



# 软件工程

## 第十一章 软件实现

乔立民

qlm@hit.edu.cn

2011年5月25日

# 主要内容

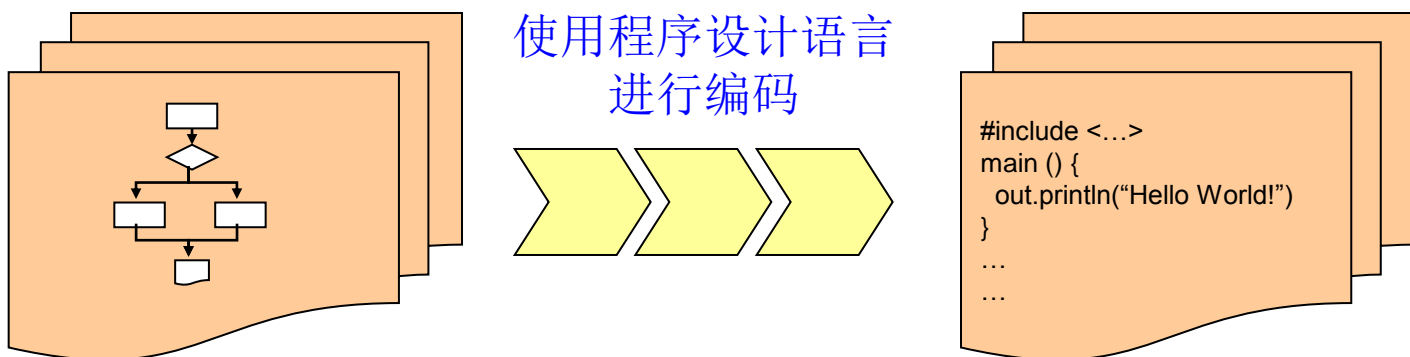
11.1 什么是软件编码

11.2 高质量的子程序

11.3 防御式编程

11.4 重构

# 软件编码



软件设计描述(SDD) → 程序代码(Program Code)

逻辑概念(不可执行) → 物理实体(可执行)

# 课堂讨论1

- 软件编码是将软件设计模型机械地转换成源程序代码，这是一种低水平的、缺乏创造性的工作。软件程序员是所谓的“软件蓝领” (**software blue-collar**)。
- 你是否认同这种观点？
- 如果不认同，你如何看待软件编码？



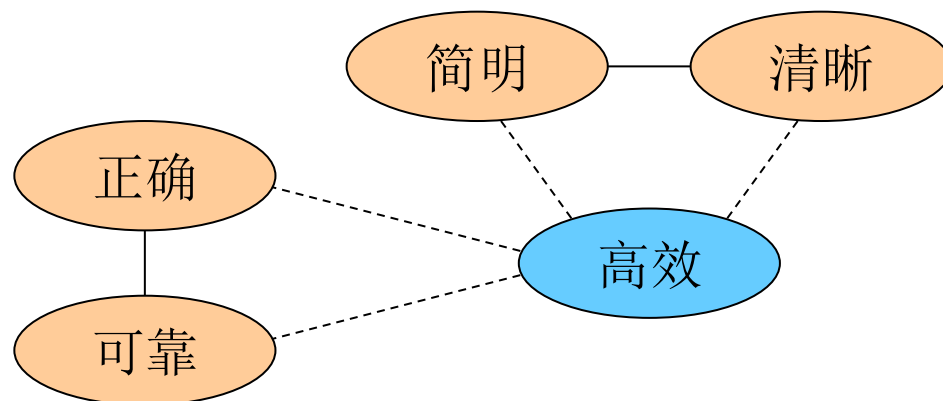
# 软件编码

- 软件编码是一个复杂而**迭代的过程**，包括程序设计(program design)和程序实现(program implementation)
- 软件编码要求
  - 正确的理解用户需求和软件设计思想
  - 正确的根据设计模型进行程序设计
  - 正确而高效率的编写源代码和测试
- **软件编码是设计的继续**，会影响软件质量和可维护性



