L4 质量管理(1)
——概述

内容提要

- 软件质量管理概述
- 软件评审
- 代码静态扫描
- 自动化测试
- 持续集成
- 部署流水线

内容提要

- **软件质量管理概述**
- 软件评审
- 代码静态扫描
- 自动化测试
- 持续集成
- 部署流水线

软件质量管理概述

- 软件质量的定义
  - ANSI/IEEE Std 729：与软件产品满足规定的和隐含的需求能力有关的特征或者特性的全体
  - Steve McConnell《代码大全》：软件质量分为内外两部分特性：外部质量特性面向软件产品的最终用户，内部质量特性不直接面向最终用户
  - Tom Demarco：软件的用户满意度是最为重要的软件质量判断标准
  - Gerald Weinberg《软件质量管理：系统化思维》：软件质量是对人（用户）的价值，强调质量的主观性。

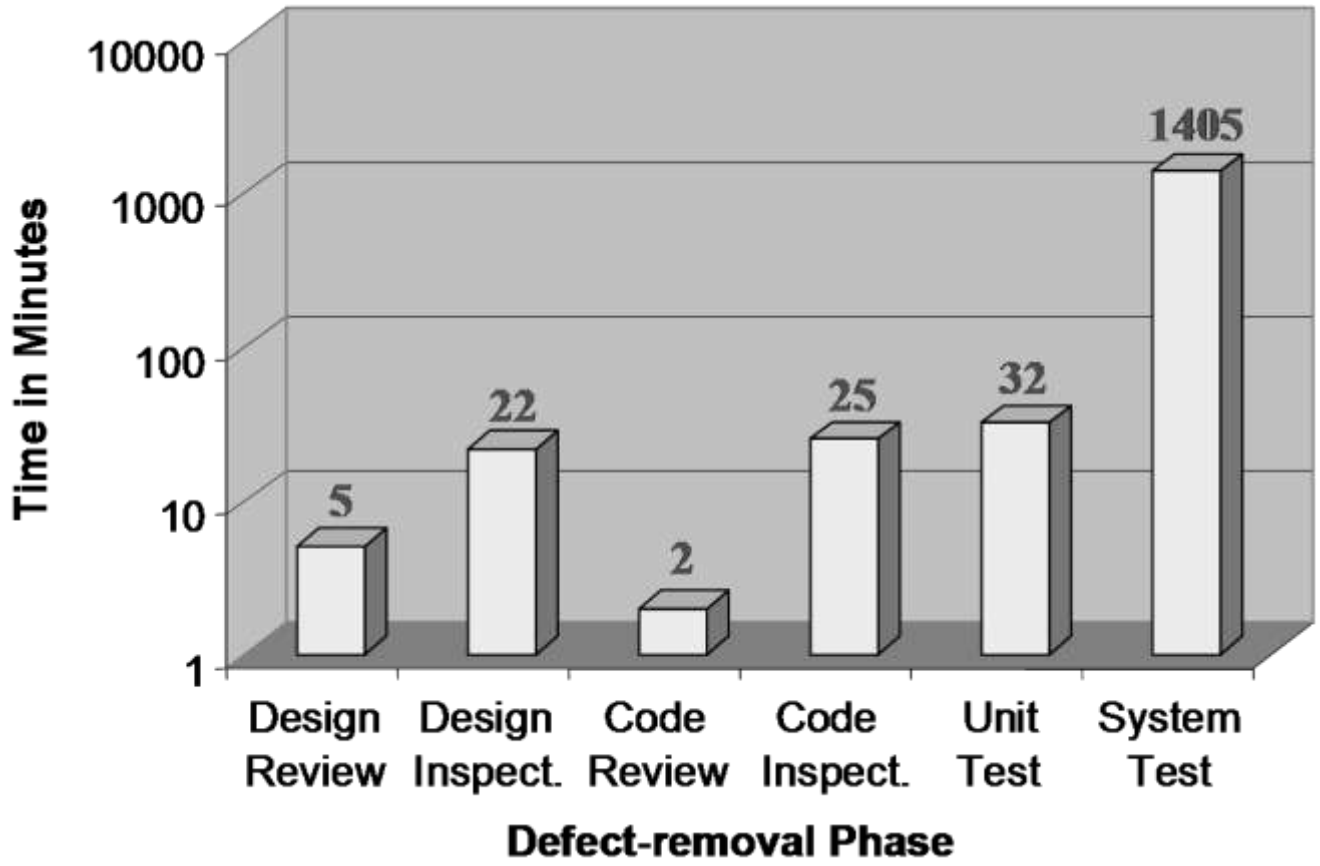
软件质量管理概述

- 软件质量的用户视角
  - 软件质量是软件满足用户需求的程度

- 明确以下问题
  - 用户究竟是谁?
  - 用户需求的优先级是什么?
  - 这种用户的优先级对软件产品的开发过程产生什么影响?
  - 怎样度量这种质量观下的质量水平?
    - **缺陷**

软件质量管理概述

- 评审与测试——消除缺陷的手段
  - 评审消除缺陷的效率往往高于测试



测试消除缺陷的典型流程

1. 发现待测程序的一个异常行为；
2. 理解程序的工作方式；
3. 调试程序，找出出错的位置，确定出错原因；
4. 确定修改方案，修改缺陷；
5. 回归测试，以确认修改有效。

评审消除缺陷的典型流程

1. 遵循评审者的逻辑来理解程序流程；
2. 发现缺陷的同时，也知道了缺陷的位置和原因；
3. 修正缺陷。

有经验的评审者可以发现80%左右的缺陷

行业数据

- **测试消除缺陷**的代价往往是**评审消除缺陷**代价的数倍

资料来源	评审消除代价	测试消除代价	应用中消除缺陷代价
IBM [Remus and Ziles 1979]	1	4.1倍于评审	
JPL[Bush 1990]	$90~$120	$10000	
[Ackerman et al. 1989]	1 Hour	2~20 Hours	
[Russell 1991]	1 Hour	2~4 Hours	33 Hours
[Shooman and Bolsky 1975]	0.6 Hour	3.05 Hours	
[Weller 1993]	0.7 Hour	6 Hours	

软件质量管理概述

- 软件产品的质量目标可以归结为首先要确保基本没有缺陷
  - **个体软件过程**采取用缺陷管理代替质量管理
  - 简化了质量管理的方法

什么是个体软件过程?

软件质量管理概述

- 个体软件过程（Personal Software Process）
  - 缘起于CMU/SEI对CMM的补充支持
    - CMM缺少对过程改进的具体指南
  - 软件工程师的**自我改进过程**
    - 从个体的角度指导每个软件工程师高质量地完成工作

软件质量管理概述

- PSP的基本原则
  - 软件系统的整体质量由该系统中**质量最差的组件**所决定
  - 软件组件的质量取决于开发这些组件的**软件工程师**，更确切地说，是由这些工程师**所使用的开发过程**所决定
  - 作为合格的软件工程师，应当**自己**度量、跟踪和管理自己的工作，应当**自己**管理软件组件的质量
  - 作为合格的软件工程师，应当建立**持续的自我改进**机制
    - 从自己开发过程的偏差中学习、总结，并将这些经验教训整合到自己的后续开发实践中

