生有关。原来我们有一个简单的 Student 类：

```
class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}
```

接着，为了记录学生参加了哪些课程，参加日期和交费的方式，我们将代码修改成：

```
class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
    String courseCodes[]; //学生参加了这些课程
    Date enrollDates[];  //每个课程的开始时间
    Payment payments[];  //每个课程的交费方式
    void enroll(String courseCode, Date enrollDate, Payment payment) {
        //将 courseCode 加到 courseCodes
        //add enrollDate to enrollDates
        //将 payment 加到 Payments
    }
    void unenroll(String courseCode) {
        ...
    }
}
```

你的任务就是在不修改 Student 这个类的前提下，实现后面的这些需求。

首先，我们先让 Student 焕发以前的苗条身形：

```
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}
```

实现登记课程的功能：

```
class Enrollment {
    String studentId;
    String courseCode;
    Date enrollDate;
    Payment payment;
}

class Enrollments {
    Enrollment enrollments[];
    void enroll(String studentId, String courseCode, Date enrollDate, Payment
payment) {
        Enrollment enrollment = new Enrollment(studentId, courseCode, ...);
        //将 enrollment 加到 enrollments
    }
    void unenroll(String studentId, String courseCode) {
        ...
    }
}

class StudentManagementSystem {
    Student students[];
    Enrollments enrollments;
}
```

12.这个系统可以让网络管理员记录他们的服务器配置。原来我们只有下面这样的一个简单的 Server 类：

```
class Server {
    String name;
    String CPUModel;
    int RAMSizeInMB;
    int diskSizeInMB;
    InetAddress ipAddress;
}
class ServerConfigSystem {
    Server servers[];
```

```
}
```

现在，有了四条新的需求。要求你在不修改 Server 这个类的前提下实现这些需求：

1.每个管理员（各自有个管理员 ID）分配去专门管理一台服务器。
2.确认一台服务器是否是 DHCP 服务器。如果是的话，记下这台服务器管理的 IP 段（比如：192.168.0.21 到 192.168.0.254）。
3.确认一台服务器是不是文件服务器。如果是的话，要求可以为每个用户（以用户 ID 来区分）分配磁盘空间，也可以确认每个用户已分配的磁盘空间。
4.一台服务器可以同时是文件服务器跟 DHCP 服务器。

为了可以分配管理员服务器：

```
class Administrator {
    String adminId;
    Server serversAdminedByHim[];
}
class ServerConfigSystem {
    Server servers[];
    Administrator admins[];
}
```

为了支持 DHCP 服务器:

```
class DHCPConfig {
    InetAddress startIP;
    InetAddress endIP;
}
class ServerConfigSystem {
    Server servers[];
    Administrator admins[];
    HashMap serverToDHCPConfig;
}
```

为了支持文件服务器：

```
class FileServerConfig {
    HashMap userIdToQuota;
    int getQuotaForUser(String userId) {
        ...
    }
    void setQuotaForUser(String userId, int quota) {
        ...
    }
}
```

```
class ServerConfigSystem {
    Server servers[];
    Administrator admins[];
    HashMap serverToDHCPConfig;
    HashMap serverToFileServerConfig;
}
```

我们不须做别的事，服务器就可以同时作为 DHCP 跟文件服务器了。

13.目前某个系统的代码如下所示:

```
class Customer {
    String id;
    String name;
    String address;
}

class Supplier {
    String id;
    String name;
    String telNo;
    String address;
}

class SalesSystem {
    Vector customers;
    Vector suppliers;
}
```

在保持代码简洁的前提下，实现下面的新需求：

1.客户可以订货。每个订单有一个 ID 区分。订单要记录预订日期，客户的 ID，预订的每项商品的名字跟数量。

2.系统可以根据客户列出所有的订单。

3.系统可以根据供应商列出所有的订单。

4.供应商可以为它指定的一些客户打折（只对一些指定的商品有效）。不同的客户、不同的商品都可以有不可的折扣。

实现预订的功能：

```
class Order {
   String orderId;
   String customerId;
   String supplierId;
   Date orderDate;
```

```
    OrderLine orderLines;
}

class OrderLine {
    String productName;
    int quantity;
}

class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
}

class SalesSystem {
    Vector customers;
    Vector suppliers;
    Orders orders;
}
```

实现根据客户打印他的所有订单的功能：

```
class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
    void printOrdersByCustomer(String customerId) {
        ...
    }
}
```

实现根据供应商打印它有的所有订单的功能：

```
class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
    void printOrdersByCustomer(String customerId) {
        ...
    }
    void printOrdersForSupplier(String supplierId) {
```

```
            ...
        }
}
```

实现打折的功能：

```
class Discount {
    String supplierId;
    String customerId;
    String productName;
    double discountRate;
}

class Discounts {
    Discount discounts[];
    void addDiscount(
                String supplierId,
                String customerId,
                String productName,
                double discountRate) {
            ...
    }
    double findDiscount(
                String supplierId,
                String customerId,
                String productName){
            ...
    }
}

class SalesSystem {
    Vector customers;
    Vector suppliers;
    Orders orders;
    Discounts discounts;
}
```

第 5 章　慎用继承

示例

这是一个会议管理系统。用来管理各种各样的会议参与者信息。数据库里面有个表 Participants，里面的每条记录表示一个参会者。因为经常会发生用户误删掉某个参会者的信息。所以现在，用户删除时，并不会真的删除那参会者的信息，而只是将该记录的删除标记设为 true。24 小时以后，系统会自动将这条记录删除。但是在这 24 小时以内，如果用户改变主意了，系统还可以将这条记录还原，将删除标记设置为 false。

请认真的读下面的代码：

```
public class DBTable {
    protected Connection conn;
    protected tableName;
    public DBTable(String tableName) {
        this.tableName = tableName;
        this.conn = ...;
    }
    public void clear() {
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public int getCount() {
        PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM"+tableName);
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
}

public class ParticipantsInDB extends DBTable {
    public ParticipantsInDB() {
        super("participants");
    }
    public void addParticipant(Participant part) {
```

```
        ...
    }

    public void deleteParticipant(String participantId) {
        setDeleteFlag(participantId, true);
    }
    public void restoreParticipant(String participantId) {
        setDeleteFlag(participantId, false);
    }
    private void setDeleteFlag(String participantId, boolean b) {
        ...
    }
    public void reallyDelete() {
        PreparedStatement st = conn.prepareStatement(
                                "DELETE FROM "+
                                tableName+
                                " WHERE deleteFlag=true");
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public int countParticipants() {
        PreparedStatement st = conn.prepareStatement(
                                "SELECT COUNT(*) FROM "+
                                tableName+
                                " WHERE deleteFlag=false");
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
}
```

注意到，countParticipants 这个方法只计算那些 deleteFlags 为 false 的记录。也就是，被删除的那些参会者不被计算在内。

上面的代码看起来还不错，但却有一个很严重的问题。什么问题？先看看下面的代码：

```
ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
System.out.println("There are "+partsInDB.getCount()+ "participants");
```

最后一行代码，会打印出"There are 1 participants"这样信息，对不？错！它打印的是"There are 2 participants"！因为最后一行调用的是 DBTable 里面的这个方法 getCount,而不是 ParticipantsInDB 的 countParticipants。getCount 一点都不知道删除标记这回事，它只是简单的计算记录数量，并不知道要计算那些真正有效的参会者（就是删除标记为 false 的）。

继承了一些不合适（或者没用的）的功能

ParticipantsInDB 继承了来自 DBTable 的方法，比如 clear 和 getCount。对于 ParticipantsInDB 来讲，clear 这个方法的确是有用的：清空所有的参会者。但 getCount 就造成了一点点小意外了：通过 ParticipantsInDB 调用 getCount 这个方法时，是取得 participants 这个表里面所有的记录，不管删除标记是 true 还是 false 的。而实际上，没人想知道这个数据。即使有人想知道，这个方法也不应该叫做 getCount，因为这名字很容易就会跟“计算所有的（有效）参会者数量”联系在一起。

因此，ParticipantsInDB 是不是真的应该继承这个方法 getCount 呢？或者我们应该怎么做比较恰当呢？

它们之间是否真的有继承关系？

当我们继承了一些我们不想要的东西，我们应该再三的想想：它们之间是不是真的有继承关系？ParticipantsInDB 必须是一个 DBTable 吗？ParticipantsInDB 希不希望别人知道它是一个 DBTable？

实际上，ParticipantsInDB 描述的是系统中所有的参会者的集合，该系统可以是个单数据库的，也可以是多数据库的，也就是说，这个类可以代表一个数据库里的一个 Participants 表，也可以代表两个数据库各自的两个 Participants 表的总和。

如果还不清楚的话，我们就这样举例吧，比如，现在我们已经有了 2000 个参会者，在两个数据库中存放，其中数据库 A 的 participants 表里面存放了 1000 个参会者，数据库 B 的 participants 这个表存放了 1000 个参会者。DBTable 顶多只能描述一个数据库里面的一张表，也就是 1000 个参会者，而 participants 则可以完全的描述这 2000 年参会者的信息。前面可以当作数据库的数据表在系统中的代表，而后者表示的应该包含更多业务逻辑的一个域对象。（原谅这边我只能用域对象这样的词来断开这样的混淆。）

因此，我们可以判断，ParticipantsInDB 跟 DBTable 之间不应该有什么继承的关系。ParticipantsInDB 不能继承 DBTable 这个类了。于是，现在 ParticipantsInDB 也没有 getCount 这个方法了。可是 ParticipantsInDB 还需要 DBTable 类里面的其他方法啊，那怎么办？所以现在我们让 ParticipantsInDB 里面引用了一个 DBTable：

```
public class DBTable {
    private Connection conn;
```

```
    private String tableName;
    public DBTable(String tableName) {
        this.tableName = tableName;
        this.conn = ...;
    }
    public void clear() {
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public int getCount() {
        PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM "+tableName);
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
    public String getTableName() {
        return tableName;
    }
    public Connection getConn() {
        return conn;
    }
}

public class ParticipantsInDB {
    private DBTable table;
    public ParticipantsInDB() {
        table = new DBTable("participants");
    }
    public void addParticipant(Participant part) {
        ...
    }
    public void deleteParticipant(String participantId) {
        setDeleteFlag(participantId, true);
    }
    public void restoreParticipant(String participantId) {
```

```
        setDeleteFlag(participantId, false);
    }
    private void setDeleteFlag(String participantId, boolean b) {
        ...
    }
    public void reallyDelete() {
        PreparedStatement st = table.getConn().prepareStatement(
                                    "DELETE FROM "+
                                    table.getTableName()+
                                    " WHERE deleteFlag=true");
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public void clear() {
        table.clear();
    }
    public int countParticipants() {
        PreparedStatement st = table.getConn().prepareStatement(
                                    "SELECT COUNT(*) FROM "+
                                    table.getTableName()+
                                    " WHERE deleteFlag=false");
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
}
```

ParticipantsInDB 不再继承 DBTable。代替的，它里面有一个属性引用了一个 DBTable 对象，然后调用这个 DBTable 的 clear, getConn, getTableName 等等方法。

代理（delegation）

其实我们这边可以看一下 ParticipantsInDB 的 clear 方法，这个方法除了直接调用 DBTable 的 clear 方法以外，什么也没做。或者说，ParticipantsInDB 只是做为一个中间介让外界调用 DBTable 的方法，我们管这样传递调用的中间介叫“代理(delegation)”。

现在，之前有 bug 的那部分代码就编译不过了：

```
ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
//编译出错：因为在 ParticipantsInDB 里面已经没有 getCount 这个方法了！
System.out.println("There are "+partsInDB.getCount()+ "participants");
```

总结一下：首先，我们发现，ParticipantsInDB 和 DBTableIn 之间没有继承关系。然后我们就将“代理”来取代它们的继承。“代理”的优点就是，我们可以控制 DBTable 的哪些方法可以“公布（就是设为 public）”（比如 clear 方法）。如果我们用了继承的话，我们就没得选择，DBTable 里面的所有 public 方法都要对外公布！

抽取出父类中没必要的功能

现在，我们来看一下另一个例子。假定一个 Component 代表一个 GUI 对象，比如按钮或者文本框之类的。请认真阅读下面的代码：

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    int width;
    int height;
    ...
    abstract void paint(Graphics graphics);
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}

class Button extends Component {
    ActionListener listeners[];
    ...
    void paint(Graphics graphics) {
        ...
```

```
        }
    }

    class Container {
        Component components[];
        void add(Component component) {
            ...
        }
    }
```

假定你现在要写一个时钟 clock 组件。它是一个有时分针在转动的圆形的钟，每次更新时针跟分针的位置来显示当前的时间。因为这也是一个 GUI 组件，所以我们同样让它继承自 Component 类：

```
    class ClockComponent extends Component {
        ...
        void paint(Graphics graphics) {
            //根据时间绘制当前的钟表图形
        }
    }
```

现在我们有一个问题了：这个组件应该是个圆形的，但是它现在却继承了 Component 的 width 跟 height 属性，也继承了 setWidth 和 setHeight 这些方法。而这些东西对一个圆形的东西是没有意义的。

当我们让一个类继承另一个类时，我们需要再三的想想：它们之间是否有继承关系？ClockComponent 是一个 Component 吗？它跟其他的 Compoent（比如 Button）是一样的吗？

跟 ParticipantsInDB 的那个案例相反的是，我们不得不承认 ClockComponent 确实也是一个 Component，否则它就不能像其他的组件那样放在一个 Container 中。因此，我们只能让它继承 Component 类（而不是用“代理”）。

它既要继承 Component，又不要 width，height，setWidth 和 setHeight 这些，我们只好将这四样东西从 Component 里面拿走。而事实上，它也应该拿走。因为已经证明了，并不是所有的组件都需要这四样东西（至少 ClockComponent 不需要）。

如果一个父类描述的东西不是所有的子类共有的，那这个父类的设计肯定不是一个好的设计。

我们有充分的理由将这些移走。

只是，如果我们从 Component 移走了这四样东西，那原来的那些类，比如 Button 就没了这四样东西，而它确实又需要这些的（我们假定按钮是方形的）。

一个可行的方案是，创建一个 RectangularComponent 类，里面有 width，height，setWidth 和 setHeight 这四样。然后让 Button 继承自这个类：

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    ...
    abstract void paint(Graphics graphics);
}

abstract class RectangularComponent extends Component {
    int width;
    int height;
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}

class Button extends RectangularComponent {
    ActionListener listeners[];
    ...
    void paint(Graphics graphics) {
        ...
    }
}

class ClockComponent extends Component {
    ...
    void paint(Graphics graphics) {
        //根据时间绘制当前的钟表图形
    }
}
```

这并不是唯一可行的方法。另一个可行的方法是，创建一个 RectangularDimension，这个类持有这四个功能，然后让 Button 去代理这个类：

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    ...
    abstract void paint(Graphics graphics);
```

```
}

class RectangularDimension {
    int width;
    int height;
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}

class Button extends Component {
    ActionListener listeners[];
    RectangularDimension dim;
    ...
    void paint(Graphics graphics) {
        ...
    }
    void setWidth(int newWidth) {
        dim.setWidth(newWidth);
    }
    void setHeight(int newHeight) {
        dim.setHeight(newHeight);
    }
}

class ClockComponent extends Component {
    ...
    void paint(Graphics graphics) {
        //根据时间绘制当前的钟表图形
    }
}
```

总结

当我们想要让一个类继承自另一个类时，我们一定要再三的检查：子类会不会继承了一些它不需要的功能（属性或者方法）？如果是的话，我们就得认真再想想：它们之间有没有真正的继承关系？如果没有的话，就用代理。如果有的话，将这些不用的功能从基类转移到另外一个合适的地方去。

引述

里斯科夫替换原则(LSP)表述：**Subtype must be substitutable for their base types.** 子类应该能够代替父类的功能。或者直接点说，我们应该做到，将所有使用父类的地方改成使用子类后，对结果一点影响都没有。或者更直白一点吧，请尽量不要用重载，重载是个很坏很坏的主意！更多的信息可以去：

http://www.objectmentor.com/resources/articles/lsp.pdf.
http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple.

Design By Contract 是个跟 LSP 有关的东西。它表述说，我们应该测试我们所有的假设。更多信息：

http://c2.com/cgi/wiki?DesignByContract.

章节练习

介绍

有些问题可以测试之前几章的观点。

问题

1.下面的代码有重复。如果要你不惜代价的移除这些重复，你要怎么做？

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    GridListener listeners[];
    void appendRow() {
        //在表格的末尾加一行
        for (int i = 0; i < listeners.length; i++) {
            listeners[i].onRowAppended();
        }
    }
    void moveRow(int existingIdx, int newIdx) {
        //移动这行
```

```
            for (int i = 0; i < listeners.length; i++) {
                listeners[i].onRowMoved(existingIdx, newIdx);
            }
        }
    }
```

2.Java 中有个类叫"HashMap"。你可以用 put 的方法，将一对 key-value 对应的组合放进这样的 Map 里面。之后，你可以通过 get 的方法，然后传递一个参数 key，它会返回这个 key 对应的 value。key 跟 value 可以是任何类型。它还有一个 size 的方法，返回这个 Map 里面的所有组合的个数。请看下面的示例代码：

```
    HashMap m=new HashMap();
    m.put("x", new Integer(123));
    m.put("y", "hello");
    m.put(new Double(120.33), "hi");
    System.out.println(m.get("x"));    //打出 123
    System.out.println(m.get("y"));    //  打出 hello
    System.out.println(m.size());      //打出 3
```

请指出下面代码中的问题：

```
    public class CourseCatalog extends HashMap {
        public void addCourse(Course c) {
            put(c.getTitle(), c);
        }
        public Course findCourse(String title) {
            return (Course)get(title);
        }
        public int countCourses() {
            return size();
        }
    }
```

3.请找出并修正下面代码中的问题。你不可以用 Java 中的所有 colletion 类。

```
    public class Node {
        private Node nextNode;
        public Node getNextNode() {
            return nextNode;
        }
        public void setNextNode(Node nextNode) {
            this.nextNode = nextNode;
        }
    }
```

```
public class LinkList {
    private Node firstNode;
    public void addNode(Node newNode) {
        ...
    }
    public Node getFirstNode() {
        return firstNode;
    }
}

public class Employee extends Node {
    String employeeId;
    String name;
    ...
}

public class EmployeeList extends LinkList {
    public void addEmployee(Employee employee) {
        addNode(employee);
    }
    public Employee getFirstEmployee() {
        return (Employee)getFirstNode();
    }
    ...
}
```

4.假设一般来讲，一个老师可以教多个学生。但是一个研究生，只能由一个研究生导师教。请找出并修正下面代码中的问题：

```
class Student {
    String studentId;
    ...
}

class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
```

```
}

class GraduateStudent extends Student {
}

class GraduateTeacher extends Teacher {
}
```

5.请找出并修正下面代码中的问题：

```
public class Button {
    private Font labelFont;
    private String labelText;
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
    public void paint(Graphics graphics) {
        //在这个 graphics 中用 label 字体(labelFont)画出 label 文本(labelText)
    }
}
public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //在这个 graphics 中画出位图(bitmap)
    }
}
```

6.一个 property 文件是一个文本文件，不过它有一些特殊的格式。比如，它应该是这样的：

```
java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello
```

也就是说，property 的文件里面，每一行都要有一个 key(字符型)，然后一个“=”号，再一个值。

为了可以更方便的创建这样的文件，你写了下面的代码，用 Java 里面的 FileWriter 类，调用它的 write 方法：

```
class PropertyFileWriter extends FileWriter {
    PropertyFileWriter(File file) {
        super(file);
    }
    void writeEntry(String key, String value) {
```

```
            super.write(key+"="+value);
        }
    }
    class App {
        void makePropertyFile() {
            PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
            try {
                fw.writeEntry("conference.abc", "10");
                fw.writeEntry("xyz", "hello");
            } finally {
                fw.close();
            }
        }
    }
```

请找出并修正代码中的问题

7.找出一个错用了继承的代码，并修正这些问题。

提示

1.参考 RentalProcessor 这个例子。

2.你其实不想继承 HashMap 里面的 put 等方法的。

3.你不想继承 getNextNode 和 addNode 这些方法的。不要让 Employee 继承 Node 这个类。建一个新类如 EmployeeNode。然后让 EmployeeList 里面包含一个 LinkedList。当增加一个 Employee，创建一个 EmployeeNode 对象。

解决方法示例

1.下面的代码有重复。如果要你不惜代价的移除这些重复，你要怎么做？

```
    interface GridListener {
        void onRowAppended();
        void onRowMoved(int existingIdx, int newIdx);
    }
    class Grid {
        GridListener listeners[];
        void appendRow() {
```

```
            //在表格的末尾加一行
            for (int i = 0; i < listeners.length; i++) {
                listeners[i].onRowAppended();
            }
        }
        void moveRow(int existingIdx, int newIdx) {
            //移动这行
            for (int i = 0; i < listeners.length; i++) {
                listeners[i].onRowMoved(existingIdx, newIdx);
            }
        }
}
```

首先，让代码看起来一样：

```
interface GridListener {
        void onRowAppended();
        void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
        GridListener listeners[];
        void appendRow() {
            //在表格的末尾加一行
            for (int i = 0; i < listeners.length; i++) {
                通知 listeners[i];
            }
        }
        void moveRow(int existingIdx, int newIdx) {
            //移动这行
            for (int i = 0; i < listeners.length; i++) {
                通知 listeners[i];
            }
        }
}
```

然后将重复的代码抽取到一个方法里面：

```
interface GridListener {
        void onRowAppended();
        void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
        ...
        void appendRow() {
            //append a row at the end of the grid.
```

```
            ???();
        }
        void moveRow(int existingIdx, int newIdx) {
            //移动这行
            ???();
        }
        void ???() {
            for (int i = 0; i < listeners.length; i++) {
                通知 listeners[i];
            }
        }
    }
```

现在要为???这些方法取一个好名字，看看这些代码做了什么。在这个例子里面，它一个一个的通知了所有的 listeners。所以用“notifyListeners”这个名字：

```
    interface GridListener {
        void onRowAppended();
        void onRowMoved(int existingIdx, int newIdx);
    }
    class Grid {
        ...
        void appendRow() {
            //在表格的末尾加一行
            notifyListeners();
        }
        void moveRow(int existingIdx, int newIdx) {
            //移动这行
            notifyListeners();
        }
        void notifyListeners() {
            for (int i = 0; i < listeners.length; i++) {
                notify listeners[i];
            }
        }
    }
```

因为"通知 listeners[i]"具体实现起来有两种，所以我们建了一个接口，然后为两种实现创建两个实现类：

```
    interface GridListener {
        void onRowAppended();
        void onRowMoved(int existingIdx, int newIdx);
```

```
}

interface ??? {
    void notify(GridListener listener);
}

class Grid {
    ...
    void appendRow() {
        //在表格的末尾加一行
        notifyListeners();
    }
    void moveRow(int existingIdx, int newIdx) {
        //移动这行
        notifyListeners();
    }
    void notifyListeners(??? someObject) {
        for (int i = 0; i < listeners.length; i++) {
            someObject.notify(listeners[i]);
        }
    }
}
```

现在为这个接口取个名字，看看这接口做了什么事。因为它只有一个 notify 方法，而且只通知 GridListener，所以取作“GridListenerNotifier”：

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}

interface GridListenerNotifier {
    void notify(GridListener listener);
}

class Grid {
    ...
    void appendRow() {
        //在表格的末尾加一行
        notifyListeners();
    }
    void moveRow(int existingIdx, int newIdx) {
        //移动这行
```

```
            notifyListeners();
        }
        void notifyListeners(GridListenerNotifier notifier) {
            for (int i = 0; i < listeners.length; i++) {
                notifier.notify(listeners[i]);
            }
        }
}
```

为每种实现创建一个实现类。可以的话，做成内类：

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}

interface GridListenerNotifier {
    void notify(GridListener listener);
}

class Grid {
    ...
    void appendRow() {
        //在表格的末尾加一行
        notifyListeners(new GridListenerNotifier() {
            void notify(GridListener listener) {
                listener.onRowAppended();
            }
        });
    }

    void moveRow(final int existingIdx, final int newIdx) {
        //移动这行
        notifyListeners(new GridListenerNotifier() {
            void notify(GridListener listener) {
                listener.onRowMoved(existingIdx, newIdx);
            }
        });
    }
    void notifyListeners(GridListenerNotifier notifier) {
        for (int i = 0; i < listeners.length; i++) {
            notifier.notify(listeners[i]);
```

```
            }
        }
    }
```

2.Java 中有个类叫"HashMap"。你可以用 put 的方法，将一对 key-value 对应的组合放进这样的 Map 里面。之后，你可以通过 get 的方法，然后传递一个参数 key，它会返回这个 key 对应的 value。key 跟 value 可以是任何类型。它还有一个 size 的方法，返回这个 Map 里面的所有组合的个数。请看下面的示例代码：

```
HashMap m=new HashMap();
m.put("x", new Integer(123));
m.put("y", "hello");
m.put(new Double(120.33), "hi");
System.out.println(m.get("x"));    //打出 123
System.out.println(m.get("y"));    //  打出 hello
System.out.println(m.size());       //打出 3
```

请指出下面代码中的问题：

```
public class CourseCatalog extends HashMap {
    public void addCourse(Course c) {
        put(c.getTitle(), c);
    }
    public Course findCourse(String title) {
        return (Course)get(title);
    }
    public int countCourses() {
        return size();
    }
}
```

CourseCatalog 这个类并不想继承 put, get 和 size 这些方法。这些方法是它内部想访问的，它希望对外公布的是另外的方法。如果你直接调用了这些方法，就有问题，比如：

```
CourseCatalog courseCatalog = new CourseCatalog();
courseCatalog.put("Hello", "World");
courseCatalog.findCourse("Hello"); //错了: "World"是一个字符串，不是一个课程 Course
```

因为没有一个地方需要将 CourseCatalog 当作一个 HashMap 用，所以我们这边用代理代替继承：

```
public class CourseCatalog {
    HashMap map;
    public void addCourse(Course c) {
        map.put(c.getTitle(), c);
```

```
        }
        public Course findCourse(String title) {
            return (Course)map.get(title);
        }
        public int countCourses() {
            return map.size();
        }
    }
```

3.请找出并修正下面代码中的问题，你不可以用 Java 中的所有 colletion 类。

```
    public class Node {
        private Node nextNode;
        public Node getNextNode() {
            return nextNode;
        }
        public void setNextNode(Node nextNode) {
            this.nextNode = nextNode;
        }
    }

    public class LinkList {
        private Node firstNode;
        public void addNode(Node newNode) {
            ...
        }
        public Node getFirstNode() {
            return firstNode;
        }
    }

    public class Employee extends Node {
        String employeeId;
        String name;
        ...
    }

    public class EmployeeList extends LinkList {
        public void addEmployee(Employee employee) {
            addNode(employee);
        }
        public Employee getFirstEmployee() {
            return (Employee)getFirstNode();
```

```
        }
        ...
    }
```

EmployeeList 其实不想继承 addNode 这个方法。它只需要内部自己可以调用这个方法，但不想外部直接访问它。 否则，下面的代码就有问题了：

```
    EmployeeList employeeList = new EmployeeList();
    Node someNode = new Node();
    employeeList.addNode(someNode);
    employeeList.getFirstEmployee();//someNode 其实是一个 Node,但我们想要的是 Employee
```

Employee 也不想继承 getNextNode 这个方法。

因为没有代码需要将 EmployeeList 当做一个 List 用，所以我们应该用代理取代继承：

```
    public class Node {
        ...
    }

    public class LinkList {
        ...
    }

    public class Employee {
        String employeeId;
        String name;
        ...
    }

    public class EmployeeNode extends Node {
        Employee employee;
    }

    public class EmployeeList {
        LinkList list;
        public void addEmployee(Employee employee) {
            list.addNode(new EmployeeNode(employee));
        }
        public Employee getFirstEmployee() {
            return ((EmployeeNode)list.getFirstNode()).getEmployee();
        }
        ...
```

```
}
```

4.假设一般来讲，一个老师可以教多个学生。但是一个研究生，只能由一个研究生导师教。请找出并修正下面代码中的问题：

```
class Student {
    String studentId;
    ...
}

class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}

class GraduateStudent extends Student {
}

class GraduateTeacher extends Teacher {
}
```

目前，我们不可以让一个非研究生导师（简称硕导吧）教一个研究生。不过问题就在 Teacher 这个类里面的 addStudent 这个方法，这个方法，可以允许 Teacher 教任何学生。这样对吗？不对。因此这个方法不应该在这边。

```
class Student {
    String studentId;
    ...
}

class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
}
```

```
class GraduateStudent extends Student {
}

class GraduateTeacher extends Teacher {
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}
```

好，现在的问题是，Teacher 这个类里面没有 addStudent 这个方法了，可是我们还需要让一个非硕导教一个非研究生。为此，我们就新建了 NonGraduateTeacher 和 NonGraduateStudent 这两个类：

```
class Student {
    String studentId;
    ...
}

class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
}

class GraduateStudent extends Student {
}

class GraduateTeacher extends Teacher {
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}

class NonGraduateStudent extends Student {
}

class NonGraduateTeacher extends Teacher {
    public void addStudent(NonGraduateStudent student) {
        studentsTaught.add(student);
    }
}
```

5.请找出并修正下面代码中的问题：

```
public class Button {
    private Font labelFont;
    private String labelText;
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
    public void paint(Graphics graphics) {
        //在这个 graphics 中用 label 字体(labelFont)画出 label 文本(labelText)
    }
}
public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //在这个 graphics 中画出位图(bitmap)
    }
}
```

对于 BitmapButton 这个类来讲，父类 Button 里面的 labelFont 和 labelText 这两个属性没什么意义。因为并不是每个按钮都需要 font 或者 text，所以我们将这两个属性从 Button 类中移除。将它们放在一个子类 LabelButton 中。现在，我们要将 Button 里面的 paint 这个方法变为抽象的：

```
public abstract class Button {
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
    abstract public void paint(Graphics graphics);
}

public class LabelButton extends Button {
    private Font labelFont;
    private String labelText;
    public void paint(Graphics graphics) {
        //在这个 graphics 中用 label 字体(labelFont)画出 label 文本(labelText)
    }
}

public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //在这个 graphics 中画出位图(bitmap)
```

```
    }
}
```

6.一个 property 文件是一个文本文件，不过它有一些特殊的格式。比如，它应该是这样的：

```
java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello
```

也就是说，property 的文件里面，每一行都要有一个 key(字符型)，然后一个“=”号，再一个值。

为了可以更方便的创建这样的文件，你写了下面的代码，用 Java 里面的 FileWriter 类，调用它的 write 方法：

```
class PropertyFileWriter extends FileWriter {
    PropertyFileWriter(String path) {
        super(new File(path));
    }
    void writeEntry(String key, String value) {
        super.write(key+"="+value);
    }
}

class App {
    void makePropertyFile() {
        PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
        try {
            fw.writeEntry("conference.abc", "10");
            fw.writeEntry("xyz", "hello");
        } finally {
            fw.close();
        }
    }
}
```

请找出并修正代码中的问题

PropertyFileWrite 并不想继承 write 这个方法。国际惯例，它希望内部调用这个方法，不过不想公布给外面调用。否则，下面的代码就出问题了：

```
PropertyFileWriter propertyFileWriter = new PropertyFileWriter(...);
propertyFileWriter.write("这不是一个正确的条目行");
```

既然没人要将 PropertyFileWriter 当作 FileWriter 看，那我们就用代理取代继承吧：

```
class PropertyFileWriter {
    FileWriter fileWriter;
    PropertyFileWriter(String path) {
        fileWriter = new FileWriter(new File(path));
    }
    void writeEntry(String key, String value) {
        fileWriter.write(key+"="+value);
    }
    void close() {
        fileWriter.close();
    }
}

class App {
    void makePropertyFile() {
        PropertyFileWriter fw = new PropertyFileWriter("f1.properties");
        try {
            fw.writeEntry("conference.abc", "10");
            fw.writeEntry("xyz", "hello");
        } finally {
            fw.close();
        }
    }
}
```

第 6 章　处理不合适的依赖

如果现在有一个类 Parent，里面有个属性的类型是 Child，add 的方法里面还有个参数的类型是 Girl：

```
class Parent{
    Child child;
void add(Girl girl){
    ...
}
 }
```

因为上面 Parent 里面用到了 Child 跟 Girl 这两个类，我们就说，Parent 引用了类 Child 跟类 Girl。现在的问题是，如果 Child 这个类或者 Girl 这个类编译不过的话，那么 Parent 这个类也编译不了了。也就是说，Parent 依赖于 Child 跟 Girl。这章讲述的，就是因为一些类的依赖造成的无法重用的问题。

示例

这是一个处理 ZIP 的程序。用户可以在主窗口中先输入要生成的目标 zip 的路径，比如 c:\f.zip ，然后输入他想压缩到这个 zip 的源文件的路径，比如
c:\f2.doc 和 c:\f2.doc 。然后这个程序就会开始压缩 f1.doc 和 f2.doc，生成 f.zip 文件。在压缩各个源文件的时候，主窗口下的状态栏都要显示相关的信息。比如，在压缩 c:\f2.doc 的时候，状态栏就显示"正在压缩 c:\f2.zip"。

目前的代码就是

```
class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //根据 UI 上给 zipFilePath 和 srcFilePaths 赋值
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, this);
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
    }
}

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            f.setStatusBarText("Zipping "+srcFilePaths[i]);
        }
    }
}
```

我们还有一个存货管理系统，里面有一些程序的数据文件，经常需要压缩起来备份。这些源数据文件都有固定的路径，所以就不需要用户特地去输入路径了。现在我们想直接把上面的这个 ZipEngine 类拿过来重用。这个存货管理系统也有一个主窗口，同样在压缩待备份文件时，状态栏上面也要显示目前正在压缩的文件名称。

现在，问题来了。我们希望可以在不用修改代码的情况下直接重用 ZipEngine 这个类。但看了上面的代码以后我们发现：在调用 makeZip 这个方法时，还需要一个传递一个 ZipMainFrame 类型的参数进来。可是很明显我们现在的这个存货管理系统里面并没有 ZipMainFrame 这样的类。也就是说，现在 ZipEngine 这个类，在我们的这个存货管理系统中用不了了。

再往远一点想，好像其他的系统，一般也不会有 ZipMainFrame 这个类。即使类名一样的，里面所做的功能也不一样。那其他的系统也重用不了这个 ZipEngine 类了。

“不合适的依赖”，让代码很难被重用

因为 ZipEngine 引用了 ZipMainFrame 这个类，当我们想重用 ZipEngine 的时候，我们就需要将 ZipMainFrame 也加进来，调用 ZipEngine 的 makeZip 方法时，还要构造一个 ZipMainFrame 对象传给它。而在新的环境中，我们不可能有一个同样的 ZipMainFrame，也不可能特地为了调用这个方法，随便创建一个 ZipMainFrame 对象给它。

一般来说，如果一个类 A 引用了一个类 B，当我们想要重用 A 这个类时，我们就还得将 B 这个类也加进我们的系统。如果 B 引用了 C，那么 B 又将 C 也一起拉了进来。而如果 B 或者 C 在一个新的系统中没有意义，或者压根儿不应该存在的情况下，真正我们想要用的 A 这个类也用不了了。

因此，“不合适的依赖”让代码很难被重用。

为了可以重用 ZipEngine，首先，我们得让 ZipEngine 不再引用 ZipMainFrame。或者说，让 ZipEngine 不用依赖于 ZipMainFrame。

那怎么做呢？回答这个问题之前，我们先回答另一个问题：给你一段代码，你怎么判断这段代码是不是包含了“不合适的依赖”？“不合适”这个词定义的标准又是什么？

怎么判断是“不合适的依赖”

方法 1：

一个简单的方法就是：我们先看一下这段代码里面有没有一些互相循环的引用。比如，ZipMainFrame 引用了 ZipEngine 这个类，而 ZipEngine 又引用了 ZipMainFrame。我们管这样的类叫“互相依赖”。互相依赖也是一种代码异味，我们就认定这样的代码，是“不合适的依赖”。

这个方法很简单。不过，这种方法并不能包含全部情况，并不是所有有“不合适的依赖”的代码，都是这种互相依赖。

方法 2：

另一个方法比较主观：在检查代码的时候，我们问自己：对于它已经引用的这些类，是它真正需要引用的吗？对于 ZipEngine，它真的需要 ZipMainFrame 这个类吗？ZipEngine 只是改变 ZipMainFrame 的状态栏上的信息。是不是只有引用了 ZipMainFrame 才能满足这样的需求，其他类行不行？有没有一个类可以取代 ZipMainFrame 呢？

而实际上，ZipEngine 并不是一定要引用 ZipMainFrame 的。它想引用的，其实只是一个可以显示信息的状态栏而已。
因此，我们就将代码改为：

```
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], StatusBar statusBar) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            statusbar.setText("Zipping "+srcFilePaths[i]);
        }
    }
}
```

现在，ZipEngine 只是引用了 StatusBar，而不再是 ZipMainFrame 了。可是这样好吗？相对好一些！因为 StatusBar 比较通用（至少有 StatusBar 这个类的系统比 ZipMainFrame 多多了），这样的话，ZipEngine 这个类的可重用性就大幅改观了。

不过，这样的方法还是太主观了。没有一个既定的标准，可以判断 ZipEngine 到底需要的是什么样的东西。比如，我们就说，ZipEngine 其实想要的也不是一个状态栏，它只是想调用一个可以显示一些信息的接口而已(而不是一个状态栏这么大的一个对象)。

方法 3：

第 3 种方法也很主观：在设计类的时候，我们先预测一个以后可能会重用这个类的系统。然后再判断，在那样的系统中，这个类能不能被重用？如果你自己都觉得以后的系统不能重用这个类的话，你就断定，这个类包含“不合适的依赖”了。

比如，我们在设计完 ZipEngine 这个类时，我们就想一下，这个类能在别的系统重用吗？可是好像别的系统，不会有 ZipMainFrame 这个类，至少一个没有 GUI 的系统会有这样的类！这样的话，那它就不应该引用 ZipMainFrame 这个类了。这个方法其实也很主观，不怎么实用。每个人预测的可能性都不一样。

方法 4

第 4 个方法比较简单而且客观了。当我们想在一个新系统中重用这个类，却发现重用不了时，我们就判断，这个类包含了“不合适的依赖”。比如，我们在存货管理系统中，要重用 ZipEngine 的时候，我们才发现，这个类重用不了。这时我们就认定，这个类有“不合适的依赖”。

后一种方法是个“懒惰而被动的”方法，因为我们真正想在具体的项目中重用的时候，才能判断出来。不过这也是个很有效的方法。

总结

要判断一个代码是不是包含了“不合适的依赖”，共有四个方法：

1.看代码：有没有互相依赖？

2.认真想想：它真正需要的是什么？

3.推测一下：它在以后的系统中可以重用吗？

4.到要重用的时候就知道了：现在我要重用这个类，能不能重用？

方法 1 跟 4 是最简单的方法，推荐初学者可以这样来判断。有更多的设计经验了，再用方法 2 跟 3 会好一些。

怎么让 ZipEngine 不再引用（依赖于）ZipMainFrame

现在我们来看看，怎么让 ZipEngine 不再引用 ZipMainFrame。其实，在介绍方法 2 的时候，我们就已经通过思考发现，ZipEngine 这个类真正需要的是什么，也找出了解决办法。不过因为方法 2 相对来讲并不是那么简单就可以用好的，所以我们先假装不知道方法 2 的结果。

我们用方法 4。现在我们是在做一个文字模式的系统（没有状态栏了，我们只能直接在没有图形的屏幕上显示这些信息），发现 ZipEngine 不能重用了。怎么办？

因为我们不能重用 ZipEngine，我们只好先将它的代码复制粘贴出来，然后再修改成下面的代码：

```
class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths);
    }
}
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[]) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            System.out.println("Zipping "+srcFilePaths[i]);
```

```
        }
    }
}
```

再看一下原来的代码是：

```
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            f.setStatusBarText("Zipping "+srcFilePaths[i]);
        }
    }
}
```

很明显，这里面有很多的重复代码（代码异味）。要消除这样的代码异味，我们就先用伪码让这两段代码看起来一样。比如，改成：

```
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[]) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            显示信息。。。
        }
    }
}
```

因为“显示信息”具体出来，有两种实现，所以我们现在创建一个接口，里面有一个方法用来显示信息。这个方法可以直接取名为“showMessage”，而根据这个接口做的事，我们也可以直接将接口名取为“MessageDisplay”或者“MessageSink”之类的：

```
interface MessageDisplay {
    void showMessage(String msg);
}
```

将 ZipEngine 改为：

```
class ZipEngine {
```

```
    void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
        }
    }
}
```

而 MessageDisplay 这个接口的两个实现类就是：

```
class ZipMainFrameMessageDisplay implements MessageDisplay {
    ZipMainFrame f;
    ZipMainFrameMessageDisplay(ZipMainFrame f) {
        this.f = f;
    }
    void showMessage(String msg) {
        f.setStatusBarText(msg);
    }
}

class SystemOutMessageDisplay implements MessageDisplay {
    void showMessage(String msg) {
        System.out.println(msg);
    }
}
```

现在两个系统也相应的做了修改：

```
class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //根据 UI 上给 zipFilePath 和 srcFilePaths 赋值
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new ZipMainFrameMessageDisplay(this));
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
```

```
    }
}

class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new SystemOutMessageDisplay());
    }
}
```

改进后的代码

下面就是改进完的代码。为了让代码看起来清楚一些，我们用了 Java 的内类：

```
interface MessageDisplay {
    void showMessage(String msg);
}

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
        //在该路径上创建 zip 文件
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //将 srcFilePaths[i]的文件加到压缩包中
            ...
            msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
        }
    }
}

class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //根据 UI 上给 zipFilePath 和 srcFilePaths 赋值
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
            void showMessage(String msg) {
```

```
                setStatusBarText(msg);
            }
        });
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
    }
}

class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
            void showMessage(String msg) {
                System.out.println(msg);
            }
        });
    }
}
```

引述

依赖反转原则（Dependency Inversion Principle ）表述：抽象不应该依赖于具体，高层的比较抽象的类不应该依赖于低层的比较具体的类。当这种问题出现的时候，我们应该抽取出更抽象的一个概念，然后让这两个类依赖于这个抽取出来的概念。更多的信息，可以看：

http://www.objectmentor.com/resources/articles/dip.pdf
http://c2.com/cgi/wiki?DependencyInversionPrinciple

章节练习

介绍

有些问题可以测试前面几章讲的观点：

问题

1.指出下面代码中的问题。接着，假设你想把 FileCopier 这个类重用在一个文字模式的文件复制系统中，该系统会在文字控制台中显示这些信息。你要怎么做？

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(this);
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));                    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}
class FileCopier {
    MainApp app;
    FileCopier(MainApp app) {
        this.app = app;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //从 source 中读出 n(<=512)个字节
            //将这 n 个字节写到目标文件里
            i += n;
            app.updateProgressBar(i, sizeOfSource);
        }
    }
}
```

2.在上面的例子中，假设你想要开发另外一个文本复制系统，进度每增加 10%，你就打印出一个*。你怎么做？

3.指出下面代码中的问题. 接着假设你想在另一个系统中重用 FaxMachine 这个类。你怎么做？

```
class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(this);
        faxMachine.sendFax("783675", "hello");
    }
}
class FaxMachine {
    MainApp app;
    FaxMachine(MainApp app) {
        this.app = app;
```

```
        }
        void sendFax(String toFaxNo, String msg) {
            FaxMachineHardware hardware = ...;
            hardware.setStationId(app.getFaxNo());
            hardware.setRecipientFaxNo(toFaxNo);
            hardware.start();
            try {
                do {
                    Graphics graphics = hardware.newPage();
                    //将 msg 画到 graphics 里面
                } while (more page is needed);
            } finally {
                hardware.done();
            }
        }
    }
```

4.指出下面代码中的问题. 接着你想在另一个系统中重用 HeatSensor 这个类。怎么做?

