《软件测试课程设计》讲稿（第 2 章）

第 1–2 周 · 各 2 学时（90 分钟）

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 2 章　软件测试基本概念

第 2 周 · 2 学时（90 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》

《软件测试课程设计》第 2 章 讲稿（第 2 周 · 90 分钟）

> 课程：软件测试课程设计（154442008） | 广东金融学院 计算机学院 | 专业选修
>
> 章节：第 2 章 软件测试基本概念 —— 软件缺陷、软件质量、测试分类与层次
>
> 学时：2 学时（90 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》，清华大学出版社 2022
>
> 配套课件：逐页还原版 87 页（`lecture2/`，云端 http://111.231.0.226:2345/lecture2/index.html）
>
> 本讲重点：测试分类与测试级别、静态测试与动态测试、黑盒测试与白盒测试
>
> 本讲难点：不同测试级别的目标与执行主体；黑盒与白盒方法的选择依据
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出软件质量的内涵与主流质量模型（ISO 9126 三层模型、Boehm 模型、ISO 25000 系列）；② 能给出软件缺陷的定义与判断准则，说明缺陷产生原因与修复成本随阶段的变化；③ 能按多个维度给测试分类（静态/动态、主动/被动、手工/自动化、脚本/探索式、黑盒/白盒/灰盒）；④ 能复述测试的四个层次及各自的任务与执行主体
能力	① 能用"是否运行程序"判断静态与动态测试；② 能针对给定功能点选择黑盒或白盒方法并说明理由；③ 能画出测试工作流程（需求分析→策略→计划→设计→执行→评估）并定位任一活动
素质（思政）	① 通过"质量＝客户满意度"与缺陷成本曲线，建立"质量即责任、越早越省"的工程观；② 通过产品评审（需求/设计评审）体会严谨求实、批评与自我批评的团队文化；③ 通过验证与确认（V&V）理解工程规范与法治意识（按标准与契约办事）

本讲要建立的一个总印象：质量是可定义、可度量的；**缺陷是质量的对立面**，而测试是用分类清晰、层次分明的活动去发现与预防缺陷。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **87 页**，是全书页数最多的一章，本讲稿为**素材全量版**（口播约 4 万字量级，≈3 小时纯朗读），**90 分钟课堂绝无可能逐页念完**。请务必按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
> 2.6、2.11、2.14、2.17、2.18、2.23、2.27、2.32、2.37、2.43、2.51、2.52、2.58、2.59、2.71、2.75、2.85。
>
> 2. **常规页快速带过（0.5–1 分钟/页）**：只讲口播第一段（结论）＋板书＋1 个提问。
>
> 3. **可课后阅读页（★）**：
> ★2.7、★2.8、★2.9、★2.12、★2.13、★2.15、★2.16、★2.19、★2.20、★2.21、★2.22、★2.62、★2.65、★2.78–2.81。
>
> 4. **练习/作业页**：2.84 作业与 2.86 实验要在课上交代清楚；2.85 思考题留作课后（答案口径见本稿该页）。

时段	页码	内容	分钟	备注
第 1 节	2.1–2.2	封面、第 1 章回顾	3	用 2 分钟建立本章地图
第 1 节	2.3–2.22	2.1 软件缺陷与质量内涵（质量模型・内部/外部/使用质量）	21	2.6、2.11、2.14、2.17、2.18 重点；★2.7–2.9、2.12–2.13、2.15–2.16、2.19–2.22 可略
第 1 节	2.23–2.32	缺陷定义・判断准则・产生・构成・分布・成本	16	2.23、2.27、2.32 必讲
——	——	**课间休息 10 分钟**	——	——
第 2 节	2.33–2.40	2.2 软件测试的分类（多维度）	10	2.37 静态/动态是核心
第 2 节	2.41–2.53	2.3 静态测试与动态测试（产品评审・静态分析・V&V）	14	2.43、2.51、2.52 是难点
第 2 节	2.54–2.57	2.4 主动/被动测试；2.5 黑盒/白盒	8	2.56 讲清选择依据
第 2 节	2.58–2.69	2.6 软件测试层次（单元→集成→系统→验收）	12	2.58、2.59 必讲；★2.62、2.65 可略
第 2 节	2.70–2.82	2.7 软件测试工作范畴（需求分析→策略→计划→设计→执行→评估）	10	2.71、2.75 重点
第 2 节	2.83–2.87	本章小结、作业、思考题、实验、结束	6	2.84、2.85、2.86 必交代

> **压缩优先级**：★2.7–2.9、2.12–2.13、2.15–2.16、2.19–2.22（质量模型细节，可布置课后阅读）→ ★2.62、2.65（示例页）→ ★2.78–2.81（脚本/探索式细节）→ 2.60、2.67、2.68（层次细节）。
>
> **2.23、2.27、2.32、2.37、2.43、2.52、2.56、2.58、2.59、2.71、2.75、2.85 是考点密集页，不要压缩。**

三、逐页讲稿

> 使用说明：每页第一段是**可直接朗读的口播**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写，不要一次写完。

【2.1】软件测试方法和技术（2 分钟）

> 同学们好，我们继续上课。上一周我们用了整整两节课，把"为什么必须做软件测试"这个问题讲透了。今天开始，我们进入第 2 章——**软件测试的基本概念**。课件上署了我的单位和姓名——广东金融学院，周宇文；这不只是课件格式，也算一份责任声明：这一章讲的质量观、缺陷观，最后要落到每一个测试人员身上。请先记住一句话：第 1 章回答"为什么测"，第 2 章回答"测什么、怎么分类、站在哪个位置测"。前者是态度，后者是概念与方法，两章合起来就是我们这门课的地基。今天我们要把"软件缺陷""软件质量""测试分类""静态与动态""黑盒与白盒""测试层次""测试工作范畴"这一串概念一个一个钉牢。请大家把教材翻到第 2 章，我们开始。

> **板书：**中间横写"第 2 章 软件测试的基本概念"；左侧竖排写"第 1 章＝为什么测（观念）"，右侧写"第 2 章＝测什么・怎么分・谁来测（概念与方法）"。

> **提问：**请用一句话说说，上一章我们反复强调的"为什么必须测试"的那个核心理由是什么？

> **预设回答：**

- 回答"因为软件一定有缺陷"——点评：方向对，但更准确的说法是"缺陷不可避免，而且缺陷越晚被发现，修复代价越高"，这才是必须测的完整理由。

- 回答"为了交付、为了不出事故"——点评：这是目的，不是理由；测试的直接价值是**尽早发现缺陷、为质量提供信息**。

- 回答"课本上这么写的"——点评：不批评，但要求补一句自己的理解，防止把课程学成背结论。

- 追问：如果缺陷不可避免，测试能把缺陷"清零"吗？——答案是"不能，测试是在降低风险，不是消灭风险"，这个口径今天要反复用。

> **易错点 / 考点：**本页是封面页，不会直接考；但课程口径要记准——本课程是《软件测试课程设计》（课程代码 154442008），专业选修、32 学时 2 学分（理论 16＋实验 16），考查课，过程性考核＋课程设计作品与答辩；教材是朱少民《软件测试方法和技术（第 4 版）》，清华大学出版社 2022 年。后面所有名词表述一律以这本教材为准。

过渡：在进入新概念之前，我们先把第 1 章的六块内容快速过一遍——下一页就是"第 1 章回顾"。

【2.2】第 1 章回顾（4 分钟）

上一章我们跑了六站，现在我一站一站带你回看，你对照自己的笔记检查一遍。第一站，**什么是软件测试**。我们给过一个可以背下来的定义：软件测试是在规定的条件下对软件进行操作，以发现程序错误、衡量软件质量，并对其是否能满足设计要求进行评估的过程。说白了，它是"为了发现缺陷而运行软件"，同时顺手给质量提供证据。第二站，**软件测试的正反两面性**。正面是"验证"——证明软件做对了它该做的事，给开发者和用户信心；反面是"发现缺陷"——带着怀疑去证伪，专挑边界、异常和你想不到的组合。这两个方向看着矛盾，其实是一枚硬币的两面。第三站，**验证软件**，也就是 Verification，回答"我们有没有把产品正确地做出来"，对的是规格和设计。第四站，**发现缺陷**，这里要提醒大家，测试的目的不是证明软件没有错，而是尽可能早地把错误找出来。第五站，**V&V**——Verification 与 Validation，前者问"做得对不对"，后者问"做的是不是用户要的"，这两句话每年都考，务必分清。第六站，**软件测试和开发的关系**，以及 **TDD 测试驱动开发**：测试不是开发的尾巴，它可以从需求阶段就并行介入，先写测试，再写实现。这六站合起来，就是第 1 章的骨架。

板书：竖排六行——① 软件测试定义 → ② 正反两面性（验证／发现缺陷）→ ③ 验证 Verification → ④ 发现缺陷 → ⑤ V&V → ⑥ 测试与开发关系／TDD。右侧留一块"待解决区"，写三个词：**缺陷？ 质量？ 分类？**

提问：Verification 和 Validation 最核心的区别是什么？请各用一句大白话说清楚。

预设回答：

- 回答"一个验证一个确认，差不多"——点评：这正是最常见的错误，两者依据不同、对象不同，不能混用；中文通常译成"验证"和"确认"，译名相近但含义不同。

- 回答"Verification 检查代码对不对，Validation 检查功能对不对"——点评：抓住了对象感，接近正确，但要补上"依据"：Verification 依据是规格、设计、接口约定；Validation 依据是用户需求和使用场景。

- 回答"两个都是测试的意思"——点评：承认它俩都在测试范畴内，但一个是"造得对不对"，一个是"造得对不对路"；用盖房子类比：Verification 是照着图纸检查墙有没有砌歪，Validation 是交房前问用户住得舒不舒服。

- 追问：如果图纸本身就错了呢？——引出"图纸错了，Verification 再严格也救不了"，这就是 Validation 必须存在的原因。

易错点 / 考点：V&V 混淆是最高频考点。判断三步法：**一看依据**（依据规格／设计＝Verification，依据用户

需求／使用场景＝Validation）；**二看提问**（"Are we building the product right?" 对的是 Verification，"Are we building the right product?" 对的是 Validation）；**三看阶段**（Validation 更靠后、更靠用户场景，常与验收测试绑定）。考试形式多为选择、判断或名词解释，也可能出在案例分析里让你指出某活动属于哪一类。记忆口诀：**"验证看规格，确认看用户"**。

> **过渡：**第 1 章给了我们"为什么测"，那么"缺陷"到底怎么定义、"质量"到底怎么衡量？这就是第 2 章要解决的问题。我们先看本章的完整地图——下一页。

【2.3】2.1 软件缺陷（3 分钟）

这一页是本章的地图，七个小节念一遍：2.1 软件缺陷、2.2 软件测试的分类、2.3 静态测试与动态测试、2.4 主动测试与被动测试、2.5 黑盒测试与白盒测试、2.6 软件测试层次、2.7 软件测试工作范畴。其中 2.1、2.2 是地基，2.5、2.6 是考试与课程设计重点。页面上还列了整门课更远的框架：后面有**四类测试方式**——静态与动态、主动与被动、手工与自动化、基于脚本与探索式；然后是**结构化测试方法**，包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖、MC/DC 覆盖和基本路径覆盖；最后是**基于需求的测试方法**，包括等价类划分、边界值分析、判定表、因果图、Pairwise（工具 ATCS，据课件数据）、正交试验法和基于场景的方法。记住：先立地基——什么算缺陷、从哪来、修它多贵。

> **板书：**板书列写七个节标题（2.1～2.7），在 2.1、2.2 下画双横线标"地基"；右侧另起一栏预先写下三个大标题：**四、测试方式｜五、结构化测试方法（ST）｜六、基于需求的测试方法（RBT）**，并注明"后续课时展开"。

提问：如果你要给"缺陷"下一个定义，你会从哪几个角度入手？

预设回答：

- 回答"程序运行不对就是缺陷"——点评：这是现象层，方向没错但太窄，忽略了文档缺陷、需求缺陷、性能缺陷这些"不报错但错"的情况。

- 回答"和需求不一致就是缺陷"——点评：抓住了核心（对照标准），值得表扬；追问一句"那需求本身写错了算不算缺陷"，引出"缺陷的对象不只是代码"。

- 回答"用户不满意就是缺陷"——点评：从用户视角看是对的，但太主观，测试要能落到可判定的依据上，所以要结合质量模型和需求。

- 点评收口：这三种回答恰好对应我们接下来要讲的三层——现象、依据、期望。

> **易错点 / 考点：**本页是目录页，考点在于**章节结构的记忆**：2.1 软件缺陷、2.2 软件测试的分类、2.3 静态／动态、2.4 主动／被动、2.5 黑盒／白盒、2.6 测试层次、2.7 工作范畴。常见混淆是"2.4 主动与被动"记成"自动与手动"——注意主动测试与被动测试是另一对概念，手工与自动化是**另一组分类维度**，两者不在一个层次上。记忆口诀：**"缺陷、分类、动静、主被、黑白、层次、范畴"**，十四字背下来。

> **过渡：**地图看完了，我们从第一站出发——2.1 软件缺陷。先看这一节的学习路线怎么走。

【2.4】2.1 软件缺陷（2 分钟）

> 这一页是 2.1 这一节的门牌。接下来这一整节都围绕"缺陷"两个字，但不会只给一个定义就结束。这一节按六个层次推进：先搞清楚**概念**——缺陷与质量是什么关系，工作中什么场景会用到；再做**讲解**，把胜任这件事必须掌握的知识和技术讲细；然后**举例**，用一个案例帮你真正看懂怎么用；接着**分享经验**，我会讲理论和实际有落差的地方；再给出**学习建议**；最后如果有，给大家**实用工具**和模板。展开来就是五个小标题：**2.1.1 软件质量的内涵、2.1.2 软件缺陷的定义、2.1.3 软件缺陷的产生、2.1.4 软件缺陷的构成、2.1.5 修复软件缺陷的代价**。换成五连问就是：质量是什么？缺陷是什么？从哪来？长什么样？修它有多贵？把这五问回答清楚，你就具备了做测试的第一块基本盘。

> **板书：**板书六个关键词竖排——**概念 → 讲解 → 举例 → 经验分享 → 学习建议 → 实用工具**；右侧再竖排五个小标题：2.1.1 质量的内涵／2.1.2 缺陷的定义／2.1.3 缺陷的产生／2.1.4 缺陷的构成／2.1.5 修复代价。

> **提问：**在"缺陷是什么"和"缺陷从哪来"之间，为什么我们要先花力气讲"质量"？

> **预设回答：**

- 回答"因为质量更重要"——点评：方向对但不完整，要说清因果关系：缺陷是**相对质量**而存在的，没有质量标准，就没有"不合格"这个判断。

- 回答"因为考试要考"——点评：坦诚，但请把"考"换成"用"：没有质量标准的团队，缺陷单会被开发一句"这不算 bug"打回来。

- 回答"因为要先有需求"——点评：很接近了，需求其实就是质量标准的一种表达形式，值得表扬。

- 追问：那么"用户说不好用"算不算缺陷？——引出下一页要讲的核心命题。

易错点 / 考点：本页是分隔页，考试不直接考，但要注意**顺序不能颠倒**：先质量、后缺陷。答题时如果先写缺陷定义再补质量，容易被判逻辑不清。判断三步法记住：**先立标准（质量）→ 再对标准找差异（缺陷）→ 最后评估严重度与代价**。

过渡：好，我们现在就走进 2.1 的第一句话——缺陷是质量的对立面。

【2.5】缺陷是质量的对立面（6 分钟）

这一页只有一句话，但这句话是整节 2.1 的钥匙：**缺陷是质量的对立面**。我们先看课件原文怎么说的：要了解什么是缺陷（defect），就必须先清楚"质量"（Quality）这个概念，因为缺陷是相对质量而存在的；违背了质量、违背了客户的意愿、不能满足客户的要求，就会引起缺陷或者产生缺陷。大家看屏幕上这张关系图，我按箭头念一遍：最左边是**用户**，用户有**要求／期望**；要求和期望汇聚起来，就形成了**质量**的标准；质量标准和现实产品之间是**矛盾／对立**的关系；在矛盾里**发现**了问题，于是出现了我们天天挂在嘴边的那个词——**Bug?** 注意这个问号：发现现象不等于判定缺陷。接着往下走：**软件测试**去**发现**，再通过**评估**去对照**质量模型**，最后才能下结论——这到底算不算缺陷、严重到什么程度。

为什么非得从质量讲起？因为**缺陷不是绝对的，它是相对于质量要求而存在的**。同一段代码，放在只给三个内部员工用的小工具里可能无所谓，放在银行转账系统里就是重大事故。抛开质量谈缺陷，就像抛开尺子谈长短。

生活里的类比最直白：同一件衬衫，地摊标准下"能穿"就算合格，出口欧标下纽扣拉力不够就是不合格。不是衣服变了，是**标准变了**。

软件行业的例子更清楚：一个页面在 4G 下 1.2 秒打开、在弱网下 8 秒才打开，如果需求写明"弱网环境首屏不超过 3 秒"（据课件数据口径，具体指标以你们项目的需求文档为准），8 秒就是缺陷；如果需求里根本没写性能指标，那它现在只能算一条"待确认项"，还不能直接判缺陷。所以我们做测试，第一步永远不是点按钮，而是拿到那把尺子——需求与质量模型。

板书：中间画箭头链：**用户 → 要求/期望 → 质量 ⊥（矛盾/对立）→ 发现 → Bug? → 软件测试 → 评估 → 质量模型**。右上角写一句结论："**没有质量标准，就没有缺陷判定**"。左下角写两个例子关键词：**衬衫纽扣拉力（生活）／弱网首屏 3 秒（软件）**。

提问：如果一个用户抱怨"这个功能很难用"，但需求文档里完全没提易用性指标，这条抱怨能不能直接登记

成缺陷?

预设回答：

- 回答"能，用户说不满意就是缺陷"——点评：用户视角值得尊重，但测试记录要可判定；更规范的做法是先登记为"待确认问题"，再回到需求和隐含期望去确认，确认后升级为缺陷。这里正好对应课件那句"违背了客户的意愿"。

- 回答"不能，需求没写就不算"——点评：太绝对，会漏掉**隐含需求**。教材口径是"规定的或隐含的要求"，用户对易用性的常识性期望属于隐含要求。

- 回答"看情况/看严重程度"——点评：有判断意识，但顺序反了，严重程度是判定**之后**才评估的。

- 追问：那"隐含需求"怎么变成可判定的标准？——引出下一页要讲的"软件质量的内涵"，以及后面章节的 ISO 质量模型。

【思政落点 1】讲一个真实的工作场景：某团队上线后收到大量用户投诉，工程师在周报里写"功能符合需求文档，判定通过"。知识点联系——课件上那个"Bug？"的问号，本质上就是**科学精神**的落点：发现异常不等于下结论，结论必须由证据、可复现步骤和明确标准支撑；同时它也是**诚信**的落点。价值观——测试人员的职责不是"证明自己没漏"，而是如实记录、如实上报，哪怕这个结果让团队难堪。写下"不通过"三个字需要勇气，而这份勇气就是这一行的职业底线。

易错点 / 考点：本页高频考点有两个。第一，**缺陷的定义必须带上"相对于质量"这个前提**，只答"程序错误"是不完整的，标准答案要写"缺陷是相对质量而存在的，违背了质量、违背客户意愿、不能满足客户要求"。第二，**判定缺陷的四个来源**：需求（含隐含需求）、设计规格、用户期望、质量模型。判断三步法：**① 找到标准 → ② 对比实际 → ③ 判定是否违背**。案例题常见问法："发现某现象，能否判为缺陷？请说明依据"——拿分点是写出"依据是需求/质量模型"，而不是写"我觉得是"。

过渡：既然判定缺陷要有一把尺子，那这把尺子——"软件质量"——本身是怎么定义的？我们把 2.1 的内容再展开一层，看下一页的五个小标题。

【2.6】2.1.1 软件质量的内涵（3 分钟）

把 2.1 再往里拆一层，这五个小标题就是接下来的行军路线。第一站 **2.1.1 软件质量的内涵**：不满足于"质量就是好"，要拿到能写进报告的定义。第二站 **2.1.2 软件缺陷的定义**：把缺陷从"感觉不对"变成有明确构成要

素的判断对象。第三站 **2.1.3 软件缺陷的产生**：缺陷来自需求、设计、编码、测试、配置和文档，我们会分析它产生的环节。第四站 **2.1.4 软件缺陷的构成**：一份缺陷报告要写哪些字段才合格，这直接影响课程设计打分。第五站 **2.1.5 修复软件缺陷的代价**：同一缺陷在需求阶段发现和在运维阶段发现，代价差几个数量级，这就是测试要"左移"的经济学依据。这五节的关系打个比方：**质量是尺子，缺陷是刻度上的偏差，产生是偏差的来源，构成是记录偏差的格式，代价是偏差的价格**。

> **板书：**横排五个方框：**质量的内涵（尺子）→ 缺陷的定义（偏差）→ 缺陷的产生（来源）→ 缺陷的构成（记录）→ 修复的代价（价格）**；下方写一行提示：考试重点在 2.1.2 与 2.1.5。

