软件构造
SOFTWARE CONSTRUCTION

主讲教师：李翔　冯桂焕

2013年秋季

课程描述

- 课程名称：
  - 软件构造（软件代码开发技术）
- 英文名称：
  - Software Construction
- 时间分配及学分
  - 课堂教学(36课时)＋课后作业与阅读(36课时)
  - 3学分

课程简介

- 目的
  - 培养良好的编码习惯和编码技巧
- 课程内容
  - 软件构造的一般原则
    - 模块设计、代码重用、类设计等
  - 软件构造的常用技巧
    - 契约式设计、防御式编程、异常处理、配置式编程、基于状态转移和基于表的软件设计、基于语法分析的设计等
  - 软件构造的形式化方法
    - 规范说明语言及其应用、形式化构造工具
  - 软件构造工作的执行要点
    - 编码及其规范、工具使用、单元测试技巧、性能优化等

教材与参考资料

- 教材与指定阅读材料
  - 《代码大全（第二版）》，Steve McConnell，电子工业出版社
  - 《程序员修炼之道-从小工到专家》，Andrew Hunt, David Thomas，电子工业出版社
- 参考书目
  - 《面向对象软件构造（英文 第二版）》，Bertrand Meyer，机械工业出版社
- 软件构造课程相关网站
  - http://cs.gmu.edu/~pammann/619-sched.html
  - http://www.site.uottawa.ca/~bochmann/SEG2106/index.html
  - http://mahler.cse.unsw.edu.au/webcms//course/index.phtml?cid=1492
  - http://sewiki.iai.uni-bonn.de/teaching/lectures/oosc/2008/schedule

课程考核

- 出勤、日常练习 (10%~20%)
- 闭卷考试 (50%~60%)
- 实践项目 (30%~40%)
  - 分析简单的代码段，找出问题并加以改正；
  - 按照指定的编码规范，分析简单的代码段，找出不规范的地方并加以改正；
  - 分析简单的代码段，进行优化处理；
  - 为简单的需求建立形式化描述。

思考

- 我到底要从这门课中学到什么？
  - 与《软件工程》的关系
  - 与《程序设计语言》（如C++、Java等）的区别
  - 与《数据结构和算法》的区别
  - 与《离散数学》的区别
  - 与《软件系统的设计与体系结构》的区别
  - 与你能够想到的其它课程的区别

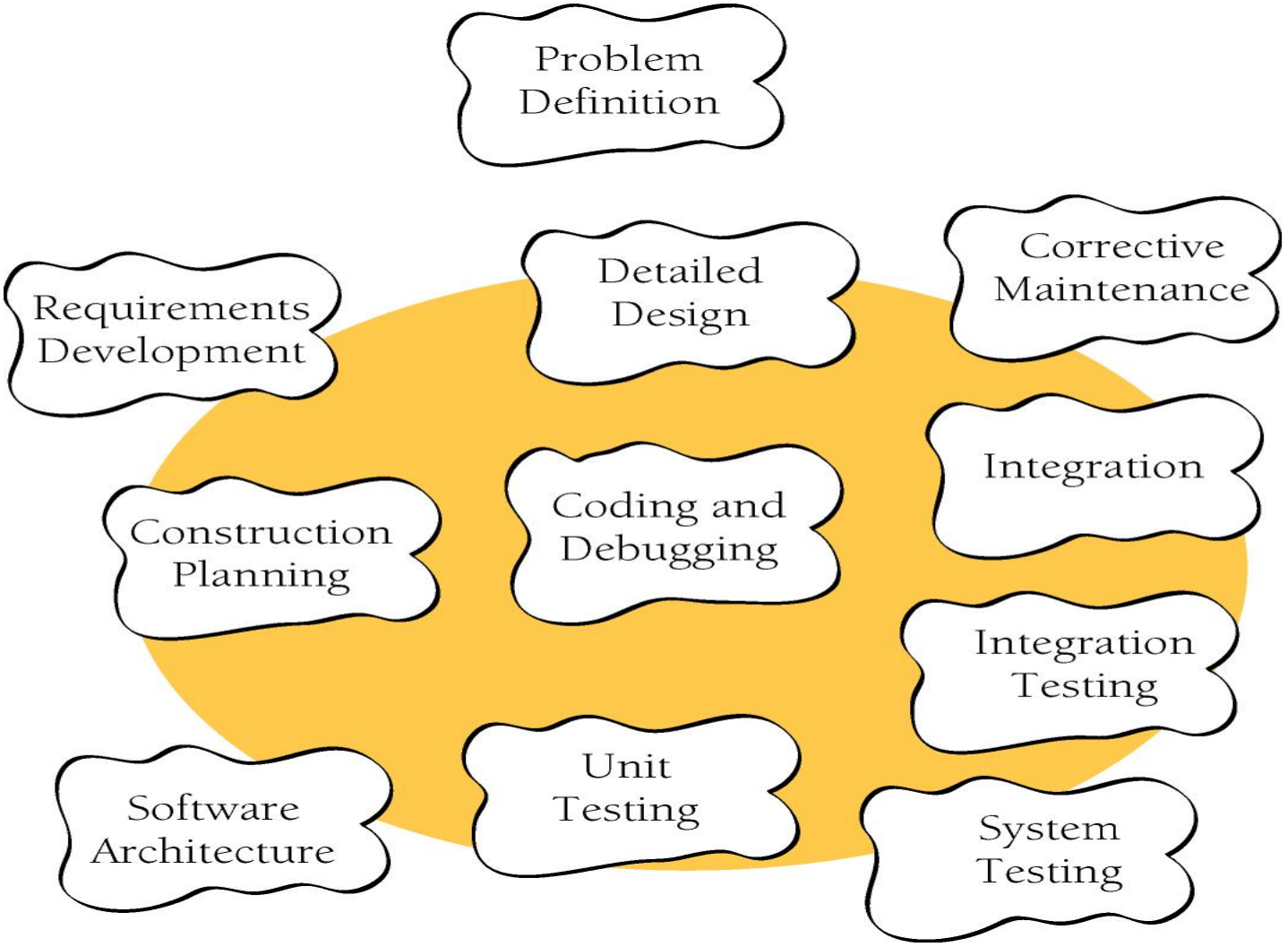
我的联系方式

- 办公室
  - 鼓楼校区费彝民楼926室
- 办公电话
  - 8362 1360 转 936
- 学院邮箱
  - fgh@software.nju.edu.cn
- TSS课程模块

课程框架

- Part I：构造技术
  - 软件构造基础
  - 模块化设计
  - 软件复用
  - 抽象数据类型
  - 类的设计和使用
  - 合约编程
  - 错误处理
  - 分布式程序构造
  - 单元测试
  - 重构
  - 极限编程
- Part II：构造管理
  - 构造管理
  - 构造标准
  - 构造环境和
- Part III：形式化构造方法
  - 形式语言基础
  - 规范说明语言与方法
  - 规范说明语言的应用
  - 形式化构造工具