```
    class Cooker {
        HeatSensor heatSensor;
        Speaker speaker;
        void alarm() {
            speaker.setFrequency(Speaker.HIGH_FREQUENCY);
            speaker.turnOn();
        }
    }

    class HeatSensor {
        Cooker cooker;
        HeatSensor(Cooker cooker) {
            this.cooker = cooker;
        }
        void checkOverHeated() {
            if (isOverHeated()) {
                cooker.alarm();
            }
        }
        boolean isOverHeated() {
            ...
        }
    }
```

5.这是一个文字处理系统。它可以让用户选择字体。用户确定要改变字体之前，系统会让用户看预览一个效果。下面是目前的代码。找出代码中的异味。如果我们想在一个 GUI 程序中重用 ChooseFontDialog 这个类，不过那个 GUI 程序不支持预览功能。怎么修改代码比较合适？

```
class WordProcessorMainFrame extends JFrame {
    void onChangeFont() {
        ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
        if (chooseFontDialog.showOnScreen()) {
            Font newFont = chooseFontDialog.getSelectedFont();
            //用新字体显示内容
        } else {
            //用原来的字体显示内容
        }
    }
    void previewWithFont(Font font) {
        //预览用该字体显示的内容
    }
}

class ChooseFontDialog extends JDialog {
    WordProcessorMainFrame mainFrame;
    Font selectedFont;
    ChooseFontDialog(WordProcessorMainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }
    boolean showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        mainFrame.previewWithFont(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```

6.找出一些包含不合适引用的代码，修正那些问题。

提示

1.互相依赖了。

2.没有提示。

3.互相依赖。不需要创建接口跟实现类。新旧系统中的不同点，用不同的对象就可以表现出来，不需要用到不同的类。

解决方法示例

1.指出下面代码中的问题指出下面代码中的问题。接着，假设你想把 FileCopier 这个类重用在一个文字模式的文件复制系统中，该系统会在文字控制台中显示这些信息。你要怎么做？

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(this);
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}
class FileCopier {
    MainApp app;
    FileCopier(MainApp app) {
        this.app = app;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //从 source 中读出 n(<=512)个字节
            //将这 n 个字节写到目标文件里
            i += n;
            app.updateProgressBar(i, sizeOfSource);
        }
    }
}
```

现在 FileCopier 依赖于 MainApp 这个类，造成不能在这个文字模式的系统中重用。我们现在先建一个接口，

然后让 FileCopier 中引用这个接口，这样的话 FileCopier 的代码在这两个系统都一样：

```
interface CopyMonitor {
    void updateProgress(int noBytesCopied, int sizeOfSource);
}
class FileCopier {
    CopyMonitor copyMonitor;
    FileCopier(CopyMonitor copyMonitor) {
        this.copyMonitor = copyMonitor;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //从 source 中读出 n(<=512)个字节
            //将这 n 个字节写到目标文件里
            i += n;
            copyMonitor.updateProgress(i, sizeOfSource);
        }
    }
}
```

现在这两个系统都要有一个 CopyMonitor 接口的实现类：

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int noBytesCopied, int sizeOfSource) {
                updateProgressBar(noBytesCopied, sizeOfSource);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}

class TextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int noBytesCopied, int sizeOfSource) {
                System.out.println(noBytesCopied*100/sizeOfSource);
```

```
                }
            });
            fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
        }
    }
```

因为两个实现类都要计算百分点，我们就直接在实现类的外面将百分点传递进来：

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int completionPercentage) {
                updateProgressBar(completionPercentage);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int percentage) {
        progressBar.setPercentage(percentage);
    }
}

class TextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int completionPercentage) {
                System.out.println(completionPercentage);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
}

interface CopyMonitor {
    void updateProgress(int completionPercentage);
}

class FileCopier {
    CopyMonitor copyMonitor;
    FileCopier(CopyMonitor copyMonitor) {
        this.copyMonitor = copyMonitor;
    }
```

```
        void copyFile(File source, File target) {
            int sizeOfSource = (int)source.length();
            for (int i = 0; i < sizeOfSource; ) {
                //从 source 中读出 n(<=512)个字节
                //将这 n 个字节写到目标文件里
                i += n;
                copyMonitor.updateProgress(i*100/sizeOfSource);
            }
        }
    }
```

2.在上面的例子中，假设你想要开发另外一个文本复制系统，进度每增加 10%，你就打印出一个*。你怎么做？

Just create another class to implement CopyMonitor:

```
class AnotherTextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            int noStarsPrinted = 0;
            void updateProgress(int completionPercentage) {
                int noStarsToPrint = completionPercentage/10;
                while (noStarsPrinted<noStarsToPrint) {
                    System.out.println("*");
                    noStarsToPrint++;
                }
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
}
```

3.指出下面代码中的问题. 接着假设你想在另一个系统中重用 FaxMachine 这个类。你怎么做？

```
class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(this);
        faxMachine.sendFax("783675", "hello");
    }
}

class FaxMachine {
    MainApp app;
    FaxMachine(MainApp app) {
```

```
            this.app = app;
        }
        void sendFax(String toFaxNo, String msg) {
            FaxMachineHardware hardware = ...;
            hardware.setStationId(app.getFaxNo());
            hardware.setRecipientFaxNo(toFaxNo);
            hardware.start();
            try {
                do {
                    Graphics graphics = hardware.newPage();
                    //将 msg 画到 graphics 里面
                } while (还有页面没打完);
            } finally {
                hardware.done();
            }
        }
    }
```

目前 FaxMachine 依赖于 MainApp，所以在其他系统不能被重用。其实，FixMachine 真正需要的，是要取得传真号码做为接收点的 id。要解决这个问题，我们只需将传真号码传给 FaxMachine，而不是 MainApp：

```
class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(faxNo);
        faxMachine.sendFax("783675", "hello");
    }
}

class FaxMachine {
    String stationId;
    FaxMachine(String stationId) {
        this.stationId = stationId;
    }
    void sendFax(String toFaxNo, String msg) {
        FaxMachineHardware hardware = ...;
        hardware.setStationId(stationId);
        hardware.setRecipientFaxNo(toFaxNo);
        hardware.start();
        try {
            do {
                Graphics graphics = hardware.newPage();
                //将 msg 画到 graphics 里面
```

```
            } while (还有页面没打完);
        } finally {
            hardware.done();
        }
    }
}
```

4.指出下面代码中的问题. 接着你想在另一个系统中重用 HeatSensor 这个类。怎么做?

```
class Cooker {
    HeatSensor heatSensor;
    Speaker speaker;
    void alarm() {
        speaker.setFrequency(Speaker.HIGH_FREQUENCY);
        speaker.turnOn();
    }
}

class HeatSensor {
    Cooker cooker;
    HeatSensor(Cooker cooker) {
        this.cooker = cooker;
    }
    void checkOverHeated() {
        if (isOverHeated()) {
            cooker.alarm();
        }
    }
    boolean isOverHeated() {
        ...
    }
}
```

目前 HeatSensor 依赖于 Cooker，导致它不能在其他系统中被重用。其实，HeatSenor 真正想要的，只是一个可以发出警报的服务，而不是整个 Cooker。要解决这样的问题，我们只需要将这个报警器传进来，而不是整个 Cooker：

```
interface Alarm {
    void turnOn();
}

class Cooker {
    HeatSensor heatSensor;
```

```
        Speaker speaker;
        Alarm getAlarm() {
            return new Alarm() {
                void turnOn() {
                    speaker.setFrequency(Speaker.HIGH_FREQUENCY);
                    speaker.turnOn();
                }
            };
        }
    }

    class HeatSensor {
        Alarm alarm;
        HeatSensor(Alarm alarm) {
            this.alarm = alarm;
        }
        void checkOverHeated() {
            if (isOverHeated()) {
                alarm.turnOn();
            }
        }
        ...
    }
```

5.这是一个文字处理系统。它可以让用户选择字体。用户确定要改变字体之前，系统会让用户看预览一个效果。下面是目前的代码。找出代码中的异味。如果我们想在一个 GUI 程序中重用 ChooseFontDialog 这个类，不过那个 GUI 程序不支持预览功能。怎么修改代码比较合适？

```
    class WordProcessorMainFrame extends JFrame {
        void onChangeFont() {
            ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
            if (chooseFontDialog.showOnScreen()) {
                Font newFont = chooseFontDialog.getSelectedFont();
                //用新字体显示内容
            } else {
                //用原来的字体显示内容
            }
        }
        void previewWithFont(Font font) {
            //预览用该字体显示的内容
        }
    }
```

```
class ChooseFontDialog extends JDialog {
    WordProcessorMainFrame mainFrame;
    Font selectedFont;
    ChooseFontDialog(WordProcessorMainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }
    booleawn showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        mainFrame.previewWithFont(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```

这里面的代码异味就是 WordProcessorMainFrame 和 ChooseFontDialog 之间的互相依赖。为了让没有预览功能的 GUI 程序可以使用 ChooseFontDialog 这个类，我们首先，将 ChooseFontDialog 对 WordProcessorMainFrame 的依赖解开。

这里，ChooseFontDialog 真正需要的，只是想通知另外一个对象说字体改变了，这样那个被通知的对象就可以让用户预览字体的效果了。

```
interface FontChangeListener {
    void onFontChanged(Font newFont);
}

class WordProcessorMainFrame extends JFrame {
    void onChangeFont() {
        ChooseFontDialog chooseFontDialog = new ChooseFontDialog(
            new FontChangeListener() {
                void onFontChanged(Font newFont) {
                    previewWithFont(newFont);
                }
            });
        if (chooseFontDialog.showOnScreen()) {
            Font newFont = chooseFontDialog.getSelectedFont();
            //用新字体显示内容
        } else {
```

```
            //用原来的字体显示内容
        }
    }
    void previewWithFont(Font font) {
        //预览用该字体显示的内容
    }
}

class ChooseFontDialog extends JDialog {
    FontChangeListener fontChangeListener;
    Font selectedFont;
    ChooseFontDialog(FontChangeListener fontChangeListener) {
        this.fontChangeListener = fontChangeListener;
    }
    boolean showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        fontChangeListener.onFontChanged(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```

第 7 章　将数据库访问，UI 和域逻辑分离

（这里面的域逻辑，原文是叫 Domain logic，我本想用业务逻辑层来说明的，可是后面又有这句话，"Domain logic is also called "domain model" or "business logic"."，即"域逻辑又称为域模型或者业务逻辑"，所以我们还是老老实实叫它域逻辑层吧）。

示例

这是一个会议管理系统。该系统会记录每个参会者的 ID，姓名，电话，和所属地区。参会者 ID 是会议组织者分配的一个唯一的数字标识。参会者的姓名是必须要提供的。电话则不是必填的。所有的参会者只能来自三个地域，中国，美国或者欧洲。

我们现在在数据库创建一个表来存储参会者的这些信息。现在表结构是这样的：

```
create table Participants (
    id int primary key,
    name varchar(20) not null,
    telNo varchar(20),
    region varchar(20)
);
```

为了可以让系统的使用者可以增加新参会者，我们又写了下面这样的代码。系统会自动从已有的所有参会者中找出最大的 ID，加 1 作为新的参会者的默认 ID，显示在界面的输入框上。用户可以直接用这个 ID，也可以自己再输入一个特定的 ID。请认真读下面代码：

```
class AddParticipantDialog extends JDialog {
    Connection dbConn;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog() {
        setupComponents();
        dbConn = ...;
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        int nextId;
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                nextId = rs.getInt(1)+1;
            } finally {
                rs.close();
            }
        } finally {
```

```
            st.close();
        }
        id.setText(new Integer(nextId).toString());
        name.setText("");
        region.setText("中国");
    }
    void onOK() {
        if (name.getText().equals("")) {
            JOptionPane.showMessageDialog(this, "名称不能为空");
            return;
        }
        if (!region.equals("中国") &&
            !region.equals("美国") &&
            !region.equals("欧洲")) {
            JOptionPane.showMessageDialog(this, "Region is unknown");
            return;
        }
        PreparedStatement st =
            dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
        try {
            st.setInt(1, Integer.parseInt(id.getText()));
            st.setString(2, name.getText());
            st.setString(3, telNo.getText());
            st.setString(4, region.getText());
            st.executeUpdate();
        } finally {
            st.close();
        }
        dispose();
    }
}
```

这段代码看起来还正常吧？但是这里面将下面三个方面的代码都混在了一起：

1.UI: JDialog, JTextField, 响应用户事件的代码。

2.数据库访问: Connection, PreparedStatement, SQL statements, ResultSet 等等。

3.域逻辑: 新参会者的默认 id，参会者的名字是必填的，所属地区的限制等等。域逻辑又称为“域模型”或者“业务逻辑”。

层次混乱造成的问题

这三方面的代码混在一起，会造成下面的问题：

1.代码很复杂。

2.代码很难重用。如果我们想创建一个 EditParticipantDialog，让用户更改参会者的信息，我们就想重用部分域逻辑（比如，地区的限制）。但实现这部分域逻辑的代码跟 AddParticipantDialog 混在了一起，根本不能重用。如果是在一个 web 系统中，就更难重用了。

3.代码很难测试。每次要测这样的一段代码，我们都要建一个数据库，还要通过一个用户操作界面来测试。

4.如果数据库表结构更改了，AddParticipantDialog 这个类，还有其他的很多地方都要跟着更改。

5.它导致我们一直在考虑一些低层的太细节的概念，比如数据库字段，表的记录之类的，而不是类，对象，方法和属性这一类的概念。或者说白了一点，一直在考虑怎么往数据库里面装数据，而没有了面向对象的概念，没有了建立业务模型的思维。

因此，我们应该将这三种类别的代码分离开（UI，数据库访问，域逻辑）。

先抽取出访问数据库的代码

我们先抽取出访问数据库的代码。我们先将数据库中所有的参会者当作一个集合体，这个集合体里面没有重复的元素，元素的排列也没有顺序。这个集合体支持增加，删除，更新和罗列的操作，然后我们将它命名为 Participants：

```
class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
}

interface ParticipantIterator {
    boolean next();
    Participant getParticipant();
}

class Participants {
    Connection dbConn;
    Participants() {
        dbConn = ...;
    }
    void addParticipant(Participant part) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
```

```
            try {
                st.setInt(1, part.getId());
                st.setString(2, part.getName());
                st.setString(3, part.getTelNo());
                st.setString(4, part.getRegion());
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
        void removeParticipant(int partId) {
            ...
        }
        void updateParticipant(Participant part) {
            ...
        }
        ParticipantIterator getAllParticipantsById() {
            ...
        }
        ParticipantIterator getParticipantsWithNameById(String name) {
            ...
        }
    }
```

在 AddParticipantDialog 中，我们还需要一个找出当前最大的参会者 ID 的功能。因此，我们还要在这个集合体中定义一个 getMaxId 的方法：

```
    class Participants {
        Connection dbConn;
        void addParticipant(Participant part) {
            ...
        }
        int getMaxId() {
            PreparedStatement st =
                dbConn.prepareStatement("select max(id) from participants");
            try {
                ResultSet rs = st.executeQuery();
                try {
                    rs.next();
                    return rs.getInt(1);
                } finally {
                    rs.close();
                }
```

```
            } finally {
                st.close();
            }
        }
        ...
    }
```

现在，AddParticipantDialog 这个类就可以简化为：

```
class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        int nextId = participants.getMaxId()+1;
        id.setText(new Integer(nextId).toString());
        name.setText("");
        telNo.setText("");
        region.setText("中国");
    }
    void onOK() {
        if (name.getText().equals("")) {
            JOptionPane.showMessageDialog(this, "名称不能为空");
            return;
        }
        if (!region.equals("中国") &&
            !region.equals("美国") &&
            !region.equals("欧洲")) {
            JOptionPane.showMessageDialog(this, "Region is unknown");
            return;
```

```
            }
            Participant part = new Participant(
                                    Integer.parseInt(id.getText()),
                                    name.getText(),
                                    telNo.getText(),
                                    region.getText());
            participants.addParticipant(part);
            dispose();
        }
    }
```

现在，AddParticipantDialog 这个类已经简单多了。从这里面，我们看不到任何跟数据库有关的东西存在了！

抽取出访问数据库代码后得到的灵活性

因为 AddParticipantDialog 这个类现在操作的，是一个参会者的集合体 Participants，而不再是数据库中的表。就算现在我们想把这些参会者的信息存储在一个 XML 文件或者一个简单的文本文件中也可以了。我们只需要修改 Participants 这个类里面每个方法的具体实现就行了，不用修改 AddParticipantDialog：

```
class Participants {
    void addParticipant(Participant part) {
        //将参会者信息存在 XML 文件里
    }
    void getMaxId() {
        //从 XML 文件中找出最大 id
    }
}
```

甚至，如果我们同时要用数据库存储和 XML 文件存储怎么办？我们就将 Participants 这个类变成一个接口，然后在一个实现类中用数据库实现这个接口，另一个实现类中用 XML 文件：

```
interface Participants {
    void addParticipant(Participant part);
    int getMaxId();
    ...
}
class ParticipantsInDB implements Participants {
    void addParticipant(Participant part) {
        ...
    }
    int getMaxId() {
```

```
            ...
        }
    }

    class ParticipantsInXMLFile implements Participants {
        void addParticipant(Participant part) {
            //将参会者信息存在 XML 文件里
        }
        int getMaxId() {
            //从 XML 文件中找出最大 id
        }
    }
    class AddParticipantDialog extends JDialog {
        AddParticipantDialog(Participants participants) {
            ...
        }
        ...
    }
```

将域逻辑跟 UI 分离

现在，我们将分离域逻辑跟 UI：

```
class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
    static Participant makeDefaultParticipant() {
        return new Participant(0, "", "", "中国");
    }
    void assertValid() throws ParticipantException {
        if (name.equals("")) {
            throw new ParticipantException("名称不能为空");
        }
        if (!region.equals("中国") &&
            !region.equals("美国") &&
            !region.equals("欧洲")) {
            throw new ParticipantException("Region is unknown");
```

```
            }
        }
    }
class ParticipantException extends Exception {
    ParticipantException(String msg) {
        super(msg);
    }
}
```

现在，AddParticipantDialog 可以简化为：

```
class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        newPart.setId(participants.getMaxId()+1);
        showParticipant(newPart);
    }
    void showParticipant(Participant part) {
        id.setText(new Integer(part.getId()).toString());
        name.setText(part.getName());
        telNo.setText(part.getTelNo());
        region.setText(part.getRegion());
    }
    Participant makeParticipant() throws ParticipantException {
        Participant part = new Participant(
                        Integer.parseInt(id.getText()),
                        name.getText(),
```

```
                    telNo.getText(),
                    region.getText());
        part.assertValid();
        return part;
    }
    void onOK() {
        try {
            participants.addParticipant(makeParticipant());
            dispose();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, e.getMessage());
        }
    }
}
```

现在，AddParticipantDialog 这个类更简单了。准确的讲，这个类基本上要做的事情就是：通过多个不同的 UI 组件显示一个参会者对象的信息（showParticipant），根据 UI 组件上面的内容生成一个参会者对象（makeParticipant）。就这样，无他。

改进后的代码

下面看一下改进完的代码：

```
class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
    static Participant makeDefaultParticipant() {
        return new Participant(0, "", "", "中国");
    }
    void assertValid() throws ParticipantException {
        if (name.equals("")) {
            throw new ParticipantException("名称不能为空");
        }
        if (!region.equals("中国") &&
            !region.equals("美国") &&
            !region.equals("欧洲")) {
            throw new ParticipantException("Region is unknown");
        }
    }
}
```

```
 class ParticipantException extends Exception {
     ParticipantException(String msg) {
         super(msg);
     }
 }

 class Participants {
     Connection dbConn;
     Participants() {
         dbConn = ...;
     }
     void addParticipant(Participant part) {
         PreparedStatement st =
             dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
         try {
             st.setInt(1, part.getId());
             st.setString(2, part.getName());
             st.setString(3, part.getTelNo());
             st.setString(4, part.getRegion());
             st.executeUpdate();
         } finally {
            st.close();
         }
     }
     int getMaxId() {
         PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
         try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return rs.getInt(1);
            } finally {
                rs.close();
            }
         } finally {
            st.close();
         }
     }
}

class AddParticipantDialog extends JDialog {
     Participants participants;
```

```
JTextField id;
JTextField name;
JTextField telNo;
JTextField region;
AddParticipantDialog(Participants participants) {
    this.participants = participants;
    setupComponents();
}
void setupComponents() {
    ...
}
void show() {
    showDefaultValues();
    setVisible(true);
}
void showDefaultValues() {
    Participant newPart = Participant.makeDefaultParticipant();
    newPart.setId(participants.getMaxId()+1);
    showParticipant(newPart);
}
void showParticipant(Participant part) {
    id.setText(new Integer(part.getId()).toString());
    name.setText(part.getName());
    telNo.setText(part.getTelNo());
    region.setText(part.getRegion());
}
Participant makeParticipant() throws ParticipantException {
    Participant part = new Participant(
                        Integer.parseInt(id.getText()),
                        name.getText(),
                        telNo.getText(),
                        region.getText());
    part.assertValid();
    return part;
}
void onOK() {
    try {
        participants.addParticipant(makeParticipant());
        dispose();
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this, e.getMessage());
    }
}
```

```
}
```

SQLException 的陷阱

事实上，代码里面还有一个问题。我们先认真的看一下 Participants 这个类：

```
class Participants {
    ...
    int getMaxId() {
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return rs.getInt(1);
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
}
```

每当我们调用 prepareStatement, executeQuery, next 和 close 这样的方法，JDBC 都会抛出 SQLException 这个异常。但是 getMaxId 这个方法本身并不能处理这个异常，我们只好将它抛出去，让外部调用 getMaxId 这个方法的地方处理。所以，我们要将代码改成：

```
class Participants {
    ...
    void getMaxId() throws SQLException {
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return st.getInt(1);
            } finally {
                rs.close();
            }
        } finally {
```

```
                st.close();
            }
        }
    }
```

问题就在这边：每次 AddParticipantDialog 里面调用 getMaxId 这个方法，它都要处理这个异常（要嘛 catch 住自己处理，要嘛继续向上抛）：

```
class AddParticipantDialog extends JDialog {
    ...
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        try {
            newPart.setId(participants.getMaxId()+1);
        } catch (SQLException e) {
            ...
        }
        showParticipant(newPart);
    }
}
```

就因为 AddParticipantDialog 现在要处理 SQLException 这样的异常了，我们一眼就可以看出这个类又跟数据库藕合，它不再是单纯操作参与者的集合体或者一个 XML 文件了。也就是说，SQLException 强制性的让我们的 AddParticipantDialog 陷入了调用数据库的坑中。如果别的环境中不用数据库存储的话，那这段客户端代码就无法重用，即使很勉强的重用了，就变成很糟糕的代码了！

为了解决这样的问题，当 Participants 这个类碰到一个数据库的异常时，它不应该直接将这个异常抛出去，而是要抛出一个跟 participants 这个集体合自己有关的异常：

```
class ParticipantsException extends Exception {
    ParticipantsException(Throwable cause) {
        super(cause);
    }
}

class Participants {
    ...
    int getMaxId() throws ParticipantsException {
        try {
            PreparedStatement st =
                dbConn.prepareStatement("select max(id) from participants");
            try {
                ResultSet rs = st.executeQuery();
                try {
```

```
                    rs.next();
                    return rs.getInt(1);
                } finally {
                    rs.close();
                }
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            //抛出跟 Participants 有关的异常。
            throw new ParticipantsException(e);
        }
    }
}
```

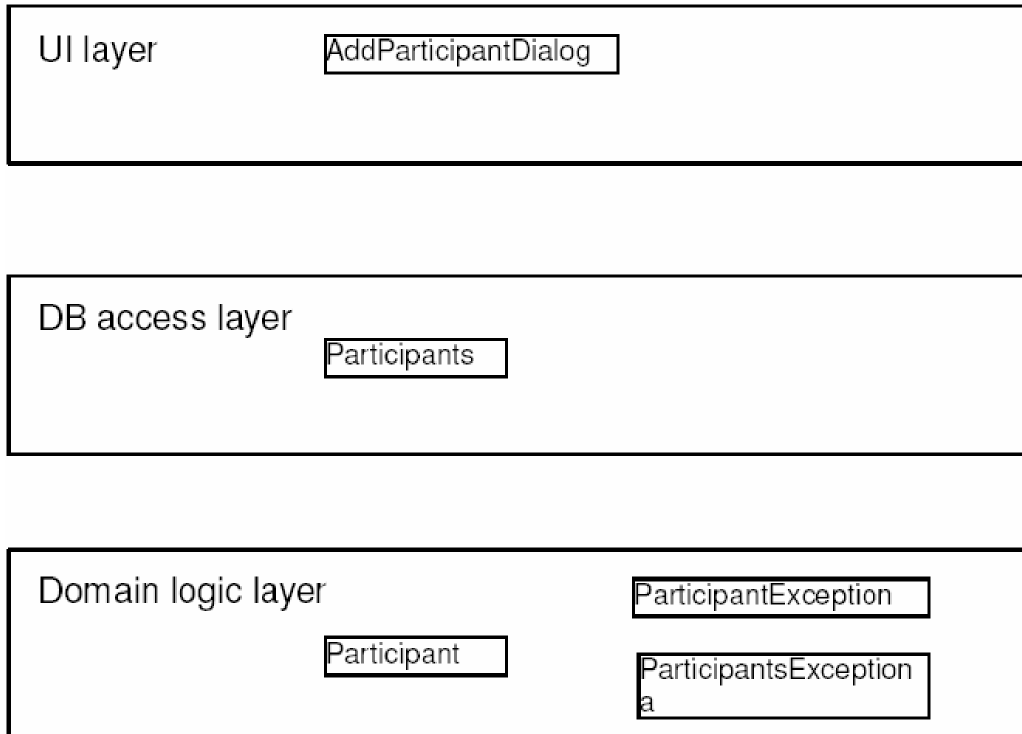
现在，AddParticipantDialog 这个类就不用处理 SQLException 这样的异常了：

```
class AddParticipantDialog extends JDialog {
    ...
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        try {
            newPart.setId(participants.getMaxId()+1);
        } catch (ParticipantsException e) {
            ...
        }
        showParticipant(newPart);
    }
}
```

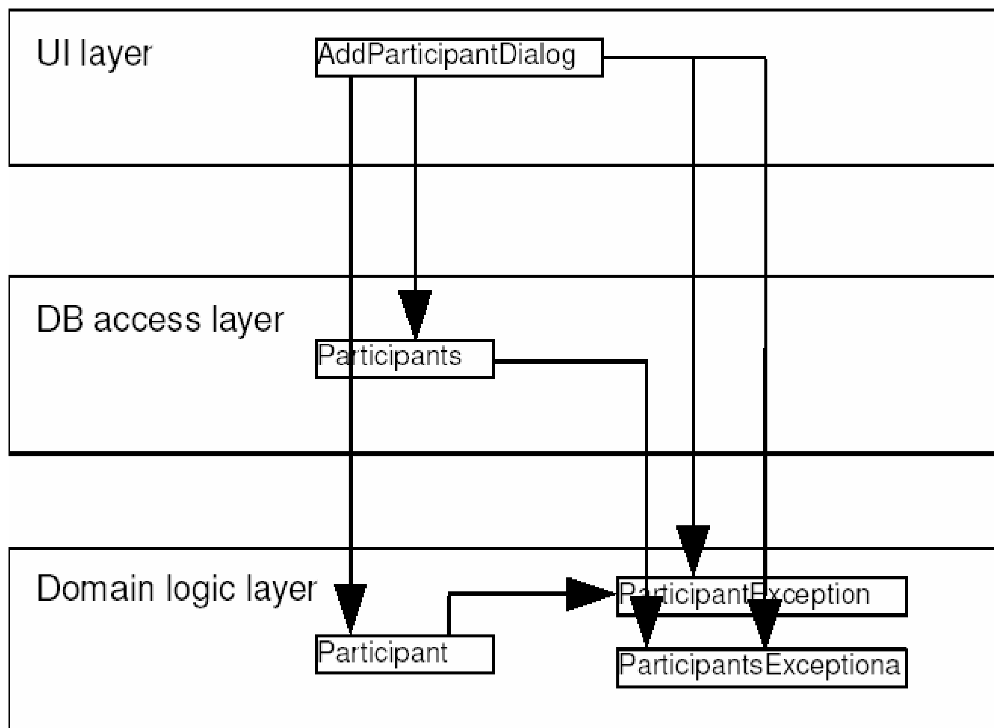
给系统分层

我们可以将上面的这个会议系统，分为三层："数据库访问层"，"域逻辑层"和"UI 层"：

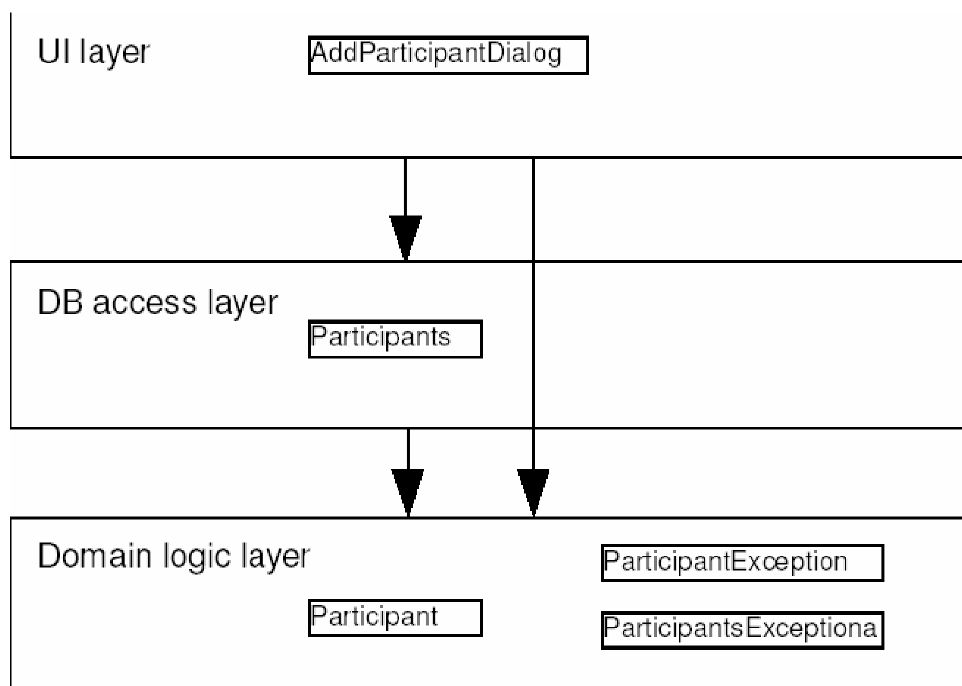
因为 Participants 这个类访问了数据库，所以我们将它放在数据库访问层。因为 Participant，ParticipantException，ParticipantsException 处理了跟域逻辑有关的事情，我们将它们放在域逻辑层。因为 AddParticipantDialog 是管理跟用户交互的界面的类，我们将它放在 UI 层：



（如果类 A 引用了类 B，那我们就从 A 画一个箭头到 B）下面就是这个系统中类之间的引用（依赖）关系：



而层之间的依赖关系就是：



我们看看从图中可以看出什么：

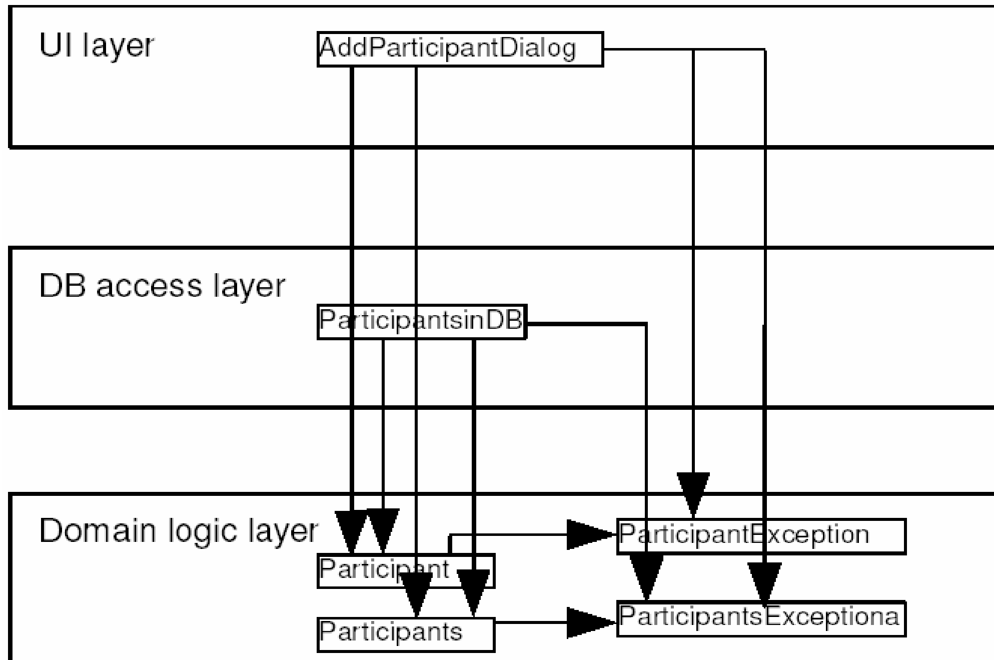
1.域逻辑层是唯一没有依赖于其他层的分层，所以这一层最容易重用。

2.数据库访问层依赖于域逻辑层，并没有依赖 UI 层。所以，不论是在文本文件存储系统中还是在 XML 存储系统中，这一层都可以重用。

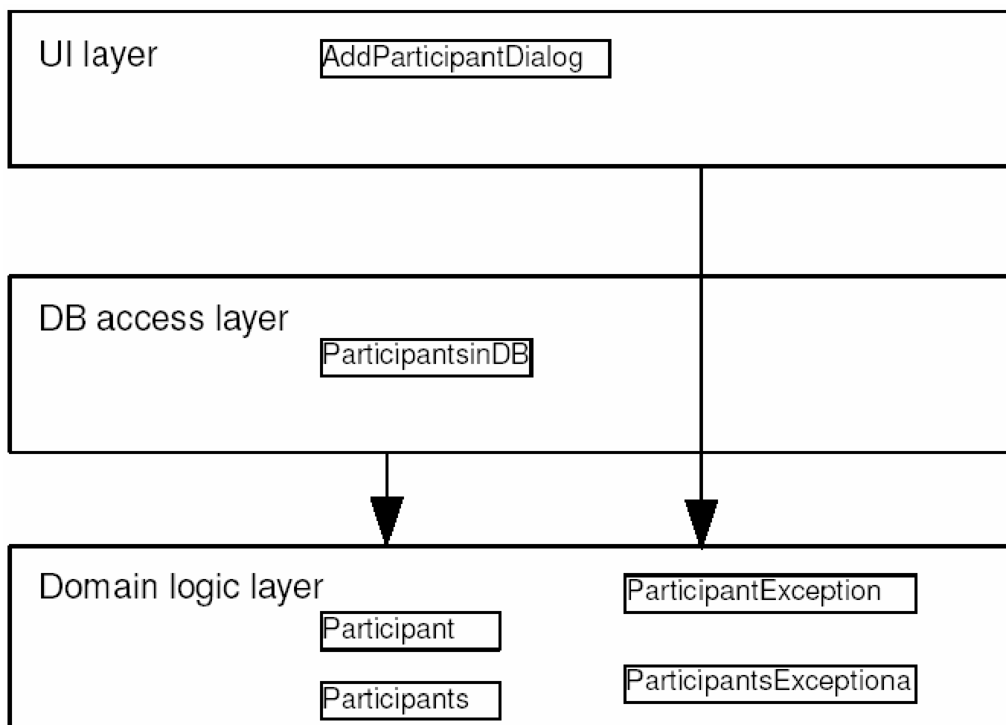
3.UI 层依赖于其他两层，因为这一层最难重用。

对于第 3 个观点（UI 层依赖于其他两层），我们再想一下：

我们不是已经让 AddParticipantDialog（UI 层）不知道数据库的存在了吗？怎么这图里面还是显示这样的依赖关系？根据我们前面讲的，我们是让 Participants 变成一个接口，它只是定义了一些业务逻辑操作，但并没有具体的实现。具体的实现则有多种情况。现在我们在数据库环境的系统中创建一个实现类叫 ParticipantsInDB，这样的话，很明显，Participants 就属于域逻辑层，而 ParticipantsInDB 则属于数据库访问层：



而层之间的依赖关系就是：



现在，各个层次间就只有两种依赖关系了：

1.UI 层依赖于域逻辑层

2.数据库访问层依赖于域逻辑层。The database access layer references the domain logic layer.

并不是所有的系统都可以做到分层间只有两个依赖关系的。比如我们刚开始的 AddParticipantDialog 这个类就不属于这样的好系统，我们甚至想把它归入某一层都做不到。只有结构比较好的系统才有可能做到这样。而这，

就是我们设计结构的目标（特别是大项目）。

很多东西都属于 UI 层

很多东西都属于 UI 层。不仅仅窗口，按钮之些属于 UI，报表，servlet，jsp，文本控制台等等也算。因此，请不要在 servlets 里面放置任何的域逻辑或者数据库访问的代码(特别要关注那些 Servlets 里面有很多代码的系统，它们是最有嫌疑的)；也不要在域逻辑中调用 System.out.print。

其他方法

将数据库看作一个对象的集合体，只是我们在 UI 层隐藏数据库层很多方法中的一个，其他的方法可以在下面的链接去看：

http://c2.com/cgi/wiki?ScatterSqlEverywhere .
http://c2.com/cgi/wiki?ModelFirst .
http://c2.com/cgi/wiki?ObjectRelationalMapping
http://www.agiledata.org/essays/mappingObjects.html
http://www.martinfowler.com/articles/dblogic.html
http://www.martinfowler.com/eaaCatalog.
http://www.objectmentor.com/resources/articles/Proxy.pdf

章节练习

介绍

有些问题可以测试前面几章的观点

问题

1.在 Participants 这个类里面实现这两个方法： updateParticipant 和 getAllParticipantsById：

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        //在这里写上代码
    }
    ParticipantIterator getAllParticipantsById() {
        //在这里写上代码
    }
```

```
}
```

