        - 原则
        - 实践操作
    - 水晶法Crystal
    - 并列争球Scrum
    - 自适应ASD
- 第三章
    - 风险risk
        - 概念
        - 风险管理
            - 风险评价
                - 风险识别
                - 风险分析
                - 风险优先级分配
            - 风险控制
                - 风险降低
                - 风险管理计划
                - 风险化解
    - 工作分解结构WBS---Work Breakdown Structure
        - 描述活动
    - 关键路径法CPM--Critical Path Method
    - 里程碑
    - 甘特图
    - 沟通数量关系
    - 可交付成果
- 第四章
    - 概念
    - 需求类型requirement type
        - 功能需求functional requirement-------功能
        - 质量需求quality requirement || 非功能需求 nonfunctional requirement-----性能
        - 设计约束design constraint----物理环境
        - 过程约束process constraint----资源
    - 需求文档
        - 需求定义
        - 需求规格说明
        - 需求性质
    - 原型化需求
        - 快速原型化

        - 抛弃型原型
        - 演化型原型
    - 确认和验证
        - 需求确认validation---确保构建了正确的系统
        - 需求验证verification---确保正确地构建系统
    - 测量需求
- 第五章
    - 设计阶段
        - 概要设计阶段
        - 详细设计阶段
    - 体系结构风格
        - 管道和过滤器pipe and filter
        - 客户-服务器 C/S
        - 对等网络peer to peer
        - 发布-订阅publish-subscribe
        - 信息库
        - 分层
        - 组合体系结构风格
    - 分解方式kinds of decomposition
        - 功能性分解functional decomposition-----把功能或需求分解成模块
        - 面相特征的设计-----功能性分解的一种，但为各个模块指定了各自的特征
        - 面向数据的分解-----如何将数据分解成模块
        - 面向进程的分解-----将系统分解成一系列并发的进程
        - 面向事件的分解-----关注系统必须处理的事件，并将事件的责任分配给不同的模块
        - 面向对象的设计-----将对象分配给模块
    - 体系结构视图architecture
        - 分解视图：传统的系统分解视图将系统描述为若干个可编程的单元
        - 依赖视图：展示了软件之间的依赖关系
        - 泛化视图：向我们展示了一个软件单元是否是另一个单元的泛化或者特化
        - 执行视图：设计人员所绘制的传统的方框-箭头图，在考虑到构件和连接器的情况下，展示了系统运行时的结构(每个构建都是不同于其他构件的实体,且可能拥有自己的程序栈)(连接器是构件之间的通信机制)
        - 实现视图：在代码单元和源文件之间建立映射
        - 部署视图：在运行时实体和计算机资源之间建立映射
        - 工作分配视图：将系统分解成可以分配给各项目团队的工作任务
    - 质量属性

    - 文档化体系结构
    - 体系结构评审
- 第六章
    - 耦合coupled
    - 内聚cohesion
    - 接口interface
    - 信息隐藏information hidden
    - 编程原则
    - 增量开发
        - 扇入fan-in:一个模块被多个模块调用
        - 扇出fan-out:一个模块调用多个模块
- 第七章
    - 可读性
    - 注释类型
- 第八章
    - 故障和失效
        - 故障的类型
            - 算法故障algorithmic fault
            - 语法故障syntax fault：
            - 计算故障computation fault、精度故障precision fault
            - 文档故障document fault
            - 压力故障stress fault、过载故障overload fault
            - 能力故障capacity fault、边界故障boundary fault
            - 吞吐量故障throughput fault、性能故障performance fault
            - 恢复故障recovery fault
            - 硬件和系统软件故障hardware and system software fault
            - 标准和过程故障standards and procedure fault
    - 测试过程
    - 单元测试
        - 覆盖
            - 黑盒测试
            - 白盒测试
                - 语句覆盖:每条语句至少执行一次
                - 判定覆盖:每个判定的真和假都要被执行至少一次
                - 条件覆盖:每个判断中的每个条件的可能取值至少满足一次
                - 判定条件覆盖:所有条件可能取值至少执行一次，同时，所有判断的可能结果至少执行一次

            - 条件组合覆盖:每个判断的所有可能的条件取值组合都至少出现一次
            - 分支测试:对代码的判定点,每条分支至少选择一次
            - 路径测试:每条不同的路径在某个测试中至少执行一次
            - 定义使用的路径测试:从每个变量的定义到该定义的使用的每一条路径,都在某个测试中得到执行
            - 所有使用的测试:测试集至少包含从每一个变量的定义到通过定义可到达的每个使用的路径
            - 所有谓词使用/部分计算使用的测试对每个变量以及每个定义,测试至少包含从定义到谓词使用的一条路径
            - 所有计算使用/部分谓词使用的测试:对每个变量以及每个定义,测试至少包含从定义到每个计算使用的一条路径
        - 环路复杂度
    - 集成测试interaction testing
        - 自顶向下Top-Down
        - 自底向上Bottom-up
        - 三明治
        - 大爆炸
- 第九章
    - 回归测试regression test
        - 举例
    - 系统测试过程
    - 性能测试performance testing
        - 目的与职责
        - 类型
    - 测试团队
        - 专业测试人员:组织并运行测试,从测试开始阶段,随着项目的进展,到设计测试计划和测试用例.
        - 非专业测试人员
    - 可靠性.可用性.可维护性
        - 定义
        - 失效数据
        - 测量
        - 增长
        - 预测
        - 重要性
    - 验收测试种类
        - 目的

