代码设计

Outline

- 设计易读的代码
- 设计易维护的代码
- 设计可靠的代码
- 使用模型辅助设计复杂代码
- 为代码开发单元测试用例
- 代码复杂度度量
- 问题代码

18.1 设计易读的代码

- 维护的需要
- 团队协作的需要

代码规范

- 格式
- 命名
- 注释
- ...

布局格式

- 使用缩进与对齐表达逻辑结构

```
//缩进方式一：
if (……)
{
        return true;
 }
 else
 {
        return false;
 }
//缩进方式二：
if (……) {
        return true;
 } else {
        return false;
 }
```

```
public class Sales extends DomainObject{
    ......
    public double getTotal(){
        Iterator iter = salesLineItemMap.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Object val = entry.getValue();
            total+= ((SalesLineItem)val).getSubTotal();
        }
        salesPO.setTotal(total);
        return total;
    }

    public void addSalesLineItem(long commodityID, long quantity){
        SalesLineItem item = new SalesLineItem(commodityID,quantity);
        salesLineItemMap.put(commodityID, item);
    }
}
```

缩进

缩进

对齐

对齐

布局格式

- 使用缩进与对齐表达逻辑结构
- 将相关逻辑组织在一起

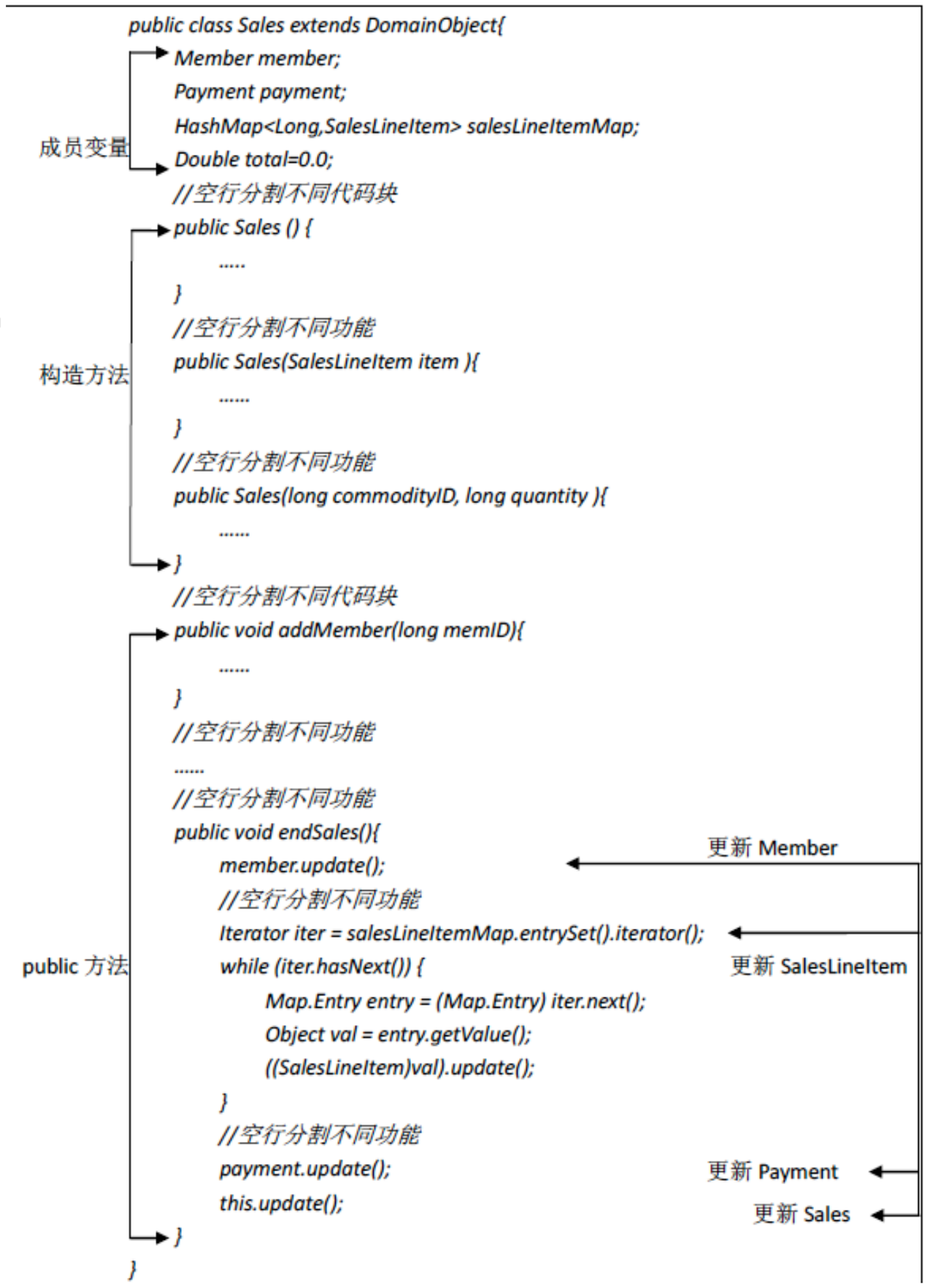
类定义的逻辑组织

- 成员变量声明；
- 构造方法与析构方法；
- public 方法；
- protected 方法；
- private 方法。

```
public void endSales(){
        Iterator iter = salesLineItemMap.entrySet().iterator();
        member.update();
        this.update();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Object val = entry.getValue();
            ((SalesLineItem)val).update();
        }
        payment.update();
}
```

相关逻辑组织混乱的代码

逻辑组织清晰的代码


public class Sales extends DomainObject{
Member member;
Payment payment;
HashMap<Long,SalesLineItem> salesLineItemMap;
Double total=0.0;
成员变量
//空行分割不同代码块
public Sales () {
.....
}
//空行分割不同功能
public Sales(SalesLineItem item){
.....
}
//空行分割不同功能
public Sales(long commodityID, long quantity){
.....
}
构造方法
//空行分割不同代码块
public void addMember(long memID){
.....
}
//空行分割不同功能
.....
//空行分割不同功能
public void endSales(){
member.update();
//空行分割不同功能
Iterator iter = salesLineItemMap.entrySet().iterator();
while (iter.hasNext()) {
Map.Entry entry = (Map.Entry) iter.next();
Object val = entry.getValue();
((SalesLineItem)val).update();
}
//空行分割不同功能
payment.update();
this.update();
}
public 方法
}
更新 Member
更新 SalesLineItem
更新 Payment
更新 Sales

