

概述

- 编码：把软件设计结果翻译成用某种程序设计语言书写的程序。
 - a natural consequence of design
 - programming language characteristics
 - good programming practice
 - coding style

程序设计语言

- 语法（syntax）用来表示构成语言的各个记号之间的组合规则。
 - for（表达式1；表达式2；表达式3）语句
 - 语法中不涉及到这些记号的含义，也不涉及使用者

程序设计语言

- 语义（semantic）用来表示按照各种表示方式所表示的各个记号的特定含义，但它不涉及到使用者。
 - for语句中：表达式1表示循环初值；表达式2表示循环条件；表达式3表示循环的增量；语句为循环体。整个语句的语义是：
 - ✓ （1）计算表达式1
 - ✓ （2）计算表达式2，若计算结果为0，则终止循环；否则转（3）
 - ✓ （3）执行循环体
 - ✓ （4）计算表达式3
 - ✓ （5）转向（2）

程序设计语言

- 语用（pragmatic）用来表示构成语言的各个记号和使用者的关系。
 - 语言是否允许递归？是否要规定递归层数的上界？这种上界如何确定？这些都属于语用上的问题。

选择的标准


- 系统用户的要求
- 测试和维护的要求
- 工程规模
- 程序员的知识
- 软件可移植性要求
- 软件的应用领域

良好编程实践

- 变量的命名要有意义且一致
 - 有意义和一致主要是对将来的维护程序员而言
 - *averageFreq, frequencyMaximum, minFr, frqncyTotl*
 - *average & mean*
 - 变量名中单词的顺序要一致
 - *frequencyMaximum & minimumFrequency*
 - *frequencyAverage, frequencyMaximum, frequencyMinimum, frequencyTotal*
 - *averageFrequency, maximumFrequency, minimumFrequency, totalFrequency*
 - 命名约定
 - Hungarian Naming Conventions
 - *ptrChTmp(pointer+character+temporary)*

良好编程实践

■ 代码的self-documenting

- ❑ 程序员根据命名约定精心选择变量名，认为无需注释了
- ❑ 问题是其他程序员，包括软件质量保证小组、维护程序员能否读懂程序
- ❑ *xCoordinateOfPositionOfRobotArm*  *xCoord*
- ❑ 在代码的起始位置加上变量名的解释， prologue comments
- ❑ 在代码内部使用inline comments

例

```
/**
 * Robot movement.
 * @param xCoord the x coordinate of the position of the robot arm.
 * @param yCoord the y coordinate of the position of the robot arm.
 */
private void robotMovement(double xCoord, double yCoord)
{
    Robot waterRobot = new Robot();
    .....
    waterRobot.move();           //move the robot
    .....
}
```

良好编程实践

■ 使用参数

```
public class GeneticAlgorithm {  
    int generation = 1;  
    while(generation <= 10000) {  
        doCrossOver();  
        doMutate();  
        generation++;  
    }  
}
```

```
public class GeneticAlgorithm {  
    private static final int MaxGenerations = 10000;  
    int generation = 1;  
    while(generation <= MaxGenerations) {  
        doCrossOver();  
        doMuatae();  
        generation++;  
    }  
}
```

良好编程实践

■ 使用参数

config.xml

```
<properties>
  <entry key="PopulationSize">1000</entry>
  <entry key="MaxGenerations">10000</entry>
</properties>
```

```
public class GeneticAlgorithm {
    private static final int MaxGenerations =
        Config.getMaxGenerations();
    int generation = 1;
    while(generation <= MaxGenerations) {
        doCrossOver();
        doMuatae();
        generation++;
    }
}
```

编码风格

程序实际上也是一种供人阅读的文章，有一个文章的风格问题，应该使程序具有良好的风格。

1、程序内部的文档

- 恰当的标识符
- 适当的注解
- 程序的视觉组织
 - 空格
 - 阶梯形式

编码风格

2、 数据说明

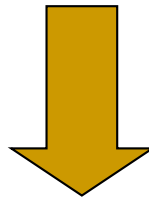
- 在设计阶段已经确定了数据结构的组织及其复杂性。在编写程序时，则需要注意数据说明的风格。
- 为了使程序中数据说明更易于理解和维护，必须注意以下几点：
 - 数据说明的次序应该标准化。
 - 说明语句中变量安排有序化。
 - 使用注释说明复杂数据结构。

编码风格

3、语句构造

- 语句构造力求简单、直接，不能为了片面追求效率而使语句复杂化。
 - 在一行内只写一条语句；
 - 尽量避免复杂的条件测试；
 - 尽量减少对“非”条件的测试；
 - 避免大量使用循环嵌套和条件嵌套；
 - 利用括号使表达式次序清晰。

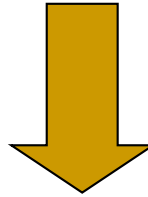
```
/* ADD AMOUNT TO TOTAL */  
TOTAL = AMOUNT+TOTAL
```



```
/* ADD MONTHLY-SALES TO ANNUAL-TOTAL */  
TOTAL = AMOUNT+TOTAL
```



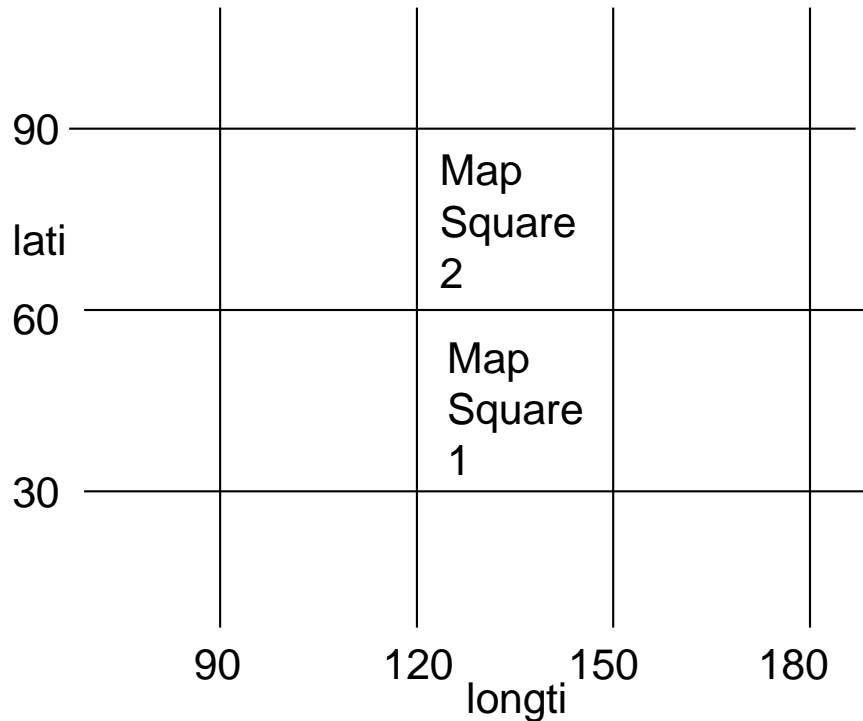
$(A < -17) \text{ANDNOT} (B \leq 49) \text{OR} C$



$(A < -17) \text{ AND NOT } (B \leq 49) \text{ OR } C$



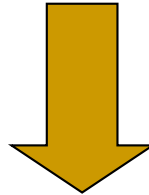

```
if(lati >30 && longti>120){if(lati <=60 && longti <=150) mapSquareNo=1;  
else if (lati <=90 && longti <=150) mapSquareNo=2; else print "Not on  
the map";} else print "Not on the map";
```



```
if (lati > 30 && longti > 120)  
{  
    if (lati <= 60 && longti <= 150)  
        mapSquareNo=1;  
    else  
        if (lati <= 90 && longti <= 150)  
            mapSquareNo=2;  
        else  
            print "Not on the map";  
}  
else  
    print "Not on the map";
```

```
if (longti > 120 && longti <= 150 && lati > 30 && lati <= 60 )  
    mapSquareNo=1;  
else  
    if (longti > 120 && longti <= 150 && lati > 60 && lati <= 90 )  
        mapSquareNo=2;  
    else  
        print "Not on the map";
```

INTEGER size, length, width, cost, price



INTEGER cost, length, price, size, width



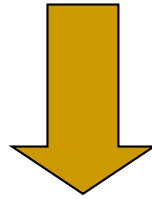
```
FOR I:=1 TO N-1 DO BEGIN T:=I;  
  FOR J:=I+1 TO N DO  
    IF A[J]<A[T] THEN T:=J; IF T<>I  
    THEN BEGIN WORK:=A[T];  
      A[T]:=A[I]; A[I]:=WORK; END END;
```



```
FOR I:=1 TO N-1 DO  
  BEGIN  
    T:=I;  
    FOR J:=I+1 TO N DO  
      IF A[J]<A[T] THEN T:=J;  
    IF T<>I THEN  
      BEGIN  
        WORK:=A[T];  
        A[T]:=A[I];  
        A[I]:=WORK;  
      END  
    END;  
  END;
```

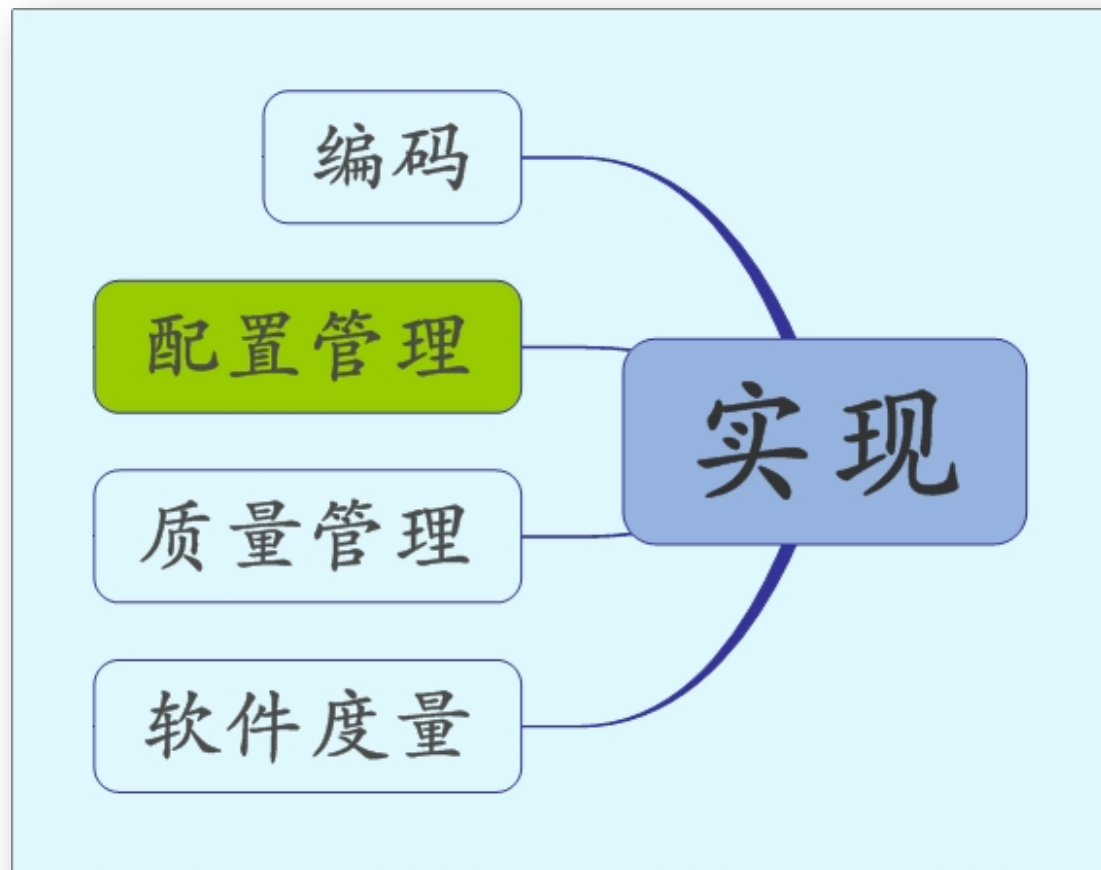


```
if ( !(char < '0' || char > '9') )
```



```
if (char >= '0' && char <= '9')
```





No matter where you are in the system life cycle, the system will **change**, and the desire to change it will persist throughout the life cycle.