第一章

什么是软件工程 What is software engineerying

软件的质量 What is good software

产品的质量:可以从用户和开发者两个视角

过程的质量:开发过程中建模，改进开发过程，积累经验

商业背景下的质量:商品价值

人员Who does software engineering

客户customer：为将要开发的软件系统支付费用的公司、组织或个人

用户user：将实际应用系统的人

开发人员developer：为客户构建软件系统的公司、组织或人

软件开发活动software develop activities

需求分析和定义 requirements analysis and definition

系统设计 system design

程序设计 program design

编写程序==程序实现 writing the programss == program implementation

单元测试 unit testing

集成测试 integration testing

系统测试 system testing

系统交付 system delivery

维护 maintenance

开发团队members of the development team

分析员analyst：需求分析和定义、系统设计、维护

设计人员designer：系统设计、程序设计、维护

程序员programmer：程序设计、程序实现、单元测试、维护

测试员tester：单元测试、集成测试、系统测试、维护

培训人员trainer：系统交付、维护

第二章

过程含义The meaning of process

过程process：一组有序的任务

生命周期life cycle：产品构建的过程，从概念到实现、交付、使用和维护

强制活动具有一致性和一定的结构

生命周期模型Life cycle Model

瀑布模型 Waterfall model

概述

1. 必须等前一阶段的工作完成后，才能开始后一阶段的工作
2. 前一阶段的输出文档就是后一阶段的输入文档，因此只有前一阶段的输出文档正确，后一阶段的工作才能获得正确的结果

优点

- 为项目提供了按阶段划分的检查点
- 当前一阶段完成后，您只需要去关注后续阶段
- 可在迭代模型中应用瀑布模型

缺点

- 不适合需求模糊或需求经常变动的系统
- 由于开销的逐步升级问题，它不希望存在早期阶段的反馈
- 在一个系统完成以前，它无法预测一个新系统引入一个机构的影响
- 在用户可能需要较长等待时间来获得一个可供使用的系统，也许会给用户的信任程度带来影响和打击
- 最终产品往往反映用户的初始需求而不是最终需求

适用

1. 需求在规划和设计阶段就已确定，且项目开发周期内需求没有或极少变化，对需求变更进行严格控制
2. 稳定的低风险项目（对目标、环境非常熟悉），规模小实现简单易受控的项目；
3. 合同式的合作方式，严格按照说明执行，客户需求明确且不参与软件实现过程

V模型 V model

概述

通过开发和测试同时进行的方式来缩短开发周期，提高开发效率

优点

相对于瀑布模型，V模型测试能够尽早的进入到开发阶段

缺点

虽然测试尽早的进入到开发阶段，但是真正进行软件测试是在编码之后，这样忽视了测试对需求分析，系统设计的验证，时间效率上也大打折扣

适用

一些传统信息系统应用的开发

原型化模型 Prototype model

概述

通过构造原型，即一个软件系统的最初版本，用于验证概念、适用设计选型、发现更多的问题和可能的解决方法

优点

有助于启发和验证系统需求 增加用户与开发人员的交流 用户在项目开发中占主导作用 满足用户的动态需求 降低开发风险

缺点

1. 原型开发会忽略掉非功能性要求 ，如性能、安全性、可靠性等；
2. 开发过程的快速更改意味着没有文档，唯一的设计描述是原型的代码，这不利于长期的维护

适用

1. 需求模糊。
2. 开发人员对算法效率、操作系统的兼容性，人机交互形式等情况不明确

螺旋模型

概述

螺旋模型（Spiral Model）采用一种周期性的方法来进行系统开发。这会导致开发出众多的中间版本。使用它，项目经理在早期就能够为客户实证某些概念。该模型是快速原型法，以进化的开发方式为中心，在每个项目阶段使用瀑布模型法。这种模型的每一个周期都包括需求定义、风险分析、工程实现和评审4个阶段，由这4个阶段进行迭代

最大的特点在于引入了其他模型不具备的风险分析，使软件在无法排除重大风险时有机会停止。 螺旋模型更适合大型的昂贵的系统级的软件应用

优点

1. 设计上的灵活性,可以在项目的各个阶段进行变更
2. 以小的分段来构建大型系统,使成本计算变得简单容易
3. 客户始终参与每个阶段的开发,保证了项目不偏离正确方向以及项目的可控性

