南京大学
NANJING UNIVERSITY

《软件工程与计算II》

代码设计

陈鑫

南京大学计算机科学与技术系

六、编 码

程序设计语言的性能和程序的编码风格，在很大程度上影响着软件的质量和维护性能。

编码部分要讨论以下几个问题：

- 程序设计语言性能的讨论
- 程序编码风格

编码风格

- 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

1、程序设计语言性能的讨论

- 软件心理学观点

(1) 一致性：表示语言使用符号的兼容程度、约束条件及语法和语义上的例外等等。

(2) 歧义性：歧义性导致程序员对程序理解的混乱。

(3) 简洁性：衡量用该语言编程的程序员必须记忆的信息量。

(4) 局部性和线性：人们的记忆和辨别能力分为联想和顺序两个方面。联想能使我们整体地记住和辨别某件东西。顺序记忆能从回忆序列中找出一个元素。局部性是语言的联想性；线性是语言的顺序性。

- 工程观点
  - (1) 使设计易于代码翻译;
  - (2) 编译程序的功效;
  - (3) 源代码的可移植性;
  - (4) 开发工具的可利用性;
  - (5) 源代码的可维护性。
- 技术性能观点
  - (1) 复杂数据结构
  - (2) 实时系统
  - (3) 特殊应用领域

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;
 }
```

缩进与对齐格式示例

```
public class SalesList 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();
        }
        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

How To Write Unmaintainable Code

```
        if(a)
if(b)x = y;
        else x = z;
```

How To Write Unmaintainable Code

Rule 11: Never use an automated source code tidier to keep code aligned.

- Lobby to have them banned on the grounds they create false deltas in CVS
- every programmer should have his own indenting style held forever sacrosanct for any module he wrote. ……
- You are now free to accidentally misalign the code to give the optical illusion that bodies of loops and ifs are longer or shorter than they really are,
- that else clauses match a different if than they really do.

How To Write Unmaintainable Code

Rule 13: Never put in any { } surrounding your if/else blocks unless they are syntactically obligatory.

If you have a deeply nested mixture of if/else statements and blocks, especially with misleading indentation, you can trip up even an expert maintenance programmer.

布局格式

- 将相关逻辑组织在一起
  - 类的逻辑组织
    - 成员变量声明;
    - 构造方法与析构方法;
    - public 方法;
    - protected 方法;
    - private 方法。
- 使用空行分割逻辑

逻辑组织清晰的代码

成员变量

构造方法

```
public class Sales extends DomainObject{
    Member member;
    Payment payment;
    HashMap<Long,SalesLineItem> salesLineItemMap;
    Double total=0.0;
    //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 ){
        ......
    }
    //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();
```

更新 Member

布局格式

- 语句分行

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

How To Write Unmaintainable Code

Rule 9. Try to pack as much as possible into a single line.

This saves the overhead of temporary variables, and makes source files shorter by eliminating new line characters and white space.

- Tip: remove all white space around operators. Good programmers can often hit the 255 character line length limit imposed by some editors. The bonus of long lines is that programmers who cannot read 6 point type must scroll to view them.

How To Write Unmaintainable Code

Rule 31. Nest as deeply as you can.

- Good coders can get up to 10 levels of ( ) on a single line and 20 { } in a single method.
- C++ coders have the additional powerful option of preprocessor nesting totally independent of the nest structure of the underlying code.
- You earn extra Brownie points whenever the beginning and end of a block appear on separate pages in a printed listing. Wherever possible, convert nested ifs into nested [? :] ternaries.

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。

命名

- 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“I”（大写i）、“1”（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……

How To Write Unmaintainable Code

Rule 3. Make sure that every method does a little bit more (or less) than its name suggests.

As a simple example, a method named isValid(x) should as a side effect convert x to binary and store the result in a database.

Rule 4. Use acronyms to keep the code terse(简洁).

Real men never define acronyms; they understand them genetically.

How To Write Unmaintainable Code

Rule 10. Cd wrttn wtht vwls s mch trsr.

- When using abbreviations inside variable or method names, break the boredom with several variants for the same word, and even spell it out longhand once in while. This helps defeat those lazy bums who use text search to understand only some aspect of your program.
- Consider variant spellings as a variant on the ploy, e.g. mixing International colour, with American color and dude-speak kulerz. If you spell out names in full, there is only one possible way to spell each name. These are too easy for the maintenance programmer to remember.
- Because there are so many different ways to abbreviate a word, with abbreviations, you can have several different variables that all have the same apparent purpose. As an added bonus, the maintenance programmer might not even notice they are separate variables

How To Write Unmaintainable Code

Rule 15. Use very long variable names or class names that differ from each other by only one character, or only in upper/lower case.