> **提问：**这五节里，你认为哪一节最直接影响你的课程设计得分？为什么？

> **预设回答：**

- 回答"2.1.4 缺陷的构成"——点评：非常实际，缺陷报告字段不全，开发和老师都无法复现，直接扣分；表扬其工程意识。

- 回答"2.1.5 修复代价"——点评：有战略眼光，因为它决定了测试什么时候介入、投入多少；但代价这一节更常出现在论述题里。

- 回答"2.1.3 产生的环节"——点评：也对，懂缺陷来源才能设计出有针对性的用例，属于"预防优于检测"的思路。

- 追问：那如果只能选一节背下来呢？——给建议：定义与代价两节必须能默写，其余理解即可。

> **易错点 / 考点：**本页是目录页。要注意区分 2.1.1 是"**软件**质量"的内涵，而不是泛泛的"质量"；很多同学在简答题里把两者混写，会丢掉"隐含要求""使用要求"这些关键词。判断三步法：**看到"内涵"想定义与出处（IEEE／ISO）→ 看到"产生"想环节（需求到维护）→ 看到"代价"想阶段（越晚越贵）**。

> **过渡：**我们从最根上的那一问开始——什么是质量？看下一页。

【2.7】2.1.1 软件质量的内涵（2 分钟）

这一页又出现"2.1.1 软件质量的内涵"，它不是重复，而是**转场**：前面是路线图，现在真正进入这一小节的正文。接下来连续几页，我们都在回答同一个问题——什么是质量？你会看到三种讲法：第一种是**图片和直觉**，先对着画面自己感受；第二种是**引文和观点**，用一段很有味道的话告诉你质量不是什么、又是什么；第三种才是**定义和标准**，IEEE 怎么说、ISO 怎么说。我特意把顺序安排成"感受—观点—定义"，是有原因的：质量

这个概念，一上来就背定义，你会觉得它是一句空话；先有画面感，定义才会落地。所以下几页请不要急着抄定义，先跟着看图、跟着想，最后我们一起把定义抄下来。这一小节是本节的**概念地基**，后面的缺陷定义、质量模型、质量标准体系全都建立在它之上。

板书：板书居中写大标题"**2.1.1 软件质量的内涵**"；下方画三格：**感受（图）→观点（引文）→定义（IEEE／ISO）**，并在"定义"格下标注"本页为概念地基"。

提问：不看课件，请用你自己的一句话给"质量"下个定义，不许说"质量就是好"。

预设回答：

- 回答"质量就是满足用户要求"——点评：抓住了"要求"，已经接近教材口径，值得表扬；再追问"用户没说的要求算不算"，引出"隐含要求"。

- 回答"质量就是不出问题、没有 bug"——点评：这是从测试视角的朴素理解，但"没有 bug"是结果不是定义，而且统计上不可能达到。

- 回答"质量就是好用、耐用"——点评：这是生活语感，属于体验层描述；提醒学生：能写进报告的表述必须可衡量，比如"平均无故障时间""缺陷密度"。

- 追问：那"贵"是不是质量的一部分？——先不回答，留到下一页那段引文里找答案。

易错点 / 考点：本页是转场页。易错点是**把"质量"与"等级""档次"混为一谈**——高档不等于高质量，高质量指的是"满足要求的程度"，而不是"用料多贵、界面多华丽"。这一点在第 2.9 页的引文里会被正面强调，考试常出判断题：**"价格高的软件质量一定更高"——错**。

过渡：那么，质量到底长什么样？我们先不看定义，看一张图。

【2.8】什么是"质量"？（4 分钟）

这一页只有一幅图和四个大字——什么是"质量"？这是纯图形页，课件故意不给文字，就是要我们**先看、先想、先说**。给大家三十秒，看三处。**一看整体感**：这幅图有没有"被认真对待过"？质量往往不是某个零件亮眼，而是整体上没一处将就。**二看细节**：边缘、接缝、比例、留白——体现质量高低的，常常是这些说不清但看得出的小地方。**三看一致性**：各部分像不像出自同一人之手？在软件里，这就是"风格统一、交互一致"。

为什么要看图？因为**质量首先是能被感知的**。用户评判软件，往往不是靠读需求文档，而是靠直觉：按钮位

置一会儿左一会儿右、加载图标时有时无，用户说不清哪里不对，但会说"这个软件很糙"；反之，间距、颜色、返回逻辑都一致，他只会说"挺好用"。

看软件实例：同一功能在三个页面都有入口，按钮文案却写成"提交""确定""保存"。功能全对、测试全过，用户就是觉得别扭——这不是功能缺陷，而是一致性体验缺失。看图训练的正是测试人员最稀缺的能力：**把"感觉不对"变成可描述、可复现、可判定**。

板书：中间写四个大字"**什么是质量？**"；下方画三个观察维度：**整体感 ／ 细节 ／ 一致性**；右下角写一行待答问题："这幅图的'认真对待'体现在哪里？"——留白，等学生回答后补写。

提问：这幅图里，哪个细节让你觉得"做的人是用心的"？请指出来，并说说你为什么这么判断。

预设回答：

- 回答"整体看起来很舒服，但我说不出哪里好"——点评：非常真实，这正是质量的特征——**说不出好，但缺了就立刻发现**；引导他把"舒服"拆成比例、间距、配色三件事。

- 回答"某个局部特别精致"——点评：观察力好，但提醒他反向思考：如果只有一个局部精致、其余潦草，整体质量是升还是降？答案是降，因为**不一致本身就是缺陷**。

- 回答"我看不出来"——点评：不要慌，看图和找缺陷一样需要训练；给方法：把图分成九宫格，逐格问"这里有没有一处显得随意"。

- 追问：如果把这幅图的某个元素换掉，你多久能发现？——引出"显著性"与"严重程度不是一回事"这个后续考点。

易错点 / 考点：本页是纯图形页，考试不会直接考。但要记住两个口径：① **质量可以被感知，但不能只靠感觉判定**，判定仍要回到要求与标准；② **一致性是质量的组成部分**，界面风格、术语、交互逻辑不统一，可以登记为缺陷（通常是易用性或一致性类缺陷）。常见的错误认知是"界面问题不算缺陷"——在真实项目和课程设计评审里，这类问题一样会被记录和扣分。

过渡：刚才我们用眼睛感受了质量，接下来听听前辈怎么用文字描述它——下一页有一段很精彩的引文。

【2.9】什么是"质量"？（6 分钟）

屏幕上这段引文，我慢慢念一遍，请记关键词："质量的概念很难定义，因为它不仅是可见的，而且在体现它

> 的作品中以某种方式被**直觉地**呈现出来。质量与大众的审美、品味或风格无关；与地位、体面或奢华无关。相反，它是在一种**接受、得体和克制**的气氛中显露出来的。"
>
> 这段话有三层意思。**第一层，"难以定义"但又"可见"**。课件专门注了英文词 intuitive，直觉。什么意思？就是刚才看图时那种感觉——你说不出标准，但一眼就能看出一个东西是被认真做过还是糊弄的。软件也一样：同一需求，两个团队都能实现功能，用户上手三分钟就能觉出差别。这种"说不出但看得见"的特性，正是质量最难教的地方。
>
> **第二层，"与审美、品味、风格无关，与地位、体面、奢华无关"**。这一句是重点，也最容易考。它明确告诉我们：**质量不是"好看""高级""贵"**。界面朴素但贴合使用习惯的软件，质量可能高于动画炫酷却操作反人类的软件；功能简单的小工具，把该做的事做到位，就是高质量的。反过来，堆料、堆特效、堆价格都不是质量的证明。对测试人员尤其重要——**判定不能被"看起来高端"带跑**。
>
> **第三层，"在一种接受、得体和克制的气氛中显露出来"**。这三个词很讲究：**接受**，用户愿意接受它，用起来没有抵触；**得体**，分寸上不多不少，比如提示语不吓人、报错不甩锅、弹窗不打扰；**克制**，不炫技、不过度设计，把注意力留给用户真正要做的事。
>
> 举个大家都有的体验：点删除时，是轻轻问一句"确定删除吗"，还是弹一堆术语让你自己猜？前者得体，后者不合格。**质量的最高境界是让人感觉不到它的存在**——用得顺，就是质量好。

> **板书：**板书三段式。上："**质量难定义，但可直觉感知（intuitive）**"；中："**≠审美 ≠品味 ≠风格 ≠地位 ≠体面 ≠奢华**"（"≠"要写得醒目）；下："**接受 · 得体 · 克制**"；右侧写一行提炼："**用得顺，就是质量好**"。

> **提问：**这段话明确说"质量与地位、体面或奢华无关"。那么一个界面炫酷、动画华丽、价格昂贵的软件，能不能因此就判它质量高？

> **预设回答：**

- 回答"不能，还要看它是否满足要求"——点评：完全正确，请他把这句完整说出来，作为全班结论；这才是教材口径。

- 回答"能，贵有贵的道理"——点评：这是典型的直觉误判，正好用课件这句话反驳：价格和奢华属于"体面"，而质量讲的是"满足要求的程度"；贵可能说明成本高，不能说明质量高。

- 回答"看用户喜不喜欢"——点评：方向对，但要补一句"喜欢"要落到可用性与需求满足度上，例如操作是否顺手、报错是否可理解，否则就成了纯主观。

- 追问：那"克制"和"功能少"是一回事吗？——不是。克制是不做无用打扰，功能少是能力不足，别把"偷懒"说成"极简"。

> **易错点 / 考点：**本页是**高频概念考点**。必背三个词：**接受、得体、克制**；必背三个"无关"：**审美、品味、风格**（以及地位、体面、奢华）。常见考法：① 判断题——"界面华丽、价格昂贵的软件质量更高"（错）；② 简答题——"如何理解'质量是在接受、得体和克制的气氛中显露出来的'"。判断三步法：**看它是否满足要求 → 看它是否打扰用户 → 看它是否炫技过度**。记忆口诀：**"三无三有"**——无关审美奢华，有关接受得体克制。

> **过渡：**感受和观点都有了，但写在测试报告里的东西不能靠直觉——下一页，我们把质量变成一句能引用的定义，而且要从客户和品牌的角度再补一刀。

【2.10】质量 =品牌 =客户满意度（5 分钟）

屏幕上这一页给出了一个等号链：**质量 = 品牌 = 客户满意度**。两个等号，我们一个一个说。为什么说**质量等于客户满意度**？因为质量最终是由用户来兑现的。你内部的代码再优雅、需求文档再规范，如果用户用着别扭、出错、不敢用，那在他心里这个软件就是"差"。我们刚才讲过，质量是"满足要求或期望的程度"，而"期望"住在用户脑子里，不在你的代码里。所以测试人员在写用例的时候，除了对着需求文档，还要反复问自己一句：**用户真正会怎么用它？**

为什么又说**质量等于品牌**？因为品牌是什么？品牌就是用户对一家公司、一个产品的**长期信任的总和**。这份信任从哪来？从每一次使用的体验里一点点攒起来。质量稳定，用户下次还敢用；质量出一次大事，十年口碑可能一夜归零。你们可以回想自己的经历：某个 App 曾经卡死丢过你的数据，之后你是不是很长一段时间不敢再打开它？这就是质量对品牌的杀伤力。

我用生活里的例子把这条链说透。一家小吃店，第一次去味道好、上菜快，你会记住它；第三次去还是好，你会推荐给朋友；第十次去依然稳定，你就成了老顾客——**这就是品牌**。中间只要有一次吃出问题还糊弄你，前面九次都白攒。软件行业更极端：软件是可以瞬间复制给几千万人的商品，**一次严重缺陷的放大倍数，远远超过一家小吃店**。

所以这一页真正想告诉大家的是：**质量不是一个技术指标，它是商业结果**。它同时决定了用户的满意度、企业的品牌和收入。这也是为什么测试岗位不是成本中心，而是品牌的守门人。同学们做课程设计时，如果只把"跑通用例"当目标，那就把这一页的意义丢了。

> **板书：**中间写等式链：**质量 = 品牌 = 客户满意度**；等式下方画一条向下的箭头写"**长期信任的累积（慢）**"，再画一条向上的箭头写"**一次严重缺陷的崩塌（快）**"；左下角写生活例"小吃店"，右下角写行业例"App 丢数据 → 卸载"。

提问：为什么说软件的质量事故比传统行业的质量事故"放大倍数"更大？

预设回答：

- 回答"因为用的人多"——点评：对了一半，还要补上"分发成本几乎为零、版本统一、缺陷瞬间同步给所有用户"，这才是软件独有的放大器。

- 回答"因为软件 bug 多"——点评：方向偏了，缺陷数量不是关键，关键是**影响半径**和**不可逆性**（数据丢了就没了）。

- 回答"因为修起来快，所以也无所谓"——点评：纠正一个常见误解：回归上线快不代表损失小，用户流失和信任损失修不回来。

- 追问：那测试能不能保证品牌不出事？——不能保证，但能把"出事概率"压到可接受水平，这就是风险管理的说法。

> **易错点 / 考点：**本页考点是这条等式的**因果方向**：是**质量决定**满意度与品牌，不是品牌决定质量。常见错误答案："品牌好所以质量好"（本末倒置）。另一个易错点是把"客户满意度"等同于"客户当时不投诉"——不投诉也可能是用户已经放弃、默默卸载了。判断三步法：**谁在用 → 用得好不好 → 会不会再来**。

> **过渡：**感受、观点、商业关系都讲完了，现在到了最关键的一步：把"软件质量"写成一个可以引用的正式定义。看下一页。

【2.11】软件质量 的内涵（6 分钟）

> 这一页进入正题，把"软件质量"变成可引用的定义。课件给了三个口径，逐条读、逐条拆。第一句，**IEEE 的定义**：质量是系统、部件或过程**满足明确需求**，以及**客户或用户需要或期望的程度**。请注意结构——它把质量拆成两边：一边是"明确需求"，写下来的、可验证的要求；另一边是"客户或用户需要或期望"，可能有、但没写下来的东西。课件在这个位置标注了"客户或用户需要或期望的程度不同"，意思是同一产品面对不同用户，期望不同，满足程度也不同，所以质量是个**程度**问题，不是"有或没有"的开关。
>
> 第二句，**ISO 口径的软件质量**：软件产品具有满足**规定的或隐含的要求**能力有关的**特征与特征的总和**。四个

关键词必须抠出来：**规定的**，需求文档、合同、标准里写清楚的；**隐含的**，用户不说但理所当然的期望，比如计算器不能算错数、保存按钮不能点了没反应；**特征**，具体属性，比如响应时间、并发数、易用性；**总和**最关键——**质量是这些特征的合力，不是某一项突出**。一个软件响应快但天天崩，不叫高质量——某一项拖了总分。课件标注的标准编号是 **ISO 8492**——这里要给大家提个醒：**这是课件的一处笔误**。该定义出自 **ISO 8402《质量管理和质量保证——术语》**；若写作 **ISO 9126**，那是指质量模型标准（2.14 页要讲的三层模型）。**课堂与作业统一按 ISO 8402 讲**；遇到课件原文时可以指出这处笔误，这也是一次"以标准原文为准、不盲从讲义"的训练。

第三句最朴素：**软件质量，就是软件产品满足使用要求的程度**。它把前两句收拢到"使用"上——**标准再多，最后都要落到能不能用、好不好用、敢不敢用**。

合起来就是可以写进课程设计报告的定义：**软件质量，是软件产品满足规定的和隐含的要求（也就是用户需要与期望）的程度，它是相关特征与特征的总和，最终体现在使用要求的满足程度上**。这句话背下来，2.1.1 就拿下了。

板书：板书三行定义：① **IEEE**：质量＝满足**明确需求** ＋ 客户/用户**需要或期望的程度**；② **软件质量（课件标注 ISO 8492，实为 ISO 8402）**：满足**规定的或隐含的**要求能力有关的**特征与特征的总和**；③ **软件质量＝满足使用要求的程度**。右侧抠四个关键词：**规定 · 隐含 · 特征 · 总和**；底下一行结论："质量是**程度**问题，不是有/无开关"。

提问：一个软件功能全对、响应很快，但每周要崩溃一次。按课件的定义，它的质量算高还是低？

预设回答：

- 回答"不高，因为可靠性差，特征总和拉低了"——点评：完全正确，请他把"特征与特征的总和"这七个字说出来，这是拿分关键词。

- 回答"高，因为大部分时候能用"——点评：这是典型的"用平均值掩盖风险"的思路；提醒他质量的"总和"不是算术平均，关键特征的缺失会直接否决整体评价。

- 回答"看用户能不能忍"——点评：有商业现实感，但作为测试结论太主观；应当写成"可靠性特征不达标，判定为高风险缺陷"。

- 追问：那"隐含要求"能不能举一个你身边的例子？——引导学生举"保存后不丢数据""点了要有反应""中文不乱码"这类例子。

易错点 / 考点：本页是**定义必考页**。三个高频失分点：① 漏掉"**隐含要求**"——只写"满足需求"，丢掉一半分数；

② 把质量说成"**没有缺陷**"——定义里从来没有这句话；③ 把"特征总和"理解成"各项加权平均"，忽略了关键特征的一票否决效应。考试形式多为名词解释、简答或案例分析；记忆口诀：**"规定＋隐含，特征成总和，落回使用中"**。判断三步法：**有没有写下来（规定）→ 用户是不是理应能得到（隐含）→ 各项特征合起来够不够（总和）**。

> **过渡：**定义有了，可"特征"到底有哪些？这就是下一页要给的——高质量软件的标准体系，我们从产品、流程、商业环境三个面去看。

【2.12】高质量软件标准体系（6 分钟）

这一页最有体系感：谈质量不能只盯着代码，要分三层看。课件把高质量软件的标准体系画成三个维度——**产品、流程、商业环境**，逐个展开。

先看**产品质量**。课件说得很明确：产品质量是人们实践产物的属性和行为，**可以认识、可以科学地描述**，并能通过方法和人类活动**改进**。这话在打气——质量不是玄学，可测量、可改进。那怎么描述？靠**质量模型**。课件列了三个经典模型：**McCall 模型、Boehm 模型、ISO 9126 模型**。它们的共同思路是把"质量"拆成若干可度量的特性，比如功能性、可靠性、易用性。写课程设计报告时，如果能在"测试范围"一节引用一个质量模型，说明测了哪几个特性、为什么选，报告立刻上档次。

再看**过程质量**。核心是：**产品的质量，是过程质量的产物**；过程不靠谱，产品靠运气。课件列了三样：**CMM 软件能力成熟度模型**，评估组织的开发过程成熟到什么等级；**ISO 9000** 国际标准过程模型，通用质量管理体系标准；以及 **SPICE**，软件过程改进和能力决断，用于评估和改进软件过程能力。共同点是**把"人治"变成"制度化"**——不指望高手救场，而是让流程稳定地产出合格产品。

最后看**商业环境**。课件列举了商业过程中的质量内容：**培训、成品制作、宣传、发布日期、客户、风险、成本、业务**等。这一层最容易被技术同学忽略：一个技术上没问题的产品，如果发布日期被强行提前导致测试被压缩，或者宣传夸大到产品根本做不到，质量照样出问题。所以"高质量"是**产品、流程、商业环境三者的合围**，缺一个角都不成立——记住：**测产品、看流程、认环境**。

> **板书：**板书三列框架——**产品（质量模型：McCall／Boehm／ISO 9126）｜流程（CMM／ISO 9000／SPICE）｜商业环境（培训・成品制作・宣传・发布日期・客户・风险・成本・业务）**；产品列下方补一句："**可认识、可描述、可改进**"；流程列下方补一句："**产品的质量是过程质量的产物**"。

> **提问：**既然产品质量可以通过测试来保证，为什么还要关心"过程质量"？

> **预设回答：**

- 回答"因为测试只能发现缺陷，不能预防缺陷"——点评：非常到位，这正是过程质量存在的理由；请他把"测试是检测、过程是预防"这一对比写在笔记上。

- 回答"因为要过 CMM 认证、要拿单"——点评：这是商业动因，确实存在，但只是手段；根本目的仍是让质量稳定可复制，而不是靠个别高手。

- 回答"过程质量是管理层的事，跟我们测试没关系"——点评：明确纠正。测试用例评审、缺陷跟踪规范、回归策略，本身就是过程质量的一部分，测试人员恰恰是过程质量的重要执行者。

- 追问：SPICE 和 CMM 有什么区别？——提示二者都面向过程能力评估与改进，记忆时抓住"能力成熟度"与"过程改进和能力决断"这两个名称即可。

> **易错点 / 考点：**本页考点集中在**名称与归属的对应**，非常容易张冠李戴。请记牢：**质量模型**是 McCall、Boehm、ISO 9126（面向**产品**）；**过程质量**是 CMM、ISO 9000、SPICE（面向**流程**）；商业环境的内容是培训、成品制作、宣传、发布日期、客户、风险、成本、业务（面向**环境**）。常考题型是连线题或选择题，比如"下列属于过程质量范畴的是"。记忆口诀：**"产品三模型，流程三体系，环境八要素"**（八要素按课件列举计数，据课件数据）。另一个易错点是把 CMM 说成"软件测试模型"——它评的是**组织开发过程的能力成熟度**，不是测试技术。

> **过渡：**体系讲完了，但测试人员最常打交道的还是产品那一列。下一页我们把"产品质量的标准"具体摊开，看看有哪几项是必须盯住的。

【2.13】产品质量的标准（6 分钟）

这一页把"产品质量"落到条目上，课件列了九项，我念一遍：**功能性 Functionality**、**可用性 Usability**、**可靠性 Reliability**、**性能 Performance**、**容量 Capacity**、**可伸缩性 Scalability**、**可维护性 Service manageability**、**兼容性 Compatibility**、**可扩展性 Extensibility**。课件把这九项列在"**非功能特性**"栏目下，须提醒：按通行 ISO 口径，**功能性属于功能特性**，其余八项才是非功能特性；答题按教材口径写。

再挑四组最容易混的讲清。**可用性和功能性**：功能性回答"有没有、对不对"，可用性回答"能不能顺利用起来"——能算对却要绕八步，功能性没问题，可用性很差。**可靠性和性能**：性能看"快不快"，可靠性看"稳不稳、