Software Construction



Part I：构造技术

- 软件构造基础
  - 软件构造的概念
  - 软件构造的重要性
  - 软件构造的常用技术
  - 常用软件隐喻

基本概念

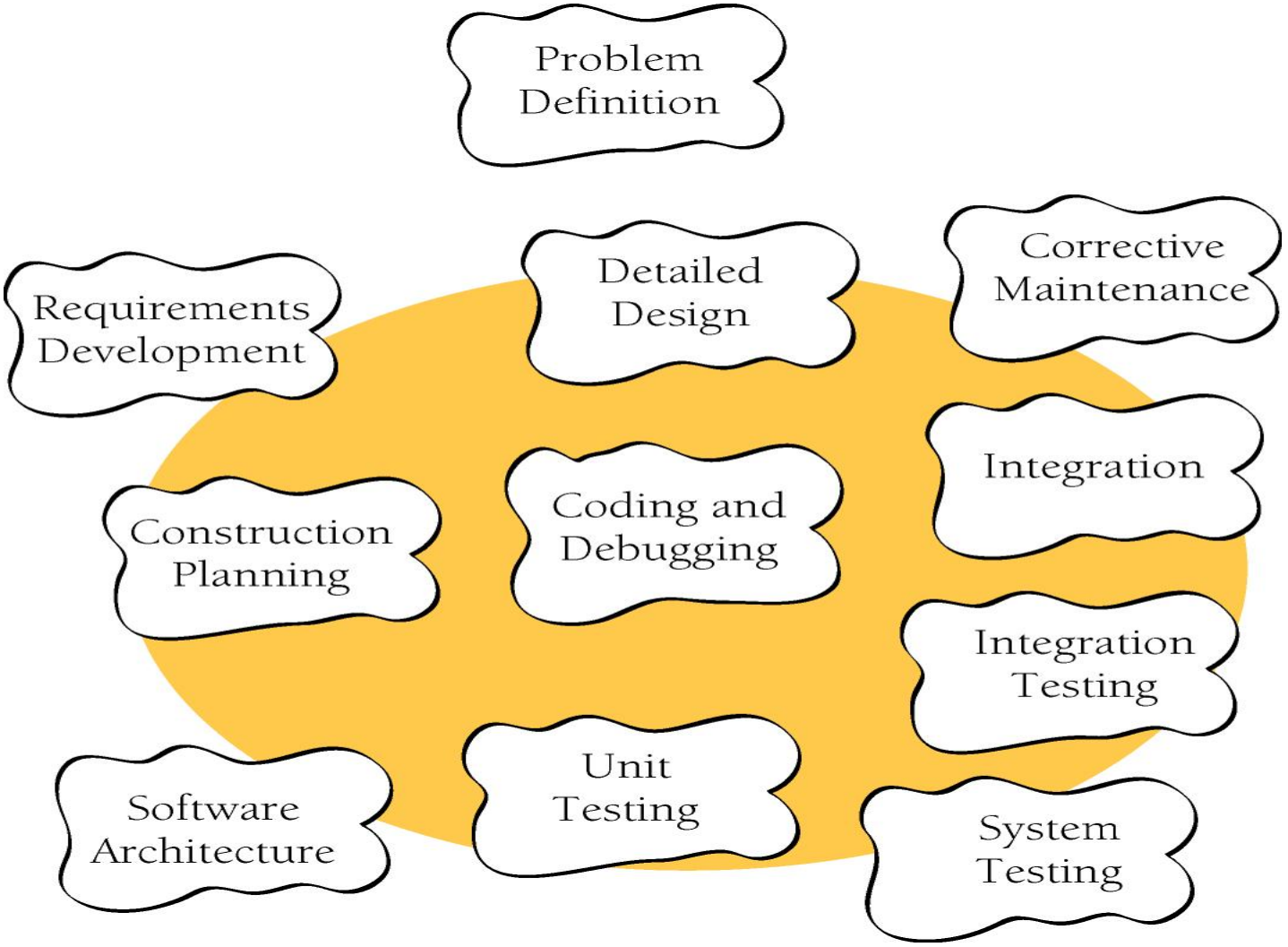
- 什么是软件构造
  - 软件构造是编码和调试
  - 部分的详细设计和单元测试
- 测试和调试的区别
  - 测试是为了发现错误
  - 调试是为了改正错误

软件开发包括：

- 问题定义
- 需求分析
- 实现计划
- 总体设计
- 详细设计
- 编码实现
- 系统集成
- 单元测试
- 系统测试
- 校正性的维护
- 功能强化

软件构造 ≠ 编码实现

Software Construction



哪些活动不属于软件构造？

- 需求分析
- 架构设计
- 用户界面设计
- 系统测试
- 系统维护
- 项目管理

软件构造的重要性

- 构造活动是开发软件的重要组成部分
  - 构造活动占整个开发活动总时间的30%-40%
- 构造活动在开发软件中处于枢纽地位
  - 需求分析与架构设计——构造——系统测试
- 把主要精力集中于构造活动，可以极大地提高程序员的生产效率
  - 不同程序员的生产率差异在10到20倍
- 创建活动的产品，源代码，往往是软件的唯一精确描述
  - 需求规格说明和设计文档可能过时，但源码不会
- 创建活动是唯一一项必不可少的工作

常用技术

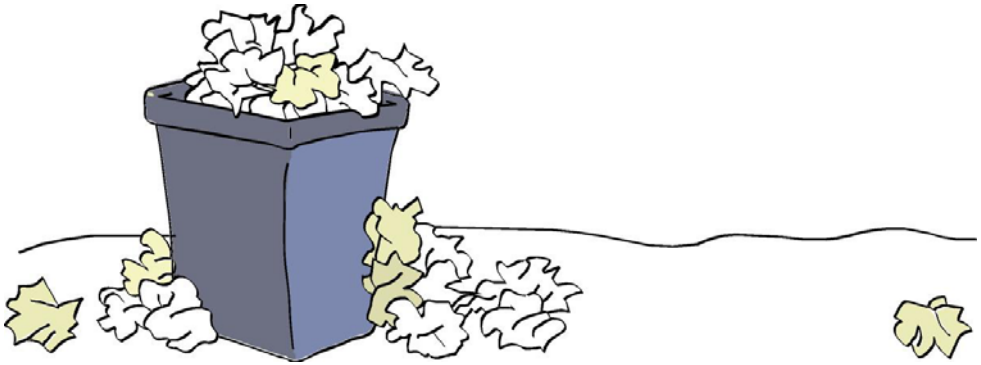
- 抽象数据类型
- 类的设计和使用
- 软件复用
- 契约编程
- 防御式编程
- 异常处理
- 配置式编程
- 基于状态转移和基于表的软件设计
- 基于语法分析的设计
- 单元测试
- 重构

常用软件隐喻（Metaphors）

- 隐喻可以帮助开发人员借助类比的力量加深对编程的理解
  - 计算机科学和软件工程领域的隐喻有哪些？
- 常见的软件隐喻
  - 软件书写：写代码（Writing Code）
  - 软件播种：生成系统（Growing a System）
  - 软件珍珠培植法：系统积累（System Accretion）
  - 软件创建：建造软件（building software）
  - 实用软件技术：智能工具箱（The Intellectual Toolbox）
  - 复合隐喻（Combing Metaphors）

隐喻

- Science
- Art

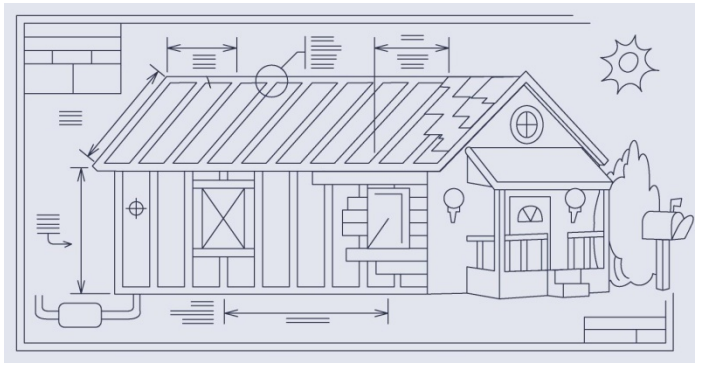


writing a casual letter

Building

Growing a System

System Accretion



软件构造的前期准备工作

- 学习内容
  - 什么是软件构造的前期准备工作
  - 前期准备工作的重要意义
  - 软件构造的前期准备工作有哪些
- 学习目标
  - 明确进入软件构造阶段的先决条件(即边界)
  - 掌握软件构造前期准备工作的具体内容
  - 能够灵活使用相应的Checklist进行核对