- An ideal variable name pair is swimmer and swimner.
- Exploit the failure of most fonts to clearly discriminate between ill1| or oO08 with identifier pairs like parseInt and parseInt or D0Calc and DOCalc. l is an exceptionally fine choice for a variable name since it will, to the casual glance, masquerade as the constant 1.
- Create variable names that differ from each other only in case e.g. HashTable and Hashtable.

How To Write Unmaintainable Code

Rule 23: To break the boredom, use a thesaurus to look up as much alternate vocabulary as possible to refer to the same action, e.g. display, show, present.

Vaguely hint there is some subtle difference, where none exists.

However, if there are two similar functions that have a crucial difference, always use the same word in describing both functions (e.g. print to mean write to a file, and to a print on a laser, and to display on the screen).

Under no circumstances, succumb to demands to write a glossary with the special purpose project vocabulary unambiguously defined. Doing so would be unprofessional breach of the structured design principle of information hiding.

How To Write Unmaintainable Code

Rule 24. In naming functions, make heavy use of abstract words like it, everything, data, handle, stuff, do, routine, perform and the digits e.g.

routineX48, PerformDataFunction, DoIt, HandleStuff and do_args_method.

How To Write Unmaintainable Code

Rule 73. If you cannot find the right English word to convey the meaning of a temporary variable (and you ignore the other suggestions about not giving meaningful names to variables), you may use a foreign language word as the name of the variable.

For example, instead of using variable "p" for a "point", you may use "punkt", which is the German word for it. Maintenance coders without your firm grasp of German will enjoy the multicultural experience of deciphering the meaning. It breaks the tedium of an otherwise tiring and thankless job.

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

- 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();
    }
}
```

How To Write Unmaintainable Code

- Rule 1 Lie in the comments.

You don't have to actively lie, just fail to keep comments as up to date with the code.

- Rule 3. Pepper the code with comments like /* add 1 to i */

however, never document stuff like the overall purpose of the package or method.

How To Write Unmaintainable Code

Rule 8. Never put a comment on a variable.

- Facts about how the variable is used, its bounds, its legal values, its implied/displayed number of decimal points, its units of measure, its display format, its data entry rules, when its value can be trusted etc. should be gleaned from the procedural code.
- If your boss forces you to write comments, lard(填满) method bodies with them, but never comment a variable, not even a temporary!

主要内容

- 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

18.2 设计易维护的代码

- 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。
  - 封装该任务

How To Write Unmaintainable Code

- Rule 5. In the interests of efficiency, avoid encapsulation.
  - Callers of a method need all the external clues they can get to remind them how the method works inside.

- Rule 6. If, for example, you were writing an airline reservation system, make sure there are at least 25 places in the code that need to be modified if you were to add another airline.
  - Never document where they are. People who come after you have no business modifying your code without thoroughly understanding every line of it.

How To Write Unmaintainable Code

- Rule 7. In the name of efficiency, use cut/paste/clone/modify.
  - This works much faster than using many small reusable modules.
- Rule 46. Instead of using a parameters to a single method, create as many separate methods as you can.
  - For example instead of setAlignment(int alignment) where alignment is an enumerated constant, for left, right, center, create three methods: setLeftAlignment, setRightAlignment, and setCenterAlignment. Of course, for the full effect, you must clone the common logic to make it hard to keep in sync.

How To Write Unmaintainable Code

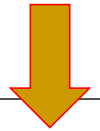
Rule 48： Declare every method and variable public.

- After all, somebody, sometime might want to use it.
- Once a method has been declared public, it can't very well be retracted, now can it?
- This makes it very difficult to later change the way anything works under the covers.
- It also has the delightful side effect of obscuring what a class is for.
- If the boss asks if you are out of your mind, tell him you are following the classic principles of transparent interfaces.

设计易维护的代码

- 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) ) {
     ……
}
```



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

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

设计易维护的代码

- 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);
}
```

表驱动编程

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

表驱动编程

```
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

数据使用

- 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。
- 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。
- 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。
- 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。

How To Write Unmaintainable Code

Rule 16. Wherever scope rules permit, reuse existing unrelated variable names. Similarly, use the same temporary variable for two unrelated purposes (purporting to save stack slots).