出错后能否恢复"；快而不稳，比慢而稳更可怕。**容量和可伸缩性**：容量是"现在能装多少"，比如并发数、数据量；可伸缩性是"需要时能不能扩上去"，加机器后能力能不能跟着涨。**可维护性和可扩展性**：前者偏"好不好查、好不好修"，后者偏"好不好加新功能"。

为什么九项要一起看？它们常常**互相拉扯**：加缓存性能上去了，可能一致性出问题；校验做严，可能可用性下降；为可扩展把结构做复杂，可能可维护性变差。测试人员的角色是**在特征之间帮团队做取舍的裁判**——说清"为保住 A 牺牲了多少 B"。

做课程设计时，别九项平均用力，而是**选三到四项最相关的特性**，写明理由，再设计对应验证方法——这比"我全都测了"更有说服力。

板书：板书九项，横排三行三列：**功能性／可用性／可靠性｜性能／容量／可伸缩性｜可维护性／兼容性／可扩展性**；右侧写四组对比：**功能 vs 可用（有没有 ↔ 顺不顺）· 性能 vs 可靠（快不快 ↔ 稳不稳）· 容量 vs 伸缩（装多少 ↔ 能不能扩）· 可维护 vs 可扩展（好不好修 ↔ 好不好加）**；角落标注"课件栏目：非功能特性（功能性按 ISO 口径属功能特性）"。

提问：一个系统"响应很快、功能很全"，但每隔几天就需要重启一次。请指出它在九项标准中哪一项最不达标，并说明可能连带影响哪些项。

预设回答：

- 回答"可靠性不达标"——点评：正确，请补充理由："稳定运行能力不足"，并追问他会不会连带影响可用性；引导他答出"频繁重启会直接伤害可用性与用户体验"。

- 回答"性能不达标"——点评：这是最常见的误判，因为"响应快"给了错觉；提醒他性能看的是速度与吞吐，重启看的是稳定性，别把两者混为一谈。

- 回答"可维护性不达标"——点评：有洞察力，频繁重启往往意味着缺陷定位困难；但这是**连带影响**，主因仍是可靠性。

- 追问：如果这个系统是银行核心系统，你会在测试报告里怎么定级？——引导学生用"影响范围＋严重程度＋发生频率"三要素定级。

【思政落点 2】讲一个课堂可讨论的情境：同一次测试里，你发现了三个缺陷，其中两个是界面小瑕疵，一个是概率极低但后果严重的隐患——比如涉及数据准确性。知识点联系——这一页九项标准里，可靠性、容量这类特性往往"平时看不出来，出事就是大事"，测试人员要有**科学的定级判断**，不能只挑好复现、好交差的缺陷报。价值观落点：**如实记录、按标准定级、不为好看的数字隐去不利结果**，这就是第 2 次课思政设计里

最看重的诚信与质量责任；一个团队的质量文化，就体现在敢不敢把那个"概率低但后果重"的缺陷摆在桌面上。

易错点 / 考点：本页是**高频考点页**。必背九项中英文对照（功能性、可用性、可靠性、性能、容量、可伸缩性、可维护性、兼容性、可扩展性），以及四组对比（功能 vs 可用、性能 vs 可靠、容量 vs 伸缩、可维护 vs 可扩展）。常见错误：① 把"性能"当"可靠性"，把"容量"当"可伸缩性"；② 把"功能性"归入非功能特性（注意课件页面本身把它列在"非功能特性"栏目下，答题按教材口径区分）；③ 认为"功能全就等于质量高"。考试形式多为选择题、判断题或案例分析题；记忆口诀：**"功能可用可靠，性能容量伸缩，维护兼容扩展"**——九个词，三三成句。判断三步法：**先问是功能还是非功能 → 再问是"现在的表现"还是"未来的能力" → 最后问它和哪一项容易被混淆**。

过渡：质量的内涵与标准我们讲完了，下一片开始，我们将进入 2.1.2——软件缺陷的定义，把"缺陷"这个东西从概念变成一份可以填写、可以评审、可以追责的记录。这是你们课程设计里马上要用到的本事。

【2.14】软件质量特征（ISO 9126）（3 分钟）

前面我们讨论的是"质量是什么"，这一页把它落到国际标准上：软件的质量到底由哪几个可检查的侧面组成。请大家记住这个标准号——ISO 9126，它是影响最大、被引用最多的软件质量模型之一，后面很多标准都是从它发展出来的。它把软件质量归纳成六大特征，我逐个讲。

第一，功能性。课件上的定义是"与一组功能及其指定性质有关的一组属性"，功能就是满足明确或隐含需求的能力。说人话：该做的事，做没做到、做得对不对。第二，可靠性，"在规定的一段时间和条件下，软件维持其性能水平的能力"——请注意"一段时间"和"规定条件"这两个限定，程序跑一次不崩不叫可靠，长期稳定运行才算。

第三，易用性，"规定或潜在的用户为使用软件所需付出的努力和所作的评价"。它有两半：一半是客观的努力，学多久、点几下；一半是主观的评价，用起来舒不舒服。第四，效率，"在规定条件下软件的性能水平与所使用资源量之间关系"，说白了就是同样的活儿，干得快不快、吃得少不少。

第五，可维护性，"与进行指定的修改所需的努力有关的属性"，改一个需求要动多少代码，这就是可维护性。第六，可移植性，"软件从一个环境转移到另一个环境的能力"，换操作系统、换数据库还能不能跑起来。

最后一句课件上的话很重要：每一个质量特征都分别与若干子特征相对应。也就是说，这六项还太粗，不能直接拿来测量，必须继续往下拆。拆下去长什么样？就是下一页那张三层模型。

> 这里我再提醒一句：这六大特征不是并列的口号，而是有优先级的。面对不同产品，团队必须根据用户场景给它们排序，否则所谓"质量"就成了没有重点的一团。

> **板书：**写"ISO 9126 六大质量特征"，一边讲一边竖着写：**功能（做没做到）→ 可靠（稳不稳）→ 易用（好不好用）→ 效率（快不快·省不省）→ 可维护（好不好改）→ 可移植（好不好搬）**；下面补一句："**每一个特征往下还有若干子特征**"。

> **提问：**如果让你给"学生选课系统"评质量，这六个特征里你会先盯住哪两个？为什么是这两个？

> **预设回答：**

- 答"可靠性和效率"：✅ 很到位。点评并追问："选课系统为什么 reliability 重要？"引导到业务本质——**抢课高峰期系统崩了，学生和教务全部受影响，这是关键业务时间窗（如开学第一周）最不能出的问题**，所以可靠性优先级最高；紧接着就是效率，因为并发量决定选课能不能在几分钟内结束。

- 答"易用性，因为界面要好看"：⚠ 只说到了外观。点评：易用性在标准里指的是"用户为使用软件所需付出的努力和评价"，**重点是任务完成成本（几步能选上课、会不会选错）**，不只是好看；界面美观只是其中一小部分。

- 答"六个都很重要，全部同等对待"：❌ 典型但错误的答案。点评：资源永远有限，**质量特征必须有优先级，依据是产品定位和用户场景**；教务内部系统的可移植性权重就很低，而面向公众的 App 则完全不同。

> **易错点 / 考点：**①中英文对应必须记牢——功能性 functionality、可靠性 reliability、易用性 usability、效率 efficiency、可维护性 maintainability、可移植性 portability；②最常见的混淆是把"效率"和"性能"划等号，标准里效率＝性能水平与资源消耗的关系，**是"性价比"不是单纯的快**；③把"易用性"等同于"界面美观"；④考点形式：给一段场景描述让你判断属于六大特征中的哪一个（选择题），或者让你补全"特性—子特性"的对应关系。**判断三步法**：先问"这个属性管什么"→ 再对照六大特征的定语（功能实现/稳定运行/用户努力/资源效率/修改成本/环境迁移）→ 最后排除"性能就是效率"这个陷阱。

> **过渡：**六大特征还要往下拆，ISO 9126 把它拆成了三层——我们看下一页这张结构图。

【2.15】ISO 9126 软件质量三层模型（2 分钟）

> 这一页是一张图，请大家把目光放到图上，它要你看出的就是一件事：软件质量不是一句话，而是一棵三层树。第一层在最上面，是总目标——软件质量本身；第二层是刚才讲的六大质量特性：功能性、可靠性、易用性、效率、可维护性、可移植性；第三层在每根枝条的末端，是质量子特性，以及再往下的度量指标。

为什么非要分三层？因为"质量好"这句话没法验证，"可靠性好"同样没法验证，必须一路拆到能测量的东西上。比如可靠性往下拆，可以拆出成熟性、容错性、易恢复性；成熟性还可以用平均无故障时间来度量，容错性可以用"注入故障后系统能否继续工作"来度量。再比如易用性往下拆，是易理解性、易学性、易操作性，可以用"新用户完成第一个任务要多久"来度量。这样一拆，质量就从一句形容词变成了可以打分、可以验收的东西。

这张图还有一个用途：它是后面所有质量模型的"母版"。无论是 Boehm 模型，还是 ISO 25000 系列，走的路子都是"总目标—特性—子特性—度量"这条主线，区别只在特性的划法和层数。

再补一句它和测试的直接关系：三层模型不是学术分类，它决定了测试设计的方法——你选定哪一个度量指标，就等于宣布了要设计哪一类测试项。

板书：画三层：**第一层 软件质量（总目标）／第二层 六大特性／第三层 子特性＋度量指标**；旁边写口诀"**一层看目标、二层看特性、三层看度量**"，再举一例：**可靠性 → 容错性 → 故障注入后能否继续服务**。

提问：为什么"软件质量好"这句话在验收会上没人能反驳，也没人能签字？这张图怎么解决这个问题？

预设回答：

- 答"因为太笼统，没有标准"：✅ 抓住要害。点评：所以三层模型的第二层把"好"拆成六个可讨论的维度，第三层再拆成可测量的指标，**质量评价从口水战变成数据对话**。

- 答"那就直接测第三层的指标，不用管前两层"：⚠ 只见树木。点评：度量指标是手段，第二层的特性是目的；**只盯指标会出现"指标好看、产品难用"的情况**，例如启动时间达标但核心任务路径极长。

- 答"这张图是给开发看的，跟测试没关系"：❌ 需要纠正。点评：恰恰相反，**测试用例的设计就是从第三层的度量指标倒推出来的**——你打算怎么度量，就决定了要设计哪些用例。

易错点 / 考点：①三层的顺序不能颠倒（总目标→特性→子特性/度量）；②"特性"与"子特性"的层级容易混，记住**子特性是能落到测量上的那一层**；③考点形式：给出一个子特性让你归类到六大特性之下（如"易恢复性"归可靠性、"易学性"归易用性、"资源特性"归效率）；④记忆口诀：**一句形容词拆不成用例，拆到第三层才叫质量**。

过渡：ISO 9126 的三层结构讲完了，但质量模型不止这一个。在它之前还有一位前辈，提出了软件质量模型史上第一个层次化结构——我们看 Boehm 模型。

【2.16】Boehm 软件质量模型（3 分钟）

这一页是 Boehm 软件质量模型。Boehm 是软件工程领域的大家，他在 20 世纪 70 年代末提出这个模型，是软件质量模型里较早、也很有影响力的一套层次化结构。课件这一页是一张关键词地图，信息密度很大，我带大家按三条线索来读。

第一条线索是"产品操作"，也就是软件在运行时表现出来的质量。上面有正确性、可靠性、效率、完整性、可用性，还有容错性、执行效率、储存效率、存取控制、存取检查这些更细的条目。可以看出，这条线索回答的是"软件跑起来对不对、稳不稳、快不快、数据安不安全"。

第二条线索是"产品修改"，也就是将来要改它的时候，成本高不高。对应的关键词有可维护性、可测试性、灵活性、扩展性、模块性、一般性、简单性、连贯性。请注意 Boehm 模型的一个历史贡献：它把可测试性和可理解性这类"开发友好"的属性也纳入了质量，而不只盯着运行结果。

第三条线索是"产品维护"和迁移，也就是软件换环境、被别的系统接着用的时候行不行。关键词有可移植性、互用性、机器独立性、软件系统独立性、通讯公开性、数据公开性、重复性、阐述性。

另外还有一组偏人的属性：可训练、沟通良好、易操作的、自我操作性、工具——这就是人和软件打交道时的质量。大家抓一个总印象就够了：Boehm 模型第一次把质量从"代码对不对"扩展到了"好不好改、好不好测、好不好搬、好不好学"，这就是它比单纯的正确性检查高明的地方。

还有一个细节值得注意：这份模型里出现了"工具""可训练""沟通良好"这样的条目，说明早在几十年前，软件质量就已经被理解为"技术加人"的合力，而不只是纯粹的技术问题。

> **板书：**黑板上横着画三条线：**产品操作（正确性 · 可靠性 · 效率 · 完整性 · 可用性）／产品修改（可维护性 · 可测试性 · 灵活性 · 扩展性 · 模块性）／产品维护与迁移（可移植性 · 互用性 · 机器独立性 · 软件系统独立性 · 通讯公开性）**；右侧补一列"**人机属性：可训练 · 沟通良好 · 易操作**"。

> **提问：**Boehm 模型把"可测试性"写成一项质量属性。请问：一个软件"可测试性差"是什么意思？它会给项目带来什么实际后果？

> **预设回答：**

- 答"就是很难测"：✅ 方向对，但要说透。点评：可测试性差的典型表现是——**模块之间耦合死、没有接口、没人能造出输入去观察输出**；后果是测试成本飙升、缺陷漏到生产环境，你在课程设计里也会遇到：写得越死，用例越难设计。

- 答"可测试性是测试人员的事，跟开发质量无关"：❌常见误解。点评：可测试性恰恰是**设计质量的体现**，它由架构和接口设计决定，测试人员只能被动承受。所以业界有句话：**测不动的东西，往往就是设计得不好的东西**。

- 答"把它当作一项功能需求写进规格书就行"：⚠想法不错但落不了地。点评：可测试性不是一条功能，而是一组设计要求（接口可注入、状态可观察、依赖可替换），需要落实在**设计评审的检查项**里。

> **易错点 / 考点：**①别把 Boehm 模型和 ISO 9126 记混——**Boehm 是层次化质量模型的早期代表，ISO 9126 是国际标准的六特性结构**；② Boehm 模型的特色在于同时覆盖"运行表现"和"修改/迁移成本"，并纳入可测试性、可理解性；③考点形式：判断"可测试性属于哪一类质量属性"（产品修改类）、或比较两个模型的异同（简答题）；④记忆口诀：**跑得好（操作）、改得动（修改）、搬得走（维护迁移）、学得会（人机）**。

> **过渡：**Boehm 模型是"前辈"，ISO 9126 是"经典"，那么今天企业真正在用的是哪一套标准？我们看最新的一代——ISO 25000 系列。

【2.17】最新质量标准：ISO 25000 系列（3 分钟）

> 这一页叫"最新质量标准：ISO 25000 系列"，它的正式名字是 ISO/IEC 25000《软件产品质量要求和评定》，英文缩写 SQuaRE，全称 Software product Quality Requirements and Evaluation。请大家记住这几个字母，它是现在国际上评价软件质量的主干标准。
>
> 为什么要出这个新标准？因为老的 ISO 9126 只管"质量模型"，老的标准 ISO 14598 只管"质量评价"，两套标准各说各话；SQuaRE 把它们整合成一条链，从"定要求"一直走到"做评价"。SQuaRE 分成几个分部：2500n 是质量管理，2501n 是质量模型，2502n 是质量测量，2503n 是质量要求，2504n 是质量评价，后面还有 25050 到 25099 的扩展部分。大家不用背全编号，但要抓住它的逻辑顺序：**先定模型（测什么维度）→ 再定测量（怎么量）→ 再定要求（量到什么程度算合格）→ 最后做评价（按证据下结论）**。
>
> 在我们国家，这套标准被等同采用为 GB/T 25000 系列，下一页要讲的产品质量模型，对应的就是 GB/T 25000.10-2016。课件这一页给出的链接是 ISO 官方的在线浏览平台，感兴趣的同学可以课后去查标准原文，考试不会考那么细，但将来做测试方案时，引用标准号是很加分的。
>
> 为什么加分？因为笼统地说"我们按照行业标准做测试"，对方没法追问；而说出"我们的质量要求和评价依据 GB/T 25000 系列"，对方马上可以追问到具体分部，这就是专业与业余的差别。

一句话总结这一页：质量模型从"一家之言"变成了"国际通用语言"，好处是——当你说"这个系统可靠性不达标"时，所有人对"可靠性"的理解是一致的。

板书：写 **ISO/IEC 25000 SQuaRE**，下面竖排：**2500n 质量管理 → 2501n 质量模型 → 2502n 质量测量 → 2503n 质量要求 → 2504n 质量评价**；右侧写对应关系："**国内等同采用：GB/T 25000 系列（产品质量模型＝GB/T 25000.10-2016）**"；再写一句逻辑链："**定模型 → 定测量 → 定要求 → 做评价**"。

提问：如果换作你给一个课程设计项目写测试方案，为什么"先引用一个质量标准"比"直接开始设计用例"更稳妥？

预设回答：

- 答"因为有依据，别人不好反驳"：✅ 说到了要点。点评：标准提供的是**共同语义和可追溯的评价口径**——你说"性能不达标"，标准会追问"你测的是时间特性还是资源特性、在什么条件下、阈值多少"。

- 答"引用标准就是走形式，最后还是要自己测"：⚠ 一半对。点评：标准不替你测，但它替你**定义验收的语言**；没有这层语言，需求方说"要流畅"，开发说"够快了"，测试夹在中间无法裁决。

- 答"越新的标准越好，老标准可以完全丢掉"：❌ 需要纠正。点评：**ISO 9126 的六特性思维今天仍然在用**，SQuaRE 是在它基础上升级整合；学习路径应该是"先懂 9126 的骨架，再看 25000 的整合"，而不是简单地新替旧。

易错点 / 考点：①标准号与名称对应：**ISO/IEC 25000＝SQuaRE＝软件产品质量要求和评定**；**GB/T 25000.10-2016 是我国等同采用的产品质量模型**；②别把 25000 与 9126 的关系说反——**9126 是前身，25000 是整合与升级**；③考点形式：给出标准编号让你说分部职责（如 2501n 管模型、2504n 管评价），或判断"ISO 25000 系列只包含质量模型"（错，它是一条从要求到评价的完整链）；④记忆口诀：**先模型、再测量、后要求、终评价**。

过渡：这页讲的是标准的"编号体系"，接下来的问题更本质：一个软件的质量，其实分层存在——内部质量、外部质量、使用质量。它们是层层影响的关系，我们看下一页那张链条图。

【2.18】内部质量→外部质量→使用质量（2 分钟）

这一页是一张关系图，讲的是质量的三层存在形态。请大家看图上从左到右的走向：内部质量、外部质量、使用质量，三者的箭头是"影响"——内部质量影响外部质量，外部质量影响使用质量；反过来看，箭头是"依

赖于"——使用质量依赖于外部质量，外部质量依赖于内部质量。同时右边还有一个"使用语境"作用于使用质量。

怎么理解这三层？内部质量是"看不见的质量"，指的是代码、文档这些中间产品本身的属性，靠内部度量来看；外部质量是"看得见的质量"，指的是软件运行起来之后表现出来的行为，靠外部度量来看；使用质量是"用得着的质量"，指的是用户在他的真实使用语境下能不能顺利完成任务，靠在使用中度量。

举个我们身边的例子：一个登录模块，代码里密码是明文比较、没有失败次数限制，这是内部质量问题；用户实际登录时能被暴力破解、后台报错，这是外部质量问题的表现；而在真实业务里，用户因此被盗号、被投诉，这就是使用质量被破坏的结果。

所以这三层不是三种无关的质量，而是同一件事的三个观察距离：**离代码越近越早能发现，离用户越近越能说明危害**。测试的价值就在于把远处的危害，用近处的、早期可测的指标提前识别出来。

还要说清一点：这三层不是三张彼此独立的检查表，而是一条证据链——内部指标异常可以预测外部行为的风险，外部行为异常可以预测用户使用效果的损失。测试报告里如果能把这条链串起来，说服力会强得多。

板书：画一条横向链：**内部质量 —影响→ 外部质量 —影响→ 使用质量**；在下方画反向箭头，标注"**依赖于**"；右侧写"**使用语境 → 影响 → 使用质量**"；再补三层度量名：**内部度量｜外部度量｜在使用中度量**，旁注一句"**越靠内部越早可测，越靠使用越能说明危害**"。

提问：如果测试资源只够盯一层，你应该盯哪一层？请说出理由。

预设回答：

- 答"盯使用质量，因为用户只在乎这个"：⚠ 重要但不可行。点评：使用质量最贴近价值，但它**只能在真实语境里长时间观察**，早期的课程设计项目根本没有真实用户；所以实践中要靠外部度量做替身。

- 答"盯内部质量，代码写好了自然没问题"：❌ 典型的"技术自负"。点评：内部干净不等于用户能用，**代码质量只是必要条件**；而且内部指标容易自己糊弄自己（注释率高不代表注释有用）。

- 答"三层都要有指标，但可以用内部和外部指标提前预测使用质量"：✅ 这是最专业的回答。点评：这正是模型画的"影响/依赖于"关系要表达的——**三层是层次递进、可传递的证据链**，早期用内部、中期用外部、上线后用实际使用度量。

易错点 / 考点：①箭头方向常考：**内部影响外部、外部影响使用；使用依赖外部、外部依赖内部**，别把"影响"和"依赖"方向写反；②"使用语境"这个要素只有使用质量这一层才有；③考点形式：判断题"内部质量好则