2.这个系统跟餐厅有关。一份订单包括 ID，餐厅的 ID 和客户的 ID，还有一些预订的食物。每项预订的食物都有 ID，数量和单价。你已经创建了这样的数据表，设计了如下的类：

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null
);

create table OrderItems (
    orderId varchar(20),
    itemId int,
    foodId varchar(20) not null,
    quantity int not null,
    unitPrice float not null,
    primary key(orderId, itemId)
);
```

类：

```
class Order {
    String orderId;
    String customerId;
    String restId;
    OrderItem items[];
}
class OrderItem {
    String foodId;
    int quantity;
    double unitPrice;
}
```

你的任务就是：

1.设计一个接口，用来访问这些订单，而且要将数据库隐藏起来。

2.设计一个上面接口的实现类，里面实现一个增加订单到数据库的方法。

3.判断它们都属于哪一个分层。

3.在上面的系统，你想把所有食物已经送出去的订单记录下来，还要记录下送出的时间。根据之前讲的章节“保持代码的简洁”，你已经知道让 Order 这个类保持“苗条”是件好事，所以你决定创建新的类，不修改 Order 这个类。所以你写了下面的代码：

```
class OrderDelivery {
    String orderId;
    Date deliveryTime;
}

interface OrderDeliveries {
    boolean isDelivered(String orderId);
    Date getDeliveryTime(String orderId);
    void markAsDelivered(String orderId, Date deliveryTime);
}
```

你去询问了一下数据库管理员，说你想把这些信息存在数据库中。可是数据库管理员说，没必要再建一个新的表，你只需在 Orders 的表里面增加一个字段，用来保存送出时间 deliveryTime 就行了。如果这个字段为空，就表示这份订单还没送出去：

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null,
    deliveryTime datetime
);
```

你的任务就是：

1.为 OrderDeliveries 做一个实现类，实现里面的方法。

2.看一下如果现在你必须修改第 2 题里面写的代码，你要改哪里？

3.如果一份订单被删除了，那相关的 OrderDelivery 怎么办？

4.下面的程序是一个叫猜单词的游戏。这游戏这样玩的，电脑会出一个谜，谜底是一个单词，比如“banana”。玩家就是要猜出这个单词是什么。每一轮用户都可以输入一个字母，如果谜底包含这个单词的话，电脑就会显示这个单词，然后其余字母还是用横杠表示。比如现在你输入了“a”，然后电脑就会显示出"-a-a-a"。下一回如果你又输入了“b”，这时电脑就会显示“ba-a-a”。下一回如果玩家输入“c”，电脑还是会显示"ba-a-a"，因为这个单词不包含“c”。玩家最多可以输入 7 次。猜出这个单词，玩家胜，否则就输了。如果玩家重复输入一个之前已经输入的字母，电脑会提示他说，该字母已经输过，这次无效（就是说不计入 7 次里面）。

你的任务就是：

1.找出并修正下面代码里面的问题。
2.将你修正完的代码分层。

```
class Hangman {
```

```
String secret = "banana";
String guessedChars = "";
static public void main(String args[]) {
    new Hangman();
}
Hangman() {
    BufferedReader br = new BufferedReader(
                                        new InputStreamReader(System.in));
    for (int k = 0; k < 7;) { //最多猜 7 次
        String s = ""; //已经部分被猜出的谜底
        for (int i=0; i<secret.length(); i++) {
            char ch = secret.charAt(i);
            if (guessedChars.indexOf(ch)<0) //这个字母用户是不是已经猜出来了吗？
                ch = '-';   //没有，就显示成横杠
            s = s+ch;
        }
        System.out.println("谜底: "+s);
       System.out.print("猜的字母: ");
       char ch = br.readLine().charAt(0); //只读出 1 个字母
        if (guessedChars.indexOf(ch)>=0) { //这个字母是否已经猜过
            System.out.println("你已经猜过这个字母了!");
            continue;
        }
        int n = numberOfFoundChars();
        guessedChars = guessedChars+ch;
        int m = numberOfFoundChars();
        if (m>n) {
            System.out.println("成功，你猜出了字母"+ch);
            System.out.println("目前已经猜出的个数: "+m);
        }
        if (m==secret.length()) {
            System.out.println("你赢了！ ");
            return;
        }
        k++;
    }
    System.out.println("你输了！ ");
}
int numberOfFoundChars() {
    int n = 0;
    for (int i=0; i< secret.length(); i++) {
        char ch = secret.charAt(i);
        if (guessedChars.indexOf(ch)>=0)
```

```
                n++;
            }
            return n;
        }
    }
```

5.这是一个有关训练课程的 Web 系统。一个用户可以在一个表单里面选择一个课程的大类，比如（“IT”，“管理”，“语言”，“服饰”），然后点提交。然后程序就会显示出该大类下的所有子类的课程。负责这个的 servlect 的代码如下：

```
public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>课程列表</TITLE><BODY>");
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Courses where subjArea=?");
        try {
            st.setString(1, request.getParameter("SubjArea"));
            ResultSet rs = st.executeQuery();
            try {
                out.println("<TABLE>");
                while (rs.next()) {
                    out.println("<TR>");
                    out.println("<TD>");
                    out.println(rs.getString(1)); //课程代码号
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(rs.getString(2)); //课程名称
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(""+rs.getInt(3)); ////课程费用
                    out.println("</TD>");
                    out.println("</TR>");
                }
                out.println("</TABLE>");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
```

```
            }
            out.println("</BODY></HTML>");
    }
}
```

你的任务就是：

1.找出并修正代码里面的问题。

2.将你修改完后的代码分层。

提示
1.为了实现 getAllParticipantsById，你要建了个类，实现 ParticipantIterator，用来从一个 ResultSet 中读取数据。
2.你应该有增加，删除，更新一份订单和读取一定范围内的订单列表的功能。比如，在增加订单这个方法里面，你要将一条订单的记录加到 Orders 这个表里面。然后增加多个记录在 OrderItems 这个表里面。
3.没有提示。
4.UI 跟域逻辑混淆起来。将域逻辑抽取出来，放在一个类里，比如 Hangman 这个类。将原来的那个类改名为 HangmanApp。你要保证是 HangmanApp 引用了 Hangman 这个类，而不是相反的。
5.没有提示。

解决方法示例

1.在 Participants 这个类里面实现这两个方法： updateParticipant 和 getAllParticipantsById：

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        //在这里写上代码
    }
    ParticipantIterator getAllParticipantsById() {
        //在这里写上代码
    }
}
```

代码可能会是这样的：

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        PreparedStatement st =
```

```
                dbConn.prepareStatement(
                    "update participants set name=?, telNo=?, region=? where id=?");
            try {
                st.setString(1, part.getName());
                st.setString(2, part.getTelNo());
                st.setString(3, part.getRegion());
                st.setInt(4, part.getId());
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
        ParticipantIterator getAllParticipantsById() {
            PreparedStatement st =
                dbConn.prepareStatement(
                    "select * from participants order by id");
            return new ParticipantResultSetIterator(st);
        }
}

class ParticipantResultSetIterator implements ParticipantIterator {
        PreparedStatement st;
        ResultSet rs;
        ParticipantResultSetIterator(PreparedStatement st) {
            this.st = st;
            this.rs=st.executeQuery();
        }
        boolean next() {
            return rs.next();
        }
        Participant getParticipant() {
            new Participant(
                rs.getInt(1),
                rs.getString(2),
                rs.getString(3),
                rs.getString(4));
        }
        void finalize() {
            rs.close();
            st.close();
        }
}
```

2.这个系统跟餐厅有关。一份订单包括 ID，餐厅的 ID 和客户的 ID，还有一些预订的食物。每项预订的食物都有 ID，数量和单价。你已经创建了这样的数据表，设计了如下的类：

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null
);
create table OrderItems (
    orderId varchar(20),
    itemId int,
    foodId varchar(20) not null,
    quantity int not null,
    unitPrice float not null,
    primary key(orderId, itemId)
);

class Order {
    String orderId;
    String customerId;
    String restId;
    OrderItem items[];
}

class OrderItem {
    String foodId;
    int quantity;
    double unitPrice;
}
```

你的任务是:

1.设计一个接口，用来访问这些订单，而且要将数据库隐藏起来。

```
interface Orders {
    void addOrder(Order order);
    void deleteOrder(String orderId);
    void updateOrder(Order order);
    OrderIterator getOrdersById();
}
```

2.设计一个上面接口的实现类，里面实现一个增加订单到数据库的方法。

```
class OrdersInDB implements Orders {
    void addOrder(Order order) {
```

```
            PreparedStatement st =
                dbConn.prepareStatement("insert into from Orders values(?,?,?)");
            try {
                st.setString(1, order.getId());
                st.setString(2, order.getCustomerId());
                st.setString(3, order.getRestId());
                st.executeUpdate();
            } finally {
                st.close();
            }
            st = dbConn.prepareStatement(
                    "insert into from OrderItems values(?,?,?,?,?)");
            try {
                for (int i = 0; i < order.items.length; i++) {
                    OrderItem orderItem = order.items[i];
                    st.setString(1, order.getId());
                    st.setInt(2, i);
                    st.setString(3, orderItem.getFoodId());
                    st.setInt(4, orderItem.getQuantity());
                    st.setDouble(5, orderItem.getUnitPrice());
                    st.executeUpdate();
                }
            } finally {
                st.close();
            }
        }
    }
```

3.判断它们都属于哪一个分层

域逻辑: Orders.

数据访问: OrdersInDB.

3.上面的系统中，你想把所有食物已经送出去的订单记录下来，还要记录下送出的时间。根据之前讲的章节“保持代码的简洁”，你已经知道让 Order 这个类保持“苗条”是件好事，所以你决定创建新的类，不修改 Order 这个类。所以你写了下面的代码：

```
class OrderDelivery {
    String orderId;
    Date deliveryTime;
}
```

```
interface OrderDeliveries {
    boolean isDelivered(String orderId);
    Date getDeliveryTime(String orderId);
    void markAsDelivered(String orderId, Date deliveryTime);
}
```

你去询问了一下数据库管理员，说你想把这些信息存在数据库中。可是数据库管理员说，没必要再建一个新的表，你只需在 Orders 的表里面增加一个字段，用来保存送出时间 deliveryTime 就行了。如果这个字段为空，就表示这份订单还没送出去：

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null,
    deliveryTime datetime
);
```

你的任务就是：

1.为 OrderDeliveries 做一个实现类，实现里面的方法。

```
class OrderDeliveriesInDB implements OrderDeliveries {
    boolean isDelivered(String orderId) {
        PreparedStatement st = dbConn.prepareStatement(
                "select deliveryTime from Orders where orderId=?");
        try {
            st.setString(1, orderId);
            ResultSet rs=st.executeQuery();
            try {
                rs.next();
                return rs.getDate(1)!=null;
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
    void markAsDelivered(String orderId, Date deliveryTime) {
        PreparedStatement st = dbConn.prepareStatement(
                "update Orders set deliveryTime=? where orderId=?");
        try {
            st.setDate(1, new java.sql.Date(deliveryTime.getTime()));
```

```
                st.setString(2, orderId);
                st.executeUpdate();
            } finally {
                st.close();
            }
        }
    }
```

2.看一下如果现在你必须修改第 2 题里面写的代码，你要改哪里？

当新增一条订单记录到 Orders 表里面时，对于 Orders 表里面的新字段（deliveryTime），必须放空，因为这份订单肯定还没有送出去。在一些数据库管理系统下，你可以不必修改之前的代码，就可以做到。但为了以防万一，还是改吧：

```
class OrdersInDB implements Orders {
    void addOrder(Order order) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from Orders values(?,?,?,?)");
        try {
            st.setString(1, order.getId());
            st.setString(2, order.getCustomerId());
            st.setString(3, order.getRestId());
            st.setNull(4, java.sql.Types.TIME);
            st.executeUpdate();
        } finally {
            st.close();
        }
        ...
    }
}
```

3.如果一份订单被删除了，那相关的 OrderDelivery 怎么办？直接删除掉，而且这样看起来也是正确的。

4. 下面的程序是一个叫猜单词的游戏。这游戏这样玩的，电脑会出一个谜，谜底是一个单词，比如“banana”。玩家就是要猜出这个单词是什么。每一轮用户都可以输入一个字母，如果谜底包含这个单词的话，电脑就会显示这个单词，然后其余字母还是用横杠表示。比如现在你输入了“a”，然后电脑就会显示出"-a-a-a"。下一回如果你又输入了“b”，这时电脑就会显示“ba-a-a”。下一回如果玩家输入“c”，电脑还是会显示"ba-a-a"，因为这个单词不包含“c”。玩家最多可以输入 7 次。猜出这个单词，玩家胜，否则就输了。如果玩家重复输入一个之前已经输入的字母，电脑会提示他说，该字母已经输过，这次无效（就是说不计入 7 次里面）。

你的任务就是：

1.找出并修正下面代码里面的问题。
2.将你修正完的代码分层。

```
class Hangman {
    String secret = "banana";
    String guessedChars = "";
    static public void main(String args[]) {
        new Hangman();
    }
    Hangman() {
        BufferedReader br = new BufferedReader(
                                new InputStreamReader(System.in));
        for (int k = 0; k < 7;) { //最多猜 7 次
            String s = ""; //已经部分被猜出的谜底
            for (int i=0; i<secret.length(); i++) {
                char ch = secret.charAt(i);
                if (guessedChars.indexOf(ch)<0) //has it been guessed?
                    ch = '-';   //没有，就显示成横杠
                s = s+ch;
            }
            System.out.println("谜底: "+s);
            System.out.print("猜的字母: ");
            char ch = br.readLine().charAt(0); //只读出 1 个字母
            if (guessedChars.indexOf(ch)>=0) { //already guessed?
                System.out.println("你已经猜过这个字母了!");
                continue;
            }
            int n = numberOfFoundChars();
            guessedChars = guessedChars+ch;
            int m = numberOfFoundChars();
            if (m>n) {
                System.out.println("成功，你猜出了字母"+ch);
                System.out.println("目前已经猜出的个数: "+m);
            }
            if (m==secret.length()) {
                System.out.println("你赢了！ ");
                return;
            }
            k++;
        }
        System.out.println("你输了！ ");
    }
    int numberOfFoundChars() {
```

```
            int n = 0;
            for (int i=0; i< secret.length(); i++) {
                char ch = secret.charAt(i);
                if (guessedChars.indexOf(ch)>=0)
                    n++;
            }
            return n;
        }
    }
```

这段代码将域逻辑跟 UI 混淆了。而且代码里面的注释可以转为代码。

我们可以将域逻辑抽取到：

```
class Hangman {
    final static int MAXNOGUESSES = 7;
    String secret = "banana";
    String guessedChars = "";
    boolean reachMaxNoGuesses() {
        return getNoGuessedChars()==MAXNOGUESSES;
    }
    int getNoGuessedChars() {
        return guessedChars.length();
    }
    boolean hasBeenGuessed(char ch) {
        return guessedChars.indexOf(ch)>=0;
    }
    String getPartiallyFoundSecret() {
        String partiallyFoundSecret = "";
        for (int i=0; i< secretLength(); i++) {
            char ch = secret.charAt(i);
            char chToShow = hasBeenGuessed(ch) ? ch : '-';
            partiallyFoundSecret = partiallyFoundSecret+chToShow;
        }
        return partiallyFoundSecret;
    }
    boolean guess(char ch) {
        int n = numberOfFoundChars();
        guessedChars = guessedChars+ch;
        int m = numberOfFoundChars();
        return m>n;
    }
    boolean isSecretFound() {
        return numberOfFoundChars()==secretLength();
```

```
    }
    int numberOfFoundChars() {
        int n = 0;
        for (int i=0; i< secretLength(); i++) {
            char ch = secret.charAt(i);
            if (guessedChars.indexOf(ch)>=0)
                n++;
        }
        return n;
    }
    int secretLength() {
        return secret.length();
    }
}
```

UI 这一层会使用域逻辑：

```
class HangmanApp {
    static public void main(String args[]) {
        new HangmanApp();
    }
    HangmanApp() {
        Hangman hangman = new Hangman();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        while (!hangman.reachMaxNoGuesses()) {
            System.out.println("谜底: "+hangman.getPartiallyFoundSecret());
            System.out.print("猜的字母: ");
            char ch = readOneChar(br);
            if (hangman.hasBeenGuessed(ch)) {
                System.out.println("你已经猜过这个字母了!");
                continue;
            }
            if (hangman.guess(ch)) {
                System.out.println("成功，你猜出了字母"+ch);
                System.out.println("目前已经猜出的个数: "+
                    hangman.numberOfFoundChars());
            }
            if (hangman.isSecretFound()) {
                System.out.println("你赢了！ ");
                return;
            }
        }
        System.out.println("你输了！ ");
```

```
        }
        char readOneChar(BufferedReader br) {
            return br.readLine().charAt(0);
        }
    }
```

5.这是一个有关训练课程的 Web 系统。一个用户可以在一个表单里面选择一个课程的大类，比如（“IT”，“管理”，“语言”，“服饰”），然后点提交。然后程序就会显示出该大类下的所有子类的课程。负责这件事的 servlect 的代码如下：

```
public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>课程列表</TITLE><BODY>");
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Courses where subjArea=?");
        try {
            st.setString(1, request.getParameter("SubjArea"));
            ResultSet rs = st.executeQuery();
            try {
                out.println("<TABLE>");
                while (rs.next()) {
                    out.println("<TR>");
                    out.println("<TD>");
                    out.println(rs.getString(1)); //课程代码号
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(rs.getString(2)); //课程名称
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(""+rs.getInt(3)); ////课程费用
                    out.println("</TD>");
                    out.println("</TR>");
                }
                out.println("</TABLE>");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
```

```
            }
            out.println("</BODY></HTML>");
        }
}
```

你的任务就是：

1.找出并修正代码里面的问题。

2.将你修改完后的代码分层。

代码将域逻辑跟数据库还有 UI 混淆了。（请记住，servlet 是 UI）。我们可以将数据库访问抽取到：

```
interface Courses {
    Course[] getCoursesInSubject(String subjArea);
}

class CoursesInDB implements Courses {
    Course[] getCoursesInSubject(String subjArea) {
            Connection dbConn = ...;
            PreparedStatement st =
                dbConn.prepareStatement("select * from Courses where subjArea=?");
            try {
                st.setString(1, subjArea);
                ResultSet rs = st.executeQuery();
                try {
                        Course courses[];
                        while (rs.next()) {
                                Course course = new Course(
                                    rs.getString(1),
                                    rs.getString(2),
                                    rs.getInt(3));
                            add course to courses;
                        }
                        return courses;
                } finally {
                        rs.close();
                }
            } finally {
                st.close();
            }
    }
}
```

然后 UI 会使用域逻辑（而不是数据库层）：

```
public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                            HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>课程列表</TITLE><BODY>");
        Courses courses =
            (Courses)getServletContext().getAttribute("Courses");
        Course coursesInSubject[] =
            courses.getCoursesInSubject(request.getParameter("SubjArea"));
        out.println("<TABLE>");
        for (int i = 0; i < coursesInSubject.length; i++) {
            Course course = coursesInSubject[i];
            out.println("<TR>");
            out.println("<TD>");
            out.println(course.getCourseCode());
            out.println("</TD>");
            out.println("<TD>");
            out.println(course.getCourseName());
            out.println("</TD>");
            out.println("<TD>");
            out.println(course.getCourseFeeInMOP());
            out.println("</TD>");
            out.println("</TR>");
        }
        out.println("</TABLE>");
        out.println("</BODY></HTML>");
    }
}
```

第 8 章 以用户例事管理项目

什么是用户例事(user story)

假定这个项目的客户是个饮料自动售货机的制造商。他们要求我们为他们的售货机开发一款软件。我们可以找他们的市场经理了解这个软件的需求。

因此，我们的客户就是他们的市场经理。谈需求的时候，有一回他这样说：“用户往售货机每塞一个硬币，售货机都要显示当前该客户已经投了多少钱。当用户投的钱够买某一款饮料时，代表这款饮料的按钮的灯就会亮。如果那个用户按了这个按钮，售货机就放一罐饮料到出口，然后找零钱给他。”

上面的话描述的是一件事情，一件用户通过系统完成他一个有价值的目标（买一罐饮料）的事。这样的过程就叫“用户案例(user case)”或者“用户例事(user story)”。也就是说，上面我们的客户所说的话，就是在描述一个用户例事（user story）。

(我解释一下为什么用例事这个词，没兴趣也可以忽略。在一个系统面前，每个用户要完成同样的目标，都要做这个系统设定的例行的事，这件事情不是一个例子，所以不叫事例，这也不是故事，也不能算一段历程，而是一个例行的事。)

如果我们想要记下这段用户例事，我们可能会用这样的格式：

名称：卖饮料

事件：

1. 用户投入一些钱。

2. 售货机显示用户已经投了多少钱。

3. 如果投入的钱足够买某种饮料，这种饮料对应的按钮的灯就会亮。

4. 用户按了某个亮了的按钮。

5. 售货机卖出一罐饮料给他。

6. 售货机找零钱给他。

注意到，一个用户例事里面的事件可以这样描述：

1. 用户做 XX。

2. 系统做 YY。

3. 用户做 ZZ。

4. 系统做 TT。

5. ...

厦门:王伟杰(Wingel)　MSN: wingel_wang@hotmail.com
邮箱: seewingel@gmail.com　博客: http://blog.csdn.net/Wingel

用户例事只是描述系统的外在行为

一个用户例事只是以客户能够明白的方式，描述了一个系统的外在行为，它完全忽略了系统的内部动作。比如，下面有下划线的那些文字，就属于不应该出现在用户例事中的系统内部动作：

1. 用户投入一些钱。

2. 售货机将塞进来的钱存在钱箱里，然后发送一条命令给屏幕，屏幕显示目前已经投入的金额。

3. 售货机查询数据库里面所有饮料的价格，判定钱足够买哪些饮料，对于钱足够买的那些饮料，对应的按钮的灯就会亮起来。

4. 用户按下一个亮起来的按钮。

5. 售货机卖出一罐饮料给用户，然后将数据库里面该饮料的存货数量减 1。

6. 售货机找零钱给用户。

不管是口头描述的，还是书面形式，这样的内容是描述用户例事时一个很常见的错误。特别的，千万不要提及任何有关数据库，记录，字段之类的对客户一点意义都没有的东西。

评估发布时间

用户例事是用来干嘛的？假定客户希望在 50 天内递交这个系统。我们做得了吗？为了解答这个问题，我们就要在项目开始的阶段，试着找出所有的用户例事，然后评估一下，每一项历程需要多长的开发时间。可是，怎么评估呢？

比如，我们现在收集了下面这些用户例事：

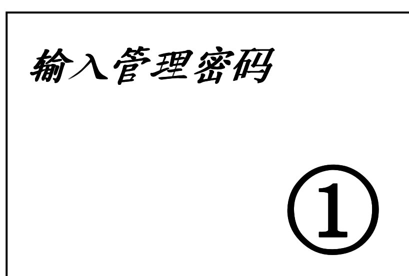
卖饮料：如上面所说的。
取消购买：在投入了一些钱后，用户可以取消购买。
输入管理密码：授权的人可以输入管理密码，然后增加存货，设定价格，拿走里面的钱等等。
补充饮料：授权的人可以在输入管理密码后增加存货。
取出钱箱里的钱：授权的人在输入管理密码后，可以取出钱箱里的钱箱里面的钱。
安全警报：有些事情经常发生的话，系统会自动打开安全警报。
打印月销售报表：授权的人可以打印出月销售报表。

然后找出里面最简单的用户例事（这里的“简单”，意思是说实现周期最短）。我们不一定非常精准的判断哪个最简单。只要挑出你觉得最简单的就行了。比如，我们觉得“输入管理密码”是最简单的用户例事。然后我们

判断说，这个用户例事算 1 个“例事点（story point）”。

用户例事	例事点
卖饮料	
取消购买	
输入管理密码	1
补充饮料	
取出钱箱里的钱	
安全警报	
打印月销售报表	

不过一般我们不会列出清单，而是做出一堆卡片贴在墙上，每张卡片记录一个用户例事，然后将例事点写在卡片上面：



这样的一张卡片就叫“例事卡（story card）”。

然后开始考虑其他用户例事。比如，对于“取出钱箱里的钱”这个例事，我们认为它跟“输入管理密码”这个例事一样简单，所以它应该也是算 1 个例事点。我们在列表里面标上。当然，实际操作的时候，我们是在“取出钱箱里的钱”的例事卡上填上例事点。

用户例事	例事点
卖饮料	
取消购买	
输入管理密码	1
补充饮料	
取出钱箱里的钱	1
安全警报	
打印月销售报表	

对于“取消购买”，我们认为它应该是“取出钱箱里的钱”的两倍的工作量，所以它算 2 个例事点。

用户例事	例事点
卖饮料	
取消购买	2
输入管理密码	1
补充饮料	
取出钱箱里的钱	1
安全警报	
打印月销售报表	

对于“卖饮料”，我们认为它应该是“取消购买”两倍的复杂度，所以它应该算 4 个例事点。

用户例事	例事点
卖饮料	4
取消购买	2
输入管理密码	1
补充饮料	
取出钱箱里的钱	1
安全警报	
打印月销售报表	

对于“补充饮料”，我们又认为它比“取出钱箱里的钱”复杂，但又比“卖饮料”简单，然后它又应该比“取消购买（ 2 个例事点）”复杂，所以我们认为它应该是 3 个例事点：

用户例事	例事点
卖饮料	4
取消购买	2
输入管理密码	1
补充饮料	3
取出钱箱里的钱	1
安全警报	
打印月销售报表	

类似的，我们认为“安全警报”应该比“补充饮料”简单一些，所以应该是 2 个例事点：

用户例事	例事点
卖饮料	4
取消购买	2
输入管理密码	1
补充饮料	3
取出钱箱里的钱	1
安全警报	2
打印月销售报表	

“打印月销售报表”应该跟“卖饮料”一样复杂，所以应该是算 4 个例事点，这样的话，我们总共有 17 个例事点：

用户例事	例事点
卖饮料	4
取消购买	2
输入管理密码	1
补充饮料	3
取出钱箱里的钱	1
安全警报	2
打印月销售报表	4
总计	17

现在挑出任意一个用户例事，估计一下它要花（你一个人）多少时间来完成。假设我们之前做过跟“取出钱箱里的钱”类似的功能，所以我们就挑这个来计算，估计它要花 5 天的时间。也就是说，一个例事点要花 5 天的时间来完成。现在我们有 17 个例事点，也就是说，我们需要 17×5＝85 天来完成这个项目（如果只是一个开发人员来做的话）。假设现在我们的团队里面有两个开发人员，所以我们就需要 85÷2＝43 天来完成。

从目前的估计来看，我们可以在 50 天的期限里面做完这个项目。但现在说这个还太早了！这样的评估，更

多的是猜测。通常开发人员在工作量上的估算都很差。事实上，人们经常会低估了工作量。那我们应该怎么估计得更准？

我们实际的开发速度是多少

为了做到更准的估计，我们就让客户先给我们两周的时间做一些实际的开发，来测量一下我们在这两周里面可以做多少的用户例事。我们叫这两周的时间为“迭代周期”。

哪些用户例事应该放在第一个迭代周期里面做？这可能完全是客户决定的，也可能是大家讨论以后决定的。不过挑出的这些用户例事的例事点不应该超过这两周的承受能力。因为一个迭代周期有 10 天（假设我们周末不工作），然后我们估计一个开发人员 5 天可以完成一个例事点。现在我们有两个开发人员，所以我们应该可以在这个迭代周期中完成（10÷5）×2＝4 个例事点。

然后客户可以在总例事点不超过 4 的前提下，挑出一些用户例事在这个迭代周期里实现。他们可能会尽量挑选他们觉得最重要的例事。比如，对他们来说，销售报表跟找零钱最重要，所以他们就挑出这两个：

用户例事	**例事点**
取出钱箱里的钱	**1**
打印月销售报表	**2**
总计	**3**

假设两周的迭代周期过去了，我们完成了“取出钱箱里的钱”，不过“打印月销售报表”没有完成，还剩 0.5 个例事点没有完成。也就是说，我们在这个迭代周期内完成了 3-0.5＝2.5 个例事点（比原来的预计要差一些）。

2.5 个例事点这样的数值，就是我们现在的参考值。也就是说，这个团队每个迭代周期可以完成 2.5 个例事点。这个参考值对我们很有用。它有两个用处：

首先，我们可以假定我们在下个迭代周期也可以完成 2.5 个例事点，然后客户选择的用户例事的例事点总和不能超过 2.5。

其次，在从第一个迭代周期取得参考值以后，我们可以重新估算我们的发布时间了。本来我们估算我们每个开发人员完成 1 个例事点的时间是 5 天，现在我们有了实践后的数据了，我们的团队（两个开发人员）可以在一个迭代周期（10 天）内完成 2.5 个例事点。现在总的例事点是 17，计算一下，我们需要 17÷2.5＝7 个迭代周期来完成。14 周，也就是 70 天。也就是说，我们满足不了客户提出的期限（50 天内）！怎么办？

预计不能如期完成时怎么办？

很明显，现在我们完成不了全部的用户例事。在这 50 天里面，我们只能完成 50÷10×2.5＝12.5 个用户例事。因为现在有 17 个例事点，我们应该让客户挑出总计 4.5 个例事点的用户例事，推迟到下一个发布周期去。客户应该选择那些比较次要的用户例事。比如，客户可以推迟“打印月销售报表”这个用户例事。

（这只是开发不能如期完成时的解决方法之一，这种方法应该是在客户比较有诚意合作的前提下使用。）

用户例事	**例事点**
卖饮料	**4**
取消购买	**2**
输入管理密码	**1**
补充饮料	**3**
取出钱箱里的钱	**1**

安全警报	**2**
总计	**13**

然后他还要挑出总和至少为 0.5 个例事点的例事。

但是如果"月销售报表"这个用户例事对客户来说也是非常重要，而且其他的例事也不能推迟，这时怎么办？这时我们就要试着简化这些用户例事。比如，原来"月销售报表"是要用一个第三方报表库来实现的，而且还要画出饼状统计图，如果我们只是生成简单的文本格式的报表的话（这格式应该可以被 Excel 导入，以便日后的处理）。那么这个用户例事的例事点就会从 4 减少到 2，节省了 2 个例事点。如果客户同意的话，我们就可以将"打印月销售报表"分成两个用户例事"生成月销售文本报表"和"生成图形报表"，然后将后者推迟到下一个发布。

现在新的例事卡片出来了：

用户例事	**例事点**
卖饮料	**4**
取消购买	**2**
输入管理密码	**1**
补充饮料	**3**
取出钱箱里的钱	**1**
安全警报	**2**
打印月销售文本报表	**2**
总计	**15**

还有其他的 2.5 个例事点要推迟，我们可以简化"卖饮料"。假设本来我们可以卖不同价格不同类别的饮料。如果现在我们只是简单支持一种价格一种类别的饮料的话，那这个用户例事的例事点可以从 4 减到 2 了。客户如果同意的话，我们就可以将"卖饮料"分割为"卖单一饮料"和"卖多种饮料"，然后将后者推迟到下个周期发布：

用户例事	**例事点**
卖单一饮料	**2**
取消购买	**2**
输入管理密码	**1**
补充饮料	**3**
取出钱箱里的钱	**1**
安全警报	**2**
打印月销售文本报表	**2**
总计	**13**

现在还剩 0.5 个例事点，我们再考虑一下"安全警报"。假设本来这个例事是要同时触发本机上的警报，跟通知附近的一个警察局的。如果现在我们只是触发本机的警报，那所花的例事点就可以从 2 减到 1 了。于是在客户同意的情况下，我们将"安全警报"分割为"本机安全警报"和"通知警察"，然后将后者推迟到下个发布：

用户例事	**例事点**
卖单一饮料	**2**
取消购买	**2**
输入管理密码	**1**
补充饮料	**3**
取出钱箱里的钱	**1**
本机安全警报	**1**
打印月销售文本报表	**2**

总计	**12**

现在我们总计有 12 个例事点要做了（<=12.5）。上面这个筛选在本次发布的用户例事的过程，叫“发布计划编制”。

增加开发人员来满足发布期限

在上面的例子中，我们以推迟部分用户例事到下个发布周期的办法来解决问题。这种“控制开发范围”通常是最好的解决办法。不过，这种解决办法实施不了的情况下，那你就只好保留所有的用户例事，然后增加更多的开发人员了。在这个例子中，假定我们需要“n”个开发人员，才能在 50 天内完成 17 个例事点。50÷10×2.5×n÷2. 算出来，n＝2.7，我们需要 3 个开发人员，也就是多加一个开发人员进来。不过注意：

团队人数加倍并不等于开发周期的减半。它可能只会缩短 1/3。如果团队超过 10 个人的话，增加更多的人员可能反而会延缓项目的进度。

而且项目开发周期越长，团队内的成员对整个项目代码的熟悉度就越少，加上不确定的人员流动，新来人员的业务不熟等其他可能性，这项目会越来越复杂。

总的意思就是，项目人数不能太多，周期不能太长。

根据参考值来掌控项目

每个迭代周期 2.5 个例事点的这个参考值，只是第一个迭代周期的数据，第二个迭代周期可能会变成 2 或者 3（一般是不会变动得太大）。假设是 2 的情况下，那对于第三个迭代周期，我们就要将参考值设为 2，然后让客户以 2 为例事点总数来挑选用户例事。

对于大多数项目，参考值很快就会稳定下来（比如在几个迭代周期后）。当这个值稳定下来后，我们就要重新估计开发周期，重新进行“发布计划编制”了。如果这个参考值告诉我们，我们每个迭代周期可以做 3 个例事点的话，我们就要让客户挑选更多的用户例事放在这次的发布计划中。相反如果这个参考值是 2 的话，我们就要让客户减少用户例事（需要的话可以分割一些用户例事），如果团队人员还不多的话，可以增加更多的开发人员。

这是项目的初始阶段绝对要注意的。

发布计划编制，估算每个用户例事时要考虑哪些细节，忽略哪些细节？

在项目初始，我们要找出这个发布周期内所有主要的用户例事，评估每个例事的例事点。可是要怎么评估里面的细节呢？比如对于“卖饮料”， “卖饮料”这个简单的标题，省略了很多的细节：用户会投入什么样的钱？纸币可以吗？人民币可以吗？按钮的灯的亮度要多少？可不可以多个按钮对应一种饮料？按钮被按下以后，要不要变暗？找零钱是不是全部找 10 分的面额？

我们是不是要考虑上面所有的细节？对于按钮灯的亮度，我们就不用考虑了，它对我们的工作量没影响。不过，零钱的面额就对我们的工作量很有影响，我们要认真考虑一下（找一堆 10 分的零钱就很容易实现；如果要尽量减少零钱的个数就比较麻烦了）。处理不同币种也要考虑。

一般情况，我们不用太担心会漏过什么细节。对于每个用户例事，只要考虑一些“重要”问题就行了。当然，这里面的“重要”，就要根据经验以及客户的观点来决定了。

如果我们不好估算的话怎么做

如果我们觉得，这个用户例事不好估算，那可能的原因就是：

1. 这个用户例事太大。这种情况我们就可以将这个用户例事分割出若干个新的用户例事，比如：
   将“卖饮料”分割出：
   1：显示总投入金额。
   2：金额够买的饮料对应的按钮灯亮起来。
   3：按下亮灯的按钮，可以买到对应的饮料。
2. 我们之前从没开发过自动售货机的程序。因此，我们不知道开发这样的程序有多复杂。这样的话，我们就要做一些实验了，比如做一个让售货机找钱的小程序。这种试验就叫“spike”（翻译不出来）。

迭代周期内的计划编制

对于这个迭代周期内选择的所有用户例事，不像在发布计划编制那样，只是考虑一些重要的细节，现在我们要从客户那里调查到所有的细节。比如对于“卖饮料”，我们可能会在白板上画出用户交互的草图，然后跟客户一起讨论：

这是一台自动售货机……
用户投入硬币……
假设他投入的是 50 分，而价格是 40 分，那么按钮就会亮起（别忘了我们现在做的只卖单一饮料）
用户按一个亮起的按钮，一罐饮料会掉到售货机的出口
找零钱……

在跟客户详细讨论完，了解了足够的细节以后，我们才发现，事实上这个用户例事“卖饮料（只卖单一的）”的例事点远远比我们预计的要麻烦，这时候应该类似前面的发布计划编制那样，1、分割出小的用户例事，挑出一些放在下一个迭代周期内；或者 2、挑出这个迭代周期内的一些用户例事放在下一个迭代周期。反之，如果发现这个用户例事比我们想像的还要简单，那我们就要增加更多的用户例事到这个迭代周期内。

用户例事只是跟用户交流的开始，而不是全部

假想现在已经从客户那边得到足够详细的需求了，我们可以开始实现了。注意，我们不用把所有用户提供的细节都记录下来。为什么呢？假设以后，你有点忘记用户例事，而客户又在你旁边，你是直接问客户，还是去找看需求文档找到你要的东西？当然是直接问客户了。客户可以提示更准确，更完整的需求给你。特别要注意的是，以后只要你一完成一个用户例事，你就要让客户看一下，或者实际的操作一下，因为客户对已经做的的东西了解得越多，那他就可以提供越准确越完整的需求。

用户挑选完用户例事以后，在之后的两个星期内，我们就要将这些用户例事逐个完成。每个用户例事我们都会设计结构，编码，测试等等。每做完一个用户例事，我们都要让客户验证一下系统是不是他所想的那样。

在这两个星期内，如果我们提早完成了用户例事，我们就要让客户挑更多的用户例事。相反的，如果我们不能及时完成，我们就要让客户知道当前的进度。

总结

我觉得这章的内容跟其他的软件工程书一样，看看，参考参考，具体的情况还是要具体的分析，不过这里面的用户例事（user story）跟例事卡片的概念就很不错，可以引用。

章节练习（这章不用章节练习）

第 9 章　用 CRC 卡协助设计

一个用户例事的例子

假定这个项目的客户是个饮料自动售货机的制造商。他们要求我们为他们的售货机开发一款软件。他们的需求是这样说的，“用户每往售货机投入一次钱，售货机都要显示当前该客户已经投了多少钱。当用户投的钱够买某一款饮料时，代表这款饮料的按钮的灯就会亮。如果那个用户按了这个按钮，售货机就出一罐饮料到出口，然后找零钱给他。”

这个用户例事就是：

名称：卖饮料

事件：

1. 用户投入一些钱。
2. 售货机显示用户已经投了多少钱。
3. 如果投入的钱足够买某种饮料，这种饮料对应的按钮的灯就会亮。
4. 用户按了某个亮了的按钮。
5. 售货机卖出一罐饮料给他。
6. 售货机找零钱给他。

用户例事的设计

为了实现这个用户例事的 1， 2 步，我们要找出这个系统要执行一些什么动作：

1.用户投入一些钱。

　动作 1：检测到有人投了钱进来

2.售货机显示用户已经投了多少钱。

　动作 2：计算出用户已经投了多少钱了。

　动作 3：显示出来。

3....

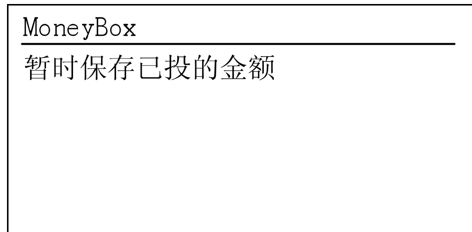
4....

5....

6....