For a fiendish variant, morph the variable, for example, assign a value to a variable at the top of a very long method, and then somewhere in the middle, change the meaning of the variable in a subtle way, such as converting it from a 0-based coordinate to a 1-based coordinate. Be certain not to document this change in meaning.

How To Write Unmaintainable Code

- Rule19：Never use i for the innermost loop variable. Use anything but.
  - Use i liberally for any other purpose especially for non-int variables. Similarly use n as a loop index.

- Rule 20. Never use local variables.
  - Whenever you feel the temptation to use one, make it into an instance or static variable instead to unselfishly share it with all the other methods of the class.
  - This will save you work later when other methods need similar declarations. C++ programmers can go a step further by making all variables global.

How To Write Unmaintainable Code

Rule 29. If you have an array with 100 elements in it, hard code the literal 100 in as many places in the program as possible.

- Never use a static final named constant for the 100, or refer to it as myArray.length.
- To make changing this constant even more difficult, use the literal 50 instead of 100/2, or 99 instead of 100-1.
- You can futher disguise the 100 by checking for a == 101 instead of a > 100 or a > 99 instead of a >= 100.

设计易维护的代码

- 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。
  - 对于这些模糊的依赖关系，需要进行明确的注释（Javadoc中的@see）

How To Write Unmaintainable Code

Rule 24: Never document whether a method actually modifies the fields in each of the passed parameters. Name your methods to suggest they only look at the fields when they actually change them.

- In Java, all primitives passed as parameters are effectively read-only because they are passed by value. The callee can modify the parameters, but that has no effect on the caller's variables.
- In contrast all objects passed are read-write. The reference is passed by value, which means the object itself is effectively passed by reference. The callee can do whatever it wants to the fields in your object.

主要内容

- 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

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

契约式设计的异常实现

```
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;
    }
}
```

契约式设计的断言实现

```
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;
    }
}
```

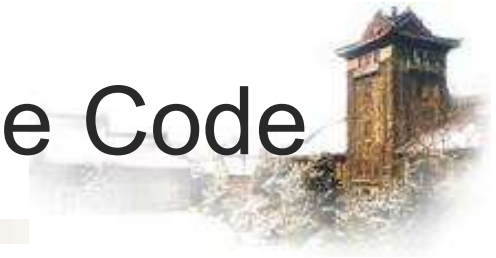
Java的断言语句

- 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：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。

How To Write Unmaintainable Code



Rule 27. Exceptions are a pain in the behind.

- Properly-written code never fails, so exceptions are actually unnecessary. Don't waste time on them. Subclassing exceptions is for incompetents who know their code will fail.
- You can greatly simplify your program by having only a single try/catch in the entire application (in main) that calls System.exit().
- Just stick a perfectly standard set of throws on every method header whether they could throw any exceptions or not.