- 软件构造的前期准备工作是复查性的活动
  - 明确前导性的工作结果有哪些
  - 重新审视前导性工作是否完成
  - 明确前导性工作完成的程度如何
- 准备工作的中心目标是降低风险
  - 糟糕的需求分析
  - 糟糕的项目计划

前期准备工作的意义

- 开始软件构造之前进行准备工作的论据
  - 从逻辑性的角度
  - 从与其它事物类比的角度
  - 从以往数据的角度

在开始实现一个系统之前，你必须十分清楚：

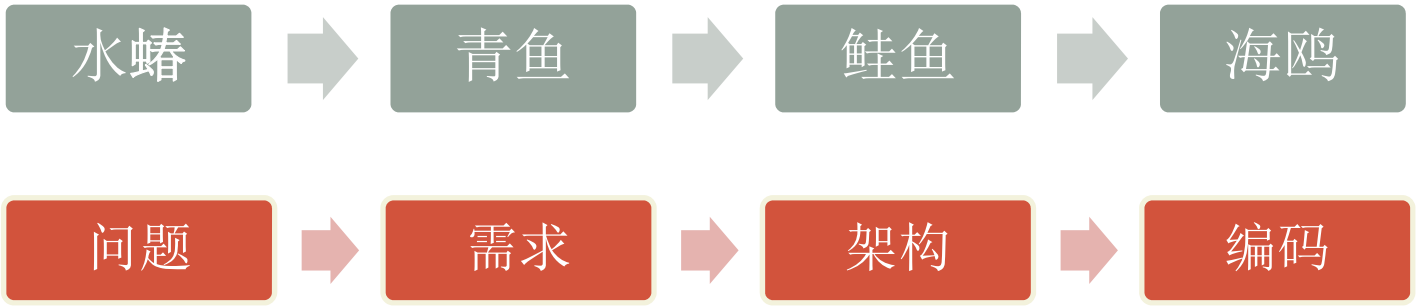
这个系统应该做什么，以及它该如何做到这些！

- 进行有效编程的要领之一：准备工作很重要。
  - 从管理的角度来看，做好计划意味着可以确定项目所需的时间、人数和计算机台数。
  - 从技术角度来看，做好计划意味着可以明确你要构造的到底是什么。

- 从逻辑的角度讲，准备工作的时间“物有所值”
  - 中国有句古话：“磨刀不误砍柴工”

与其它事物类比

- 与建筑的构造类比
  - 在着手构造一座大厦之前对设计图纸的审核工作是必不可少的
- 软件食物链（注意：这就是一个隐喻）



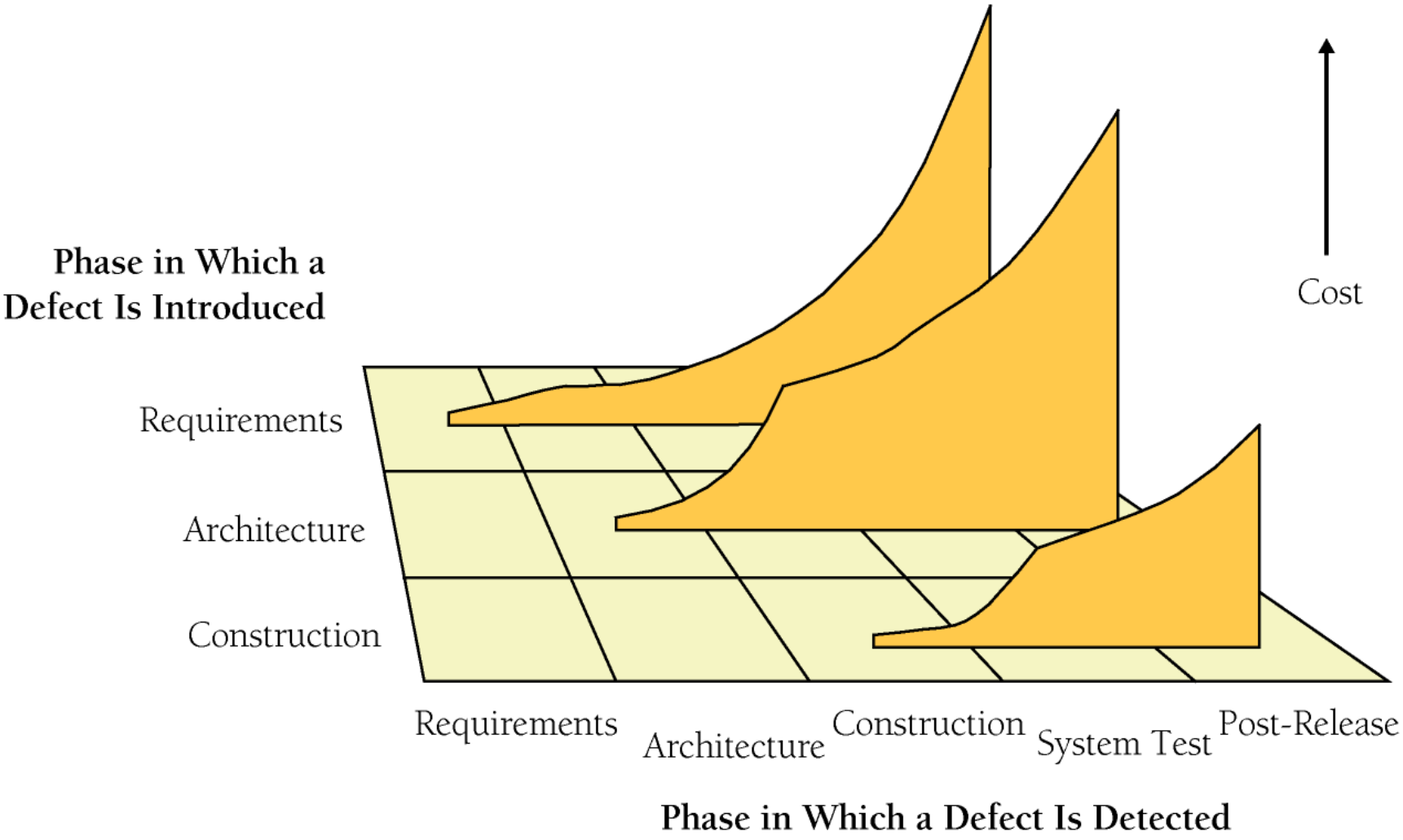
从以往数据的角度

- 发现错误的时间要尽可能接近引入该错误的时间
  - 缺陷在软件食物链中的时间越长，造成的损害越严重

惠普、IBM、休斯顿飞机公司以及TRW等的经验：

在构造活动开始之前清除一个错误的返工成本仅为开发过程最后阶段做同样事情的**1/10**或**1/100**！

修复缺陷的成本随着“从引入缺陷到检测该缺陷之间的时间”变长而急剧增加



前期准备工作的内容

- 前期准备工作并非一成不变
  - 构造活动的准备工作要根据项目特点调整
  - 具体细节随项目的不同会有很大变化
- 构造开始前的准备工作有哪些
  - 辨明软件项目的类型
  - 明确问题定义
  - 明确需求规约
  - 明确软件架构的各组成部分

众多的软件项目种类

- 在软件开发领域中有无数种不同的项目
  - 大量的需求收集和描述技术
  - 各种各样的设计方法
  - 数不胜数的编程语言
  - 大量的测试手段

Caper Jones (Software Productivity Research):

他和同事在20年研究和开发中见过不止700编程语言、40种需求收集方法、50种软件设计方法、30种测试方法!