布局格式

- 使用缩进与对齐表达逻辑结构
- 将相关逻辑组织在一起
- 使用空行分割逻辑
- 语句分行

```
switch (type) {
        case 1:
                ....
                break;
        //空行分割
        case 2:
                ....
                break;
        //空行分割
        default:
                ....
                break;
}
```

空行分割复杂控制结构

```
return this==obj
            || (this.obj instanceof Myclass
                && this.field == obj.field) ;
```

长句断行

命名

- 使用有意义的名称进行命名。例如对“销售信息”类，命名为Sales 而不是ClassA。
    - 使用名词命名类、属性和数据；
    - 使用名词或者形容词命名接口；
    - 使用动词或者“动词+名词”命名函数和方法；
    - 使用合适的命名将函数和方法定义的明显、清晰，包括返回值、参数、异常等。
- 名称要与实际内容相符。例如，使用Payment 命名“账单”类明显比使用“Change”更相符，因为“账单”类的职责不仅仅是计算“Change”，还要维护账单数据。
- 如果存在惯例，命名时要遵守惯例。例如，Java 语言的命名惯例是：使用小写词命名包；类、接口名称中每个单词的首字母大写；变量、方法名称的第一个单词小写，后续单词的首字母大写；常量的每个单词大写，单词间使用“_”连接。

- 临时变量命名要符合常规。像for 循环计数器、键盘输入字符等临时变量一般不要求使用有意义的名称，但是要使用符合常规的名称，例如使用i、j 命名整数而不是字符，使用c、s 命名字符而不是整数。
- 不要使用太长的名称，不利于拼写和记忆。
- 不要使用易混字符进行命名，常见的易混字符例如“l”（大写i）、“l”（数字1）与“l”（小写L）、0（数字零）与O（字母）等。使用易混字符的命名例如D0Calc 与DOCalc。
- 不要仅仅使用不易区分的多个名称，例如Sales与Sale，SalesLineItem与SalesLineitem。
- 不要使用没有任何逻辑的字母缩写进行命名，例如wrttn、wtht、vwls、smch、trsr……

注释

- 注释类型(Java)
  - 语句注释（//）
  - 标准注释(/* */)
  - 文档注释(/** */)

文档注释的内容

- 包的总结和概述，每个包都要有概述；
- 类和接口的描述，每个类和接口都要有概述；
- 类方法的描述，每个方法都要有功能概述，都要定义完整的接口描述；
- 字段的描述，重要字段含义、用法与约束的描述。

Javadoc

- 为了方便使用注释文档化程序代码，人们还为Java程序提供了Javadoc工具。只要程序员注释程序时使用特定的标签，Javadoc就能从代码中抽取出注释形成一个HTML格式的代码文档。

- 在描述类与接口时，Javadoc常用的标签是：
  - @author：作者名
  - @version：版本号
  - @see：引用
  - @since：最早使用该方法/类/接口的JDK版本
  - @deprecated 引起不推荐使用的警告

Javadoc

- 在描述方法时，Javadoc常用的标签是：
    - @param 参数及其意义
    - @return 返回值
    - @throws 异常类及抛出条件
    - @see：引用
    - @since：最早使用该方法/类/接口的JDK版本
    - @deprecated 引起不推荐使用的警告

```java
/**
 * LoginController的职责是将登录界面（LoginDialog）发来的请求
 *  转发给后台逻辑（User）处理
 * LoginController接收界面传递的用户ID和密码
 *  经User验证后，返回登录成功true或者失败false
 * @author xxx.
 * @version 1.0
 * @see presentation.LoginDialog
 */
public class LoginController {

    /**
     * 验证登录是否有效.
     *
     * @param id long型，界面传递来的用户标识
     * @param password   String型，界面传递来的用户密码
     * @return 成功返回true，失败返回false
     * @throws DBException   数据连接失败
     * @see businesslogic.domain.User
     */
    public boolean login(long id, String password) throw DBException{
        User user;
        user = new User(id);
        return user.login(password);

    }
}
```

内部注释

- 注释要有意义，不要简单重复代码的含义
- 重视对数据类型的注释
- 重视对复杂控制结构的注释

```
public class Sales extends DomainObject{
    //为了快速存取，使用HashMap组织销售商品项列表 ←——注释数据类型
    //Key是商品ID，取值范围是1...MAXID
     HashMap<Long,SalesLineItem> salesLineItemMap;
    ……

    public void endSales(){
        //更新Member信息 ←——没有意义的注释
        member.update();

        //更新salesLineItem信息
        Iterator iter = salesLineItemMap.entrySet().iterator();
        while (iter.hasNext()) {//逐一遍历销售商品项 ← 注释控制结构
            Map.Entry entry = (Map.Entry) iter.next();
            Object val = entry.getValue();
            ((SalesLineItem)val).update();
        }

        payment.update();

        this.update();
    }
}
```

18.2 设计易维护的代码

- 1 小型任务
  - 要让程序代码可修改，就要控制代码的复杂度。这首先要求每个函数或方法的代码应该是内聚的，恰好完成一个功能与目标。
  - 如果内聚的代码本身比较简单，复杂性可控，那么它就具有比较好的可维护性。反之，内聚的代码也可以比较复杂，典型表现是完成一个功能需要多个步骤、代码比较长，那么就需要将其进一步分解为多个高内聚、低耦合的小型任务。

设计易维护的代码

- 1 小型任务
- 2 复杂决策
  - 使用新的布尔变量简化复杂决策

```
If   ( (atEndofStream) &&(error!= inputError ) ) &&
     ( ( MIN_LINES<=lineCount) && lineCount<= MAX_LINES) ) &&
     ( ! errorProcessing(error) ) {
     ……
}
```

图 18-10 复杂决策示例

```
boolean allDataReaded=    ( (atEndofStream) &&(error!= inputError ) );
boolean validLineCount =( MIN_LINES<=lineCount)&& lineCount<= MAX_LINES);

If ( allDataReaded && validLineCount && ( ! errorProcessing(error) ) ) {
     ……
}
```