使用质量一定好"（错，是影响不是等同）；④记忆口诀：**内影响外、外影响用；用依赖外、外依赖内；语境只在最外层**。

过渡：三层里最靠内、也最早能测的那一层就是内部度量，它到底量什么、有什么好处？我们看下一页。

【2.19】内部度量（2 分钟）

这一页讲内部度量。课件上给了两张图，主题是内部度量的对象和它的价值，备注里有一句英文提示——Economic risk mitigation，也就是"缓解经济风险"，另外还点出 ISO/IEC 9126-1:2001 把软件质量模型分成内部质量与外部质量模型、使用质量模型。

内部度量量的是什么呢？量的是**中间产品自身的可测量属性**：代码和设计文档的规模、结构复杂度、模块之间的耦合程度、注释情况、需求到代码的可追溯性等等。这些东西有一个共同特点——**不需要把系统完整跑起来就能测**，静态评审、静态分析工具就能给出数据。

它的价值就在那句"缓解经济风险"上。软件工程里有个普遍规律：缺陷发现得越晚，修复代价越高，因为越晚就意味着越多的设计、代码和测试已经建立在错误的基础之上。内部度量让团队在编码阶段、甚至在评审阶段就拿到质量数据，从而把修复成本压在最便宜的那个时间点上。

但请注意内部度量的局限：它测的是"中间产品的属性"，不是"用户能感知的效果"。圈复杂度低不代表用户能顺利下单，注释率高也不代表注释说得对。所以内部度量只能告诉你"哪里可能有问题"，不能替你做"能不能交付"的最终判断。

再给一个实操建议：内部度量最好成组使用。只看复杂度容易被误导，但把"复杂度偏高＋被频繁修改＋评审中问题集中"三个信号叠加起来，高风险模块基本就锁定了，测试和评审资源就该优先压到这些模块上，这就是最朴素的数据驱动测试。

板书：写"**内部度量＝量中间产品本身**"，下面列：**规模 · 复杂度 · 耦合度 · 可追溯性 · 注释/规范符合度**；再写两个关键词："**不需要运行程序**"＋"**Economic risk mitigation（缓解经济风险）**"；旁边画一条代价曲线并标注"**越晚发现，修复越贵**"。

提问：既然内部度量不能证明用户满意，为什么还要花人力去做静态分析和代码评审？

预设回答：

- 答"为了提前发现问题，省钱"：✅ 对了，就是"缓解经济风险"。点评：追问一句"省在哪里"——**省在还没有把错误放大器做大的时候修复**，一个需求理解偏差在需求阶段改是写几行字，在上线后改可能是全量回归。

- 答"因为标准要求做，属于流程合规"：⚠ 只答了形式。点评：合规只是手段，真正目的是**让不可见的东西变得可见**；代码质量数据是团队判断"能不能进入下一阶段"的门禁依据。

- 答"内部度量好，那就以圈复杂度为唯一指标考核开发"：❌ 非常危险的做法。点评：单一指标一定被"优化"（程序员把函数拆碎，复杂度好看但可读性更差），**内部度量要成组看、结合评审判断**，不能拿来当 KPI 硬压。

> **易错点 / 考点：**①内部度量**不需要运行程序**，这是它与外部度量最本质的区别；②不要把内部度量与静态测试混为一谈：**内部度量是"量数据"，静态测试是"找缺陷"**，两者常同时进行但目的不同；③考点形式：判断某项活动属于内部度量还是外部度量（如"统计圈复杂度"属内部，"测量登录响应时间"属外部）；④口诀：**内部量自己（代码文档）、外部量行为、使用量效果**。

> **过渡：**内部、外部、使用这三级到 ISO 25010 里被重新组织成了两个模型：一个管产品本身，一个管用户使用。先看产品质量模型。

【2.20】产品质量模型（3 分钟）

这一页是产品质量模型。课件给了两个关键信息：一个是 ISO 官方的标准在线浏览地址（iso.org 的 OBP 平台，标准号是 ISO/IEC 25010 第 1 版），另一个是 **GB/T 25000.10-2016 质量模型**——也就是说，我们国家等同采用的标准就是它，考试和写测试方案时引用这个国标号最稳。

产品质量模型要回答的问题是：**软件这个"产品"本身，应该从哪几个维度评价？** 在 GB/T 25000.10 里，产品质量特性有八个：功能适合性、性能效率、兼容性、易用性、可靠性、安全性、可维护性、可移植性。

我们来对比一下它和 ISO 9126 的六大特征：可靠性、易用性、可维护性、可移植性这四项保留了下来；功能性细化为"功能适合性"，强调的是"功能是否满足规定和隐含需求"，而不只是"有没有这个功能"；效率改称"性能效率"，把时间行为和资源利用都纳进来；新增了两项——**兼容性**（与其他产品共享环境、交换信息的能力，比如能不能和现有系统对接、能不能和别的软件共存）和**安全性**（信息保密性、完整性、抗抵赖、可核查、真实性）。

为什么要新增这两项？看看今天的软件形态就明白了：系统都是联网的、都要集成、都存着用户数据。**在**

> **9126 的年代，安全性和兼容性是"附加项"；在今天，它们是"入场券"**——一个不能对接的单机系统，在业务上几乎没有价值；一个泄露用户数据的系统，功能再全也是负分。
>
> 还有一层：产品质量模型里的这些特性都是"产品属性"，必须靠需求、设计、编码、测试共同实现，不能指望某个阶段事后补出来。所以需求评审时就该问：这个需求的兼容性要求是什么、安全要求是什么，而不是等系统做完再补一轮安全测试。

> **板书：**写"**产品质量模型（GB/T 25000.10-2016 / ISO/IEC 25010）**"，下面竖排八个：**功能适合性 · 性能效率 · 兼容性 · 易用性 · 可靠性 · 安全性 · 可维护性 · 可移植性**；右侧写"**相比 ISO 9126：保留 4 项，功能性→功能适合性，效率→性能效率，新增 兼容性＋安全性**"。

> **提问：**为什么标准要把"安全性"从功能性下面的一个子特性，提升为一个独立的质量特性？

> **预设回答：**

- 答"因为安全问题越来越严重"：✅ 方向对。点评：更准确地说，是**安全已经不只是"某个功能的属性"，而是贯穿全部功能的横切要求**——登录、支付、上传、日志都涉及，放在功能性子项里会被漏掉。

- 答"提升层级只是标准改版的形式调整，没有实质意义"：⚠ 低估了标准演进。点评：标准属性的"层级位置"决定了**评审和测试的关注度**；一个条目从子特性升到特性，意味着它必须有独立的测试类型、独立的验收门槛。

- 答"安全是运维和安全团队的事，测试不用管"：❌ 必须纠正。点评：安全性作为产品质量特性，**在测试范围内**，常见的有认证授权、输入校验、敏感信息保护等测试项；出了问题，测试同样是责任方之一。

> **易错点 / 考点：**①标准号对应要准：**产品质量模型＝ISO/IEC 25010（GB/T 25000.10-2016）**；②八个特性最容易漏掉"兼容性"和"安全性"；③不要把"性能效率"写成"性能"或"效率"就草草带过，标准含义是**时间行为＋资源利用**；④考点形式：给出场景判断属于哪个特性（"换数据库后系统仍能运行"→可移植性；"与第三方支付接口正常交互"→兼容性；"越权访问被拦截"→安全性）；⑤记忆口诀：**功能性能、兼容易用、可靠安全、可维护可移植**。

> **过渡：**产品质量模型管的是"产品本身好不好"，可用户其实不关心产品，只关心"我用得顺不顺"。这就轮到另一个模型——使用质量模型。

【2.21】使用质量模型（2 分钟）

这一页是使用质量模型。它和上一页的产品质量模型是一对：产品质量模型站在**产品**角度问"这个软件本身有哪些质量维度"，使用质量模型站在**用户**角度问"用户在实际使用语境里，能不能达成他的目标"。课件这一页也是以图为主，备注同样提示了 Economic risk mitigation 和 ISO/IEC 9126-1:2001 的两分法。

使用质量的特性，按现行的 GB/T 25000.10／ISO 25010，一般列五个：有效性、效率、满意度、免于风险、语境覆盖。有效性是说用户能不能准确、完整地达成目标；效率是说达成目标要花多少时间、多少成本；满意度是用户的主观感受，包括有用性、可信度、愉悦感、舒适度；免于风险是说软件有没有给用户的经济、健康、数据安全带来风险；语境覆盖是说它能不能在用户真实的使用场景下（不同设备、不同环境、不同任务）都成立。

我提醒大家注意一点：在早期的 ISO 9126 里，使用质量是四项——有效性、生产率、安全性、满意度；到了 25010 时代扩展成上面这五项，把"风险"和"语境覆盖"单独提出来。这背后的变化很有味道：**标准开始承认——同一个软件在不同的使用语境下，使用质量可以是完全不同的**。手机上流畅，不代表在弱网的山村里可用；办公室里好用，不代表戴着手套的车间工人能用。

再把它和验收测试挂上钩：验收测试的判定准则，本质上就是使用质量的指标化——任务完成率、平均完成时间、严重问题数、无障碍可用程度等等。如果一份验收方案里只有"功能是否符合需求"这一条，那它评的是产品，不是使用。

> **板书：**写"**使用质量模型（用户视角）**"，横排五个：**有效性 · 效率 · 满意度 · 免于风险 · 语境覆盖**；下面写对比："**ISO 9126 时代四项：有效性 · 生产率 · 安全性 · 满意度 → 25010 扩展为五项**"；右侧写一句"**同一软件，不同语境，使用质量不同**"。

提问：请举一个"产品质量不错、但使用质量很差"的例子，并说明差在哪一项上。

预设回答：

- 答"功能齐全但老年人不会用，满意度低"：✅ 很好的例子。点评：这属于**易用性与满意度**问题，它提醒我们——产品的功能适合性达标，不代表使用质量达标，**用户的达成成本和感受是独立维度**。

- 答"产品质量和使用质量是一回事，只是叫法不同"：❌ 概念混淆。点评：两者观察对象不同——**前者看产品属性，后者看使用结果**；一个产品可以特性全达标却在真实语境里失败（比如强光下屏幕看不清）。

- 答"只要用户投诉多就是使用质量差"：⚠ 不完整。点评：投诉只是满意度的信号，使用质量还要看**有效性**

（任务是否真的完成）和免于风险（有没有造成损失）；很多失败的用户根本不投诉，直接流失。

易错点 / 考点：①两套模型不要张冠李戴：**产品质量模型 8 特性，使用质量模型 5 特性**；②"语境覆盖"只出现在使用质量模型里，是 25010 的新增项；③考点形式：给出用户场景判断属于使用质量的哪一项（"新用户 3 分钟完成注册"→有效性/效率；"系统不会误导用户造成资金损失"→免于风险）；④口诀：**有效、高效、满意、无风险、语境覆盖**。

过渡：光讲模型还是抽象的，下一页课件给了一个具体例子——把它套到 Web Portal 上看，使用质量到底怎么评。

【2.22】示例：Web Portal 的使用质量（2 分钟）

这一页是示例，课件用一个 Web Portal（门户网站）来说明"使用质量"怎么落地。所谓门户，通常是企业的信息入口：登录、导航、公告、单点登录到各个业务系统。我们就把刚才那五项使用质量特性，一项一项套到这个门户上。

第一项有效性：用户能不能真的完成他要做的事——登录成功、找到要办的事项、点进正确的业务系统，而不是在导航里迷路。第二项效率：完成这些事要花多少时间和步骤，比如从首页到目标系统要不要点五次、加载要不要等十秒。第三项满意度：用户对界面的信任感和舒适度，包括信息是不是准确、样式是不是一致、提示是不是清楚。第四项免于风险：门户里往往挂着单点登录和员工信息，一旦被冒用或者跳转到钓鱼页面，损失是直接的，所以风险要单独评。第五项语境覆盖：员工在电脑上用、在手机审批时用、在弱网的外勤现场用，效果可能完全不同。

大家可以发现，这里用的方法很朴素：**把抽象的质量特性，翻译成具体用户能感知的问题**。这正是使用质量评价的核心动作——不是问"这个系统质量好不好"，而是问"哪一类用户，在什么语境下，完成哪一件事，会失败"。

再补一层：门户的使用质量评价通常不是一次性结论，而要分语境、分角色来看。同一套指标，对天天使用的老员工和偶尔登录的新员工，答案可能完全相反；对电脑端和对手机端，也可能一个通过、一个不通过。所以报告要按语境分组给结论，而不是给一个全局平均分——平均值最擅长掩盖问题。

板书：写"**示例：Web Portal 的使用质量**"，左侧列五特性，右侧并列写出问句：**有效性→能否完成登录/办事；效率→要几步・等多久；满意度→是否信任舒适；免于风险→账号与信息是否安全；语境覆盖→PC/手**

> **机/弱网是否都可办**；下方写一句总结："**把特性翻译成用户能感知的问题**"。

> **提问：**如果只能选三项指标来评价这个门户的使用质量，你选哪三项？为什么？

> **预设回答：**

- 答"有效性、效率、免于风险"：✅ 很务实。点评：这三项**能直接量化、且直接对应业务损失**——办事失败、耗时过长、账号被冒用，都是能被统计和追责的。

- 答"满意度最重要，发个问卷就行了"：⚠ 只靠主观数据不够。点评：问卷受样本和情绪影响大，**要与客观指标互相印证**；满意度高但任务完成率低，往往说明用户根本没意识到自己用错了。

- 答"门户是内部系统，风险可以不用评"：❌ 危险的判断。点评：门户通常是**单点登录的入口，是攻击价值最高的一环**；它一旦失守，后面所有业务系统一起暴露。

> **易错点 / 考点：**①示例类题目的答题结构：**先定用户与语境 → 再按特性逐项分解 → 最后给可测指标**；②最常见的错误是只说界面感受，不给可测口径；③考点形式：给一个系统（如选课系统、外卖 App）让你按使用质量五特性列评价角度；④口诀：**先问谁在用、在哪用，再问能不能办成、要多久、信不信、险不险**。

> **过渡：**从这一页开始，我们的关注点从"什么是好质量"转到反面：质量出了问题，就是缺陷。先给缺陷下一个正式定义。

【2.23】2.1.2 软件缺陷的定义（3 分钟）

我们进入 2.1.2，软件缺陷的定义。课件上给的核心表述是：**任何程序、系统中的问题，比如与产品设计书的不一致性、不能满足用户的需求，都属于软件缺陷。**这句话虽然短，但包含了两个很关键的判断来源，我拆开讲。

第一个来源是"与产品设计书的不一致性"。也就是说，规格说明、设计文档是基准，程序的行为跟基准不符，就是缺陷。比如需求书写着"密码长度不少于 8 位"，程序却允许 6 位，这就是不一致，不需要讨论。

第二个来源是"不能满足用户的需求"。请注意这一句的分量——它意味着**即使程序完全符合设计书，只要用户的需求没有被满足，仍然算缺陷**。为什么？因为设计书本身可能写错了、写漏了。用户说"我要能导出 Excel"，需求文档里没写，程序自然也没有这个功能，从"一致性"角度它没错，从"满足需求"角度它就是缺陷。

这两个来源正好构成了我们后面反复用的两条判定思路：**对照规格（正向思维：该做的做了没有）**和**对照用**

户期望（反向思维：还有什么情况让他失望）。课件这一页还配了图，表达的也是同样的意思——缺陷的判定不能只看代码对错，还要看它是否偏离了设计和用户的期望。

所以大家记住一个判断口径：**缺陷不是"程序里有错误"这么窄，而是"任何让产品与设计或用户期望产生偏离的问题"**，它可能藏在代码里，也可能藏在需求、设计、文档甚至配置里。

还有一个实务提醒：判定缺陷时一定要写清"期望"和"实际"。只说一句"这样不对"，开发会反问"那应该怎样"；把规格条款或用户场景引出来，问题才是可讨论、可裁决的。这也是缺陷报告里最容易被新手省略、又最不能省略的一栏。

> **板书：**写"**2.1.2 软件缺陷的定义**"；下面两条并列：**① 与产品设计书不一致（对照规格）／② 不能满足用户需求（对照期望）**；右侧补一句："**判定缺陷的两把尺子：规格 + 用户期望**"；再写"**缺陷可以藏在需求、设计、代码、文档、配置里**"。

提问：需求文档漏写了一个功能，程序按文档做完了，测试该不该把它记为缺陷？请说明理由。

预设回答：

- 答"该记，因为用户需求没被满足"：✅ 符合课件口径。点评：但要点出处理方式——**这类问题的根因在需求，不在代码**，报告时要写清"需求遗漏"，让责任流向正确的位置，而不是直接指责开发。

- 答"不该记，程序符合规格说明就是对的"：❌ 很常见，但只对了一半。点评：符合规格是"通过下限"，不是"质量上限"；**用户期望才是最终验收标准**，漏需求造成的问题最终仍会以用户投诉的形式回到团队。

- 答"记也可以，但要等需求方确认再说"：⚠ 流程上没错，时机上要小心。点评：**先把发现记录下来，再走确认流程**，不要因为"口径未定"就不记录；很多缺陷就是这样在讨论中消失的。

易错点 / 考点：①缺陷定义的两个来源要能同时说出来，只答"程序错误"是不完整的；②"与设计书不一致"和"不满足用户需求"可能同时成立、也可能只成立一条，考试常给场景让你判断属于哪一种；③考点形式：判断"符合规格说明但用户不满意"是否算缺陷（算）；④记忆口诀：**两把尺子量缺陷——一把量规格，一把量期望**。

> **过渡：**定义讲完了，我们来看一个有意思的历史现场：软件缺陷这个词，最早是怎么来的？请看"First Bug"。

【2.24】First Bug（3 分钟）

这一页标题叫 First Bug，"第一只虫子"。图上的人物是 Grace Hopper，格蕾丝·霍珀，1906 到 1992 年，她是计算机史上的传奇人物：参与早期计算机的研制、推动了编译器的发展，被称为 COBOL 之母，也是美国海军的少将。

课件这一页讲的是那个著名的故事：早期计算机使用大量机电继电器，某次机器故障排查时，工程师从继电器里取出一只飞蛾，把它贴进维护日志，并写下"发现 bug 的第一个实际案例"。从此，"bug"这个词在计算机领域就流行开来，排除故障也被叫做 debugging——"捉虫"。这里补一个更严谨的细节：bug 这个说法在更早的工程领域就已经用来指代故障了，但这只飞蛾让它在计算机行业里彻底流行，成为我们今天通用的术语。

我想请大家注意这个故事里最易被忽略的一点：**为什么工程师要把一只飞虫郑重地贴进日志本、还写上一句话？** 因为这只虫子不是一个笑话，这是一份**缺陷记录**——记录现象、保留证据、说明原因、供后来人查阅。这正是本课要培养的基本功：发现问题要如实记录，留下可追溯的证据，而不是口头说一句"刚才有点问题，重启一下好了"。

同学们，测试工作看起来是在跟代码打交道，本质上是在跟"诚实"打交道。把不利于自己的结果隐去、把偶发问题当作没发生，短期看省事，长期看就是把风险留给用户。霍珀那一代人留下的不只是"bug"这个词，还有一种一丝不苟的记录习惯。

再深一层看，记录习惯背后是工程的可重复性：只有把现象、环境、步骤写下来，别人才能复现、才能验证修复、才能在下次遇到相似问题时快速定位。测试人员交出去的最有价值的资产，往往不是"我发现了 bug"，而是一份别人照着就能复现的问题记录。

板书：写"**First Bug —— Grace Hopper（1906–1992）**"；下面写 **1947 年哈佛 Mark II 继电器中的飞蛾 → 贴入维护日志 → "第一个实际的 bug" → debugging（捉虫）**；右侧写一句落点："**发现 → 记录 → 留证据 → 可追溯**"。

【思政落点 1】讲什么故事：霍珀团队把一只飞蛾贴进日志本并写明"这是第一个实际发现的 bug"。怎么联系知识点：软件缺陷定义要求"可追溯的问题记录"，缺陷报告的最小要素就是现象、环境、证据、影响。落到什么价值观：**科学精神与职业诚信——如实记录每一个问题、不隐去不利结果，是测试人员的职业底线，也是工程文化的基石。**

提问：如果一次测试中发现的问题是偶发的、复现不了，你会怎么处理？直接忽略还是记录？

预设回答：

- 答"记录下来，标注偶发和已尝试的复现步骤"：✅ 正确做法。点评：偶发问题往往是**并发、时序、资源竞争**类缺陷的前兆，最容易被忽略，也最容易在生产环境放大；记录时要把环境、时间、操作序列都留下来。

- 答"复现不了就当没有，省得浪费开发时间"：❌ 这是典型的隐瞒式处理。点评：把"我没法复现"变成"系统没问题"，本质上是**用省略换安心**；真出事故时，团队连线索都没有。

- 答"先记，但优先级放低，等再出现时合并"：✅ 稳妥。点评：补充一句——**允许低优先级，但不允许不记录**；同时要注明"暂不可复现"，方便后续出现同类现象时快速关联。

易错点 / 考点：①年代与人名对应：**Grace Hopper，1906–1992，First Bug 故事**；②"bug 一词由这只飞蛾首创"是流传很广的误解，准确说法是它让该词在计算机领域流行；③考点形式：名词解释"debugging"，或简答"从 First Bug 看缺陷记录应包括哪些要素"；④口诀：**先记下来，再想办法复现**。

过渡：认识了"第一只虫"，我们回到术语本身——缺陷在中文和英文里有非常多的说法，它们之间有微妙的差别，这正是下一页那张对照表要解决的问题。

【2.25】缺陷 – Defect, Bug（3 分钟）

这一页的标题是"缺陷 – Defect, Bug"，课件用一张表列出了缺陷在中英文里的各种说法，一共九组：缺点 defect、偏差 variance、谬误 fault、失败 failure、问题 problem、矛盾 inconsistency、错误 error、毛病 incident、异常 anomy。大家第一眼可能觉得"这些不都差不多吗"？请注意，正是这些词的混用造成了团队沟通和缺陷统计里的大量扯皮，所以要把它们的关系理清楚。