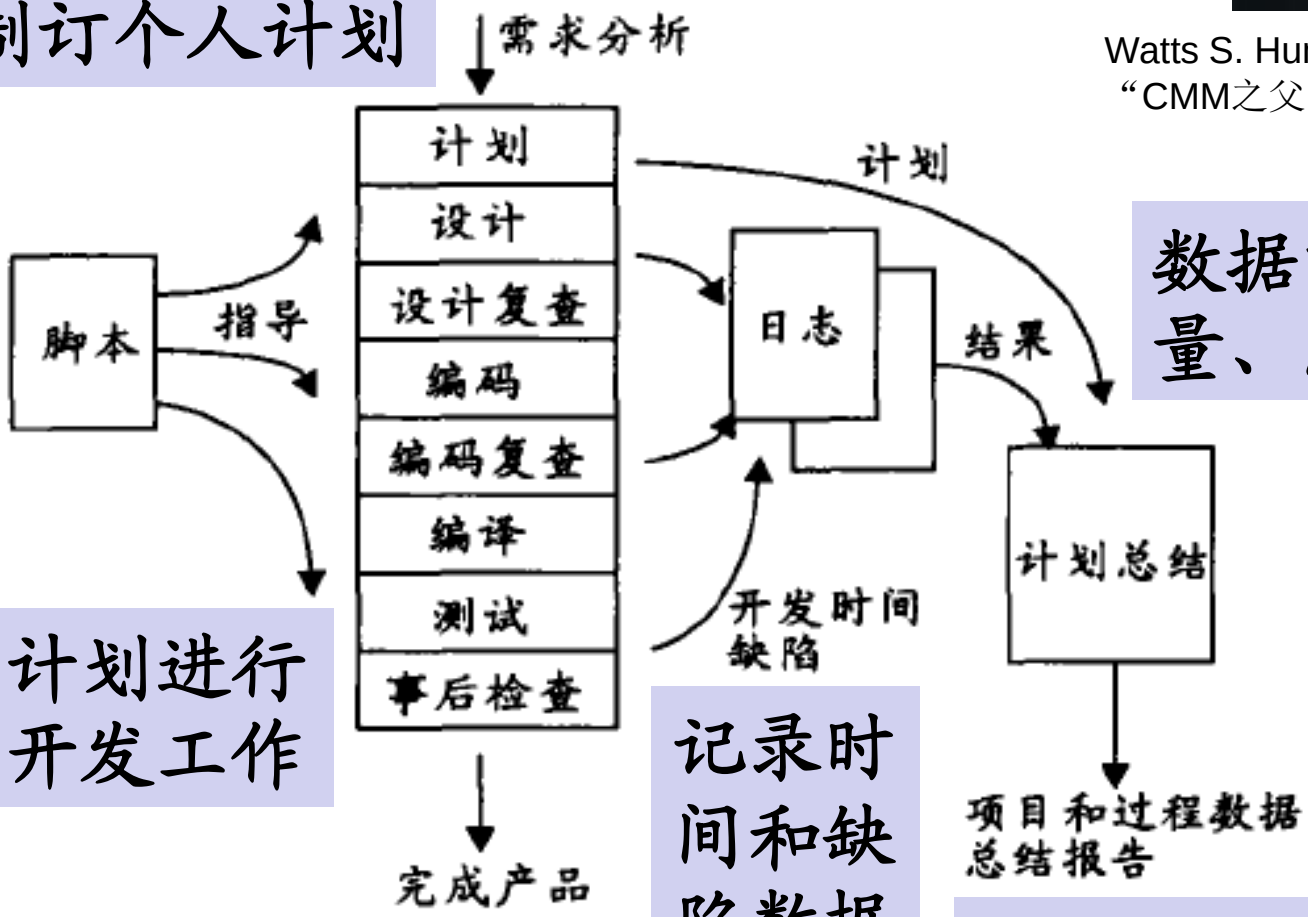
突出个体软件工程师在管理和改进自身过程中的能动性

软件质量管理概述

Watts S. Humphrey (1927-2010)
“CMM之父”“软件质量之父”

- PSP的基本流程（Humphrey, 2005）

制订个人计划



数据汇总、度量、总结分析

按照计划进行各项开发工作

记录时间和缺陷数据

数据及报告提交

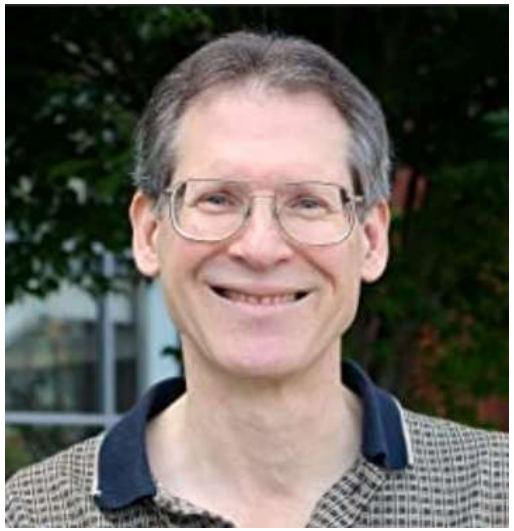
内容提要

- 软件质量管理概述
- **软件评审**
- 代码静态扫描
- 自动化测试
- 持续集成
- 部署流水线

软件评审

“不管你有没有发现他们，缺陷总是存在，问题只是你最终发现他们时，需要多少纠正成本。

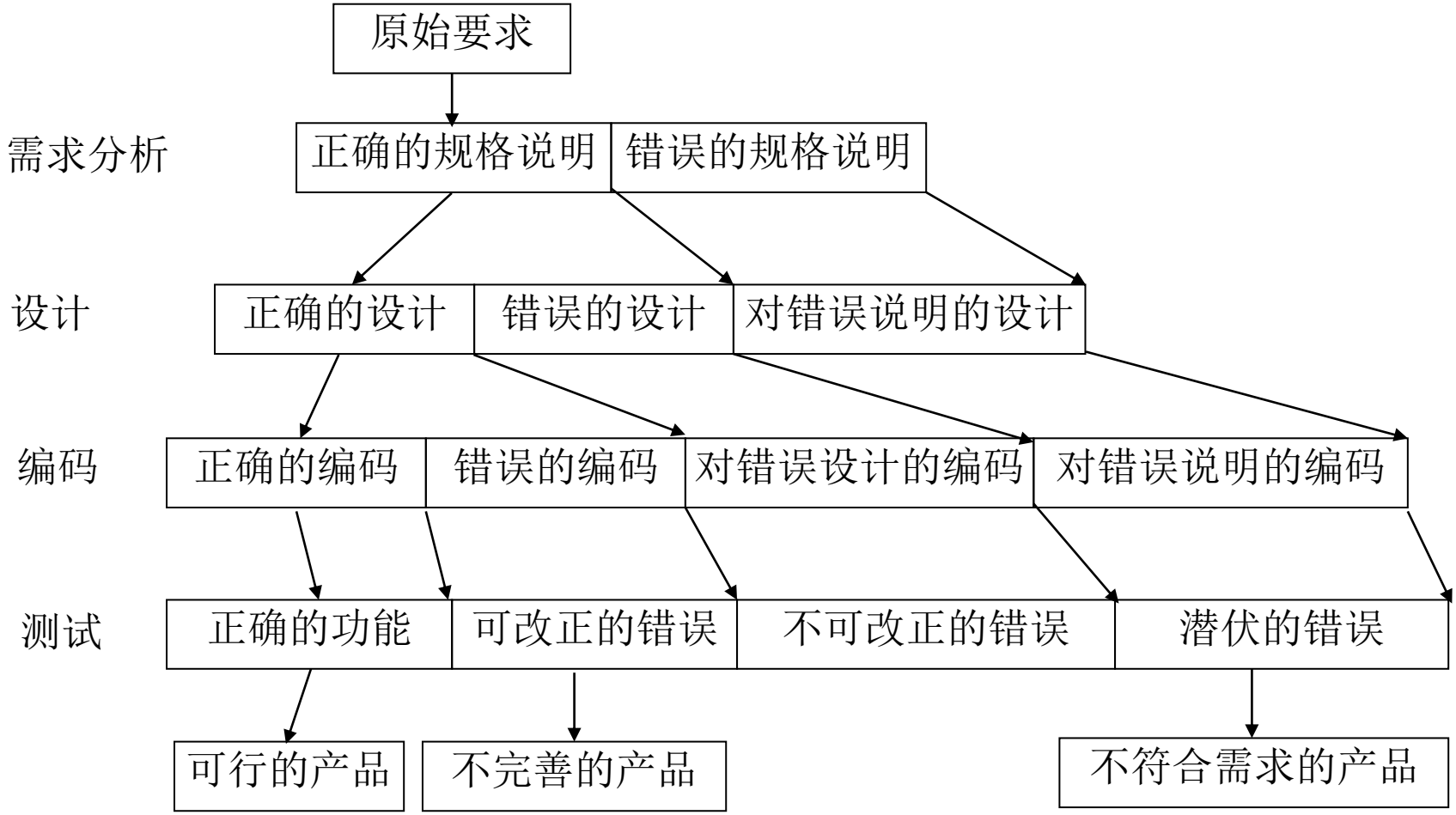
“评审的投入把质量成本从昂贵的后期返工转变为早期的缺陷发现。”