缺点

1. 很难让用户确信这种演化方法的结果是可以控制的
2. 建设周期长，而软件技术发展比较快，所以经常出现软件开发完毕后，和当前的技术水平有了较大的差距，无法满足当前用户需求

适用

庞大、复杂并具有高风险的系统

增量开发模型incremental development

优点

1. 降低适用需求变更的成本：可以更经济、更容易、更快速的相应变更。
2. 尽快得到问题反馈。
3. 交付和部署更快。
4. 用户可以更早的使用软件并创造价值。

缺点

1. 过程不可见，不方便管理。
2. 缺乏整体规划，导致功能堆砌。
3. 越往后变更越困难，成本逐渐上升。

适用

1. 小型系统或功能；
2. 需要快速上线的电商等类型系统。
3. 开发团队人力资源、时间不充足的情况

迭代开发iterative development

优点

1. 在需求分析阶段就给出了相对完成的架构设计方案，便于后面迭代的扩展和完善；
2. 第一阶段核心功能交互用户后，可以及早获取反馈结果，对后期的迭代起到指导作用；
3. 人员分配灵活，前期不用投入很多人力；
4. 在前期能够很好的控制风险，并且解决难度系数较低，影响范围也较小。

缺点

1. 对项目需求明晰度要求很高；
2. 对整个项目周期要求较宽裕，政治任务需排除

极限编程XP

激发开发人员创造性,使管理负担最小的一组技术

原则

交流:客户与开发人员之间持续交换看法

简单性:鼓励开发人员选择最简单的设计或实现来处理客户需求

勇气:尽早和经常交付功能的承诺

反馈:相互提供反馈,包含反馈循环

实践操作

规则游戏,小的发布,隐喻,简单设计,首先编写测试,重构,对编程,集体所有权,持续集成,可以忍受的步伐,在现场的客户,代码标准

水晶法Crystal

每个不同的项目都需要一套不同的策略,约定和方法论

并列争球Scrum

使用迭代方法,每30天一次的迭代成为"冲刺sprint"

自适应ASD

围绕着构造的构件来组织并实现特征

第三章

风险risk

概念

具有负面后果,人们不希望发生的事件

- 风险影响risk impact:与事件有关的损失
- 风险概率risk probablity:事件发生的可能性
- 风险控制risk control:能够改变结果的程度
- 风险暴露risk exposure:风险影响*风险概率

风险管理

风险评价

风险识别

风险分析

风险优先级分配

风险控制

风险降低

风险管理计划

风险化解

工作分解结构WBS---Work Breakdown Structure

把项目描述为由若干个离散部分构成的集合

描述活动

- 前驱:活动开始之前必须发生的一个事件或一组事件
- 工期:活动完成所需的时间长度
- 截止日期:活动必须完成的日期
- 终点:活动已经结束

关键路径法CPM--Critical Path Method

real time || actual time:估算完成活动必须的时间量

available time:完成活动可用的时间量

slack time || float time:活动可用时间和真实之间之差

里程碑

活动(项目的一部分)的完成，某一特定的时刻

甘特图

对项目的描述,显示在什么地方活动是并行进行的,用颜色或图标来指明完成的程度

沟通数量关系

N(N-1)/2

可交付成果

在某一过程、阶段或项目完成时，必须产出的任何独特并可核实的产 品、成果或服务能力。可交付成果可能是有形的，也可能是无形的

类型

内部可交付

外部可交付

第四章

概念

需求：对期望的行为的表达

需求处理的是对象或实体,他们可能处于的状态,以及用于改变状态或对象特性的可能

规格说明specification:决定我们的软件系统将完成哪些需求

设计阶段design:定制关于如何实现指定行为的计划

需求分析员requirement analyst或系统分析员system analyst

需求类型requirement type

功能需求functional requirement-------功能

根据要求的活动来描述需要的行为

质量需求quality requirement || 非功能需求 nonfunctional requirement-----性能

描述一些软件解决方案必须拥有的质量特性

设计约束design constraint----物理环境

已经做出的设计决策或限制问题解决方案集的设计决策

过程约束process constraint----资源

用于构建系统的技术和资源的限制

需求文档

需求定义

客户想要的每一件事情的完整列表，用客户的术语记录需求

1. 概述系统的总体目的和范围
2. 描述系统开发的背景和理由
3. 记录问题的总体情况后,描述可接受解决方案的基本特性
4. 描述系统运转的环境
5. 概论描述客户提出的解决问题的提议
6. 列出我们对环境行为所做的一切假设