辨明软件项目的类型

- 最常见的三种类型
  - 商业系统（Business Systems）
  - 使命攸关的系统（Mission-Critical Systems）
  - 性命攸关的系统（Embedded Life-Critical Systems）
- 不同类型的项目倾向于不同的开发方式
  - 商业系统倾向于使用高度迭代的开发方式
  - 性命攸关的系统往往要求更加序列式的方法

	软件种类		
	商业系统	使命攸关的系统	性命攸关的嵌入式系统
典型应用	Internet 站点 Intranet 站点 库存管理 游戏 管理信息系统(MIS) 工资系统	嵌入式软件 游戏 Internet 站点 盒装软件 软件工具 Web services	航空软件 嵌入式软件 医疗设备 操作系统 盒装软件
生命周期模型	敏捷开发（极限编程、Scrum、time-box 开发等等） 渐进原型（prototyping）	分阶段交付 渐进交付 螺旋型开发	分阶段交付 螺旋型开发 渐进交付

需求	非形式化的需求规格	半形式化的需求规格 随需的需求评审	形式化的需求规格 形式化的需求检查
设计	设计与编码是结合的	架构设计 非形式化的详细设计 随需的设计评审	架构设计 形式化的架构检查 形式化的详细设计 形式化的详细设计检查
构建	结对编程或独立编码 非正式的 check-in 手续或没有 check-in 手续	结对编程或独立编码 非正式的check-in手续 随需代码评审	结对编程或独立编码 正式的check-in手续 正式的代码检查
测试与 QA	开发者测试自己的代码 测试先行开发 很少或没有测试（由单独的测试小组来做）	开发者测试自己的代码 测试先行开发 单独的测试小组	开发者测试自己的代码 测试先行开发 单独的测试小组 单独的 QA 小组

序列式开发和迭代式开发

- 序列式开发
  - 使用瀑布式模型进行开发
  - 适用于需求相对稳定的情况
- 迭代式开发
  - 整个开发工作被组织为一系列的迭代过程。每一次迭代都包括了需求分析、设计、实现与测试。
  - 适用于需求容易变化的情况

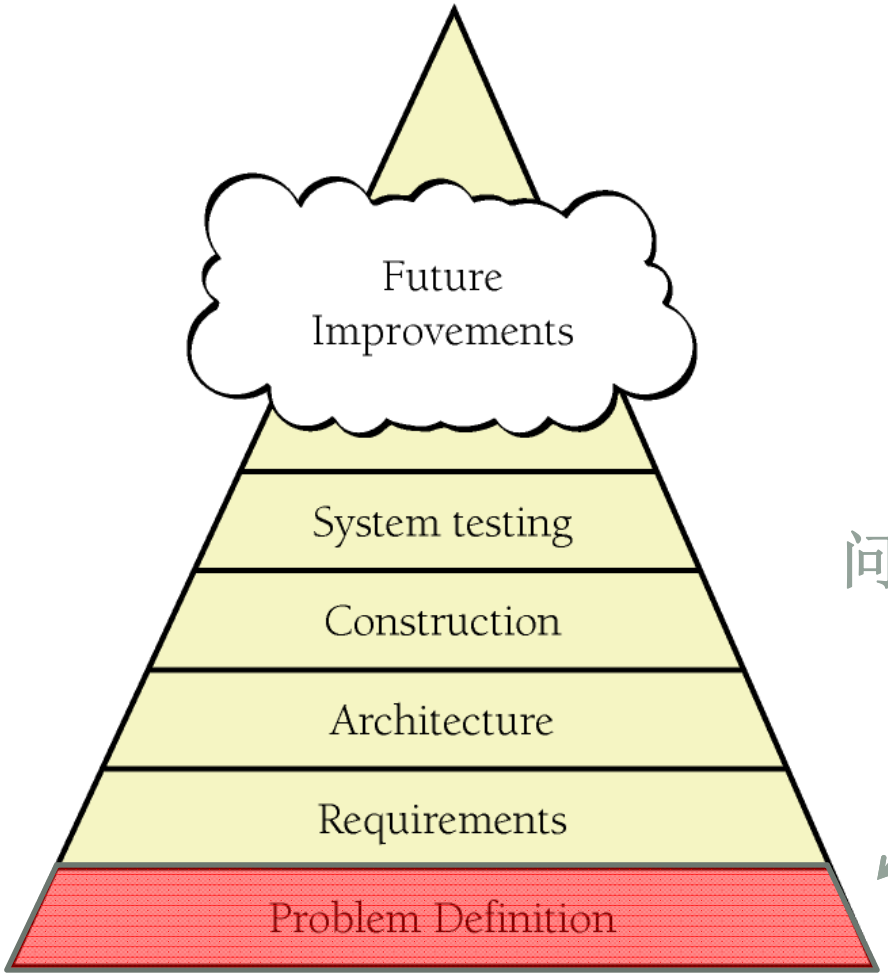
如何选择序列式或迭代式开发方法

- 倾向于序列式开发的情况
  - 需求相当稳定
  - 设计相对简单、容易
  - 后期改变需求、设计和编码的代价非常高
- 倾向于迭代式开发的情况
  - 需求相对不稳定，或需要不断加深理解
  - 设计复杂，具有挑战性
  - 后期改变需求、设计和编码的代价比较低

Check问题定义

- 问题定义（Problem Definition）
  - 问题定义：项目要解决的问题是什么
- 明确的问题定义是第一项先决条件
  - 在开始软件构造之前首先要check是否已经对系统要解决的问题做出了明确的陈述。
- 问题定义也称为
  - 产品设想（Product Vision）
  - 任务陈述（Mission Statement）
  - 产品定义（Product Definition）

问题定义是整个开发的基础



问题定义决定了项目的方向

关于问题定义的注意点

- 问题定义并不涉及任何可能的解决方案
  - 一个好的定义：
    - We can't keep up with orders for the Gigatron
  - 一个不好的定义：
    - We need to optimize our automated data-entry system to keep up with orders for the Gigatron
- 问题定义应该用客户容易理解的语言编写
- 问题定义应该从客户的角度编写

Check需求定义

- 需求定义（Requirement Definition）
  - 需求定义详细规定了软件系统应该做什么
- 明确的需求定义是另一项先决条件
  - 在开始软件构造之前要check是否已经充分详尽地描述了系统所要做的事情。
- 需求定义也称为
  - 软件需求（Software Requirement）
  - 功能规约（Function Specification）

核对表1：功能需求

- 功能需求
  - 规定开发人员必须在产品中实现的软件功能，用户利用这些功能来完成任务，满足业务需求。
- 是否详细定义了系统的全部输入
  - 输入的来源、精度、取值范围、出现频率等
- 是否详细定义了系统的全部输出
  - 输入的目的地、精度、取值范围、出现频率等
- 是否详细定义了所有输出格式
  - Web页面、报表、磁盘文件

核对表1：功能需求(续)

- 是否详细定义了所有硬件及软件的外部接口
  - 外部接口指该软件实体与外部硬件对接以及其它软件交互部分的接口
- 是否详细定义了全部外部通信接口
  - 握手协议、纠错协议、通信协议等
- 是否列出了用户想要做的全部事情
- 是否详细定义了每个任务所用及得到的数据

核对表2：非功能需求(QoS需求)

- 非功能需求
  - 指软件产品在功能以外的服务质量（Quality of Service，QoS）方面的需求
- 是否为必要操作定义了期望的相应时间
  - 在实时系统中是响应时间必要的
- 是否详细定义了其它与计时有关的考虑
  - 处理时间、数据传输率、系统吞吐量
- 是否详细定义了安全级别