所以这三者不是同义词替换，而是一条因果链。**error（错误）**通常指人犯的错，比如需求理解错了、代码写错了一位。**fault 或 defect（谬误／缺陷）**指的是留在产品里的那个错误状态，比如代码里那行错误的判断条件，它是静态存在的。**failure（失败／失效）**指的是运行时表现出来的不正常结果，比如用户提交表单后系统崩溃。

三者构成一条链：**人犯 error → 产品里留下 fault/defect → 运行时出现 failure**。这也是为什么缺陷管理里必须区分"根因"和"现象"：用户看到的是 failure，工程师要找的是 fault，而真正要改进流程去解决的是 error 的产生原因。

剩下的几个词也有各自的语境：variance（偏差）常用在"实际结果与预期不一致"；inconsistency（矛盾）

强调与需求、文档之间的相互冲突；problem（问题）是最宽泛的口语说法；incident（毛病／事件）在运维和缺陷追踪系统里常指一次故障事件；anomy（异常）指偏离常规的状态。**术语可以多，但描述缺陷时的口径必须统一**，这也是缺陷报告要使用标准字段和统一分级的原因。

> **板书：**写"**缺陷的多种说法**"，左中右三列对照：**缺点 defect｜偏差 variance｜谬误 fault｜失败 failure｜问题 problem｜矛盾 inconsistency｜错误 error｜毛病 incident｜异常 anomy**；下面画链条：**error（人为错误）→ fault / defect（产品中的缺陷状态）→ failure（运行中的失效）**；旁注"**用户看到 failure，工程师找 fault，流程要治 error**"。

提问：线上用户报"系统崩溃了"，工程师说"这是环境问题不是缺陷"。这句话问题出在哪？

预设回答：

- 答"环境问题也是缺陷，因为用户受到了影响"：✅ 直击要点。点评：要补充专业表述——**用户感知到 failure，就必须先立案追踪**，至于根因是代码、配置、环境还是数据，是调查结论，不能作为"不记录"的理由。

- 答"工程师说得对，代码没问题就不是 bug"：❌ 混淆了 fault 和 failure 的层次。点评：**缺陷的定义看的是"偏离期望"，不是"代码有没有错"**；环境适配失败本身就是可移植性或兼容性上的质量问题。

- 答"两边说的不是一回事，各说各的"：⚠ 看到了术语分歧，但止步于此。点评：这恰恰说明**团队需要统一的缺陷定义和分级口径**，否则同一件事有人算 bug、有人算配置问题，统计数字永远对不上。

易错点 / 考点：① error / fault(defect) / failure 三者的层次关系是本节最核心的考点，务必按"人—产品—运行"这条线记；②九个英文词不要张冠李戴，尤其 fault 与 failure 常被互换；③考点形式：给出场景判断属于 error、fault 还是 failure（"程序员把 '>=' 写成 '>'"→error/fault，"边界值输入时结果错误"→failure）；④记忆口诀：**人错 error、物错 fault、跑错 failure**。

> **过渡：**术语口径统一之后，还要有权威定义做依据。下一页我们就看两个标准的原话：IEEE 729 和 ISO 29119。

【2.26】软件缺陷（3 分钟）

这一页给出软件缺陷的权威定义，课件引了两处。第一处是 IEEE 729（1983 年）给出的标准定义，它从两个角度描述：**从产品内部看，软件缺陷是软件产品开发或维护过程中所存在的错误、毛病等各种问题；从外部看，软件缺陷是系统所需要实现的某种功能的失效或违背。**请大家注意这个"内外两看"的结构：内部看是"问

题存在"，外部看是"功能失效"，一个偏静态、一个偏动态，正好和前面讲的 error/fault/failure 呼应。

第二处是 ISO 29119 的定义，也是我们这门课后面做测试要常引用的标准。它把缺陷描述为：**组件或系统中会导致其无法执行所要求功能的瑕疵**；同时给出扩展说法：**任何依据需求规格说明、设计文档等基准而产生偏离的情况**，都算缺陷。下面还有一条非常重要的注（NOTE）：缺陷可能是在**评审、测试、分析、编译或使用软件产品及相关文档**的过程中被发现的——但不仅限于这些活动。

这条注把三件事说透了。第一，缺陷的发现手段不止"运行测试"一种，**评审、静态分析、编译告警乃至用户使用**都是发现渠道；第二，缺陷的载体不止程序代码，**文档、需求、设计**同样可能有缺陷；第三，只要偏离了基准或期望，就是缺陷，不需要等到它造成线上事故。

所以这一页请记住一句话：**缺陷是偏离期望的任何瑕疵，发现它的时机和手段可以多种多样，但判定它的尺子只有两把——基准（规格、设计、文档）和期望（用户需求）。**

最后给一个学习建议：这两个定义不要死背英文原句，要抓住它们共同的内核——偏离基准或期望，并且导致应有的功能无法实现。抓住内核，遇到没见过的表述，你也能判断它算不算缺陷。

板书：写"**软件缺陷的权威定义**"；左侧写 **IEEE 729（1983）：内部看＝开发/维护中存在的错误、毛病；外部看＝所需功能的失效或违背**；右侧写 **ISO 29119：会导致组件或系统无法执行所要求功能的瑕疵；任何偏离需求规格说明、设计文档等基准的情况**；下面写 NOTE 的三个发现渠道：**评审 · 测试 · 分析 · 编译 · 使用（不限于此）**，再画一条底线："**基准 ＋ 期望＝判定缺陷的两把尺子**"。

提问：ISO 29119 的注里说缺陷可能在"编译"和"使用"阶段被发现。请各举一个例子，并说明它们为什么容易被团队忽视。

预设回答：

- 答"编译告警说明代码有问题，但大家习惯性忽略告警"：✅ 非常真实。点评：大量空指针、未初始化变量、类型截断都藏在编译告警里，**把告警当噪音，就是把缺陷放进版本库**。

- 答"用户使用中发现的问题不算缺陷，因为没有用例"：❌ 需要纠正。点评：用户是真实语境下的测试者，**生产环境的用户反馈是缺陷的重要来源**；区别只在于它是最昂贵的一种发现渠道。

- 答"评审发现的只能算建议，不能记缺陷"：⚠ 概念不准。点评：评审发现的需求二义、设计错误都是缺陷，**只是缺陷的载体不是代码**；记录时应标明发现阶段，便于做缺陷来源分析。

易错点 / 考点：① IEEE 729 的"内部／外部"两分法是经典考法，要能准确复述；② ISO 29119 定义里的两个

要点——"无法执行所要求的功能"和"偏离需求规格说明、设计文档等基准"；③最容易漏的就是那条 NOTE：**缺陷发现不限于测试**，还可能是评审、分析、编译、使用；④考点形式：名词解释、判断"只有运行程序发现的问题才叫缺陷"（错）、简答"缺陷可能在哪些阶段被发现"；⑤记忆口诀：**内看问题、外看失效；偏离基准、即为缺陷；发现渠道、不止测试**。

过渡：到这里，第 2 章关于"质量"和"缺陷"的基本概念就讲完了——质量模型告诉我们什么是好，缺陷定义告诉我们什么是坏。接下来我们会用这些口径进入下一部分：缺陷是怎么分类的、测试又是怎么分类的。

【2.27】软件缺陷的现象（6 分钟）

前面我们已经知道什么是软件缺陷。可到了实际工作里，摆在测试人员面前的问题非常具体：屏幕上出现的这个现象，到底算不算缺陷？要回答它，先得学会认缺陷的"长相"。课件把软件缺陷的现象归纳成六类。第一类，功能、特性没有实现，或者只实现了一部分——需求说要支持批量导入，结果只能一条一条录，这就是没实现。第二类，设计不合理，本身就存在缺陷——两个模块都往同一张表里写数据，谁也没加锁，代码每行看着都对，可设计从一开始就是错的。第三类，实际结果和预期结果不一致——你输入三加五，它给你九，这是最典型的缺陷现象。第四类，运行出错，包括运行中断、系统崩溃、界面混乱——点一下按钮程序直接退出，或者弹出一堆重叠的窗口，用户根本没法用。第五类，数据结果不正确、精度不够——金额算出来多了两分钱，坐标算出来偏了几米，功能"跑通了"，但结果是错的。第六类，用户不能接受的其他问题，比如存取时间过长、界面不美观——这一类最容易吵架，因为需求文档里往往没写"多快算快""多好看算好看"，可用户就是不接受。这六类，你就把它当成一张筛查清单：拿到一个现象，先往这六类里对号入座，能对上号的，基本可以判定为缺陷；对不上号的，先别急着报，回去查需求、问用户。注意了，前三类和第四、第五类属于"硬缺陷"，有明确对错，可以直接判定；第六类属于"软缺陷"，要靠用户感受和业务约定来判定，这正是我们下一页要解决的问题。

板书：缺陷六现象 → ①功能未实现/部分实现 ②设计不合理 ③实际结果≠预期结果 ④运行出错（中断/崩溃/界面混乱）⑤数据不正确/精度不够 ⑥用户不能接受（存取慢/界面不美观）

提问：登录页面上，密码输入框比用户名输入框窄了 2 个像素。这算不算缺陷？依据是我们刚讲的哪一类现象？

预设回答：第一种答"不算，功能全都正常"——先肯定他关注了功能，再把他拉回第六类"用户不能接受的其他问题"，追问：如果不算缺陷，那 UI 走查时提的这个意见该叫什么？提示它属于界面设计与一致性范畴。第二种答"算，因为界面不好看"——肯定结论，但要求他说出对应哪一类现象，训练"用清单说话"的习惯，

不能凭感觉定性。第三种答"看需求或 UI 规范怎么写"——这是最佳答案，请他说出需求里没写时怎么办，自然引到"判断依据"上，为下一页的判断准则做铺垫。

易错点 / 考点：常见混淆有三种：把"数据结果不正确、精度不够"当成"功能没实现"；把"存取时间过长"当成性能没做，忽略了它在这页属于"用户不能接受"这一类；把界面问题一律判为缺陷，不看设计规范和用户约定。考试形式多为选择或判断："下列哪项不属于软件缺陷的现象"。判断三步法：一查需求有没有要求，二看结果对不对，三问用户能不能接受。

过渡：现象能对上号，可"预期结果"到底由谁说了算？同一段行为，有人说是缺陷，有人说是特性，这就引出了下一页——软件缺陷的判断准则。

【2.28】2.1.3 软件缺陷的判断准则（8 分钟）

上一页我们学会了从现象上认缺陷，但现象只是表面。一句"实际结果和预期结果不一致"，难点从来不在"实际结果"——跑一下就有；难点在"预期结果"从哪儿来。这个负责给出预期、并判定测试通不通过的机制，业界有个专门的名字，叫测试预言，Test Oracle。课件给的定义是：Test Oracle 就是决定一项测试是否通过的一种判断机制；它要求把被测试系统的实际输出与所期望的输出进行比较，从而判断有没有差异，也就是判断是不是缺陷。那么预期从哪儿来？课件列了七类依据，我们分三组记。第一组是最有分量的两个来源：需求规格说明书和其他需求、设计规范文档，这是第一位的依据；还有竞争对手的产品——用户常说"人家某某软件就是这么做的"，竞品可以当参照，但要谨慎，别人的行为不等于你的需求。第二组是四类偏自动化的预言：启发式测试预言，靠经验规则判断，比如"金额不可能为负数"；统计测试预言，靠统计规律判断，比如响应时间的分布有没有偏离历史基线；一致性测试预言，靠同一系统不同实现之间的一致性判断，比如新旧版本对同一个请求的结果应当一致；基于模型的测试预言，把一个模型当作正确答案的来源，拿实际输出跟模型推演的结果比。第三组是人类预言，Human oracle，说白就是由人来判断，靠测试人员的经验、业务知识，甚至最终靠用户的感受。这一页要建立一个意识：判断缺陷不是拍脑袋，而是有依据、可追溯的。你报一个缺陷，就必须说清楚"我拿什么当预期"。

板书：判断准则（Test Oracle）＝决定测试是否通过的比较机制：实际输出 vs 期望输出 → ①需求规格说明书/需求与设计规范文档 ②竞争对手产品 ③启发式预言 ④统计预言 ⑤一致性预言 ⑥基于模型的预言 ⑦人类预言

提问：测试一个计算器应用，输入 0.1 加 0.2，屏幕上显示 0.30000000000000004。你用什么当预期结果？请说出依据属于课件里的哪一类。

预设回答：答"预期是 0.3，所以是缺陷"——肯定他发现问题的敏锐，追问依据：需求文档有没有规定精度和舍入规则？若有，属于第一类依据；若没有，就要靠启发式预言（常识规则）或人类预言来判断，并说明这类问题在不同业务里结论可能不同。答"浮点数就这样，不算缺陷"——引导他区分"实现限制"与"用户可接受"，请他从用户视角判断这句结果用户能不能接受，再强调结论必须有依据、能说服人。答"用另一个计算器比对"——这是很聪明的答案，指出这属于一致性或竞品参照的思路，同时提醒：参照物的行为同样要先确认是对的。

易错点 / 考点：最常错的是把 Test Oracle 理解成"测试工具"或者"测试用例"，其实它是"判断是否通过的机制"。第二个混淆是把"人类预言"当成不专业、靠感觉，实际上人工判断在易用性、业务规则、界面审美上是不可替代的。考试形式常见为名词解释、选择（下列哪个不属于缺陷判断依据）和案例题（给出场景问依据是什么）。记忆口诀：物、竞、启、统、一、模、人——文档为准，竞品参照，四类预言兜底，人能拍板。

【思政落点 1】：讲"判断缺陷必须有依据"，可以联系工程中的标准意识：需求文档、设计规范、行业标准就是判断是非的尺子。测试人员手里的尺子如果有偏差，报出去的缺陷就站不住脚，团队会因此反复扯皮。做质量工作的人，既要有自己的判断，更要尊重客观依据，用数据和规范说话，而不是用嗓门大小说话——这就是科学精神在测试岗位上的具体落点。

过渡：知道了怎么判断缺陷，接下来要回答一个更根本的问题：这些缺陷到底是从哪儿冒出来的？

【2.29】2.1.4 软件缺陷的产生（7 分钟）

缺陷从哪儿来？课件把产生原因归成三大类。第一类是技术问题。里面又分三种：算法错误，思路本身就是错的；计算和精度问题，比如浮点数累积误差、单位换算错误；接口参数传递不匹配，A 模块传来的单位是厘米，B 模块当成米收下，单看两边都没错，一对接就出事。第二类是团队工作。这一条最容易被忽视，杀伤力却最大——沟通不充分、误解。评审会上大家都点头，散会后三个人脑子里装着三个版本的需求；接口约定只写在聊天记录里，没人落进文档；跨部门、跨地域的团队各做各的。这类缺陷不是谁技术差，是信息传递断了。第三类是软件本身的问题：文档错误，说明书写错了参数；用户使用场合，也就是 user scenario 考虑不周，没想过用户会在地铁里断网操作；时间上不协调或不一致带来的问题，比如定时任务和用户操作撞在一起、几台机器的时钟不同步；还有系统的自我恢复、数据的异地备份、灾难性恢复等，这些功能平时不跑，一跑就是大事故。课件这一页还挂了参考文献，主题是"缺陷驱动的过程改进"。它在告诉我们：分析缺陷产生的原因，不是为了追责某个人，而是为了回头改流程。同一个原因反复出现，说明流程有洞，而不是某个人不小心。给你一个判断经验：如果某个缺陷让你觉得"这怎么能怪开发"，那它大概率属于团队工作或软件本身这两类，解决方案在流程和文档里，不在某一行代码里。

板书：缺陷产生三来源 → 技术（算法错误／计算与精度／接口参数传递不匹配）｜团队（沟通不充分、误解）｜软件本身（文档错误、用户使用场合、时间不协调或不一致、自我恢复与数据异地备份/灾难恢复）

提问：一个接口联调失败：前端按"秒"传超时时间，后端按"毫秒"解释，结果请求 3 秒就超时。这个缺陷属于课件所说的哪一类原因？根子上该怎么修？

预设回答：答"技术问题，接口参数不匹配"——正确，但要追问他会不会把它只当代码 bug 修，提醒他真正的修法是补上接口契约与单位约定，属于流程改进。答"沟通问题"——也说得通，表扬他从团队角度看问题，说明技术类与团队类常常并存，评审与文档化是共同解药。答"前后端都有责任"——把它变成教学素材：缺陷归因的目的是定位可改进的环节，不是分摊责任，请他给出具体改进动作（统一单位、接口文档、联调用例）。

易错点 / 考点：常错在把"缺陷产生原因"与"缺陷现象"混为一谈：现象回答"看到什么"，原因回答"为什么会有"。另一处混淆是把"文档错误"归到技术问题，其实课件把它放在"软件本身"一类。考试常考分类题（下列哪项属于团队工作导致的缺陷）和简答（简述缺陷产生的三类原因）。记忆抓手：技术、协作、软件——写错了、没沟通清、想漏了场景。

过渡：三类原因是从性质上分，把缺陷按来源归好类之后，我们还得知道每一类各占多少、哪一类是重灾区，这就有了缺陷的构成。

【2.30】2.1.5 软件缺陷的构成（6 分钟）

接着看 2.1.5 软件缺陷的构成。上一页讲的是原因的大类，这一页要做的是把缺陷变成可以统计、可以量化的结构。所谓构成，就是把收集到的缺陷按来源或者根源做归类统计，看每一类各占多少。课件这一页配了两张图，出处标得很清楚：一张来自 2010 年 1 月发表在 International Journal of Industrial Engineering and Management 第一卷第四期第 155 到 161 页的论文，题目是《Requirements-based testing process in practice》，讲的是需求驱动的测试流程实践；另一张来自 scopemaster 的技术博客，题目是《软件缺陷的根本原因》。这里我要特别提醒：课件图上的百分比我不要求你们背，不同项目、不同统计口径差别很大，背数字没有意义；你要学的是看图的方法。图上的分类通常包括需求类、设计类、编码类、文档类、环境与配置类、人员与沟通类。看这张图要问三个问题：第一，哪一类贡献的数量最多？第二，严重级别最高的那些缺陷集中在哪一类？第三，上一次做过程改进以后，这一类有没有下降？这三个问题回答清楚了，过程改进才有靶子，否则所谓改进就是拍脑袋。回到课程设计上，你们后面要记录缺陷，我要求每条缺陷都标注来源类别和发现阶段，目的就是让你们最后能画出属于自己项目的那张构成图，用数据说明"我们的问题主要出

在需求还是编码"。

> **板书：**缺陷的构成＝按来源/根源归类统计 → 需求｜设计｜编码｜文档｜环境与配置｜人员与沟通；图谱出处：IJIEM 2010,1(4):155-161《Requirements-based testing process in practice》＋scopemaster《root causes of software bugs》；三问：哪类最多？严重缺陷集中在哪类？改进后是否下降？

> **提问：**如果你们项目统计出来的缺陷构成图显示：需求类缺陷占的数量最多，而且严重级别高的缺陷也几乎都在需求类。下个迭代你打算改哪三件事？

> **预设回答：**答"让开发小心点"——点评这个答案没抓住根源，需求类缺陷靠开发小心是治不住的，引导他往评审和需求确认上想。答"加强需求评审、需求要让用户确认、需求变更要走流程"——这是好答案，帮他排优先级：先做实需求评审的检查单，再补需求基线确认。答"测试人员提前介入需求评审"——很好，进一步点出这就是测试左移，让测试人员在需求阶段就提出二义性和可测性问题。

> **【思政落点 2】**：缺陷构成分析里有一个职业底线问题：不能只统计"好修好说"的缺陷，把难看的、自己漏掉的、指标不利的隐去。数据一旦被美化，过程改进就失去了方向，用户拿到了虚假的质量信号。将来你们做缺陷记录和测试报告，要如实记、如实报，不因为影响考核就删除或降级——诚信是质量工作者的第一块基石。

> **易错点 / 考点：**常错有两个：把"构成"当成"某一类缺陷的详细步骤"，其实它讲的是各类缺陷的比例结构；另一个是把构成图当成背数字的材料。考试常考简答（什么是缺陷的构成分析，它的作用是什么）和案例题（根据给定的缺陷清单做归类统计，并给出改进建议）。记忆抓手：归好类、看比例、找靶子、改流程。

> **过渡：**构成回答的是"哪一类多"，但缺陷还有一个更隐蔽的维度——它是在什么时候被引入、又在什么时候才被发现。

【2.31】软件缺陷在不同阶段的分布（7 分钟）

有了构成，再看分布。分布讲的是时间维度：缺陷是在哪个阶段被引入的，又是在哪个阶段被发现的。课件这一页给了两句关键结论。第一句：在真正的程序测试之前，通过审查、评审，就可以发现更多的缺陷。注意这个"真正的程序测试之前"，指的是代码还没跑起来、甚至还没写完的时候。为什么强调这一点？因为很多同学脑子里有个默认流程：写完代码交给测试，测试就是点点点。可事实上，评审和静态检查能抓到的缺陷数量相当可观，而且成本低得多。第二句更有意思：需求的缺陷会在需求评审、设计、编码、测试等过程

中逐步被发现，很难在需求分析这一个阶段就全部发现。这句话包含两层含义。一层是需求缺陷的影响会一路向下传：设计、编码、测试各个阶段都可能因为它出问题，所以你在编码阶段看到的一个 bug，根子可能在一周前的需求文档里。另一层是别指望需求评审一次就把需求缺陷清干净，缺陷的发现是一个随时间逐步收敛的过程，所以要反复评审、持续跟踪。这页的图把两个分布画在一起：一条是缺陷被引入的阶段分布，一条是缺陷被发现或被修复的阶段分布，两条线错开的那片区域，就是缺陷潜伏的时间。潜伏得越久，暴露得越晚，代价就越大——这就直接接到下一页。