需求规格说明

将需求重新陈述为关于要构建的系统将如何运转的规格说明,从开发人员的角度编写, (按系统的接口编写)提及系统通过其接口可访问的环境实体

1. 文档化接口过程中，详细描述所有的输入和输出

2. 根据输入和输出重新陈述要求的功能
3. 对每个客户的质量需求，设计适配标准，以便最后证明我们的系统满足这些质量要求

需求性质

正确性，一致性，无二义性，完备性，可行性，相关性，可测试性，可跟踪性

原型化需求

快速原型化

为了回答需求的问题而构造软件

抛弃型原型

为了对问题或者提议的解决方案有更多的了解而开发的软件，永远不会作为交付软件的一部分

快速但不考虑质量，结构差，效率低，不进行错误检查

演化型原型

不仅帮助我们回答问题，而且还会演变成最终产品

确认和验证

需求确认validation---确保构建了正确的系统

检查我们的需求定义是否准确地反映了客户端需要

需求验证verification---确保正确地构建系统

测量需求

测量集中于产品、过程、资源

开发人员对需求测评：

1. 设计人员完全理解该需求
2. 需求中某些部分对设计人员来讲是新的
3. 需求中有些部分和设计人员以前设计过的需求有很大不同，但设计人员认为可以开发
4. 不理解需求中的某些部分，并且不能肯定是否能够开发出好的设计
5. 完全不理解需求，无法开发

第五章

设计阶段

概要设计阶段

详细设计阶段

体系结构风格

知道我们如何把问题分解为软件单元以及这些单元彼此应当如何交互；关于完成一般性设计所给予的建议

已建立的、大规模的系统结构模式

管道和过滤器pipe and filter

过滤器：将输入数据转换后得到输出数据

管道：简单的将数据从一个过滤器传输到下一个过滤器的连接

客户-服务器 C/S

服务器：提供服务

客户：通过请求/应答协议访问服务

客户知道向 哪个服务器发送请求，但是服务器却不知道为哪个用户提供服务，只是负责处理请求并回复用户

对等网络peer to peer

每个构件都只执行他自己的进程，并且对其他同级构件， 每个构件本身即是服务器又是客户端。每个构件都有一个接口，不仅指定了该构件所提供的服务，还指定了向其他同级构件发送的请求

发布-订阅publish-subscribe

通过对事件的广播和反应来实现交互，（一个构件对事件感兴趣并订阅，则事件发生时，另一个构件进行发布来通知订阅者）

优点

- 为系统演化和可定制性提供了强力的支持
- 可在其他系统中轻松复用发布-订阅的构件

缺点

- 不宜与测试
- 会减弱系统的可扩展性和可复用性

信息库

分层

组合体系结构风格

分解方式kinds of decomposition

功能性分解**functional decomposition-----**把功能或需求分解成模块

面相特征的设计**-----**功能性分解的一种，但为各个模块指定了各自的特征

面向数据的分解**-----**如何将数据分解成模块

面向进程的分解**-----**将系统分解成一系列并发的进程

面向事件的分解**-----**关注系统必须处理的事件，并将事件的责任分配给不同的模块

面向对象的设计**-----**将对象分配给模块

体系结构视图**architecture**

分解视图：传统的系统分解视图将系统描述为若干个可编程的单元

依赖视图：展示了软件之间的依赖关系

泛化视图：向我们展示了一个软件单元是否是另一个单元的泛化或者特化

执行视图：设计人员所绘制的传统的方框-箭头图，在考虑到构件和连接器的情况下，展示了系统运行时的结构(每个构建都是不同于其他构件的

实体,且可能拥有自己的程序栈)(连接器是构件之间的通信机制)