主要内容

- 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

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

决策表

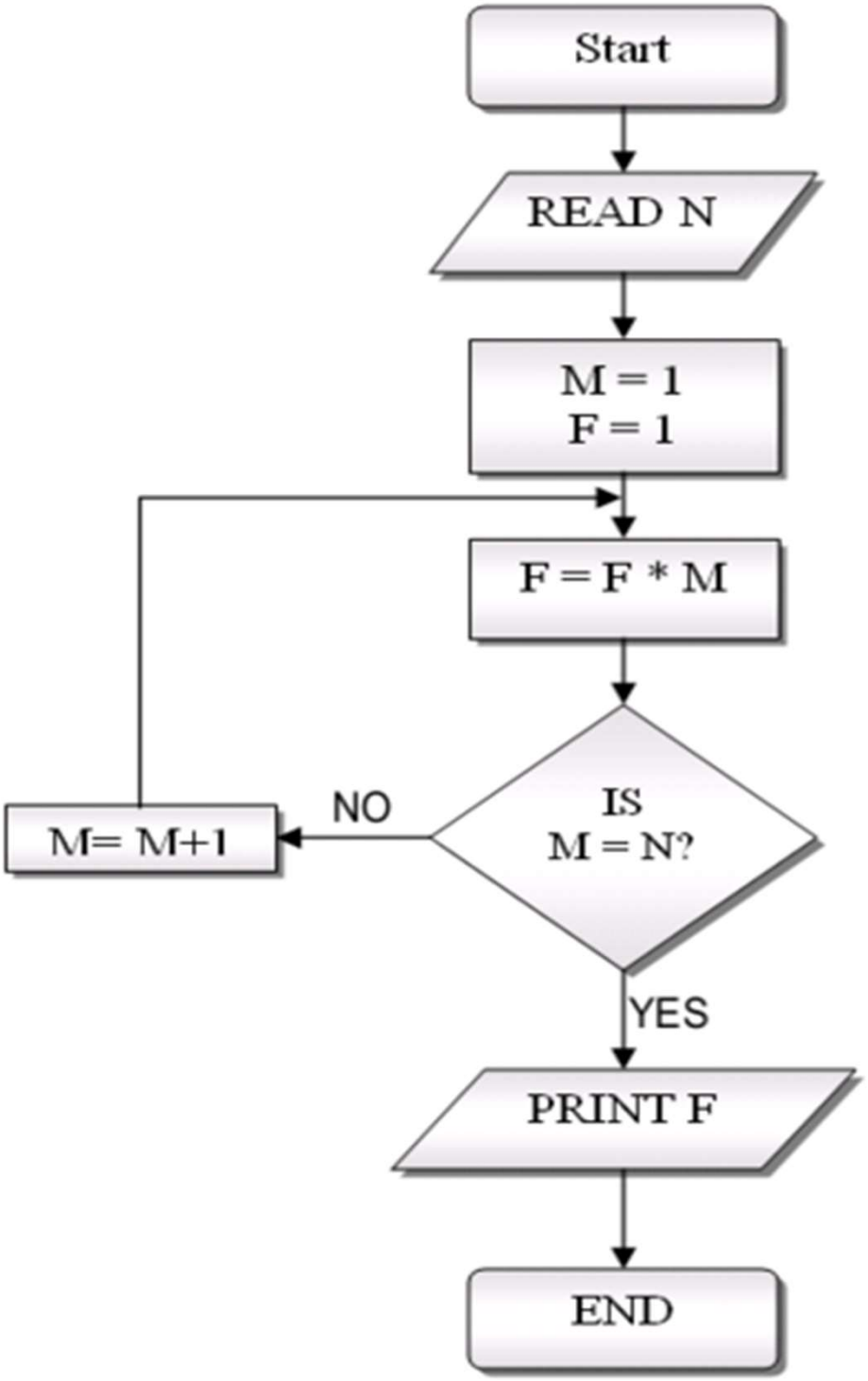
条件和行动	规则
条件声明（Condition Statement）	条件选项（Condition Entry）
行动声明（Action Statement）	行动选项（Action Entry）

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

伪代码

- 结合使用程序语言的逻辑结构和自然语言的表达能力描述程序逻辑：
  - 叙述上采用了编程语言的三种控制结构：顺序、条件决策和循环；
  - 使用了一些类似于编程语言关键字的词语来表明叙述的逻辑，例如IF、THEN、ELSE、DO、DO WHILE、DO UNTIL等等；
  - 在格式上，使用和编程语言相同的缩进方式来表明程序逻辑结构。
  - 尽量使用简短语句以利于理解，只使用名词和动词，避免使用容易产生歧义的形容词和副词。

程序流程图



主要内容

- 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

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：
  - 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。
  - 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。

Sales.total()方法代码

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

Mock Object

```
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;
        }
}
```