板书：分布＝引入阶段 × 发现阶段 → 引入曲线 vs 发现曲线 → 两线错开区＝缺陷潜伏期；结论①真正的程序测试之前，审查/评审已能发现更多缺陷 ②需求缺陷会经需求评审→设计→编码→测试逐步发现，难在需求分析一次清空

提问：一个上线后才被用户发现的计算错误，追查发现是需求文档里"折扣按四舍五入"这句话没写清楚。请说出这个缺陷的引入阶段和发现阶段，并解释为什么它的代价特别高。

预设回答：答"引入在需求阶段，发现在上线后"——正确，追问中间的三个环节（需求评审、设计、编码、测试）为什么都没有拦住它，让学生看到每个环节都本可以拦住。答"是开发写错了"——借此纠正归因偏差：需求二义性没有被评审识别，问题在信息传递和评审有效性上，不全在编码。答"这种缺陷没法避免"——点评他悲观的部分有现实依据（需求缺陷很难一次清空），但要求他说出可以改善的动作，比如需求评审检查单、原型确认、用例反查需求覆盖。

易错点 / 考点：最常错的是把"缺陷引入阶段"和"缺陷发现阶段"当成同一个东西，画图时两条线重合。第二个易错点是把评审当成"走形式、浪费时间"。考试常考判断（缺陷大多是在编码阶段引入的）、识图题（读出两条曲线的错位区并说明含义）和简答（为什么说需求缺陷很难在需求分析阶段全部发现）。记忆抓手：引入是源头，发现是关口，两个关口错开多大，返工就多贵。

过渡：潜伏期越长代价越大——那到底大成什么样？课件用两个关键词给出了答案：劣质成本，以及非线性增长。

【2.32】2.1.6 修复软件缺陷的成本（8 分钟）

2.1.6 修复软件缺陷的成本。这一页有两个关键词。第一个叫劣质成本，英文缩写 COPQ，Cost of Poor Quality。缺陷带来的成本就叫劣质成本。注意这个名字的立场：它不是"测试成本"，也不是"开发成本"，而是"因为质量不好而额外付出的钱"。它包含什么？返工的人力、重新测试的时间、紧急发版的加班、线上事故的赔付、客户流失、品牌受损，甚至包括你反复开会澄清、写事故报告所花的时间。第二个关键词是：非

线性增长。课件明确说，缺陷不及时处理带来的成本很高，而且是呈非线性增长的。什么意思？缺陷留在系统里的时间越长，修复它的代价不是每天多一点点，而是越往后越陡。原因很直观：需求阶段的一个缺陷，当场改，就是改一句话、动一段文档；拖到设计阶段改，要改设计、改接口；拖到编码阶段改，要改代码，而且可能已经被几十处引用；拖到测试或上线以后才发现，还要加上定位成本、回归成本、错误数据的修复成本，以及对已经造成后果的追责成本。你还会发现一个规律：越晚发现的缺陷越难定位，因为代码和设计已经改了很多轮，现场都被破坏了。所以测试左移不是一句口号，它背后是一条经济规律。这里我要提醒一个常见误区：不要以为"测试花的时间越少，项目就越省"。省下来的测试时间，往往会在后期以几倍甚至几十倍的返工还回去。测试不是单纯的成本项，它是把风险成本提前锁住的保险。

板书：修复成本 → ①劣质成本 COPQ（Cost of Poor Quality）：返工＋重测＋紧急发版＋事故赔付＋客户流失＋品牌损失 ②非线性增长：需求→设计→编码→测试→上线，越往后越陡（定位/回归/数据修复/追责成本叠加）→ 结论：测试左移＝经济规律

提问：项目快到截止日期了，有人提议"这轮回归测试砍掉一半，能省三天时间"。请你用这一页的两个概念回应他，并给出你的判断。

预设回答：答"省三天，但上线后可能要赔三十天"——好答案，让他说清 COPQ 里具体包含哪些项目，把直觉变成可计算的账：三天测试人力对比潜在的返工、赔付与流失。答"砍了就砍了，风险我不管"——不要简单否定，把它当教学素材，请他写下这个决定的风险清单，再讨论谁为劣质成本最终买单。答"只能砍低风险模块的用例，高风险必须保住"——这是最职业的答案，表扬并补充：要基于缺陷分布和影响面排序，而不是平均砍一半，这就是风险驱动的测试。

【思政落点 3】：成本曲线讲的是"欠下的质量债早晚要还"，这正好对应职业责任：明知道某个功能没有验证充分，却为了赶进度签字放行，代价最终由用户承担——历史上不少事故就是这么发生的。同学们将来在岗位上，可能会面对"能不能先上线"的压力，请记住，你有责任把风险讲清楚、写清楚，而不是替别人承担不该承担的沉默。

易错点 / 考点：常错有两个：把 COPQ 理解成"测试部门的花费"，其实它是质量不好造成的全部额外损失；把非线性增长理解成"线性增长得更快一点"，忽略了曲线后半段的陡增。考试常见为选择题（COPQ 的含义）、判断题（缺陷修复成本随阶段线性增长——错）和案例题（比较在不同阶段修复同一缺陷的代价并给结论）。记忆抓手：质量不好多花的钱＝COPQ；越晚越贵，非直线。

过渡：到这里，2.1 软件缺陷就讲完了——从现象、准则、产生、构成、分布到成本。接下来我们要问：用什么办法把这些缺陷找出来？这就是 2.2 软件测试的分类。

【2.33】2.2 软件测试的分类（3 分钟）

从这一页开始进入 2.2，软件测试的分类。前面 2.1 讲的是缺陷：是什么、怎么判断、从哪儿来、怎么分布、代价多大。讲透了缺陷，接下来自然要问：用什么办法把它找出来？办法有很多，名字也很多——功能、性能、静态、白盒、回归……不先建立分类框架，这些名词就是一盘散沙，到了项目上也不知道该选哪个。所以 2.2 的任务，就是把这些方法挂到一棵分类树上去。这一节按六个步骤走：概念，这个维度是什么、什么场景用；做法，要掌握哪些知识和技术；举例，用案例帮你看懂怎么用；经验分享，讲理论与实际的差距；学习建议；以及对应的实用工具和模板。重点是分类维度和选择依据，不是背名词。学完你要能做到两件事：拿到一个项目，能说出该做哪几类测试；听到一个陌生名词，能判断它挂在哪个维度上。

板书：2.2 软件测试的分类 → 学习路径：概念（什么场景用）→ 讲解（要掌握什么）→ 举例（案例/Demo）→ 经验分享（理论与实际的差距）→ 学习建议 → 实用工具/模板；目标：拿到项目能选类型，听到名词能定位维度

提问：在进入分类之前先热身：请凭直觉说出你知道的三种测试名称，并试着说说它们分别是从什么角度来划分的。

预设回答：答"功能测试、性能测试、安全测试"——表扬，指出这三个都是从"测试目的"这个角度划分的，属于同一根主枝，让学生体会同一维度下可以有很多类型。答"单元测试、集成测试、系统测试"——指出这是从"测试对象或范围"划分的，与上一组不是同一个角度，正好说明为什么需要多个维度。答"白盒、黑盒"——指出这是从"是否针对内部结构"划分的，属于另一个视角。三种回答都记在黑板不同位置，直接变成分类树的雏形。

易错点 / 考点：常见错误是把不同维度的名词混在一起当成平级分类（比如"单元测试和性能测试哪个先做"），考核中常有"下列分类不属于同一维度的是"。判断三步法：先问"按什么分"，再问"这一维有哪些取值"，最后才问"该选哪个"。

过渡：框架交代完了，我们先看这张分类总览图，把整节的骨架装进脑子里。

【2.34】软件测试的分类（5 分钟）

这一页是 2.2 的分类总览图，一张图把整节的骨架画出来了。看这种图有个方法：先看它从哪几个方向分叉，再看每一根分叉下面挂了哪几片叶子。图的主干就是"软件测试"这四个字，往外伸出几根主枝，对应我们后面要逐个讲的分类维度。第一根主枝，按测试的对象或范围分：单元测试、集成测试、系统测试，一直到验

收测试，越往后测的对象越大、越接近用户。第二根主枝，按测试的目的分，也就是我们常说的测试类型：功能、性能、可靠性、安全性、兼容性、回归等等。第三根主枝，按被测程序有没有被执行来分：静态测试和动态测试。第四根主枝，按测试有没有盯着内部结构和实现算法来分：白盒和黑盒，中间还有灰盒。第五根主枝，按测试方法分：精准测试、变异测试、蜕变测试、MBT 这些相对进阶的方法。同学们看这张图的时候，脑子里的动作应该是：先把五根主枝记住，再把每一个你听说过的测试名词往枝上挂。挂得上去，说明你的知识是有结构的；挂不上去，说明你只是记住了名字。这里有一个非常重要的提醒：这些维度不是互相排斥的单选题，而是可以叠加的坐标。同一个测试活动，可以同时是系统测试、性能测试、动态测试、黑盒测试，并且用基于模型的方法来设计。所以以后不要问"这个项目到底该做黑盒还是白盒"，而要问"在哪个层次、为了什么目的、用什么视角、借助什么方法"。

板书：分类总览图 → 主干：软件测试 → 五主枝：①对象/范围（单元→集成→系统→验收）②目的/类型（功能・性能・可靠性・安全・兼容・回归）③是否执行程序（静态/动态）④是否针对内部结构（白盒/黑盒/灰盒）⑤方法（精准・变异・蜕变・MBT）；维度可叠加＝坐标系，不是单选题

提问：请给"用 Selenium 脚本跑一遍电商网站的下单流程，看页面和订单数据是否正确"这个活动打上四个坐标标签。

预设回答：答"系统测试＋功能测试＋动态测试＋黑盒测试"——标准答案，请他逐个说明理由，尤其解释为什么是黑盒（脚本只操作界面、不关心内部实现）。答"自动化测试、回归测试"——认可这两点，但指出自动化属于"执行方式"这一对，回归属于"测试目的"，维度不同，帮他区分。答"白盒测试，因为脚本能看到代码"——这是典型错误，借此讲清白盒的关键是"是否针对内部结构和实现算法来设计用例"，而不是"手里有没有代码"，脚本走的是界面，仍是黑盒。

易错点 / 考点：常错在把分类维度当成分等级、分先后，或者认为一个测试只能属于一类。考试常见为多选题（下列哪些维度可以用来给同一个测试活动打标签）和识图题（从分类图中指出某名词所属维度）。记忆口诀：范围、目的、跑不跑、看里看外、用什么法。

过渡：五根主枝先记牢，接下来我们逐个拆开看，先从最常用的按对象和目的分类讲起。

【2.35】不同维度的分类（8 分钟）

这一页把五个分类维度展开讲。第一个维度，按测试的对象或范围分，范围从小到大排：底层测试，也就是单元测试，测的是最小的可测单位，通常是一个函数、一个类或者一个模块；接口测试或集成测试，测的是模块拼起来之后能不能正确协作，重点在接口、数据传递和调用顺序；系统测试，把整个系统装进接近真实

的环境，按需求从头到尾验证；再往上就是业务层的验收测试，由用户或者业务方确认"这是不是我要的东西"。记住一条主线：越往下，测的是"零件对不对"，执行主体多是开发人员；越往上，测的是"整机能不能交付"，执行主体逐渐转向测试人员和用户。第二个维度，按测试的目的分，也就是测试类型：功能测试看功能对不对；性能测试看快不快、扛不扛得住；可靠性测试看长时间运行稳不稳；安全性测试看会不会被攻破；兼容性测试看在不同浏览器、操作系统、设备、数据下是否都正常；回归测试看改了一处有没有碰坏别处。第三个维度，按被测软件在测试过程中是否被执行，分为静态测试和动态测试。第四个维度，按是否针对系统的内部结构和具体实现算法，分为白盒测试和黑盒测试。第五个维度，按测试方法分：精准测试、变异测试、蜕变测试、MBT，也就是基于模型的测试，这些属于进阶方法，后面章节会逐个展开。这五个维度的顺序和课件一致，请大家按这个顺序记，不要打乱。

板书：五个维度 → ①对象/范围：单元（底层）→ 接口/集成 → 系统 → 业务层验收 ②目的/类型：功能・性能・可靠性・安全・兼容・回归 ③是否执行程序：静态 / 动态 ④是否针对内部结构与算法：白盒 / 黑盒 ⑤方法：精准测试・变异测试・蜕变测试・MBT

提问：一个银行 App 要上线新版本。请分别从"对象/范围"和"目的/类型"两个维度，各说出两个必须做的测试，并说明理由。

预设回答：答"单元测试和集成测试；功能测试和性能测试"——达标，请他把每个测试对应的对象说清楚，特别是集成测试要盯接口和转账对账的数据一致性。答"就做一个系统测试，覆盖全部功能"——借这个答案讲"层次不能互相替代"：系统测试跑通了，不代表单元内部的分支和边界都被覆盖，缺陷定位也会困难得多。答"安全性测试最重要，因为涉及资金"——认可风险判断，同时补充：安全和性能再重要也不能替代功能与回归，要按风险排优先级而不是做单选题。

易错点 / 考点：最容易错的是把"对象/范围"和"目的/类型"混为一谈，比如认为"系统测试"和"功能测试"是同一个维度的平级名词。第二个易错点是把"验收测试"只当成一个流程节点，忘了它是测试范围维度上的最高层。考试常见为分类连线题、多选题（下列哪些属于按测试目的分类）。判断三步法：先问"测的是谁"（范围），再问"为了什么"（目的），最后问"怎么测"（静态动态、黑白、方法）。

过渡：五个维度搭好了架子，但还有一组更细的说法需要单独讲——就是同样是做测试，方式上可以差得很远。

【2.36】软件测试方式（6 分钟）

这一页讲软件测试方式，课件列了四对，一共八种：静态测试和动态测试；主动测试和被动测试；手工测试

和自动化测试；基于脚本的测试和探索式测试。为什么要把它们成对摆出来？因为它们回答的是四个不同的问题：程序跑不跑？——这是静态对动态。测试人员要不要主动去驱动系统？——这是主动对被动。用例由人来执行还是由机器执行？——这是手工对自动化。用例是提前设计好了照做，还是边测边设计？——这是基于脚本的测试与探索式测试。请注意，每一对里的两个选项之间，不是"新的取代旧的"，也不是"好的和坏的"，而是各有各的适用场景，实际项目里往往是组合使用的。举个例子说明它们能叠加：一个电商系统的支付模块，你既可以做动态的功能测试，也可以做静态的代码评审；回归用例可以交给自动化跑，同时用探索式的手工测试专门去找自动化脚本想不到的路径；上线以后还要靠被动测试，从监控和日志里发现问题。所以学这四对，目标不是选边站，而是建立一个坐标——拿到一个测试任务，你要能说清楚它在四个坐标上分别落在哪里。接下来我们逐对来看。首先是静态测试和动态测试，这是最基础、也最能体现测试左移的一对。

板书：测试方式四对 → 跑不跑：静态 / 动态｜谁驱动：主动 / 被动｜谁执行：手工 / 自动化｜用例何时设计：基于脚本 / 探索式；四对＝四个坐标轴，可叠加组合使用

提问：请判断这句话对不对，并说明理由："自动化测试比手工测试先进，所以项目里应该尽量用自动化测试替代手工测试。"

预设回答：答"不对，有些场景必须手工"——追问是哪些场景，引导他说出探索式测试、易用性判断、需求频繁变动的部分，把结论落到"场景决定方式"。答"对，自动化效率高"——不直接否定效率这个事实，而是请他算一笔账：只跑一次的用例，写脚本的成本能不能收回来，让他自己发现自动化的前提是"可重复、稳定、量大"。答"两者结合，回归用自动化，新功能用探索式手工"——最佳答案，表扬并补一句：企业里这正是常见的分工方式，自动化守住既有功能，人去找未知的问题。

易错点 / 考点：常错在把四对方式当成互斥的选项，认为一个测试只能属于每对中的一边；第二个易错点是认为"自动化一定优于手工"。考试常见为判断、选择和简答（简述软件测试的几种方式并各举一例）。记忆口诀：跑不跑、谁驱动、谁执行、怎么设计。

过渡：四对里先看第一对，也是最基础、最容易考的一对——静态测试和动态测试。

【2.37】静态测试 vs. 动态测试（10 分钟）

静态测试和动态测试的区别，一句话就能说清：程序有没有跑起来。动态测试是通过运行程序发现错误——你设计输入、执行、观察输出，再跟预期比，我们在上一节做的练习基本都是这一类。静态测试不执行程序，而是通过人工评审、代码扫描和静态分析来发现错误，文档评审、代码走查、工具扫描都算。这里要划清几个边界。第一，静态测试查什么？查需求文档里的二义性和遗漏，查设计里不合理的地方，查代码里的空指

针、数组越界、资源没释放、命名和编码规范问题。这些问题如果留到运行时才发现，代价会大得多——有些问题比如内存泄漏，跑一次两次根本看不出来。第二，静态测试不等于"随便看看"。课件这一页特意列出了评审会议里的角色分工：主持人负责组织会议、控制节奏、确保议题不跑偏；作者是被评审对象的作者，负责讲解、答疑并记录问题；记录员负责如实记录每一个问题和决议，不能漏记、不能美化；列席人员旁听学习并提供补充视角；内审员负责检查过程是否合规；技术专业人员提供领域内的技术判断；用户代表则从需求和业务角度把关。有了这套分工，评审才是"有产出、可跟踪"的活动，而不是一次聊天。第三，不要以为静态测试就是评审，代码的静态分析主要靠工具来做，但人工的代码评审同样不可或缺，两者是互补的。最后给一个判断口径：凡是"我读了、我看了、我扫了"的，都是静态测试；凡是"我运行了、我点了、我发请求了"的，都是动态测试。

板书：静态 vs 动态 → 静态＝不执行程序：人工评审（走查/会议评审/互为评审）＋代码扫描/静态分析 → 查需求二义、设计不合理、代码规范与隐患；动态＝运行程序：设计输入→执行→比对输出；评审角色分工：主持人（组织控场）·作者（讲解答疑记录）·记录员（如实记录）·列席人员（旁听补充）·内审员（查合规）·技术专业人员（技术判断）·用户代表（需求业务把关）

提问：一个模块刚写完，还没运行过一次。用静态方法你能查出哪些类型的问题？请至少说出三类，并各举一个具体例子。

预设回答：答"规范问题、空指针、资源没释放"——达标，请他补充"需求或设计层面的问题"，比如接口参数单位没写清、异常分支没有设计，让他意识到静态测试不只看代码。答"没运行怎么知道对不对"——这是很有代表性的误解，借此讲清：静态测试查的是"能不能对"，而不是"跑出来对不对"，规范、结构、逻辑路径上的矛盾在纸上就能看出来。答"只能靠工具扫描"——补充人工评审的不可替代性，比如变量命名歧义、注释与代码不符、业务规则理解偏差，工具往往扫不出来。

易错点 / 考点：最常错的是把"静态测试"等同于"测试文档"或者"不重要的测试"；第二个易错点是以为静态测试只针对代码，漏掉需求与设计文档。考试常见为选择（下列哪项属于静态测试）、判断（静态测试不执行程序——对）和案例题（给定一段代码，指出可用静态分析发现的隐患）。记忆抓手：不动程序就是静态；评审看文档、走查看代码、扫描靠工具。

过渡：静态和动态讲的是"程序跑不跑"，接下来这对讲的是"测试人员要不要主动去驱动系统"——主动测试与被动测试。

【2.38】主动测试 vs. 被动测试（8 分钟）

第二对是主动测试和被动测试。主动测试，是测试人员主动出击：主动设计用例、构造输入、驱动系统、观察输出。我们平时课堂练习做的几乎都是主动测试——测什么、怎么测、什么时候测，控制权都在你手里。被动测试不一样，它不主动去打扰系统，而是在系统正常运行的过程中收集数据、观察行为，从日志、监控指标、用户反馈、缺陷报告、线上数据里发现问题。打个比方：主动测试像体检，你自己走到医院，医生按项目给你逐项查；被动测试像戴手环，不打扰你的生活，却持续记录心率、睡眠，异常了才报警。为什么需要被动测试？因为有些缺陷只在真实环境、真实数据、真实并发下才暴露，你在实验室里构造不出来；而且被动测试不会干扰用户，特别适合线上系统、生产环境和已经在跑的产品。它当然也有局限：不主动，就不知道系统在你没观察到的地方发生了什么；没有现成的预期结果可以对照，判断更多要靠基线、统计和趋势。所以在企业里两者是配合使用的：上线前靠主动测试把关，上线后靠被动测试感知。顺带说一句，很多同学把"被动测试"理解成"消极测试、随便测测"，这是完全错的——要做好被动测试，前提是对系统非常了解，知道该采集哪些指标、该盯哪些日志、该关注哪些异常模式，工作量一点都不小。

板书：主动 vs 被动 → 主动：测试人员设计用例、构造输入、驱动系统、观察输出（控制权在测试方，如功能/回归测试）｜被动：不干扰系统运行，采集日志・监控・用户反馈・线上数据，靠基线与统计判断（适合生产环境、灰度发布、Beta 与线上监控）｜关系：上线前主动把关，上线后被动感知，配合使用