—*First Law of System Engineering*

概述

- 开发软件系统的过程中，变更是不可避免的
 - 用户要变更需求 ← 知道了更多他们想要的功能
 - 开发人员要变更技术方法 ← 知道了哪个技术方法更好
 - 项目管理人员要变更项目策略 ← 知道了哪种管理方法更好
 - **Why?**

概述

- 软件配置管理（**S**oftware **C**onfiguration **M**anagement）
 - 变更管理
 - 一组管理变更的活动（umbrella activity）
 - 贯穿于整个软件过程中的普适性活动
 - “如果你不控制变更，那么变更将控制你”
 - 一个未受控制的变更流可以很容易的将一个运行良好的软件项目带入混乱，结果会影响软件质量并且会推迟软件交付。
 - 目标
 - 每个工作产品都可以标识、跟踪和控制
 - 每个变更都可以跟踪和分析
 - 每个需要知道变更的人员都得到通知

一个SCM场景

- 一个典型的SCM场景包括
 - 项目经理：管理软件项目组
 - 配置管理员：负责配置管理过程
 - 软件工程师：负责开发、维护软件产品
 - 用户：使用产品
- 由4人组成的团队开发的15000行代码的小型软件
- 项目经理的角色和任务
 - 确保在预定的期限内开发完成
 - 控制开发的进度，识别问题并对其做出反应
 - 生成并分析软件系统状态报告

一个SCM场景

- 配置管理员的角色和任务
 - 确保代码的开发、变更和测试过程能够被跟踪
 - 负责建立正式的变更机制，包括变更请求、评估变更和授权变更
- 软件工程师的角色和任务
 - 高效地工作
 - 有效地沟通和协调
 - 通过配置管理系统上传、下载代码，解决代码冲突
- 用户的角色和任务
 - 遵守正式的变更请求过程

一个SCM场景

- SCM系统支持所有的角色和任务
 - 项目经理将SCM系统看作审核机制
 - 配置管理员将SCM系统看作控制、跟踪机制
 - 软件工程师将SCM系统看作变更、构建和访问控制机制
 - 用户将SCM系统看作质量保证机制

SCM概念

- 软件配置（**S**oftware **C**onfiguration）
 - 计算机程序（源代码和可执行程序）
 - 描述计算机程序的文档（针对技术开发者和用户）
 - 数据（包含在程序内部和程序外部）
 - 在技术文档中明确说明最终组成软件产品的功能或物理属性
 - 包含了所有在软件过程中产生的信息
- 其中每一项就称为一个软件配置项（**S**oftware **C**onfiguration **I**tem）
 - 一个文档、一个全套的测试用例或一个已命名的程序构件
 - 特定版本的编辑器、编译器和其他CASE工具

基线

■ 基线（baseline）

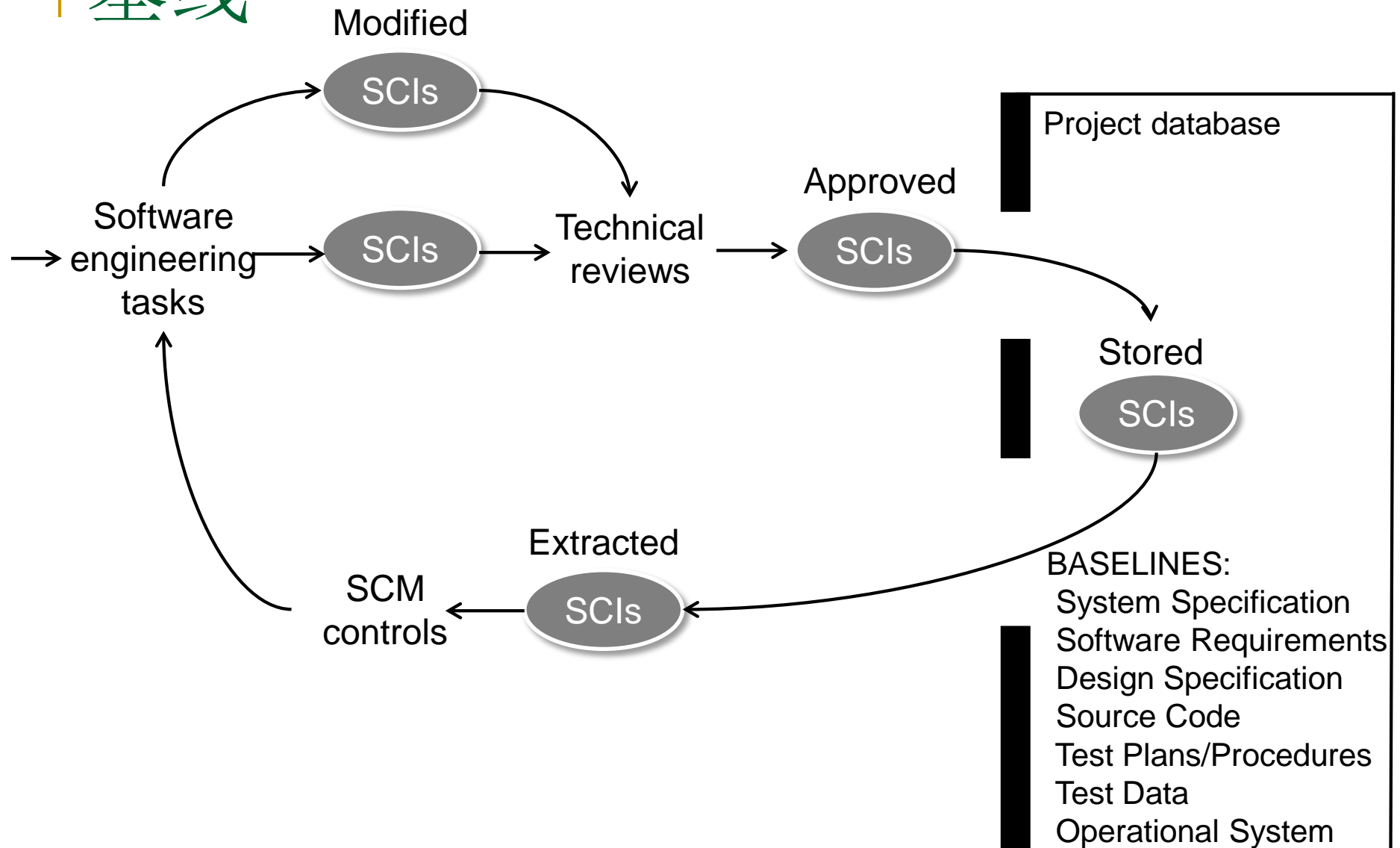
A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

—*IEEE Std. No.610.12-1990*

基线

- 基线是评审过的一个或多个SCI，每一个基线都是下一步开发的出发点和基础，是软件开发中的里程碑。
 - 设计模型已经文档化、评审过，错误已被发现并修改，已被认可了，就成为一条基线
- 在SCI变成基线之前，变化可以迅速而非正式地进行。一旦基线已经建立，变化可以进行，但是，必须应用特定的、正式的规程来评估和验证每个变化。

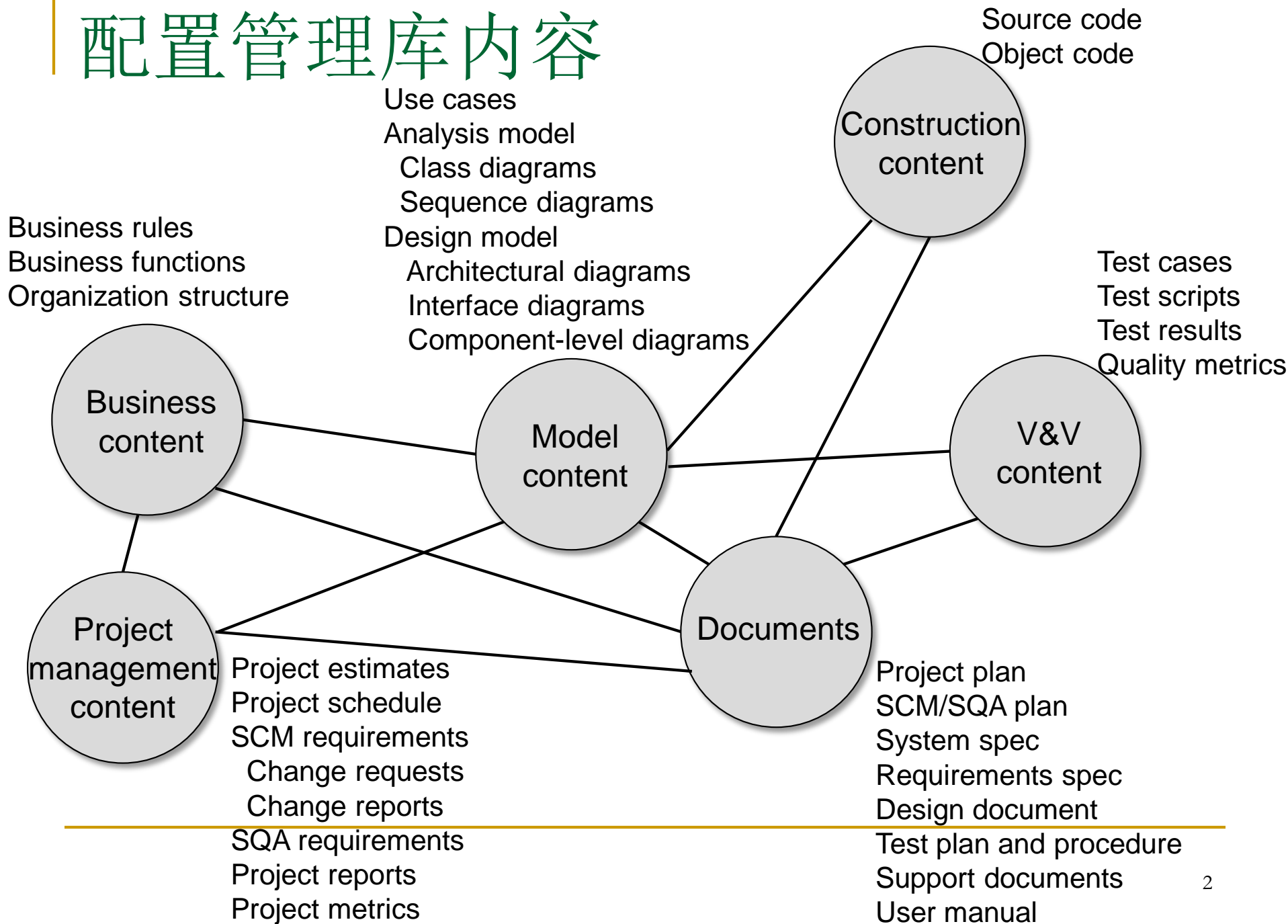
基线



配置管理库（SCM Repository）

- 配置管理库也称受控库，用于存储软件配置项以及相关配置管理信息。
- 配置管理库特征
 - 版本控制：保留所有历史版本，能够回溯
 - 关联跟踪
 - 一个UML类图变更了，配置管理库能够检测出相关的类和接口的描述，以及代码也需要修改，并把受影响的SCI提交给开发人员
 - 需求跟踪
 - 正向跟踪：检查SRS中的每个需求是否都能在后继工作产品中找到对应点
 - 逆向跟踪：检查设计文档、代码、测试用例等工作产品是否都能在SRS中找到出处
 - 审核跟踪：变更源（when, why, who）