- 2 复杂决策
  - 使用新的布尔变量简化复杂决策
  - 使用有意义的名称封装复杂决策
    - 对于决策“ If((id>0) && (id<=MAX_ID))”，可以封装为“If (isIdValid(id))”，方法isIdValid(id)的内容为 “return ((id>0) && (id<=MAX_ID))”。
  - 表驱动编程

```
/*各个不同级别的赠送事件可以同时触发，例如新会员一次性购
 *买产生了6000 积分，就同时触发1 级、2 级与3 级三个事件
*/

//prePoint 是增加之前的积分额度；
//postPoint 是增加之后的积分额度；

//如果首次积分超过1000，触发1 级礼品赠送事件
If ( (prePoint <1000) && (postPoint>=1000) ) {
      triggerGiftEvent (1);
}

//如果首次积分超过2000，触发2 级礼品赠送事件
If ( (prePoint <2000) && (postPoint>=2000) ) {
      triggerGiftEvent (2);
}

//如果首次积分超过5000，触发3 级礼品赠送事件
If ( (prePoint <5000) && (postPoint>=5000) ) {
      triggerGiftEvent (3);
}
```

prePoint(小于)	postPoint（大于等于）	Event Level
1000	1000	1
2000	2000	2
5000	5000	3

```
prePointArray = { 1000,    2000,    5000 };
postPointArray = { 1000,    2000,    5000 };
levelArray = { 1,    2,    3 };
for (int i=0;i<=2; i++) {
        if ( (prePoint< prePointArray[i]) && (postPoint>= postPointArray[i])) {
                triggerGiftEvent (levelArray[i]);
        }
}
```

设计易维护的代码

- 1 小型任务
- 2 复杂决策
- 3 数据使用

数据使用

- （1）不要将变量应用于与命名不相符的目的。例如使用变量total表示销售的总价，而不是临时客串for循环的计数器。
- （2）不要将单个变量用于多个目的。在代码的前半部分使用total表示销售总价，在代码后半部分不再需要“销售总价”信息时再用total客串for循环的计数器也是不允许的。
- （3）限制全局变量的使用，如果不得不使用全局变量，就明确注释全局变量的声明和使用处。
- （4）不要使用突兀的数字与字符，例如15（天）、“MALE”等，要将它们定义为常量或变量后使用。

设计易维护的代码

- 1 小型任务
- 2 复杂决策
- 3 数据使用
- 4 明确依赖关系
  - 类之间模糊的依赖关系会影响到代码的理解与修改，非常容易导致修改时产生未预期的连锁反应。

18.3 设计可靠的代码

- 契约式设计
  - 异常方式
  - 断言方式
- 防御式编程

```
public class Sales extends DomainObject{
    ……
    public double getChange(double payment) throws PreException,
                                                    PostException {
        //前置条件检查
        If ( payment<=0) || (payment <total) {
            throw new PreException("Sales.getChange: Payment"+
                                    String.valueOf(payment)+
                                    "; Total "+String.valueOf(total));
        }
        ……
        //返回result之前进行后置条件检查
        If (result!= (payment-total) ) {
            throw new PostException("Sales.getChange: Payment"+
                                    String.valueOf(payment)+
                                    "; Total "+String.valueOf(total));
        }
        return result;
    }
}
```

异常方式

```
public class Sales extends DomainObject{
    ……
    public double getChange(double payment) throws AssertionError {
        //前置条件检查
        assert ( ( payment>0) && (payment >= total))    :
                ("Sales.getChange: Payment"+String.valueOf(payment)+
                        "; Total "+String.valueOf(total));
        ……
        //返回result之前进行后置条件检查
        assert    (result== (payment-total) )    :
                ("Sales.getChange: Payment"+String.valueOf(payment)+
                        "; Total "+String.valueOf(total));
        return result;
    }
}
```

断言方式

Java中断言语句的实现

- 为了方便实现契约式设计，Java 提供了断言语句：“assert Expression1(: Expression2) ;”：
    - Expression1 是一个布尔表达式，在契约式设计中可以将其设置为前置条件或者后置条件；
    - Expression2 是一个值，各种常见类型都可以；
    - 如果Expression1 为true，断言不影响程序执行；
    - 如果Expression1 为false，断言抛出AssertionError 异常，如果存在Expression2 就使用它作为参数构造AssertionError。

防御式编程常见场景

- 防御式编程的基本思想是：在一个方法与其他方法、操作系统、硬件等外界环境交互时，不能确保外界都是正确的，所以要在外界发生错误时，保护方法内部不受损害。
- 常见场景
    - 输入参数是否合法?
    - 用户输入是否有效?
    - 外部文件是否存在?
    - 对其他对象的引用是否为NULL?
    - 其他对象是否已初始化?
    - 其他对象的某个方法是否已执行?
    - 其他对象的返回值是否正确?
    - 数据库系统连接是否正常?
    - 网络连接是否正常?
    - 网络接收的信息是否有效?
- 异常和断言都可以用来实现防御式编程，两种实现方式的差异与契约式设计的实现一样。

18.4 使用模型辅助设计复杂代码

- 决策表
- 伪代码
- 程序流程图

条件和行动	规则
条件声明（Condition Statement）	条件选项（Condition Entry）
行动声明（Action Statement）	行动选项（Action Entry）

