24-26

chp24 重构

- 1软件演化的类型
- 2重构简介
  - 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
  - 重构的理由
    - 代码重复
      - 复制粘贴即代码之谬
    - 冗长的方法
    - 循环过长或者嵌套过深
    - 内聚性太差的类
      - 每个类应该只负责具有内在相互关联的任务
    - 类接口未能提供层次一致的抽象
    - 拥有太多参数的方法列表
    - 类的内部修改往往局限于某个部分
      - 按照独立的功能拆分多个类
    - 变化导致对多个类的相同更改
    - 对继承体系的同样修改
    - case语句需要做相同的修改
    - 同时使用的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类自身的还多
    - 过多使用基本数据类型
    - 某个类无所事事
    - 一系列传递流浪数据的子程序
    - 子程序命名不规范
    - 数据成员被设置成公用
    - 注释用于解释难懂的代码
    - 使用了全局变量
    - 子程序使用前后使用了设置代码

- 程序中有超前设计的代码（当前用不到的）
- 拒绝重构的理由

3特定的重构

数据层的重构

- 消除程序中魔数
- 清晰没有误解的命名
- 表达式内联化
- 表达式函数化（复用）
- 引入中间变量
- 基础数据类型转化为类对象
- 类型码转换为枚举、类
- 将数组转化为对象
- 封装集合类：提供只读对象以及对应的添加删除方法
- 用数据替代传统记录

语句层级的重构

分解布尔表达式

- 引入中间变量简化复杂表达式
- 通过变量名说明表达式含义

将表达式转换成命名准确的布尔函数

- 提高可读性
- 复用

- 多态替换条件语句
- 创建和使用null对象，而不是检测空值

子程序级的重构

类实现的重构

- 改变成员函数和成员数据的位置
- 特殊代码提升为派生类
- 相同代码放到基类中

类接口的重构

- 将成员函数放到另一个类中

将一个类变成两个

- 每个类保持职责单一

- 删除类
- 去除委托关系
- 去掉中间人
  - A-B-C 有时候直接A调用C更好
- 用委托代替继承
- 尽量使用包含，而不是继承
- 不能修改的成员，去掉set函数
- 封装不使用的成员函数

- 系统级的重构
  - 为无法控制的数据创建明确的索引源
  - 将单向的类联系改为双向
    - 两个类相互之间需要调用时
  - 双向的类联系改为单向
    - 与上述相反
  - 用factoryMethod模式而不是简单的构造函数
  - 保持错误处理的统一
    - 用异常代理错误码，或者反之

- 4安全的重构
  - 保存初始代码
  - 重构的步伐要小一些
  - 同一时间只做一项重构
  - 把要做的事情一条条列出来
  - 设置一个停车厂。把一些可以放到后面重构的工作列出来
  - 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
    - 相对于大规模修改，小的改动更容易出错

- 根据重构风险级别来调整重构方法

- 不宜重构的情况：
  - 不要把重构当成先写后改的代名词
  - 避免用重构代替重写
    - 代码太烂：考虑推到重来，重新设计、重新开发

- 5重构策略
  - 重构时机
    - 增加方法时进行重构
    - 添加类的时候进行重构
    - 修补缺陷的时候进行重构
  - 重构重点
    - 关注容易出错的模块
    - 关注高复杂度的模块
    - 维护环境下，改善手中处理的代码
      - 不要让代码在你手上变烂
    - 定义清楚干净代码和拙劣代码之间的边界，把代码挪到边界内

- 从最初状态大多数写得很糟糕的遗产代码--->>>目标状态 大多数写得很好的重构后的代码

chp25 代码调整策略

- 25.1 性能概述
  - 程序可读性、可维护性和性能之间的取舍的艺术
  - 质量特性和性能
  - 性能和代码调整
    - 程序需求
      - 在你着手解决性能问题之前，先确认必要性
    - 程序设计
      - 一开始有所偏重，性能和易维护性
    - 类和子程序设计
      - 合适的数据模型和算法
    - 同操作系统的交互

- 代码编译
  - 编译器级别进行代码的优化
- 硬件
  - 考虑性价比
- 代码调整（最后一波）
  - 较小规模的修改

- 25.2 代码调整简介
  - pareto法则
    - 用20%的努力获取80%的成效
    - the best is the enemy of the good(完美是优良之大敌）
  - 何时调整代码
    - 程序应该使用高质量的设计，把程序编写正确。使其模块化并易于修改，将让后期的维护工作变得很容易。最后再去调优性能（如果有必要）
  - 编译器优化
- 25.3 蜜糖和哥斯拉
  - 常见的低效率方面
    - I/O
    - 分页
    - 系统调用
    - 解释型语言
    - 错误
  - 常见错误的相对效率
- 25.4 性能测量
  - 性能往往是反直觉的，编译器优化都会影响你程序的实际效果
  - 性能测量应当精确
- 25.5 反复调整
  - 通常大部分优化算法单独看起来无法达成目标，但是结合到一起可以达到目标
- 25.6 代码调整方法总结

chp26代码调整技术

- 26.1 逻辑
  - 在知道答案后停止判断

- 按照出现频率来调整判断顺序
- 相似逻辑结构之间的性能比较（case |if else）
- 表查询替代复杂表达式（状态机）
- 懒加载，使用惰性求值

- 26.2 循环
  - 将判断外提
  - 合并|融合
  - 尽可能减少循环内部的工作
  - 把最忙的循环放在最内
  - 削减强度
- 26.3 数据变换
  - 使用整型数而不是浮点数
  - 数组维度尽可能少
  - 尽可能减少数组引用
  - 使用辅助索引
    - 字符串长度索引
  - 使用缓存机制
    - 缓存优化，代价越高，请求相同的次数越多，使用缓存的效果越好
- 26.4 表达式
  - 利用代数恒等式替换
  - 削减计算强度
  - 编译期初始化
    - 尽可能耗时的计算挪到编译期算好
  - 小心系统函数
  - 使用正确的常量类型
  - 预先算出结果
    - 提取程序中公共的部分，暂存起来使用。减少复杂耗时的部分的计算次数
  - 删除公共子表达式
- 26.5 子程序
  - 将方法体整的简短的各种好处
  - 将子程序重写为内联

- 26.6 用低级语言重写代码
    - 一般5%的代码往往会占据50%的执行时间
- 26.7 变得越多，事情反而越没变

- **愈是追求完美，越有可能完不成任务。程序员首先应该实现程序应该具备的所有功能，然后再使程序臻于完美。而这时，需要精益求精的部分通常是很少的。**
- **不要优化一次尝到甜头就止步不前，要多次进行优化**