可是这些动作的执行者是谁？假设用类 MoneyBox 代表钱箱，它要做的事情有：每次有人投钱他都要检测到

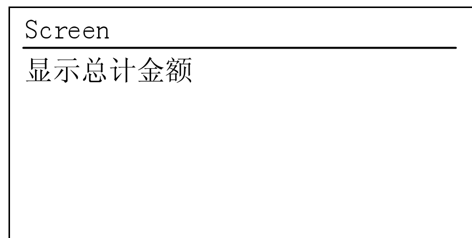
（动作 1），然后还要保存这个用户的已投金额（动作 2）。而从软件设计的角度在考虑问题时，这个类的职责应该是：



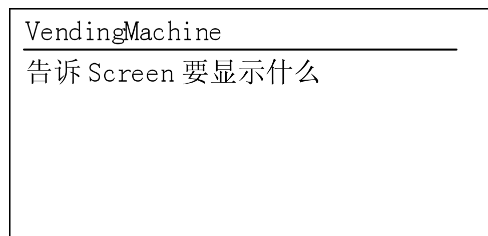
写在卡上的“暂时保存已投的钱额”就是这个类的“职责”。

为了实现这种职责，MoneyBox 可能需要一些属性和方法，比如一个叫“余额”的属性加上 putMoney，takeMoney，returnMoney，getAmount 等等的这些方法……等等等等，我们考虑得太远了！我们现在是在考虑抽象的概念，而不是具体的实现：属性，方法等。

好，我们再重头考虑一遍。用户投入一些钱。MoneyBox 检测到有人投钱。它会把这个人投钱的总计金额记住。不过，谁来显示这个总计金额（动作 3）？假设我们还有一个 Screen 类（多做一张卡）：

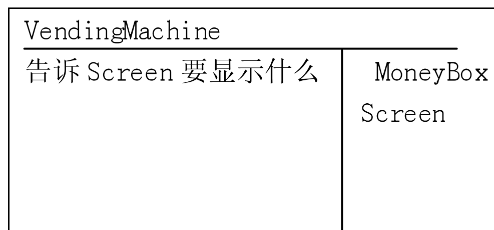


那现在谁来把总计金额这个数据从 MoneyBox 传给 Screen？这件事情，应该交给一个第三方的类来完成，从它的职责看来，这个类有枢纽的作用，不如建一个 VendingMachine 类吧（再建一张新卡）：



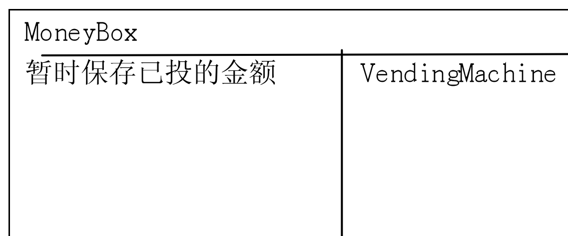
一个用户投入一些钱。MoneyBox 检测到有人投钱，保存已投的金额，通知 VendingMachine。VendingMachine 从 MoneyBox 里面取出总计金额这个数据，传给 Screen 去显示。

如果我们希望可以强调说，VendingMachine 要跟 MoneyBox 还有 Screen 一起协作，我们可以将 VendingMachine 的卡片改成这样：



卡片右边的“MoneyBox”和“Screen”表示：VendingMachine 要跟 MoneyBox 和 Screen 共同协作。

因为只要有人投钱，MoneyBox 就要通知 VendingMachine，还要把数据传给它。为了强调 VendingMachine 也是 MoneyBox 的协作者，我们将 MoneyBox 这张卡片也改了：

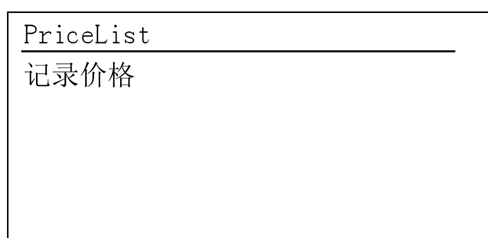


因为在这些卡里面，我们写上了类名，它的职责，以及它的协作关系，我们管这样的卡片叫“CRC 卡”。CRC 就是 Class，Responsibility 和 Collaboration 的简称。

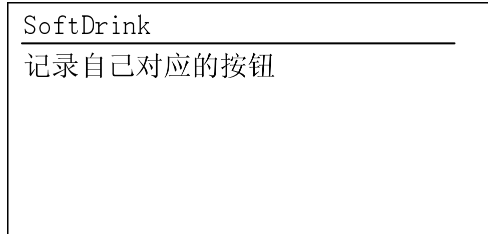
现在，我们再来看看系统在第三个步骤里面要执行的动作：

1. 用户投入一些钱。
   动作 1：检测到有人投了钱进来
2. 售货机显示用户已经投了多少钱。
   动作 2：计算出用户已经投了多少钱了。
   动作 3：显示出来。
3. 如果投入的钱足够买某种饮料，这种饮料对应的按钮的灯就会亮。
   动作 4：判断每种饮料的价格
   动作 5：如果已投金额超过某种饮料的价格，取得该饮料对应的按钮
   动作 6：让那个按钮亮起来
4. ...
5. ...
6. ...

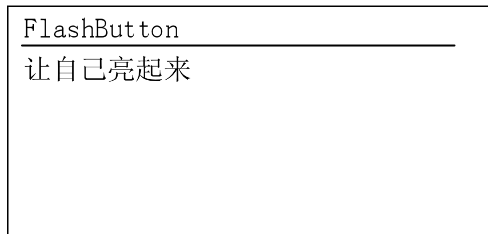
现在谁来执行这 4,5,6 这 3 个动作？假定 VendingMachine 取得所有饮料的价格（动作 4）。可是哪里去取所有饮料的价格？所以我们又要创建一个 PriceList 的对象（再建个新卡吧）：



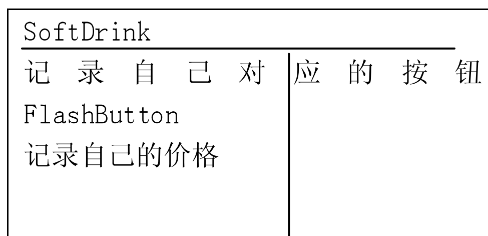
VendingMachine 取得所有价格以后，判断价格跟投入金额的大小，如果价格都比投入金额大，那么就什么都不做。假定有些饮料的价格比投入金额小，现在 VendingMachine 就要要判断哪些按钮对应这些饮料。这边我们再创建一个 SoftDrink 类，它的每一个实例代表一种饮料，这个对象保存它所对应的按钮（动作 5）：



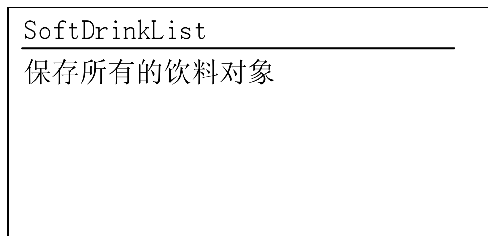
对于按钮，我们又要创建一个新的类 FlashButton：



现在既然我们已经有 SoftDrink 这个类了，为什么不让这个类自己记录自己的价格，顺便把它跟 FlashButton 的协作关系也表示上？



这样我们就不需要 PriceList 这个类了，不过我们还需要一个对象，来保存所有的饮料对象。我们管它叫 SoftDrinkList：



好，现在的思路就是，VendingMachine 取得所有的饮料对象 SoftDrinkList，逐个判断它们的价格。对于价格对投入金额小的饮料 SoftDrink，VendingMachine 取出它对应的按钮 FlashButton，让它亮起来。

好，现在我们可以把目前三个步骤 6 个动作看下来了： MoneyBox 检测有人投钱（动作 1），保存已投的金额（动作 2），通知 VendingMachine，VendingMachine 取得已投金额,传给 Screen 显示（动作 3），VendingMachine 取得所有的饮料对象 SoftDrinkList，逐个判断价格跟已投金额的大小（动作 4）,如果哪个饮料的价格比已投金额小，找到它对应的按钮（动作 5）,让它亮起来（动作 6）。

现在，我们把这个用户例事里面系统所有要执行的动作列出来吧：

1. 用户投入一些钱。

   动作 1：检测到有人投了钱进来

2. 售货机显示用户已经投了多少钱。
   动作 2：计算出用户已经投了多少钱了。
   动作 3：显示出来。
3. 如果投入的钱足够买某种饮料，这种饮料对应的按钮的灯就会亮。
   动作 4：判断每种饮料的价格
   动作 5：如果已投金额超过某种饮料的价格，取得该饮料对应的按钮
   动作 6：让那个按钮亮起来

4. 用户按了某个亮了的按钮。
   动作 7：注意到有人按了按钮
5. 售货机卖出一罐饮料给他。
   动作 8：检查哪个按钮被按了。
   动作 9：找到该按钮对应的饮料。
   动作 10：将一罐饮料放在出口。
   动作 11：将按钮熄灭。
6. 售货机找零钱给他。
   动作 12：找零钱。

这之间的过程，我想不必我再描述。下面是其他可能要建的 CRC 卡：

SoftDrinkDispenser	
将饮料放在出口	SoftDrink

CRC 卡的典型应用

为什么用 CRC 卡，而不用文档或者更先进的 UML 工具？

1. 卡片上面的空间很小，这样就可以防止我们给这个类太多的职责。如果一个类的职责太多的话（比如，超过 4 个），尝试以更抽象的方式去考虑一下，将职责划分。
2. CRC 卡主要是用在探索或者讨论类的设计的阶段。如果我们觉得这个设计不行的话，我们既不用修改文档，也不用修改类图，只要把卡片丢了就行了。此外，一旦设计完成，我们就可以把所有的卡丢了。它们不是用来做文档的。
3. 如果我们觉得现在的卡片不合适，之前设计的比较好，我们只要简单的把之前的卡片拿出来组合就行了。

CRC 卡主要是用来快速的组织设计。我们不应该花很长的时候在做 CRC 卡上，也不要指望按照 CRC 卡的设计就一切 OK 了。在编码的时候，肯定还会不断的调整设计。将设计与编码结合起来，我们才能做出好而有效的设计。

对于 CRC 的使用方法并没有一个正式的定义，或者说，它要怎么用，并没有任何限制。我们不用太在意每张卡上写了什么，遗漏了什么。对于有些人来说，在每张卡上写个类名就足够了。而有些人认为，把职责也写在 CRC 卡上，会帮助他们更好的思考。而有些人则希望把协作关系也写上去。我们也不用在意每张卡要跟哪张卡放在一

起。一句话，CRC 是用来帮忙理清设计的思路，它不是 UML 图，也不是精确的类结构图。只要我们在处理这些卡的时候不断的讨论，我们设计的思路将会变成非常清楚。

引述
http://c2.com/doc/oopsla89/paper.html.

http://c2.com/cgi/wiki?CrcCards .

第 10 章　验收测试（Acceptance Test）

我们是不是正确的实现了一个用户例事

（用户例事：user story，在第 8 章有讲）

假设这个项目的客户是一个会议展览的组织者。他们希望我们开发一款软件，可以帮助他们管理会议的所有参会者信息。我们编制好发布计划，在目前的迭代周期中，我们要实现 4 个用户例事。下面是其中的一个用户例事：

名称：**导入参会者信息**
事件：
1. 一个用户让系统读取一个记录了一批参会者信息的文本文件。里面的信息有 ID，密码，名称，地址跟邮箱。
2. 系统将这些参会者的信息保存下来。之后，只要用户输入一个参会者的 ID，系统就可以取出对应的参会者信息。
3. 系统导入完成后，各给每个参会者发送一封邮件，里面包含该参会者的 ID 和密码。

我们开始询问客户这个用户例事的细节。比如：
这个文本文件的格式是什么样的？假定客户说，文本文件里面的每行字符串包含一个参会者。这个参会者每项数据会用制表符隔开。
ID，密码，姓名，地址还有邮箱都是一定会出现在文本文件里面的吗？假定客户说有些参会者的地址会留空，其他数据都要出现。否则，系统将跳过这行。
如果参会者的 ID 已经存在的话怎么办？假定客户说，那这行会被跳过。
等等。

我们问完客户，用 CRC 卡或者其他的方法来快速组织跟讨论设计，写代码，同时改进设计。假定两天以后，我们完成了所有的代码，代码结构的设计也很合理了。好，现在我们还要做一件重要的事件：测试我们的代码是不是正确的实现了用户例事。

怎么测试

怎么测试？比如，我们运行下面的“测试用例”：

测试用例 1：导入参会者

1. 创建下面这样的文件：

```
p001    123456      Mary Lam        abc      mary@hotmail.com
p004    888999      John Chan       def      john@yahoo.com
p002    mypasswd    Paul Lei        ghi      paul@excite.com
```

2. 删除系统里面已有的参会者信息，防止 p001，p002，p004 已经存在。
3. 运行系统，将上面的文件导入到数据库里面。
4. 检查系统是不是正确的导入了文件。这里面，我们肯定有一个用户例事是让用户输入一个参会者的 ID，然后系统显出这个参会者的所有信息。我们可以先实现这个用户例事，然后输入 p001,看看系统会不会显示 p001 的正确信息（123456，Mary Lam 等等），然后再输入 p002 和 p004。
5. 检查系统有没有发邮件。我们可以联系 Mary，John 和 Paul，确认一下他们有没有收到邮件，邮件里面的内容是不是正确的。

这样的测试就叫“验收测试”或者“功能测试”。这样的测试只是测试系统的外部行为，忽略系统里面有哪些类，哪些模块。

为了测试在没有填入某个参会者的地址的情况下，系统是不是正确的导入，我们又建了下面这样的测试用例：

测试用例 2：导入没有地址的参会者。

1. 创建下面的文件：

```
p001 123456      Mary Lam      abc      mary@hotmail.com
p004 888999      John Chan                john@yahoo.com
p002 mypasswd    Paul Lei        ghi      paul@excite.com
```

2. 删除系统里面的所有参会者信息。
3. 运行系统，让它将上面的文件导入数据库。
4. 输入 p004，看看这个参会者的信息是不是正确的。
5. 联系 John，问他有没有收到邮件，以及邮件的内容是不是正确的。

现在，如果有不是地址的其他信息留空的话，系统是不是正确运行？我们又建了下面这个测试用例：

测试用例 3：导入没有其他信息的参会者

1. 创建下面的文件：

123456 Mary Lam abc mary@hotmail.com
p004 888999 def john@yahoo.com
p002 Paul Lei ghi paul@excite.com
p005 secret Mike Chan jkl

2. 删除系统里面的所有参会者信息。
3. 运行系统，让它将上面的文件导入数据库。
4. 分别输入空字符串，p002，p004 和 p005，看一下系统会不会显示对应的信息，显示的信息对不对。
5. 联系 Mary，Paul 和 Mike，问他们有没有收到邮件，以及邮件的内容是不是正确的。

为了测试在某个参会者 ID 已经存在的情况下，系统还会不会导入该参会者的信息，我们又建了下面这个测试用例。

测试用例 4：导入跟已有参会者 ID 重复的参会者

1. 创建下面的文件：
p001 123456Mary Lam abc mary@hotmail.com
p001 888999John Chan def john@yahoo.com

2. 删除系统里面的所有参会者信息。
3. 运行系统，让它将上面的文件导入数据库。
4. 输入 p001，看看系统会显示 Mary 的信息，还是显示 John 的信息。而根据用户例事的描述，它应该显示 Mary 的信息。
5. 联系 Mary，看看她有没有收到内容正确的邮件。
6. 联系 John，看看他有没有收到邮件（他不应该收到）。

手动测试的问题

我们已经创建了 4 个测试用例。他们都有一个共同点：用例里面的每个步骤（创建文本文件，运行系统进行导入，输入参会者 ID 检查显示信息是否准确，联系参会者看有没有收到正确邮件）都要手动来完成。因此，这些用例肯定要花很多时间和精力。

这是一个很严重的问题。我们在实现其他用户例事的时候，可能要改动到这个用户例事相关的代码。而这些改动可能会引起一些 bug。所以每次改动完代码，我们都要运行一下上面 4 个测试用例，看看能不能顺利通过，如果不能通过的话，我们就可以立刻调查，看看问题出在哪里。可是现在的问题是，每运行一次这些测试用例的代价都很大，我们不可能经常去跑这些测试用例。

为了解决这个问题，我们希望这些测试用户可以在不需人为干预的情况下自动运行。这样的测试，就叫“自动验收测试（Automatic Acceptance Test)”。

自动验收测试

我们写了下面的代码来自动运行这些测试用例，每个方法对应一个测试用例：

```
class AcceptTestForImportParticipants {
   static void testImport() {
      ...
   }
   static void testImportWithoutAddress() {
      ...
   }
   static void testImportWithoutOtherInfo() {
      ...
   }
   static void testImportDupId() {
      ...
   }
}
```

这里面，testImport 会实现测试用例 1 的测试。在写 testImport 这个方法前，我们再看看测试用例 1：

测试用例 1：导入参会者

1. 创建下面这样的文件：

```
p001   123456     Mary Lam     abc    mary@hotmail.com
p004   888999     John Chan    def    john@yahoo.com
p002   mypasswd   Paul Lei     ghi    paul@excite.com
```

2. 删除系统里面已有的参会者信息，防止 p001，p002，p004 已经存在。
3. 运行系统，将上面的文件导入到数据库里面。
4. 检查系统是不是正确的导入了文件。这里面，我们肯定有一个用户例事是让用户输入一个参会者的 ID，然后系统显出这个参会者的所有信息。我们可以先实现这个用户例事，然后输入 p001,看看系统会不会显示 p001 的正确信息（123456，Mary Lam 等等），然后输入 p002 和 p004。
5. 检查系统有没有发邮件。我们可以联系 Mary，John 和 Paul，确认一下他们有没有收到邮件，邮件里面的内容是不是正确的。

现在我们想要实现上面这个用例的每个步骤。我们会用一个“Command”来代表测试用例里面的一个步骤。要运行一个步骤（Command），只要调用这个 command 的 run 方法就行。如果运行成功的话，会返回 true：

```
interface Command {
   boolean run();
}
```

现在，实现第 1 个步骤（创建一个文本文件）：

```
class AcceptTestForImportParticipants {
   static void testImport() {
       Command commands[] = {
           new CreateImportFileCommand(
              "p001\t123456\tMary Lam\tabc\tmary@hotmail.com\n"+
              "p004\t888999\tJohn Chan\tdef\tjohn@yahoo.com\n"+
              "p002\tmypasswd\tPaul Lei\tghi\tpaul@excite.com"),
           ...
       };
       runCommands(commands);
   }
   static void runCommands(Command commands[]) {
       for (int i = 0; i < commands.length; i++) {
           if (!commands[i].run()) {
              System.out.println("Test failed!");
           }
       }
   }
}

class CreateImportFileCommand implements Command {
   String fileContent;
   CreateImportFileCommand(String fileContent) {
       this.fileContent = fileContent;
   }
   boolean run() {
       FileWriter fileWriter = new FileWriter("sourceFile.txt");
       try {
           fileWriter.write(fileContent);
           return true;
       } finally {
           fileWriter.close();
       }
   }
}
```

实现第 2 个步骤（删除系统里面的所有参会者信息）：

```
class AcceptTestForImportParticipants {
   static void testImport() {
       Command commands[] = {
           new CreateImportFileCommand(...),
           new DeleteAllParticipantsCommand(),
```

```
            ...
        };
        runCommands(commands);
    }
}
```

为了实现 DeleteAllParticipantsCommand，我们现在假定在 Participants 这个类里面有提供 deleteAll 的方法。这样，DeleteAllParticipantsCommand 就可以简单的调用它：

```
class DeleteAllParticipantsCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().parts.deleteAll();
        return true;
    }
}
class ConferenceSystem {
    Participants parts;
    static ConferenceSystem getInstance() {
        ...
    }
}
class Participants {
    void deleteAll() {
        ...
    }
}
```

实现第 3 个步骤（让系统导入文件）

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            ...
        };
        runCommands(commands);
    }
}

class ImportParticipantsFromFileCommand implements Command {
```

```
    boolean run() {
        ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
        return true;
    }
}

class Participants {
    void deleteAll() {
        ...
    }
    void importFromFile(String path) {
        ...
    }
}
```

实现第 4 个步骤（从系统重新取到 p001 等等的参会者信息）:

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            ...
        };
        runCommands(commands);
    }
}

class CheckParticipantStoredCommand implements Command {
    Participant part;
    CheckParticipantStoredCommand(String partId, String password, ...) {
        part = new Participant(partId, password, ...);
    }
    boolean run() {
        return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
    }
}

class Participants {
```

```
    boolean checkParticipantStored(Participant part) {
        return part.equals(getParticipantById(part.getId()));
    }

    Participant getParticipantById(String partId) {
        ...
    }
}

class Participant {
    boolean equals(Object obj) {
        ...
    }
}
```

现在我们要实现第 5 个步骤了（检查系统有没有发邮件。我们可以联系 Mary，John 和 Paul，确认一下他们有没有收到邮件，邮件里面的内容是不是正确的。)，不过要实现这步好像非常麻烦。比如，要写代码实现访问 Mary 的邮箱就很麻烦，更何况我们还没有她的密码。而且，说不定她已经将系统发给她的邮件删掉了，或者那封邮件还在邮件服务器上没发出去，所以，直接去查她的邮箱这条路行不能。我们只能降低要求。现在我们不检查她有没有收到邮件，我们检查我们的系统有没有发送过这封邮件。严格的讲，我们要检查有没有发送 SMTP 命令，命令里面的接收人，邮件内容是不是正确的。这也不是那么容易的事。因此，我们再降低一下要求。我们假定系统会记录下每封已发邮件的收件人跟内容。这样，在这个测试里面，我们只要检查这个记录就行了：

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001","123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id & ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ..."),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ..."),
            ...
        };
        runCommands(commands);
    }
}

class CheckRemoveOldestMailCommand implements Command {
    MailRecord mailRecord;
```

```
    CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
        mailRecord = new MailRecord(recipientEmail, mailContent);
    }
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailRecord.equals(mailLog.takeOldestMailRecord());
    }
}

class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    static ConferenceSystem getInstance() {
        ...
    }
}

class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
}

class MailRecord {
    MailRecord(String recipientEmail, String mailContent) {
        ...
    }
    boolean equals(Object obj) {
        ...
    }
}
```

事实上，这里有个问题：当测试用例在运行的时候，系统的邮件日记里面可能还有一些残留的记录。所以我们在测试前，还要先清空邮件日记：

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new EmptyMailLogCommand(),
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
```

```
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ..."),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ..."),
            ...
        };
        runCommands(commands);
    }
}

class EmptyMailLogCommand implements Command {
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        mailLog.empty();
        return true;
    }
}
class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
    void empty() {
        ...
    }
}
```

我们发现在每个测试用例的开始，我们都要清空一堆东西，比如邮件日记，参会者信息等等。我们需要一个command 是清空或者初始化系统所有东西的。我们姑且叫它 SystemInit 吧。因此，我们不再需要 DeleteAllParticipants 和 EmptyMailLog 这两个 command 了：

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new SystemInitCommand(),
            new CreateImportFileCommand(...),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id& ..."),
```

```
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ..."),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ..."),
            ...
        };
        runCommands(commands);
    }
}

class SystemInitCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().init();
        return true;
    }
}

class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    void init() {
        parts.deleteAll();
        mailLog.empty();
    }
}
```

我们现在已经有一个完整的自动测试用例 1 了。下面就是完整的代码：

```
class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    void init() {
        parts.deleteAll();
        mailLog.empty();
    }
    static ConferenceSystem getInstance() {
        ...
    }
}

class Participants {
    void deleteAll() {
        ...
    }
    void importFromFile(String path) {
        ...
```

```
    }
    boolean checkParticipantStored(Participant part) {
        return part.equals(getParticipantById(part.getId()));
    }
    Participant getParticipantById(String partId) {
        ...
    }
}

class Participant {
    boolean equals(Object obj) {
        ...
    }
}

class MailRecord {
    MailRecord(String recipientEmail, String mailContent) {
        ...
    }
    boolean equals(Object obj) {
        ...
    }
}

class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
}

class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
           new SystemInitCommand(),
           new CreateImportFileCommand(...),
           new ImportParticipantsFromFileCommand(),
           new CheckParticipantStoredCommand("p001","123456", "Mary Lam", ...),
           new CheckParticipantStoredCommand("p002", ...),
           new CheckParticipantStoredCommand("p004", ...),
           new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id& ..."),
           new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ..."),
           new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ..."),
```

```
            ...
        };
        runCommands(commands);
    }
    static void testImportWithoutAddress() {
        ...
    }
    static void testImportWithoutOtherInfo() {
        ...
    }
    static void testImportDupId() {
        ...
    }
    static void runCommands(Command commands[]) {
        for (int i = 0; i < commands.length; i++) {
            if (!commands[i].run()) {
                System.out.println("Test failed!");
            }
        }
    }
}

interface Command {
    boolean run();
}

class SystemInitCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().init();
        return true;
    }
}

class CreateImportFileCommand implements Command {
    String fileContent;
    CreateImportFileCommand(String fileContent) {
        this.fileContent = fileContent;
    }
    boolean run() {
        FileWriter fileWriter = new FileWriter("sourceFile.txt");
        try {
            fileWriter.write(fileContent);
            return true;
```

```
        } finally {
            fileWriter.close();
        }
    }
}

class ImportParticipantsFromFileCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
        return true;
    }
}

class CheckParticipantStoredCommand implements Command {
    Participant part;
    CheckParticipantStoredCommand(String partId, String password, ...) {
        part = new Participant(partId, password, ...);
    }
    boolean run() {
        return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
    }
}

class CheckRemoveOldestMailCommand implements Command {
    MailRecord mailRecord;
    CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
        mailRecord = new MailRecord(recipientEmail, mailContent);
    }
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailRecord.equals(mailLog.takeOldestMailRecord());
    }
}
```

Command 不应该包括域逻辑

上面所有的 XXXCommand 的类都有一个共同点：它们都只有一点点的代码。这可不是偶然的。因为它们是在模拟用户在系统上的一些操作或者说一些请求（比如，要求系统导入一个文件）。而系统已经有代码实现了这些请求，所以所有的 Commands 只是简单的调用系统已经存在的代码。比如，当我们在写 CheckParticipantStoredCommand，Participants 这个类没有 getParticipantById 这个方法，可是我们却需要用到，就直接在 CheckParticipantStoredCommand 里面加了这样的代码：

```
class CheckParticipantStoredCommand implements Command {
```

```
    Participant part;
    ..
    boolean run() {
        return part.equals(getParticipantById(part.getId()));
    }
    Participant getParticipantById(String partId) {
        //一堆代码来从数据库里面取出数据。
        ...
        ...
        ...
    }
}
```

而之后，我们在实现用户例事的时候，却又发现这个方法刚好可以用到。于是就直接拿来用了。也就是说，现在我们的“正式”代码里面调用了测试的代码。这样，可是一个大问题。因为 CheckParticipantStoredCommand 只是用来测试的，正式发布的时候这些测试的代码都要排除。可是我们这边却又想留着 CheckParticipantStoredCommand 这个类，因为这个方法 getParticipantById 很有用。真是两难啊。这种情况下应该怎么做？把 getParticipantById 放在“正式”代码里面就行了！

所有的 XXXCommand 这样的测试类，都不应该包含域逻辑代码。域逻辑只能在“正式”代码里面实现，比如 Participants，Participant 这样的类。测试代码可以调用“正式”代码，而“正式”代码绝对不能调用测试代码。如果你觉得测试类里面的一些代码有用，那么你就将那些代码移到“正式”代码里面。

先写测试代码，作为需求文档

假想在实现“导入参会者信息”这个用户例事时，我们是跟客户一起写了这个这四个自动测试用例。然后运行一下这些用例看能不能通过。如果可以的话，证明我们的用户例事已经实现了（不过最好还是在客户面前运行一遍）。不过，假设我们在下面这样的用例（这个用例是客户提供的）中失败了：

```
p001    123456      Mary Lam     abc    mary@extremely.long.domain.com
p004    888999      John Chan    def    john@yahoo.com
p002    mypasswd    Paul Lei     ghi    paul@excite.com
```

通过 2 个小时的 debug，我们终于发现，我们在数据库里面定义“邮箱”这个字段的长度为 26 个字符，而 Mary 的邮箱有 32 个字符。因此，数据库只保存了“mary@extremely.long.doma”，而不是“mary@extremely.long.domain.com”。因此，在取 p001 的信息时，checkParticipantStored 这个方法失败了。

如果在写“正式”代码之前，我们先跟客户一起写了测试代码，我们就会知道至少需要 32 个字符的长度来存储邮箱地址。创建数据库的时候就可以避免这个错误。我们也就省了这两个小时的 debug 时间。

因此，测试代码应该在实现用户例事之前写。测试用例是我们软件需求的一个重要组成部分（现场客户是另一个重要部分）。作为需求的一部分，测试用例可以指导我们实现用户例事。而现场客户的优点就是，最新的信

息，更有效的沟通（跟人对话总比看文档更有效果）；而测试用例的优点则是更准确，自动化的测试则可以很频繁的运行。

让客户看得懂测试用例

只有客户知道需求，所以只有客户最能够判断我们的测试用例是否准确。我们能做的，就是帮助他们将需求写成测试用例，绝对不能自做主张。可是，现在我们的代码，非开发人员的客户根据就看不懂：

```
class AcceptTestForImportParticipants {
   static void testImport() {
       Command commands[] = {
           new SystemInitCommand(),
           new CreateImportFileCommand(...),
           new ImportParticipantsFromFileCommand(),
           new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
           new CheckParticipantStoredCommand("p002", ...),
           new CheckParticipantStoredCommand("p004", ...),
           new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id& ..."),
           new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ..."),
           new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ..."),
       };
       runCommands(commands);
   }
}
```

这是一个很严重的问题。如果客户看不懂测试用例，他怎么准确的判断这个是不是刚好他想要的？为了能让他看懂，我们先在一个文本文件里面写下这样的测试用例吧，顺便给这个文本文件取名为 testImport.test 吧：

```
SystemInit
CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,def,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,def,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...
```

我们管这个叫“测试文件”。测试文件里面，基本上每个 command 都占一行，不过不同的是 CreateImportFile 占了 5 行：

```
CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,def,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
```

从客户的角度，系统可以运行这个“testImport.test”文件。或者说，系统应该可以执行这个文件里面的每一条 command。

为了支持这种测试文件，我们就要做一个可以解析这种测试文件，然后生成相应的 command 的类：

```
class TestFileParser {
   CommandParser commandParsers[] = {
       new SystemInitCommandParser(),
       new CheckParticipantStoredCommandParser(),
       new CreateImportFileCommandParser(),
       ...
   };
   Command[] parseCommandsFromFile(String pathToTestFile) {
       TokenIterator tokenIterator = parseFileToTokens(pathToTestFile);
       return parseCommands(tokenIterator);
   }
   TokenIterator parseFileToTokens(String path) {
       ...
   }
   Command[] parseCommands(TokenIterator tokenIterator) {
       Command commands[];
      while (tokenIterator.hasToken()) {
          Command nextCommand = parseCommand(tokenIterator);
          append nextCommand to commands;
      }
      return commands;
   }
   Command parseCommand(TokenIterator tokenIterator) {
      for (int i = 0; i < commandParsers.length; i++) {
          Command command = commandParsers[i].parse(tokenIterator);
          if (command != null) {
              return command;
          }
      }
```

```
            throw new CommandSyntaxException();
        }
    }

    interface TokenIterator {
        boolean hasToken();
        String peekToken();
        String takeToken();
        void next();
    }

    interface CommandParser {
        Command parse(TokenIterator tokenIterator);
    }

    class SystemInitCommandParser implements CommandParser {
        Command parse(TokenIterator tokenIterator) {
            if (tokenIterator.peekToken().equals("SystemInit")) {
                tokenIterator.next();
                return new SystemInitCommand();
            } else {
                return null;
            }
        }
    }

    class CheckParticipantStoredCommandParser implements CommandParser {
        Command parse(TokenIterator tokenIterator) {
            if (tokenIterator.peekToken().equals("CheckParticipantStored")) {
                tokenIterator.next();
                String partId = tokenIterator.takeToken();
                String password = tokenIterator.takeToken();
                String name = tokenIterator.takeToken();
                String address = tokenIterator.takeToken();
                String email = tokenIterator.takeToken();
                return new CheckParticipantStoredCommand(
                    partId,
                    password,
                    name,
                    address,
                    email);
            } else {
                return null;
```

```
        }
    }
}

class CreateImportFileCommandParser implements CommandParser {
    ...
}
```

测试文件不一定是文本文件

测试文件不一定是个文本文件。比如，它可以是个 excel 文件，也可以用 HTML 或者其他格式的文件作测试文件。只要我们能够解析出来就行了。

用测试用例防止系统走下坡路

众所周知，如果没有完整的测试，系统的错误，会随着开发的进行逐渐增多。测试用例的一个好处，就是可以防止这种情况。

每次我们跟客户一起建了一个测试文件，我们都把它放在一个名为“测试队列”的文件夹里。开始的时候，所有的测试文件都放在这边。每运行通过一个测试文件，我们就将这个测试文件放到另外一个叫“通过”的文件夹里。之后不管是做重构，还是增加新的功能，或者修改代码以后，都要将“通过”这个文件夹里面的所有测试文件跑一遍。如果有测试文件通不过，那证明代码出错了，我们要马上修改代码，让它通过。而不是将通不过的测试文件放回到“测试队列”的文件夹里。这么说吧，除非是测试文件本身出问题了，否则，绝对不能把测试文件从“通过”放回到“测试队列”中去。

当最终“测试队列”里面的文件都移到“通过”队列后，系统完成了。

引述

http://c2.com/cgi/wiki?AcceptanceTest .

Ward Cunningham 写了一个叫“Fit”的验收测试框架。可以在 http://fit.c2.com 找到。

FitNesse 是一个基于 Web 的以“Fit”为构架做起来的测试环境，让开发人员直接创建和运行验收测试 http://fitnesse.org。

章节练习

1. 将上面讲的测试用命 2，3，4 都写成测试文件，并实现所有需要的 Commands（你可以假设系统已经包含需要的

功能代码)。

解决方法示例

测试用例 2 的测试文件：

```
SystemInit
CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...
```

测试用例 3 的测试文件：

```
SystemInit
CreateImportFile
    ,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,,def,john@yahoo.com
    p002,,Paul Lei,ghi,paul@excite.com
    p005,secret,Mike Chan,jkl,
CreateImportFileEnd
ImportFromFile
CheckNoParticipantsStored
CheckMailLogEmpty
```

我们要建两个新的 Command：CheckNoParticipantsStored 和 CheckMailLogEmpty.

```
class CheckNoParticipantsStoredCommand implements Command {
    boolean run() {
        Participants parts = ConferenceSystem.getInstance().parts;
        return parts.isEmpty();
    }
}

class CheckMailLogEmptyCommand implements Command {
```

```
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailLog.isEmpty();
    }
}
```

测试用例 4 的测试文件：

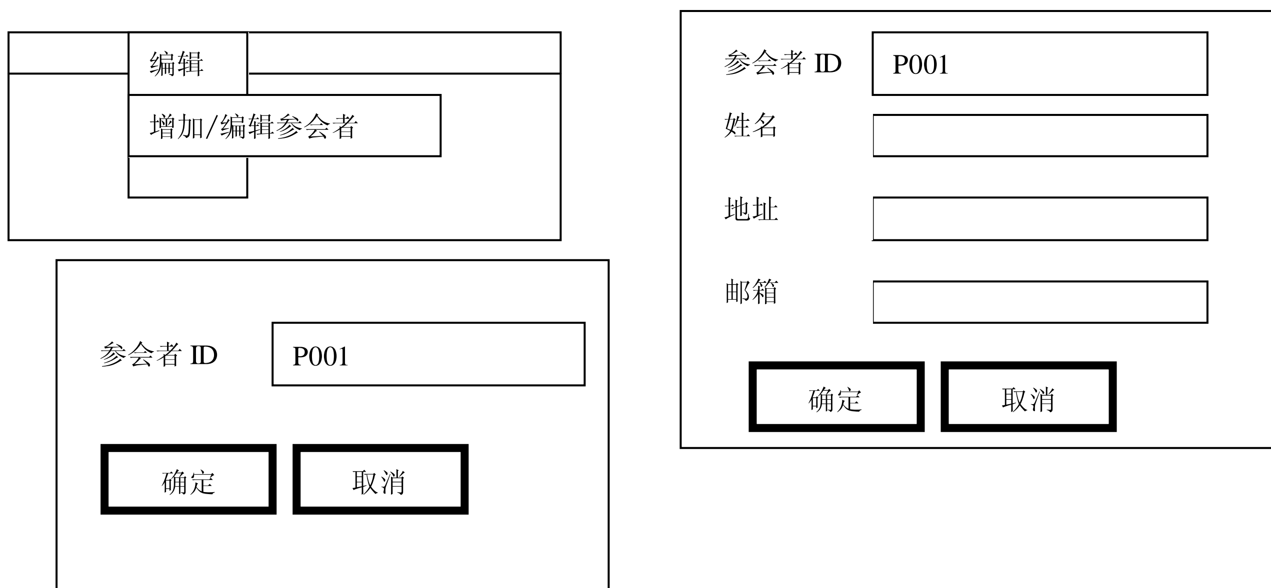
```
SystemInit
CreateImportFile
   p001,123456,Mary Lam,abc,mary@hotmail.com
   p001,888999,John Chan,def,john@yahoo.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
```

第 11 章 对 UI 进行验收测试

怎么操作 UI

假设客户要求我们实现下面的用户例事（user story）：
名称：增加或者编辑一条参会者信息
事件：
1.用户输入一个参会者的 ID。
2.如果这是一个新的参会者 ID，用户为这个新的参会者输入姓名，地址和该参会者的邮箱。
3.如果这是一个已有参会者的 ID，系统会调出该参会者的姓名，地址，邮箱，显示出来让用户编辑。
4.系统保存用户修改后的参会者的信息。之后，用户输入这个 ID 的话，系统会取出这个参会者的新信息。

经过讨论，客户接受我们提供的这样的方案：
用户点击主窗口里面的菜单“增加/编辑参会者”，系统会弹出一个对话框，用户在这个对话框里面输入参会者的 ID，点击“确定”，系统会调出另外一个对话框让用户输入或者编辑参会者的信息（ID 不能编辑）。然后用户点“确定”，系统保存这些信息。两个弹出对话框都有一个“取消”按钮。下面是 UI 的草图：



问题是，怎么测试 UI 的功能？

分开测试每个 UI 组件

上面的用户例事中有 3 个 UI 组件：

1. 菜单“增加/编辑参会者”。
2. 让用户输入参会者 ID 的弹出窗口。我们叫它 ParticipantIdDialog。
3. 让用户输入或者编辑参会者信息的窗口，我们叫它 ParticipantDetailsDialog。

通常很难把几个 UI 组件放在一起测试（为什么呢？唉，这很难解释。因此，请相信我。如果不相信的话，你们可以自己试试看）。因此，我们通常会传开测试很个 UI 组件。

因为这里面最复杂的组件是 ParticipantDetailsDialog，我们就来看一下要怎么测试这个组件。

如何测试 ParticipantDetailsDialog

我们可以用下面的测试用例来测试：初始化系统，给 ParticipantDetailsDialog 提供一个新的参会者 ID，接着输入姓名，地址，邮箱，点击确定，看看系统会不会保存这条信息。

下面是测试文件：

```
SystemInit
ShowParticipantDetailsDialog,p001
SetParticpantName,Mike Chan
```

```
SetParticpantPassword,888888
SetParticpantAddress,aaabbb
SetParticpantEmail,mike@excite.com
ClickOK
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

这里面 ShowParticipantDetailsDialog 命令创建一个 ParticipantDetailsDialog，然后提供 p001 作为参会者 ID。SetParticipantName 则模拟用户输入姓名。类似的 SetParticipantPassword，SetParticipantAddress 和 SetParticipantEmail 模拟用户输入密码，地址和邮箱。ClickOK 模拟用户点击“确定”按钮。

不过，这里面有个问题。比如，SetParticipantName 这条命令的方法里面，它怎么得到 ParticipantDetailsDialog 的实例？我们的系统里面并没有一个全局变量让它引用到。同样的，ParticipantDetailsDialog 是由 ShowParticipantDetailsDialog 命令动态创建的，ClickOK 也引用不到。

因此，为了可以让 SetParticipantName 和 ClickOK 可以引用到 ShowParticipantDetailsDialog 创建的对话框，我们就在这个测试文件里面放一个变量，比如：

```
SystemInit
dialog=ShowParticipantDetailsDialog,p001
SetParticpantName,dialog,Mike Chan
SetParticpantPassword,dialog,888888
SetParticpantAddress,dialog,aaabbb
SetParticpantEmail,dialog,mike@excite.com
ClickOK,dialog
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

不过，这样的话，这个测试文件就有点像编程语言了。一个可行的解决方法是，将 SetParticipantName 变成 ShowParticipantDetailsDialog 的子命令，比如：

```
SystemInit
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

从 ShowParticipantDetailsDialogStart 到 ShowParticipantDetailsDialogEnd 这里面的 7 行文字中，其实只定义了一条命令：ShowParticipantDetailsDialog。里面的 5 行每行都是一个子命令。ShowParticipantDetailsDialog 在运行的时候，会创一个 ParticipantDetailsDialog 对象，然后让它的所有 5