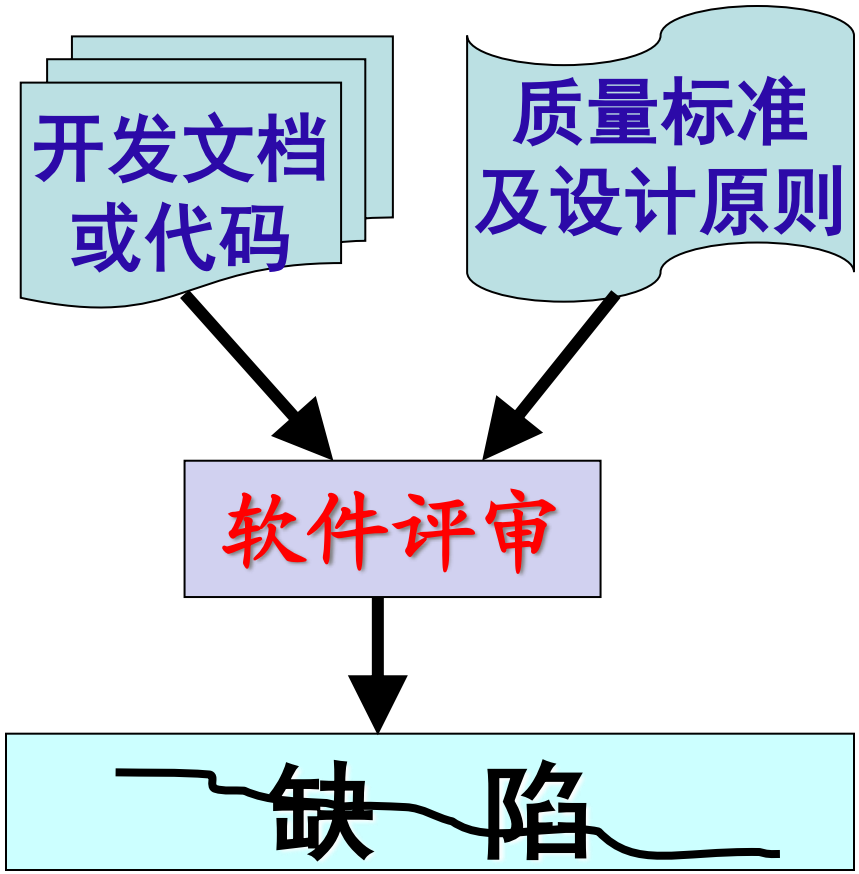
Karl E. Wiegers

软件评审

- 错误的累积效应



软件评审



- 手段
  - 通过文档阅读、讨论等进行主观判断
- 目的
  - 尽早从软件工作产品中识别并消除缺陷，从而提高软件开发质量和效率
- 其他的好处
  - 促进质量文化
  - 促进优秀开发实践的交流和传播
  - 激发团队合作精神

软件评审分类

- **管理评审**：向上层管理者提供信息，以帮助决策
- **同级(同行)评审**：寻找产品中的缺陷和改进契机
- **项目总结评审**：对完成后的项目或阶段进行评审，以吸取经验
- **状态评审**：向项目组提供最新的项目状态信息

各种同级评审方法

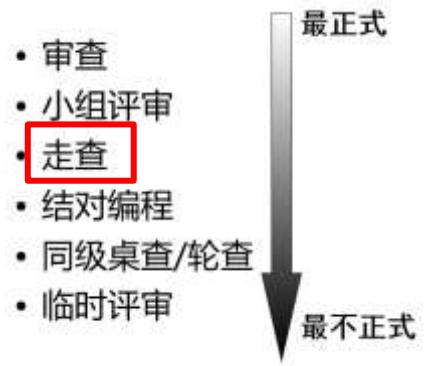
- 审查
- 小组评审
- 走查
- 结对编程
- 同级桌查/轮查
- 临时评审

审查 Inspection

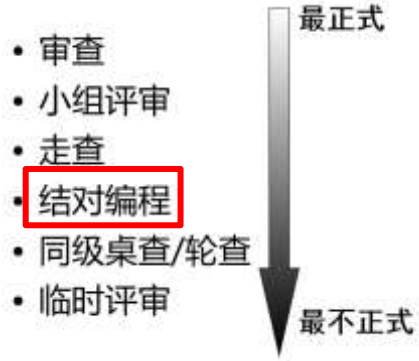
- 严格定义评审过程，明确分工
  - 五种角色
    - 作者Author/Producer
      - 被评审工作产品的创建者或维护者
    - 评审组长Review Leader
      - 计划、安排、组织评审活动
    - 评审员Reviewer
    - 记录员Recorder
    - 阅读者Reader
      - 展示工作产品各个部分，引导评审员进行评论
  - 五个阶段：概览、准备、审查会议、修改及验证、过程分析

走查(Walkthrough)

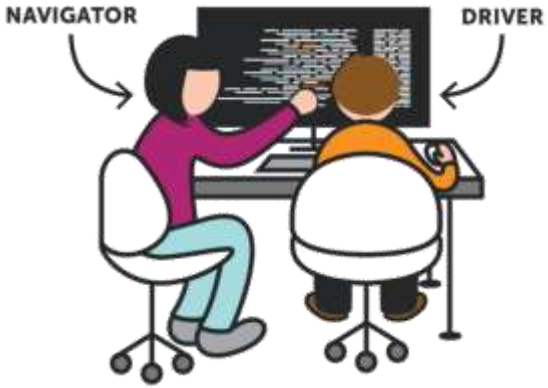
- 非正式的评审，由文档或代码作者向审查者进行介绍
- 没有事先的计划，适用于促进其他人的了解或寻求特定问题解决方案
- 代码走查
  - 开发人员向其他人进行代码的阐述和解释，帮助审查者识别出错误或对某些问题提出解决方案



结对编程



- “结对”可以不仅在编码活动中出现
- XP中提出的开发模式
  - 两个开发人员使用同一台机器完成同一段程序的编码工作
- 结对的队员是自由组合和经常变动的：促进团队成员的互相学习并保障编码的质量
- 事实证明，结对编程在提高软件质量的同时并不会降低效率



其他同级评审方法

- 小组评审
  - 不需要执行会议等严格过程的审查
  - 以小组为单位，相比于正式的Inspection具有一定的灵活性，具有讨论的性质
- 桌查/轮查
  - 由单个或多个评审者独立进行评审
- 临时评审
  - 开发者请同事临时帮助查找或解决某个特定问题