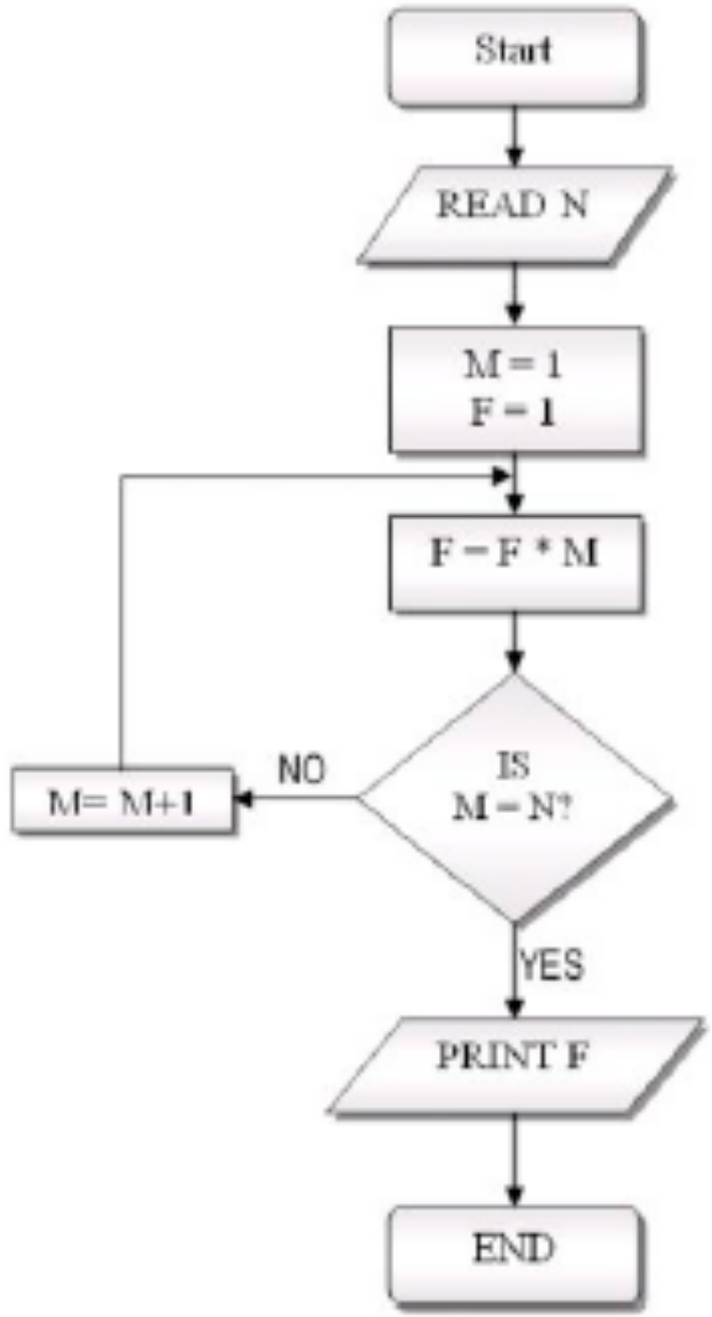
决策表的基本结构

条件和行动	规则		
prePoint postPoint	<1000 >=1000	<2000 >=2000	<5000 >=5000
Gift Event Level 1	X		
Gift Event Level 2		X	
Gift Event Level 3			X

决策表示例

```
得到 SalesLineItem 的迭代器：
While 迭代器 hasNext () { //逐一遍历，遍历中：
        找到 SalesLineItem 对象；
        按照更新步骤进行更新：得到相应的 Mapper；
                                将自己的信息转变为层间传递的 PO 对象；
                                将 PO 对象交给 Mapper；
                                Mapper 完成更新；
}
```

伪代码



程序流程图

18.5 为代码开发单元测试用例

- 为方法开发测试用例主要使用两种线索：
  - （1）方法的规格；
    - 根据第一种线索，可以使用基于规格的测试技术开发测试用例，等价类划分和边界值分析是开发单元测试用例常用的黑盒测试方法。
  - （2）方法代码的逻辑结构。
    - 根据第二种线索，可以使用基于代码的测试技术开发测试用例，对关键、复杂的代码使用路径覆盖，对复杂代码使用分支覆盖，简单情况使用语句覆盖。

```
public class Sales extends DomainObject{
    ……
    List<SalesLineItem> salesList = new List<SalesLineItem>();
    public void addSalesLineItem(SalesLineItem item){
        salesList.add(item);
    }
    public double total(){
        Double total=0.0;
        Iterator iter = salesList.iterator();
        while (iter.hasNext()) {
            Object val = iter.next();
            total+= ((SalesLineItem)val).subTotal();
        }
        return total;
    }
}
```

Sales.total()方法代码

```
public class MockSalesLineItem extends SalesLineItem{
        double price;
        double quantity;
        ……
        public MockSalesLineItem(double p, int q ){
                price=p;
                quantity=q;
        }
        Public double subTotal () {
                return price*quantity;
        }
}
```

Mock Object

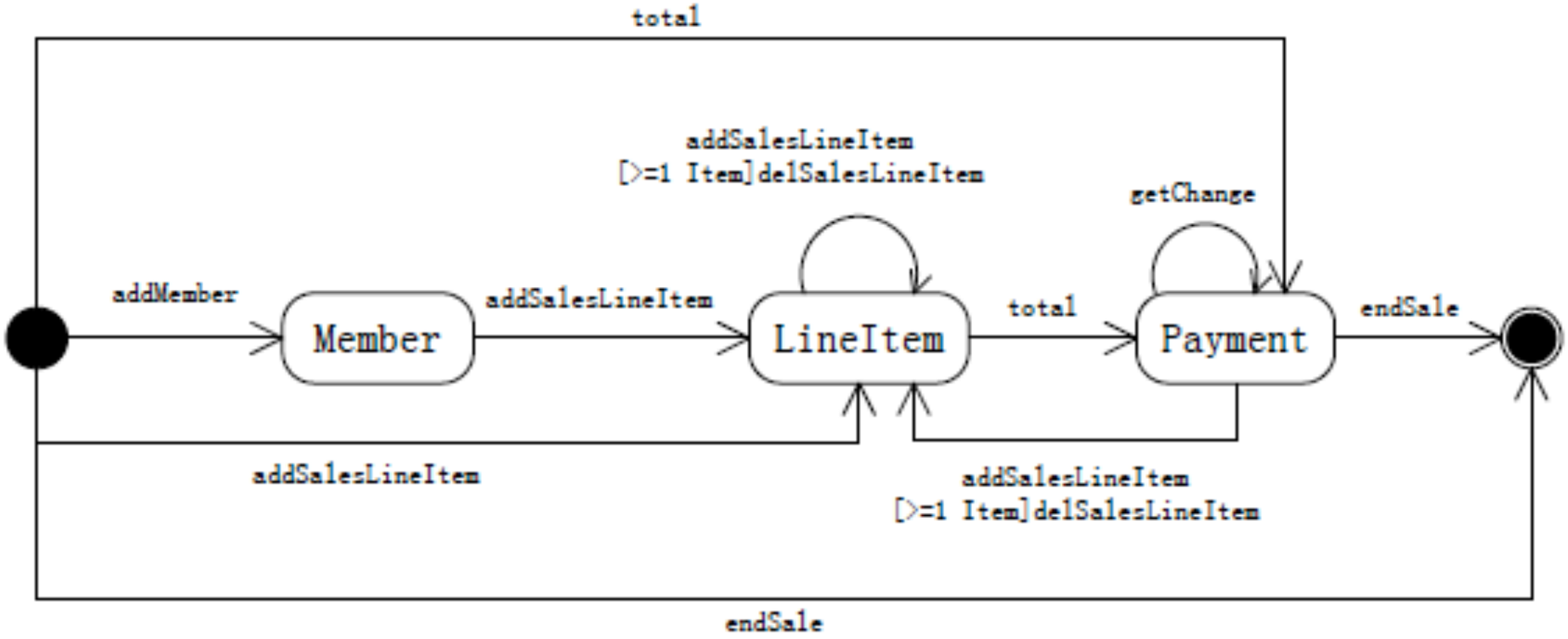
```
public class TotalTester {
        @Test
        public void testTotal () {
            MockSalesLineItem mockSalesLineItem1
                                = new MockSalesLineItem (50, 2);
            MockSalesLineItem mockSalesLineItem2;
                                = new MockSalesLineItem (40, 3);
            Sales sale=new Sales();
            sale.addSalesLineItem(mockSalesLineItem1) ;
            sale.addSalesLineItem(mockSalesLineItem2) ;

            assertEquals (220, sale.total () );
        }
}
```