配置管理库内容



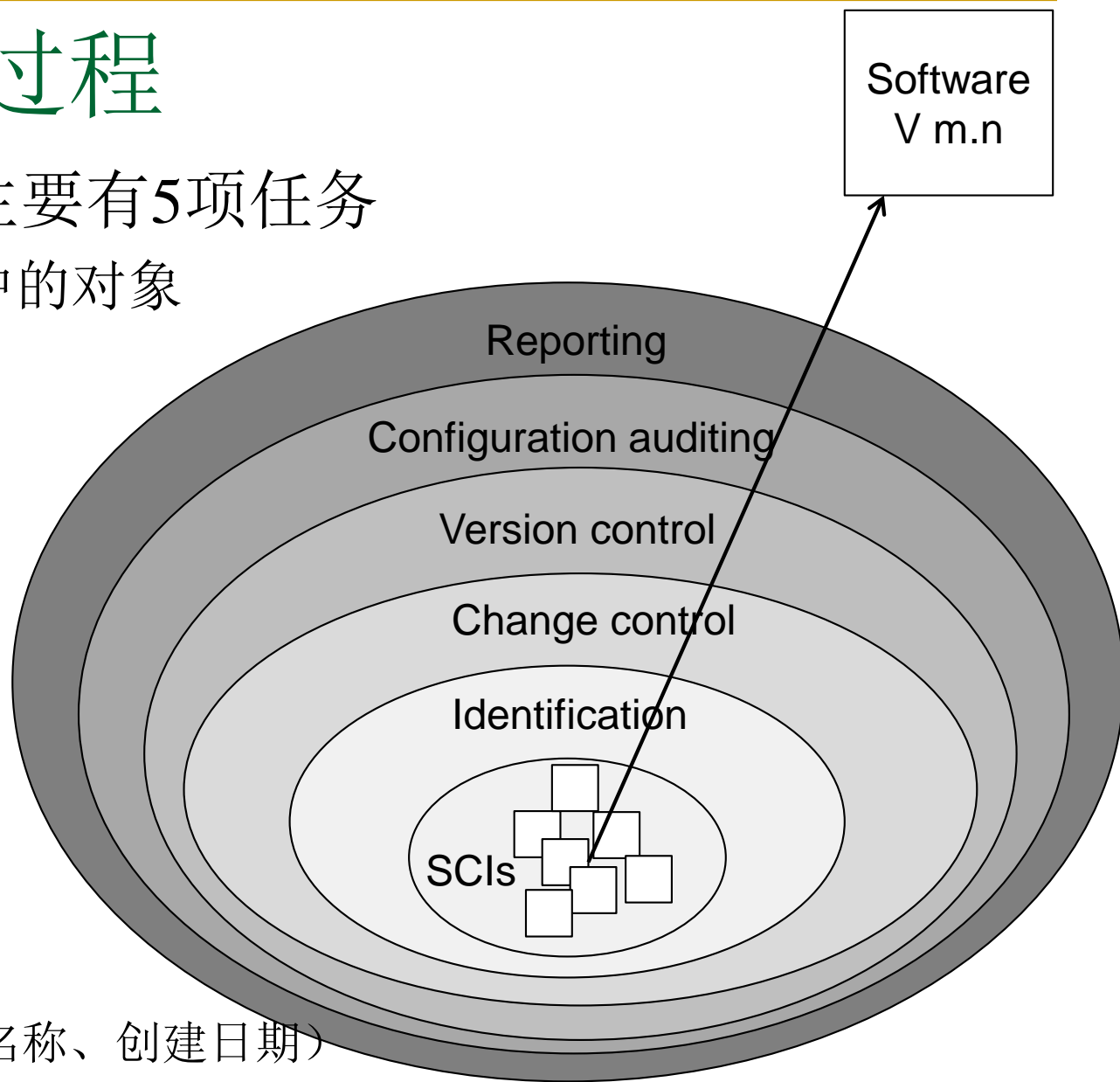
配置管理过程

■ 软件配置管理主要有5项任务

- ❑ 标识软件配置中的对象
- ❑ 变更控制
- ❑ 版本控制
- ❑ 配置审核
- ❑ 状态报告

■ 例：当一个新的SCI被创建时

- ❑ 标识该SCI
- ❑ 无变更控制
- ❑ 生成一个版本号
- ❑ 创建SCI的记录（名称、创建日期）
- ❑ 报告给需要知道的人



标识

■ 标识软件配置中的对象

- 单独命名每个配置项

- 标识出两类对象

 - 基本对象：软件工程过程中产生的信息单元（源代码、测试用例集）

 - 聚合对象：基本对象和其他聚合对象的集合

- 每个对象都有一组能唯一标识它的特征

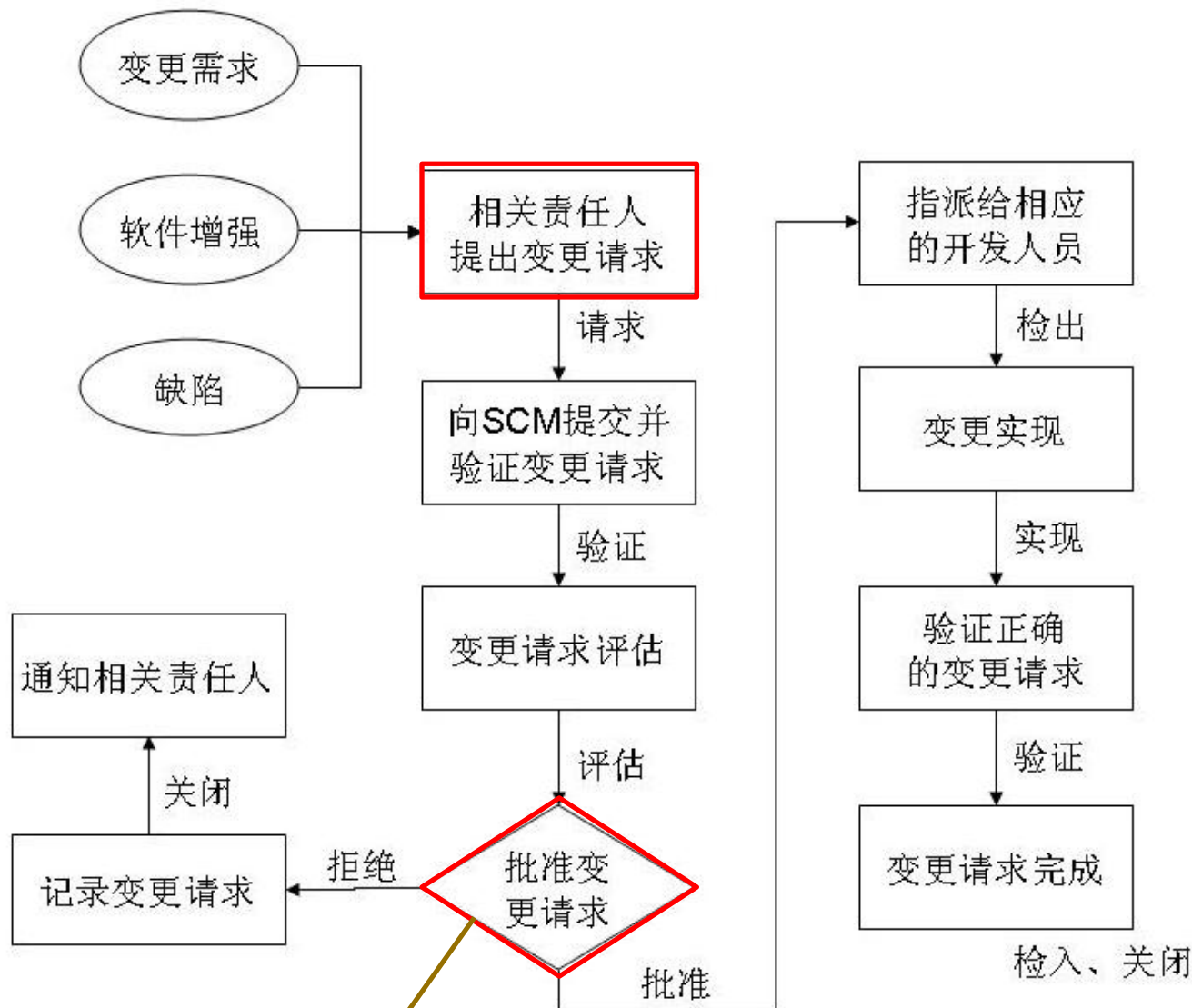
 - 名称：无二义性地标识该对象的一个字符串

 - 描述：标识该对象所表示的SCI类型、变更、版本等的一组数据项

 - 资源表：该对象需要的实体（数据类型、变量名）

 - 实现：基本对象（“文本单元”），聚合对象（Null）

变更控制



变更控制委员会 (change control board, CCB)

变更请求单

Change Request Form

Project: VIDEO/AppProcessing

Number: 23/02

Change requester: An Jicun

Date: 20/12/15

Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: Qi Rongzhi

Analysis date: 25/12/15

Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

Change assessment: Relatively simple to implement by changing the display colour according to status. A table must be added to relate status to colours. No changes to associated components are required.

Change priority: Medium

CCB decision date: 03/01/16

Decision: Accept change. Change to be implemented in Release 1.2

决定是否变更的因素

- 考虑不执行变更的结果
 - 若变更和系统故障相关，需考虑故障的严重程度
 - 系统崩溃 & 显示的颜色不对
- 变更受益方
 - 多数用户受益？提出变更的人受益？
- 变更影响的用户数量
 - 若变更影响的用户数量较少，则优先级较低
- 执行变更的成本
 - 影响多数系统构件、变更消耗时间长
- 产品发布周期
 - 一个新版本刚刚发布，推迟变更直到计划发布下一个版本



变更记录

- 在序言性注释处增加构件的变更记录
- 编写脚本，扫描所有构件的变更记录，形成构件变更报告

```
// VIDEO project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: Zeng Tao
// Creation date: 13/11/2015
//
// © HOHai University 2015
//
// Modification history
// Version Modifier   Date           Change           Reason
// 1.0      ZengTao   11/11/2015     Add header       Submitted to CM
// 1.1      QiRz      13/11/2015     New field        Change req. R23/02
```

