中心主题

- **chp3-前期准备**
  - 架构的先决条件
    - 架构典型组成
      - 程序：定义分块
        - 每个块应该负责自己的部分，尽量减少对其他块的依赖
        - 明确通信规则，确定接口与调用关系
      - 数据设计
      - 可伸缩性
      - 错误处理（90%的代码占比）
        - 给出一种“一致的处理错误策略”
      - 容错性
        - 重试；备选逻辑；默认值
      - 重复造轮子问题
        - 前提是要说明自己定制的组件的独特性
      - 架构描述：欠描述和过度描述之间，把握好度
  - 花在前期准备上的时间长度
    - 子主题 1
- **chp4-关键的构建决策**
  - 选择编程语言
    - 每种编程语言都有优点缺点，你自己要清楚使用语言的优缺点
  - 编程约定
    - 开发前提前制定约定
- **chp5-软件构建中的设计**
  - 5.1 设计中的挑战
    - 设计是一个险恶的问题
      - 首先把问题解决一遍，发现要考虑的额外环节。定义问题（类似原型模式？）
      - 再次解决该问题，形成可行性方案
    - 设计过程，不断犯错不断改正

- 在编码前确定，代价最低
- 设计就是取舍的过程
  - 平衡各个方案的利弊
- 设计是不确定性的
  - 每个人的想法都不一样
- 设计是一个启发式的过程
- 设计是自然形成的
  - 是在不断的设计评估、非正式讨论、写实验代码以及修改实验代码中完成的

5.2 关键的设计概念

- 软件的首要技术使命：管理复杂度
  - 偶然的难题和本质的难题
  - 大系统拆分成小系统降低复杂度
  - 系统的目标：
    - 自己容易理解
    - 别人容易看懂
    - 很少有错误的代码
- 理想的设计特征
  - 最小化复杂度
    - 设计要简单易理解
    - 专注于一部分时安心忽略掉其他部分
  - 易于维护
    - 时刻想着这些维护程序员可能会就你写的代码提出的问题，把程序员当做听众
  - 松散耦合
    - 各部分组件之间关联最小
    - 合理化的抽象、封装 设计出关联尽可能少的类
  - 可扩展性
    - 增强系统功能不会破坏底层结构
    - 越是可能发生的改动，越不会给系统造成破坏
  - 可重用性
    - 设计的系统组成部分能在其他系统中复用

- 高扇入
  - 大量的类使用了某个给定的类，系统很好的利用了较低层次的类
- 低扇出
  - 一个类少量或者适中的引用其他的类（少于7个）
- 可移植性
- 精简性
  - 设计出来的系统没有多余的部分，不要有任何多余的代码
- 层次性
  - 假设你正在设计新系统要用到老代码-那就要做个兼容转换层。
- 标准技术
  - 一个系统依赖的外来的古怪的东西越多，别人在第一次理解他的时候就越头疼
- 设计的层次
  - 1.软件系统
  - 2.分解为子系统和包
  - 3.分解为具体的类
    - 把子系统分解为适当的类，确定类的接口以及互相交互的细节
  - 4.分解为类的数据结构和子程序
    - 通过定义子程序加深对接口类的理解
    - 返过来有助于对类接口的进一步修改，返回3的设计
  - 5.子程序的内部设计

5.3设计构造块：启发式方法

- 1 寻找现实世界中的对象
  - 辨识对象及其属性（field method（共有私有））
- 2.形成一致的抽象
  - 基类是一种抽象，能使你忽略子类特殊的细节
- 3.封装实现细节
  - 不能让你看到复杂概念的任何细节
- 4.当继承能简化设计就继承
- 5.信息隐藏
  - 将一个地方的设计和实现隐藏起来，使程序的其他部分看不到他们（隐藏复杂度）；接口编程