# 软件编码的基本要求

- 养成良好的编码习惯；
- 选择合适并且熟悉的编码语言；
- 对代码进行持续不断的测试与优化；



# 软件编码的基本要求

结果名次 编码要求	评判项目	清晰性		效率		开发时间
		程序	输出	内存数	语句数	
程序可读性最佳		1-2	2	3	3	4
输出可读性最佳		1-2	1	5	5	2-3
空间复杂性最佳		4	4	1	2	5
语句简洁性最佳		5	3	2	1	2-3
开发时间最短		3	5	4	4	1

# 主要内容

11.1 什么是软件编码

11.2 高质量的子程序

11.3 防御式编程

11.4 重构



# 高质量的子程序

- 子程序是为实现一个特定目的而编写的一个可被调用的方法(**method**)或过程(**procedure**)。

## C++示例：低质量的子程序

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
    double & estimRevenue, double ytdRevenue, int screenX, int screenY,
    COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
    int expenseType )
{
    int i;
    for ( i = 0; i < 100; i++ ) {
        inputRec.revenue[i] = 0;
        inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
    }
    UpdateCorpDatabase( empRec );
    estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
    newColor = prevColor;
    status = SUCCESS;
    if ( expenseType == 1 ) {
        for ( i = 0; i < 12; i++ )
            profit[i] = revenue[i] - expense.type1[i];
    }
    else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
    }
    else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
}
```

# 高质量的子程序


## ■ 示例程序的问题

- 有一个很差劲的名字
- 布局不好
- 输入变量的值被改变了，如果是输入变量就不应该被修改
- 没有单一目的，做了许多事情
- 没有注意防范数据错误
- 使用了若干神秘数值，4，1，2，3
- 未使用其中一些参数：screenX，screenY
- 参数传递方法有误：prevColor为引用参数，但未赋值
- 参数太多了
- 参数顺序混乱，且未注释

# 创建子程序的理由

- 降低复杂度
- 引入中间、易懂的抽象