条子命令使用这个 ParticipantDetailsDialog。

下面就是这些命令：

```
class ShowParticipantDetailsDialogCommand implements Command {
   String partId;
   SubCommand subCommands[];
   ShowParticipantDetailsDialogCommand(String partId) {
       this.partId=partId;
   }
   void addSubCommand(SubCommand subCommand) {
       //将 subCommand 增加到 subCommands
       ...
   }
   boolean run() {
       ParticipantDetailsDialog partDetailsDlg=
           new ParticipantDetailsDialog(partId);
       //不过调用 dialog 里面的 showModal 方法，否则它真会显示在屏幕上，然后等待用户的交互。
       //然后等到用户关掉了这窗口之后，下面的代码才会继续运行。
       For (int i=0; i<subCommands.length; i++) {
           subCommands[i].setDialogToUse(partDetailsDlg);
           if (!subCommands[i].run()) {
              return false;
           }
       }
       return true;
   }
   static abstract class SubCommand implements Command {
       ParticipantDetailsDialog partDetailsDlg;
       void setDialogToUse(ParticipantDetailsDialog partDetailsDlg) {
           this.partDetailsDlg=partDetailsDlg;
       }
   }
   static class SetParticipantNameCommand extends SubCommand {
       String partName;
       SetParticipantNameCommand(String partName) {
           this.partName=partName;
       }
       boolean run() {
           partDetailsDlg.setParticipantName(partName);
           return true;
       }
   }
```

```
    static class ClickOKCommand extends SubCommand {
        boolean run() {
            partDetailsDlg.clickOK();
            return true;
        }
    }
}

class ParticipantDetailsDialog extends Jdialog {
    String partId;
    JtextField partName;
    JtextField partEmail;
    ...
    JButton OK;
    Jbutton cancel;
    ParticipantDetailsDialog(String partId) {
        this.partId=partId;
    }
    void setParticipantName(String name) {
        partName.setText(name);
    }
    void setParticipantPassword(String password) { ... }
    void setParticipantEmail(String email) { ... }
    void setParticipantAddress(String address) { ... }
    void clickOK() { ... }
    void clickCancel() { ... }
}
```

注意到 ShowParticipantDetailsDialog 创建了一个 ParticipantDetailsDialog，但却没调用 showModal。

测试其他情况

我们已经测试了 ParticipantDetailsDialog 处理一个新参会者信息的情形。我们还要测试当用户输入的参会者 ID 已经存在或者用户点了“取消”的情况。然后，我们还要测试 ParticipantIdDialog。因为菜单项太简单了，我们不测它。

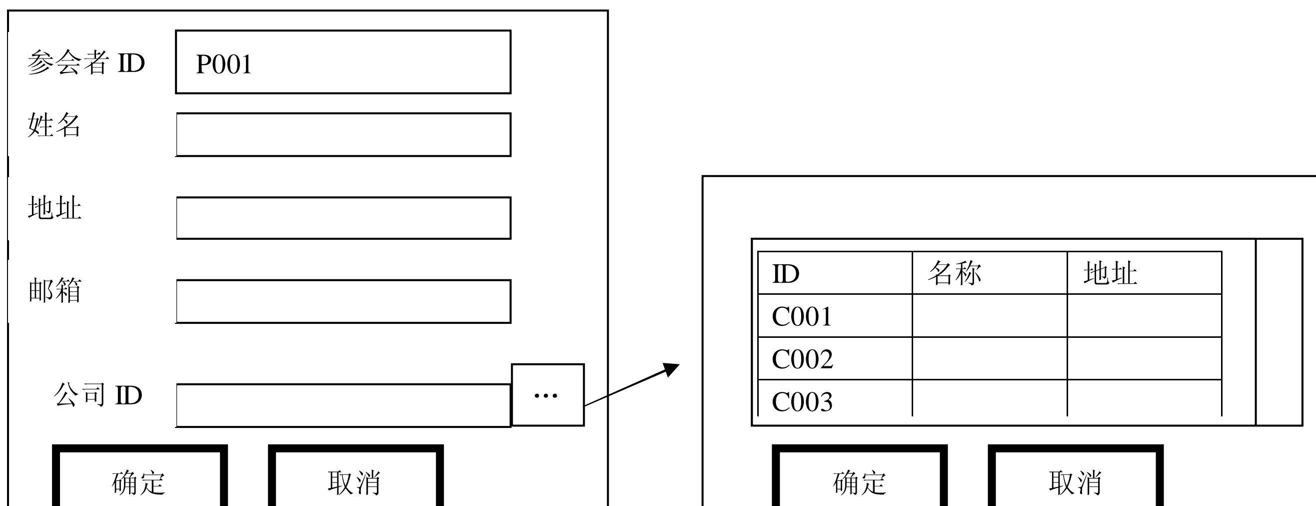
不过测试这些情况的方法一样，我们这边就不提及了。

如果要测试的对话框会弹出另一个对话框怎么办？

经常我们会在一个对话框弹出另一个对话框。比如，ParticipantDetailsDialog 还要求用户输入该参会者的

公司 ID。用户只要点击公司 ID 的文本框右边的一个“...”的按钮，就会弹出一个对话框（我们叫它 CompaniesDialog)，显示出系统里面的所有公司。用户选择其中一个，保存。系统就是自动把用户选择的公司的 ID 放在公司 ID 文本框中。

下面是个草图：



问题是，选择公司的功能怎么测试？下面的测试文件行吗？

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ShowCompaniesDialog
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

这里，为了可以有一些公司来让用户选择，我们又增加了两条 AddCompany 命令，来增加两个公司到系统里面（相应的 ID 是 c001 和 c002）。ShowCompaniesDialog 会弹出 CompaniesDialog。

不过 ShowCompaniesDialog 这条命令不工作。因为我们还需要跟 CompaniesDialog 交互，比如选择一个公司以后，点“确定”。因此，我们要改成：

```
SystemInit
AddCompany,c001,...
```

```
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
   SetParticpantName,Mike Chan
   SetParticpantPassword,888888
   SetParticpantAddress,aaabbb
   SetParticpantEmail,mike@excite.com
   ShowCompaniesDialogStart
       SelectRowInGrid,1
       ClickOK
   ShowCompaniesDialogEnd
   GetCompanyId,c002
   ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

这里面我们用了 SelectRowInGrid 这条命令，来选择公司列表里面的第二行（假定第一行的 index 的 0），对应就是 c002.然后我们用 ClickOK 的命令来点确定。ParticipantDetailsDialog 命令里面，我们用了一个 GetCompanyId 的子命令来检查 c002 是不是已经自动变为公司的 ID。

这种方案就可行了。不过，我们是在同时测试 ParticipantDetailsDialog 和 CompaniesDialog。之前提过，分开测试每个 UI 组件会更简单。因此，我们希望 ParticipantDetailsDialog 不会使用另一个对话框。那它真的需要 CompaniesDialog 吗？其实，它真正需要的，只是从某个类里面取得公司 ID。我们可以将 CompaniesDialog 抽象成一个提供字符串的类，或者一个字符串源，然后让 ParticipantDetailsDialog 使用这个字符串源：

```
interface StringSource {
   String getString();
}

class CompaniesDialog extends Jdialog implements StringSource {
   String getString() {
       showModal();
       return OKClicked() ? getSelectedCompanyId() : null;
   }
   boolean OKClicked() { ... }
   String getSelectedCompanyId() { ... }
}

class ParticipantDetailsDialog extends Jdialog {
   String partId;
   StringSource companyIdSource;
   JtextField partName;
   JtextField partEmail;
   JtextField compId;
```

```
    ...
    JButton OK;
    Jbutton cancel;
    Jbutton chooseCompany;
    ParticipantDetailsDialog(String partId) {
        this.partId=partId;
        this.companyIdSource=new CompaniesDialog();
    }
    void clickChooseCompany() {
        String companyId=companyIdSource.getString();
        if (companyId!=null) {
            compId.setText(companyId);
        }
    }
}
```

原来的测试文件可以分为两个。一个测试文件是 ParticipantDetailsDialog：

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowParticipantDetailsDialogStart,p001
    SetParticpantName,Mike Chan
    SetParticpantPassword,888888
    SetParticpantAddress,aaabbb
    SetParticpantEmail,mike@excite.com
    ClickChooseCompany,c002
    GetCompanyId,c002
    ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,...
```

另一个是：

```
SystemInit
AddCompany,c001,...
AddCompany,c002,...
ShowCompaniesDialogStart
    SelectRowInGrid,1
    ClickOK
    GetSelectedCompanyId,c002
ShowCompaniesDialogEnd
```

ClickChooseCompany 这条子命令可以以下面方式实现：

```
class ShowParticipantDetailsDialogCommand implements Command {
   ...
   static class ClickChooseCompanyCommand extends SubCommand {
       String companyId;
       ClickChooseCompanyCommand(String companyId) {
           this.companyId=companyId;
       }
       boolean run() {
           partDetailsDlg.companyIdSource=new StringSource() {
              String getString() {
                  return companyId;
              }
           });
           partDetailsDlg.clickChooseCompany();
           return true;
       }
   }
}
```

引述

http://c2.com/cgi/wiki?GuiTesting.

章节练习

问题

1. 写一个测试文件，测试 ParticipantDetailsDialog 能不能处理用户点击“取消”的情况。实现需要的命令。
2. 写一个测试文件，用来测试 ParticipantDetailsDialog 能不能处理用户输入的参会者 ID 已经存在的情况。实现相应的命令。
3. 实现 ParticipantDetailsDialog 里面的 setParticipantAddress。
4. 写一些测试文件来测试 ParticipantIdDialog。实现相应的命令。

提示

1，2，3. 没有提示。

4. 应该有一个子命令 GetParticipantId 来检查用户输入的参会者 ID 是否存在。

解决方法示例

1. 写一个测试文件，测试 ParticipantDetailsDialog 能不能处理用户点击“取消”的情况。实现需要的命令。

```
SystemInit
ShowParticipantDetailsDialogStart,p001
    SetParticpantName,Mike Chan
    SetParticpantPassword,888888
    SetParticpantAddress,aaabbb
    SetParticpantEmail,mike@excite.com
    ClickCancel
ShowParticipantDetailsDialogEnd
CheckNoParticipantsStored
```

不用实现什么命令。

2. 写一个测试文件，用来测试 ParticipantDetailsDialog 能不能处理用户输入的参会者 ID 已经存在的情况。实现相应的命令。

```
SystemInit
AddParticipant,p001,Mary Lam,123456,abc,mary@hotmail.com
AddParticipant,p004,John Chan,888999,def,john@yahoo.com
ShowParticipantDetailsDialogStart,p001
    GetParticpantName,Mary Lam
    GetParticpantPassword,123456
    GetParticpantAddress,abc
    GetParticpantEmail,mary@hotmail.com
    SetParticpantName,Mike Chan
    SetParticpantPassword,888888
    SetParticpantAddress,aaabbb
    SetParticpantEmail,mike@excite.com
    ClickOK
ShowParticipantDetailsDialogEnd
CheckParticipantStored,p001,888888,Mike Chan,aaabbb,mike@excite.com
```

实现的命令:

```
class AddParticipantCommand implements Command {
    Participant part;
    AddParticipantCommand(Participant part) {
        this.part=part;
    }
    boolean run() {
        ConferenceSystem.getInstance().parts.addParticipant(part);
```

```
        return true;
    }
}

class ShowParticipantDetailsDialogCommand implements Command {
    ...
    static class GetParticipantNameCommand extends SubCommand {
        String partName;
        GetParticipantNameCommand(String partName) {
            this.partName=partName;
        }
        boolean run() {
            return partDetailsDlg.getParticipantName.equals(partName);
        }
    }
    static class GetParticipantPasswordCommand extends SubCommand {
        ...
    }
    static class GetParticipantAddressCommand extends SubCommand {
        ...
    }
    static class GetParticipantEmailCommand extends SubCommand {
        ...
    }
}

class ParticipantDetailsDialog extends JDialog {
    ...
    JTextField partName;
    ...
    String getParticipantName() {
        return partName.getText();
    }
}
```

3.实现 ParticipantDetailsDialog 里面的 setParticipantAddress。

```
class ShowParticipantDetailsDialogCommand implements Command {
    ...
    static class SetParticipantAddressCommand extends SubCommand {
        String partAddress;
        SetParticipantAddressCommand(String partAddress) {
            this.partAddress=partAddress;
```

```
        }
        boolean run() {
            partDetailsDlg.setParticipantAddress(partAddress);
            return true;
        }
    }
    static class ClickOKCommand extends SubCommand {
        boolean run() {
            partDetailsDlg.clickOK();
            return true;
        }
    }
}

class ParticipantDetailsDialog extends JDialog {
    ...
    JTextField partAddress;
    ...
    void setParticipantAddress(String address) {
        partAddress.setText(address);
    }
}
```

4. 写一些测试文件来测试 ParticipantIdDialog。实现相应的命令。

测试它能不能返回用户输入的参会者 ID：

```
SystemInit
ShowParticipantIDDialogStart
   SetParticpantID,p001
   ClickOK
   GetParticpantID,p001
ShowParticipantIDDialogEnd
```

测试在用户点了 Cancel 的情况下，它能不能返回空值：

```
SystemInit
ShowParticipantIDDialogStart
   SetParticpantID,p001
   ClickCancel
   GetParticpantID,
ShowParticipantIDDialogEnd
```

实现的命令：

```
class ShowParticipantIDDialogCommand implements Command {
   ...
   static abstract class SubCommand implements Command {
      ...
   }
   boolean run() {
      ParticipantIDDialog partIDDlg=new ParticipantIDDialog();
      for (int i=0; i<subCommands.length; i++) {
          subCommands[i].setDialogToUse(partIDDlg);
          if (!subCommands[i].run()) {
             return false;
          }
      }
      return true;
   }
   static class SetParticipantIDCommand extends SubCommand {
      String partID;
      SetParticipantIDCommand(String partID) {
          this.partID=partID;
      }
      boolean run() {
          partIDDlg.setParticipantID(partID);
          return true;
      }
   }
   static class GetParticipantIDCommand extends SubCommand {
      String partID;
      GetParticipantIDCommand(String partID) {
          this.partID=partID;
      }
      boolean run() {
          return partIDDlg.getParticipantID().equals(partID);
      }
   }
   static class ClickOKCommand extends SubCommand {
      boolean run() {
         partIDDlg.clickOK();
         return true;
      }
   }
   static class ClickCancelCommand extends SubCommand {
      boolean run() {
```

```
            partIDDlg.clickCancel();
            return true;
        }
    }
}

class ParticipantIDDialog extends JDialog {
    ...
    JTextField partID;
    ...
    void setParticipantID(String id) {
        partID.setText(id);
    }
    void clickOK() { ... }
    void clickCancel() { ... }
}
```

第 12 章　单元测试

单元测试

假定你在写一个 CourseCatalog 类，这个类用来记录一些课程的信息：

```
class CourseCatalog {
    CourseCatalog() {
        ...
    }
    void add(Course course) {
        ...
    }
    void remove(Course course) {
        ...
    }
    Course findCourseWithId(String id) {
        ...
    }
    Course[] findCoursesWithTitle(String title) {
        ...
    }
```

```
}

class Course {
   Course(String id, String title, ...) {
   }
   String getId() {
      ...
   }
   String getTitle() {
      ...
   }
}
```

为了保证 CourseCatalog 这个类没有 bug，我们应该测试它。比如，为了知道它的 add 方法是不是真的能够将一个 Course 加进去，我们用下面的代码来测试：

```
class TestCourseCatalog {
   static void testAdd() {
      CourseCatalog cat = new CourseCatalog();
      Course course = new Course("c001", "Java prgoramming", ...);
      cat.add(course);
      if (!cat.findCourseWithId(course.getId()).equals(course)) {
         throw new TestFailException();
      }
   }
   public static void main(String args[]) {
      testAdd();
   }
}
```

因为 testAdd 方法里面调用了一个 equal 方法，来比较两个 Course 对象是不是相等的，所以 Course 类就要提供一个 equals 方法：

```
class Course {
   ...
   boolean equals(Object obj) {
      ...
   }
}
```

可能我们还是对 add 的方法没什么信心。我们还担心它不能存储 2 个以上的课程。因此，我们又写了这样一个测试：

```
class TestCourseCatalog {
   static void testAdd() {
       ...
   }
   static void testAddTwo() {
       CourseCatalog cat = new CourseCatalog();
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "OO design", ...);
       cat.add(course1);
       cat.add(course2);
       if (!cat.findCourseWithId(course1.getId()).equals(course1)) {
           throw new TestFailException();
       }
       if (!cat.findCourseWithId(course2.getId()).equals(course2)) {
           throw new TestFailException();
       }
   }
   public static void main(String args[]) {
       testAdd();
       testAddTwo();
   }
}
```

类似的，我们可以测试 CourseCatalog 里面的 remove 方法：

```
class TestCourseCatalog {
   static void testAdd() {
       ...
   }
   static void testAddTwo() {
       ...
   }
   static void testRemove() {
       CourseCatalog cat = new CourseCatalog();
       Course course = new Course("c001", "Java prgoramming", ...);
       cat.add(course);
       cat.remove(course);
       if (cat.findCourseWithId(course.getId()) != null) {
           throw new TestFailException();
       }
   }
   public static void main(String args[]) {
       testAdd();
```

```
        testAddTwo();
        testRemove();
    }
}
```

上面的 testAdd，testAddTwo，和 testRemove 的方法，是用来测试 CourseCatalog 类。每个方法都是一个测试用例。不过，它们并不叫验收测试，而叫“单元测试”。因为它们只测一个单元（CourseCatalog 类）。

单元测试跟验收测试有什么区别？验收测试测试的是系统的外部行为，而单元测试是测试系统内部结构，它只测一个单元（类，甚至一个方法）。验收测试属于客户的，我们没有权利决定验收测试的内容。我们顶多只是帮忙客户根据用户例事写出验收测试。单元测试属于我们，因为系统里面有什么类，每个类都做什么，是由我们决定的。客户就没有权利涉及了，而且我们也不需要他的参与。我们只是根据我们对这个单元（类）的期望写出单元测试。因此，这种测试又叫“程序员测试”。

使用 JUnit

写单元测试的时候，我们可以使用一个开源包叫“JUnit”。在 JUnit 下，我们可以将上面的测试代码改成这样：

```
import junit.framework.*;
import junit.swingui.*;
public class TestCourseCatalog extends TestCase {
   CourseCatalog cat;
   protected void setUp() {
       cat = new CourseCatalog();
   }
   public void testAdd() {
       Course course = new Course("c001", "Java prgoramming", ...);
       cat.add(course);
       assertEquals(cat.findCourseWithId(course.getId()), course);
   }
   public void testAddTwo() {
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "OO design", ...);
       cat.add(course1);
       cat.add(course2);
       assertEquals(cat.findCourseWithId(course1.getId()), course1);
       assertEquals(cat.findCourseWithId(course2.getId()), course2);
   }
   public void testRemove() {
```

```
        Course course = new Course("c001", "Java prgoramming", ...);
        cat.add(course);
        cat.remove(course);
        assertTrue(cat.findCourseWithId(course.getId()) == null);
    }
    public static void main(String args[]) {
        TestRunner.run(TestCourseCatalog.class);
    }
}
```

下面我们会解释这些改动。

assertEquals(X, Y)方法检查 X 跟 Y 是不是相等（它会调用 X 的 equals 方法）。如果不相等的话，它会抛出一个异常。assertEquals 这个方法是由 TestCase 类提供的。因为我们的 TestCourseCatalog 继承自 TestCase，我们可以直接调用 assertEquals。

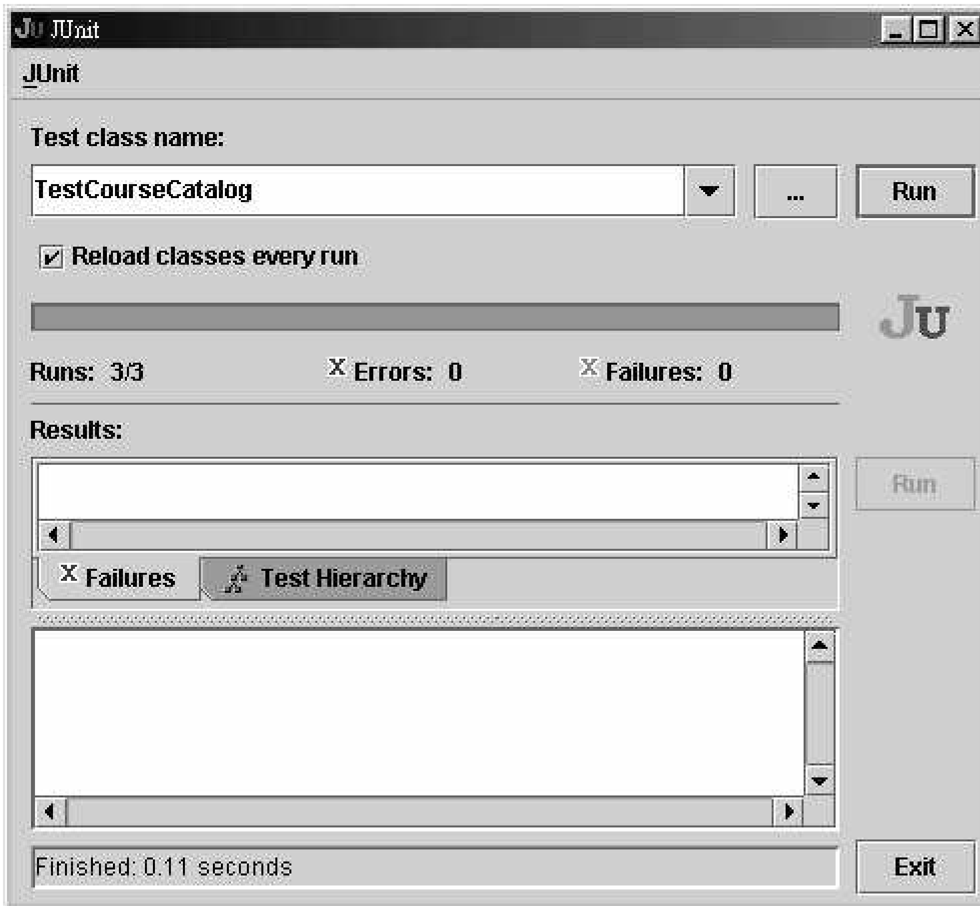
TestCase 是由 JUnit 提供的。在 junit.framework 包里。因此，我们需要导入 junit.framework.*包。

testAdd，testAddTwo 和 testRemove 都要创建一个 CourseCatalog 对象。我们要把创建对象的代码放在 setUp 方法里。每个测试用例运行的时候，都会调用 setUp 方法。

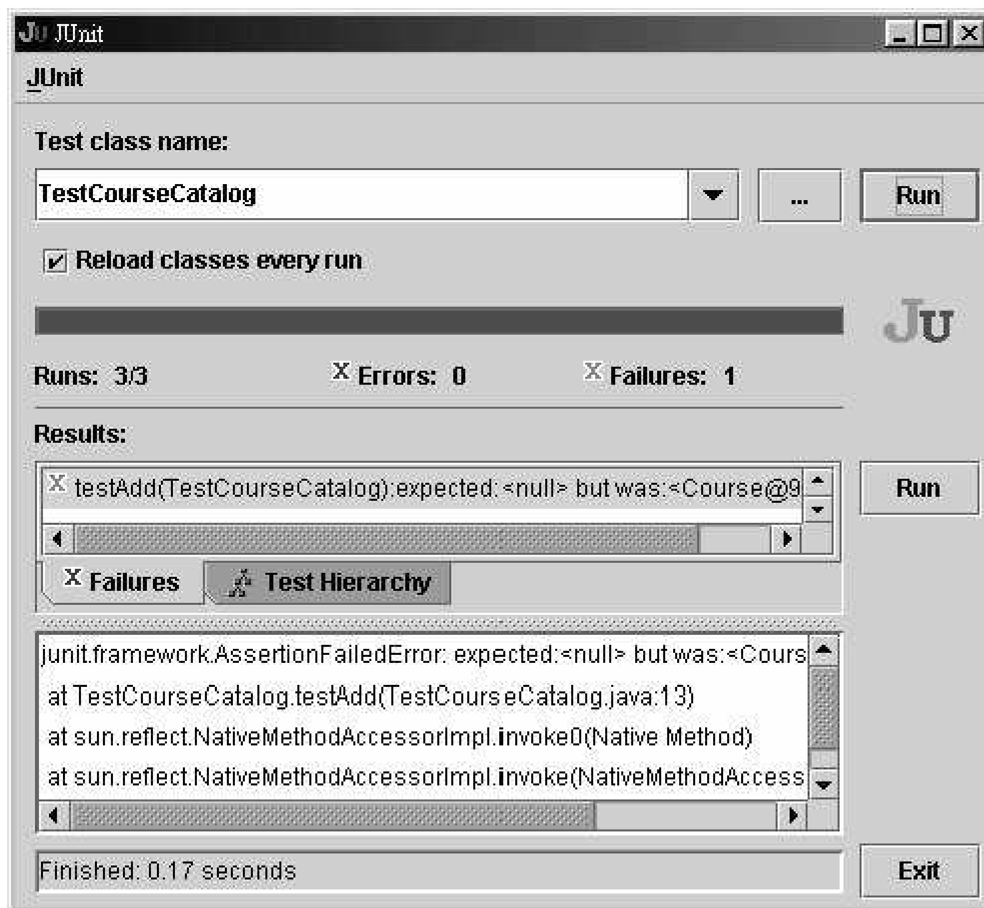
因为 cat 这个变量是在 setUp 方法里面初始化的，不过它还被 testAdd 等等方法用到，所以它应该是个属性，而不仅仅是个本地变量。此外，现在所有的测试用例的方法，比如 testAdd 等等都是实例方法（原来都是静态方法）。

testRemove 用到了 assertTrue(X)，它用来检查 X 是不是 true 值。如果不是的话，就会抛出异常，就像 assertEquals 一样。它了是 TestCase 类提供的。

main 方法调用了 TestRunner 的静态方法 run，它会显示一个窗口，然后运行这 3 个测试用例：testAdd，testAddTwo 和 testRemove。如果 3 个测试用例都通过的话，它会显示一个绿色的进度条：



。如果有哪个测试用例失败了，它会显示一个红色进度条，并说明源文件里的哪行代码出现错误了：



TestRunner 也是 JUnit 提供的。在 junit.swingui 包里。

TestRunner 是怎么知道我们定义了哪些测试用例的（testAdd 等等）？我们调用的 TestRunner.run(TestCourseCatalog.class)，它利用了 Java 的反射，找出 TestCourseCatalog 中所有以“test”开头的无参数的 public 方法。它把每个这样的方法都当做一个测试用例。每个测试用例都是在不同的 TestCourseCatalog 对象里面运行的。比如，在运行 testAdd 这个测试用例的时候，它在是一个 TestCourseCatalog 实例中运行的，而在运行 testRemove 的时候，却是在 TestCourseCatalog 的另一个实例中运行的。就类似于下面这样：

```
TestCourseCatalog testCourseCatalogForTestAdd=new TestCourseCatalog();
testCourseCatalogForTestAdd.setUp();
testCourseCatalogForTestAdd.testAdd();
…
TestCourseCatalog testCourseCatalogForTestRemove=new TestCourseCatalog();
testCourseCatalogForTestRemove.setUp();
testCourseCatalogForTestRemove.testRemove();
```

另外，TestCourseCatalog 类必须定义为 public。

通常，有些 Test 类会定义 setUp 跟 tearDown 这两个方法，它们是做什么的呢？如果我们在 setUp 创建一个数据库连接，通常我们会在 tearDown 关掉连接。相似的，如果我们在 setUp 创建一个临时文件，通常会在 tearDown

里面删除它。

我们需要对所有的类进行单元测试吗？

基本上，我们应该测试系统里面每个类的每个方法，除非方法简单到压根儿不用测，比如 Course 类：

```
class Course {
   String id;
   String title;
   Course(String id, String title, ...) {
       this.id = id;
       this.title = title;
   }
   String getId() {
       return id;
   }
   String getTitle() {
       return title;
   }
   boolean equals(Object obj) {
       if (obj instanceof Course) {
           Course c = (Course)obj;
           return this.id.equals(c.id) && this.title.equals(c.title);
       }
       return false;
   }
}
```

因为构造函数，getId 和 getTitle 方法都太简单了，我们不需要测试。我们只需要测试 equals 方法就行了。

```
import junit.framework.*;
import junit.swingui.*;
public class TestCourse extends TestCase {
   public void testEquals() {
       Course course1 = new Course("c001", "Java prgoramming");
       Course course2 = new Course("c001", "Java prgoramming");
       Course course3 = new Course("c001", "OO design");
       Course course4 = new Course("c002", "Java prgoramming");
       assertEquals(course1, course2);
       assertTrue(!course1.equals(course3));
```

```
            assertTrue(!course1.equals(course4));
        }
        public static void main(String args[]) {
            TestRunner.run(TestCourse.class);
        }
    }
```

怎么运行所有的单元测试

现在系统里面有 4 个单元测试，其中三个是测试 CourseCatalog；一个是测试 Course。我们希望经常可以一次性就运行所有的单元测试。因此，可以这样做：

```
public class TestAll {
    public static void main(String args[]) {
        TestRunner.run(TestAll.class);
    }
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(TestCourseCatalog.class));
        suite.addTest(new TestSuite(TestCourse.class));
        return suite;
    }
}
```

suite 方法创建并返回一个 TestSuite 对象，里面包含系统里面所有的单元测试。我们可以将多个测试用例加到一个 TestSuite 里面。我们也可以将几个 TestSuites 加到另一个 TestSuite 里面。当 TestRunner 运行一个 TestSuite,它会运行这个 TestSuite 里面所有测试用例。main 方法里面调用了 TestRunner.run(TestAll.class)。之前提过的，它会调用这个类里面所有以“test”开头的没有参数的 public 方法。不过，在搜索这些方法之前，它会先检查 TestAll 里面有没有一个静态方法“suite”。如果有的话，它会调用这个 suite 方法，然后测试 suite 方法返回的 TestSuite 对象里面的测试用例，而不去搜索 testXXX 方法了。

我们如果创建了一个新类，比如 TimeTable，我们就要将相应的 TestTimeTable 加到 suite 里面。这点很容易忘记：

```
public class TestAll {
    public static void main(String args[]) {
        TestRunner.run(TestAll.class);
    }
    public static Test suite() {
        TestSuite suite = new TestSuite();
```

```
        suite.addTest(new TestSuite(TestCourseCatalog.class));
        suite.addTest(new TestSuite(TestCourse.class));
        suite.addTest(new TestSuite(TestTimeTable.class));
        return suite;
    }
}
```

什么时候运行单元测试

什么时候运行单元测试？比如

在我们写完，或者改完 CourseCatalog，我们应该运行 TestCourseCatalog 里面的测试用例。
在我们写完，或者改完 Course，我们应该运行 TestCourse 里面的测试用例。
在我们即将运行所有的验收测试前，我们应该先运行所有的测试用例。
在我们有时间的时候，我们应该运行所有的测试用例。

更新单元测试

假设我们在 CourseCatalog 里面增加了一个 getCount 方法。我们应该在 TestCourseCatalog 增加 1 个或者多个测试用例来测试它，比如：

```
public class TestCourseCatalog extends TestCase {
    CourseCatalog cat;
    ...
    public void testGetCountOnEmptyCatalog() {
        assertEquals(cat.getCount(), 0);
    }
    public void testGetCount() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "Java prgoramming", ...);
        cat.add(course1);
        cat.add(course2);
        assertEquals(cat.getCount(), 2);
    }
}
```

用单元测试来防止以后出现同样的 bug

当我们在代码里面找到了一个 bug，比如，CourseCatalog 里面的 remove(Course course)方法应该只测试这个 course 对象的，但它却把所有跟这个课程名称相同的都删除了。在找到这个 bug 之后，我们不应该马上修复。而是先增加下面这样的测试用例：

```
public class TestCourseCatalog extends TestCase {
   CourseCatalog cat;
   ...
   public void testRemoveKeepOthersWithSameTitle() {
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "Java prgoramming", ...);
       cat.add(course1);
       cat.add(course2);
       cat.remove(course1);
       assertEquals(cat.findCourseWithId(course2.getId()), course2);
   }
}
```

当然，现在这个测试用例肯定失败。它可以当做一个 bug 报告，告诉我们代码里面有一个 bug。原来的 bug 报告（比如是一份文档）可以放一边去了。我们每次运行起所有的测试用例，就能知道哪里有 bug 了，绝对不会遗漏哪个 bug。

单元测试的另一个重要作用就是，如果以后在 remove 方法里面又出现了同样的 bug，这个单元测试可以马上告诉我们，出错了。

怎么测试是否有抛异常

我们可以调用 CourseCatalog 里面的 remove 方法删除一个 Course 对象。如果在 CourseCatalog 里面没有这个课程的话，remove 要做什么？它什么也不做，就抛出一个错误码，或者抛出一个异常。现在，我们假设它抛出一个 CourseNotFoundException：

```
class CourseNotFoundException extends Exception {
}
class CourseCatalog {
   void remove(Course course) throws CourseNotFoundException {
       ...
   }
```

```
}
```

相应的，我们要测试它有没有真的抛出异常。因此，我们可以在 TestCourseCatalog 增加这个一个测试用例：

```
public class TestCourseCatalog extends TestCase {
   CourseCatalog cat;
   ...
   public void testRemove() {
       ...
   }
   public void testRemoveNotFound() {
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "OO design", ...);
       cat.add(course1);
       cat.remove(course2); //怎么测试它有没有抛出 CourseNotFoundException?
   }
}
```

问题是，怎么测试它是不是真的抛出一个异常？我们可以这样写：

```
public class TestCourseCatalog extends TestCase {
   ...
   public void testRemoveNotFound() {
       Course course1 = new Course("c001", "Java prgoramming", ...);
       Course course2 = new Course("c002", "OO design", ...);
       cat.add(course1);
       try {
           cat.remove(course2);
           assertTrue(false);
       } catch (CourseNotFoundException e) {
       }
   }
}
```

如果没有抛出异常，assertTrue(false) 就会被执行，它会让这个测试过不了。如果 remove 确实抛出了 CourseNotFoundException，就会跳过这一句进入 catch 语句，这个测试就算通过了。如果代码抛出了其他的异常，那这个异常就不会被 catch 住，直接给了 JUnit。JUnit 就会认为这个测试失败。

不过，除了这样写以外，JUnit 还给我们提供了这样的方法：

```
public void testRemoveNotFound() {
   ...
```

```
        try {
            cat.remove(course2);
            fail(); //跟 assertTrue(false)一样。
        } catch (CourseNotFoundException e) {
        }
    }
```

引述

http://www.junit.org.

http://c2.com/cgi/wiki?UnitTest.