核对表2：非功能需求(QoS需求)(续)

- 是否详细定义了可靠性
  - 包括软件失灵的后果、需要保护的重要信息、错误检测以及恢复策略
- 是否详细定义了内存、硬盘的使用要求
- 是否详细定义了系统的可维护性
  - 功能的变更、操作环境的变更和接口的变更
- 是否包括对“成功”、“失败”的定义

需求其它方面的核对表

- 需求的质量
  - 需求定义自身所表现出来的质量
  - 需求的清晰性、一致性、可测试性等
- 需求的完备性
  - 需求是否可以覆盖问题定义所描述的信息
  - 是否不包含不能实现的需求

稳定需求的神话

- 需求变更的主要来源
  - 开发过程帮助客户更好的理解自己的需求

- 认识到需求变更的必要性，并采取措施使得变更的负面影响最小化

IBM等研究发现：

平均项目开发过程中，需求会有**25%**的变化；

需求变更导致的返工占到返工总量的**75%**到**85%**。

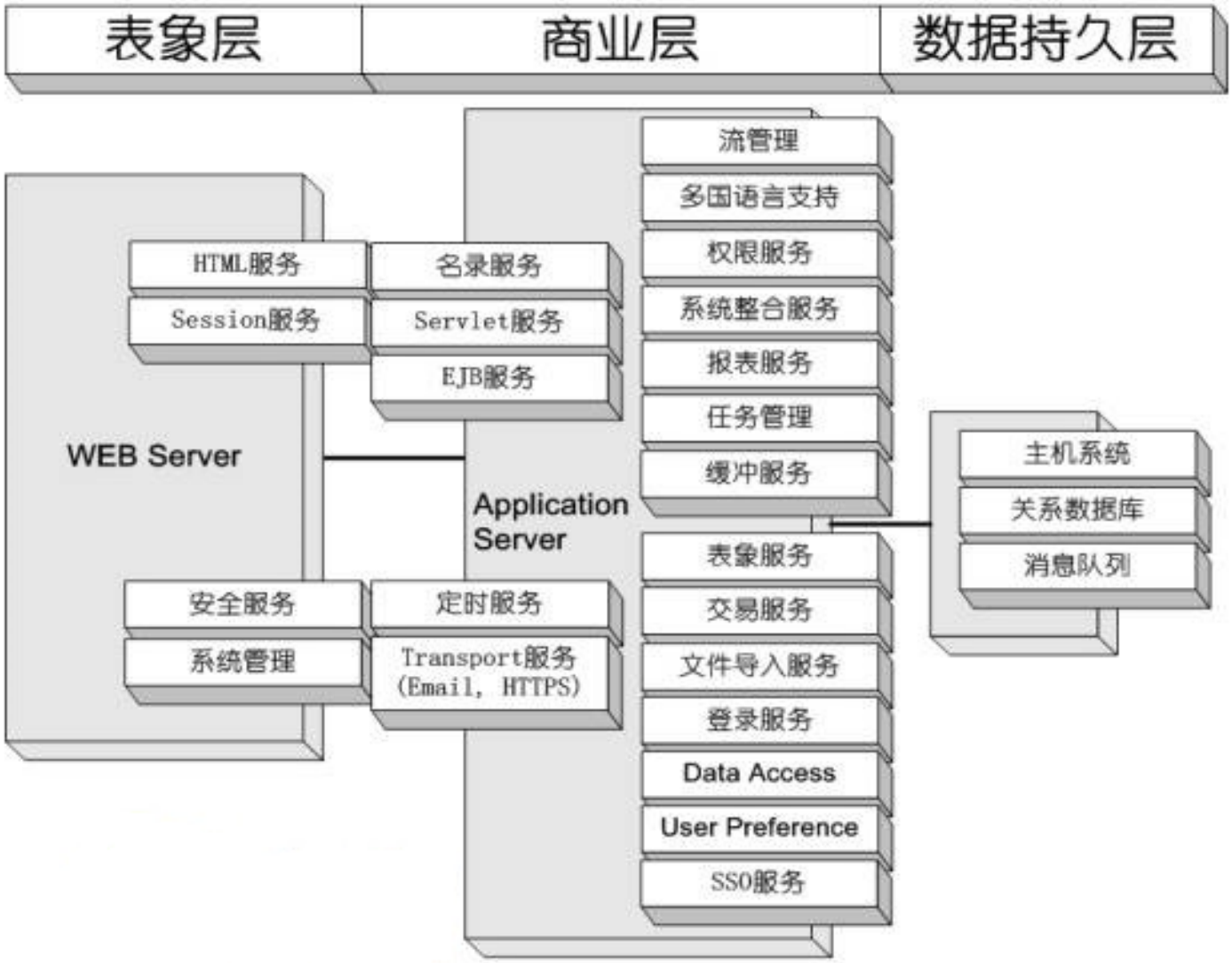
Check软件架构

- 软件架构（Software Architecture）
  - 软件设计的高层部分，用于支撑更加细节的设计的框架
- 明确的架构设计也是构造的先决条件
  - 在开始软件构造之前要check是否已经在整个系统范围内定义了相应的框架结构。
- 架构也称为
  - 系统架构（System Architecture）
  - 顶层设计（Top-level Design）