JUnit测试代码

为类开发测试用例

- 在复杂类中，常常有着多变的状态，每次一个方法的执行改变了类状态时，都会给其他方法带来影响，也就是说复杂类的多个方法间是互相依赖的。
- 所以，除了测试类的每一个方法之外，还要测试类不同方法之间的互相影响情况。


total
addSalesLineItem
[>=1 Item]delSalesLineItem
getChange
addMember
Member
addSalesLineItem
LineItem
total
Payment
endSale
addSalesLineItem
addSalesLineItem
[>=1 Item]delSalesLineItem
endSale

Sales类的状态图

输入		预期输出状态
方法	当前状态	
addMember	Start	Member
	Member	非法
	LineItem	非法
	Payment	非法
	End	非法
addSalesLineItem	Start	LineItem
	Member	LineItem
	LineItem	LineItem
	Payment	LineItem
	End	非法
[>=1 Item] delSalesLineItem	……	……
total	……	……
getChange	……	……
endSale	……	……

测试用例线索

ID	输入		预期输出
	前置语句	方法	
1	s=new Sales();	s.addMember(2);	No Exception
2	s=new Sales(); s.addMember(1);		MemberLable Invalid Time
3	s=new Sales(); s.addSalesLineItem(1);		

4	s=new Sales();
	s.addSalesLineItem(1);
	s.total();		

5	s=new Sales();
	s.addSalesLineItem(1);
	s.total();
	s.getChange(100);
	s.endSale();
			Sales dose not Exists

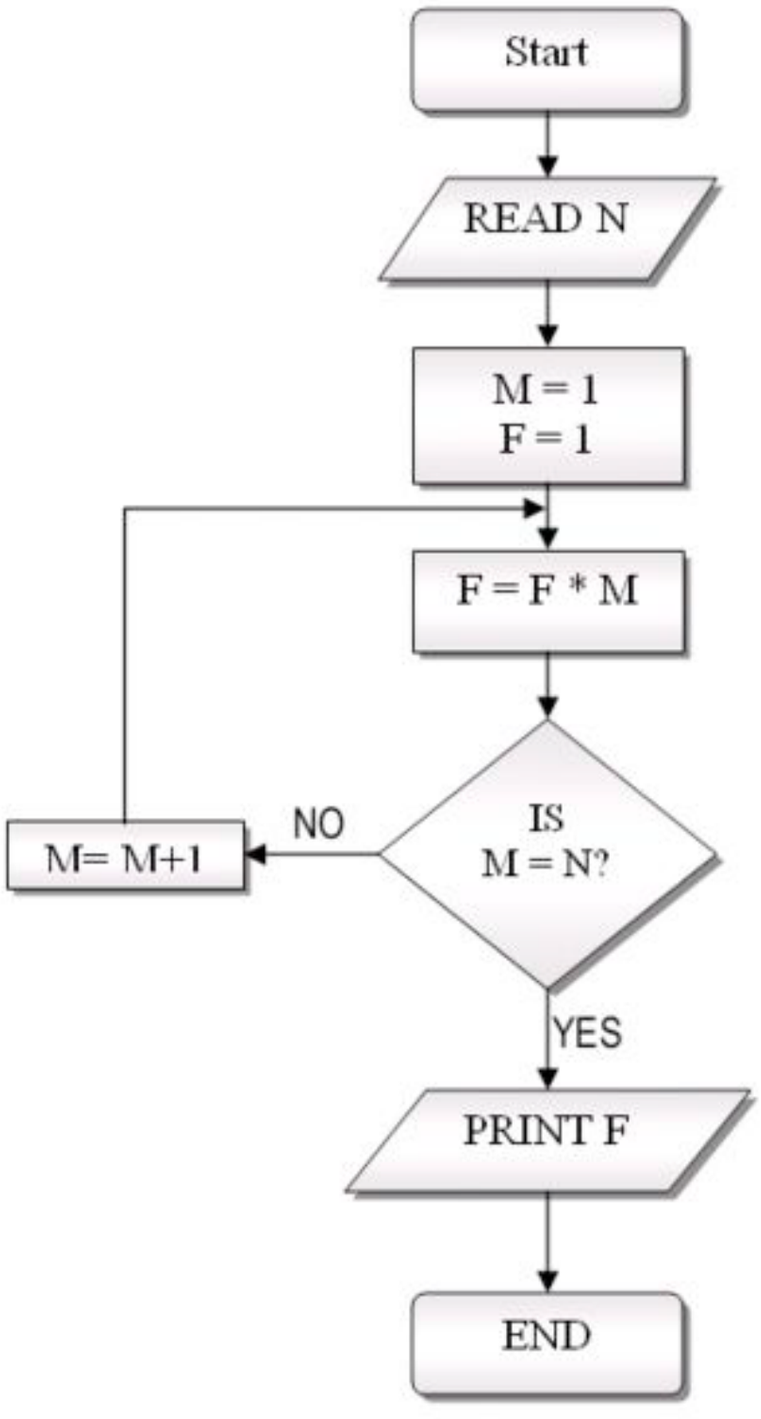
.....			