团队开发的版本问题

■ 场景1：小型程序开发

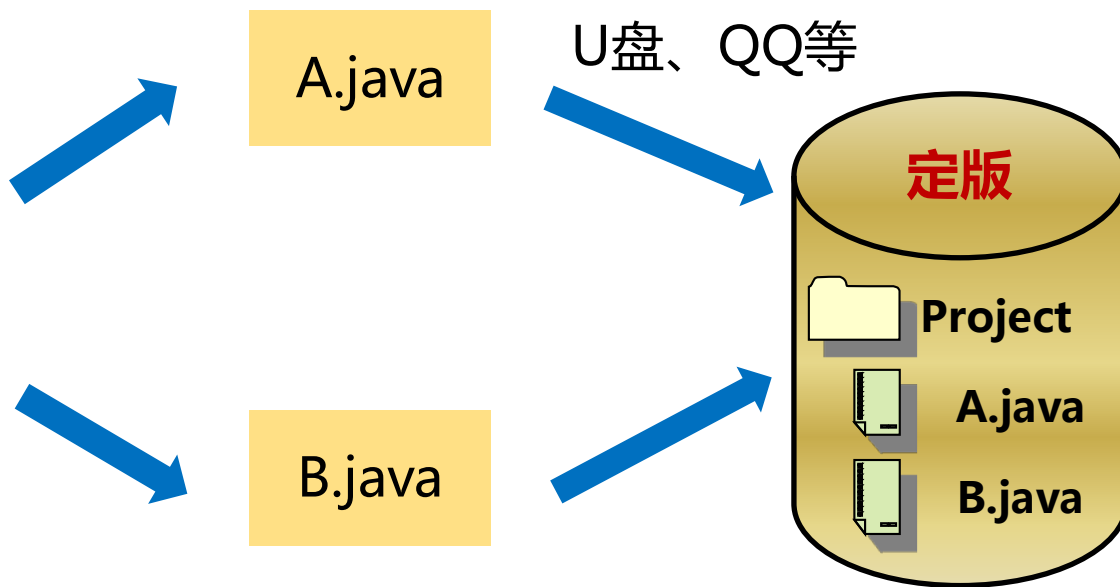
- 单独开发，两个程序员不修改同一个代码文件
- 手工合并代码



Jane

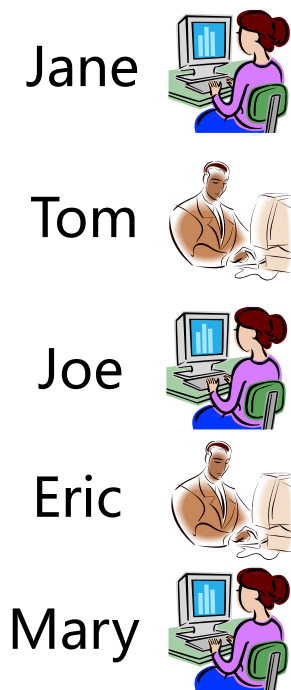


Tom



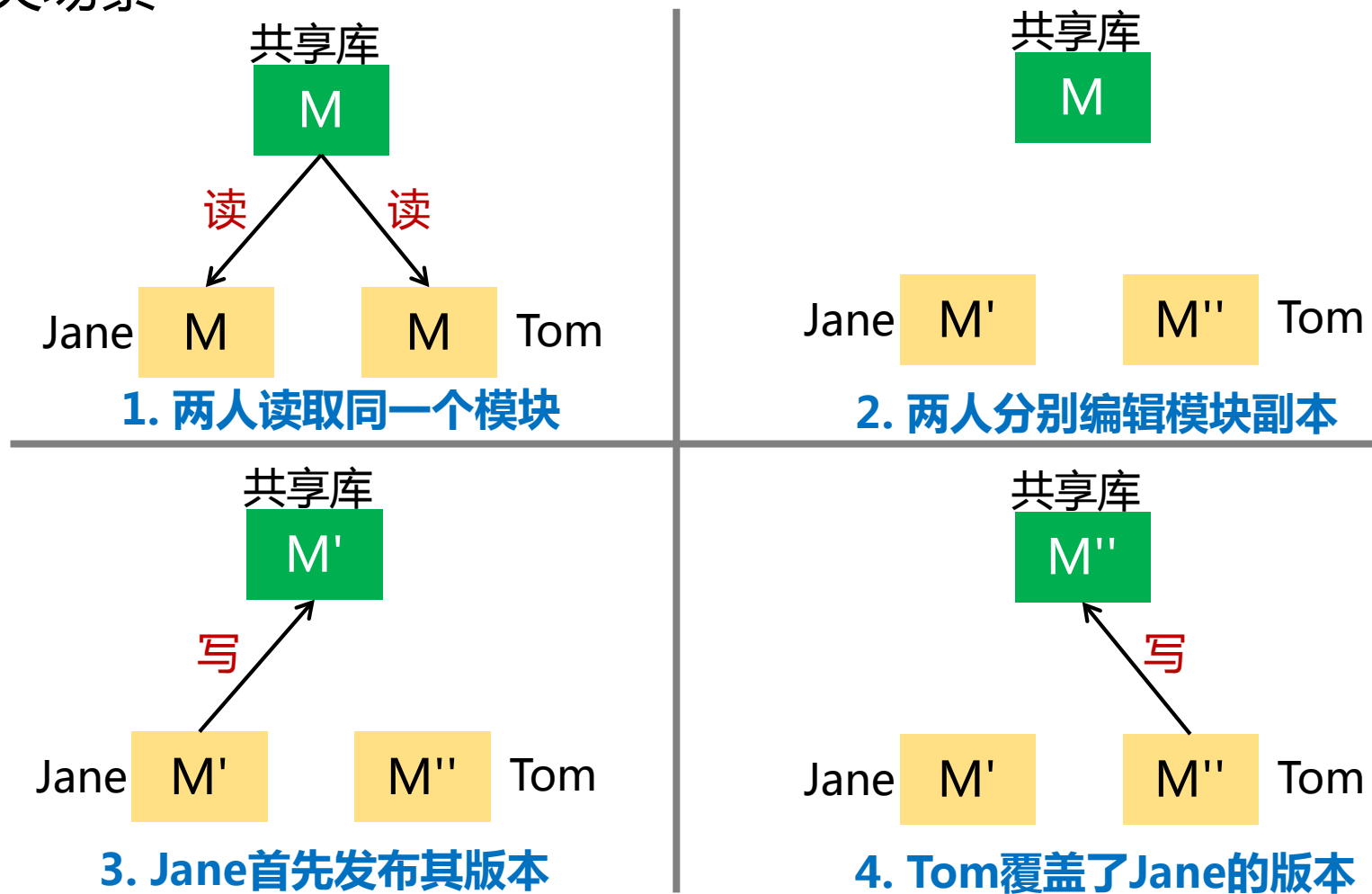
团队开发的版本问题

- 场景2：大型项目开发
 - 多人协作开发
 - 版本管理



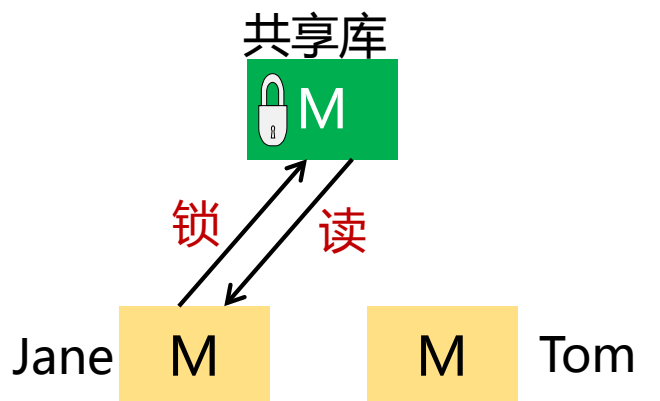
团队开发的版本问题

■ 冲突场景

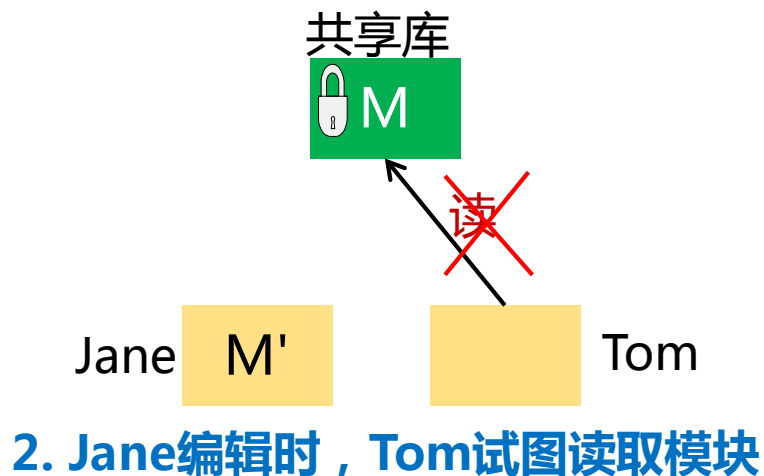


团队开发的版本问题

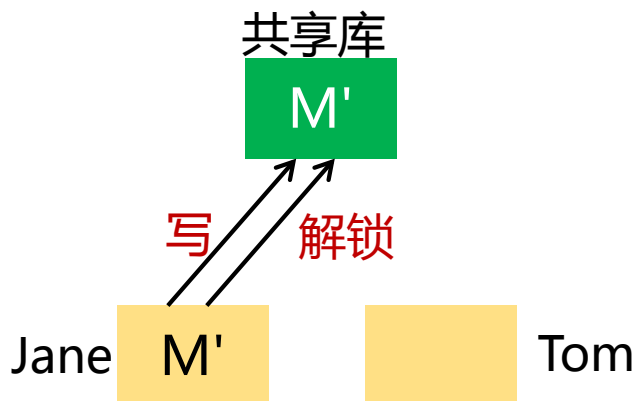
■ 冲突解决策略1：独占工作模式



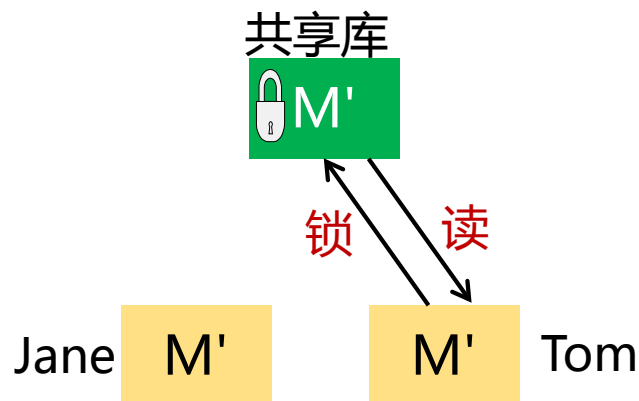
1. Jane锁定模块，并编辑副本



2. Jane编辑时，Tom试图读取模块



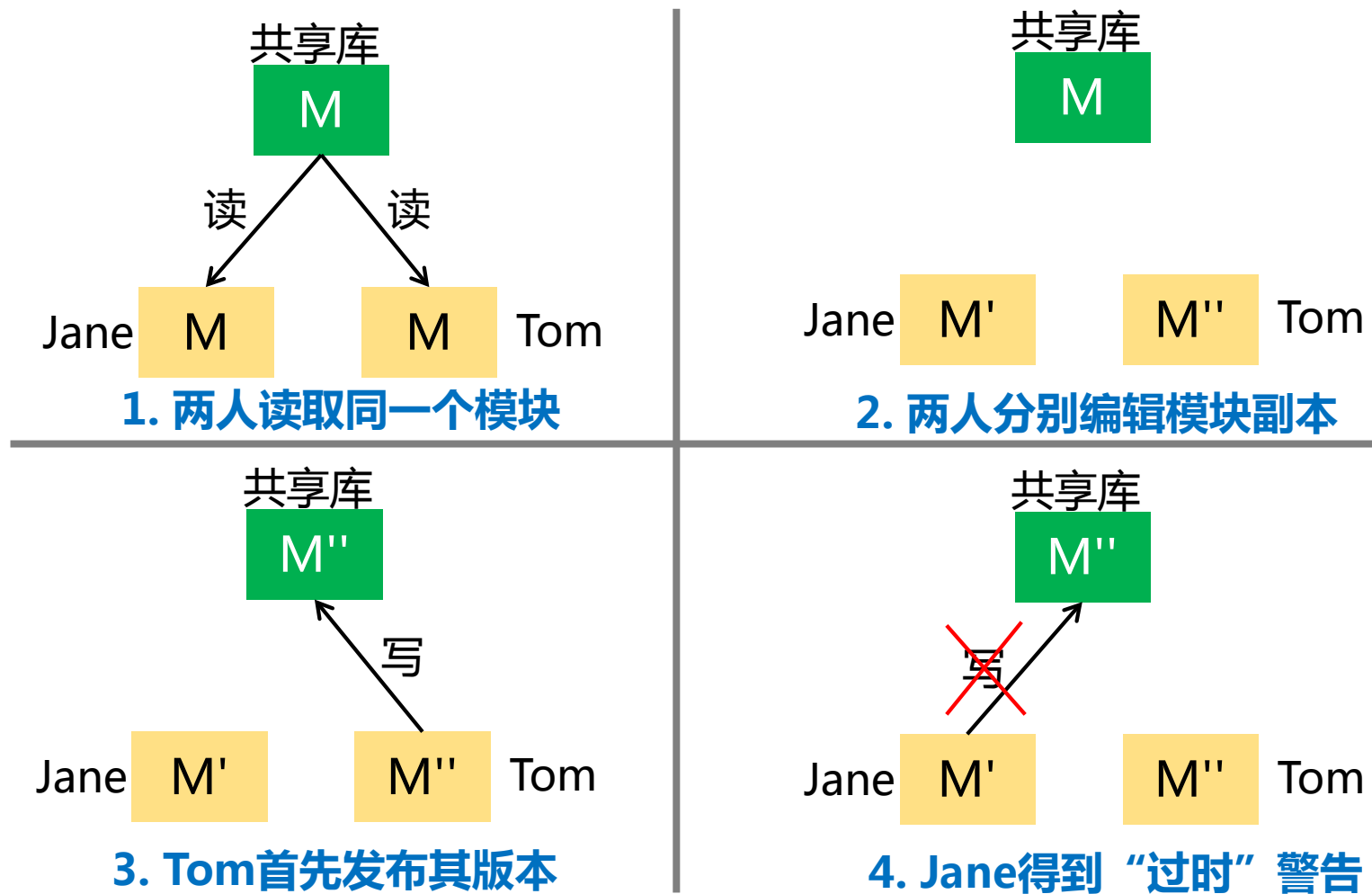
3. Jane写入版本并解锁



4. Tom加锁、读取、编辑新版本

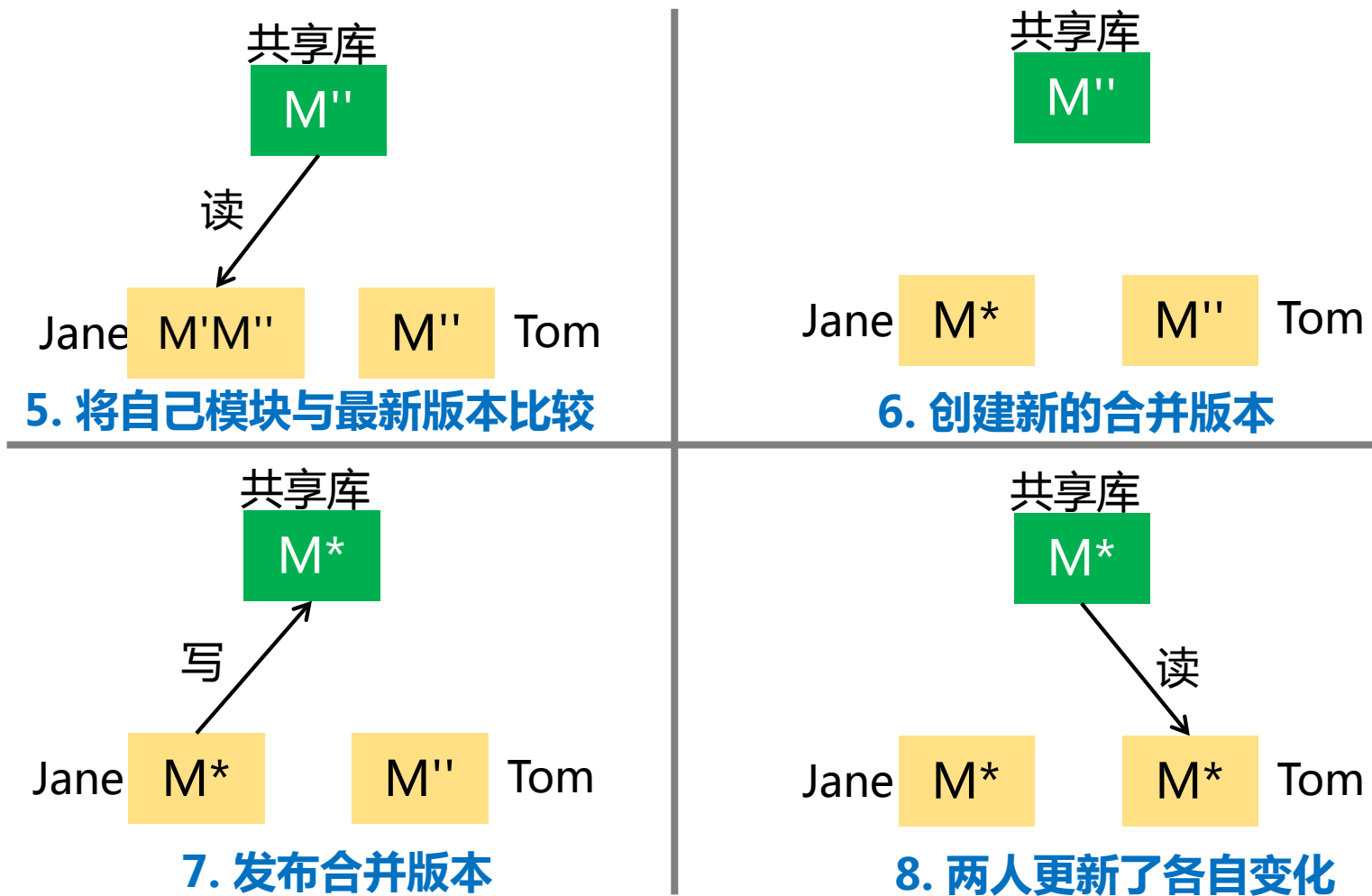
团队开发的版本问题

■ 冲突解决策略2：并行工作模式



团队开发的版本问题

■ 冲突解决策略2：并行工作模式



版本控制

■ 版本控制

- 管理在软件工程过程中所创建的构件的不同版本

Codeline (A)



Codeline (B)

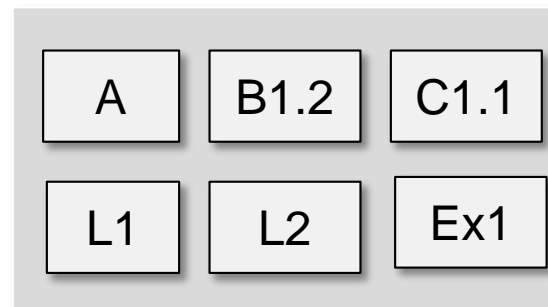


Codeline (C)