形式不同，但目的都是为了更早发现缺陷，提高产品质量。

评审误区1

- 现象：评审会议陷入到对问题的**解决**上
- 后果：
  - 停止了对评审目标文档或代码的检查
  - 对所讨论的问题不感兴趣的评审者停止了思考
  - 当会议结束的时间快到了，评审者会很快翻过剩下的材料，宣布评审的成功，而这些材料中很可能隐藏了主要缺陷
- 对策：评审组长要控制会议的焦点，集中在**发现**问题上

评审误区2

- 现象：评审的焦点放在**文档形式**而不是内容本身上
- 原因：缺乏充足的准备以及对项目特定需求和设计的了解，只做了表面的检查
- 对策：
  - 事先定义编码标准和文档标准模板，将产品是否符合模板和标准作为评审准入条件—由QA执行先期的常规形式审查
  - 设定和明确**评审的目标**

评审误区3

- 现象：评审结果被用于**对工作者**的工作评估
- 后果：
  - 为了避免惩罚，开发者可能不会将他们的产品交付评审，同时也会拒绝评审同事的工作以避免使别人受到惩罚
  - 在评审过程中，评审者可能不会直接指出错误，相反会事后告诉作者，这样就不会得罪作者
  - 开发者会采取一些欺骗手段或在评审会上为自己辩解
  - 尽量少地找出错误成为评审心照不宣的目标，从而降低评审的意义（投资回报降低）
- 对策：评审时，**对事**不对人

代码评审 Code Review

- 针对**代码**这种工作产品的评审
  - 通过阅读代码来检查**代码质量**的活动
    - 代码**与编码标准的符合性**以及功能的**正确性**等
- 在软件开发过程中，代码评审具有多种形式
  - 正式的过程控制
    - 有代码评审人员从代码提交、合并的流程中进行控制；评审不通过的代码不能进入代码库
    - 包括代码的编码规范、实现逻辑、异常处理等内容
  - 非正式的本地评审
    - 对本地代码的评审或特定问题的查看
    - 采用Pair Programming编写代码本身就带有代码评审的特点

代码评审的指导原则

- 每个人将代码**提交到主干之前**，必须要有**同行**来**评审**他们的变更
- 每个人应该**持续关注其他成员的提交活动**，以便识别和审查出潜在的冲突
- 定义哪些变更属于**高风险的变更**（例如数据库变更、安全敏感的身份验证模块变更），从而决定是否需要请领域专家来审查
- 如果提交的变更规模太大了，以至于让人很难理解（读了几遍还无法理解，或者需要提交者进行解释），那么这个变更需要**分解成多个较小的变更来提交**，使之一目了然

代码评审工具

- Phabricator
  - 基于Web的协同开发工具
  - 提供代码差异比较，可对代码差异进行显示，并由评审者填写备注和评论
  - 允许通过Audit特性先提交代码，后补充评审
- Collaborator
  - 支持多种代码库
  - 支持对GitHub的Pull Request的评审处理
  - 对接Jira中的问题单
- Gerrit
  - 开源的轻量级代码审查工具；支持与Jenkins集成

内容提要

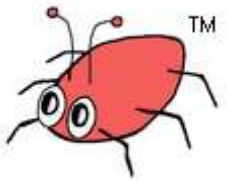
- 软件质量管理概述
- 软件评审
- **代码静态扫描**
- 自动化测试
- 持续集成
- 部署流水线

代码静态扫描

- 通过代码静态分析发现可能的缺陷
  - 通过程序分析技术判断是否存在逻辑问题
  - 通过给定的规则判断代码是否规范
  - 并不需要运行程序
- 可能存在一些“误报”
  - 程序分析技术本身的限制
  - 开发人员并不关心或可以容忍的问题
- 企业往往会根据实际情况定制自己的规则
- 如果是对遗留软件进行扫描，那么问题可能非常多，导致开发人员无法有效处理
  - 采用控制新增、精选规则等方式进行管理

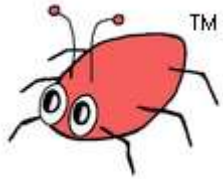
代码静态扫描

- FindBugs/SpotBugs
- SonarQube
- CodeSonar
- PMD
- Klocwork
- CheckStyle




代码静态扫描

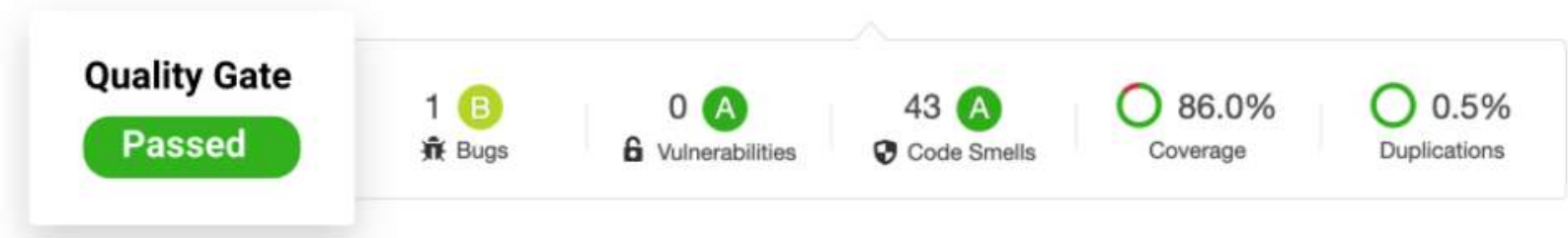
- SpotBugs
  - 针对Java代码
  - 十大类代码问题（FindBugs支持九大类）
    - 包括Bad practice（例如定义clone方法但没有实现Cloneable接口）、Correctness（例如无限循环）、Experimental（例如资源没有try-finally清理）、Vulnerability（例如返回值暴露内部数据结构）、Performance（例如用java.lang.String(String)创建一个String）、Security（例如硬编码数据库密码）



  - 提供Eclipse、Ant、Maven、Gradle的插件
    - 开发时的bug检测
    - 编译构建时的质量控制

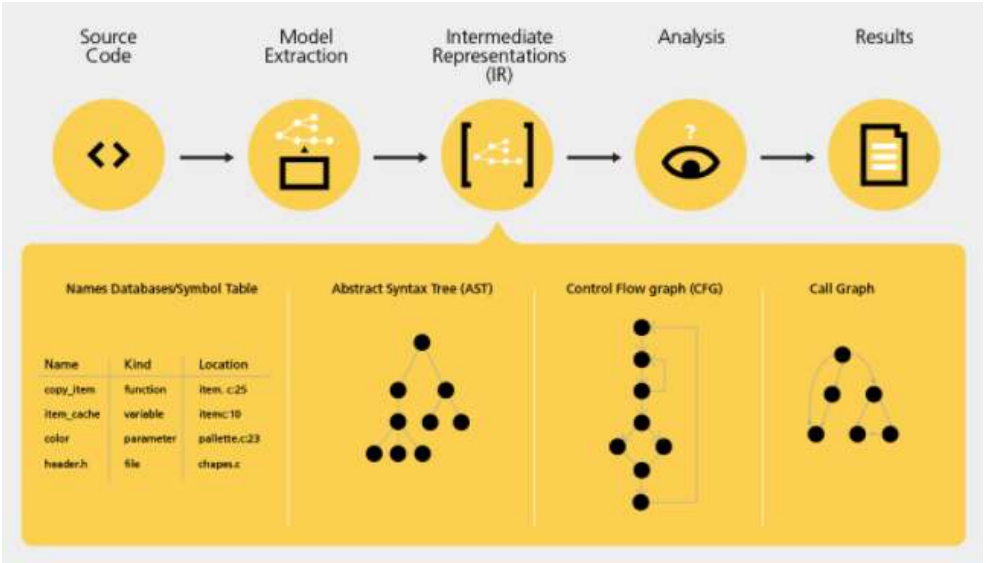
代码静态扫描

- SonarQube
  - 支持Java、C#、C/C++、JavaScript、TypeScript、Python、Go、Swift、PHP、Ruby、Scala等多种语言
  - SonarLint是支持Eclipse、IntelliJ IDEA、VS Code等IDE的插件，与SonarQube实例无缝对接
  - 支持GitHub、GitLab、Azure DevOps、BitBucket、Docker的集成
  - 支持质量门禁（Quality Gate）
  - 发布代码的质量、代码可维护性、安全性分析