实现视图：在代码单元和源文件之间建立映射

部署视图：在运行时实体和计算机资源之间建立映射

工作分配视图：将系统分解成可以分配给各项目团队的工作任务

质量属性

- 可修改性：我们的设计能便于修改
- 性能
- 安全性
- 可靠性
- 鲁棒性
- 易使用性
- 商业目标

文档化体系结构

视图间映射

文档化设计合理性

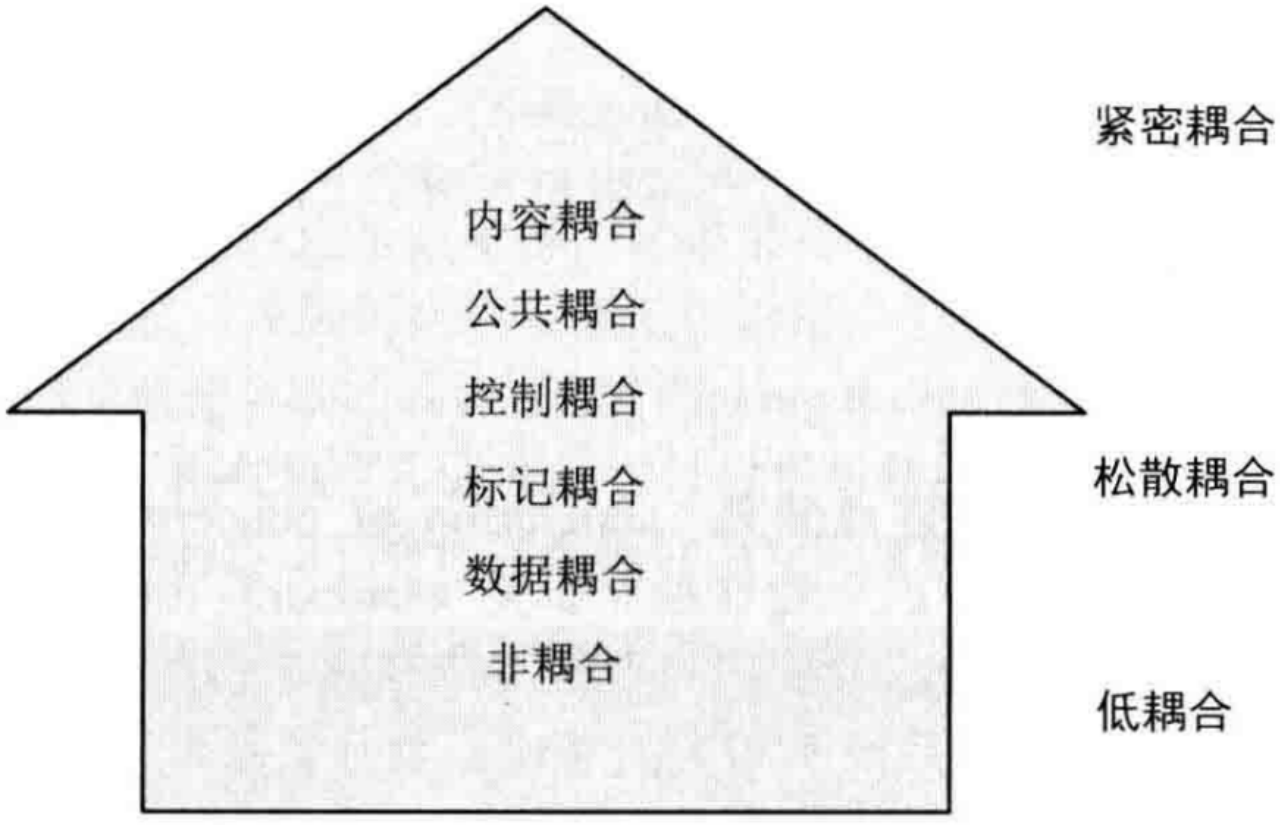
体系结构评审

确认

验证

第六章

耦合coupled



内容耦合content coupled:一个模块实际修改另一个模块,被修改模块完全依赖于修改它的模块

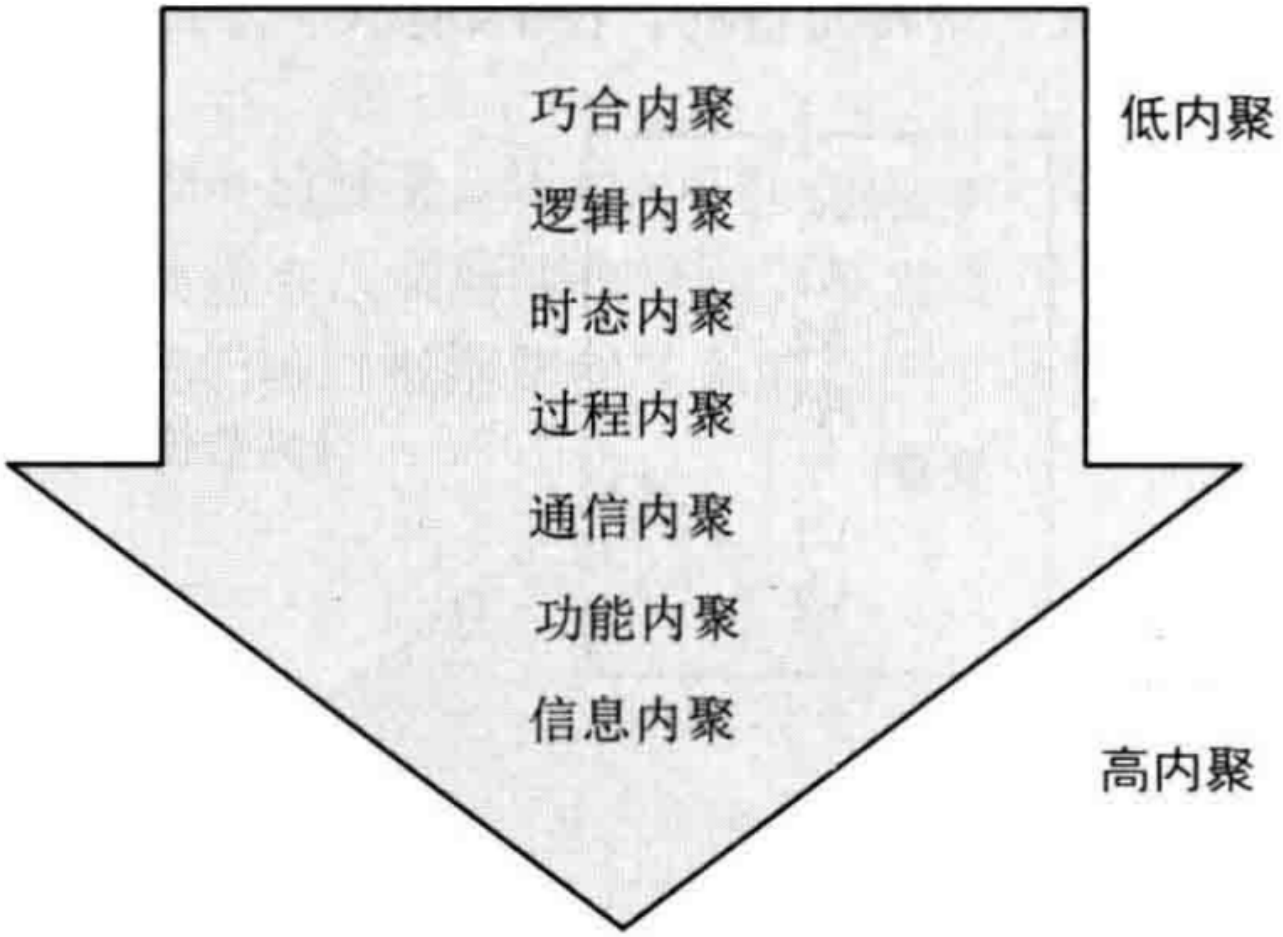
公共耦合common coupled:是哦那个公共数据存储区来访问数据,依赖于公共数据