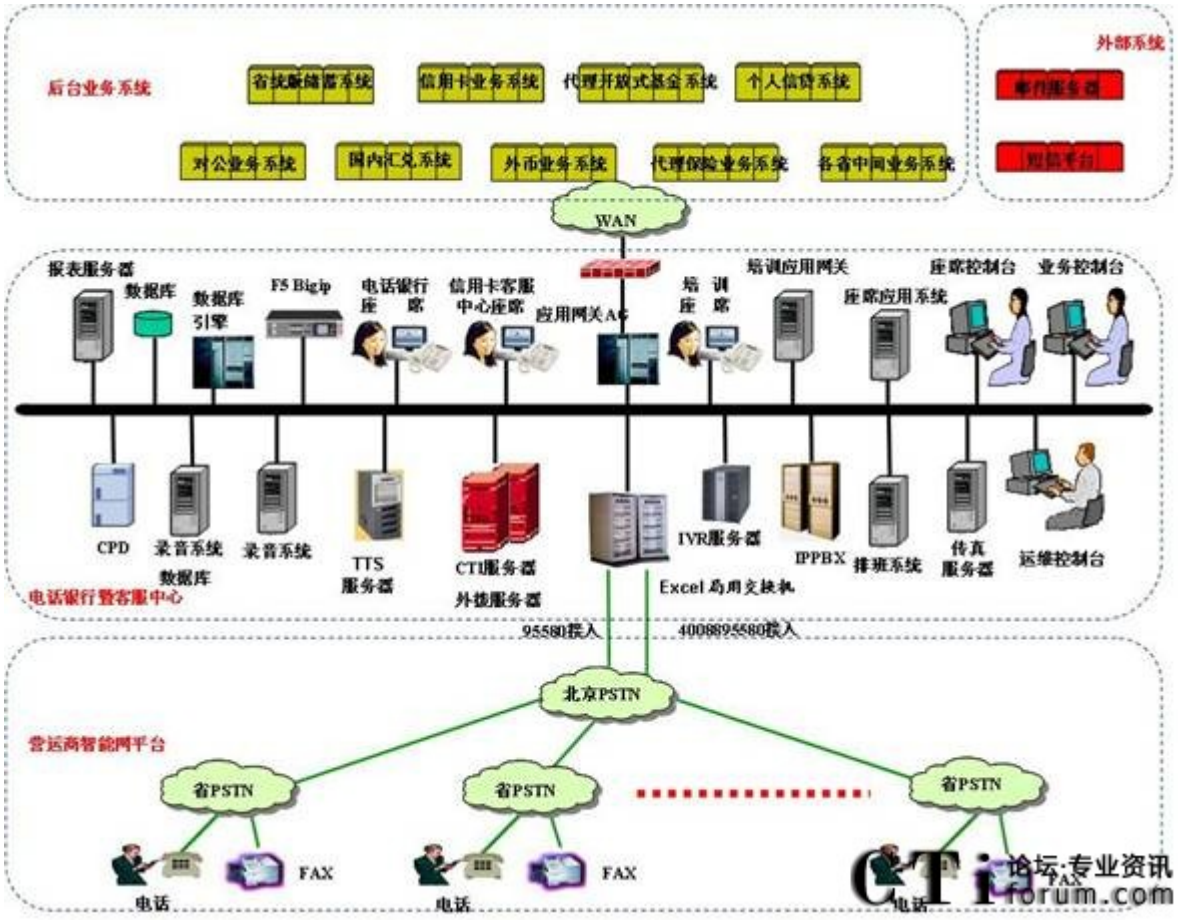
软件架构

- 软件架构是构建计算机软件实践的基础
  - 软件架构定义了一张用于描述整个系统各个方面的草图
  - 是一个软件系统从整体到部分的最高层次的划分
  - 建造一个系统所作出的最高层次的、以后难以更改的，商业的和技术的决定
- 软件架构的两个要素：
  - 元件划分和设计决定

一个逻辑架构的例子


表象层
商业层
数据持久层
流管理
多国语言支持
权限服务
系统整合服务
报表服务
任务管理
缓冲服务
HTML服务
名录服务
Session服务
Servlet服务
EJB服务
WEB Server
Application Server
主机系统
关系数据库
消息队列
表象服务
交易服务
文件导入服务
登录服务
Data Access
User Preference
SSO服务
安全服务
定时服务
系统管理
Transport服务 (Email, HTTPS)

一个物理架构的例子


后台业务系统
外部系统
WAN
报表服务器
数据库
数据库引擎
F5 Big-ip
电话银行座席
信用卡客服中心座席
应用网关AG
培训座席
培训应用网关
座席应用系统
座席控制台
业务控制台
CPD
录音系统
数据库
录音系统
TTS服务器
CTI服务器
外拨服务器
IVR服务器
Excel 局用交换机
IPPBX
排班系统
传真服务器
运维控制台
电话银行暨客服中心
95580接入
4008895580接入
北京PSTN
省PSTN
省PSTN
省PSTN
电话
FAX
电话
FAX
电话
FAX
论坛·专业资讯
forum.com

花费在前期准备上的时间

- 依项目的需要而变化
  - 在完成需求之后，估计余下部分需要的时间是明智的！

一般说来：

一个运作良好的项目会在需求、架构以及其他前期计划方面投入**10%~20%**的工作量和**20%~30%**的时间（不包括详细设计）

关键的构造决策

- 选择适当的编程语言
  - 编程语言会影响程序员的思维
- 制定编程规范来使构造工作更为专注
- Programming into a language
  - 先确定需表达的思想，然后考虑如何用语言提供的工具来表达
- 选择主要构造方法
  - 编码、团队工作、质量保证、工具

Sub-Part I:

- 构造中的设计
  - 设计中的挑战
  - 关键设计概念
  - 启发式方法
  - 设计实践

设计和构造

- 设计是软件构造的一个活动吗？
  - 设计是把需求分析和编码调试连在一起的活动
  - 小规模项目中，很多活动都属于构造
  - 大项目中，架构大多只解决系统层问题
- 哪些活动属于设计？
  - 用伪码写出一个类接口
  - 编码前画出类的关系图
  - 关于设计模式的选取
- 设计是一个庞大的话题！

设计中的挑战

- 设计是险恶的（wicked）问题
  - 只有通过解决或部分解决才能明确
- 设计是了无章法的过程
  - 会犯很多错误，难以判断何时设计“足够好”
- 设计需要取舍和调整顺序
- 设计会受到很多限制
- 设计是不确定的
- 设计是一个启发式过程

管理复杂度

- 是软件的首要技术使命，是软件开发中最为重要的技术话题
  - Fred Brooks 《没有银弹：软件工程中本质性与偶然性》，1987
- 项目失败的主要原因
  - 需求、规划和管理
  - 技术因素：复杂度失控

- “没有谁的的大脑能容下一个现代的计算机程序”——Dijkstra
  - 把任何人在同一时间需要处理的本质(essential)复杂度的量减到最少；
  - 不让偶然(accidental)的复杂度无谓地快速增长

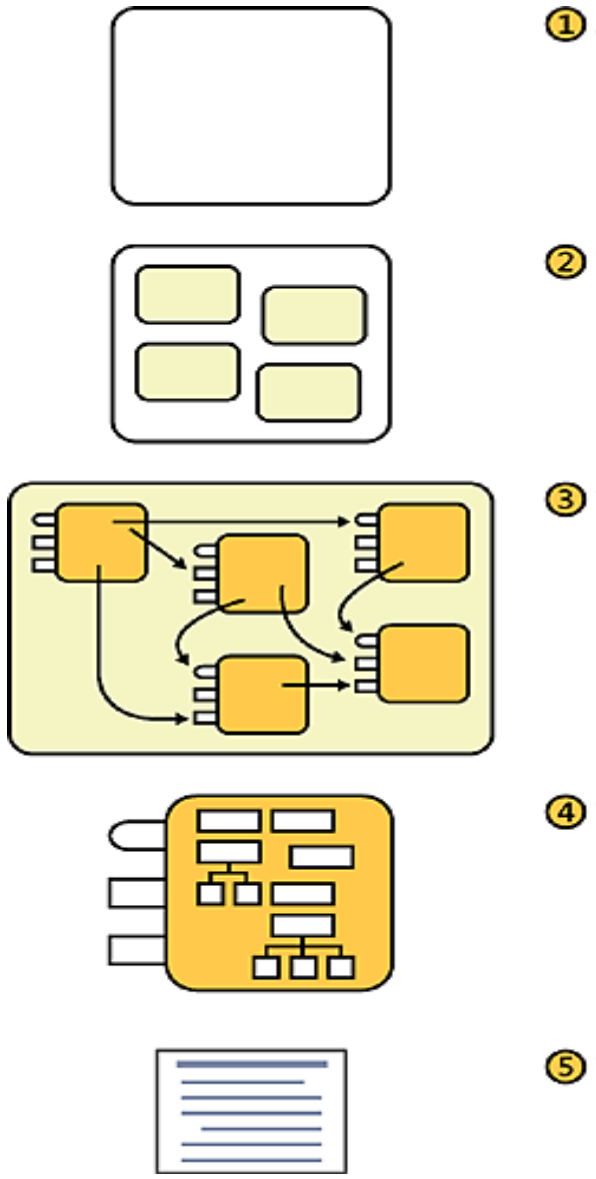
管理复杂度是软件开发中最为重要的技术目标！

高质量设计的理想特征

- 最小复杂度：简单、容易理解
- 易于维护
- 松散耦合
- 可扩展性
- 可重用性
- 高扇入、低扇出
- 可移植性
- 精简性：没有多余代码
- 层次性：能在任意层次上观察系统
- 标准技术

设计的层次

- 软件系统
- 分解为子系统或包
- 分解为类
- 分解为类中的数据和子程序
- 子程序内部



子系统或包层

- 定义清楚允许各子系统如何使用其他子系统
  - 即不同子系统之间的通信规则

通过限制子系统间通信，使每个子系统更有意义
最简单：一个子系统去调用另一个子系统的子程序
稍复杂：一个子系统包含另一个子系统中的类
最复杂：一个子系统中的类继承自另一个子系统中的类