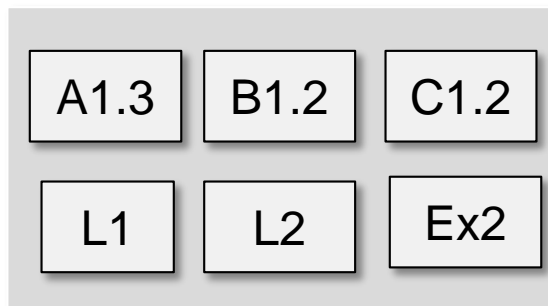


Libraries and External Components

Baseline – V1

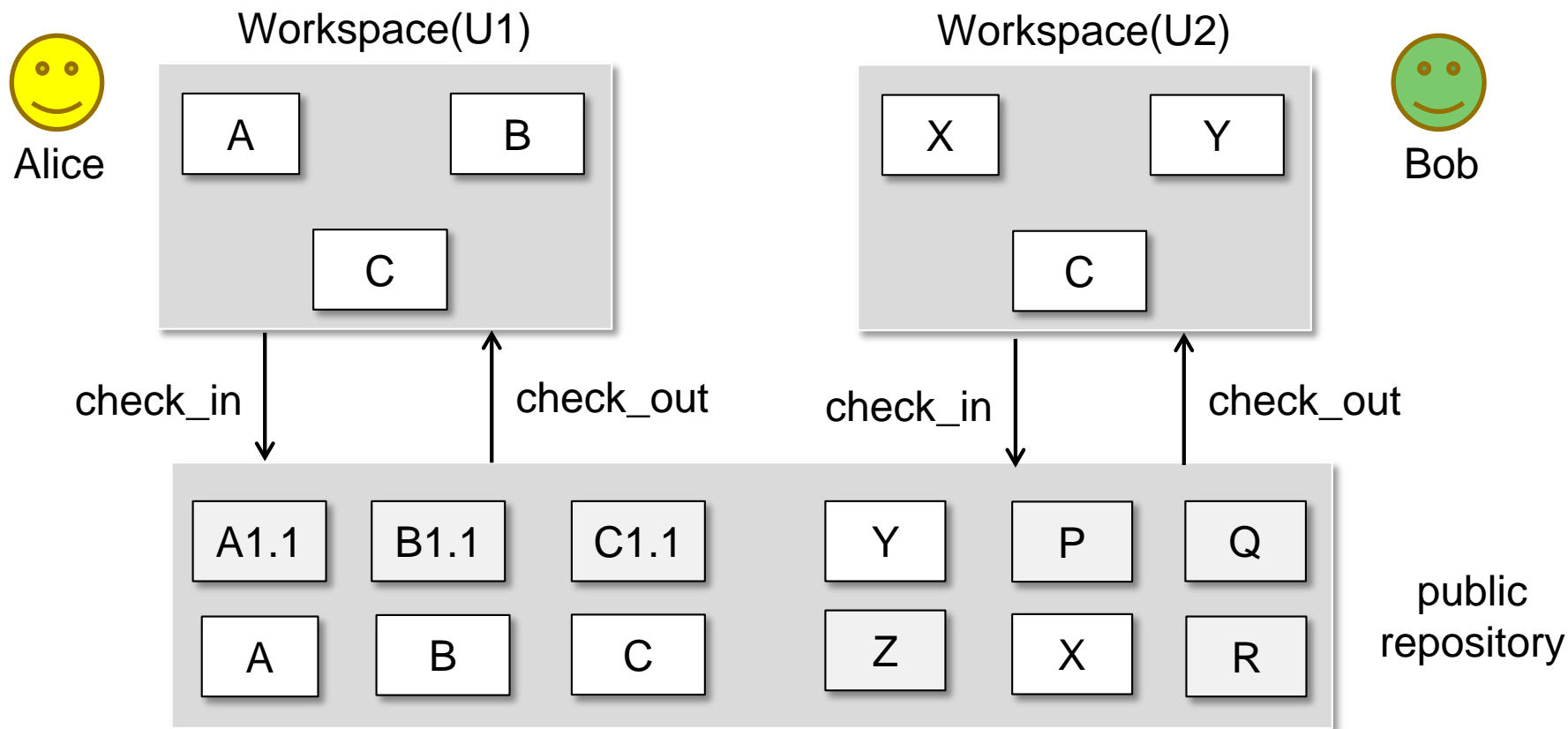


Baseline – V2



版本控制

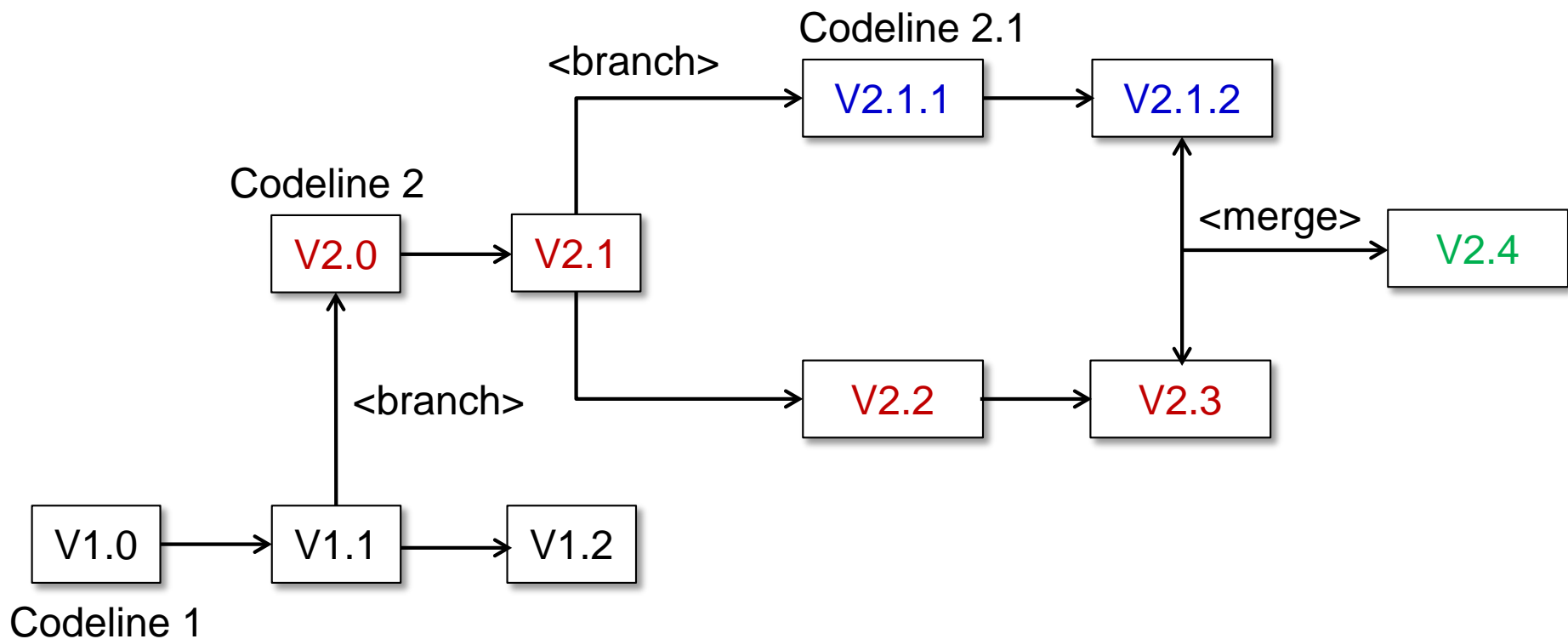
- 场景：软件开发是团队活动，不同的团队成员需要同时修改同一个构件



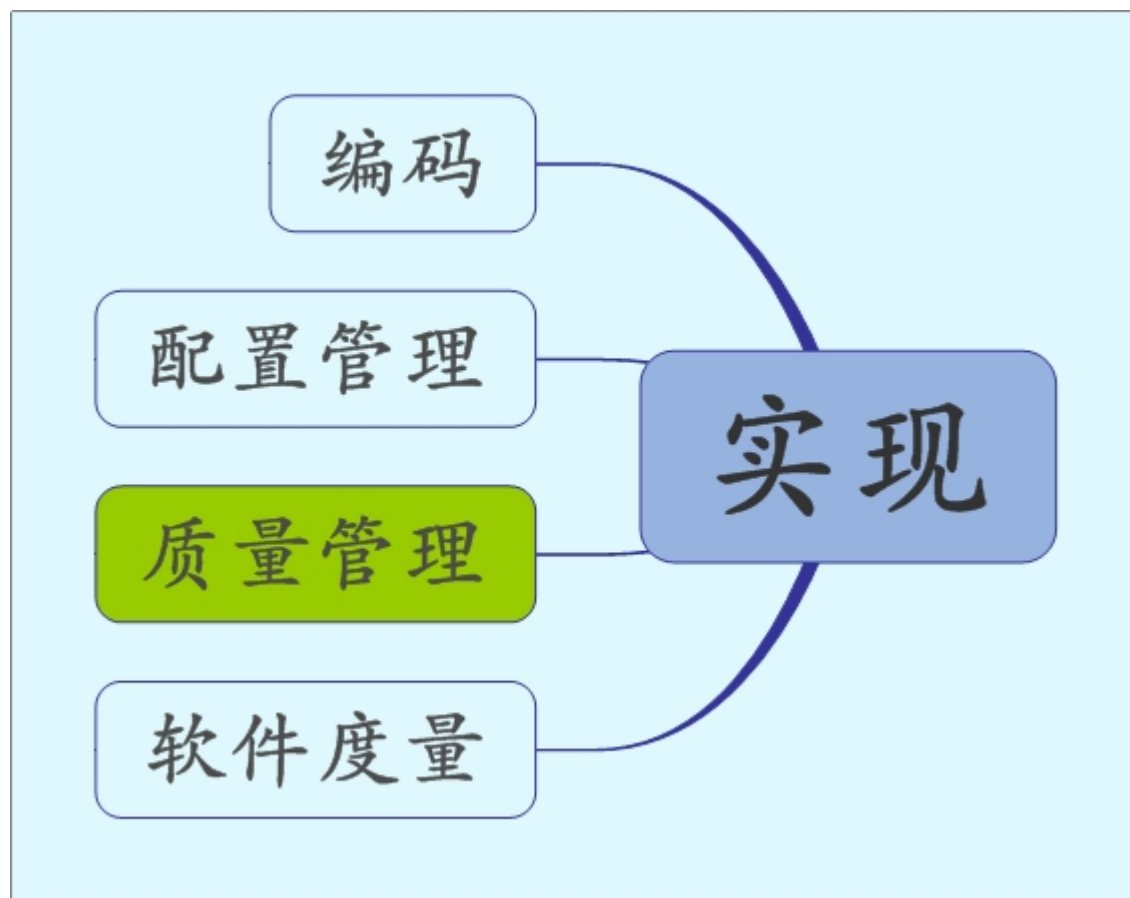
版本控制系统：CVS & Subversion

分支与合并

- 独立开发并修改同一个构件的结果是产生版本分支



- 如果所做的修改在代码的不同部位，则自动合并
- 如果在代码的相同部位做修改，则发生冲突，需要开发人员检查并解决冲突



概述

- 场景： Who is to blame?
 - 用户责怪开发人员，草率的开发过程导致低质量的软件
 - 开发人员责怪用户，不合理的交付期限、持续的需求变更、在未完成全部确认前要求交付软件
 - Who is right?
 - Both

概述

- 1960s, 大型软件开发出现了质量问题
 - 交付的软件运行速度慢、不可靠、难于维护和复用
 - “Let’s Stop Wasting \$78 Billion a Year”
 - “American businesses spend billions for software that doesn’t do what it’s supposed to do”
- 吸收制造业的质量管理技术，加上新型软件开发技术以及软件测试技术，提高软件质量
- 软件质量管理

质量管理原则

- 在组织层上，质量管理关注于建立标准组织过程