- 1. 秘密和隐私权
  - 类的可见性设计：哪些特性对外，哪些内部使用
- 2. 类接口设计：冰山原则
  - 过程是迭代式的，如果一次确认不了，那就多试几次，直到设计稳定下来
- 3.两种隐藏类型
  - 隐藏复杂度
  - 隐藏变化源：复杂的数据类型，文件结构，晦涩的算法等等
- 4.信息隐藏的障碍
  - 信息过度分散
    - 全局常量替换多处写死的代码
    - 数据细节过分暴露（ArrayList全局数组）
  - 循环依赖
  - 类内数据误认为是全局数据
  - 担心带来额外的性能损耗
    - 高度模块化的编码设计,更容易找出瓶颈，并针对小的模块进行重写升级
  - 信息隐藏的价值
    - 养成“我该隐藏什么的提问习惯”
- 5.找出容易改变的区域
  - 目标：把不稳定的区域隔离出来，将变化限制在一个方法、类或者包的内部（优秀设计师的潜质）
  - 1.找出看起来容易变化的项目
  - 2.把容易变化的项目分离出来
  - 3.把看起来容易变化的项目隔离出来
    - 设计好类接口，通过接口保护类的内部隐私
    - 容易变化的地方
      - 1.强业务相关逻辑
      - 2.硬件依赖
      - 3.输入和输出
      - 4.困难的设计区域和构建区域
        - 方便后期对其进行的设计和构建进行重写

- 方便后期对其扩务的设计和构建进行重与
- 4.预料不同程度的变化
  - （考虑成本）重点解决那些容易变化切容易处理的地方
  - 找出程序中最小子集确定不容易变化的核心；在此基础上慢慢添加功能，依据信息隐藏的原则进行设计
- 6.保持松散耦合
  - 表示类与类之间，方法与方法之间关系的紧密程度。努力使模块之间的连接尽量的简单
  - 1、耦合评判标准
    - 规模（模块之间的连接数）
    - 可见性：模块间的连接关系
    - 灵活性：模块间的连接是否容易改动
  - 2.耦合的种类
    - 简单数据参数耦合：模块之间通过基本数据类型参数传递数据
    - 简单对象耦合：模块间的通过实例化的模块对象
    - 对象参数耦合：obj1 要求obj2传递obj3给它进行交互
    - 语义上的耦合：模块1根据模块的具体逻辑做特殊处理
- 7.查阅常用的设计模式：（陷阱：不要为了模式而模式）
- 8. 其他的启发式方法
  - 高内聚性(尽量使类包含一组功能密切相关的功能)
  - 构造分层结构
    - 最通用、抽象的放在最上面。越具体的放到最底层
  - 严格描述类契约
  - 分配职责|确定每个对象该负什么职责
  - 为测试而设计
    - 确保每个模块都可以独立测试
  - 避免失误
    - 注意从失败中汲取经验
  - 有意识的选择绑定时间
  - 考虑使用蛮力解决
    - 可行的蛮力解决方案好于优雅却不能用的解决方案
  - 画图说明

- 画图说明
  - 一幅图顶的上1千句话
- 保持设计的模块化
  - 模块化的目标，使你的的模块黑盒化，对于任何输入你都能准确预测输出，

- 5.4 设计实践
  - 迭代
    - 设计中的一次迭代，可以对高层抽象和底层细节进行互相验证，从而可以在后面的迭代中更好的设计高层逻辑
  - 分而治之
    - 将程序分模块，然后处理每个模块内的细节，遇到解决难题就可以尝试迭代。
  - 自上而下和自下而上的设计方法
    - 前者是一种分解策略：从一般性问题出发，把问题分解成可控的部分
    - 后者是一种合成策略：后者从可控的部分出发，构造一种通用方案
  - 建立实验原型
    - 写出用于回答特定设计问题的、量最少并且能够随时扔掉的代码
    - 风险点：原型代码要做到用完即废弃，不能当做系统实现代码
  - 合作设计
  - 要做多少设计才够
  - 记录设计成果
    - wiki、总结记录、照片、uml
- 5.5 对流行的设计方法的评论
  - 设计任何细节和不做任何设计是两种肯定错误的处理方式