提问：产品已经上线三个月，用户投诉"偶尔下单失败，但好像又自己好了"，而且无法稳定复现。这时你更该用主动测试还是被动测试？具体怎么做？

预设回答：答"被动测试，去看日志和监控"——正确，追问看什么：失败请求的时间分布、并发峰值、下游依赖的响应时间与错误码、是否有重试和自动恢复的痕迹。答"主动测试，我写脚本一直点下单"——肯定这个动作的价值，但指出难点在于"无法稳定复现"，先靠被动测试收集线索缩小范围，再用主动测试验证假设，让两者形成闭环。答"这种问题查不出来"——借机讲灰度与埋点的重要性：可观测性本身就是设计的一部分，没有日志和监控，被动测试就无从下手。

易错点 / 考点：最容易错的是把"被动测试"理解成被动消极、不设计、不投入；第二个易错点是认为被动测试只能发现问题不能定位问题，其实通过日志与监控的关联分析，被动测试经常能给出非常精确的线索。考试常见为选择、判断（被动测试需要运行系统但不主动输入——对）和案例题（给定线上问题场景，选择测试方式并说明理由）。判断三步法：谁在驱动？系统在不在真实运行？判断依据是预期还是基线？

过渡：主动和被动讲的是"谁来驱动"，接下来这对讲的是"谁来执行"——手工测试和自动化测试。

【2.39】手工测试 vs. 自动化测试（8 分钟）

最后一对，手工测试和自动化测试。手工测试，就是人按用例一步一步执行，自己观察、自己判断结果。自动化测试，是用脚本或工具来执行并比对结果，机器跑、机器判，或者机器跑、人来判。先破除一个常见误解：自动化测试不是"更高级的手工测试"，它不是万能的。哪些适合自动化？回归测试，因为用例稳定、要反复跑很多遍；性能测试和压力测试，要模拟成百上千并发，人根本做不了；大批量的数据驱动测试；以及持续集成里的冒烟测试。哪些必须靠手工？需求还在频繁变动、脚本写了马上就废的；探索式测试，靠人的直觉、经验和联想去找问题；易用性、界面美观这类靠主观感受判断的；还有一次性的、只跑一遍的验证。从投入上看，自动化有前期成本：写脚本、搭环境、准备测试数据，还有持续的维护成本。用例只跑一两次，自动化是亏的；用例要跑上百次，自动化才划算。还有一个很重要的经验：脚本会腐化，界面一改、接口一变，脚本就红一片，所以自动化测试本身也是需要长期维护的资产，不是写完就一劳永逸。课件这一页配了三张图，画的就是手工与自动化在不同场景下的分工，以及投入与产出的对比，大家结合图来理解：图的重点不是告诉你哪个更好，而是告诉你边界在哪里。

板书：手工 vs 自动化 → 手工：人执行、人判断（适用：需求多变、探索式、易用性主观判断、一次性验证）｜自动化：脚本/工具执行与比对（适用：稳定回归、性能与并发、数据驱动、持续集成冒烟）｜成本：自动化有编写＋环境＋数据＋维护成本，重复次数足够多才划算；脚本会腐化，需持续维护

提问：你们要在两周内交付一个学生选课系统，功能还在改，回归用例大约 30 条。你会选择把回归全部自动化吗？请说明理由。

预设回答：答"会，自动化效率高"——请他算账：30 条用例、两周内可能只跑两三次，而写脚本加调试可能要三四天，还要跟着需求变化改脚本，让他自己发现不划算。答"不会，需求还在变，脚本维护成本太高，先手工跑几轮，等需求稳定再挑高频用例自动化"——这是最职业的答案，表扬并补充：可以只自动化"核心且稳定"的若干条，作为冒烟用。答"自动化能发现更多缺陷"——纠正这个误解：自动化擅长"验证已知的预期"，不擅长发现未知问题，发现新缺陷更多依赖人的探索与思考，这正是探索式测试的价值所在。

易错点 / 考点：常错有三个：认为自动化一定优于手工；把"自动化测试"等同于"性能测试工具"，忽略它只是一种执行方式；忽略脚本的维护成本，以为写完就完了。考试常见为选择、判断（所有回归测试都应当自动化——错）和简答（比较手工测试与自动化测试的适用场景）。记忆抓手：机器擅长"反复核对已知"，人擅长"发现未知"；量大稳定交给机器，变化探索留给人。

过渡：四种测试方式我们讲了三对——静态与动态、主动与被动、手工与自动化，还剩最后一对：基于脚本

> 的测试与探索式测试，下一页接着讲。

【2.40】基于脚本的测试 vs. 探索式测试（4 分钟）

> 上一页我们对比了手工测试和自动化测试，这一页把第 2.2 节最后一组测试方式补齐，也是最有意思的一组：基于脚本的测试和探索式测试。
>
> 先说基于脚本的测试（Scripted Testing）。它的做法很直观：先把测试设计做完，把用例一条条写清楚，再按计划执行，执行完了再分析结果。也就是说，"测试设计、测试执行、分析"这三件事被拆成了先后阶段——先想清楚，再动手，最后复盘。
>
> 探索式测试（Exploratory Testing）刚好相反，它把学习、设计、执行、分析这四件事拧成一个循环：一边了解被测软件，一边设计下一步怎么测，马上执行，看到结果再回头修正自己的理解和思路。请注意，测试设计并没有消失，它只是从"事前一次性做完"变成了"执行过程中持续进行"。这个概念最早由测试专家 Cem Kaner 提出，他强调的是测试者的自由与责任并存。
>
> 打个比方。基于脚本的测试像照着菜谱做菜：菜谱写好了，按步骤下锅，味道不对就回去改菜谱。探索式测试像老厨师试新食材：先尝一口，再换个火候，边做边判断，手上没有现成菜谱，但脑子里一直在设计。
>
> 软件行业里这两种方式都在用，而且常常混着用。需求稳定、接口明确的回归场景，比如每次发版都要跑的登录、下单、支付主流程，适合用脚本化的用例守住底线；而产品刚上线、需求还很模糊、文档只有薄薄几页的时候，你写不出准确的用例，硬写出来的用例测的往往不是真风险，这时候就该让有经验的测试人员去"边学边测"，快速把主要风险点摸出来。
>
> 所以选择依据可以记三条：需求稳定不稳定、被测对象熟不熟悉、时间紧不紧。需求稳、对象熟、要长期回归，就脚本化；需求在变、对象生、要快速摸风险，就探索式。

> **板书：**基于脚本：设计 → 执行 → 分析（先写后跑；可复用、可回归） ｜ 探索式：学习 ↔ 设计 ↔ 执行 ↔ 分析（循环；边学边测） ｜ 选择依据：需求稳定度 · 对象熟悉度 · 时间压力 · 是否长期回归

> **提问：**一个刚上线的新 App，需求文档只有一页，团队里也没人测过，你手上没有任何历史用例。这时候你是先花两天写一份两百条的用例库，还是先动手点一小时？为什么？

> **预设回答：**

- 答"先动手点一小时"：✅ 多数情况下对。追问：那点之前要不要有目标？——要，探索式测试不是瞎点，得

带着"这次要摸清哪块风险"的任务去点，并且边点边记录。

- 答"先写两百条用例"：❌ 典型误区。点评：需求只有一页，你写出来的用例大概率是对着自己想象的规格在测，覆盖率看着高，真风险碰不到；而且两天时间对"刚上线"这种节奏太贵了。

- 答"两个都要，先探索再补脚本"：✅✅ 最接近实务。探索阶段摸到的问题，正好沉淀成后面可回归的用例。

> **易错点 / 考点：**最大的坑是把探索式测试等同于"随意乱点"（ad hoc 测试）。考试常见判断题："探索式测试就是不做测试设计、不用写用例的测试"——错。探索式测试的设计工作一直存在，只是它和执行交织在一起，并且要求测试者记录过程、复盘发现、把有效路径沉淀成可回归的用例。判断三步法：有没有带着目标去测？有没有边测边设计？有没有记录与沉淀？

> **过渡：**到这里，第 2.2 节"软件测试方式"的四组对比就全讲完了。接下来我们进入 2.3 节，讲一对更基础、更本质的划分：静态测试和动态测试。

【2.41】2.3 静态测试和动态测试（1 分钟）

> 从这一页开始，我们进入 2.3 节：静态测试和动态测试。
>
> 这一节要回答的核心问题只有一个：测试到底要不要把程序跑起来。不运行被测程序，靠人看文档、看代码，靠工具扫代码，这叫静态测试；把程序运行起来，喂数据、看输出、比结果，这叫动态测试。这一条分界线，是后面所有测试技术的一级分类，务必先立住。
>
> 再补一句它的分量：这组分类不是学术摆设，它直接决定你下一步能拿出什么手段——是坐在评审桌上挑问题，还是到环境里去跑用例。选错了手段，力气就白费了。
>
> 本节我们按三步走：第一步，讲产品评审，也就是对需求文档、设计文档和代码的评审——解决"该不该这么写"；第二步，讲静态分析，也就是不运行程序的代码检查——解决"代码里藏着什么问题"；第三步，讲验证与确认，也就是 V&V——解决"我们做的到底对不对、是不是用户真正要的"。
>
> 请大家带着一个问题听本节：静态测试不跑程序，它凭什么能发现缺陷？如果这个问题想通了，你就会明白教材为什么反复强调"越早发现越便宜"，也会明白测试为什么要往需求、设计这些"纸面产物"上延伸。顺便提醒：这一节的关键词——评审、静态分析、验证、确认——全是后面考试的高频词，听到一个就在脑子里归好类。

> **板书：**2.3 静态 vs 动态 ｜ 分界线：是否运行被测程序 ｜ 路线：产品评审（需求/设计/代码）→ 静态分析

（不跑程序查代码）→ 验证与确认（V&V）

提问：一个程序运行起来一切正常，是不是就说明它没问题？

预设回答：

- 答"是"：❌只能说明"跑到的那些路径"正常。追问：没跑到的路径、异常分支、界面文字错误、文档与代码不一致，动态测试能发现吗？

- 答"不是，还要看代码和文档"：✅对，这正是静态测试的用武之地。

- 答"得看需求"：✔️已经触及 V&V 的核心，本节最后一块内容就会展开。

易错点 / 考点：别把"静态"理解成"不动脑子地看"，也别把"动态"理解成"随便跑跑"。判断口径只有一条：被测程序有没有被执行。选择、判断题常把"代码评审""代码走查""静态扫描"和"运行程序调试"混在一起考，按这条口径一一对号即可。

过渡：先看第一块——产品评审。它管的是文档和代码的质量，也是所有测试活动里最能省钱的环节。

【2.42】静态测试（6 分钟）

什么是静态测试？教材给的定义很干净：静态测试包括对软件产品的需求和设计文档、代码的评审，比如技术评审、文档评审等，以及对代码的静态分析。一句话，凡是"不运行被测程序、靠人对产物进行检查"的活动，基本都算静态测试。

这里有一条特别容易被忽略的边界，请注意：管理评审、流程评审不属于静态测试，它们属于质量保证，也就是 QA。为什么？因为静态测试盯的是"产品"——需求、设计、代码这些交付物；而管理评审、流程评审盯的是"过程"——项目怎么管的、流程执行得规不规范。对象不一样，归属就不一样。这条边界考试很爱考。

再看评审的主要形式，教材列了三种：互为评审（Peer review）、走查（walk-through）、会议评审（Inspection）。我们可以按"正式程度"给它们排个队。互为评审最轻，就是同事之间互相看一遍，随时可以发生；走查是作者带着几个人把自己的文档或代码过一遍，边讲边收集意见，介于正式与非正式之间；会议评审最正式，有明确的角色分工——主持人、作者、记录员、检查人员，要按检查单逐条核对，发现问题要记录、要跟踪、要闭环。前面那张图里出现的那些角色，说的就是会议评审的阵势。

最后还有一句：代码的静态分析主要采用工具进行，但人工的代码评审也不可或缺。为什么人工不能省？因

为工具只认规则，不认业务。工具能告诉你"这个变量定义了没用"，但告诉不了你"这段业务逻辑把用户等级算反了"——后者只有懂需求的人看得出来。

生活类比：静态测试像交稿前的校对。你不是把文章念给别人听（那是动态测试），而是拿着红笔逐字看——错别字、句子不通、前后矛盾、引用不准。文章还没"运行"，错误就能挑出一大半。

板书：静态测试＝不运行程序，检查"产物" | 对象：需求文档 · 设计文档 · 代码 | 手段：评审（Peer review / Walk-through / Inspection）＋ 代码静态分析 | 边界：管理评审、流程评审 → QA，不属于静态测试 | 工具为主，人工不可省

提问：下面四项里，哪一项不属于静态测试？A 需求文档评审；B 代码走查；C 管理评审；D 用工具扫描代码。

预设回答：

- 答 C：✅ 正确。管理评审评的是管理过程和项目状态，属于质量保证（QA）。

- 答 D：❌ 常见错误，以为"用工具"就不算静态测试。点评：判断标准是有没有运行被测程序，跟用不用工具无关。

- 答 B：❌ 走查是评审的主要形式之一，是标准的静态测试。

易错点 / 考点：三个高频混淆点。①把管理评审、流程评审错当成静态测试——记住"评产品是测试，评过程是 QA"。②把"互为评审 / 走查 / 会议评审"的正式程度记反——口诀"互看最轻、走查居中、会议最重"。③以为静态分析全靠工具——教材明确说人工的代码评审不可或缺。考试可能出选择题判断归属，也可能在案例题里问"以下哪些活动属于静态测试"。

过渡：三种评审形式里，用得最多、收益也最大的，是产品评审。它评的到底是什么？下一页我们专门讲。

【2.43】2.3.1 产品评审（5 分钟）

2.3.1 讲产品评审。先明确评审对象：需求文档、设计和代码。也就是说，从"要做什么"到"打算怎么做"再到"真的怎么写的"，每一个阶段的产物都要被检查一遍。

教材有两句话要划重点。第一句：通过软件评审，可以更早地发现需求工程、软件设计等各个方面的问题，大大减少大量的后期返工，将质量成本从昂贵的后期返工转化为前期的缺陷发现。说白了，就是把"花钱改"提前成"花时间看"。第二句：评审是对软件元素或者项目状态的一种评估手段，以确定其是否与期望的结

果保持一致，并使其得到改进。请注意后半句"并使其得到改进"——评审的目的不是挑毛病，更不是追责，而是让产物变好。

为什么"更早"这么值钱？回到第 1 章讲过的缺陷修复成本曲线：同一个缺陷，在需求评审时发现，改的是一句话；在设计阶段发现，改的是一张图；到编码阶段发现，改的是代码加联调；等上线之后被用户发现，改的是补丁、是数据、是口碑，甚至可能是召回和赔偿。中间差的不是一点点工作量，而是数量级。

举个身边的例子。学生选课系统里有一句需求写着"系统应支持高并发选课"。评审时有人追问：高并发是多少？一千人同时点，还是两万人同时点？课程容量怎么算、超卖怎么处理？这些问题在评审桌上问出来，答案是改几行文字；如果等到选课当天系统卡死、课程被超选，再回头找原因，那就是一场事故。

所以测试人员参加产品评审，不是去"旁听"的，而是去提问、去挑二义、去预判可测性。你在评审会上问的每一个问题，都可能是后面省下来的一轮返工。

板书：2.3.1 产品评审 ｜ 对象：需求文档 → 设计 → 代码 ｜ 价值：后期返工 → 前期缺陷发现（质量成本前移） ｜ 定义：评估产物是否与期望一致，并使其得到改进 ｜ 测试人员角色：提问 · 挑二义 · 预判可测性

提问："把质量成本从后期返工转化为前期发现"，这句话落到你自己的项目里，具体是哪一笔钱、哪一段时间？

预设回答：

- 答"评审多花两天，省下上线后通宵修 bug"：✅ 方向对。追问：能不能举出你们小组真实的一次返工？

- 答"评审就是开会，浪费时间"：❌ 这正是要纠正的观念。点评：不评审不等于省时间，只是把成本推到了后面；越靠后越贵，还可能贵到修不起。

- 答"需求改一句话，代码要改一片"：✅✅ 好，说明理解了"缺陷扩散"——需求阶段的错会被设计、编码逐层放大。

易错点 / 考点：两个易错点。①记错评审对象：产品评审的对象是需求文档、设计和代码，不包括"人员绩效""项目进度"。②把评审等同于"找茬、考核"——教材强调评审是评估并改进产物。案例题里问"如何降低后期返工成本"，标准答案里一定有"尽早开展需求与设计评审"这一条。

过渡：产品评审里最先做、收益最大的是需求评审。需求评审判到底要解决哪些问题？下一页给了五类清单。

【2.44】需求评审解决的问题（4 分钟）

这一页列出需求评审要解决的问题，一共五类。我把它当作一张"找茬清单"来用，五个字：错、漏、糊、多、乱。

第一类，不正确的需求认识。需求方说的是 A，写进文档变成了 B，大家还都以为写的是 A。比如学生选课系统里"允许学生退课"，被写成了"允许管理员替学生退课"，一个主语的变化，权限就全错了。

第二类，丢掉的需求点。该有的漏了——需求里通篇讲选课，却没人提"退课之后名额什么时候释放""密码忘了怎么找回"。这类问题的可怕之处在于，漏掉的东西不会在评审桌上喊疼，它会在上线后用户投诉时才出现。

第三类，模糊的描述。像"界面友好""操作简便""响应速度快""尽量兼容主流浏览器"，这些词人人能念、人人解释不同，全是不可测的表述。模糊的需求等于把决策权偷偷交给了开发，最后验收时谁都不认账。

第四类，多余的或没意义的需求。需求文档里塞进了根本用不上、或者与目标无关的东西：校园系统只有两万师生，却要求"支持百万级并发"。多写的不只是文字，是设计复杂度、开发工作量和测试成本。

第五类，不一致的理解。同一句话，开发读出一个意思，测试读出另一个意思，客户心里还有第三个意思。评审桌上不说穿，等到测试提了缺陷，开发一句"我按需求做的"，就变成了扯皮。

你可以把需求评审理解成给图纸做"体检"：这五类问题，就是最常查出的五种病。查得越早，图纸越好改。

> **板书：**需求评审五类问题：① 不正确的需求认识 ② 丢掉的需求点 ③ 模糊的描述 ④ 多余的 / 没意义的需求 ⑤ 不一致的理解 ｜ 口诀：错 · 漏 · 糊 · 多 · 乱

提问："系统应具有良好的用户体验"——这句话属于这五类里的哪一类？为什么必须改掉才能继续往下做？

预设回答：

- 答"模糊的描述"：✅ 对。但它同时还可能引发"不一致的理解"——开发觉得是响应快，测试觉得是界面好看，客户想的是流程简单。

- 答"多余的需求"：❌ 不准确。它不是多余，而是本该说清楚却没说清楚，属于模糊。点评：区分的诀窍是问"这句话能不能验收"——不能验收的就是模糊。

- 答"改不了，用户就是这么说的"：⚠ 这正是评审要干预的地方。点评：把用户的原话翻译成可验收的指标，是需求评审的产出之一，比如"95% 的请求在 2 秒内返回"。

易错点 / 考点：五类问题容易混，尤其"模糊的描述"和"不一致的理解"——前者是单句话本身说不清，后者是

多个人的理解对不上，而一句话模糊往往顺带造成理解不一致。判断题常把"模糊"归到"多余"里去设陷阱。答题时按"错、漏、糊、多、乱"五个字逐条对照，就不容易漏项。

过渡：知道了要解决什么问题，还得知道按什么尺子来判断。下一页给出需求评审的六条标准。

【2.45】需求评审的标准（4 分钟）

需求评审不能凭感觉，得有一把尺子。教材给了六条标准：正确性、完备性、易理解性、一致性、易测试性、易追溯性。我给大家一个口诀：正确完备好理解，一致可测可追溯。

正确性，是说需求真的反映了用户的真实需要，没有写错、没有写偏。一条错误的需求，写得再漂亮也是负分。

完备性，是说该有的都有，功能、性能、约束、异常处理、边界情况，一个都不漏。判断完备性有个笨办法：拿业务流程从头走到尾，每一步都问一句"需求里写了吗"。

易理解性，是指描述清楚、没有二义，开发、测试、客户看同一句话，理解一致。凡是需要"再解释一下才明白"的需求，都不算过关。

一致性，是指需求之间不打架：文档内部不矛盾，跟上一版不矛盾，跟相关标准、跟已有系统也不矛盾。常见的坑是同一份文档里，前面写"学生最多选 6 门课"，后面写"最多选 8 门"。

易测试性，是指这条需求可以被验证、可以量化。这一条对在座的各位最直接——需求不可测，测试就没法设计用例，也没法判定通过与否。

易追溯性，是指每条需求都有唯一标识，能从需求追到设计、追到代码、追到测试用例，反过来也能从用例追回需求。有了追溯，才能回答"这条需求到底测了没有"和"这个用例是为哪条需求写的"。

这六条不是理论摆设。你们做课程设计的时候，拿这六条去审自己写的需求，能挑出一堆问题。

板书：需求评审六标准：正确性 · 完备性 · 易理解性 · 一致性 · 易测试性 · 易追溯性 ｜ 口诀：正确完备好理解，一致可测可追溯

提问：这六条里，哪一条最直接决定"测试能不能开工"？请说明理由。

预设回答：

- 答"易测试性"：✅ 方向正确。追问：如果一条需求不可测，你作为测试人员该怎么办？——不是放弃，而是推动把它改写成可量化、可判定的表述。

- 答"完备性"：✔️ 也很关键，缺了需求就没法设计覆盖；但完备性缺的是"内容"，易测试性缺的是"可验证性"，后者直接卡住用例设计。

- 答"一致性"：⚠ 一致性差会让用例互相冲突，但通常可以靠澄清解决，卡不死测试。

> **易错点 / 考点：**六条标准几乎每年都会以简答或选择的形式出现，务必按口诀背全，尤其别把"易测试性"漏掉——它最容易被学生忽略，却和工作最相关。另有一个高频概念题：需求"可测试"的标准表述是什么？答题要点是"可量化、有明确判定准则"。选择题里常把"可维护性""可移植性"混进来当干扰项——那是 ISO/IEC 25010 的质量特性，不是需求评审标准。