常见的子系统

- 业务规则
  - 和法律、规则、政策等相关的内容，如学分绩计算
- 用户界面
- 数据库访问
- 对系统的依赖性
  - 和运行平台相关内容

类层次

- 对子系统进行分解，确保分解出的细节能够用单个类实现，并明确类的接口

- 类与对象
  - 类：在程序源码中存在的静态事物
  - 对象：程序运行期间实际存在的具体实体
  - 如Student类，和LiLei,HanMeimei等对象

子程序层

- 为每个子程序布置详细功能
  - 类层次定义了对外接口
  - 这里细化出类的私有子程序
    - 也可能会对接口进行修改

- 子程序内部设计
  - 通常由程序员个人完成
  - 编写伪码、选择算法、组织代码块、编写代码
  - 包括子程序内部的设计

构造的启发式方法

- 软件设计是非确定性的
- 启发式方法就是不断“试错”的过程

- 铭记软件的首要技术使命

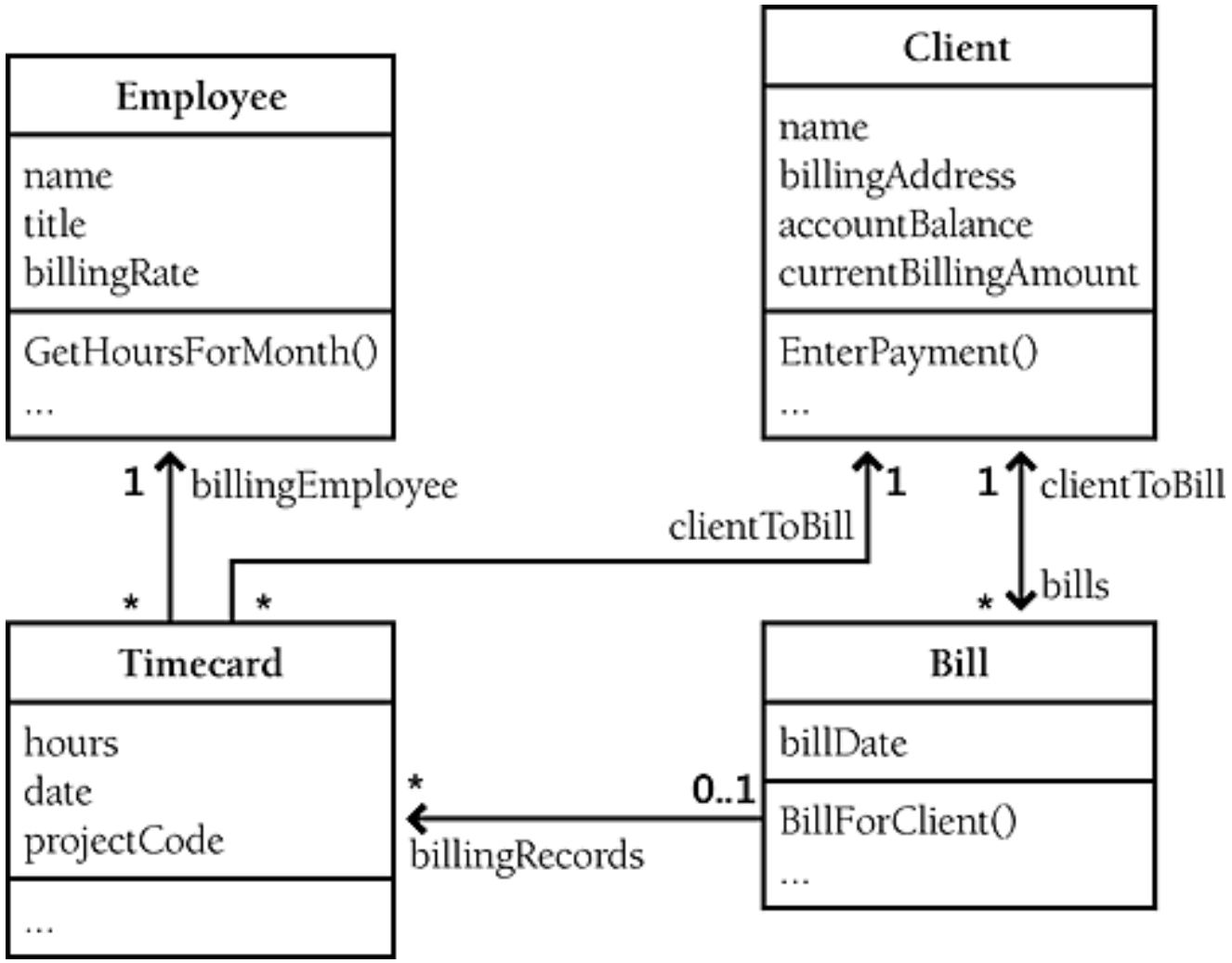
管理复杂度是软件开发中最为重要的技术目标！

启发式设计1：找出现实中的对象

- 软件设计首选且最流行的方法
  - 面向对象

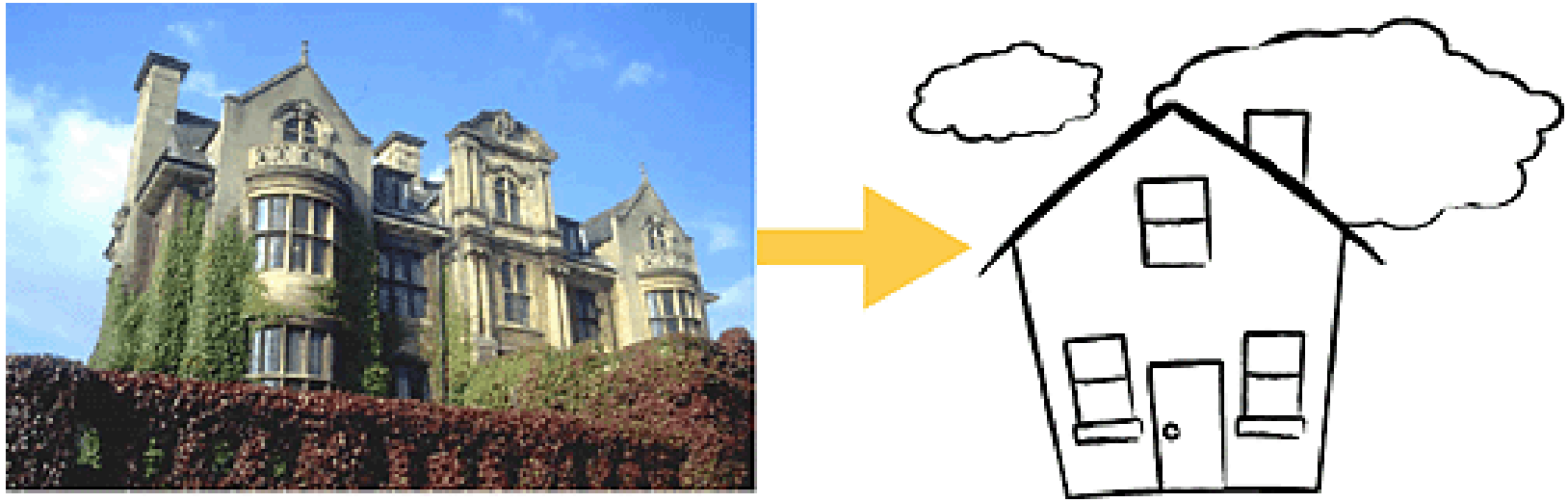
- 步骤：非顺序、可重复
  - 辨识对象及其属性（method & data）
  - 确定可对各个对象的操作
  - 确定对象能对其他对象进行的操作
  - 区分哪些部分对其他对象可见（public or private）
  - 确定公开接口