章节练习

问题

1. 测试 CourseCatalog 里的 findCoursesWithTitle 方法。
2. 如果我们调用 CourseCatalog 的 add 方法来增加一个 Course 对象，但这个 Course 的 ID 跟 CourseCatalog 里的已有 Course 的 ID 冲突了，add 方法就会抛出 CourseDuplicateException。测试这个行为。
3. 给 CourseCatalog 增加一个 clear 方法。它会删除 CourseCatalog 里的所有 Course 对象。测试这个方法。

Hints

1. 没有提示。
2. 没有提示。

3. 为了测试这个 clear 方法，你可能需要在 CourseCatalog 里面增加一个新方法。

解决方法示例

1. 测试 CourseCatalog 里的 findCoursesWithTitle 方法。

```
public class TestCourseCatalog extends TestCase {
        ...
    public void testFindCoursesWithTitle() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        Course course3 = new Course("c003", "Java Programming", ...);
        cat.add(course1);
```

```
            cat.add(course2);
            cat.add(course3);
            Course courses[] = cat.findCoursesWithTitle(course1.getTitle());
            assertEquals(courses.length, 2);
            assertEquals(courses[0], course1);
            assertEquals(courses[1], course3);
        }
    }
```

2. 如果我们调用 CourseCatalog 的 add 方法增加一个 Course 对象，但这个 Course 的 ID 跟 CourseCatalog 里的已有 Course 的 ID 冲突了，add 方法就会抛出 CourseDuplicateException。测试这个行为。

```
public class TestCourseCatalog extends TestCase {
        ...
    public void testAddDup() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        try {
            cat.add(course2);
            fail();
        } catch (CourseDuplicateException e) {
        }
    }
}
```

3. 给 CourseCatalog 增加一个 clear 方法。它会删除 CourseCatalog 里的所有 Course 对象。测试这个方法。

```
public class TestCourseCatalog extends TestCase {
    public void testClear() {
        Course course1 = new Course("c001", "Java prgoramming", ...);
        Course course2 = new Course("c002", "OO design", ...);
        cat.add(course1);
        cat.add(course2);
        cat.clear();
        assertEquals(cat.countCourses(), 0);
    }
}
class CourseCatalog {
    ...
    int countCourses() {
        ...
    }
}
```

第 13 章　测试驱动编程

实现一个跟学生选修课程的用户例事

假设我们现在在做一个学生管理系统。目前我们做到了一个跟学生选修课程有关的用户例事。学生可以选修一门课程，并交上这门课程需要的学费。这次选修的信息要记录下来（比如，学生 ID，课程编码，日期，操作人等等）。交费的细节也要记录下来（数目，现金还是信用卡等等）。这个学生跟这门课程都必须已经在这系统注册过的。这门课程的位置没满的话，学生才可以选修。每门课程都有个位置总数。如果有人预订了这门课程，系统会为这个预订保留 24 个小时，不过用户也可以更改这个预订。一门课程可以由多个“模块”组成。比如，“Java 开发”这门课程可以由“Java 语言”和“JDBC”模块组成。每个模块其实也相当于一门课程。不同的课程可能会共用同样的模块。有些模块是不能单独选修的，它只能做为一些课程的子模块。如果学生想选修这个模块，他就必须选修相应的整个父课程。每个模块也有个位置总数，只有这门课程的所有模块的位置都没满的情况下，学生才可以选修这门课程。

假设现在系统里面有这些类：

```
class Student {
   ...
}

class Course {
   ...
   Course[] getModules() { ... }
}

class Reservation {       //预订
   Date reserveDate;
   int daysReserved;
   ...
}

class StudentSet {
   ...
}

class CourseSet {
   ...
}
```

```
class ReservationSet {
   Reservation[] getReservationsFor(String courseCode) { ... }
   ...
}
```

我们考虑到，为了实现这个这个用户例事，我们应该创建一个 Enrollment 类和 EnrollmentSet 类。每次选修要被增加到数据库里面时，必须先检查一下：

```
class Enrollment {
   ...
}
class EnrollmentSet {
   void add(Enrollment enrollment) {
       assertValid(enrollment);
       addToDB(enrollment);
   }
   void assertValid(Enrollment enrollment) {
       ...
   }
   void addToDB(Enrollment enrollment) {
       ...
   }
}
```

现在，我们继续完善这些类：

```
class Enrollment {
   String studentId;
   String courseCode;
   Date enrolDate;
   Payment payment;
}

class Payment {
   int amount;
   Payment(int amount) {
       ...
   }
}

class CashPayment extends Payment {
   CashPayment(int amount) {
       ...
```

```
    }
}

class CreditCardPayment extends Payment {
    String cardNo;
    String nameOnCard;
    Date expiryDate;
    CreditCardPayment(String cardNo,String nameOnCard, Date expiryDate,int amount) {
        ...
    }
}

class EnrollmentSet {
    StudentSet studentSet;
    CourseSet courseSet;
    ReservationSet reservationSet;
    EnrollmentSet(StudentSet studentSet, CourseSet courseSet, ReservationSet reservationSet) {
        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    void add(Enrollment enrollment) {
        assertValid(enrollment);
        addToDB(enrollment);
    }
    void assertValid(Enrollment enrollment) {
        assertStudentExists(enrollment.getStudentId());
        assertCourseExists(enrollment.getCourseCode());
        assertPaymentCorrect(enrollment);
        assertHasFreeSeat(enrollment);
        assertCanEnrolInIsolation(enrollment);
    }
}
```

还没结束。我们还要写 assertStudentExists，assertCourseExists，assertPaymentCorrect，assertHasFreeSeat，assertCanEnrollInIsolation 和 addToDB 这些方法。让我们一个一个实现这些方法。先实现 assertStudentExists：

```
class EnrollmentSet {
    ...
    void assertStudentExists(String studentId) {
        studentSet.assertStudentExists(studentId);
    }
```

```
}
```

这个很简单，我们先假设 StudentSet 类里面已经有 assertStudentExists 这样一个方法。所以这个方法搞定了。类似的 assertCourseExists：

```
class EnrollmentSet {
    ...
    void assertCourseExists(String courseCode) {
        courseSet.assertCourseExists(courseCode);
    }
}
```

接着，实现 assertPaymentCorrect：

```
class EnrollmentSet {
    ...
    void assertPaymentCorrect(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (enrollment.getPayment().getAmount()!=course.getFee()) {
            throw new IncorrectPaymentException();
        }
    }
}
```

这个也不难。
下面实现 assertHasFreeSeat：

```
class EnrollmentSet {
    ...
    void assertHasFreeSeat(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (course.getNoSeats() <= getNoEnrollments(course)+getNoActiveReservations(course)) {
            throw new CourseFullException();
        }
    }
}
```

这个就有点难了：这个方法要求我们还要写两个方法：getNoEnrollments 和 getNoActiveReservations。所以我们需要一个一个写：

```
class EnrollmentSet {
    Connection conn;
    ...
```

```
    EnrollmentSet(
        Connection conn,
        StudentSet studentSet,
        CourseSet courseSet,
        ReservationSet reservationSet) {
        this.conn = conn;
        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    int getNoEnrollments(Course course) {
        PreparedStatement st =
            conn.prepareStatement("select count(*) from enrollments where courseCode=?");
        try {
            st.setString(1, course.getCode());
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        } finally {
            st.close();
        }
    }
}
```

然后：

```
class EnrollmentSet {
    ...
    int getNoActiveReservations(Course course) {
        return reservationSet.getNoActiveReservations(course.getCode());
    }
}
```

假设 ReservationSet 类里面没有 getNoActiveReservations 方法。我们需要增加它：

```
class ReservationSet {
    Reservation[] getReservationsFor(String courseCode) { ... }
    int getNoActiveReservations(String courseCode) {
        int activeCount = 0;
        Reservation reservations[] = getReservationsFor(courseCode);
        for (int i = 0; i < reservations.length; i++) {
            if (reservations[i].isActive()) {
                activeCount++;
```

```
            }
        }
        return activeCount;
    }
}
```

我们又要在 Reservation 增加 isActive 方法：

```
class Reservation {
    Date reserveDate;
    int daysReserved;
    boolean isActive() {
        Date today = new Date();
        return getLastReservedDate().before(today);
    }
    Date getLastReservedDate() {
        GregorianCalendar lastReservedDate = new GregorianCalendar();
        lastReservedDate.setTime(reserveDate);
        lastReservedDate.add(GregorianCalendar.DATE, daysReserved);
        return lastReservedDate.getTime();
    }
}
```

接着，实现 assertCanEnrolInIsolation：

```
class EnrollmentSet {
    ...
    void assertCanEnrolInIsolation(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (course.isModule() && !course.canEnrolInIsolation()) {
            throw new CannotEnrolInIsolation();
        }
    }
}
```

我们假设 Course 类里面已经有 isModule 和 canEnrolInIsolation 方法了：

```
class Course {
    boolean isModule() { ... }
    boolean canEnrolInIsolation() { ... }
}
```

最后，实现 addToDB 方法。我们假设 enrollments 这个表是这样的：

```
CREATE TABLE enrollments (
   courseCode VARCHAR(50),
   studentId VARCHAR(50),
   enrolDate DATE,
   amount INT,
   paymentType VARCHAR(20),
   cardNo VARCHAR(20),
   expiryDate DATE,
   nameOnCard VARCHAR(50)
);
```

addToDB 方法：

```
class EnrollmentSet {
   private void addToDB(Enrollment enrollment) {
       PreparedStatement st =
           conn.prepareStatement(
              "insert into enrollments values (?,?,?,?,?,?,?,?)");
       try {
           st.setString(1, enrollment.getCourseCode());
           st.setString(2, enrollment.getStudentId());
           st.setDate(3, new Date(enrollment.getEnrolDate().getTime()));
           Payment payment = enrollment.getPayment();
           st.setInt(4, payment.getAmount());
           if (payment instanceof CashPayment) {
              st.setString(5, "Cash");
              st.setNull(6, Types.VARCHAR);
              st.setNull(7, Types.DATE);
              st.setNull(8, Types.VARCHAR);
           } else if (payment instanceof CreditCardPayment) {
              CreditCardPayment creditCardPayment =
                  (CreditCardPayment) payment;
              st.setString(5, "CreditCard");
              st.setString(6, creditCardPayment.getCardNo());
              st.setDate(
                  7,
                  new Date(creditCardPayment.getExpiryDate().getTime()));
              st.setString(8, creditCardPayment.getNameOnCard());
           }
           st.executeUpdate();
       } finally {
           st.close();
```

```
        }
    }
}
```

现在我们写完所有代码了。我们一口气写了 150 行的代码，而且在这过程当中，我们没运行过一次我们写的代码。此时，我们脑子里面最大的问题是：我们的代码是不是正确的？事实上，当我们在写这些代码的时候，这个问题就一直困扰着我们。代码写得越复杂（比如 addToDB, getNoEnrollments, isActive），我们就越担心。代码写得越多，我们越担心。比如，当我们发现我们为了实现一个方法时，还要写两个其他的方法时，我们已经开始担心了。只有写过的代码都运行测试一下，我们的担心才会缓解。怎么测试？

怎么测试刚写的代码

如果我们写了些有 GUI 有关的代码，我们一般是倾向于这样测试的：直接运行系统，手动输入一些数据，然后到数据库看看是不是按照我们预期的运行了。不过，这样可不好，因为这种测试不是自动化的，不能经常运行。再比如，有人修改了这些代码，出现了个 bug，如果他没有运行整个系统，或者只是很简单的测试了一下，他可能发现不了这个错误。因此，一个好的方式就是有一些自动单元测试。

那怎么对 EnrollmentSet 的 add 方法进行单元测试？我们可能先增加一个有效的选修：

```
class EnrollmentSetTest extends TestCase {
    testAddValidEnrollment() {
        Connection conn = ...;
        try {
            EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
            enrollmentSet.deleteAll();
            Enrollment enrollment =
                new Enrollment(
                    "s001",
                    "c001",
                    new Date(),
                    new CashPayment(100));
            enrollmentSet.add(enrollment);
            assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
        } finally {
            conn.close();
        }
    }
}
```

这个测试还没完成。还缺少许多东西：数据库连接没有创建，我们要将“s001”这个学生增加到数据库里面，我们还要将“c001” 这个课程加到数据库中。所以，先开一个数据库连接吧。假设我们使用的是 postgreSQL 数

据库，里面有个“testdb”数据库：

```
class EnrollmentSetTest extends TestCase {
   testAddValidEnrollment() {
      Class.forName("org.postgresql.Driver");
      Connection conn =
         DriverManager.getConnection(
            "jdbc:postgresql://localhost/testdb",
            "username",
            "password");
      try {
         EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
         enrollmentSet.deleteAll();
         Enrollment enrollment =
            new Enrollment(
               "s001",
               "c001",
               new Date(),
               new CashPayment(100));
         enrollmentSet.add(enrollment);
         assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
      } finally {
         conn.close();
      }
   }
}
```

接着，将学生“s001”增加到数据库里。为此，我们要创建一个 Student 对象。这可不是一件容易的事，因为 Student 的构造函数需要一堆参数：

```
class Student {
   String studentId;
   String idCardNo;
   String name;
   boolean isMale;
   Date dateOfBirth;
   ContactInfo address;
   Employment employment;

   Student(
      String studentId,
      String idCardNo,
      String name,
```

```
        boolean isMale,
        Date dateOfBirth,
        ContactInfo address,
        Employment employment) {
        ...
    }
}

class ContactInfo {
    String email;
    String telNo;
    String faxNo;
    String postalAddress;
    public ContactInfo(String email, String telNo, String faxNo, String postalAddress) {
        ...
    }
}

public class Employment {
    String companyName;
    String jobTitle;
    int yearsOfService;
    ContactInfo contactInfo;
    public Employment(
        String companyName,
        String jobTitle,
        int yearsOfService,
        ContactInfo contactInfo) {
        ...
    }
}
```

因此，为了创建一个 student 对象，代码会变成：

```
class EnrollmentSetTest extends TestCase {
    testAddValidEnrollment() {
        Class.forName("org.postgresql.Driver");
        Connection conn =
            DriverManager.getConnection(
                "jdbc:postgresql://localhost/testdb",
                "username",
                "password");
        try {
```

```
            StudentSet studentSet = new StudentSet(conn);
            studentSet.deleteAll();
            Student student =
                new Student(
                    "s001",
                    "9/29741/8",
                    "Paul Chan",
                    true,
                    new Date(),
                    new ContactInfo(
                        "paul@yahoo.com",
                        "123456",
                        "123457",
                        "postal address"),
                    new Employment(
                        "ABC Ltd.",
                        "Manager",
                        2,
                        new ContactInfo(
                            "info@abc.com",
                            "111000",
                            "111001",
                            "ABC postal address")));
            studentSet.add(student);
            EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
            enrollmentSet.deleteAll();
            Enrollment enrollment =
                new Enrollment(
                    "s001",
                    "c001",
                    new Date(),
                    new CashPayment(100));
            enrollmentSet.add(enrollment);
            assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
        } finally {
            conn.close();
        }
    }
}
```

这真是糟透了。我们只想要这个学生的 id 和实例，根本不需要他的姓名，身份证号，生日，联系方式，职业信息这些东西。可以将那些信息设为 null 吗？

```
Student student = new Student("s001", null, null,true, null, null, null);
studentSet.add(student);
```

不行。因为将学生保存到数据库中时，要执行许多的检查，联系方式也会被存到数据库中。如果你将引用设为 null，程序就会出错。所以我们只能乖乖输入所有的数据。

不过，恐怖的事情还会接踵而来。测试还没完成思绪。我们还要将课程“c001”加到数据库中。我们还会遇到一个类似的难点：我们明明只需要课程编码，费用和位置数这些信息，我们不需要课程名称，课程简介，课程说明，时间表等一堆其他的信息。

受不了了。这样下去，为 5 行的代码要写上百行的测试代码。要想办法解决这种问题！

测试先行的解决方法

让我们先抛开所有的代码，以另一种大不同的方式来实现这个用户例事。首先，我们先写单元测试（是的，我们不写其他代码，就先写单元测试!)。测试很简单：我们需要一个 Enrollment 和一个 EnrollmentSet 对象，然后将那个 enrollment 加到 enrollmentSet 中：

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.add(enrollment);
   }
}
```

注意到，此刻我们并没有 Enrollment 类，也没有 EnrollmentSet 类。此外，Enrollment 的构造函数只有两个参数：学生 ID 和课程编码。为什么要这些参数？很简单，因为我现在只想提供这两个参数，我不想要其他参数。类似的，我现在提供不了任何参数给 EnrollmentSet 的构造函数，所以它没有参数。

不过怎么测试 enrollment 有没有真的加到数据库中？我们要去观察整个数据库吗：

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.add(enrollment);
       Class.forName("org.postgresql.Driver");
       Connection conn =
           DriverManager.getConnection(
```

```
                "jdbc:postgresql://localhost/testdb",
                "username",
                "password");
            ...
        }
    }
```

这里面还是太多代码了，而且检查数据库很麻烦。我们要想一些别的办法，不要去访问数据库。比如，EnrollmentSet 只需将 Enrollment 对象的信息写到某个地方。这地方不一定就是个数据库。就叫 EnrollmentStorage 吧。现在我们在测试里面创建一个 EnrollmentStorage，让 EnrollmentSet 使用它，最后看一下它有没有将 Enrollment 增加到 EnrollmentStorage 中：

```
interface EnrollmentStorage {
   void add(Enrollment enrollment);
}

class EnrollmentStorageForTest implements EnrollmentStorage {
   Enrollment enrollmentStored;
   void add(Enrollment enrollment) {
       enrollmentStored = enrollment;
   }
}

class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       EnrollmentStorageForTest storage = new EnrollmentStorageForTest();
       enrollmentSet.setStorage(storage);
       enrollmentSet.add(enrollment);
       assertTrue(storage.enrollmentStored==enrollment);
   }
}
```

当然，此刻单元测试根本编译不过。我们现在就要创建 Enrollment 类和 EnrollmentSet 类。不过，我们只写这个单元测试需要的代码，这样才能尽量快的将单元测试跑起来：

```
class Enrollment {
   Enrollment(String studentId, String courseCode) {
   }
}

class EnrollmentSet {
```

```
    void setStorage(EnrollmentStorage storage) {
    }
    void add(Enrollment enrollment) {
    }
}
```

是的，这些方法都是空的！如我们所说的，我们希望尽快将单元测试跑起来。现在，运行单元测试吧。它失败了。为什么明知它会失败还要运行它呢？这个待会儿再解释。现在，先简单的记住一件事，在写实现代码前，运行了一个测试，然后看着它通不过。现在，我们继续写代码，以让这个测试能过：

```
class EnrollmentSet {
    EnrollmentStorage storage;

    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
}
```

只有 3 行的代码。1 分钟内就能搞定了，现在我们再运行这个测试。它通过了。一个进步啊。接站，我们要写另外一个单元测试了。比如，测试它会不会检查这个学生是否注册过的。不过，在此之前，我们可以简化一下之前的测试代码，将 EnrollmentStorageForTest 变成匿名类：

```
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        final Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
                assertTrue(enrollmentToStore == enrollment);
            }
        });
        enrollmentSet.add(enrollment);
    }
}
```

因为我们又改了代码，再运行一下单元测试，看看能不能通过。通过了。所以，我们安全上垒，继续做下面的事吧。

测试->检查学生是否注册的代码

现在，写一个测试，测试看系统会不会真的检查学生是否注册。这个测试用例是一个已注册过的学生的一次选修。不过怎么告诉 EnrollmentSet 这个学生已经注册了？因为我们知道，将这个学生增加到 StudentSet 是极大的工作量。就像将数据库换成 EnrollmentStorage 一样，我们可以用 StudentRegistryChecker 代替 StudentSet：

```
interface StudentRegistryChecker {
   boolean isRegistered(String studentId);
}

class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
       ...
   }
   void testStudentRegistered() {
       final Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
           boolean isRegistered(String studentId) {
              return studentId.equals("s001");
           }
       });
       enrollmentSet.setStorage(new EnrollmentStorage() {
           void add(Enrollment enrollmentToStore) {
              assertTrue(enrollmentToStore == enrollment);
           }
       });
       enrollmentSet.add(enrollment);
   }
}
```

不过，这个跟 testAddEnrollment 很相似。不同的是，后者是测试 enrollment 有没有增加到数据存储里面，而前者又多做了一点点事而已。不过对同一样行为测试两次一点意义都没有。testStudentRegistered 应该只是测试增加 enrollment 之前的检查注册的行为。所以 EnrollmentSet 应该有一个 assertValid 的方法，这样我们就可以调用这个方法，而不是去调用 add 方法了：

```
class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
       ...
   }
   void testStudentRegistered() {
```

```
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return studentId.equals("s001");
            }
        });
        enrollmentSet.assertValid(enrollment);
    }
}
```

现在测试清楚多了。当然，我们还要测试它在将 enrollment 加到数据存储之前，是不是真的调用了 assertValid 方法。不过此时，先把它放在 TODO 列表里面就行了：

//TODO　测试 add 方法有没有调用 assertValid.

我们先让 testStudentRegistered 过了再说。所以，先写相应的方法，让方法编译得过：

```
class EnrollmentSet {
    EnrollmentStorage storage;
    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
    }
    void assertValid(Enrollment enrollment) {
    }
}
```

现在，运行两个测试。喔！竟然两个测试都过了！因为此时它认为的有的 enrollment 都是有效的。所以我们提供的真正有效的 enrollment 自然不费力就通过检查了。当然，我们明白 assertValid 方法里面应该有一些代码，比如通过 StudentRegistryChecker 去遍历所有的学生。不过，在此之前，先让这个测试过不了再说。所以，我们试着检查一个没有注册过的学生。我们期望它可以抛出一个 StudentNotFoundException 异常：

```
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        ...
    }
    void testStudentRegistered() {
        ...
```

```
    }
    void testStudentUnregistered() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return false;
            }
        });
        try {
            enrollmentSet.assertValid(enrollment);
            fail();
        } catch (StudentNotFoundException e) {
        }
    }
}
```

它编译不了，因为 StudentNotFoundException 还没定义。创建一个空类让测试编译得过：

```
class StudentNotFoundException extends RuntimeException {
}
```

现在运行所有测试。如果期望中的，它失败了。我们现在写代码让它通过：

```
class EnrollmentSet {
    EnrollmentStorage storage;
    StudentRegistryChecker studentRegistryChecker;
    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
        this.studentRegistryChecker = registryChecker;
    }
    void assertValid(Enrollment enrollment) {
        if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
            throw new StudentNotFoundException();
        }
    }
}
```

不过，我们需要 Enrollment 里面有个 getStudentId 方法。通过我们会为这个方法，写一个测试，然后再让它通过。不过这个方法太简单了，我们相信我们不需要测试这个方法。所以我们继续写了下面的方法：

```
class Enrollment {
   String studentId;
   Enrollment(String studentId, String courseCode) {
       this.studentId = studentId;
   }
   String getStudentId() {
       return studentId;
   }
}
```

注意到，这是我们第一次使用那个被传递给 Enrollment 的构造函数的学生 ID。之前我们都不需要它。

现在运行所有测试。令人惊讶的是，testStudentRegistered 和 testStudentUnregistered 都失败了！bug 很有可能是在我们刚增加的 10 行代码里面。最有可能的地方就是 assertValid 方法：

```
void assertValid(Enrollment enrollment) {
   if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
       throw new StudentNotFoundException();
   }
}
```

很明显，如果这个学生已经注册过，就会抛出一个异常。这很显不对。所以，我们应该这样修复：

```
void assertValid(Enrollment enrollment) {
   if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
       throw new StudentNotFoundException();
   }
}
```

现在运行的有测试。都通过了。从这个事件我们可以看到“测一点，写一点”的一个好处：如果我们弄出了一个 bug，它会马上被发现，然后很容易定位到出错源。比较一下，我们原来写了 150 行没有测试的代码的方式。

测试->检查课程是否还有位置的代码

现在，我们再挑另一件事情来测试。比如，我们可以测试一下，系统会不会真的检查课程是否已经注册。不过这件事应该跟检查学生的注册一样，我们先跳过。我们挑课程空位的检查来测。不过课程空位的检查还包括两个部分，一个是检查已被占用的位置，还有一个是被预订的位置。我们先忽略后者，将它加到 TODO 里面，现在我们就有两个 TODO 了：

```
//TODO    测试 add 方法有没有调用 assertValid.
//TODO    测试系统有没有考虑到预订的情况
```

现在，我们来测试看，系统有没有考虑课程的位置被选修的学生占用的情况。这个测试里面，我们假设课程已经有两个学生选择了，而且这个课程的位置总数就是 2 个。然后我们尝试再增加一个选修，看看它会不会抛出 ClassFullException 这样的异常：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       //怎么让 enrollmentSet 认为已经有两个学生选修了 c001?
       try {
           enrollmentSet.assertValid(enrollment);
           fail();
       } catch (ClassFullException e) {
       }
   }
}
```

问题是：怎么让 enrollmentSet 认为已经有两个学生选修了 c001？我们可以增加两个 enrollment，不过这样太麻烦了，因为我们还要创建两个 enrollment 对象，保证它们是有效的，还要通过读取数据存储来确保这两个 enrollment 对象确实有被增加进去。工作量太大了。所以，我们用一个 EnrollmentCounter 代替：

```
interface EnrollmentCounter {
   int getNoEnrollments(String courseCode);
}

class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
           int getNoEnrollments(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       try {
           enrollmentSet.assertValid(enrollment);
           fail();
       } catch (ClassFullException e) {
       }
   }
```

```
}
```

但是我们还要让 enrollmentSet 认为课程 c001 的位置总数是 2。为此，我们让它使用一个 CourseLookup 类去根据这个课程编码取得相应的 Course 对象，然后从 Course 对象里面取得位置总数。不过创建一个 Course 对象还是太费劲了。一个更简单的方法就是让它使用一个 ClassSizeGetter：

```
public interface ClassSizeGetter {
   public int getClassSize(String courseCode);
}

class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsTaken() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
           int getClassSize(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
           int getNoEnrollments(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       try {
           enrollmentSet.assertValid(enrollment);
           fail();
       } catch (ClassFullException e) {
       }
   }
}
```

测试还编译不过。先创建 ClassFullException 类、setClassSizeGetter 和 setEnrollmentCounter 方法：

```
class ClassFullException extends RuntimeException {
}
class EnrollmentSet {
   EnrollmentStorage storage;
   StudentRegistryChecker studentRegistryChecker;

   void assertValid(Enrollment enrollment) {
       ...
```

```
    }
    void setClassSizeGetter(ClassSizeGetter sizeGetter) {
    }
    void setEnrollmentCounter(EnrollmentCounter counter) {
    }
}
```

现在运行所有测试。失败了。不过现在出错情况很特殊：它抛出一个 NullPointerException。通过调查，我们发现是因为 EnrollmentSet 里面的 studentRegistryChecker 是个 null。对，它是用来检查学生 s001 是否已经注册的类。不过我们并没有创建一个 StudentRegistryChecker 实例。为此，我们将将之前的代码拷出来，用来创建一个 StudentRegistryChecker 实例：

```
void testAllSeatsTaken() {
    Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
        boolean isRegistered(String studentId) {
            return false;
        }
    });
    enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
        int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
        }
    });
    enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
        int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
        }
    });
    try {
        enrollmentSet.assertValid(enrollment);
        fail();
    } catch (ClassFullException e) {
    }
}
```

但是这里面还是太多工作量了。事实上，我们又重新测试了系统检查学生注册的功能。之前说过，对同样的东西没必要多次测试。这个是建议 EnrollmentSet 类里面应该有一个 assertHasSeat 方法。这样我们就可以单独测试这个方法了。

当然，在此之前，我们要先测试 EnrollmentSet 在增加 enrollment 的时候，有没有先调用 assertHasSeat 的方法。相应地，assertValid 方法要改名成 assertStudentRegistered（顺便把 TODO 改一下）：

```
//TODO  测试add方法有没有调用assertStudentRegistered和assertHasSeat。
//TODO  测试系统有没有考虑到预订的情况

class EnrollmentSet {
   ...
   void assertStudentRegistered(Enrollment enrollment) {
      ...
   }
   void assertHasSeat(Enrollment enrollment) {
   }
}

class EnrollmentSetTest extends TestCase {
   void testAddEnrollment() {
      ...
   }
   void testStudentRegistered() {
      ...
      enrollmentSet.assertStudentRegistered(enrollment);
      ...
   }
   void testStudentUnregistered() {
      ...
      enrollmentSet.assertStudentRegistered(enrollment);
      ...
   }
   void testAllSeatsTaken() {
      Enrollment enrollment = new Enrollment("s001", "c001");
      EnrollmentSet enrollmentSet = new EnrollmentSet();
      enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
          int getClassSize(String courseCode) {
             return courseCode.equals("c001") ? 2 : 0;
          }
      });
      enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
          int getNoEnrollments(String courseCode) {
             return courseCode.equals("c001") ? 2 : 0;
          }
      });
      try {
          enrollmentSet.assertHasSeat(enrollment);
          fail();
      } catch (ClassFullException e) {
```

```
            }
        }
    }
```

现在再运行所有的测试。当前在做的这个测试用例失败了。这是预料中的。现在我们让这个测试通过：

```
class EnrollmentSet {
    ...
    void assertStudentRegistered(Enrollment enrollment) {
        ...
    }
    void assertHasSeat(Enrollment enrollment) {
        String courseCode = enrollment.getCourseCode();
        if (enrollmentCounter.getNoEnrollments(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
}
```

这里面用到了 Enrollment 类里面的 getCourseCode 方法。它是一个很简单的方法，我们不测，直接写出来就行了：

```
class Enrollment {
    String studentId;
    String courseCode;
    Enrollment(String studentId, String courseCode) {
        this.studentId = studentId;
        this.courseCode = courseCode;
    }
    String getStudentId() { ... }
    String getCourseCode() {
        return courseCode;
    }
}
```

注意，这是我们第一次使用传递到 Enrollment 的构造函数里的课程编码。

现在，运行所有的测试。他们都通过了。为了保证代码确实正确，我们应该测试课程有空位的情况。我们只要简单的 Copy/Paste 加修改一下 testAllSeatsTaken 的代码就行了。将已选修该课程的人数从 2 变成 1，希望期望它不会抛异常：

```
public void testHasAFreeSeat() {
```

```
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 1 : 0;
            }
        });
        enrollmentSet.assertHasSeat(enrollment);
    }
```

现在运行所有测试。都通过了。

系统有没有考虑到父课程的选修情况

不过我们还没有完全通过测试。我们还有一个有趣的情况没考虑：如果该课程同时还是别的课程子模块，当我们要检查它还有没有空位时，我们还要检查它的父课程的情况。比如，假设 c002 是 c001 的子模块，它的位置总数是 5。c002 的已选人数是 2，c001 的已选人数是 3，那么 c001 就没有空位了。我们是通过 EnrollmentCounter 的 getNoEnrollments 去取得某个课程的已选人数，那现在，如果让它去取 c001 的已选人数，它是返回 2，还是返回 5 呢？根据我们的思想，我们希望它返回的应该是 5。为了让代码更精确，我们将 getNoEnrollments 改名为 getNoSeatsTaken（选择这门课程的人数，不等于这门课程被占用的位置的情况）：

```
    interface EnrollmentCounter {
        int getNoSeatsTaken(String courseCode);
    }
```

为了安全起见，再运行一下测试用例，它们应该全部通过。（至于 getNoSeatsTaken 这个方法的测试，不是我们现在要考虑的。）

测试系统有没有考虑预订的情况

现在，从我们的 TODO 列表里面挑一个出来做吧：

//TODO　　测试 add 方法有没有调用 assertStudentRegistered 和 assertHasSeat。

//TODO　测试系统有没有考虑到预订的情况

我们先做第 2 个 TODO。我们要创建一个 ReservationCounter 类，它根据课程编码取得它的预订人数（当然，如果它是一个子模块的话，也要包括它父课程的预订人数）。我们要创建一个上下文环境是这个课程有一些人选修了，剩下的位置也全被预订了。这个是用来测试 assertHasSeat 有没有同时考虑已选人数跟预订的情况。它跟 testAllSeatsTaken 很像：

```
public interface ReservationCounter {
   public int getNoSeatsReserved(String courseCode);
}

class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsReservedOrTaken() {
       Enrollment enrollment = new Enrollment("s001", "c001");
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
           int getClassSize(String courseCode) {
              return courseCode.equals("c001") ? 3 : 0;
           }
       });
       enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
           int getNoSeatsTaken(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       enrollmentSet.setReservationCounter(new ReservationCounter() {
           int getNoSeatsReserved(String courseCode) {
              return courseCode.equals("c001") ? 1 : 0;
           }
       });
       try {
           enrollmentSet.assertHasSeat(enrollment);
           fail();
       } catch (ClassFullException e) {
       }
   }
}
```

创建一个空的 setReservationCounter 方法，让测试代码编译通过：

```
class EnrollmentSet {
   ...
   void setReservationCounter(ReservationCounter counter) {
```

```
    }
}
```

运行这些测试用例，当前在做的这个用例失败了。预料之中。让用例通过：

```
class EnrollmentSet {
    ...
    ReservationCounter reservationCounter;
    ...
    public void assertHasSeat(Enrollment enrollment) {
        String courseCode = enrollment.getCourseCode();
        if (enrollmentCounter.getNoSeatsTaken(courseCode)
            + reservationCounter.getNoSeatsReserved(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
    void setReservationCounter(ReservationCounter counter) {
        this.reservationCounter = counter;
    }
}
```

现在运行所有用例。当前的这个测试用例通过了。不过之前的那两个测试用例却失败了：testAllSeatsTaken 和 testHasAFreeSeat。他们都抛出了 NullPointerException。这是因为这些测试都没有创建一个 ReservationCounter 实例让 EnrollmentSet 使用。一个快速的方法就是：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
```

```
                return 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
    public void testHasAFreeSeat() {
        ...
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
        ...
    }
}
```

现在运行所有测试，它们都通过了。

测试 add 方法

接着我们挑 TODO 里面的最后一条：

//TODO　测试 add 方法有没有调用 assertStudentRegistered 和 assertHasSeat。

我们可以这样做：

```
class EnrollmentSetTest extends TestCase {
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
            void assertHasSeat(Enrollment enrollment) {
```

```
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment("s001", "c001");
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

这个测试要求 add 方法先调用 assertStudentRegistered，再调用 assertHasSeat，然后最后调用将 enrollment 增加到 storage 里面。

现在运行所有测试，当前的测试用例会失败。因为此时 add 方法都没有先验证有效：

```
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        storage.add(enrollment);
    }
    ...
}
```

现在当前的测试通过了，不过第一个用例又抛出 NullPointerException：

```
void testAddEnrollment() {
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.add(enrollment);
}
```

因为现在 add 方法要验证环境，但它要用到的对象都没有实例化（比如 StudentRegistryChecker，ClassSizeGetter 等等)。我们可以建立所有的实例，不过那会是蛮大的工作量。跟之前一样，我们将要测的代码

分离到一个新的方法里面，这样我们的测试代码就不会重复测试到一些东西了。我们将新的方法叫做 addToStorage：

```
void testAddEnrollment() {
   final Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStorage(new EnrollmentStorage() {
       void add(Enrollment enrollmentToStore) {
           assertTrue(enrollmentToStore == enrollment);
       }
   });
   enrollmentSet.addToStorage(enrollment);
}
```

建一个新的空方法 addToStorage：

```
class EnrollmentSet {
   ...
   void addToStorage(Enrollment enrollment) {
   }
}
```

因为它现在是在测试 addToStorage 而不再是 add 方法，所以我们将 testAddEnrollment 改名为 testAddToStorage：

```
void testAddToStorage() {
   final Enrollment enrollment = new Enrollment("s001", "c001");
   EnrollmentSet enrollmentSet = new EnrollmentSet();
   enrollmentSet.setStorage(new EnrollmentStorage() {
       void add(Enrollment enrollmentToStore) {
           assertTrue(enrollmentToStore == enrollment);
       }
   });
   enrollmentSet.addToStorage(enrollment);
}
```

运行这个测试，我们期望它是失败的。不过它竟然没有失败！因为此时 addToStorage 并没有调用 EnrollmentStorage 里面的 add 方法，所以我们的验证代码没有被执行到。说明我们的这个测试实际上是没什么用的。这就是为什么我们要在实现代码之前，先保证我们的测试通不过。如果我们的测试通过了，没有像预期中的失败了，说明我们的测试有问题。现在回到手中的例子。怎么修正这个测试？我们可以再用一个调用记录：

```
void testAddToStorage() {
   final StringBuffer callLog = new StringBuffer();
```

```
        final Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
                callLog.append("x");
                assertTrue(enrollmentToStore == enrollment);
            }
        });
        enrollmentSet.addToStorage(enrollment);
        assertEquals(callLog.toString(), "x");
    }
```

现在再次运行它，它会失败。很好，现在写出实现代码，来让测试通过吧：

```
class EnrollmentSet {
    ...
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    void addToStorage(Enrollment enrollment) {
        storage.add(enrollment);
    }
}
```

现在运行所有测试，它们应该能够全部通过。

改变 Enrollment 的构造函数，让所有已有的测试用例失败

现在，我们的 TODO 列表是空的。不过这可不代表我们没事做了。从用户例事中，我们很明显还有很多事要做。比如，检查学费，把选修信息保存到数据库等等。现在，我们来测试一下，看它会不会检查交费信息：

```
interface CourseFeeLookup {
    int getFee(String courseCode);
}

class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentCorrect() {
        Enrollment enrollment = new Enrollment("s001", "c001", new CashPayment(100));
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
```

```
                int getFee(String courseCode) {
                    return courseCode.equals("c001") ? 100 : 0;
                }
            });
            enrollmentSet.assertPaymentCorrect(enrollment);
        }
    }
```

注意到我们将一个 payment 对象作为 Enrollment 的第三个构造函数。这样看起来是不错。毕竟，一个 enrollment 必须有个交费情况。为了让编译通过，我们又在构造函数里面加了第三个参数：

```
class Enrollment {
    Enrollment(String studentId, String courseCode, Payment payment) {
        ...
    }
}
```

不过，现在原有的测试都被弄坏了：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAddEnrollment() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //没有这个构造函数
        ...
    }
    void testStudentRegistered() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //没有这个构造函数
        ...
        enrollmentSet.assertStudentRegistered(enrollment);
    }
    void testStudentUnregistered() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //没有这个构造函数
        ...
        enrollmentSet.assertStudentRegistered(enrollment);
        ...
    }
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //没有这个构造函数
        ...
        enrollmentSet.assertHasSeat(enrollment);
        ...
    }
}
```

但对于所有的测试用例，比如 testStudentRegistered 和 testAllSeatsTaken，它们根本不关心交费情况。这个说明它们不应该跟 Enrollment 一起运作，它们应该跟它们关心的东西运作，那我们就不应该增加后面那个构造函数。比如，testStudentRegistered 应该只跟学生 ID 一起运作,这句话听起来有点玄，不过看完下面就明白了。现在，我们增加下面的构造函数：

```
class Enrollment {
   ...
   Enrollment(String studentId, String courseCode) {
       ...
   }
   Enrollment(String studentId, String courseCode, Payment payment) {
       this.studentId = studentId;
       this.courseCode = courseCode;
       this.payment = payment;
   }
}
```

现在，逐个更改原来的测试代码，让它们不再使用原来的构造函数。

我们先来看第一个测试用例：testAddToStorage。它需要传递一个 Enrollment 给 addToStorage，因为 addToStorage 要将这个 enrollment 对象存放到 EnrollmentStorage 里面。但是这个 Enrollment 对象不需要什么数据，只要一个 Enrollment 实例就够了。所以我们将测试代码改为：

```
class EnrollmentSetTest extends TestCase {
   public void testAddToStorage() {
       final StringBuffer callLog = new StringBuffer();
       final Enrollment enrollment = new Enrollment(null, null, null);//用 null 作参数
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setStorage(new EnrollmentStorage() {
           public void add(Enrollment enrollmentToStore) {
              callLog.append("x");
              assertTrue(enrollmentToStore == enrollment);
           }
       });
       enrollmentSet.addToStorage(enrollment);
       assertEquals(callLog.toString(), "x");
   }
   ...
}
```

运行一下确保它能够通过。现在看一下 testStudentRegistered 方法。它关心的不是整个 enrollment 对象，它只需要一个学生 ID 就行了：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testStudentRegistered() {
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
           boolean isRegistered(String studentId) {
              return studentId.equals("s001");
           }
       });
       enrollmentSet.assertStudentRegistered("s001");
   }
}
```

现在代码肯定编译不过，因为 EnrollmentSet 里面的 assertStudentRegistered 方法的参数是一个 Enrollment 对象。我们可以修改一下，但是如果将它的参数改了，别的地方又会出问题了。目前我们是在做将两个参数的 Enrollment 构造函数改为三个参数的整合，没精力同时管那么多事情，所以将简单的做个重载函数，将原来的 assetStudentRegistered 里面的代码拷过来修改：

```
class EnrollmentSet {
   ...
   void assertStudentRegistered(Enrollment enrollment) {
       if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
           throw new StudentNotFoundException();
       }
   }
   void assertStudentRegistered(String studentId) {
       if (!studentRegistryChecker.isRegistered(studentId)) {
           throw new StudentNotFoundException();
       }
   }
}
```

然后增加一个 TODO：

//TODO 处理将 Enrollment 作为参数的 assertStudentRegistered 方法

现在再运行一下测试，它应该可以通过。再来看测试用例：testStudentUnregistered。它跟 testStudentRegistered 类似：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testStudentUnregistered() {
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
```

```
            boolean isRegistered(String studentId) {
                return false;
            }
        });
        try {
            enrollmentSet.assertStudentRegistered("s001");
            fail();
        } catch (StudentNotFoundException e) {
        }
    }
}
```

现在再来看 testAllSeatsTaken。它需要的是一个课程 ID 而不是一个 Enrollment 对象：

```
class EnrollmentSetTest extends TestCase {
    ...
    public void testAllSeatsTaken() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            public int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            public int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            public int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat("c001");
            fail();
        } catch (ClassFullException e) {
        }
    }
}
```

类似，我们要创建一个 assertHasSeat 的重载：

```
class EnrollmentSet {
   ...
   void assertHasSeat(Enrollment enrollment) {
       ...
   }
   void assertHasSeat(String courseCode) {
       if (enrollmentCounter.getNoSeatsTaken(courseCode)
           + reservationCounter.getNoSeatsReserved(courseCode)
           >= classSizeGetter.getClassSize(courseCode)) {
           throw new ClassFullException();
       }
   }
}
```

加个 TODO：

```
//TODO  处理将 Enrollment 作为参数的 assertStudentRegistered 方法
//TODO  处理将 Enrollment 作为参数的 assertHasSeat 方法
```

运行一下所有的测试，它们应该还是能通过。类似的，修改测试用例：testHasAFreeSeat：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testHasAFreeSeat() {
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
           int getClassSize(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
           int getNoSeatsTaken(String courseCode) {
              return courseCode.equals("c001") ? 1 : 0;
           }
       });
       enrollmentSet.setReservationCounter(new ReservationCounter() {
           int getNoSeatsReserved(String courseCode) {
              return 0;
           }
       });
       enrollmentSet.assertHasSeat("c001");
   }
}
```

运行所有的测试用例，它们还是可以通过。现在修改 testAllSeatsReservedOrTaken，让它也只是使用一个课程 ID：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testAllSeatsReservedOrTaken() {
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
           public int getClassSize(String courseCode) {
              return courseCode.equals("c001") ? 3 : 0;
           }
       });
       enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
           public int getNoSeatsTaken(String courseCode) {
              return courseCode.equals("c001") ? 2 : 0;
           }
       });
       enrollmentSet.setReservationCounter(new ReservationCounter() {
           public int getNoSeatsReserved(String courseCode) {
              return courseCode.equals("c001") ? 1 : 0;
           }
       });
       try {
           enrollmentSet.assertHasSeat("c001");
           fail();
       } catch (ClassFullException e) {
       }
   }
}
```

运行所有测试用例，还是能通过。好，再来看 testValidateBeforeAddToStorage。它只是测试调用顺序，压根儿不需要用到 Enrollment 对象。它可以开开心心的使用一个 null：

```
class EnrollmentSetTest extends TestCase {
   ...
   public void testValidateBeforeAddToStorage() {
       final StringBuffer callLog = new StringBuffer();
       EnrollmentSet enrollmentSet = new EnrollmentSet() {
           public void assertStudentRegistered(Enrollment enrollment) {
              callLog.append("x");
           }
           public void assertHasSeat(Enrollment enrollment) {
```

```
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        enrollmentSet.add(null);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

运行所有用例，它们还是可以通过。

现在，我们已经成功的从 2 参数的构造函数转变为 3 个参数了。现在我们把 2 个参数的构造函数删掉。然后运行所有的测试用例，还是全部通过。

现在，检查所有的 TODO：

//TODO 处理将 Enrollment 作为参数的 assertStudentRegistered 方法
//TODO 处理将 Enrollment 作为参数的 assertHasSeat 方法

先做第 1 个。看看旧版本的 assertStudentRegistered 方法被谁使用（有些 IDE 可以帮你做到）。我们发现只有下面一个地方用到：

```
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    ...
}
```

很简单，改成使用新版本的就行了：

```
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
```

```
    ...
}
```

运行一下测试用例，发现 testValidateBeforeAddToStorage 竟然抛出一个 NullPointerException：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
            void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        enrollmentSet.add(null);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

因为 add 方法里面，我们想从 enrollment 里面取出学生 ID。但是这个测试用例提供一个 null 给 add 方法。所以我们现在老老实实的创建一个 Enrollment 对象给 add 方法吧。此外，我们发现我们现在重载了 EnrollmentSet 里面老版本的 assertStudentRegistered 方法。同样改成新的：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                callLog.append("x");
            }
            void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
```

```
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

现在没有哪段代码使用旧版本的 assertStudentRegistered 了，删掉。
现在没有编译错误了。运行所有的测试用例。好，还是都能通过。

现在我们再检查最后一个 TODO：

//TODO 处理将 Enrollment 作为参数的 assertHasSeat 方法

这个跟前面的那个 TODO 很相似。因此，直接显示修改后的结果：

```
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment.getCourseCode());
        addToStorage(enrollment);
    }
    ...
}

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
```

```
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

现在看来，所有的测试都 OK 了，我们可以继续测试那个 payment 了，是不是？！？！

绝对不是!!!!!!!!!!!!!