 - 质量管理小组负责定义该过程
- 在项目层上，质量管理关注于标准过程的实施
- 质量管理关注于建立项目的质量计划
 - 质量计划给出项目的质量目标
 - 定义项目使用的过程 and 标准

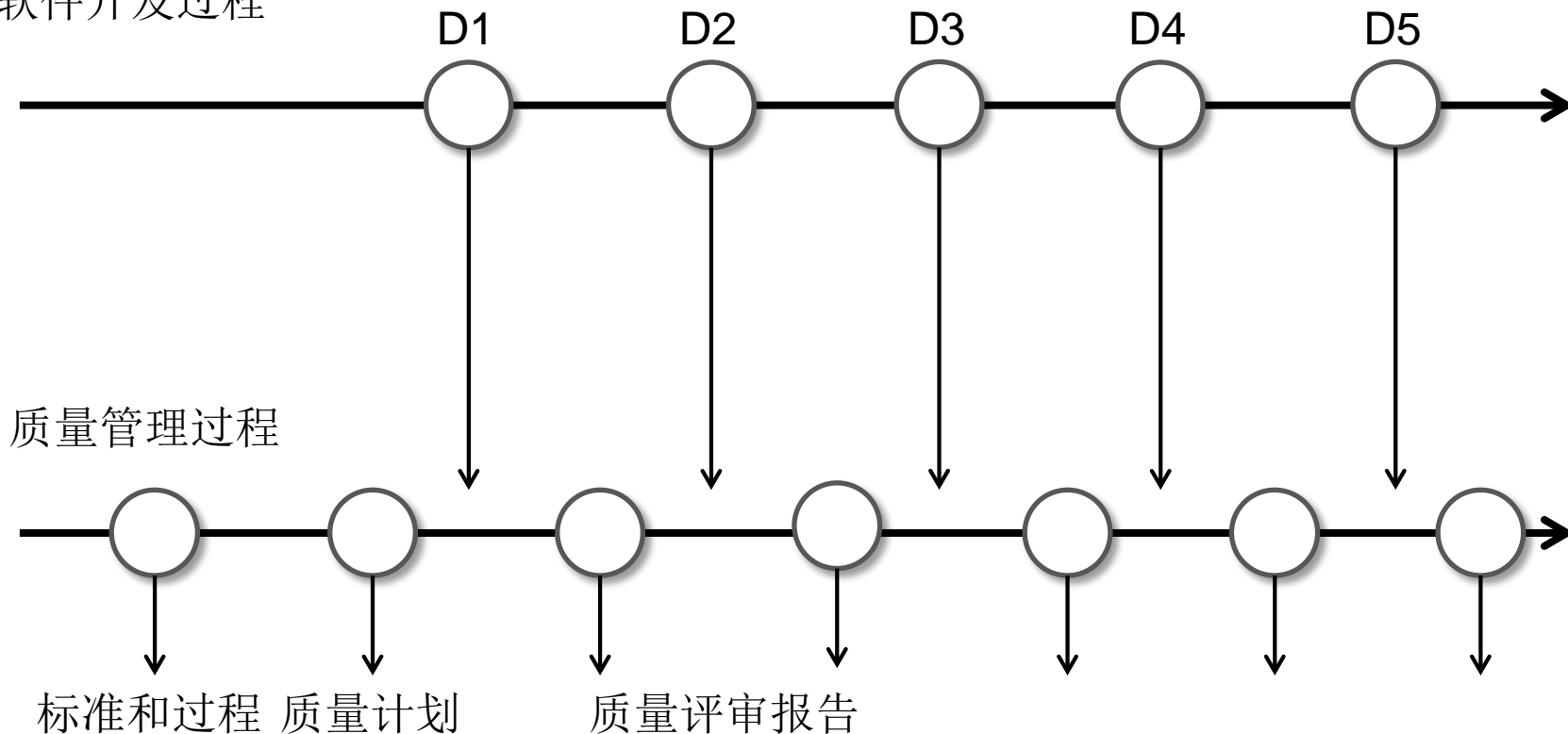
质量管理过程

■ 质量管理过程

- 检测软件开发过程，确保工作产品符合组织的标准 and 目标
- 质量管理小组独立于开发团队，直接向高层汇报

质量管理过程

软件开发过程



软件质量

An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

—*The Business Value of Quality*
Bessin

软件质量

- 有用的产品
 - 满足涉众明确提出的需求
 - 满足所有高质量软件所具有的隐含的需求

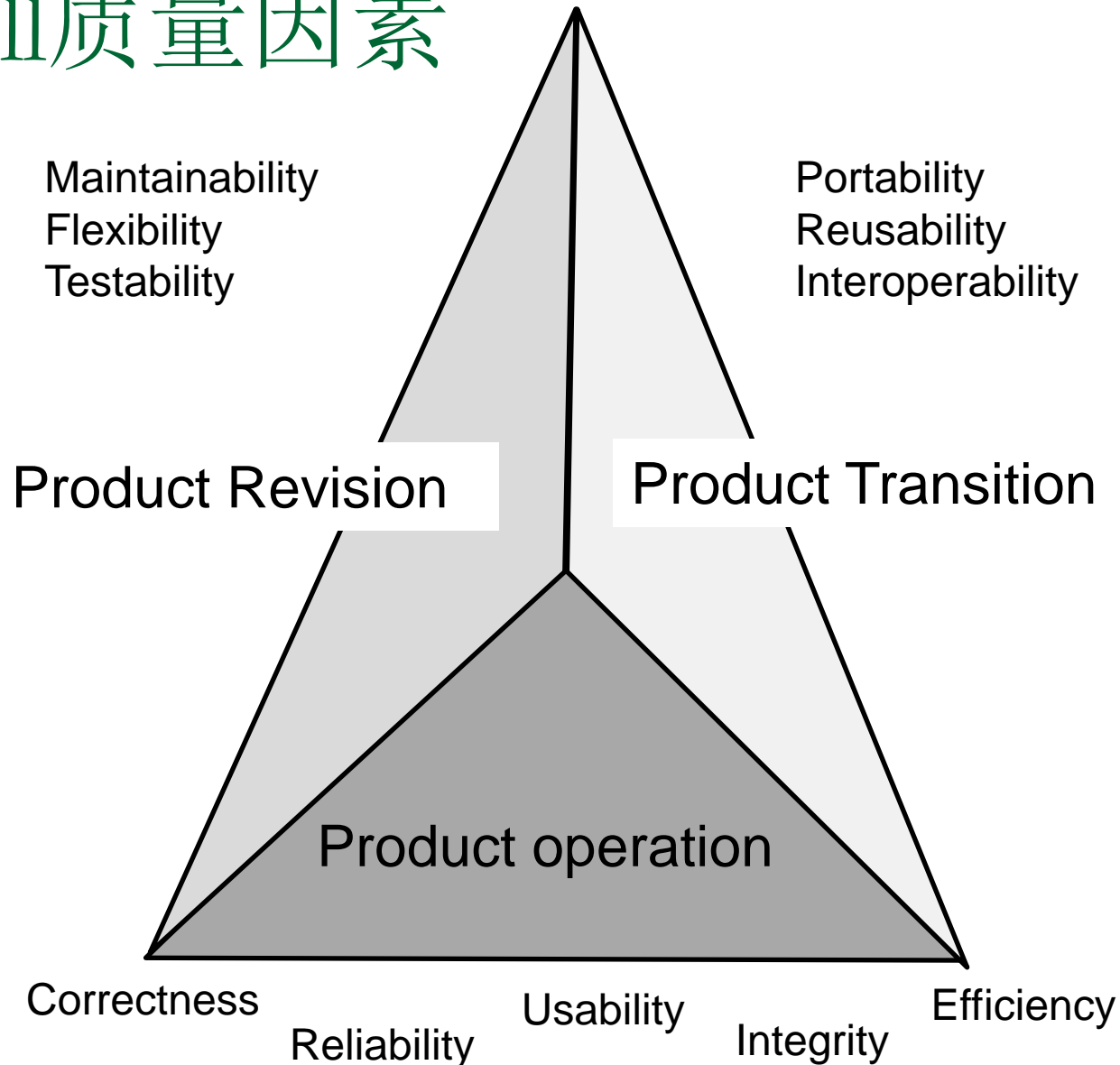


软件质量

- 为软件公司和软件的使用者增加显著的价值
 - 对软件公司而言，高质量软件需要更少的维护、更少需要修复的缺陷、无需提供更多的客户支持
 - 对软件使用者而言，高质量软件有助于推进业务过程，带来更大的收益



McCall质量因素



实现高质量软件的活动

- 软件工程方法
- 项目管理技术
- 质量控制
 - 评审模型，确保它们是完整、一致的
 - 检查代码，揭示并修复错误
 - 测试活动揭示处理逻辑、数据操作、接口通信的错误
 - 度量和反馈，调整软件过程，使得工作产品满足质量目标
- 质量保证
 - 质量管理

软件质量保证

Software quality assurance(SQA) is an umbrella activity that is applied throughout the software process.