JUnit测试代码

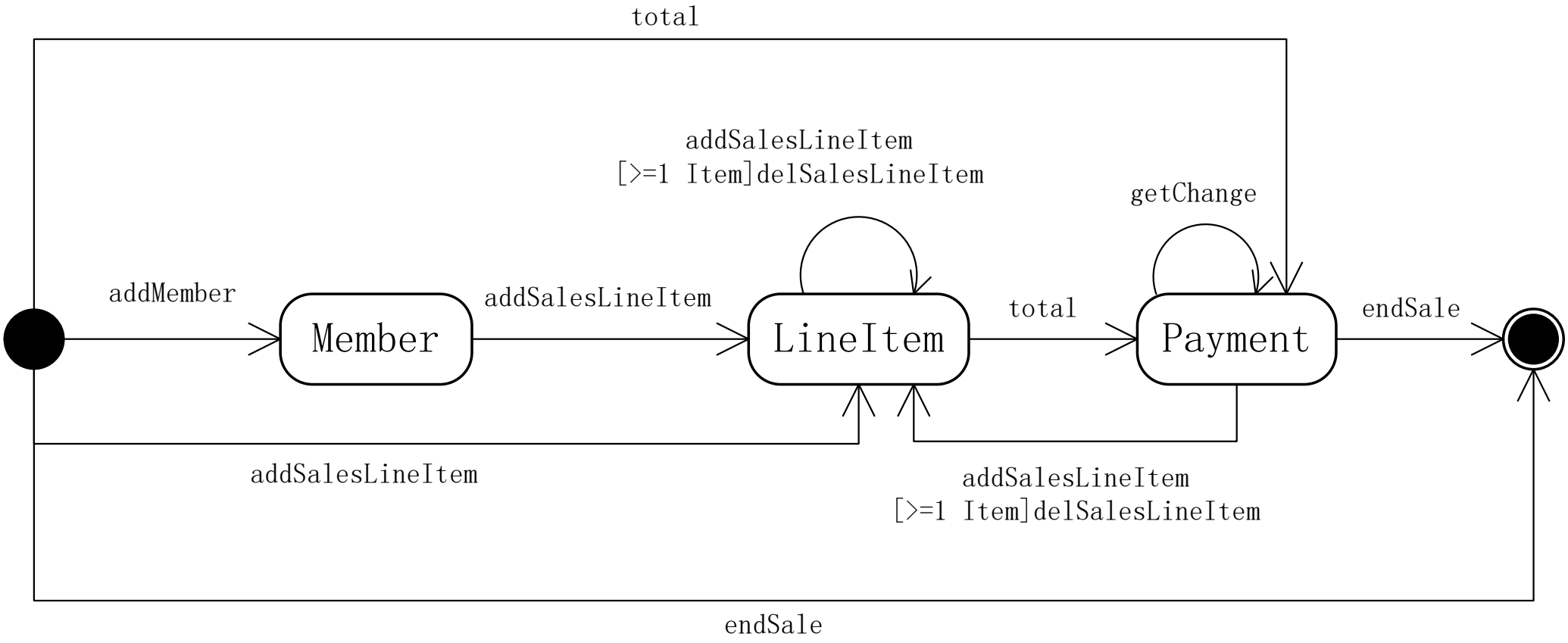
```
public class TotalTester {
        @Test
        public void testTotal () {
            MockSalesLineItem mockSalesLineItem1
                                    = new MockSalesLineItem (50, 2);
            MockSalesLineItem mockSalesLineItem2;
                                    = new MockSalesLineItem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

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

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。

Sales类的状态图


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

测试用例线索

输入		预期输出
方法	当前状态	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

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

.....			

主要内容

- 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

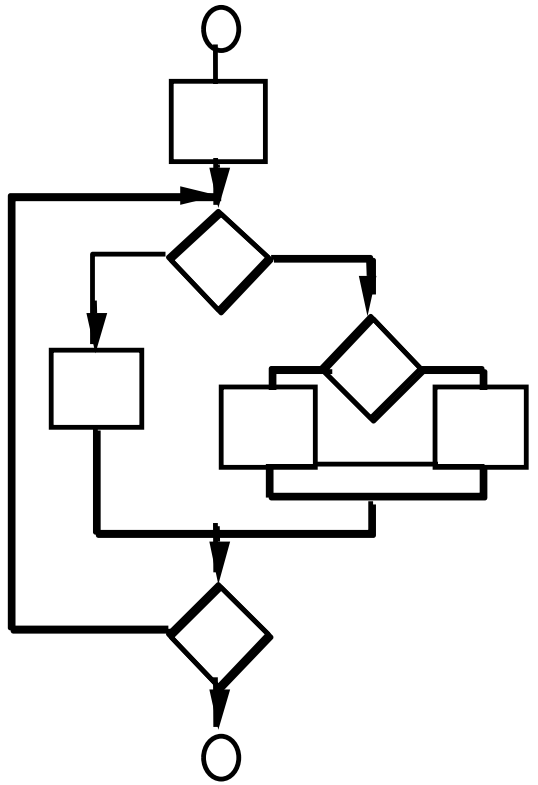
18.6 代码复杂度度量

- 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。

圈复杂度

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。



圈复杂度

- 还有一种简单的算法是直接计数程序中决策点的数量：
  1. 从1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1：if, while, repeat, for。
  3. 给case语句中的每一种情况都加1。

- [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。

类的复杂度

- [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。

总结

- 高质量的程序代码需要进行仔细设计
  - 易读
  - 易维护
  - 可靠
- 可以使用技术模型手段帮助进行代码设计
- 要使用单元测试验证已完成的代码
- 代码复杂度在一定程度上可以客观反映代码质量
- 要避免常见的代码错误