我们要测试代码是不是如我们预期的调用了某一个方法，我们要判断两点：1.它确实调用了我们希望它调用的方法。2.它调用时传递过去当参数的对象，应该是我们预期的对象。

上面的测试用例里面，我们只验证它是不是调用了这些方法，却没有验证它传递的参数是否正确。所以，我们应该这样修改：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                assertEquals(studentId, "a");
                callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
                assertEquals(courseCode, "b");
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment("a", "b", null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

现在运行所有的测试，它们应该还是能全部通过。最后，我们准备测试交费了：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentCorrect() {
       Payment payment = new CashPayment(100);
       Enrollment enrollment = new Enrollment("s001", "c001", payment);
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
           int getFee(String courseCode) {
              return courseCode.equals("c001") ? 100 : 0;
           }
       });
       enrollmentSet.assertPaymentCorrect(enrollment);
   }
}
```

从之前的整合过程中，我们明白，它实际上需要的不是一个 Enrollment 对象。它只需要一个课程代码（去找到课程费用）：

```
class EnrollmentSetTest extends TestCase {
   ...
   void testPaymentCorrect() {
       Payment payment = new CashPayment(100);
       EnrollmentSet enrollmentSet = new EnrollmentSet();
       enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
           int getFee(String courseCode) {
              return courseCode.equals("c001") ? 100 : 0;
           }
       });
       enrollmentSet.assertPaymentCorrect("c001", payment);
   }
}
```

为了让它编译通过，我们现在要创建一个 CashPayment 类和空构造函数，一个空的 setCourseFeeLookup 方法和空的 assertPaymentCorrect 方法：

```
class CashPayment extends Payment {
   CashPayment(int amount) {
   }
}
class EnrollmentSet {
   ...
   void setCourseFeeLookup(CourseFeeLookup lookup) {
```

```
    }
    void assertPaymentCorrect(String string, Payment payment) {
    }
}
```

现在运行这个测试。它竟然通过了！这个说明这个测试还不足够。各中的原因，我想读者仔细看一下也明白了。现在，我们增加另一个测试，这个测试应该失败：交费金额不对的情况。我们只要简单的将交费的金额从 100 改为 101，然后期望它抛出一个 IncorrectPaymentException：

```
class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentIncorrect() {
        Payment payment = new CashPayment(100);
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
            public int getFee(String courseCode) {
                return courseCode.equals("c001") ? 101 : 0;
            }
        });
        try {
            enrollmentSet.assertPaymentCorrect("c001", payment);
            fail();
        } catch (IncorrectPaymentException e) {
        }
    }
}
```

为此，创建一个 IncorrectPaymentException 类。然后运行这个测试，它应该失败。现在，写出实现让这个测试可以通过：

```
class EnrollmentSet {
    CourseFeeLookup courseFeeLookup;
    ...
    void setCourseFeeLookup(CourseFeeLookup lookup) {
        this.courseFeeLookup = lookup;
    }
    void assertPaymentCorrect(String courseCode, Payment payment) {
        if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
            throw new IncorrectPaymentException();
        }
    }
}
```

现在，我们要创建一个 getAmount 方法，还有一堆相关的代码：

```
abstract class Payment {
   abstract int getAmount();
}

class CashPayment extends Payment {
   int amount;
   public CashPayment(int amount) {
       this.amount = amount;
   }
   int getAmount() {
       return amount;
   }
}
```

现在运行所有的测试用例，它们应该能全部通过。现在看起来我们已经实现了交费检查。
还没呢。

对 EnrollmentSet 的结构维持完整的概念

EnrollmentSet 有个 bug。如果我们对这个用户例事进行验收测试的话，我们就会发现，它在将选修信息保存到数据库前，并没有检查交费信息。现在，我们好好把 EnrollmentSet 从头看到尾：

```
class EnrollmentSet {
   EnrollmentStorage storage;
   StudentRegistryChecker studentRegistryChecker;
   EnrollmentCounter enrollmentCounter;
   ClassSizeGetter classSizeGetter;
   ReservationCounter reservationCounter;
   CourseFeeLookup courseFeeLookup;
   void setStorage(EnrollmentStorage storage) {
       this.storage = storage;
   }
   void add(Enrollment enrollment) {
       assertStudentRegistered(enrollment.getStudentId());
       assertHasSeat(enrollment.getCourseCode());
       //没有调用 assertPaymentCorrect!
       addToStorage(enrollment);
   }
```

```
    void assertStudentRegistered(String studentId) {
        if (!studentRegistryChecker.isRegistered(studentId)) {
            throw new StudentNotFoundException();
        }
    }
    void assertHasSeat(String courseCode) {
        if (enrollmentCounter.getNoSeatsTaken(courseCode)
            + reservationCounter.getNoSeatsReserved(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
        this.studentRegistryChecker = registryChecker;
    }
    void setClassSizeGetter(ClassSizeGetter sizeGetter) {
        this.classSizeGetter = sizeGetter;
    }
    void setEnrollmentCounter(EnrollmentCounter counter) {
        this.enrollmentCounter = counter;
    }
    void setReservationCounter(ReservationCounter counter) {
        this.reservationCounter = counter;
    }
    void addToStorage(Enrollment enrollment) {
        storage.add(enrollment);
    }
    void setCourseFeeLookup(CourseFeeLookup lookup) {
        this.courseFeeLookup = lookup;
    }
    void assertPaymentCorrect(String courseCode, Payment payment) {
        if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
            throw new IncorrectPaymentException();
        }
    }
}
```

我们测试了 assertPaymentCorrect，可是我们忘记测试 add 方法有没有调用这个方法了。事实上，在整个开发过程中，我们对整个 EnrollmentSet 类的想法是没有确定的。因此每个行为都分开测试了，很容易遗漏部分的功能。然后整合的 bug 就出现了。因此，我们要经常要审视一下整个类的结构，在脑海里面来维持一个完整的蓝图。

现在回到这个 bug。跟之前一样，写一个通不过的测试来重现这个 bug。我们可以增加

testValidateBeforeAddToStorage 方法：

```
class EnrollmentSetTest extends TestCase {
   void testValidateBeforeAddToStorage() {
       final StringBuffer callLog = new StringBuffer();
       final Payment payment = new CashPayment(5);
       EnrollmentSet enrollmentSet = new EnrollmentSet() {
           void assertStudentRegistered(String studentId) {
              assertEquals(studentId, "a");
              callLog.append("x");
           }
           void assertHasSeat(String courseCode) {
              assertEquals(courseCode, "b");
              callLog.append("y");
           }
           void assertPaymentCorrect(
              String courseCode,
              Payment payment2) {
              assertEquals(courseCode, "b");
              assertTrue(payment2 == payment);
              callLog.append("t");
           }
       };
       enrollmentSet.setStorage(new EnrollmentStorage() {
           void add(Enrollment enrollmentToStore) {
              callLog.append("z");
           }
       });
       Enrollment enrollment = new Enrollment("a", "b", payment);
       enrollmentSet.add(enrollment);
       assertEquals(callLog.toString(), "xytz");
   }
}
```

运行它，然后它失败了。很好，写下需要的代码：

```
class EnrollmentSet {
   void add(Enrollment enrollment) {
       assertStudentRegistered(enrollment.getStudentId());
       assertHasSeat(enrollment.getCourseCode());
       assertPaymentCorrect(enrollment.getCourseCode(), enrollment.getPayment());
       addToStorage(enrollment);
   }
```

```
}
```

现在运行所有测试用例，全部通过。

TDD 及它的优点

上面这种编程的方式，就叫“测试驱动开发 Test Driven Development (TDD)”，因为我们总是在写真正代码之前写一个通不过的测试，然后再写真正的代码，让测试通过。

跟测试后行的开发方式相比，它有如下好处：

1. 为了更容易的写单元测试，我们会广泛的使用接口（比如 StudentRegistryChecker 等）。这个会让单元测试代码很容易读跟写，因为测试代码里面没有多余的数据。如果我们不用 TDD 而是直接写实现的话，我们经常会使用现成的类（比如 StudentSet），测试为了调用现成的类，就不得不创建很多多余的数据，创建很巨型的对象，就像 Student 或者 Course。

2. 因为广泛的使用接口，我们的类之间就不会藕合（比如 EnrollmentSet 就一点都不知道 StudentSet 的存在），因此重用性更好。

3. 写单元测试的时候，很容易就可以为一个行为写一个测试用例，让它通过，然后为另一种行为写另一个测试用例。也就是说，整个任务会被划分成很多小的任务，独立完成。如果我们不用 TDD 而直接实现的话，我们很容易就会同时把所有的行为都实现了。这样花的时间长，而且在这相当长的时间里面，写的代码都是没有测试过，不能保证准确性的。相反的，用 TDD 的话，我们只实现要测的行为的代码。它只花费很少的时间（几分钟），而且可以马上测试。

要做什么，不要做什么

1. 不要在测试里面包含多余的数据（比如，如果你只需要一个学生 ID，那就不要创建整个 Enrollment 对象；如果你只想检查一个学生是否已经注册，用一个 StudentRegistryCheck 接口，而不是用整个 StudentSet 类。为此，你就会更多的使用接口和匿名类。

2. 如果你有一回，需要写一堆的代码来建立测试的上下文（比如，建立一个数据库连接），那你绝对不要忍受这种痛苦。如果它确实发生了，那就用接口吧（比如，EnrollmentStorage）。

3. 不在要两个测试里面测试了同样的东西（比如，当测试验证的时候，就不要测试存储的代码）。

4. 在写出通不过的测试之前，绝对不要写代码，除非代码非常简单。

5. 不要（至少尽量避免）一次性将所有的测试弄坏（比如，你想给 Enrollment 的构造函数加一个参数，不

要删除原来的构造函数，让它们共存，逐个将调用原有构造函数的地方改成调用新的构造函数，运行所有的测试用例后，在删除原来的构造函数)。

6.不要一次性写太多东西，或者做一些要花费几个小时的事情。尽量将这些东西划分为更小的行为。使用 TODO 列表。

7.每写完一个测试用例，在几分钟内通过它，并不是放在一边。

8.每做一个或者一些小改动后，就要运行所有测试用例。

9.先挑出你比较有兴趣的，或者可以从中尝到东西的任务，而不是先挑那些烦人的任务。

10.测试调用顺序的时候，一定要使用调用日记。

11.任何时候，脑中都要有个完整的概念。

引述

http://www.objectmentor.com/writeUps/TestDrivenDevelopment .
http://www.objectmentor.com/resources/articles/xpepisode.htm .
http://www.mockobjects.com .
http://www.testdriven.com .

章节练习

介绍

你要用 TDD 来完成。

你要使用 TODO 列表。

问题

1.写一些代码，计算一个指定目录（这目录至少 2GB 大小）下的文件数量。子目录下的文件忽略，并且子目录不当文件计算。

2.编写一段代码，删除一个目录，及里面的所有文件。要删除一个目录前，你必须要先清空里面的文件。

3.编写一段代码，为每个传真生成一个唯一的编号。编号有 3 部分组成，就像 1/2004/HR。第 1 部分是从 1 开始的序列号。下一个传真就是 2.然后依次加 1。2.第 2 部分是当前年份。第 3 部分是使用这个传真的部门 ID。

4.编写一段代码，检查两个职员对象是否相等。一个职员对象都有个 ID，拥有证书列表，一个直属上司（也是职员）。一个证书是由文字描述和颁发年份组成。

5.编写一个 EnrollmentDBStorage 类，实现 EnrollmentStorage 接口。它应该可以将一个 Enrollment 对象存到数据库中。你可以使用这章里面的数据表结构，也可以自己设计表。你可以自己假定数据库名称，使用驱动等等。

6.编写代码，计算指定课程已经被占的位置数（由它跟它父课程的选修情况决定）。你应该有个类，实现这章里面已经有的 EnrollmentCounter 接口。当你想要访问数据库时，用接口代替。
7.编写代码，计算指定课程已被预订的位置数（由它跟它父课程的预订情况决定）。你应该有个类，实现这章里面已经有的 ReservationCounter 接口。当你想要访问数据库时，用接口代替。你应该只计算那些还有效的预订。

提示

1.考虑使用下面的接口代替访问文件系统：

```
public interface FileLookup {
    public String[] getFilesIn(String dirPath);
}

public interface DirChecker {
    public boolean isDir(String path);
}

public interface FileSizeGetter {
    public long getSize(String path);
}
```

2.跟前一个类似。

3.考虑使用下面的接口代替系统时钟：

```
public interface YearGetter {
    public int getCurrentYear();
}
```

为了避免在不同的测试里面测试相同的东西，你可以想像 FaxCodeGenerator 的结构是这样的：

```
public class FaxCodeGenerator {
    public String getSequenceNo() {
       ...
    }
    public String generate() {
       return getSequenceNo() + ...
    }
}
```

然后分开测试 getSequenceNo 和 generate 方法。
或者，你也可以将 getSequenceNo 放在另一个类里面：

```
public class SequenceNumberGenerator {
```

```
    public String getSequenceNo() {
       ...
    }
}

public class FaxCodeGenerator {
    public String generate() {
       ...
    }
}
```

然后分开测试每个类。

4. 在测试里面，你会创建一些职员实例，证书列表的实例等等。尽量少使用一些没用的数据，合适的话可以多用一些 null。

5. 因为代码要访问数据库，你就不能用接口代替数据库了。所以，你必须创建一个数据库和一个表来运行这些测试。

6. 下面的接口可以让写测试变得更容易：

```
public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}
public interface EnrollmentSource {
    public List getEnrollmentsFor(String courseCode);
}

In addition, you can use the following interface to replace the database:
public interface CourseSource {
    public Course getCourse(String courseCode);
}
```

7. 跟之前那个类似。不过，测试代码可能会跟前面写的差不多。如果可以移除这些重复的话，更好了（测试代码跟实现代码的重复）。

解决方法示例

1. 写一些代码，计算一个指定目录（这目录至少 2GB 大小）下的文件数量。子目录下的文件忽略，并且子目录不当文件计算。

这里面主要的问题是建立测试上下文，我们要创建一个目录，并将一些文件和目录放进去。特别了，里面的文件总大小要超过 2GB。我们要保证被删除的目录初始时是空的，否则我们就要先清空里面内容。当然，我们还要保证目录里面没有有用的文件。因此，建立测试上下文要花费很多的精力，代码，执行时间（来创建大文件）和硬盘空间。为了解决这个问题，我们可以用一些接口来代替系统文件：

```
public interface FileLookup {
    public String[] getFilesIn(String dirPath);
}
public interface DirChecker {
    public boolean isDir(String path);
}
public interface FileSizeGetter {
    public long getSize(String path);
}
```

测试：

```
public class FileCounterTest extends TestCase {
    public void testCount() {
       final long KB = 1024;
       final long MB = 1024 * KB;
       final long GB = 1024 * MB;
       FileCounter fileCounter = new FileCounter();
       fileCounter.setFileSizeGetter(new FileSizeGetter() {
           public long getSize(String path) {
              if (path.equals("dir/a")) {
                  return 2 * GB;
              }
              if (path.equals("dir/b")) {
                  return 2 * GB - 1;
              }
              if (path.equals("dir/c")) {
                  return 2 * GB + 1;
              }
              fail();
              return 0;
           }
       });
       fileCounter.setDirChecker(new DirChecker() {
           public boolean isDir(String path) {
              return false;
           }
```

```
        });
        fileCounter.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                if (dirPath.equals("dir")) {
                    return new String[] { "a", "b", "c" };
                }
                return null;
            }
        });
        assertEquals(fileCounter.countLargeFilesIn("dir"), 2);
    }
    public void testIgnoreDirectories() {
        FileCounter fileCounter = new FileCounter();
        fileCounter.setFileSizeGetter(new FileSizeGetter() {
            public long getSize(String path) {
                fail();
                return 0;
            }
        });
        fileCounter.setDirChecker(new DirChecker() {
            public boolean isDir(String path) {
                return true;
            }
        });
        fileCounter.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                if (dirPath.equals("dir")) {
                    return new String[] { "d1" };
                }
                return null;
            }
        });
        assertEquals(fileCounter.countLargeFilesIn("dir"), 0);
    }
}
```

实现代码：

```
public class FileCounter {
    private FileLookup fileLookup;
    private FileSizeGetter fileSizeGetter;
    private DirChecker dirChecker;
    public void setFileLookup(FileLookup lookup) {
```

```
        this.fileLookup = lookup;
    }
    public void setDirChecker(DirChecker checker) {
        this.dirChecker = checker;
    }
    public void setFileSizeGetter(FileSizeGetter getter) {
        this.fileSizeGetter = getter;
    }
    public int countLargeFilesIn(String pathToDir) {
        final long KB = 1024;
        final long MB = 1024 * KB;
        final long GB = 1024 * MB;
        String files[] = fileLookup.getFilesIn(pathToDir);
        int noLargeFiles = 0;
        for (int i = 0; i < files.length; i++) {
           if (!dirChecker.isDir(files[i])) {
               if (fileSizeGetter.getSize(pathToDir + "/" + files[i])
                   >= 2 * GB) {
                   noLargeFiles++;
               }
           }
        }
        return noLargeFiles;
     }
}
```

最后，我们要用真正的文件系统实现这些接口。因为这些代码不用单元测试，所以最好尽量简洁：

```
public class FileLookupInJava implements FileLookup {
    public String[] getFilesIn(String dirPath) {
       return new File(dirPath).list();
    }
}
public class FileSizeGetterInJava implements FileSizeGetter {
    public long getSize(String path) {
       return new File(path).length();
    }
}
public class DirCheckerInJava implements DirChecker {
    public boolean isDir(String path) {
       return new File(path).isDirectory();
    }
}
```

FileCounter 在缺省情况下是使用这些具体实现类的：

```
public class FileCounter {
    public FileCounter() {
        setFileLookup(new FileLookupInJava());
        setFileSizeGetter(new FileSizeGetterInJava());
        setDirChecker(new DirCheckerInJava());
    }
    ...
}
```

（译者的感觉：这里面提供的这种解藕的方式，一个接口，两个实现，一个实现类是在测试代码中的假类，一个是正式代码里面的实现类，这样写法实在有点恶心，代码不好读不说，还要多增加一堆接口。其实有其他方法可以实现类似的功能：

1. Java 的匿名类不仅可以实现接口，还可以重载类的：

比如实现代码中:

```
public class DirChecker{
    public boolean isDir(String path){
        return new File(path).isDirectory();
    }
}
```

测试代码中：

```
fileCounter.setDirChecker(new DirChecker() {
    @Override
    public boolean isDir(String path) {
        return false;//这个返回的是要用测试用的假值。
    }
});
```

1. 有个 EasyMock 包，它也有提供假类的功能：

比如实现代码中:

```
public class DirChecker{
    public boolean isDir(String path){
        return new File(path).isDirectory();
    }
}
```

测试代码中：

```
DirChecker dirChecker=EasyMock.createMock(DirChecker.class);
```

EasyMock.expect(dirChecker.isDir()).andReturn(false);//这段代码不仅会返回 false 的假值，还可以验证被测代码有没有调用 isDir 这个方法。具体可以参考 EasyMock 的用法。

下面的解决方法中都有奇多无比的接口，几乎每个类之间都通过接口交互。希望读者可以耐心的看下去，因为我们要看的是 TDD 的思路，大家一定要坚持啊！
)

2.编写一段代码，删除一个目录，及里面的所有文件。要删除一个目录前，你必须要先清空里面的文件。

像之前那问题一样，我们要用一些接口来代替文件系统：

```
public interface FileLookup {
    public String[] getFilesIn(String path);
}
public interface FileRemover {
    public void delete(String path);
}
```

测试：

```
public class RecursiveFileRemoverTest extends TestCase {
    public void testEmpty() {
        final StringBuffer deleteLog = new StringBuffer();
        RecursiveFileRemover remover = new RecursiveFileRemover();
        remover.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                return null;
            }
        });
        remover.setFileRemover(new FileRemover() {
            public void delete(String path) {
                deleteLog.append("<" + path + ">");
            }
        });
        remover.delete("dir");
        assertEquals(deleteLog.toString(), "<dir>");
    }
    public void testDeleteFilesFirst() {
        final StringBuffer deleteLog = new StringBuffer();
        RecursiveFileRemover remover = new RecursiveFileRemover();
        remover.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                if (dirPath.equals("dir")) {
```

```
                return new String[] { "a", "b" };
            }
            return null;
        }
    });
    remover.setFileRemover(new FileRemover() {
        public void delete(String path) {
            deleteLog.append("<" + path + ">");
        }
    });
    remover.delete("dir");
    assertEquals(deleteLog.toString(), "<dir/a><dir/b><dir>");
  }
  public void testRecursion() {
    final StringBuffer deleteLog = new StringBuffer();
    RecursiveFileRemover remover = new RecursiveFileRemover();
    remover.setFileLookup(new FileLookup() {
        public String[] getFilesIn(String dirPath) {
            if (dirPath.equals("d1")) {
                return new String[] { "d2" };
            }
            if (dirPath.equals("d1/d2")) {
                return new String[] { "f1" };
            }
            return null;
        }
    });
    remover.setFileRemover(new FileRemover() {
        public void delete(String path) {
            deleteLog.append("<" + path + ">");
        }
    });
    remover.delete("d1");
    assertEquals(deleteLog.toString(), "<d1/d2/f1><d1/d2><d1>");
  }
}
```

实现代码

```
public class RecursiveFileRemover {
  private FileLookup fileLookup;
  private FileRemover fileRemover;
  public void setFileLookup(FileLookup lookup) {
    this.fileLookup = lookup;
```

```
        }
        public void setFileRemover(FileRemover remover) {
            this.fileRemover = remover;
        }
        public void delete(String path) {
            String filesInDir[] = fileLookup.getFilesIn(path);
            if (filesInDir != null) {
                for (int i = 0; i < filesInDir.length; i++) {
                    String pathToChild = path + "/" + filesInDir[i];
                    delete(pathToChild);
                }
            }
            fileRemover.delete(path);
        }
    }
```

RecursiveFileRemover 缺省时应该是使用 FileLookup 和 FileRemover 的实现类，这些实现类依赖于文件系统：

```
    public class FileLookupInJava implements FileLookup {
        public String[] getFilesIn(String path) {
            return new File(path).list();
        }
    }
    public class FileRemoverInJava implements FileRemover {
        public void delete(String path) {
            new File(path).delete();
        }
    }
    public class RecursiveFileRemover {
        public RecursiveFileRemover() {
            setFileLookup(new FileLookupInJava());
            setFileRemover(new FileRemoverInJava());
        }
        ...
    }
```

3.编写一段代码，为每个传真生成一个唯一的编号。编号有 3 部分组成，就像 1/2004/HR。第 1 部分是从 1 开始的序列号。下一个传真就是 2.然后依次加 1。2.第 2 部分是当前年份。第 3 部分是使用这个传真的部门 ID。

这里面最明显的问题就是编号要找出当前年份。为了建立上下文，让它生成 1/2004/HR，我们可能要将系统时钟调为 2004 年，否则如果年份是 2005 的话，测试肯定过不了。改变系统时间是一件很恶心的做法。一个好的方法就是用一个接口代替系统时钟：

```
public interface YearGetter {
    public int getCurrentYear();
}
```

然后我们就可以很简单的写测试代码了。不过，我们写测试的时候要考虑两个行为：一个是传真编号由序列号，当前年份和部门 ID 组成；还有一个是序列号是递增的。所以分开测试和实现这两个行为会更容易。

下面是传真编号行为的测试和实现：

```
public interface NumberGenerator {
   public void setSeqNo(int seqNo);
   public int generate();
}

public class FaxCodeGeneratorTest extends TestCase {
   public void testCombineFormat() {
       FaxCodeGenerator generator = new FaxCodeGenerator("a");
       generator.setNumberGenerator(new NumberGenerator() {
           public int generate() {
              return 3;
           }
       });
       generator.setYearGetter(new YearGetter() {
           public int getCurrentYear() {
              return 2004;
           }
       });
       assertEquals(generator.generate(), "3/2004/a");
   }
   public void testInitialSeqNo() {
       final StringBuffer callLog = new StringBuffer();
       FaxCodeGenerator generator = new FaxCodeGenerator("a");
       generator.setNumberGenerator(new NumberGenerator() {
           public void setSeqNo(int seqNo) {
              assertEquals(seqNo, 1);
              callLog.append("x");
           }
           public int generate() {
              fail();
              return 0;
           }
       });
```

```
        assertEquals(callLog.toString(), "x");
    }
}

public class FaxCodeGenerator {
    private final static int INITIAL_SEQ_NO=1;
    private NumberGenerator numberGenerator;
    private String departId;
    private YearGetter yearGetter;
    public FaxCodeGenerator(String departId) {
        this.departId = departId;
    }
    public void setYearGetter(YearGetter getter) {
        this.yearGetter = getter;
    }
    public String generate() {
        return numberGenerator.generate()
            + "/"
            + yearGetter.getCurrentYear()
            + "/"
            + departId;
    }

    public void setNumberGenerator(NumberGenerator generator) {
        this.numberGenerator = generator;
        this.numberGenerator.setSeqNo(INITIAL_SEQ_NO);
    }
}
```

下面是序列号行为的测试和实现：

```
public class SeqNoGeneratorTest extends TestCase {
    public void testInitialValue() {
        SeqNoGenerator generator = new SeqNoGenerator(10);
        assertEquals(generator.generate(), 10);
    }
    public void testIncrementAfterGenerate() {
        SeqNoGenerator generator = new SeqNoGenerator(10);
        generator.generate();
        assertEquals(generator.generate(), 11);
    }
}
```

```
public class SeqNoGenerator implements NumberGenerator {
   private int seqNo;
   public SeqNoGenerator(int initialSeqNo) {
       this.seqNo = initialSeqNo;
   }
   public int generate() {
       return seqNo++;
   }
   public void setSeqNo(int seqNo) {
       this.seqNo = seqNo;
   }
}
```

最后我们要使用系统时钟来实现 YearGetter 接口。为了保证代码的简洁，我们也要将它分为两个行为：从系统中取得当前时间和从时间中取出年份。前一个测试不了，不过后一个测试得了。后一个的代码是：

```
public class YearGetterFromCalendarTest extends TestCase {
   public void testGetCurrentYear() {
       GregorianCalendar calendar = new GregorianCalendar(2003, 0, 23);
       YearGetterFromCalendar getter = new YearGetterFromCalendar(calendar);
       assertEquals(getter.getCurrentYear(), 2003);
   }
}

public class YearGetterFromCalendar implements YearGetter {
   private GregorianCalendar calendar;
   public YearGetterFromCalendar(GregorianCalendar calendar) {
       this.calendar = calendar;
   }
   public int getCurrentYear() {
       return calendar.get(Calendar.YEAR);
   }
}
```

FaxCodeGenerator 缺省时要使用 YearGetterFromCalendar 这个实现类：

```
public class FaxCodeGenerator {
    public FaxCodeGenerator(String departId) {
       this.departId = departId;
       setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
    }
    ...
}
```

它缺省时也要使用 SeqNoGenerator 这个实现类：

```
public class FaxCodeGenerator {
    public FaxCodeGenerator(String departId) {
        this.departId = departId;
        setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
        setNumberGenerator(new SeqNoGenerator(INITIAL_SEQ_NO));
    }
    ...
}
```

4.编写一段代码，检查两个职员对象是否相等。一个职员对象都有个 ID，拥有证书列表，一个直属上司（也是职员）。一个证书是由文字描述和颁发年份组成。

这里面的困难就是，我们要建立一堆没用的数据（2 个职员对象），但大多数的数据都没用。这里面 Employee 类真正的行为是怎么判断两个对象相等。它要检查它们的 3 个属性都相等（ID，证书列表，上司）。而它们的真正 ID，证书列表和上司其实无关紧要。

测试代码：

```
public interface EqualityChecker {
    public boolean eachEquals(Object objList1[], Object objList2[]);
}

public class EmployeeTest extends TestCase {
    public void testEquals() {
        final StringBuffer callLog = new StringBuffer();
        final QualificationList qualiList1 = new QualificationList();
        final QualificationList qualiList2 = new QualificationList();
        final Employee superior1 = new Employee(null, null, null);
        final Employee superior2 = new Employee(null, null, null);
        Employee employee1 = new Employee("id1", qualiList1, superior1);
        Employee employee2 = new Employee("id2", qualiList2, superior2);
        employee1.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                callLog.append("x");
                assertEquals(objList1.length, 3);
                assertEquals(objList1[0], "id1");
                assertSame(objList1[1], qualiList1);
                //assertSame(X, Y) 表示 assertTrue(X==Y)，由 JUnit 提供。
                assertSame(objList1[2], superior1);
```

```
                assertEquals(objList2.length, 3);
                assertEquals(objList2[0], "id2");
                assertSame(objList2[1], qualiList2);
                assertSame(objList2[2], superior2);
                return true;
            }
        });
        assertTrue(employee1.equals(employee2));
        assertEquals(callLog.toString(), "x");
    }
    public void testNonEmployee() {
        Employee employee = new Employee("id", null, null);
        employee.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                fail();
                return false;
            }
        });
        assertFalse(employee.equals("non-employee"));
    }
}
```

实现代码：

```
public class Employee {
    private String id;
    private QualificationList qualiList;
    private Employee superior;
    private EqualityChecker equalityChecker;
    public Employee(
        String id,
        QualificationList qualiList,
        Employee superior) {
        this.id = id;
        this.qualiList = qualiList;
        this.superior = superior;
    }
    public boolean equals(Object obj) {
        return obj instanceof Employee ? equals((Employee) obj) : false;
    }
    private boolean equals(Employee employee) {
        return equalityChecker.eachEquals(
            getElementsForEqualityCheck(),
```

```
            employee.getElementsForEqualityCheck());
    }
    private Object[] getElementsForEqualityCheck() {
        return new Object[] { id, qualiList, superior };
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
}
```

我们要提供 EqualityChecker 的实现类：

```
public class ShortCircuitEqualityCheckerTest extends TestCase {
    public void testOneElement() {
        ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
        assertTrue(
            checker.eachEquals(new Object[] { "a" }, new Object[] { "a" }));
        assertFalse(
            checker.eachEquals(new Object[] { "a" }, new Object[] { "b" }));
    }
    public void testNotSameLength() {
        ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
        assertFalse(checker.eachEquals(new Object[] { "a" }, new Object[0]));
    }
    public void testFirstElementNotEqual() {
        ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
        assertFalse(
            checker.eachEquals(
                new Object[] { "a", "b" },
                new Object[] { "c", "b" }));
    }
    public void testFirstElementEqual() {
        ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
        assertTrue(
            checker.eachEquals(
                new Object[] { "a", "b" },
                new Object[] { "a", "b" }));
        assertFalse(
            checker.eachEquals(
                new Object[] { "a", "b" },
                new Object[] { "a", "c" }));
    }
}
```

Employee 类缺省时要使用这个实现：

```
public class Employee {
   public Employee(
       String id,
       QualificationList qualiList,
       Employee superior) {
       ...
       equalityChecker = new ShortCircuitEqualityChecker();
   }
   ...
}
```

实现完 Employee 对象后，我们还要实现 QualificationList 里面使用的 EqualityChecker 的实现类：

```
public class QualificationListTest extends TestCase {
   public void testEquals() {
       final StringBuffer callLog = new StringBuffer();
       QualificationList list1 = new QualificationList() {
           public List getQualifications() {
              List qualifications = new ArrayList();
              qualifications.add("a");
              qualifications.add("b");
              return qualifications;
           }
       };
       QualificationList list2 = new QualificationList() {
           public List getQualifications() {
              List qualifications = new ArrayList();
              qualifications.add("c");
              qualifications.add("d");
              return qualifications;
          }
       };
       list1.setEqualityChecker(new EqualityChecker() {
          public boolean eachEquals(Object[] objList1, Object[] objList2) {
              callLog.append("x");
              assertEquals(objList1.length, 2);
              assertEquals(objList1[0], "a");
              assertEquals(objList1[1], "b");
              assertEquals(objList2.length, 2);
              assertEquals(objList2[0], "c");
```

```
                assertEquals(objList2[1], "d");
                return true;
            }
        });
        assertTrue(list1.equals(list2));
        assertEquals(callLog.toString(), "x");
    }
    public void testNonQualificationList() {
        QualificationList list = new QualificationList();
        list.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                fail();
                return true;
            }
        });
        assertFalse(list.equals("non-qualification-list"));
    }
}

public class QualificationList {
    private EqualityChecker equalityChecker;
    public QualificationList() {
        setEqualityChecker(new ShortCircuitEqualityChecker());
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
    public boolean equals(Object obj) {
        return obj instanceof QualificationList
            && equals((QualificationList) obj);
    }
    public List getQualifications() {
        return null;  //TODO: 需要的时候实现它。
    }
    private boolean equals(QualificationList list) {
        return equalityChecker.eachEquals(
            getQualifications().toArray(),
            list.getQualifications().toArray());
    }
}
```

接着，我们还需要实现 Qualification 里面使用的 EqualityChecker 的实现类：

```
public class QualificationTest extends TestCase {
   public void testEquals() {
      Qualification qualification1 = new Qualification("desc1", 2003);
      Qualification qualification2 = new Qualification("desc2", 2004);
      qualification1.setEqualityChecker(new EqualityChecker() {
          public boolean eachEquals(Object[] objList1, Object[] objList2) {
             assertEquals(objList1.length, 2);
             assertEquals(objList1[0], "desc1");
             assertEquals(objList1[1], new Integer(2003));
             assertEquals(objList2.length, 2);
             assertEquals(objList2[0], "desc2");
             assertEquals(objList2[1], new Integer(2004));
             return true;
          }
      });
      assertTrue(qualification1.equals(qualification2));
    }
    public void testNotQualification() {
      Qualification qualification = new Qualification("desc", 2003);
      qualification.setEqualityChecker(new EqualityChecker() {
          public boolean eachEquals(Object[] objList1, Object[] objList2) {
             fail();
             return false;
          }
      });
      assertFalse(qualification.equals("non-qualification"));
    }
}
public class Qualification {
    private String desc;
    private int yearWhenAchieved;
    private EqualityChecker equalityChecker;
    public Qualification(String desc, int yearWhenAchieved) {
      this.desc = desc;
      this.yearWhenAchieved = yearWhenAchieved;
      setEqualityChecker(new ShortCircuitEqualityChecker());
    }
    public boolean equals(Object obj) {
      return (obj instanceof Qualification) && equals((Qualification) obj);
    }
    public boolean equals(Qualification qualification) {
      return equalityChecker.eachEquals(
          makeElementsForEqualityCheck(),
```

```
            qualification.makeElementsForEqualityCheck());
    }
    private Object[] makeElementsForEqualityCheck() {
        return new Object[] { desc, new Integer(yearWhenAchieved)};
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
}
```

5.编写一个 EnrollmentDBStorage 类，实现 EnrollmentStorage 接口。它应该可以将一个 Enrollment 对象存到数据库中。你可以使用这章里面的数据表结构，也可以自己设计表。你可以自己假定数据库名称，使用驱动等等。

```
public class EnrollmentDBStorageTest extends TestCase {
    private Connection conn;
    protected void setUp() throws Exception {
        Class.forName("org.postgresql.Driver");
        conn =
            DriverManager.getConnection(
                "jdbc:postgresql://localhost/testdb",
                "testuser",
                "testpassword");
        deleteAllEnrollments();
    }
    protected void tearDown() throws Exception {
        conn.close();
    }
    private void deleteAllEnrollments() throws SQLException {
        PreparedStatement st =
            conn.prepareStatement("delete from enrollments");
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    public void testAdd() throws SQLException {
        EnrollmentDBStorage storage = new EnrollmentDBStorage(conn);
        Date enrolDate = new GregorianCalendar(2004, 0, 28).getTime();
        Payment payment = new CashPayment(200);
        Enrollment enrollment =
            new Enrollment("s001", "c001", enrolDate, payment);
        storage.add(enrollment);
```

```
        PreparedStatement st =
           conn.prepareStatement(
              "select * from enrollments where studentId=? and courseCode=?");
        try {
           st.setString(1, "s001");
           st.setString(2, "c001");
           ResultSet rs = st.executeQuery();
           assertTrue(rs.next());
           assertEquals(rs.getDate("enrolDate"), enrolDate);
           assertEquals(rs.getInt("amount"), 200);
           assertEquals(rs.getString("paymentType"), "Cash");
           assertNull(rs.getObject("cardNo"));
           //assertNull(X) 等于 assertTrue(X == null)。JUnit 提供
           assertNull(rs.getObject("expiryDate"));
           assertNull(rs.getObject("nameOnCard"));
           assertFalse(rs.next());
        } finally {
           st.close();
        }
    }
}

public class EnrollmentDBStorage implements EnrollmentStorage {
    private Connection conn;

    public EnrollmentDBStorage(Connection conn) {
        this.conn = conn;
    }
    public void add(Enrollment enrollment) {
        try {
           PreparedStatement st =
              conn.prepareStatement(
                 "insert into enrollments values(?,?,?,?,?,?,?,?)");
            try {
              st.setString(1, enrollment.getCourseCode());
              st.setString(2, enrollment.getStudentId());
              st.setDate(
                 3,
                 new java.sql.Date(enrollment.getEnrolDate().getTime()));
              CashPayment payment = (CashPayment) enrollment.getPayment();
              st.setInt(4, payment.getAmount());
              st.setString(5, "Cash");
              st.setNull(6, Types.VARCHAR);
```

```
                st.setNull(7, Types.DATE);
                st.setNull(8, Types.VARCHAR);
                st.executeUpdate();
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

6.编写代码，计算指定课程已经被占的位置数（由它跟它父课程的选修情况决定）。你应该有个类，实现这章里面已经有的 EnrollmentCounter 接口。当你想要访问数据库时，用接口代替。

测试：

```
public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}
public interface EnrollmentSource {
    public List getEnrollmentsFor(String courseCode);
}
public class EnrollmentCounterFromSrcTest extends TestCase {
    public void testConsiderAncestor() {
        EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc() {
            public int getNoEnollmentsForSingleCourse(String courseCode) {
                if (courseCode.equals("c001")) {
                    return 4;
                }
                if (courseCode.equals("c002")) {
                    return 2;
                }
                if (courseCode.equals("c003")) {
                    return 1;
                }
                fail();
                return 0;
            }
        };
        counter.setCourseParentGetter(new CourseParentGetter() {
            public String getParentCode(String courseCode) {
                if (courseCode.equals("c003")) {
```

```
                return "c002";
            }
            if (courseCode.equals("c002")) {
                return "c001";
            }
            if (courseCode.equals("c001")) {
                return null;
            }
            fail();
            return null;
        }
    });
    assertEquals(counter.getNoSeatsTaken("c003"), 7);
  }
  public void testSingleCourse() {
    EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc();
    counter.setEnrollmentSource(new EnrollmentSource() {
        public List getEnrollmentsFor(String courseCode) {
            List enrollments = new ArrayList();
            enrollments.add("e1");
            enrollments.add("e2");
            return enrollments;
        }
    });
    assertEquals(counter.getNoEnollmentsForSingleCourse("c001"), 2);
  }
}
```

实现：

```
public class EnrollmentCounterFromSrc implements EnrollmentCounter {
  private EnrollmentSource enrollmentSource;
  private CourseParentGetter courseParentGetter;
  public int getNoSeatsTaken(String courseCode) {
    int noSeatsTaken = 0;
    for (;;) {
        noSeatsTaken += getNoEnollmentsForSingleCourse(courseCode);
        courseCode = courseParentGetter.getParentCode(courseCode);
        if (courseCode == null) {
            return noSeatsTaken;
        }
    }
  }
```

```
    public int getNoEnollmentsForSingleCourse(String courseCode) {
        return enrollmentSource.getEnrollmentsFor(courseCode).size();
    }
    public void setCourseParentGetter(CourseParentGetter getter) {
        this.courseParentGetter = getter;
    }
    public void setEnrollmentSource(EnrollmentSource source) {
        this.enrollmentSource = source;
    }
}
```

我们要还提供 CourseParentGetter 的一个实现类，它从一个 CourseSource 取得课程信息：

```
public interface CourseSource {
    public Course getCourse(String courseCode);
}

public class CourseParentGetterFromSrcTest extends TestCase {
    public void testHasParent() {
        CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
        getter.setCourseSource(new CourseSource() {
            public Course getCourse(String courseCode) {
                Course c002 = new Course() {
                    public String getParentCode() {
                        return "c001";
                    }
                };
                return courseCode.equals("c002") ? c002 : null;
            }
        });
        assertEquals(getter.getParentCode("c002"), "c001");
    }
    public void testHasNoParent() {
        CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
        getter.setCourseSource(new CourseSource() {
            public Course getCourse(String courseCode) {
                Course c002 = new Course() {
                    public String getParentCode() {
                        return null;
                    }
                };
                return courseCode.equals("c002") ? c002 : null;
            }
```

```
        });
        assertNull(getter.getParentCode("c002"));
    }
}
```