测试用例

18.6 代码复杂度度量

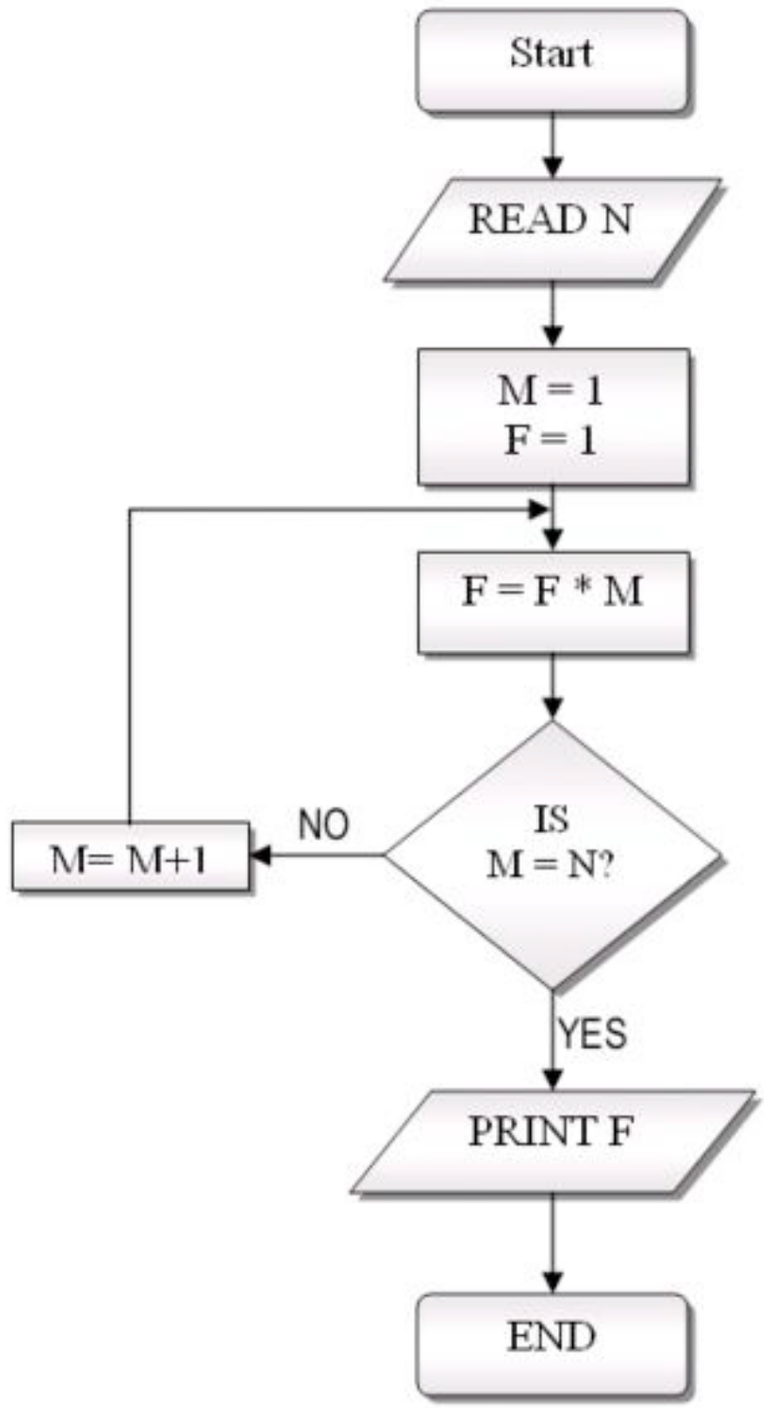
- 程序复杂度是造成各种编程困难的主要原因。[Dijkstra1972]很早就指出：“有能力的程序员会充分认识到自己的大脑容量是多么地有限；所以，他会非常谦卑的处理编程任务。”
- 为了帮助程序员处理程序复杂度，人们提出了很多程序复杂度的度量手段，其中McCabe的圈复杂度[McCabe1976]得到了比较大的关注。

- 衡量圈复杂度的基本思路是计算程序中独立路径的最大数量。第一种计算方法是建立程序的流程图G，假设图的节点数为N，边数为E，那么复杂度V（G）=E-N+2。以右图为例，其节点数为8，边数为8，则程序的复杂度为 2。通过直接分析右图也可以发现，它的确有两条路径。



- 还有一种简单的算法是直接计数程序中决策点的数量：(I) 从I开始，一直往下通过程序。 (2) 一旦遇到下列关键字，或者同类的词，就加I： if, while, repeat, for。 (3) 给case语句中的每一种情况都加I。

- 例如，右图所描述的程序很明显只有一个DO-While语句，所以复杂度为2。



度量的意义

- 基于圈复杂度，你可以衡量一下程序代码是否需要调整。[McConnell2004]认为：
- 0-5 子程序可能还不错；
- 6-10 得想办法简化子程序了；
- 10+ 把子程序的某一个部分拆分成另一个子程序并调用它。 10个决策点的上限并不是绝对的。应该把决策点的数量当做一个警示，该警示说明某个子程序可能需要重新设计了。
- [Chidamber1994]基于所拥有方法的代码复杂度定义了类的复杂度：
    - 类的加权方法= Sum（Ci） i = from 1 to n
- 其中，n为一个类的方法数量，Ci是第i个方法的代码复杂度。

代码大全

变量

- 变量定义
- 变量初始化
- 作用域
- 持续性

变量定义

- 关闭隐式声明
- 声明全部的变量
- 遵循某种命名规则
- 检查变量名

- acctNo or acctNum

变量初始化

- 在声明变量的时候初始化
- 在靠近变量第一次使用的位置初始化
- 理想情况下，在靠近第一次使用变量的位置声明和定义该变量
- 在可能的情况下使用final或者const
- 特别注意计数器和累加器
- 在类的构造函数里初始化该类的数据成员
- 检查是否需要重新初始化
- 一次性初始化具名常量；用可执行代码来初始化变量
- 使用编译器设置来自动化初始化所有变量
- 利用编译器的警告信息
- 检查输入参数的合法性
- 用内存访问检查工具来检查错误的指针
- 在程序开始时初始化工作内存

作用域

- 使变量应用局部化（空间）
  - 变量跨度尽可能小
- 尽可能缩短变量的存活时间（时间）
  - 变量生存时间尽可能小

减小作用域的一般原则

- 在循环开始之前再去初始化该循环里使用的变量，而不是在该循环所属子程序的开始处初始化这些变量
- 直到变量即将被使用时再赋值
- 把相关语句放到一起
- 把相关语句组提取成单独的子程序
- 开始时采用最严格的可见性，然后根据需要扩展变量的作用域