代码静态扫描

- CodeSonar
  - 支持C/C++、Java、C#、二进制可执行代码或库
  - 支持Eclipse、GitLab、Jenkins、Jira、Visual Studio、Visual Studio Code的集成
    - 在开发环境中自动检测
    - 自动纳入CI流程
    - 自动创建Jira Issue
  - 提供丰富的代码安全性和质量的扫描能力



代码静态扫描

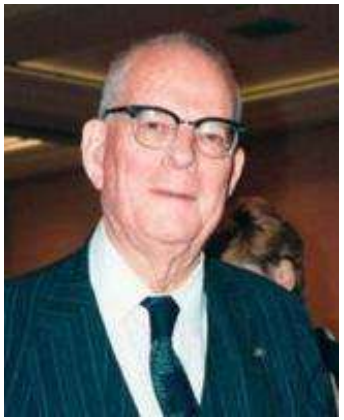
- 使软件开发过程受益
  - 实时检测各类代码问题
    - 潜在的bug
    - 代码异味（smell）
    - 安全性问题
    - 性能问题
    - ……
  - 通过门禁（gate）阻止不良代码进入发布
  - 与缺陷管理系统集成，有效追踪代码的问题

内容提要

- 软件质量管理概述
- 软件评审
- 代码缺陷静态扫描
- **自动化测试**
- 持续集成
- 部署流水线

自动化测试

停止依赖于大批量检查来保证质量的做法。
改进过程，从一开始就将质量内嵌于产品之中。



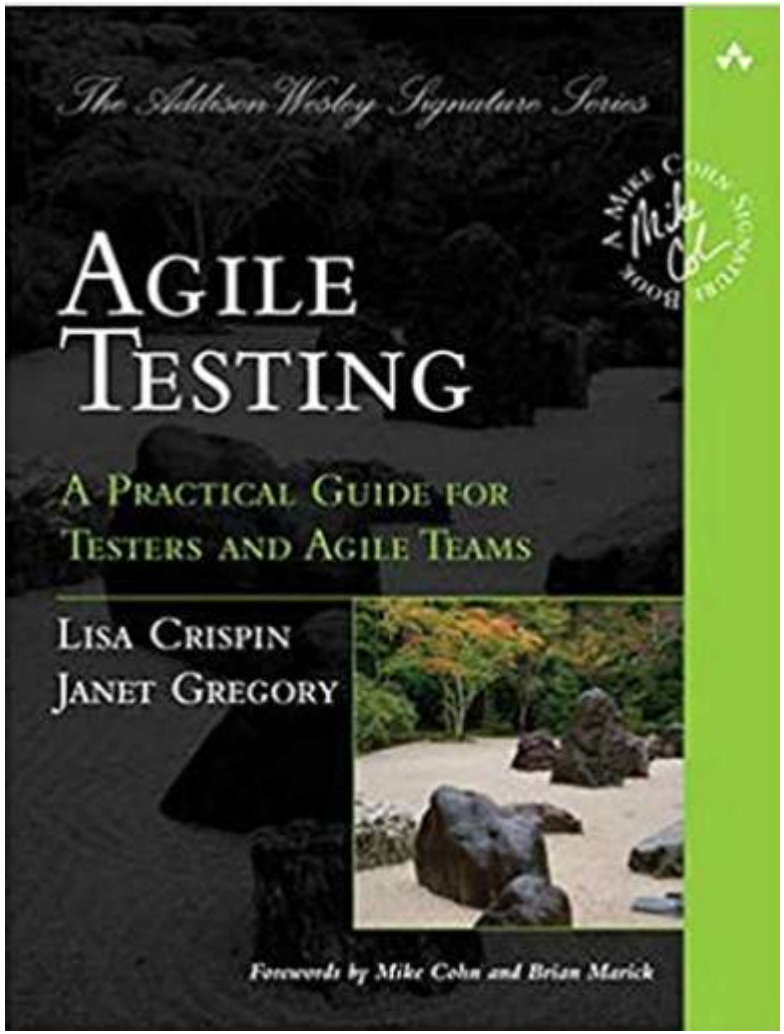
Deming (1900-1993)

自动化测试

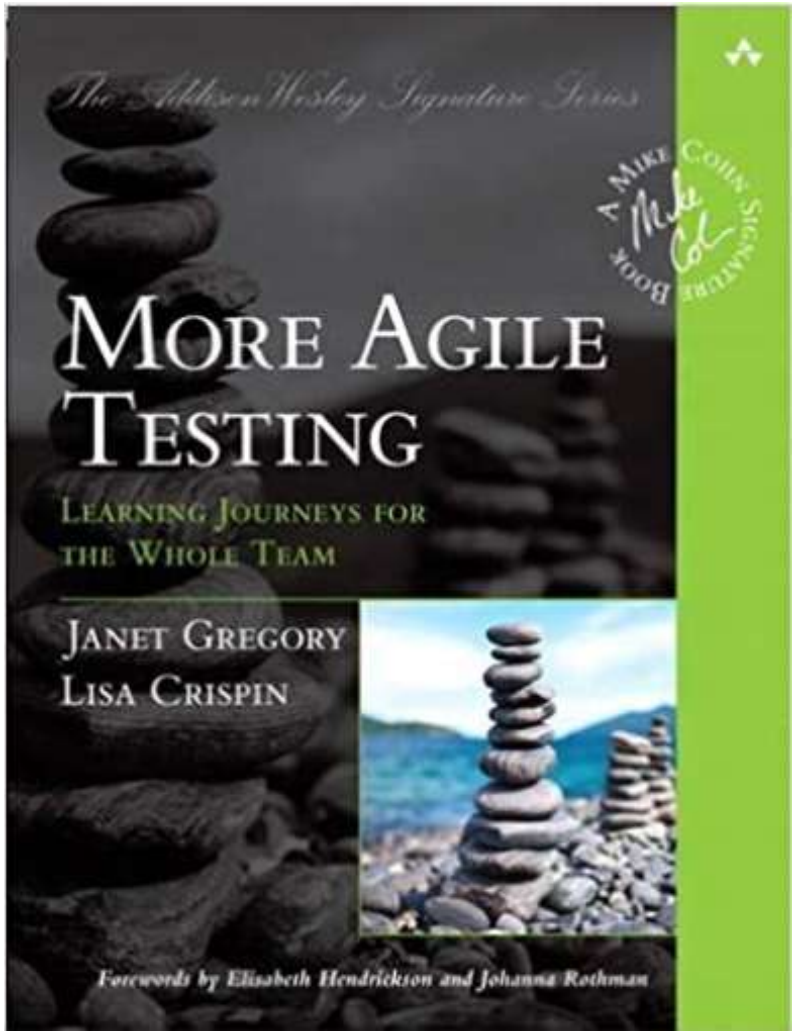
停止依赖于大批量检查来保证质量的做法。
改进过程，从一开始就将质量内嵌于产品之中。