```
if ( node <> NULL ) then
  while ( node.next <> NULL ) do
    node = node.next
    leafName = node.name
  end while
else
  leafName = ""
end if
```

 leafName = GetLeafName( node )

- 避免代码重复
- 隐藏顺序
- 提高可移植性
- 改善性能

# 在子程序层上设计

## ■ 功能内聚

## ■ 好的子程序名字

- 描述子程序所作的所有事情
- 给函数命名时要对返回值有所描述
- 为常用操作确立命名规则

## ■ 合理使用子程序参数

- 按照 输入-修改-输出的顺序排列参数，建立相应命名规则(i\_,m\_,o\_)
- 使用所有参数
- 不要把子程序的参数用做工作变量
- 子程序的参数个数限制在大约7个以内
- 为子程序传递用以维持其接口抽象的变量或对象

# 主要内容

11.1 什么是软件编码

11.2 高质量的子程序

11.3 防御式编程

11.4 重构

# 防御式编程

- 子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据（在一开始不要在代码中引入错误）
- 检查所有来源于外部的数据的值
- 检查子程序所有输入参数的值
- 决定如何处理错误的输入数据

# 1.断言（Assertions）

- 断言是指在开发期间使用的、让程序在运行时进行自检的代码（通常是一个子程序或宏）。
- 断言为真，则表明程序运行正常，而断言为假，则意味着它已经在代码中发现了意料之外的错误。
- 常见的断言特性
  - 前置条件断言：代码执行之前必须具备的特性
  - 后置条件断言：代码执行之后必须具备的特性
  - 前后不变断言：代码执行前后不能变化的特性

**assert denominator !=0 : "denominator is unexpectedly equal to 0.";**

# 1.断言（Assertions）

## ■ 用断言检查如下假定：

- 输入参数或输出参数取值处于预期的范围内
- 子程序开始（或结束）执行时文件或流是处于打开（或关闭）的状态
- 子程序开始（或结束）执行时，文件或流的读写位置处于开头（或结尾）处
- 文件或流已用只读、只写或可读可写方式打开
- 仅用于输入的变量的值没有被子程序所修改
- 指针非空
- 传入子程序的数组或其他容器至少能容纳X个数据元素
- 表已初始化，存储着真实的数值
- 子程序开始（或结束）执行时，某个容器是空的（或满的）



# 使用断言的建议

- 用错误处理代码来处理预期会发生的情况，用断言来处理绝不应该发生的情况
- 避免把需要执行的代码放到断言中
- 用断言来注释并验证前条件和后条件
- 对于高健壮性的代码，应该先使用断言再用错误处理

## 2. 错误处理技术

- 错误是程序中可预料到的（IF）
- 错误处理技术有：
  - 返回中立值（没有危害的值），如0、空格等
  - 换用下一个正确的数据
  - 返回与前次相同的数据
  - 换用最接近的合法值
  - 把警告信息记录到日志文件中
  - 返回一个错误码
  - 当错误发生时显示出错消息
  - 调用错误处理子程序或对象
  - 用最妥当的方式在局部处理错误
  - 关闭程序

# 健壮性与正确性

- **正确性：**意味着永远不返回不准确的结果，哪怕不返回结果也比返回不准确的结果好
- **健壮性：**意味着要不断尝试采取某些措施，以保证软件可以持续地运转下去，哪怕有时做出一些不够准确的结果

### 3.异常

- 异常是把代码中的错误或异常事件传递给调用方代码的一种特殊手段
- 如果在一个子程序中遇到了预料之外的情况，但不知道该如何处理的话，它就可以抛出一个异常

### 3.异常

- 异常和继承有一点是相同的：审慎明智地使用时，可降低复杂度；草率粗心地使用时，会让代码无法理解
  - 用异常通知程序的其他部分，发生了不可忽略的错误
  - 只在真正例外的情况下才抛出异常
  - 不能用异常来推卸责任
  - 在恰当的抽象层次抛出异常
  - 在异常消息中加入关于导致异常发生的全部信息
  - 避免使用空的`catch`语句
  - 了解所用函数库可能抛出的异常
  - 考虑创建一个集中的异常报告机制
  - 考虑异常的替换方案

## 4. 隔离程序

- 隔栏(**barricade**)是一种容损策略
- 以防御式编程为目的而进行隔离的一种方法，是把某些接口选定为“安全”区域的边界。对穿越安全区域边界的数据进行合法性校验，并当数据非法时做出敏锐的反应
  - 在输入数据时校验数据格式，将其转换为恰当的类型

## 5.辅助调试的代码

- 辅助调试代码

- 输出信息，如`System.out.println`,`alert()`

# 主要内容

11.1 什么是软件编码

11.2 高质量的子程序

11.3 防御式编程

11.4 重构



# 重构

- 在不改变软件外部行为的前提下，对其内部结构进行改变，使之更容易理解并便于修改。

—Martin Fowler

- 重构的类型
  - 数据级重构
  - 语句级重构
  - 子程序级重构
  - 类实现重构
  - 类接口重构
  - 系统级重构

# 重构

## ■ 重构的理由

- 代码重复
- 冗长的子程序
- 循环过长或嵌套过深
- 内聚性太差的类
- 类的接口未能提供层次一致的抽象
- 拥有太多参数的参数列表
- 类的内部修改往往被局限于某个部分
- 变化导致对多个类的相同修改
- 对继承体系的同样修改
- **case**语句需要做相同的修改
- 同时使用相关数据并未以类的方式进行组织
- 成员函数使用其他类的特征比使用自身类的特征还要多
- 过多使用基本数据类型

# 重构

## ■ 重构的理由

- 某个类无所事事
- 一系列传递流浪数据的子程序
- 中间人对象无事可做
- 某个类同其他类关系过于亲密
- 子程序命名不恰当
- 数据成员被设置为公用
- 某个派生类仅使用了基类的很少一部分成员函数
- 注释被用于解释难懂的代码
- 使用了全局变量
- 在子程序调用前使用了设置代码，或在调用后使用了收尾代码
- 程序中的一些代码似乎是在将来的某个时候才会用到的

# 重构

## ■ 重构策略

- 在增加子程序时进行重构
- 在添加类时进行重构
- 在修补缺陷时进行重构
- 关注易于出错的模块
- 关注高度复杂的模块
- 在维护环境下，改善你手中正在处理的代码
- 定义清楚干净代码和拙劣代码之间的边界，然后尝试把代码移过这条边界

# 重构

## ■ 安全的重构

- 保存初始代码
- 重构的步伐请小些
- 同一时间只做一项重构
- 把要做的事情一条条列出来
- 设置一个停车场
- 多使用检查点
- 利用编译器警告信息
- 重新测试
- 增加测试用例
- 检查对代码的修改
- 根据重构风险级别来调整重构方法



结束

2011年5月25日