持续性

- 在程序中加入调试代码或者断言来检查那些关键变量的合理取值
- 准备抛弃变量时给它们赋上“不合理的值”
- 编写代码时要假设数据并没有持续性
- 养成在使用所有数据之前声明和初始化的习惯

持续性的多种形态

- 特定代码端或子程序的生命期
    - for循环里声明的变量
- 只要你允许，它就会持续下去
    - new
- 程序的声明期
    - static
- 永远持续
    - 存储在文件

为变量制定单一用途

- //compute roots of a quadratic equation
- //this code assumes that (b*b-4*a*c) is positive
- Temp=Sqrt(b*b-4*a*c);
- Root[0]=(-b+temp)/(2*a);
- Root[0]=(-b-temp)/(2*a);

- //swap the roots
- Temp = root[0];
- Root[0] = root[1];
- Root[1]=temp;

两个变量用于两种用途

- //compute roots of a quadratic equation
- //this code assumes that (b*b-4*a*c) is positive
- discriminant=Sqrt(b*b-4*a*c);
- Root[0]=(-b+discriminant)/(2*a);
- Root[0]=(-b-discriminant)/(2*a);

- //swap the roots
- oldRoot= root[0];
- Root[0] = root[1];
- Root[1]=oldRoot;

避免让代码具有隐含意义

- Bad example
  - pageCount的取值可能表示已打印纸张的数量，除非它等于-1，在这种情况下表明有错误发生

变量的命名

- X =x-xx;
- Xxx= fido+SalesTax(fido);
- X = x+ LateFee(x1, x) + xxx;
- X = x + Interest(x1,x);

```
Balance          = Balance - LastPayment;
MonthlyTotal     = NewPurchases + SalesTax( NewPurchases );
Balance          = Balance + LateFee( CustomerID, Balance ) + MonthlyTotal;
Balance          = Balance + Interest( CustomerID,  Balance );
```

变量代表的实体	恰当的名称	不恰当的名称
火车速度	Velocity、TrainVelocity、VelocityInMPH	VELT，V，TV，X，X1
今天日期	CurrentDate、CrntDate	CD，Current，C，X
每页行数	LinesPerPage	LPP，Lines，X，X1

数值理论

- 避免使用“神秘数值”
- 如果需要，可以使用硬编码的0和1
  - 0表示起始值
  - 1表示增量
- 预防除0的错误
- 使类型转换变得明显
- 避免混合类型的比较
- 注意编译器的警告

整数

- 检查整数除法
- 检查整数溢出
- 检查中间结果溢出

浮点数

- 避免数量级相差巨大的数之间的加减运算
- 避免等量判断
  - 10个0.1相加等于1?
- 处理舍入误差
- 检查语言和函数库对特定数据类型的支持

创建子程序的正当理由

- 降低复杂度
- 引入中间、易懂的抽象
- 避免代码重复
- 支持子类化
- 隐藏顺序
- 隐藏指针操作
- 提高可移植性
- 简化复杂的布尔判断
- 改善性能
- 确保所有的子程序都很小

好的子程序名字

- 描述子程序所做的所有事情
  - computReportTotals
  - computReportTotalsAndOpenOutputFile
- 避免使用无意义的、模糊或者表述不清的动词
  - HandleCalculation, PerformService,ProcessInput
- 不要同多数字来形成不同的子程序名字
  - Part1,part2
- 根据需要确定子程序名字的长度
  - 9-15为佳
- 给函数命名时要对返回值有所描述
  - isReady, currentColor
- 给过程起名时使用语气强烈的动词加宾语的形式
  - printDocument
- 准确使用对仗词
  - Add/remove
- 为常用操作确立命名规则

算法的设计

- 我们一个重要的关注点是实现的性能或效率。直觉上你要使得代码运行得尽可能快。但是，这隐含了一些代价。
    - 编写更快代码的代价。可能会使代码更加复杂，从而花费更多的时间编写。
    - 测试代码的代价。代码的复杂度要求更多的测试用例或测试数据。
    - 用户理解代码的时间代价。
    - 需要修改代码时，修改代码的时间代价。

在执行时间与设计质量、标准、和客户需求之间平衡考虑

- 如果速度对实现来说很重要的化，你就必须了解你所使用的编译器是如何优化代码的。否则，优化反而会让看起来更快的代码实际变得更慢。例如，你要编写一个三维数组的代码。为了提高效率，你决定用一维数组来实现。然后自己计算索引：
  - index=3*i+2*j+k;
- 但是，编译器可能在寄存器里面计算数组索引，那么执行的时间很少。如果编译器在寄存器中使用加法递增技术，而不是每次位置计算都使用加法和乘法，那么一维数组技术可能会导致实际上执行时间的增加

一般控制问题

- 布尔表达式
- 复合语句
- 空语句
- 驯服危险的深层嵌套
- 程序的复杂度

布尔表达式

- 用true和false做布尔判断
  - 不要用0和1
  - 隐式地比较布尔值与true和false
- 简化复杂的表达式
  - 拆分复杂的判断并引入新的布尔变量
  - 把复杂的表达式做成布尔函数
  - 用决策表代替复杂的条件
- 编写肯定形式的布尔表达式
  - If(!statusOK) or if(statusOk)
  - 用DeMorgan定律简化否定的布尔判断
    - If(!displayOK||!printerOK)-》if(!(displayOK&&printerOK))
- 用括号使布尔表达式更清晰
- 短路求值

布尔表达式

- 按照数轴的顺序编写数值表达式
- 与0比较
    - 隐式地比较逻辑变量
        - While(!done)
    - 把数与0相比较
        - While(balance!=0)
    - 在C中显示地比较字符和零终止符（’\0’）
        - While(*charPtr !=‘\0’)…
    - 把指针与NULL相比较
        - While（bufferPtr != NULL)

复合语句

- 把括号对一起写出
- 用括号被条件表达清楚

空语句

- While(recordArray.Read(index++)!=recordArray.EmptyRecord());
- ->可以通过加{}等来强调空语句
- 或者为创建一个DoNothing()预处理宏或者内联函数

更加清晰的非空循环体

- RecordType record = recordArray.Read(index);
- Index++;
- While(record!= recordArray.EmptyRecord()){
  - record = recordArray.Read(index);
  - index++;
- }