控制耦合control coupled:某个模块通过传递参数或返回代码控制另一个模块的活动

标记耦合stamp coupled:使用复杂的数据结构从一个模块到另一个模块传递消息----传递数据结构

数据耦合data coupled:模块之间传送到是数据值

内聚cohesion



巧合内聚coincidental cohesion:模块各个部分互不相关

逻辑内聚logical cohesion:模块中各部分通过代码的逻辑结构相关联

时态内聚temporal cohesion:被划分为表示不同执行状态的模块--初始化,读进输入,计算,打印输出和清除

过程内聚procedurally cohesive:模块中的功能组合在一起只是为了确保顺序

通信内聚communicational cohesive:围绕数据集构造

功能内聚functional cohesion:一个模块中包含了所有必需元素,每个处理元素对执行单个功能来说都是必须的

信息内聚informational cohesion:在功能内聚基础上,将其调整为数据抽象化和基于对象的设计

接口interface

接口:为系统其余部分定义了该软件单元提供的服务,以及如何获取这些服务

对其他开发人员封装和隐藏了软件单元的设计和实现细节

信息隐藏information hidden

编程原则

- 单一职责原则(Single Responsibility Principle):有且只有一个原因因此类的变更
- 开闭原则(Open Closed Principle):对扩展开放,对修改封闭
- 里氏替换原则(Liskov Substitution Principle):子类必须完全实现父类的方法,子类可以有自己的属性和方法
- 迪米特法则(Law of Demeter)，又叫“最少知道法则”:一个对象应该对其他对象有最少的了解
- 接口隔离原则(Interface Segregation Principle):客户端不应该依赖它不需要的接口，或者说类间的依赖关系应该建立在最小的接口上
- 依赖倒置原则(Dependence Inversion Principle):类之间不直接发生依赖关系，其依赖关系是通过接口或抽象类产生的