- 测试是跨职能部门的活动，是整个团队的责任
- 应该从项目一开始就一直做测试
- **质量内嵌**是指从多个层次（单元、组件和验收）上写自动化测试，并将其作为部署流水线的一部分来执行

自动化测试



（2009）



（2014）

自动化测试

- 测试的分类

Brian Marick

业务导向的

自动的 功能验收测试	手工的 演示 易用性测试 探索性测试
单元测试 集成测试 系统测试 自动的	非功能验收测试， 包括容量测试、 安全性测试等 手工的/自动的

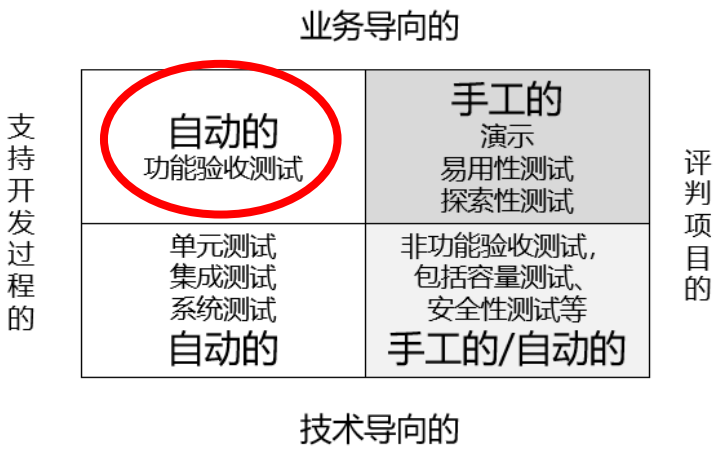
支持开发过程的

评判项目的

技术导向的

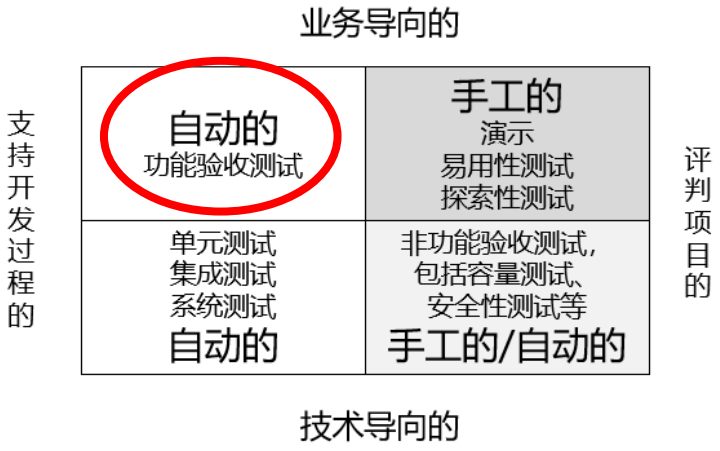
自动化测试

- 业务导向且支持开发过程的测试
  - 开发一个用户故事之前，应该写好验收测试，采取完美的自动化形式
  - 自动化测试工具
    - Cucumber、JBehave、Concordion、Twist
  - 从验收的角度描述一个**场景(scenario)**
    - Given-when-then
    - 假如-当-那么
  - Happy Path
    Alternative Path
    Sad path
    - 正常流程与其他流程



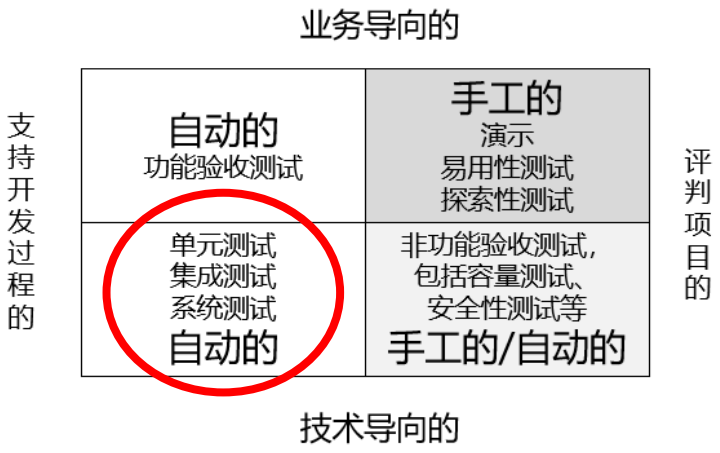
自动化测试

- 业务导向且支持开发过程的测试
  - 自动化验收测试的维护成本可能很高
    - 大而复杂的验收测试可能产生极大的维护成本
  - 并非所有的东西都需要自动化
    - 比如易用性测试、界面一致性测试
  - 我们倾向于：自动化验收测试完全覆盖Happy Path
    - 并且仅覆盖一些极其重要的部分
    - 前提是其他类型自动化回归测试是全面的
  - 把对Happy Path的自动化验收测试作为**冒烟测试**
    - 对其他测试做战略选择



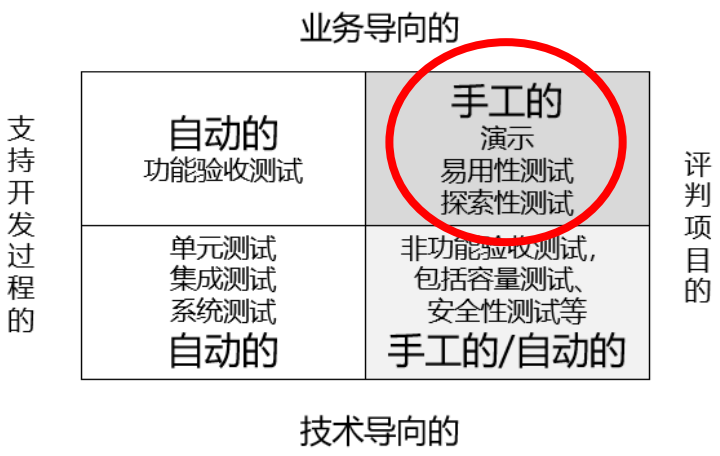
自动化测试

- 技术导向且支持开发过程的测试
  - 单元测试
    - 单独测试一个特定的代码段
    - 不依赖于数据库、文件系统或其他外部系统交互
  - 组件测试/集成测试
    - 捕获应用系统不同部分之间交互时产生的缺陷
    - 需要更多的“准备工作”
      - DB、FS、其他系统等
  - 部署测试
    - 检查部署过程是否正常
      - 应用程序是否正确安装配置
      - 服务之间是否正常通信



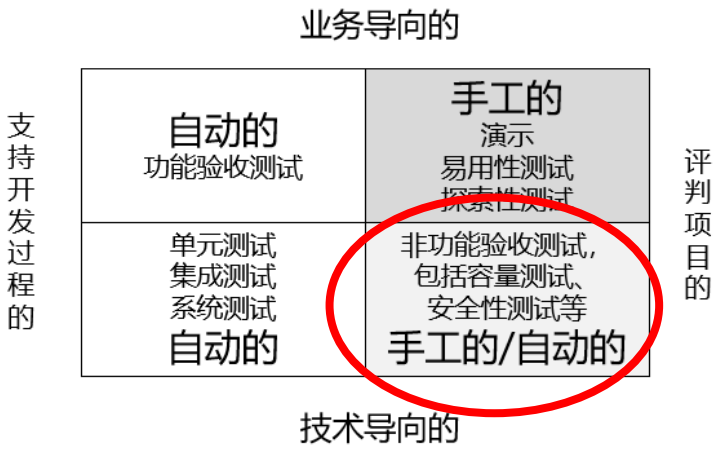
自动化测试

- 业务导向且评价项目的测试
  - 通过手工测试验证我们交付给用户的软件是否符合其期望
    - 不仅验证是否符合需求规格说明（如果有的话）
    - 还验证需求规格说明本身的正确性
  - 演示：面向客户/用户
    - 每个迭代结束时都要演示新的功能
    - 开发过程中尽可能多的演示
  - 探索性测试(exploratory test)
    - 通过测试创造新的测试
  - 易用性测试(usability test)
    - 观察用户使用并记录使用情况



