【软件构造】第一章第二节 软件开发的质量属性

软件构造第一章第二节 软件开发的质量属性

上一节告诉我们不同视角下“软件构造的结果”分别是什么，在这一节我们要清楚“什么样的结果”算好的。这一章我们介绍了内部、外部的若干质量属性，其中 可理解性(第四章)、可复用性(第五章)、可维护性(第六章)、健壮性(第七章)、性能(第八章)是五个关键质量目标。

Outline

- 软件系统的质量属性
  - 外部与内部质量因素
  - 重要的外部质量因素
  - 质量因素之间的权衡
- 软件构建的五个关键质量目标
  - 了解有哪些质量目标
  - 违反有什么后果
  - 每种质量因素需要研究的施工技术

Notes

【外部和内部质量属性】

- 外部质量属性是指正确性、外观、速度等影响客户的属性
- 内部属性是指易于理解、可读性等影响开发人员和软件自身的属性
- 二者关系：外部受内部制约

外部质量属性

【正确性】

- 在规格说明书描述范围之内满足正确性
- 保证正确性的技术
  - 有限制的正确：只保证自己层面正确，假设调用的都是正确的
  - 测试与调试
  - 防御性编程
  - 形式化编程（采用很多数学技术）

【健壮性】

- 碰到异常情况进行适当的响应
- 出现规格说明书说明之外的情况由健壮性处理
  - 响应异常情况
  - 给出错误提示

    - 正常退出或降级

【可扩展性】

- 软件产品适应规格变化的容易程度
- 传统方法通过固化需求（瀑布模型）进行编程
- 两个基本策略
    - 设计简洁
    - 离散化：低耦合

【可复用性】

- 软件模块能否被其他程序很方便地使用
- 例子：开发备注、封装

【兼容性】

- 能够与其他人员进行交互
- 跨平台、跨软件
- 实现方法：一致性和标准化（一致的方法和标准）
    - 标准文件格式
    - 标准数据结构
    - 标准用户接口
    - 最通用：标准协议

【效率】

- 程序运行中对CPU、硬盘的占用带宽；
- 实现效率是不能牺牲正确性，要再多指标之间权衡
- 实现方法：
    - 好的算法
    - I/O技术
    - 内存管理
- 功能问题都可以加一层抽象进行处理；性能问题都可以去掉一层抽象来解决

【可移植性】

- 是否容易由一个环境转移到另一个环境
- 由于访问OS本地类库、插件等问题导致的移植后无法正常运行

【应用性】

- 用户是否容易使用，不影响专业人员的使用情况下，方便初学者
- 方法：
    - 结构清晰的设置
    - UI设计：理解用户需求

【功能性】

- 蠕变特征（不好的现象：开发者开发越来越多的功能，造成程序的复杂和不灵活）
- 原则：在保证整体质量不降低的情况下进行更新
- 策略：增量式模型

【及时性】

- 在规定时间内完成：时间效率高

【其他质量特性】

- 可验证性：如管理系统的效果难以验证
- 完整性：不会被非法访问干扰修改，防止数据不一致（如使用private）
- 可修改性
- 资金

内部质量属性

- 从LOC（line of code）到圈复杂度：用来衡量一个模型判定结构的复杂程序
- 耦合度和内聚度
- 代码是否可读、可理解、简洁
- 完整性
- 大小

【均衡决策】

- 完整性与易用性冲突
- 经济性与功能性冲突
- 性能与可复用、可移植性冲突
- 及时性与可延展性冲突

以效率为导向，以正确性为最重要

【**OOP**如何保障质量属性（一些技术，在后续博客中会有所涉及）】

- **Correctness**: encapsulation, decentralization
- **Robustness**: encapsulation, error handling
- **Extendibility**: encapsulation, information hiding
- **Reusability**: modularity, component, models, patterns
- **Compatibility**: standardized module and interface
- **Portability**: information hiding, abstraction
- **Ease of use**: GUI components, framework
- **Efficiency**: reusable components,
- **Timeliness**: modeling, reuse
- **Economy**: reuse
- **Functionality**: extendibility

#五个关键的质量属性

- easy to understand
- ready for change
- cheap for develop
- safe from bugs
- efficient to run

【可理解性】

- 在构建时
  - 代码层要注意（函数规约）
    - 变量 / 子程序 / 语句 的命名与构造标准
    - 代码布局与风格
    - 注释
    - 复杂度
  - 组件层要注意构件和项目的可理解性
    - 包的组织
    - 文件的组织
    - 命名空间

        - 在时段中，代码层注意重构
    - 在运行时，代码层注意跟踪日志

【可复用性】

- 构建时
    - 代码层应注意
        - ADT / OOP
        - 接口与实现分离
        - 继承 / 重载 / 重写
        - 组合 / 代理
        - 多态
        - 自类型与泛型编程
        - OO设计模式
    - 组件层注意
        - API接口设计
        - 类库
        - 框架

【可维护性与适用性】

- 构建时（面对需求的改变，能否做出及时的调整）
    - 代码层可采用
        - 模块化设计
        - 高内聚，低耦合
        - SOLID原则
        - OO设计模式
        - 面向图表的编程
        - 面向状态编程
        - 面向语法编程
    - 组件层除注意SOLID原则外，还应考虑GRASP原则
    - 在时段内使用SCM进行版本控制

【健壮性】

- Code level-build time-Moment
    - 错误处理
    - 异常处理
    - 断言
    - 防御型编程
    - 测试优先编程
- Component level-buildtime-period
    - 单元测试
    - 集成测试
- Build time-period
    - 回归测试
- run time-moment
    - 测试转储
- run time-period
    - 跟踪日志

【性能】

- 构建时，使用指定的设计模式
- 运行时
  - 在代码层次
    - 通过内存管理考虑空间复杂度
    - 通过算法性能计算时间复杂度
    - 利用代码调优生成更高效的目标代码
    - 在时段内进行性能分析和调整
  - 在组件层次
    - 采用分布式系统
    - 编写多线程的并行程序