增量开发

扇入fan-in:一个模块被多个模块调用

扇出fan-out:一个模块调用多个模块

第七章

可读性

可读性大于功能性

注释类型

序言性注释

功能性注释

第八章

故障和失效

故障的类型

算法故障algorithmic fault

由于处理步骤中的某些错误，是的对给定的输入，构件的算法或逻辑没有产生适当的输出

包括：分支太早，分支太迟，对错误的条件进行了测试，忘记了初始化变量或忘记了设置循环不变量，忘记针对特定的条件进行测试，对不适合的数据类型变量进行比较

语法故障syntax fault：

计算故障computation fault、精度故障precision fault

一个公式的实现是错误的，或者计算结果没有达到要求的精度

文档故障document fault

文档与程序实际做的事情不一致，导致后期大量产生其他故障

压力故障stress fault、过载故障overload fault

需求规格说明详细说明系统的用户数目，设备数目和通信需要。这些在程序设计中表现为对队列长度、缓冲区大小、表的维度等的限制

当这些数据结构超过了他们规定的能力，就发生了故障

能力故障capacity fault、边界故障boundary fault

系统活动达到指定的极限时，系统性能会变得不可接受

计时故障timing fault、协调故障coordination fault

开发实施系统时，一个关键的考虑因素是几个同时执行的或按仔细定义的顺序执行的进程之间的协调问题，党协调这些事件的代码不适当时会出现故障

难以识别和改正的原因：

1. 对于设计人员和程序员来讲，通常很难遇见所有可能的系统状态
2. 由于计时和处理中涉及太多因素，在一个故障发生后也许不能重现该故障

吞吐量故障throughput fault、性能故障performance fault

系统不能以需求规定的速度执行

恢复故障recovery fault

系统遇到失效时，不能表现的像设计人员希望的或客户要求的那样

硬件和系统软件故障hardware and system software fault

标准和过程故障standards and procedure fault

测试过程

1. 文档评审
2. 单元测试
3. 集成测试
4. 功能测试
5. 性能测试
6. 验收测试
7. 安装测试

单元测试

覆盖

黑盒测试

目的

1. 验证软件产品是否符合需求文档的设计
2. 证实软件产品符合终端用户的需求

优点

1. 从产品功能角度测试，可以最大限度的满足用户的需求
2. 相同的动作可以重复执行，最枯燥的部分可由机器完成
3. 依据测试用例有针对性地寻找问题，定位更加准确，容易生成测试数据
4. 可将测试直接和程序/系统要完成的操作相关联

缺点-不确定性:是否所选择的测试用例将会揭示某个特定的故障

1. 代码得不到测试因为黑盒测试不会去查看系统的内部实现
2. 如果规格说明设计错误，很难发现
3. 测试不能充分地进行
4. 测试结果的准确性取决于测试用例的设计
5. 自动化测试的复用性较低

白盒测试

语句覆盖:每条语句至少执行一次

【优点】：可以很直观地从源代码得到测试用例，无须细分每条判定表达式。

【缺点】：由于这种测试方法仅仅针对程序逻辑中显式存在的语句，但对于隐藏的条件是无法测试的。如在多分支的逻辑运算中无法全面的考虑。语句覆盖是最弱的逻辑覆盖

判定覆盖:每个判定的真和假都要被执行至少一次

【优点】：判定覆盖具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性，无须细分每个判定就可以得到测试用例。

【缺点】：往往大部分的判定语句是由多个逻辑条件组合而成，若仅仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。判定覆盖仍是弱的逻辑覆盖。

条件覆盖:每个判断中的每个条件的可能取值至少满足一次

【优点】：增加了对条件判定情况的测试，增加了测试路径。

【缺点】：条件覆盖不一定包含判定覆盖。例如，我们刚才设计的用例就没有覆盖判断M的Y分支和判断N的N分支。条件覆盖只能保证每个条件至少有一次为真，而不考虑所

有的判定结果

判定条件覆盖:所有条件可能取值至少执行一次，同时，所有判断的可能结果至少执行一次

【优点】：能同时满足判定、条件两种覆盖标准。

【缺点】：判定/条件覆盖准则的缺点是未考虑条件的组合情况

条件组合覆盖:每个判断的所有可能的条件取值组合都至少出现一次

【优点】：条件组合覆盖准则满足判定覆盖、条件覆盖和判定/条件覆盖准则。

【缺点】：线性地增加了测试用例的数量

分支测试:对代码的判定点,每条分支至少选择一次

路径测试:每条不同的路径在某个测试中至少执行一次

【优点】：这种测试方法可以对程序进行彻底的测试，比前面五种的覆盖面都广。

【缺点】：需要设计大量、复杂的测试用例，使得工作量呈指数级增长，不见得把所有的条件组合都覆盖

定义使用的路径测试:从每个变量的定义到该定义的使用的每一条路径,都在某个测试中得到执行

所有使用的测试:测试集至少包含从每一个变量的定义到通过定义可到达的每个使用的路径

所有谓词使用/部分计算使用的测试对每个变量以及每个定义,测试至少包含从定义到谓词使用的一条路径

所有计算使用/部分谓词使用的测试:对每个变量以及每个定义,测试至少包含从定义到每个计算使用的一条路径

环路复杂度

边-节点+2

数圈圈+1

集成测试interaction testing

自顶向下Top-Down

- 目的：从顶层控制（主控模块）开始，采用同设计顺序一样的思路对被测系统进行测试，来验证系统的稳定性
- 定义：自顶向下的集成测试就是按照系统层次结构图，以主程序模块为中心，自上而下按照深度优先或者广度优先策略，对各个模块一边组装一边进行测试