基于时间的计费系统举例



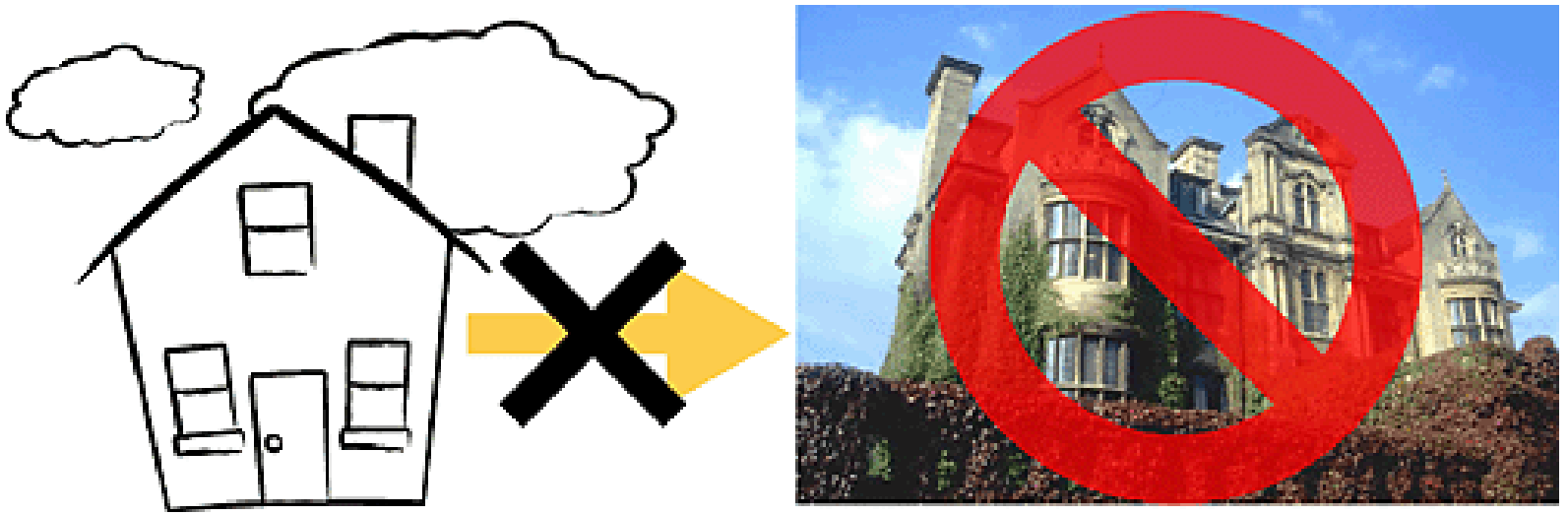
启发式设计2：形成一致的抽象

- 目的：忽略细节，在不同层次处理不同细节
  - 基类、接口都是抽象的例子
  - 优秀程序员要在子程序接口、类接口、包接口层次上进行抽象



启发式设计3：封装实现细节

- 抽象：从较高层次看待对象
- 封装：除此之外，不能看到其他细节
  - 管理复杂度：能看到的就是全部

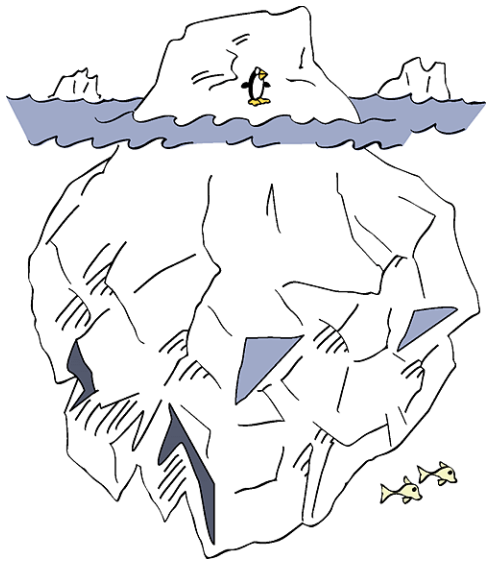


启发式设计4：继承

- 对大同小异的对象，定义对象之间的相同点和不同点
  - 如中学生、大学生
- 继承的好处
  - 简化编程
    - 基本子程序处理基类属性的事项，另外一些子程序处理依赖特定子类的特定操作
  - 面向对象编程中最强大的工具之一，需谨慎使用！

启发式设计4：信息隐藏

- 结构化设计和面向对象设计的基础
  - 结构化设计：黑盒子
  - 面向对象设计：封装、模块化
- “开发人员把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”
- 是减少重复工作的强大技术
  - 对类而言：类的接口应尽可能少地暴露其内部工作机制



信息隐藏举例

- 程序中的所有对象通过名为id的成员变量来保存唯一的ID

- 设计一
  - ID定义为整数
  - 全局变量g_maxId
  - Id = ++g_maxId

问题：
想保留部分ID怎么办？
希望使用非连续ID如何？
怎样确保ID值不会超过预期的最大范围？

改进

- 隐藏创建新ID的方法
  - 优点：修改取值范围只对NewId子程序进行
  - Id = NewId();

Q：如果要将ID修改为字符串呢？

- 隐藏ID类型
  - C++中可使用typedef将ID定义为IdType

隐藏设计决策对于减少“改动所影响的代码量”而言是至关重要的！

隐藏什么？

- 复杂度
  - 只在特别关注的时候去应付它
- 变化源
  - 变化发生时，将影响限制在局部范围
  - 包括复杂的数据类型、文件结构、布尔判断等

- 养成询问“我该隐藏些什么”的习惯！
  - 不要由于惯用某些技术而导致心理障碍
  - 信息隐藏同样有助于设计类的公开接口

启发式设计5：找出容易变化的区域

- 目标：把不稳定的区域隔离出来，把变化所带来的影响限制在一个子程序、类或者包的内部

进行分离和隔离

- 业务规则
- 对硬件的依赖性
- 输入和输出
- 非标准的语言特性
- 困难的设计和构造区域
- 状态变量
  - 使用枚举类型取代布尔类型
  - 使用访问器子程序
- 数据量的限制
  - 通过具名常量隐藏细节

启发式设计6：保持松散耦合

- 耦合度表示类之间或子程序之间关系的紧密程度
  - 一个模块越容易被其他模块所调用，它们的耦合关系就越松散
- 目标：耦合度小就是美

- 耦合种类
  - 简单数据参数耦合：模块间只传递简单数据类型
  - 简单对象耦合：一个模块实例化一个对象
  - 对象参数耦合：对象1要求对象2给它对象3
  - 危险的语义耦合！

语义耦合举例

- 一个模块不仅使用了另一个模块的语法元素，而且还使用了有关那个模块内部工作细节的语义知识
  - 模块1向模块2传递控制标志，告诉模块2要做什么
  - 模块2在模块1修改了某个全局数据，模块1使用该全局数据
  - 模块1的接口要求它的Init[illegible]在Routine()之前被调用。模块2实例化模块1后[illegible]outine()
  - 模块1把BaseObject传给[illegible]ule2。Module2将其转换为DerivedObject

危险！

如果对一个模块的使用要求你同时关注好几件事——内部工作细节、对全局数据的修改、不确定的功能点——那么就失去了抽象的能力，模块所具有的管理复杂度的能力也削弱或完全丧失了。

其他启发式方法

- 应用设计模式
- 高内聚
- 为测试而设计
- 创建中央控制点
- 保持设计的模块化
- ……