SQA基本要素

■ 标准

- IEEE ISO
- SQA确保软件组织采纳和遵循这些标准

■ 评审和审计

- 技术评审是为了发现错误
- 审计是SQA执行的一种评审方法

■ 测试

- SQA确保制定正确的测试计划，高效地执行测试

■ 错误收集和分析

- SQA收集和分析错误数据，以便更好地理解错误是如何引入的，以及如何消除这些错误

SQA基本要素

- 变更管理
- 教育
 - SQA牵头软件过程改进
- 安全管理
 - SQA确保使用正确的过程和技术，保证软件的安全性
- 风险管理
 - SQA确保正确执行风险管理

评审

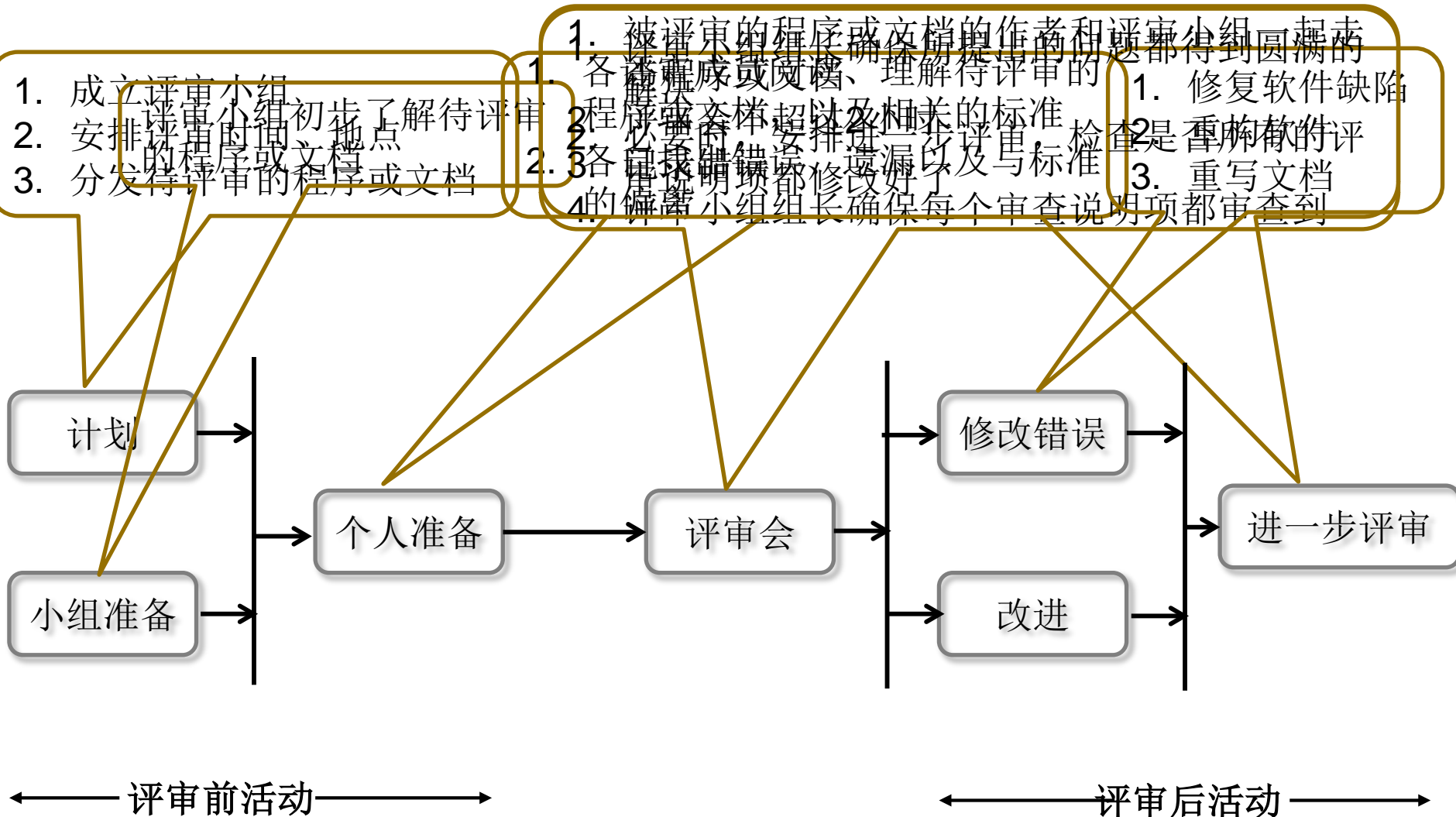
不管你有没有发现他们，缺陷总是存在，问题只是你最终发现它们时，需要多少纠正成本。评审的投入把质量成本从昂贵的、后期返工转变为早期的缺陷发现。

——卡尔·威格

技术评审（TR）

- 软件过程早期查错最有效的机制
 - 如果在早期发现错误，修改的成本就较少
- 非正式评审（informal review）
 - 桌面检查（desk check）
 - 与同事一起检查
 - 没有预先计划和准备、不开会
 - 检查清单
 - 少于1~2小时
 - 结对编程
- 正式技术评审（formal technical review, FTR）
 - 走查（walkthroughs）
 - 审查（inspections）

评审过程



代码审查（code inspection）

- Capers Jones分析了超过12,000个软件开发项目，其中使用正式代码审查的项目，发现潜在缺陷率约在60-65%之间，若是非正式的代码审查，发现潜在缺陷率不到50%。大部分的测试，发现的潜在缺陷率会在30%左右。
- 一般的代码审查速度约是每小时200~400行代码



代码审查清单

Fault class	Inspection check
数据缺陷	<ul style="list-style-type: none">• 是否所有变量在使用前都初始化了？• 是否所有的常量都命名了？• 数组的上界是否等于数组的长度或长度-1？• 如果使用了字符串，分隔符是否已确定是哪种形式？• 是否有可能出现缓存溢出？
控制缺陷	<ul style="list-style-type: none">• 每个条件选择语句中的条件判断是否正确？• 每个循环是否设置了长度和正确的终止条件？• 复合语句是否正确的加上了括号？• 在switch case语句中，是否所有可能的条件都考虑了？• 在switch case语句中，是否每个case中都使用了break？
输入/输出缺陷	<ul style="list-style-type: none">• 是否所有的输入变量都使用了？• 是否所有的输出变量在输出前都赋值了？• 所有输入是否都进行了检查（检测正确的类型，长度，格式和范围）？
异常管理缺陷	<ul style="list-style-type: none">• 是否所有可能的错误情形都考虑到了？

代码审查清单

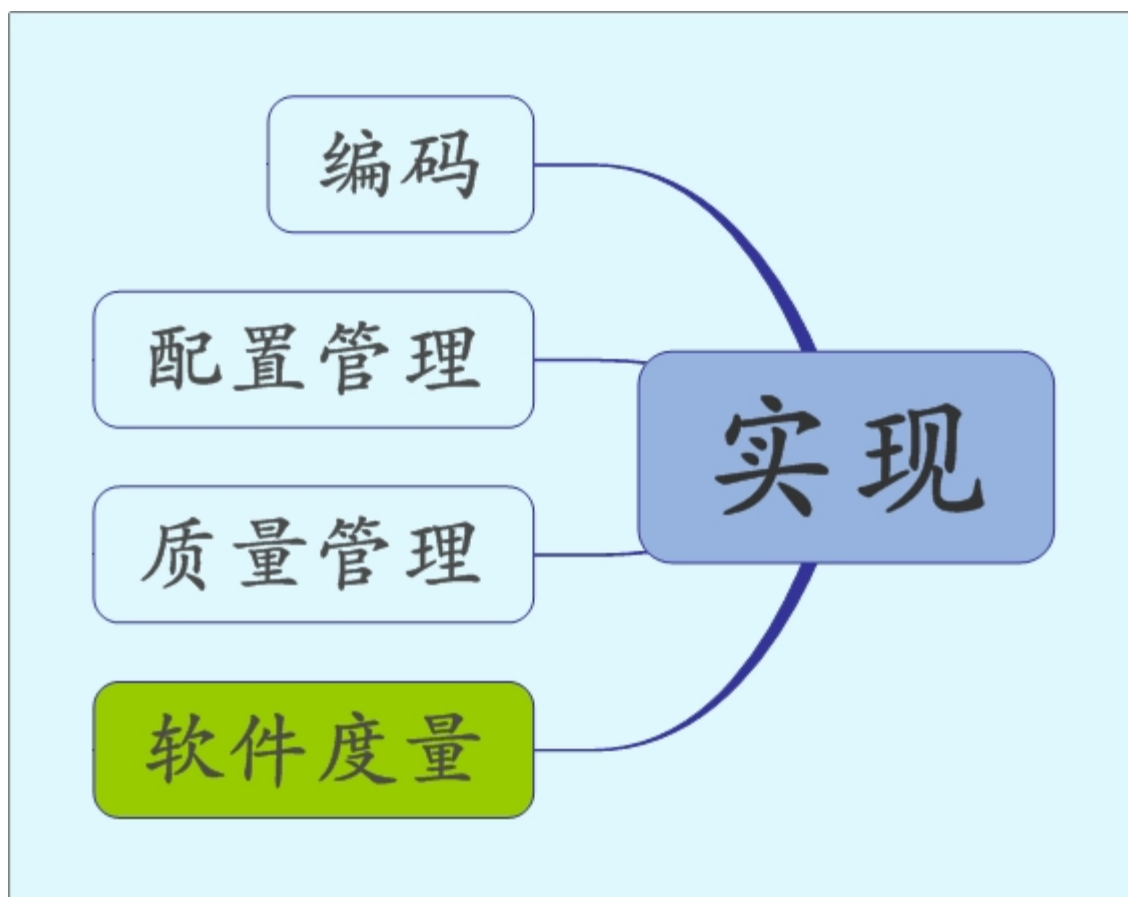
Fault class	Inspection check
接口缺陷	<ul style="list-style-type: none">• 是否所有的函数和方法的调用，都传入了正确数量的参数？• 是否所有的形参和实参的类型都匹配？• 所有参数的顺序正确吗？
常规项	<ul style="list-style-type: none">• 代码能够工作么？它有没有实现预期的功能，逻辑是否正确等。• 所有的代码是否简单易懂？• 代码符合你所遵循的编程规范么？通常包括大括号的位置，变量名和函数名，行的长度，缩进，格式和注释。• 是否存在多余的或是重复的代码？• 代码是否尽可能的模块化了？• 是否有被注释掉的代码？• 是否有可以被库函数替代的代码？• 是否有可以删除的日志或调试代码？