自动化测试

- 技术导向且评价项目的测试
  - 非功能验收测试
    - 非功能需求不是“面向业务”的
    - 但它仍然是需求，并且应该与功能需求有同等重要地位
    - 非功能验收测试与功能验收测试大相径庭
      - 可能需要更多资源
      - 可能花费更多时间
      - 可能优先级更低
      - 可能运行频率更低
    - 考虑尽早准备一些此类测试



自动化测试

- 现实中的情况与应对策略
  - 新项目
    - 选择技术平台和测试工具
    - 建立一个简单的自动化构建
    - 遵守INVEST原则建立用户故事并考虑其验收条件

客户、分析师、测试人员

测试人员、开发人员

开发人员

定义验收条件	实现验收测试自动化	编码实现

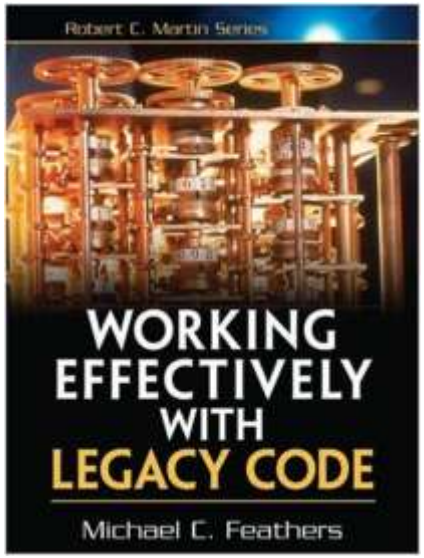
一旦自动化测试失败，就要以高优先级去修复

自动化测试

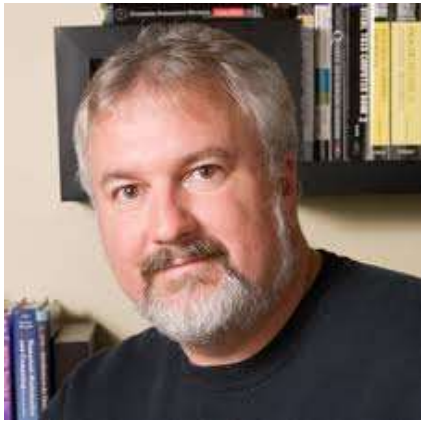
- 现实中的情况与应对策略
  - 项目进行中（并非总是从0开始）
    - 选择最常见、最重要且高价值的用例为起点
    - 测试覆盖范围可以略宽于通常的用户故事级别
      - 获得更加广泛的测试覆盖
    - 只对Happy Path进行自动化测试（资源不足时）
      - 确认基本功能正常且交付了价值
    - 会有较多的手工测试
      - 灵活性高、适应不同的变化
      - 当基本稳定时，可以考虑逐步自动化
      - 当总是发生测试失败，则可以延迟自动化或者在自动化测试中忽略它

自动化测试

- 现实中的情况与应对策略
  - 遗留系统
    - “没有自动化测试的系统就是遗留系统”
      - 简单粗暴却不无道理



测试那些你修改的代码！



- 建立自动化测试环境、自动化构建流程
- 创建“冒烟测试”
  - 覆盖核心功能

自动化测试

- 现实中的情况与应对策略
  - 遗留系统

只写那些有价值的自动化测试！

- 遗留系统通常没有标准组件化、结构比较差
- 测试结束后仔细验证系统的状态
  - 如果还有时间，验证Alternative Path和Sad Path
- 实现系统功能的代码 vs 支撑性的框架代码
  - 对稳定的、不需要修改的底层支撑性代码补充自动化测试并没有什么价值（除非这些代码被修改了）

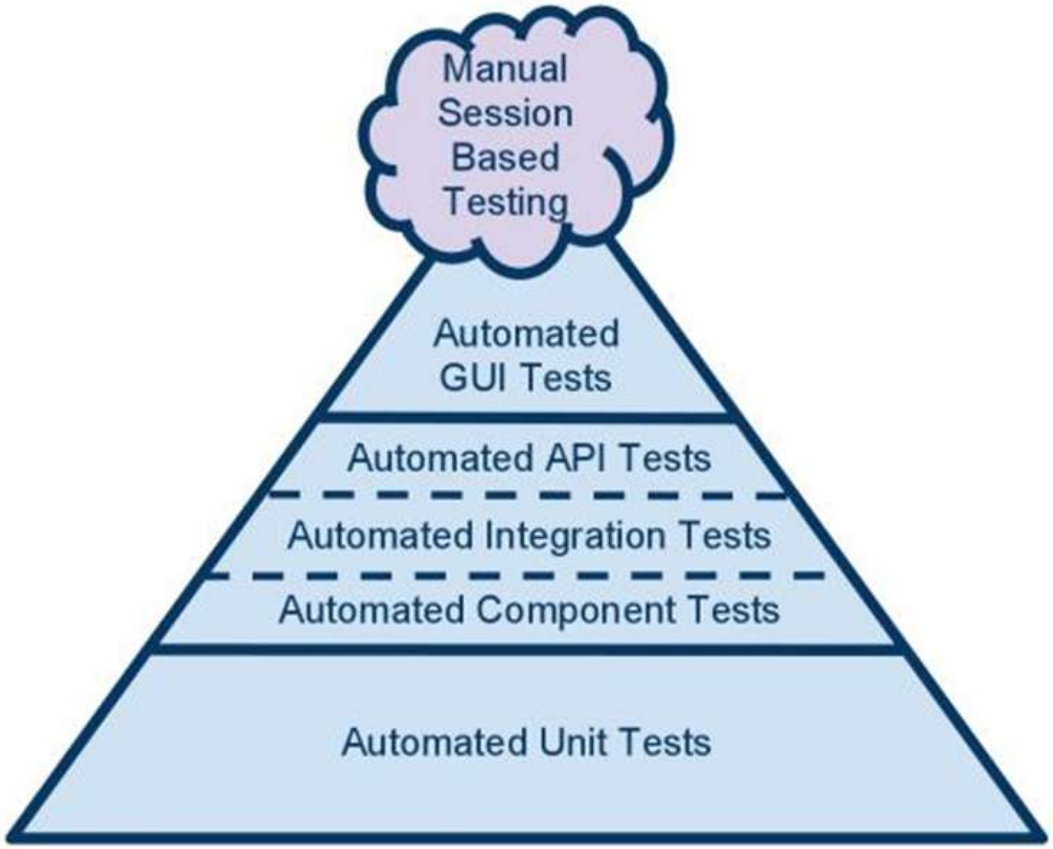
自动化测试

- 现实中的情况与应对策略
  - 集成测试
    - 组件测试与集成测试的界限并不十分清晰
    - 这里“集成测试”是指确保系统的每个独立部分都能够正确作用于其依赖的那些服务的测试
    - 集成测试的上下文（Context）
      - 与真正依赖的外部系统交互，或者用外部服务供应商提供的替代系统
      - 运行于自己创建的测试用具（test harness），并且这些测试用具也是代码库的一部分
    - 部署到生产环境前，避免与真正的外部系统交互
      - 用“防火墙”进行隔离，或者通过配置与模拟系统交互
      - 例如在线支付