好处：

- 故障定位更容易。
- 有可能获得早期的原型。
- 关键模块按优先级进行测试；可以发现并修复主要的设计缺陷。

缺点：

- 需要许多插桩。
- 较低级别的模块未充分测试

自底向上Bottom-up

- 目的：从依赖性最小的底层模块开始，按照层次结构图，逐层向上集成，验证系统的稳定性
- 定义：自底向上集成是从系统层次结构图的最底层模块开始进行组装和集成测试的方式。对于某一个层次的特定模块，因为它的子模块（包括子模块的所有下属模块）已经组装并测试完成，所以不再需要桩模块。在测试过程中，如果想要从子模块得到信息可以通过直接运行子模块得到。也就是说，在集成测试的过程中只需要开发相应的驱动模块就可以了

好处：

- 故障定位更容易。
- 不像Big-bang方法那样浪费时间等待所有模块的开发

缺点：

- 控制应用程序流程的关键模块（在软件体系结构的最高级别）最后经过测试，并且可能容易出现缺陷。
- 早期的原型是不可能的

三明治

- 定义:三明治集成是一种混合增殖式测试策略，综合了自顶向下和自底向上两种集成方法的优点，因此也属于基于功能分解集成。如果借助图来介绍三明治集成的话，就是在各个子树上真正进行大爆炸集成。桩和驱动器的开发工作都比较小，不过代价是作为大爆炸集成的后果，在一定程度上增加了定位缺陷的难度

优点：

- 它将自顶向下和自底向上的集成方法有机地结合起来，不需要写桩程序因为在测试初自底向上集成已经验证了底层模块的正确性 缺点：
- 在真正集成之前每一个独立的模块没有完全测试过

缺点：

- 在被集成之前，中间层不能尽早得到充分测试

大爆炸

- 一种集成测试方法，其中所有组件或模块都立即集成在一起，然后作为一个单元进行测试

好处：

- 适用于小型系统。

缺点：

- 故障定位很困难
- 考虑到在这种方法中需要测试的接口数量众多，很容易会漏掉一些要测试的接口链接。
- 由于集成测试只能在设计完“所有”模块之后才能开始，因此测试团队在测试阶段的执行时间将减少。
- 由于所有模块都经过一次测试，因此高风险关键模块不会被隔离并优先进行测试。处理用户界面的外围模块也不是隔离的，并且不会进行优先级测试

第九章

回归测试regression test

针对故障修改以及故障发现的指导原则

用于识别在改正当前故障的同时可能引入的新故障

用于新的版本或发布的一种测试，为验证与旧版本或发布版本相比，它是否仍然以同样的方式执行相同的功能

步骤：

1. 插入你的代码
2. 测试被新代码影响的功能
3. 测试上一版本的基本功能，验证仍能正常使用(实际的回归测试)
4. 继续现版本的功能测试

举例

系统测试过程

功能测试

性能测试

验收测试

可接受性测试：公测/内测

安装测试

性能测试performance testing

目的与职责

类型

压力测试

容量测试

配置测试

兼容测试

回归测试

安全性测试

计时测试

环境测试

质量测试

恢复测试

维护测试

文档测试

人为因素测试

测试团队

专业测试人员:组织并运行测试,从测试开始阶段,随着项目的进展,到设计测试计划和测试用例.

非专业测试人员

可靠性.可用性.可维护性

定义

失效数据

测量

增长

预测

重要性

验收测试种类

目的

种类

结果

测试文档

测试计划**test plan**

测试分析报告

第十章

培训手段

- 文档
- 图元和联机帮助
- 演示和上课
- 专家用户

培训文档种类

- 程序员指南
- 用户手册
- 管理员手册
- 系统概况指南
- 教学软件和自动化系统概述

第十一章

系统类型

ESP系统

S系统S-system

P系统P-system

E系统E-system

维护类型

改正性维护corrective maintenance-----有bug，有错误

适应性维护adaptive maintenance-----环境改变导致的改变

完善性维护**perfective maintenance-----**提升性能或修改文档提高可读性等

预防性维护**preventive maintenance-----**预防失效的发生

名词解释

XP

Extreme Programing

极限编程

ASD

Adaptive Software Development

自适应软件开发

CASE

Computer-Aided Software Engineering

计算机辅助软件工程：特定工具提高开发效率和质量

PRED/MMRE

预测误差/相对误差平均值

MSC

Message Sequence Chats

消息时序图

DFD

Data-Flow Diagram

数据流图

OCL

Object Constraint Language

对象约束语言

SDL

Specification and Description Language

规格说明语言

WBS

work breakdown structure

工作分解结构

SCR

Software Cost Reduction

软件成本降低

SAD

Software Architecture Document

软件体系结构文档

SRS

software requirement specification

软件需求说明书

COTS

商用现成品或技术

commercial off-the-shelf

ADT

Abstract Data Type

抽象数据类型

CBSE

Component-based Software Engineering

基于构件的软件工程

UML

统一建模语言

CPM

关键路径法

critical path method

CCB

变更控制委员会

change control board

简答

画图

UML类图

UML用例图

UML状态图

数据流图