实现代码：

```
public class CourseParentGetterFromSrc implements CourseParentGetter {
   private CourseSource courseSource;
   public String getParentCode(String courseCode) {
       return courseSource.getCourse(courseCode).getParentCode();
   }
   public void setCourseSource(CourseSource source) {
       this.courseSource = source;
   }
}

public class CourseParentGetterFromSrc implements CourseParentGetter {
   public CourseParentGetterFromSrc() {
       setCourseSource(new CourseDBSource());
   }
   ...
}

public class EnrollmentCounterFromSrc implements EnrollmentCounter {
    public EnrollmentCounterFromSrc() {
       setEnrollmentSource(new EnrollmentDBSource());
       setCourseParentGetter(new CourseParentGetterFromSrc());
    }
    ...
}
```

7.编写代码，计算指定课程已被预订的位置数（由它跟它父课程的预订情况决定）。你应该有个类，实现这章里面已经有的 ReservationCounter 接口。当你想要访问数据库时，用接口代替。你应该只计算那些还有效的预订。

当我们在做这个的时候，我们发现，这个测试跟前一个问题的测试很相似。它说明它们共用一些相同的逻辑：它们都要计算从当前课程，当前课程的父课程，父课程的父课程等里面取出的数目的总和。所以，我们可以将同样的逻辑抽取出来，分开测试：

```
public interface CourseIntGetter {
    public int getInt(String courseCode);
}
```

```
public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}

public class CourseAncestorsTraverserTest extends TestCase {
    public void testSum() {
        CourseAncestorsTraverser traverser = new CourseAncestorsTraverser();
        traverser.setCourseParentGetter(new CourseParentGetter() {
            public String getParentCode(String courseCode) {
                if (courseCode.equals("c003")) {
                    return "c002";
                }
                if (courseCode.equals("c002")) {
                    return "c001";
                }
                if (courseCode.equals("c001")) {
                    return null;
                }
                fail();
                return null;
            }
        });
        CourseIntGetter intGetter = new CourseIntGetter() {
            public int getInt(String courseCode) {
                if (courseCode.equals("c001")) {
                    return 1;
                }
                if (courseCode.equals("c002")) {
                    return 2;
                }
                if (courseCode.equals("c003")) {
                    return 5;
                }
                fail();
                return 0;
            }
        };
        assertEquals(traverser.sum("c003", intGetter), 8);
    }
}
```

实现类：

```
public class CourseAncestorsTraverser {
   private CourseParentGetter courseParentGetter;
   public void setCourseParentGetter(CourseParentGetter getter) {
       this.courseParentGetter = getter;
   }
   public int sum(String courseCode, CourseIntGetter intGetter) {
       int sum = 0;
       for (;;) {
           sum += intGetter.getInt(courseCode);
          courseCode = courseParentGetter.getParentCode(courseCode);
           if (courseCode == null) {
              return sum;
           }
       }
   }
}

public class CourseParentGetterFromSrc implements CourseParentGetter {
   ...
}
public class CourseAncestorsTraverser {
   public CourseAncestorsTraverser() {
       setCourseParentGetter(new CourseParentGetterFromSrc());
   }
   ...
}
```

为了取得已选人数和预订人数，我们只需要提供 CourseIntGetter 的不同接口。 对于已选人数：

```
public class CourseNoEnrollmentsGetterTest extends TestCase {
   public void testCount() {
       CourseNoEnrollmentsGetter getter = new CourseNoEnrollmentsGetter();
       getter.setEnrollmentSource(new EnrollmentSource() {
          public List getEnrollmentsFor(String courseCode) {
              if (courseCode.equals("c001")) {
                  List enrollments = new ArrayList();
                  enrollments.add("e1");
                  enrollments.add("e2");
                  return enrollments;
              }
              fail();
              return null;
          }
```

```
        });
        assertEquals(getter.getInt("c001"), 2);
    }
}

public class CourseNoEnrollmentsGetter implements CourseIntGetter {
    private EnrollmentSource enrollmentSource;
    public void setEnrollmentSource(EnrollmentSource source) {
        this.enrollmentSource = source;
    }
    public int getInt(String courseCode) {
        return enrollmentSource.getEnrollmentsFor(courseCode).size();
    }
}

public class CourseNoEnrollmentsGetter implements CourseIntGetter {
    public CourseNoEnrollmentsGetter() {
        setEnrollmentSource(new EnrollmentDBSource());
    }
    ...
}
```

最后，CourseAncestorsTraverser 也可以当做一个 EnrollmentCounter:

```
public class CourseAncestorsTraverser implements EnrollmentCounter {
    ...
    public int getNoSeatsTaken(String courseCode) {
        return sum(courseCode, new CourseNoEnrollmentsGetter());
    }
}
```

对于预订人数，它也类似：

```
public class CourseNoReservationsGetterTest extends TestCase {
    public void testCountOnlyActive() {
        CourseNoReservationsGetter getter = new CourseNoReservationsGetter();
        getter.setReservationSource(new ReservationSource() {
            public List getReservationsFor(String courseCode) {
                if (courseCode.equals("c001")) {
                    List reservations = new ArrayList();
                    reservations.add(new Reservation() {
                        public boolean isActive() {
                            return true;
```

```
                    }
                });
                reservations.add(new Reservation() {
                    public boolean isActive() {
                        return false;
                    }
                });
                reservations.add(new Reservation() {
                    public boolean isActive() {
                        return true;
                    }
                 });
                 return reservations;
            }
            fail();
            return null;
        }
    });
    assertEquals(getter.getInt("c001"), 2);
  }
}

public class CourseNoReservationsGetter implements CourseIntGetter {
  private ReservationSource reservationSource;

  public CourseNoReservationsGetter() {
      setReservationSource(new ReservationDBSource());
  }
  public int getInt(String courseCode) {
      int result = 0;
      List reservations = reservationSource.getReservationsFor(courseCode);
      for (Iterator iter = reservations.iterator(); iter.hasNext();) {
         Reservation reservation = (Reservation) iter.next();
         if (reservation.isActive()) {
             result++;
          }
      }
      return result;
  }
  public void setReservationSource(ReservationSource source) {
      this.reservationSource = source;
  }
}
```

```
public class CourseAncestorsTraverser
   implements EnrollmentCounter, ReservationCounter {
   ...
   public int getNoSeatsReserved(String courseCode) {
       return sum(courseCode, new CourseNoReservationsGetter());
   }
}
```

Reservation 类自身还要有个方法，用来检查它是否有效。为此，我们用一个接口来代替系统时钟：

```
public interface Clock {
   public Date getCurrentDate();
}

public class ReservationTest extends TestCase {
   public void testActive() {
       Reservation reservation = new Reservation();
       reservation.setClock(new Clock() {
          public Date getCurrentDate() {
              return new GregorianCalendar(2004, 0, 22).getTime();
          }
       });
       reservation.setReserveDate(
          new GregorianCalendar(2004, 0, 20).getTime());
       reservation.setDaysReserved(3);
       assertTrue(reservation.isActive());
   }
   public void testInactive() {
       Reservation reservation = new Reservation();
       reservation.setClock(new Clock() {
          public Date getCurrentDate() {
              return new GregorianCalendar(2004, 0, 22).getTime();
          }
       });
       reservation.setReserveDate(
          new GregorianCalendar(2004, 0, 20).getTime());
       reservation.setDaysReserved(2);
       assertFalse(reservation.isActive());
   }
}

public class Reservation {
```

```
    private Clock clock;
    private Date reserveDate;
    private int daysReserved;
    public boolean isActive() {
        GregorianCalendar lastEffectiveDate = new GregorianCalendar();
        lastEffectiveDate.setTime(reserveDate);
        lastEffectiveDate.add(Calendar.DAY_OF_MONTH, daysReserved - 1);
        return !clock.getCurrentDate().after(lastEffectiveDate.getTime());
    }
    public void setDaysReserved(int daysReserved) {
        this.daysReserved = daysReserved;
    }
    public void setReserveDate(Date reserveDate) {
        this.reserveDate = reserveDate;
    }
    public void setClock(Clock clock) {
        this.clock = clock;
    }
}

public class ClockInJava implements Clock {
    public Date getCurrentDate() {
        return new Date();
    }
}
public class Reservation {
    public Reservation() {
        setClock(new ClockInJava());
    }
    ...
}
```

第 14 章 结对编程

两个人怎么一起编程

假定小王，我们的一个开发人员，要开始维护一个团队开发完成的系统。他加入这团队已经有一段时间了，所以他对这个系统还是相当熟悉的。不过，他现在好像被难住了。小许，一个比较有经验的开发人员，注意到小王正坐在电脑前一脸茫然。

小许："Hi，小王，你在干嘛？"

小王："我在处理这里面的一个 DataAccesser 类。不过不懂怎么下手。"

小许对这个系统不怎么熟悉。他就问："什么是 DataAccesser？我看看。"同时拿起一块椅子坐在小王旁边。

注：小王跟小许现在就在同一台电脑上编程。这就叫"结对编程"了。

小王在 IDE 里面打开 DataAccesser 这个类。一堆的代码映入小许的眼里。小许："这是干嘛的？"

小王："它用来访问数据库的。"

这个解释太模糊了。小许得看一下这个类的属性跟方法。他说："我看看它的属性。"

当我们在给一个人解释一些已有的代码或者设计时，最好先让他看一下代码，然后边解释，不看代码解释起来很费解（这点地球人应该都知道了）。

小王指着 courseData 这个属性：

```
public abstract class DataAccesser {
   private CourseData courseData;
   private Table table;
   ...
}
```

说，"CourseData 这东西很怪异。它几乎包含系统里的所有数据。看起来真烦。你看看。"然后又打开 CourseData 类：

```
public class CourseData {
   ...
   private Students students;
   private Courses courses;
   private Enrollments enrollments;
   private IdCounter idCounter;
   ...
}
```

然后他指着 students 属性：

```
public class CourseData {
   ...
   private Students students;
```

```
    private Courses courses;
    private Enrollments enrollments;
    private IdCounter idCounter;
    ...
}
```

解释道："比如，Students 继承了 DataAccesser，它代表数据库里面的所有学生。"同时他又打开 Students 类让小许瞄一下：

```
public class Students extends DataAccesser {
    ...
}
```

返回到 CourseData 类，然后指着 courses 属性：

```
public class CourseData {
    ...
    private Students students;
    private Courses courses;
    private Enrollments enrollments;
    private IdCounter idCounter;
    ...
}
```

说："类似的，Courses 代表数据库里的所有课程。Enrollments 代表所有的选修信息等等。"

小许："噢，我明白了，回到 DataAccesser 类吧。"

小王又打开 DataAccesser 类。小许指着 table 这个属性：

```
public abstract class DataAccesser {
    private CourseData courseData;
    private Table table;
    ...
}
```

问道："这是什么？"

小王："Table 类很简单，它里面只有表名跟字段名。"

小许："好。"现在小许知道 DataAccesser 里面有个 Table 的引用，然后还有个 courseData 可以引用系统里面所有其他的 DataAccessers。小许继续问道："它有什么方法？"

小王往下拉，找一些重要的方法。它指着 deleteAll 方法：

```
public void deleteAll() throws DataAccessException {
```

```
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
             ((DataAccesser) refs.elementAt(i)).deleteAll();
    PreparedStatement st;
    try {
        st =
            getConnection().prepareStatement("DELETE FROM " + table.getName());
        try {
            executeUpdate(st);
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

然后说："比如，deleteAll 这个方法可以删除这个表里面所有的记录。"

调用 getRefsAccessersForDeleteAll 方法的这行代码引起了小许的注意。他指着这行：

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
        for (int i = 0; i < refs.size(); i++)
             ((DataAccesser) refs.elementAt(i)).deleteAll();
    ...
}
```

问："这是干什么的？"
小王："它取出系统里面所有引用了当前 DataAccesser 的其他 DataAccesser 对象。"小王指着循环：

```
public void deleteAll() throws DataAccessException {
    Vector refs = getRefsAccessersForDeleteAll();
    if (refs != null)
    for (int i = 0; i < refs.size(); i++)
          ((DataAccesser) refs.elementAt(i)).deleteAll();
    ...
```

解释道："这里面先调用其他 DataAccesser 的 deleteAll 方法。"

小许不怎么明白小王在说什么。所以他又问道："你能不能举个例子？"

注：如果我们不懂对方在说什么，最好的办法就是让他举一个例子。这是沟通（也是结对编程）中最重要的方法。

小王："好，比如，如果你现在想删除所有的学生，因为可能有一些选修记录里面引用着这些学生，所以应该先删除那些选修记录。"

现在小许推断出，DataAccesser 对象代表数据库里的一张表。这张表很灵巧，可以执行级联的删除。"好，我明白了。我们再看下一个方法吧。"小许说。

小王："好。"然后他继续往下拉，又找到一个重要的方法。"你看。"他指着"update"方法：

```
public int update(Object[][] fieldsAndValues, Object[][] keyFieldsAndValues)
   throws DataAccessException {
   try {
       PreparedStatement st =
           getConnection().prepareStatement(
              "UPDATE "
                  + table.getName()
                  + " SET "
                  + getParameterString(fieldsAndValues)
                  + getConditionString(keyFieldsAndValues, "="));
       try {
           try {
              setParameters(st, fieldsAndValues, 1);
              setParameters(
                  st,
                  keyFieldsAndValues,
                  fieldsAndValues.length + 1);
           } catch (InvalidArgumentException e1) {
              e1.printStackTrace();
           }
           return executeUpdate(st);
       } finally {
           st.close();
       }
   } catch (SQLException e) {
       throw new DataAccessException(e);
   }
}
```

小王解释道："这方法更新这个表里面的部分数据。"他指着 fieldsAndValues 和 keyFieldsAndValues 这两个参数：

```
public int update(Object[][] fieldsAndValues, Object[][] keyFieldsAndValues)
    throws DataAccessException {
    ...
}
```

说："这两个参数我真是极其讨厌。你看看它们是怎么用的。"然后他用 IDE 提供的功能，找出一个调用了这个方法的地方。"你看。"

```
private void updateStudentAnyway(Student student) throws DataAccessException {
    Object[][] fieldsAndValues = { {
            StudentsTable.eFirstNameField, student.getEFirstName()}, {
            StudentsTable.cFirstNameField, student.getCFirstName()
        }, {
            StudentsTable.eLastNameField, student.getELastName()
        }, {
            StudentsTable.cLastNameField, student.getCLastName()
        }, {
            StudentsTable.idTypeField, student.getIdType()
        }, {
            StudentsTable.idNoField, student.getIdNo()
        }, {
            StudentsTable.nationalityField, student.getNationality()
        }, {
            StudentsTable.genderField, new Boolean(student.isMale())
        }, {
            StudentsTable.birthDateField, student.getBirthDate()
        }, {
            StudentsTable.regionField, student.getRegionId()
        }, {
            StudentsTable.eAddressField, student.getEAddress()
        }, {
            StudentsTable.cAddressField, student.getCAddress()
        }, {
            StudentsTable.telField, student.getTel()
        }, {
            StudentsTable.mobileField, student.getMobile()
        }, {
            StudentsTable.faxField, student.getFax()
        }, {
            StudentsTable.emailField, student.getEmail()
        }
    };
```

```
    Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,
student.getStudentNo()}
    };
    update(fieldsAndValues, keyFieldsAndValues);
}
```

小王:“看，这方法调用update方法，来更新学生信息。”然后他指着:

```
private void updateStudentAnyway(Student student) throws DataAccessException {
  Object[][] fieldsAndValues = { {
          StudentsTable.eFirstNameField, student.getEFirstName()}, {
          StudentsTable.cFirstNameField, student.getCFirstName()
      },
    ...
}
```

说:“这是一个2维数组，前者表示要被设的字段名，后面是该字段的值。”然后他指着:

```
private void updateStudentAnyway(Student student) throws DataAccessException {
    Object[][] fieldsAndValues = { {
        ...
    };
    Object[][] keyFieldsAndValues = { { StudentsTable.studentNoField,student.getStudentNo()}
    };
    update(fieldsAndValues, keyFieldsAndValues);
}
```

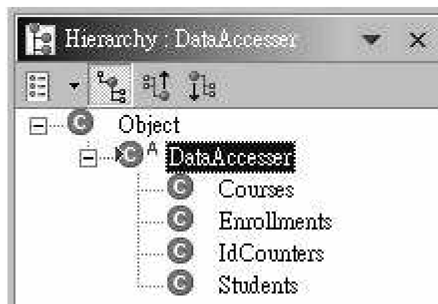
说:“这也是一个2维数组。它代表一个SQL条件。这个2维数组太烂了。”

小许也同意，这个数组实在太差了。然后他考虑用一个列表代替，这个列表里面的每个对象都是由“field-value”这样的一对值组成组成的。“好，可以了，”他认为他看的代码足够了，“所以，你现在打算改一下DataAccesser这个类？”

小王:“对啊，我觉得这个类太丑了。”他在等小许的建议。

小许想了一会儿，说。“好，我先看一下继承DataAccesser的最简单的类是什么样的。”

小王:“好。”他右击DataAccesser，然后选择“Open Type Hierarchy”，IDE列出所有所有DataAccesser的子类:



这个功能让小许惊讶了一下下。他把这个功能记在心里。

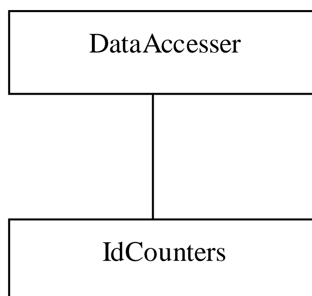
注：一些知识就这样传递过去了。有经验的开发人员，有时候还是可以从年青的开发人员上学得一些东西的。

小王指着“IdCounters”类说“这应该是最简单的”。然后他打开这个类：

小许浏览了一下 IdCounters 类的代码，发现它只用了 DataAccesser 很少的一些方法。“好，我们开始吧，我们建一个 DBTable 类吧。”

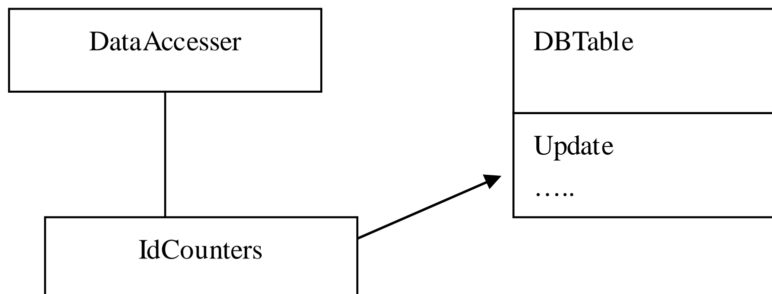
小王不明白小许要做什么。“好，不过你要做什么？”

小许：“现在 IdCounters 是继承了 DataAccesser。是吧？”小许在纸上画了一下草图：

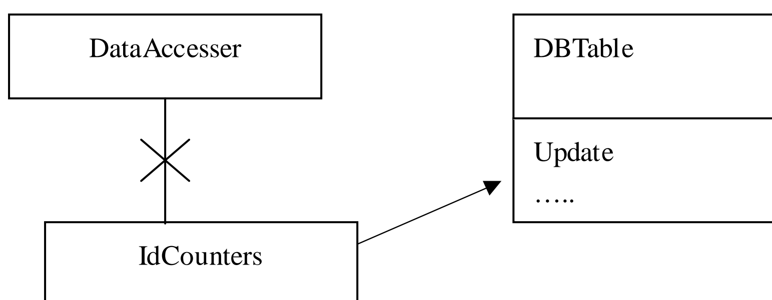


注：小许正在向小王展示他脑里想的设计。通常，交流设计最好的方法就是边给他看代码边解释。不过，现在还没有代码，所以图表是最好的形式。注意到上面的图表有点像 UML，但又不是完全 UML 的格式（比如，UML 里面继承是要用一个空箭头表示）。但这样就可以了，因为继承的关系在他们的对话里面就说得很清楚了。

他继续道：“我建 DBTable 这个类，提供 DataAccesser 的功能，然后让 IdCounters 引用一个 DBTable 对象，而不是继承 DataAccesser 类。”他继承画：



“现在我不想动到 DataAccesser。当 IdCounters 不再使用 DataAccesser 后，我们就把继承去掉。”然后他画了个叉叉：



注：这里又违反了 UML 的格式。在 UML 里面并没有这种叉叉来代表取消继承。不过对一般人来讲这样已经很明显了。

小王现在有点儿兴奋了，因为他发现他可以一点一点的将对 DataAccesser 的依赖去掉。“好！来吧”。

注：小王跟小许刚才一起设计。在结对编程里面，一起设计是主要活动之一。他们也一起重构。小王提供他对这个系统里面类的了解，然后建议说 DataAccesser 这个类太丑了，想把它换掉（重构什么）。小许提供了重构的对策，让调用端代码逐渐使用 DBTable 来一点一点将对 DataAccesser 的依赖去掉（怎么重构）。

在结对编程里，有着不同的知识/技能的人可以把他们的技能共享来解决一个困难的问题。比如小王对系统的了解比较多，但是 OO 设计和重构的能力比较弱，小许 OO 的设计和重构能力比较强，不过他对系统不了解。显然，让他们中任一人单独重构这个系统都很困难。但如果是一起干活的话，知识被合用，重构变得轻松多了。上面的情节只是有效合作的一种。还有其他的合作：数据库管理员跟程序员的合作（优化访问数据库的代码），UI 设计跟程序员的合作（开发 UI），程序员之间的合作（一个人调用另一个人写的代码），高级程序员和设计师/架构师的合作（将设计实现，看看设计行不行得通）等等。

在结对编程里面，经常可以交流知识跟好的实践经验。现在小王对重构有更多的了解，而小许对现有系统了解更深。

在结对编程里面，程序员会比较开心，做事也比较有信心。比如，小王开始很担心。不过跟小许结对后，他舒展多了。事实上，在小王将小许的想法实践成功以后，小许也会有自信。

小许：“好，我们来建 DBTable 这个类吧。”

小王有点迷惑了。他问：“我们不是应该先写一个通不过的测试吗？”

小许：“啊，对！我们先写测试吧。”

所以小王建了个 DBTableTest 类：

```
public class DBTableTest extends TestCase {
}
```

注：小王跟小许要写代码测试 DBTable 了。对，除了一起设计以外，一起测试也是结对编程的另一个主要活动。

小王开始写一个方法，来测试 deleteAll 方法。他打出“public”，然后写了一个方法以“test”开头：

```
public class DBTableTest extends TestCase {
   public test
}
```

小许指向这个位置：

```
public class DBTableTest extends TestCase {
   public test
}
```

提醒道：“你漏了个 void 了。”（可能读者看到这里面有觉得有点好笑，不过你们可以找一个伙伴试一下结对写代码，如果你出现一个这样低级的错误，然后你的伙伴提醒你，你会觉得这样的合作还挺很好玩，不过你也会感觉挺不好意思的。）

注：小许发现这个错误，然后立即指正它。对，除了一起设计，一起测试，一起查错也是结对编程的主要活动。

小王马上修正了错误：

```
public class DBTableTest extends TestCase {
   public void test
}
```

小王：“测什么？ testDeleteAll 怎么样？”

小许：“好。”小王就开始写代码了，并边写边说：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
   }
}
```

小王：“我们 new 一个 DBTable。”

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
   }
}
```

小王："然后我们可以调用 deleteAll。"

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.deleteAll();
   }
}
```

小王在写上面这些代码的时候，小许静静的在一边不说话。他在想怎么测 deleteAll，然后有了个主意。

小王："怎么样，怎么测？"

小许："我们先建一个接口，模拟数据库服务吧，它可以执行一个 SQL 语句。"

小王还没见过有人这样写单元测试的。因此他怎么不明白小许什么意思。他就直说了："不好意思，你在说什么？"

小许："要测 DBTable，我们可能要建一个数据库和一个测试用的表。这很麻烦。此外，运行这些测试还很费时间。因此，我们应该找个东西可以模拟替代数据库。"

小王还是不怎么明白，因为这样说太抽象了。他就说："不好意思，还是不明白。"

小许说："没事，你按照我说的编写，过一会儿就明白了。现在建一个名字为 DBServer 的接口。"小王按照他说的建了：

```
public interface DBServer {
}
```

然后准备写第一个方法。

注：如果我们不能明白另一个人的设计意图，那最好的方法就是，写代码。谁来写？让比较不明白的那个人写。如果仍然是由小许来写代码的话，小王就会在一边呆呆的不知道做什么，或者直接睡着了。

小许说："public， int， executeUpdate， 括号。"小王照着打：

```
public interface DBServer {
   public int executeUpdate()
}
```

小许继承说，"String，SQL。好了。"小王打了：

```
public interface DBServer {
   public int executeUpdate(String sql);
}
```

小许说：“回到测试类。”小王又打开那个类：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.deleteAll();
   }
}
```

小许指着这行：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.deleteAll();
   }
}
```

说：“在前面加一行。table，点，setDBServer，括号。”小王照着写：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.setDBServer()
       table.deleteAll();
   }
}
```

小许继承道，“ new， DBS，自动完成，括号。”小许的意思是说，用 IDE 的自动填写的功能让“DBS”变成“DBServer”。小王也知道自动完成的功能，所以他知道小许的意思。IDE 列出了所有以 DBS 开头的类，事实上有许多类都是以“DBS”开头的，比较 DBSanityChecker，DBSuperUser，但小王一下子就选中了 DBServer ：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.setDBServer(new DBServer())
       table.deleteAll();
   }
}
```

这意味着小王已经慢慢明白小许的意图了。

小许指着这个位置：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable();
       table.setDBServer(new DBServer())
```

```
            table.deleteAll();
        }
    }
```

说，“在括号里面，用自动完成。”小王不知道这是什么用图的，不过他还是照做了，系统自动为它生成了：

```
    public class DBTableTest extends TestCase {
        public void testDeleteAll() {
            DBTable table = new DBTable();
            table.setDBServer(new DBServer() {
                public int executeUpdate(String sql) {
                    return 0;
                }
            })
            table.deleteAll();
        }
    }
```

“哇哦！强悍！我之前都不知道。”小王惊讶的说。

注：知识在这边又交流了。学会了这个技巧以后，小王以后写代码的效率大大提高了。如果他以后跟其他同事合作，他也会把这个技巧教给他们。这样的话，这个技巧会让整个团队知道。

小许指着这行：

```
    public class DBTableTest extends TestCase {
        public void testDeleteAll() {
            DBTable table = new DBTable();
            table.setDBServer(new DBServer() {
                public int executeUpdate(String sql) {
                    return 0;
                }
            })
            table.deleteAll();
        }
    }
```

说：“在前面增加一行。assertEquals， sql， 双引号，delete，from。”

```
    public class DBTableTest extends TestCase {
        public void testDeleteAll() {
            DBTable table = new DBTable();
            table.setDBServer(new DBServer() {
                public int executeUpdate(String sql) {
                    assertEquals(sql, "delete from")
                    return 0;
                }
```

```
        })
        table.deleteAll();
    }
}
```

小许犹豫了一下，然后指着这个位置：

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable();
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

说："传一个'abc'的字符串给这构造函数。"小王照做了：

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

小许指着这个位置：

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from")
                return 0;
```

```
            }
        })
        table.deleteAll();
    }
}
```

说："from，空格，abc。"小王照着写了，现在他明白这个"abc"是表名：

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("abc");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from abc");
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

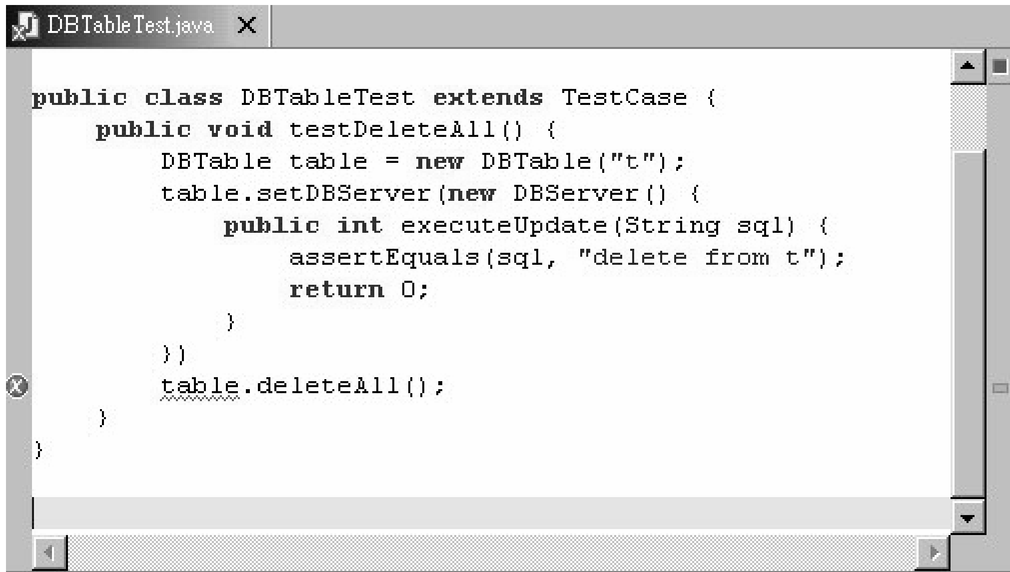
小王问："为什么不叫't'？它不是比'abc'清楚些吗？"小许也觉得't'会比'abc'清楚一些。所以他说，"没错，那用't'吧。"小王又改成"t"：

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        })
        table.deleteAll();
    }
}
```

注：按小许的意思写完所有代码后，小王终于明白怎么用 DBServer 模拟数据库来对 DBTable 进行单元测试。这个技巧就是用假对象测试。知识又传递了。

小王跟小许一起做了另一个设计，那就是表名要怎么取。但这回是小王提醒了小许。这说明，即使一个人在某一方面很擅长，另一个人还是有空间做一些有意义的贡献的。

不过，这时 IDE 提示了一个语法错误：



小许还在检查错在哪里的时候，小王已经发现错在哪了："喔，这边少了个分号!"所以他更正：

```
public class DBTableTest extends TestCase {
   public void testDeleteAll() {
       DBTable table = new DBTable("t");
       table.setDBServer(new DBServer() {
           public int executeUpdate(String sql) {
              assertEquals(sql, "delete from t");
              return 0;
           }
       });
       table.deleteAll();
   }
}
```

注：这次小王比较早发现了问题。有经验不代表每次都能更快找到错误。每个人都可能擅长找某一方面的问题，而不擅于其他方面的。幸运的是，两个人的"盲区"通常不会重叠。比如，小王在建 testDeleteAll 这个方法的时候没有意识到少了个"void"修饰符，不过小许却很容易就发现了。相反，小许没有注意到这里少了个分号，而这个错误对小王来说很显眼。我们这里面讲的是语法的错误，但逻辑的错误道理也是一样的，两人搭档的话，很快就可以发现 bug 出在哪里，否则，找这些逻辑错误就要花更长的时间了。

然后 IDE 又指出了其他的语法错误：

厦门:王伟杰(Wingel)　MSN: wingel_wang@hotmail.com
邮箱: seewingel@gmail.com　博客: http://blog.csdn.net/Wingel

DBTableTest.java

```
public class DBTableTest extends TestCase {
    public void testDeleteAll() {
        DBTable table = new DBTable("t");
        table.setDBServer(new DBServer() {
            public int executeUpdate(String sql) {
                assertEquals(sql, "delete from t");
                return 0;
            }
        });
        table.deleteAll();
    }
}
```

为了解决这个问题，小王跟小许定义了 DBTable 的构造函数，setDBServer 和 deleteAll 这两个方法：

```
public class DBTable {
   private String tableName;
   private DBServer dbServer;

   public DBTable(String tableName) {
       this.tableName = tableName;
   }
   public void setDBServer(DBServer dbServer) {
       this.dbServer = dbServer;
   }
   public void deleteAll() {
   }
}
```

他们运行了测试，然后测试竟然通过了！

注：因为意料外的事情发生了，他们又要一起 debug 了。

小许之前也碰过这种错误，所以他很快就知道问题出在哪里，他指着这行：

```
public void testDeleteAll() {
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
       public int executeUpdate(String sql) {
           assertEquals(sql, "delete from t");
           return 0;
       }
```

```
    });
    table.deleteAll();
}
```

说："在这行前面加行代码，final，StringBuffer，callLog，equal，new，StringBuffer"，而小王照着写了：

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}
```

然后小许指着这行：

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}
```

说："在前面加行代码。callLog，点，append，双引号，x，双引号"，然后小王又照着写了：

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
}
```

小许指着下面这行：

```
public void testDeleteAll() {
   final StringBuffer callLog = new StringBuffer();
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
       public int executeUpdate(String sql) {
           callLog.append("x");
           assertEquals(sql, "delete from t");
           return 0;
       }
   });
   table.deleteAll();
}
```

想告诉小王在前面加代码，不过这回小王已经知道他要做什么了，然后自己就加了：

```
public void testDeleteAll() {
   final StringBuffer callLog = new StringBuffer();
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
       public int executeUpdate(String sql) {
           callLog.append("x");
           assertEquals(sql, "delete from t");
           return 0;
       }
   });
   table.deleteAll();
   assertEquals(callLog.toString(), "x");
}
```

注：在写完这些代码以后，小王现在知道怎么用一个调用日记来记录调用的顺序。知识又传递了。

他们又运行了测试，然后失败了。

注：现在他们要实现 DBTable 类里面的 deleteAll 方法了。对，除了一起设计，一起测试以外，一起写代码也是结对编程的主要活动。

小王说："好，我们把 DataAccesser 的代码拷过来吧。"

```
public void deleteAll() throws DataAccessException {
   Vector refs = getRefsAccessersForDeleteAll();
   if (refs != null)
       for (int i = 0; i < refs.size(); i++)
```

```
            ((DataAccesser) refs.elementAt(i)).deleteAll();
    PreparedStatement st;
    try {
        st =
            getConnection().prepareStatement("DELETE FROM " + table.getName());
        try {
            executeUpdate(st);
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
}
```

小王把级联删除有关的代码删掉，然后用 DBServer 类来代替 JDBC 的 connection 和 preparedStatement：

```
public void deleteAll() {
    dbServer.executeUpdate("DELETE FROM " + tableName);
}
```

然后他们又跑了一下代码，还是通不过！小许指着这行：

```
public void testDeleteAll() {
    final StringBuffer callLog = new StringBuffer();
    DBTable table = new DBTable("t");
    table.setDBServer(new DBServer() {
        public int executeUpdate(String sql) {
            callLog.append("x");
            assertEquals(sql, "delete from t");
            return 0;
        }
    });
    table.deleteAll();
    assertEquals(callLog.toString(), "x");
}
```

说："在这边设个断点吧。"不过，小王突然叫了起来，"啊！这边的问题，这里面是用大写字母！不过测试里面是用小写字母的！"

```
public void deleteAll() {
    dbServer.executeUpdate("DELETE FROM " + tableName);
}
```

所以他们把测试改成：

```
public void testDeleteAll() {
   final StringBuffer callLog = new StringBuffer();
   DBTable table = new DBTable("t");
   table.setDBServer(new DBServer() {
       public int executeUpdate(String sql) {
           callLog.append("x");
           assertEquals(sql.toLowerCase(), "delete from t");
           return 0;
       }
   });
   table.deleteAll();
   assertEquals(callLog.toString(), "x");
}
```

然后跑了一下测试，测试通过了。现在任务做完了，然后他们把已有的测试都跑了一遍。

在这期间，他们休息了一下，去喝一些水（也是一起喝的，因为他们现在是较好的搭档了。）。

注：因为结对编程两人的精力都会很集中，精神容易紧张，所以经常的休息开发效率才会更高。

在这次结对编程以后，小王跟小许对系统的这部分代码更熟悉了（DBTable，DBServer 等等）。如果他们中有人要去休假或者离开公司的话，另一个也可以维护。也就是说，结对编程可以减少职工离职对公司的损失。

接着，另一个同事小强走过来问道：“小王，我要开始做这个复本打印的用户例事。你之前好像有写过输入标志的代码，能不能跟我搭档一下？”

小王：“当然。”现在小王开始跟小强配合了，然后小许可以跟其他人搭档了。

注：经常的换搭档是好事（特别是在一个新任务的开始阶段）。现在小王要继续把他懂的一些东西教给小强。同时他也可以从小强身上学到一些（比如，复本打印是怎么实现的）。

小王知道标志要怎么保存，他可以帮小强实现取出标记来打印复本的功能。这也是小强找小王配合的原因。

结对编程的好处：

联合两人的知识去对付一个难题。
知识互相传递。
更有效的查错跟纠错。
程序员都很开心。
减少员工离职的损失。

结对编程需要的一些技能：

用代码解释已有的设计结构。
用例子来解释。
用图表来解释设计思路。
如果你无法把你的设计思路表达清楚，把代码写出来。
让比较迷惑的搭档来写代码，这样他就可以较好的融入你的概念。
经常的休息。
经常的更换搭档。

需要把程序员的数量加倍吗？

即使结对编程有许多的好处，但如果我们让两个程序员做一件事的话，那开发速度不是会减半？我们要不要把程序员数量加倍？

Laurie Williams 有做一个研究表明，在一个大学的环境里面，让两个人做一件事情，花费的时间比两个人分工所需的时间多 15％。也就是说，我们没必要加倍，我们只需要增加 15％的程序员，或者让程序员多工作 15％的时间就行了。

所以，一方面来讲结对编程有很大的好处，但同时它也要多花费 15％的开发时间。这样做值吗？Laurie Williams 的研究还表明，结对编程开发出来的软件，bug 的数量比分工开发出来的软件少不止 15％。软件有 bug 是肯定要修复的。所以软件不只要考虑开发周期，还要考虑维护的时间。从产业的统计数据来看，Alistair Cockburn 和 Laurie Williams 得出结论说，单人修复软件 bug 所花的时间，是结对的人所花时间的 15-60 倍。注意，不是多 15％到 60％，而是 15 到 60 倍！很明显，结对编程远远就抵消了那多费的 15％的开发时间。

什么情况结对编程行不通

结对编程不是万灵的。因为它需要两个人不断的沟通，一起做决定，如果不能沟通或者做不了决定的话，结对编程就行不通了。

什么情况不能沟通？比如，小许为了赶工期，弄得非常紧张，但他的开发经理老包坚持让他去配合小王。这样小许就很不情愿的去跟小王搭档。他让小王把代码给他看看。尽管他觉得 DataAccesser 这个类很不合适，但他一心在想着他的项目截止日期。他说道："哦！去掉这个类太花时间了。你简单把它改名成 DBTable 吧。"小王说："不行！现在整个系统的结构全部集中在数据库上，再这样下去系统会崩掉的。我们一定要解决这个结构！老包让你过来帮我的……"之后，小王在写代码的时候，小许根本没认真看。他一点建议都没给。

上面这种情况，就是不能沟通的，因为小许不愿意跟小王结对编程。他们表面上是搭档的，事实上根本没有。他们没有共同的目标（比如，改进系统的结构）。如果人没有共同的目标，那他们就没有沟通的欲望。

这不是无法沟通的唯一原因。比如，如果一个搭档比较情绪化，不喜欢跟人交流，那就无法沟通了。当小王和小许在写 testDeleteAll 这个方法的时候，小王建议用"t"来代替"abc"作表名。如果小许这样回答："abc 就行了，我写了好几年的测试，都是用这种数据的。你懂什么？你才刚毕业好不好？我在编程的时候，你还在幼

儿园呢。”

在交流的方式上，沟通可能也会有问题。如果一个搭档缺乏信心，他什么意见都不敢提。小王跟小许在一起写 DBTable 的时候，正确的情况下，小王问小许为什么要在写 DBTable 类之前写一个失败的测试。可是如果小王被小许的经验和能力震住了，一个问题也不敢提？那系统开发过程中，就得不到小王任何有意义的建议了。

缺乏自信只是不提意见的一个原因。另一个原因看起来就很傻了。当小王还给小许解释 DataAccesser 类里面的 deleteAll 方法中有关级联删除的代码时，如果小许觉得不懂这些代码是一件很蠢的事情，然后就不细看，也不提问题直接跳过去，那会怎么样？小许对系统了解就不够多，也就给不了重构的什么好建议了。

总之，这下面有一些觉见的问题，会造成结对编程无法正常工作：

不情愿的配合。

拒绝别人的意见，甚至攻击对方。

小心翼翼有意见不敢提。

怕别人觉得自己笨不敢问问题。

如果这些问题真的发生了呢？不过这个跟管理以及个人性格更有关系，最好的方式就是不让他们结对，或者让他们跟别的人结对。 （看到这边，你们可以尝试找个人跟你 Pair 了。）