自动化测试

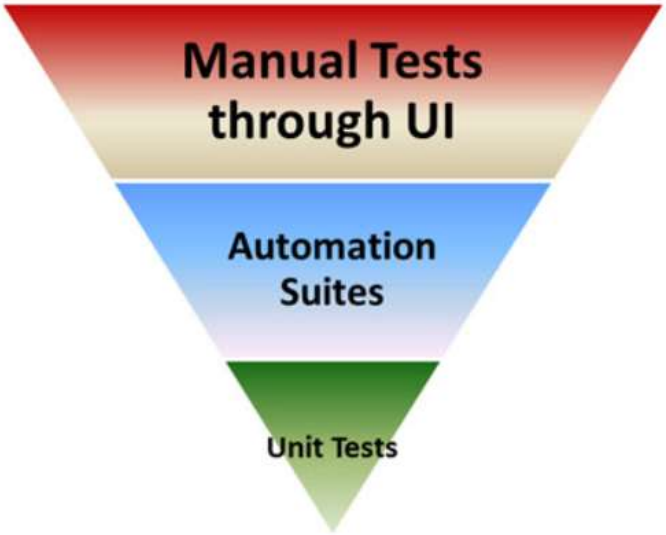
- 在自动化测试中尽早发现错误

Martin Fowler



理想的自动化测试金字塔

非理想的自动化测试倒金字塔



内容提要

- 软件质量管理概述
- 软件评审
- 代码缺陷静态扫描
- 自动化测试
- **持续集成**
- 部署流水线

持续集成

- 基本的持续集成系统
  - 持续集成“是实践，并不是工具”
  - 开源工具与商用工具
    - Jenkins，Hudson，CruiseControl
    - Bamboo， Go，TeamCity，Pulse……
  - 使用持续集成服务器
    - 提交新代码时
      1. 如果有构建正在运行，等待；如果构建失败，先修复
      2. 构建成功且测试通过，则拉取代码更新自己的开发环境
      3. 开发机上构建、测试，确保本地正常
      4. 提交代码，等待构建结果
      5. 如果失败，则立即修复，然后转到3；如果成功，YES!

持续集成

- 前提条件
  - 频繁提交
    - 每天提交几次
    - 定期提交到主干（每次修改都比较小）
    - 在有限情况中使用分支，在主干上进行持续集成
  - 创建全面的自动化测试套件
    - 单元测试
    - 组件测试
    - 验收测试

持续集成

- 前提条件
  - 保持较短的构建和测试过程
  - 管理开发工作区
    - 开发机本地环境的自动化过程应当与持续集成环境一致
    - 细心的配置管理
      - 不仅管理代码，还包括测试数据、数据库脚本、构建脚本、部署脚本
    - 第三方依赖管理
      - 确保库文件和组件的版本正确
    - 确保自动化测试能在开发机上运行
      - 原则：在开发机本地有一套可构建的环境
      - 核心：快速验证是根本

持续集成

- 必不可少的实践
  - 构建失败后不要提交新的代码
  - 提交前在本地运行所有的提交测试，或者让持续集成服务器完成此事
  - 等提交测试通过后再继续工作
  - 回家之前，构建必须是成功状态
  - 时刻准备回滚到前一个版本
  - 构建失败时要规定一个修复时间，超时即必须回滚
  - 不要将失败的测试注释掉
  - 为自己导致的问题负责（代码集体所有权）
  - 测试驱动开发

持续集成

- 推荐的实践
  - 极限编程开发实践
    - 测试驱动开发、代码集体所有权、重构
  - 若违背架构原则，就让构建失败
  - 若测试运行变慢，就让构建失败
  - 若有编译警告或代码风格问题，就让构建失败
    - 退一步：编译警告或者风格问题的个数不能增加
    - 有时这个决策有些困难，例如要不要把CheckStyle加到流程中

内容提要

- 软件质量管理概述
- 软件评审
- 代码缺陷静态扫描
- 自动化测试
- 持续集成
- **部署流水线**

部署流水线

- 持续集成的主要关注对象是开发团队
- 然而……
  - 构建和运维团队的人员一直等待说明文档或缺陷修复
  - 测试人员等待“好的”版本构建出来
  - 新功能开发完成几周后，开发团队才收到线上缺陷报告
  - 开发快完成时才发现当前软件架构无法满足系统的一些非功能需求
- 交付流程的瓶颈在哪里？
  - 需要更完整的 端到端 的方法来交付软件

部署流水线

- 软件从版本控制库到用户手中这一过程的自动化表现形式
- 软件从开发到发布过程中共同具有的阶段
  - **提交阶段**：从技术上断言系统是可工作的
  - **自动化验收测试阶段**：从功能和非功能角度上断言系统时可工作的
  - **手工测试阶段**：断言系统是可用的，验证系统是否为用户提供了价值
  - **发布阶段**：将软件交付给用户
- 部署流水线由上述阶段，以及为软件交付流程建模所需的其他阶段组成

部署流水线

- 端到端的部署流水线系统
  - 一个“拉式系统”
    - 测试人员、运维人员、开发人员、管理人员按需查看和使用所需的东西
  - 快速运行构建、测试和部署系统
  - 在开发和部署上获得了一定程度的自由和灵活性
  - 以更高的质量和相当低的成本与风险来创建、测试、部署复杂系统
  - 详细内容将在后续课程中细化展开

课程项目实践

- 提供一套原始实现代码
  - 代码需要托管到华为CodeArts（原DevCloud）
- 各组可以基于现有的代码进行修改，或自行重新开发
- 各组创建各自的项目
  - 建议利用CodeArts上的Scrum项目模板
  - 注意区域（例如华东-上海；不同区域上的项目不互通）
- 自行选择部署服务器（公网）
  - 如无公网服务器，可将部署包手工部署在本地或内网服务器上

课程项目实践

- 实践要求
  - 对发布目标进行初步计划
  - 讨论制定Product Backlog
  - 编写相应的Epic、Feature以及用户故事(Story)
  - 讨论迭代，细化任务
  - 体会对故事点的认知以及估算效果（比如实际可用时间会打多少折扣）
  - 体会每次迭代的固定time box
    - 考虑最后一周为阶段性版本发布
  - 采用代码扫描、自动化测试、代码评审、持续集成和持续交付等技术提升开发质量和效率

课程项目实践

- 自习教室座位预约平台
  - **分析需求，整理用户故事列表**
    - **初步分析用户故事的主要价值**
    - **对Epic、Feature、Story进行识别**
    - **对用户故事列表的优先级进行大致排序**
  - 开发
    - 确定开发节奏（迭代周期），创建迭代
    - 选择用户故事，细化任务
    - 测试应该包含在开发任务中，不单独列测试任务
  - 持续集成和部署流水线
    - 熟悉代码托管、云端编译构建等全流程活动

课程项目实践

- 自习教室座位预约平台
  - 用户故事的验收测试用例
  - 作为用户，我要能预约座位，以便能在所选的时间段和座位上自习
    - 测试：正常预约（预约时段晚于当前时间，且座位可用）
    - 测试：预约时段座位不可用（包括座位已经被占用，管理员座位未开放）
    - 测试：预约时间不正确（包括预约时段在当前时间之前，预约时段不是整点）
    - 测试：用户在预约时段已经有另一个预约
  - 查询指定时间的可用座位?
  - 查询指定时间、指定座位是否可用?

课程项目实践

- 自习教室座位预约平台
  - 用户故事的验收测试用例
  - 预约座位
  - 测试：正常预约（预约时段晚于当前时间，且座位可用）
    - **给定**用户A，座位S，且座位S的占用时段为空，且当前时间为2023年4月6日8:00
    - **当**A预定S，预定时间段为2023年4月6日10:00-11:00
    - **那么**预约成功
  - 设计方案?

Next Lecture

- 质量管理(2)