驯服危险的深层嵌套

- 通过重复检测条件中的某一部分来简化嵌套的if语句
- 用break块来简化嵌套if
- 把嵌套if转换成一组if-then-else语句
- 把嵌套if转换成case语句
- 把深层嵌套的代码抽取出来放进单独的子程序
- 使用面向对象的方法

[Green1997] How To Write Unmaintainable Code

- (1) 在注释中“说谎”。甚至于你并不需要编谎，只要不让代码和注释保持同步就可以。
- (2) 到处都使用/*add 1 to i*/这样的注释，从不注释包、类或者方法的整体意图。
- (3) 让每个方法都比它的名字多做点事。比如isValid(x)还将x装换为二进制存储在数据库中。

- (4) 以简洁的名义，大量使用首字母缩写。声称“好汉”是天生就能理解各种缩写词的。
- (5) 以效率的名义，避免使用封装。调用者可以知道被调用方法的内部实现。
- (6) 如果你在写一个飞机订票系统，当要增加一条航线的时候，确保至少要修改25个地方。而且不要记录这25个地方在哪里，让那个维护你代码的家伙通读你的每一行代码去吧。

- (7) 以效率的名义，使用复制/粘贴/克隆（clone）/修改等手段，毕竟这比复用很多小的模块要快的多。
- (8) 从来不对变量注释。要将关于变量用法、边界、有效值、精度、单位、显示格式、输入规则之类的信息散落到整个程序代码中。如果老板强制要求你写注释，就写满重复方法正文的注释，但绝不注释变量，连临时变量都不！
- (9) 在一行中写尽可能多的代码，名义上是为了使代码行数最少。不要忘了顺便把所有操作符周围的空白全部删除，并尽量让代码达到编辑器限制的255个字符长度。

- (10) 在使用缩写词命名方法与变量时，为了避免无聊，为一个单词定义多种不同的缩写，可以考虑在拼写上做点文章，最好把多个名字拼写地看不出差异。千万不要给出单词的全部字符，因为这不仅只能有一种写法，而且太容易被维护的程序员理解了。
- (11) 不要使用任何代码格式整理工具，不要将代码自动对齐。这样，你就可以“无意间”错误对齐控制结构，以产生误解了。例如，你可以将代码写成这样：
- if(a)
- if(b)x = y;
- else x = z;

- (12) 除非有强制要求，绝不使用 { } 界定if/else的代码块。如果你有一个嵌套很深的if/else结构，再加上对齐的误导，你都能骗倒一个专家级维护程序员了。
- (13)使用多个很长的变量名或者类名，而且它们的名字之间只有一个字母不同甚至只是大小写不一样。就像swimmer 与swimner、HashTable 与Hashtable。可以利用常见的字体显示问题，使用il1l或者 oO08这样难以分辨的字符。
- (14) 只要生命周期范围许可，就重用那些无关的变量。可以将同一个临时变量用于两种完全无关的用途。例如，在一个长方法的顶部给变量赋一个值，然后在中间的某个地方巧妙地改变变量的含义，例如将从0开始的数组坐标改为从1开始的数组坐标。需要确认的是不要记录这些改变。

- (15) 永远不要使用i作为循环的计数变量，哪怕使用c、s都可以。i就用来表示字符串吧。
- (16) 永远不要使用局部变量，需要临时使用数据的时候，就让其成为成员变量或者静态变量，而且要非常无私地与类的其他方法共享。
- (17) 为了防止无聊，从同义词词典中找出那些近义词，例如display、show、present，用它们命名相同的行为。这样，不同命名的行为粗看上去似乎很不相同，但其实完全一样。反过来，对于那些区别很大的行为，你可以使用相同的名字，例如使用print同时指代写文件、打印机打印和屏幕显示。在任何情况下，都不要定义能够消除项目词汇歧义的词汇表，要声称这是违反信息隐藏法则的不专业行为。

- (18) 给方法命名时，经常使用抽象意义的单词。比如rountineX48, PerformDataFunction, DoIt, HandleStuff和do_args_method等。
- (19) 不要注释是否修改了“引用”传递来的参数。如果方法修改了“引用”传递来的参数，那么就将这个方法命名为看上去只是查询的样子。
- (20) 从来不处理异常，名义上是因为好的代码不会失败，所以异常不会出现。
- (21) 如果数组有100个元素，就在代码中到处使用硬编码“100”，而不是使用常量或者变量来指代100。为了给修改增加难度，在需要使用100/2的地方直接使用50，在需要使用100-1的地方直接使用99，如此之类。

- (22) 在代码中到处都保留那些已经不再使用、过期的变量或者方法。要对外声称，谁知道什么时候就需要改回来呢？自己可不想在改回来的时候重新写一次这些代码。如果你再能在这些代码上留下令人一头雾水的注释，就可以确保没有哪个维护程序员敢动这些代码了。
- (23) 把所有的成员方法和成员变量都声明为public。这在增加将来修改难度的同时，还可以在大量的public方法中混淆类的真正职责。如果老板责备你太不小心了，你就告诉他你在按照接口透明的原则编程。

练习 - 表驱动

- For the first $10,000 of income, the tax is 10%
- For the next $10,000 of income above $10,000, the tax is 12 percent
- For the next $10,000 of income above $20,000, the tax is 15 percent
- For the next $10,000 of income above $30,000, the tax is 18 percent
- For any income above $40,000, the tax is 20 percent

- tax = 0.
- if (taxable_income == 0) goto EXIT;
- if (taxable_income > 10000) tax = tax + 1000;
- else{ tax = tax + .10*taxable_income;
- goto EXIT;
- }
- if (taxable_income > 20000) tax = tax + 1200;
- else{ tax = tax + .12*(taxable_income-10000):
- goto EXIT;
- }
- if (taxable_income > 30000) tax = tax + 1500;
- else{ tax = tax + .15*(taxable_income-20000);
- goto EXIT;
- }
- if (taxable_income < 40000){
- tax = tax + .18*(taxable_income-30000);
- goto EXIT;
- }
- else
- tax = tax + 1800. + .20*(taxable_income-40000);
- EXIT;

- Define a tax table for each “bracket” of tax liability

Bracket	Base	Percent
0	0	10
10,000	1000	12
20,000	2200	15
30,000	3700	18
40,000	55000	20

- Simplified algorithm

```
for (int i=2; level=1; i <= 5; i++)
        if (taxable_icome > bracket[i])
                level = level + 1;
tax= base[level]+percent[level] * (taxable_income - bracket[level]);
```