代码审查清单

Fault class	Inspection check
文档	<ul style="list-style-type: none">• 是否有注释，并且描述了代码的意图？• 所有的函数都有注释吗？• 对非常规行为和边界情况处理是否有描述？• 第三方库的使用和函数是否有文档？• 数据结构和计量单位是否进行了解释？• 是否有未完成的代码？如果是的话，是不是应该移除，或者用合适的标记（TODO）进行标记？

SQA措施

- 评审
- 软件测试
- **程序正确性证明 (correctness proof)**
 - 证明程序能够完成预定的功能（满足规格说明）
 - Floyd不变式断言法
 - Hoare规则公理法
 - Dijkstra弱谓词变换方法
 -



项目案例

■ 人物



小王：软件项目负责人



老王：公司技术总监

项目案例

- 在项目策划阶段的碰头会上
 - 老王问小王项目开发估计需要多少时间，多少成本？
 - 小王回答说“时间估计不会太长，成本也在一个可接受的范围之内”，老王显然对这种回答不满意，他希望能够得到一个较为准确定量性的描述。
 - 经过一番考虑后，小王回答说“时间7—8个月，成本需40—45万”，老王显然对这种回答也不满意，况且用户要求在6个月内完成项目。于是他进一步问道“你是如何得到这组数据”，小王显然没有准备，也没有充分的依据，于是他哑口无言。

项目案例

- 在制定项目计划时
 - 小王不知如何预测项目可能所需的工作量？
 - 小王不知如何预测项目可能所需的成本？
 - 小王不知所制定的计划是否可行和科学？
 - 因此，小王尽管制定了软件开发计划，但对于该计划能否得到有效的实施、实施能否遵循计划执行没有足够的信心

项目案例

- 项目已进展了2个月，各个方面进展尚可，在某周的碰头会上，老王继续向小王发问
 - “目前软件质量如何？”，小王回答道“不错”
 - 老王对这种回答不满意，他希望能够得到一个较为准确定量性的描述，但是小王又没有办法给他一个更加确切的答复，实际上连他自己也没有办法说清楚目前软件产品的质量情况，因为他只有直观的、定性了解。

概述

- 任何工程化的工作都需要度量
 - 准确了解工程的实施情况
- 软件工程需要定量、科学的描述
 - 实施前：估算成本和工作量，辅助制定软件项目的计划
 - 实施过程中：提供软件开发的可视性，跟踪和控制软件项目的开发，评估软件开发质量，进行质量控制
 - 实施完成后：对项目的实施情况进行评估，为后续项目积累经验数据
- 定量、科学的描述有助于获取软件项目以及所开发的软件的某种可视性，促进软件项目的管理
- 定量的信息描述必须在软件项目开发过程中采集

基本概念

- 对事物属性的**定性**描述
 - 个子很高, 软件的成本很高
- 对事物属性的**定量**描述
 - 个子有1.9米, 软件成本是 23.5万

基本概念

- 软件度量（**Metrics**）是指对软件产品、软件开发过程及软件开发项目的定量描述。
- 软件度量三维度
 - 产品度量：软件开发过程中所生成的各种文档和程序
 - 过程度量：与软件开发有关的各种活动，如软件设计等
 - 项目度量：度量项目规模、项目成本、项目进度等

产品度量

- 需求模型度量
- 设计模型度量
- 程序代码度量
- 测试度量
- 维护度量

需求模型度量

- 检测需求模型，预测将来系统的规模
- 面向功能的度量
 - **功能点度量**（function point, FP）：度量系统提供的功能
 - 基于系统功能的一种规模估算方法
 - IBM的工程师Allan Albrech提出
 - 目前主要基于经验公式

功能点度量基本步骤

计算各要素
的基本计数



应用复杂度
加权因子



应用技术复
杂性因子



计算调整后
的功能点

计算各要素的基本计数

■ 5个信息量

- 外部**输入**数（Number of external inputs, EIs）
 - 用户向软件输入的项数，它们向软件提供面向应用的数据
- 外部**输出**数（Number of external outputs, EOs）
 - 软件用户输出的项数，它们向用户提供面向应用的信息
- 外部**查询**数（Number of external inquiries, EQs）
 - 查询即一次联机输入，它导致软件以联机输出方式产生某种即时响应
- 内部逻辑**文件**（Number of internal logical files, ILFs）
 - 逻辑主文件（数据的一个逻辑组合，可能是大型数据库的一部分或一个独立的文件）的数目
- 外部**接口**文件（Number of external interface files, EIFs）
 - 机器可读的全部接口（硬盘上的数据文件）的数量，用这些接口把信息传送给另一个系统



应用复杂度加权因子

加权因子 wf

信息量	计数		简单	平均	复杂	
EIs	a_1	×	3	4	6	$= a_1 * wf$
EOs	a_2	×	4	5	7	$= a_2 * wf$
EQs	a_3	×	3	4	6	$= a_3 * wf$
ILFs	a_4	×	7	10	15	$= a_4 * wf$
EIFs	a_5	×	5	7	10	$= a_5 * wf$

未调整的功
能点数UFP

$$\sum_{i=1}^5 a_i * wf$$

应用技术复杂性因子

序号	F_i	技术因素
1	F_1	数据通信
2	F_2	分布式数据处理
3	F_3	性能标准
4	F_4	高负荷的硬件
5	F_5	高处理率
6	F_6	联机数据输入
7	F_7	终端用户效率
8	F_8	联机更新
9	F_9	复杂的计算
10	F_{10}	可重用性
11	F_{11}	安装方便
12	F_{12}	操作方便
13	F_{13}	可移植性
14	F_{14}	可维护性

■ F_i 的取值(0,1,2,3,4,5)

- 0-没有影响
- 1-偶有影响
- 2-轻微影响
- 3-平均影响
- 4-较大影响
- 5-严重影响

技术复杂性因子

$$TCF = 0.65 + 0.01 \times \sum_{i=1}^{14} F_i$$

其中，0.65和 0.01都是经验常数

$$0.65 \leq TCF \leq 1.35$$



计算调整后的功能点

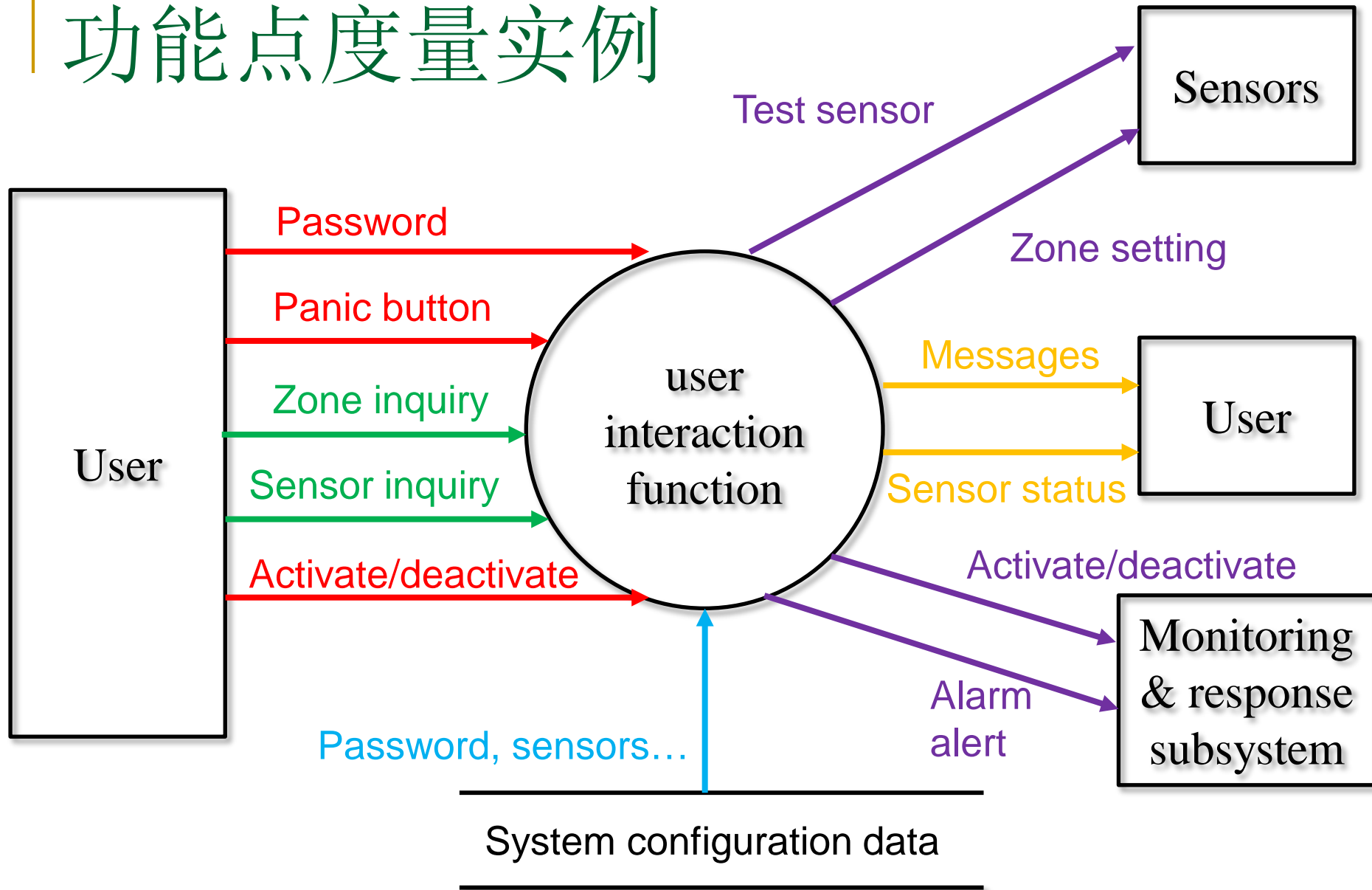
■ $FP = UFP \times TCF$

$$= \sum_{i=1}^5 a_i * wf \times \left[0.65 + 0.01 \times \sum_{i=1}^{14} F_i \right]$$

- 主要靠经验公式
- 在判断加权因子复杂级别和技术因素的影响程度时，存在着相当大的主观因素。



功能点度量实例



功能点度量实例

加权因子 wf

信息量	计数		简单	平均	复杂		
EIs	3	×	3	4	6	=	9
EOs	2	×	4	5	7	=	8
EQs	2	×	3	4	6	=	6
ILFs	1	×	7	10	15	=	7
EIFs	4	×	5	7	10	=	20
未调整的功能点数UFP							→ 50

功能点度量实例

- 假设该实例是一个中等复杂度的产品
- $\Sigma(F_i)=46$

$$FP = UFP \times TCF = 50 \times [0.65 + (0.01 \times 46)] = 56$$

代码行（Lines of Code）

- 代码行技术是比较简单的定量估算软件规模的方法。
- 依据以往开发类似产品的经验和历史数据，估计实现一个功能所需要的源程序行数。
- 当有以往开发类似产品的历史数据可供参考时，估计出的数值还是比较准确的。把实现每个功能所需要的源程序行数累加起来，就可得到实现整个软件所需要的源程序行数。

估算方法:

- 由多名有经验的软件工程师分别做出估计。
- 每个人都估计程序的最小规模(a)、最大规模(b)和最可能的规模(m),
- 分别算出这3种规模的平均值、和之后, 再用下式计算程序规模的估计值:

$$L = \frac{\bar{a} + 4\bar{m} + \bar{b}}{6}$$

单位:

LOC或KLOC。

LOC & FP

Programming language	LOC per FP			
	Avg.	Median	Low	High
Ada	154	-	104	205
ASP	56	50	32	106
Assembler	337	315	91	694
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	704
FORTRAN	90	118	35	-
HTML	43	42	35	53
Java	55	53	9	214
JSP	59	-	-	-
Javascript	54	55	45	63