> **过渡：**标准有了，方法呢？下一页讲怎么把需求评审真正做好——八条注意事项。

【2.46】如何做好需求评审？（6 分钟）

> 这一页讲方法：怎么才能把需求评审做好。教材给了八条，我按"会前、会中、会后"帮大家归一下类。
>
> 会前两条。第一，有明确的评审标准，比如准备一份需求质量 checklist——就把上一页那六条当尺子。第二，熟悉评审内容，为评审做好准备。这一条最实在：参会前把需求文档读一遍，把看不懂的地方标出来，把想问的问题写下来。临时进会场才第一次读文档的人，只能跟着别人点头，评审会就变成了走过场。
>
> 会中四条。第三，该参加的人都需要参加。需求评审最怕两种缺席：写代码的没来，和真正用系统的人没来。第四，针对问题阐述观点，而非针对个人。可以说"这段描述会导致权限判断错误"，不要说"这是谁写的"。第五，从客户角度想问题，多问几个为什么。不要只问"这个功能怎么实现"，更要问"用户为什么要这个功能、不做行不行"。第六，在会前或会后提出自己建设性的意见。有些复杂意见当场说不清，写成书面意见反而更有效。
>
> 会后两条。第七，对发现的问题跟踪到底。记录下来的问题必须有人改、有人复核、有人关闭，不能散会即结束。第八，针对需求文档等报告问题——把问题落成书面记录，而不是停留在口头，否则过两天就没人认账。
>
> 最后还有一句总要求：检查需求定义是否合理、清楚，是否具有可测试性。这三问可以作为每次评审的自检清单。

> **板书：**会前：明确 checklist · 熟悉内容做准备 | 会中：该到的都到 · 对事不对人 · 站在客户角度多问为什么 · 提建设性意见 | 会后：跟踪到底 · 书面报告问题 | 自检三问：是否合理 · 是否清楚 · 是否可测

> **提问：**评审会上，你发现同事负责的那段需求有严重二义性，很可能引发返工。你打算怎么开口？

> **预设回答：**

- 答"直接说，这段写得有问题"：⚠ 事实没错，但把"问题"和"人"绑在了一起。点评：换成"这段描述可能有两种理解，我们确认一下是哪种"，效果完全不同。

- 答"不说，免得得罪人"：❌ 这是最坏的选择。点评：含糊的需求一定会变成后期的返工或线上缺陷，代价由整个团队和用户承担。

- 答"先私下沟通，再在会上正式提出来"：✅✅ 很成熟。既照顾了对方的感受，又保证问题被记录和跟踪。

> **易错点 / 考点：**八条里最常考的是"针对问题而非针对个人"和"对发现的问题跟踪到底"，案例题常设"评审会开成了批斗会"或"问题记录了没人管"的情境，让你分析并给出改进措施。另外要记住：评审不是一次性会议，而是一个"发现—记录—修改—复核—关闭"的闭环，缺了后半段，前面全白做。

> **【思政落点 1】**请记住评审席上的两条底线。第一条叫"对事不对人"：我们用证据和事实说话，指出的是文档里的缺陷，不是同事的能力。第二条叫"跟踪到底"：发现了问题却因为怕麻烦、怕担责而放过，等于把风险转嫁给用户和团队。将来你们做测试、报缺陷，一定会遇到"这个 bug 报上去领导不高兴"的时刻——请记住，如实记录、如实上报，是对用户负责，也是对自己的职业信誉负责。质量这条线，是靠一个个不肯将就的人守住的。

> **过渡：**需求评审判的是"做什么"。接下来，我们要评"打算怎么做"——设计评审。

【2.47】设计评审（5 分钟）

> 需求评审核对了"要做什么"，设计评审要核对"打算怎么做"。教材把设计评审的目标讲得很明确：保证需求能在设计中得到准确和完整的表示，也就是保证系统架构设计和产品功能规格说明书的质量。它评的核心产物，是设计规格说明书。
>
> 这一页的图给了四个抓手，我逐一拆开。

第一，系统架构。看分层是否清晰、模块之间耦合紧不紧、有没有单点、扩展和降级怎么考虑。架构评审是"高层"的评审，改起来最贵，所以最应该在纸面上就改掉。

第二，可测试性。这是测试人员最该盯的地方：接口有没有暴露、能不能 mock、关键路径有没有日志、内部状态可不可观测、能不能构造出需要的数据。可测试性差的设计，测试阶段只能靠"点点点"，覆盖率上不去，问题也定位不了。

第三，系统部署。部署拓扑、环境依赖、配置管理、升级与回滚方案，这些看着离测试很远，其实直接决定你能不能搭出一套可复现的测试环境。

第四，从高层到底层。设计评审不是一次过，而是层层往下：先审总体架构，再审模块设计、接口设计、数据库设计，最后审到产品功能规格说明书。借助 UML 等建模工具，用类图、时序图、状态图把设计"画出来"，画不清、看不懂的地方就是风险点。

这里面最需要各位记住的一句话是：不断从测试角度去问开发。设计评审不是开发的独角戏，测试人员问的每一个"这里怎么测""出错了怎么发现"，都是在为后面的测试省时间。

板书：设计评审 ｜ 目标：保证需求在设计中得到准确、完整的表示（系统架构设计＋功能规格说明书） ｜ 抓手：系统架构 · 可测试性 · 系统部署 ｜ 方式：从高层到底层 · 借 UML 等建模工具 · 不断从测试角度问开发

提问：作为测试人员参加设计评审，你最该问的三个问题是什么？请给出你的清单。

预设回答：

- 答"这个模块怎么测？出错了怎么发现？"：✅ 直击可测试性。可追问：如果没有日志、没有独立接口，你还能怎么测？

- 答"需求里这条功能在设计里有没有体现？"：✅ 这正是核对"需求的准确、完整表示"，属于设计评审的本职。

- 答"这段代码为什么这么写？"：❌ 越位了。点评：代码怎么写是编码阶段和代码评审的事；设计评审关注结构、接口、部署与可测试性，顺序错了会失焦。

易错点 / 考点：常见混淆是"设计评审"和"代码评审"的边界：设计评审看架构、接口、数据、部署与可测试性这些"纸面设计"，代码评审看具体实现。判断题还爱考"设计评审只在编码前做一次"——错，它随设计逐步细化、从高层到底层持续进行。记忆抓手："架构—可测—部署"三问，再加一句"从测试角度问开发"。

过渡：设计评审到底能拦下哪些问题？下一页给了一张很典型的清单。

【2.48】设计评审解决的问题（4 分钟）

这一页把设计评审常能拦下的问题列了出来，我逐条讲，你会发现每一条都对应后面测试阶段的一场噩梦。

第一，是否有设计规范。团队有没有统一的分层方式、命名约定、接口约定、异常处理约定。没有规范，代码就是"每人一套写法"，后期维护和测试都无从下手。

第二，系统架构设计的不合理。比如该分的层没分、耦合过紧、职责不清，一个改动要牵动半个系统。架构一旦定型，改造成本极高。

第三，单点失效。某个组件、某个服务、某个数据库一旦挂掉，整条业务链就断了，既没有备份也没有降级。这类问题在设计图上看得见，在代码里反而看不出来。

第四，数据的不完整性、不一致性。字段该有的没有、该唯一的没约束、同一份数据在多处存储口径不一，最后表现成"同一个学生在两个页面看到的成绩不一样"。

第五，缺乏可测试性。没有对外接口、不能单独运行、状态不可观测、测试数据造不出来。这一条直接决定测试的效率和可信度。

第六，具体功能设计的问题。比如流程走不通、异常分支没考虑、权限设计有漏洞——这些是"在设计层面就能看出来的功能缺陷"。

结论很清楚：这些问题在设计评审阶段被发现，代价是改图纸；留到系统测试阶段，代价就是改代码、改数据、改部署，甚至推翻重来。

> **板书：**设计评审常见问题：设计规范缺失 · 架构设计不合理 · 单点失效 · 数据不完整 / 不一致 · 缺乏可测试性 · 具体功能设计缺陷 ｜ 一句话：图纸上改便宜，代码里改昂贵

> **提问：**为什么"缺乏可测试性"必须在设计评审阶段解决，而不是等测试执行时再抱怨？

> **预设回答：**

- 答"测试阶段已经改不动了"：✅ 对。设计定型后再加接口、加日志、加 mock 点，等于返工重构，还要重新回归。

- 答"可测试性是测试人员的事"：❌ 需要纠正。点评：可测试性首先是设计属性，测试人员只能提前提要求、给建议，却没法替开发做设计。

- 答"只要功能对了，可测试性差点没关系"：❌危险。追问：一个没有日志、没有独立接口的模块线上出问题，你怎么定位？——功能对不代表可维护、可诊断。

> **易错点 / 考点：**"单点失效"和"数据不一致"是设计评审的经典考点，案例题里常给一段架构描述让你找问题，答题时要能说出"问题是什么、会造成什么后果、怎么改"。另外记住判断口径：凡是"改起来要动结构"的问题，都该在设计评审解决；凡是"改一行代码"的问题，可以留到代码评审。

> **过渡：**评审是"靠人看"，静态分析是"靠技术和工具看"。2.3.2 节，我们从这里开始。

【2.49】2.3.2 静态分析（2 分钟）

> 进入 2.3.2，静态分析。这一页只有两张图，我请大家把它当成一张"全景图"来看，看图的时候抓三件事。
>
> 第一件，静态分析管什么。它不运行程序，而是把需求和设计模型、源代码当成分析对象，从中提取信息，检查程序逻辑上的各种缺陷和可疑的程序构造。所以它的产出通常不是"通过 / 不通过"，而是一批"值得你去看一眼"的发现。
>
> 第二件，谁来分析。一头是人，一头是工具。人靠经验和理解，工具靠规则和算法，两者不是替代关系，而是配合关系——下一页我们就把这两种情况拆开讲。
>
> 第三件，它站在整个测试流程的什么位置。静态分析站在最前面，紧挨着编码。这时候程序还没跑起来，问题还在"纸面"上，改起来的代价最小。
>
> 请带着一个判断标准去看这两张图：凡是"不运行被测程序、直接对代码或模型做检查"的，都是静态分析。至于它具体分成哪两种情况、用了哪些技术、怎么从代码里一层层抽信息，接下来两页会逐层展开。

> **板书：**2.3.2 静态分析 ｜ 不运行程序，直接分析需求 / 设计模型 / 源代码 ｜ 两类执行者：人工（经验）＋工具（规则、算法） ｜ 位置：紧贴编码，越早越便宜

> **提问：**静态分析的产出，为什么通常不是"合格 / 不合格"，而是一批"可疑点"？

> **预设回答：**

- 答"因为工具会误报"：✅关键原因。静态分析从程序里提取信息做推断，推断不等于事实，所以结果需要人来判断。

- 答"因为它不运行程序，拿不到真实结果"：✅也对。没有运行结果，就只能提示"可疑"，不能判定"失败"。

- 答"因为它只检查格式"：❌太窄了。点评：静态分析不只查格式和风格，还能查程序逻辑缺陷与可疑构造，这一点下一页会展开。

> **易错点 / 考点：**别把静态分析窄化成"代码风格检查"。教材的定义包含两层：既看代码是否满足规范性、质量要求，也检查程序逻辑中的缺陷与可疑构造。选择、判断题常把"静态分析能发现运行时的性能瓶颈"设为错误选项——性能必须运行程序才能测，属于动态测试。

> **过渡：**静态分析具体分成哪两种情况？下一页给出答案：人工检测和计算机辅助静态分析。

【2.50】静态分析分为两种情况（4 分钟）

静态分析分两种情况，教材讲得很清楚：一种是人工检测，一种是计算机辅助静态分析。

先说人工检测。它偏重于编码风格、质量的检验，由人对设计、代码进行分析，能有效地发现逻辑设计和编码错误。注意"有效"这两个字——人工最擅长发现的是"需要理解意图才能判断"的问题，比如命名与语义不符、注释和代码打架、边界条件想漏了、异常处理写反了。这些问题工具几乎看不出来，因为它需要知道"这段代码本来想干什么"。

再说计算机辅助静态分析。它利用静态分析工具对被测程序进行特性分析，从程序中提取一些信息，以便检查程序逻辑的各种缺陷和可疑的程序构造。工具的优势是快、全、不知疲倦：几十万行代码几分钟扫完，规则一视同仁，还能出各种度量报告。但工具的短板也很明显：它不懂业务，只会按规则报警，容易误报，也可能漏报。

所以两者是配合关系，不是替代关系。现实中的用法是：先用工具做"初筛"，让它把可疑点全扫出来、按严重程度排好序；再由人做"复判"，逐条判断哪些是真问题、哪些是误报、哪些虽然报警但业务上无所谓。工具负责广度，人负责深度。

打个比方，这就像体检。仪器负责测指标——血压、血糖、心电图，又快又客观；医生负责看报告下结论——这个指标确实高，但结合你的具体情况要不要紧。只信仪器会吓人，只凭感觉会漏病。

> **板书：**静态分析两种情况 | ① 人工检测：偏重编码风格与质量，擅长发现逻辑设计和编码错误 | ② 计算机辅助静态分析：工具提取程序信息，检查逻辑缺陷与可疑构造 | 关系：工具初筛（广度）＋人工复判（深度）

> **提问：**你用一个静态分析工具扫了项目，报出 300 条告警。你打算怎么处理这 300 条？

预设回答：

- 答"逐条从头看一遍"：⚠ 态度对，但方法笨。点评：先按规则类型和严重级别归类、排序，优先看高危项和集中出现的规则，效率高得多。

- 答"先按严重程度分级，再逐类确认"：✅ 实务做法。追问：如果某一类规则误报率很高，你会怎么处理？——可以调规则或加白名单，但要有记录，不能直接关掉不看。

- 答"300 条太多，先不管"：❌ 最危险。点评：告警堆积正是静态分析工具被弃用的头号原因；正确做法是建基线——新代码不新增告警，存量逐步收敛。

易错点 / 考点：高频考点是把两种方式的能力边界搞混：人工检测偏重编码风格与质量，能发现逻辑设计和编码错误；工具负责特性分析、提取程序信息、发现可疑构造。还要记住工具不能替代人工。考试若问"静态分析能否完全自动化"，答案是否定的。

过渡：两种方式里，"计算机辅助静态分析"背后是一整套代码分析技术。下一页我们走进代码静态分析的流水线。

【2.51】代码静态分析（6 分钟）

这一页讲代码静态分析的原理。教材的定义是：通过词法分析（Lexer）、语法分析（Parser）、控制流分析、数据流分析等技术对程序代码进行扫描，验证代码是否满足规范性、质量要求等。

我们顺着流水线走一遍。第一步，词法分析。程序源代码经过词法分析器（Lexer），被切成一个个 Token，也就是各种不同种类的单词：关键字、标识符、常量、运算符、分隔符。这一步只认"词"，不认"句"。第二步，语法分析。Token 交给语法分析器（Parser），它做分析和语法检查，产出一棵抽象语法树（AST）——树形结构把"谁是谁的组成部分"表达清楚。走到这里，工具才算"读懂了句法"。

再往下是两座桥。一是控制流：通常用控制流图（CFG）来表示程序的控制流，节点是基本块，边是可能的跳转，分支和循环一眼就能看出来。二是数据流：通常用静态单赋值（SSA）来表示程序中数据的使用-定义链（Use-Def Chain），每个变量只被赋值一次，于是"这个值从哪儿来、到哪儿去"被追得清清楚楚。前面说的"变量定义了没用""用了没定义"，就是在这两座桥上被抓出来的。

教材还提到更高级的一类技术：静态分析可以采用模拟程序执行的技术，比如符号执行、抽象解释、值依赖分析，并采用数学约束求解工具进行路径约减或可达性分析，以减少误报、增加效率。通俗地说，符号执行

就是拿"符号值"代替具体输入，把程序当数学题去推；路径一多就会"路径爆炸"，这时候用约束求解做路径约减和可达性判断，把走不到的路径砍掉，效率才上得来。

回头看前面那两张图：左边是"程序怎么被读懂"，右边是"读懂之后能查出什么"。这条流水线的每一级，都对应着一类能查出来的缺陷。

> **板书：**程序 → Lexer（词法分析）→ Token → Parser（语法分析）→ AST ｜ 控制流：CFG ｜ 数据流：SSA（Use-Def Chain） ｜ 高级：符号执行 · 抽象解释 · 值依赖分析 + 约束求解（路径约减 / 可达性分析 → 减少误报）

提问：为什么"变量定义了没用""变量用了没定义"这类问题工具能轻松发现，而"这个循环少跑了一次"却很难？

预设回答：

- 答"前者是语法和数据流问题，后者是语义问题"：✅ 抓到要害。追问：数据流分析为什么能抓到"定义未使用"？——因为它建了 Use-Def 链，赋值和使用的关系是显式的。

- 答"循环跑几次要看运行结果"：✅ 对。循环次数往往依赖具体输入或复杂条件，静态推不出来，得靠符号执行，或者干脆交给动态测试。

- 答"工具不行，人工扫一遍就行"：⚠ 部分对。人工也能看出循环边界问题，但几十万行代码靠人扫不现实；正确姿势是工具负责可判定的模式，人负责需要理解意图的判断。

易错点 / 考点：这一页的专业词最容易记混：Lexer 做词法分析、产出 Token；Parser 做语法分析、产出 AST；CFG 管控制流，SSA 管数据流。答题时别把 CFG 和 SSA 的功能对调。另一个高频点：符号执行、抽象解释属于"模拟程序执行"的静态分析技术，注意它们仍然不真正运行被测程序，别误判成动态测试——这是选择题的经典陷阱。

过渡：静态测试的手段讲完了——评审和静态分析。但它们有一个共同的局限：只能验证"符合不符合规格"。那"规格本身对不对"由谁来管？下一页，验证与确认。

【2.52】2.3.3 验证与确认（V&V）（6 分钟）

这一页是 2.3 节的收口，也是整章最经典的一对概念。教材说：软件测试是由"验证（Verification）"和"有效性确认（Validation）"活动构成的整体。注意"整体"两个字——它是两件事合起来的，缺一不可。

先看验证，Verification，教材给的原问句是：Are we building the product right？我们是否正确地构造了软件？也就是"是否正确地做事"，验证开发过程是否遵守已定义好的内容，验证产品满足规格设计说明书的一致性。它的参照物是——规格设计说明书。

再看有效性确认，Validation，原问句是：Are we building the right product？我们是否构造了正是用户所需要的软件？也就是"是否正在做正确的事"，验证产品所实现的功能是否满足用户的需求。它的参照物是——用户需求。

一个是"正确地做事"，一个是"做正确的事"。口诀记牢：验证看文档，确认看用户——跟规格说明书比是 Verification，跟用户需求比是 Validation。

举个例子。团队按设计说明书严格实现了"成绩报表导出"：字段、格式全对，验证 100% 通过。可用户真正想要的是"成绩一出来就自动推送到手机"，报表根本没人下载。功能没错，方向错了——这就是典型的"验证通过、确认失败"。

再打个比方：盖房子时监理拿图纸核对钢筋型号、承重墙位置，这是验证；你搬进去住才发现厨房小得转不开身——图纸不合你的需要，这是确认出了问题。

完整的 V&V 告诉我们：既要问"有没有按规格做对"，也要问"这个规格是不是用户真正要的"。前者靠评审、静态分析和测试查一致性，后者靠需求评审、用户验收和真实使用查价值。

> **板书：**2.3.3 V&V ｜ Verification（验证）：Are we building the product right？正确地做事 → 对照"规格设计说明书"｜ Validation（有效性确认）：Are we building the right product？做正确的事 → 对照"用户需求"｜ 口诀：验证看文档，确认看用户

提问：一个项目的测试用例 100% 全部通过，能说明这个软件成功吗？为什么？

预设回答：

- 答"不能，用例只覆盖了规格，没覆盖用户真实需要"：✅✅ 标准答案。追问：那这些用例白跑了吗？——不白跑，验证通过说明实现与规格一致，这是必要但不充分的条件。

- 答"能，测都过了还不好？"：❌ 混淆了 Verification 和 Validation。点评：用例的设计依据通常是规格说明书，它证明的是"正确地做事"，证明不了"做正确的事"。

- 答"还要看用户验收"：✅ 对，验收测试正是 Validation 的主要手段，用户说好用才算数。

易错点 / 考点：这是全章最容易考、也最容易错的一对概念。①中译名易混：Verification 一般译"验证"，

Validation 译"确认（有效性确认）"；英文题干问哪个对应哪个，一律按参照物判断。②参照物易混：记住"验证看文档、确认看用户"。③常见陷阱选项："验证和确认是一回事"——错，两者的手段与视角不同，但目标统一，都是为了提高质量。判断三步法：先找参照物 → 再定 V 还是 V → 最后判断这句描述对不对。另外记住：评审是静态的 V&V，测试是动态的 V&V。

【思政落点 2】这一页其实藏着一个很硬的职业态度：不要满足于"文档上通过了"。规格说明书也是人写的，它也可能错、也可能过时。真正严谨的工程思维，是既不迷信文档，也不迷信自己的实现，而是始终追问一句"用户到底要什么"。将来你们做测试、写报告，如果发现"功能全部通过但没人愿意用"，请如实写出来，而不是把不利结论藏起来——诚实报出真相，比给出一个漂亮的通过率更重要。这就是科学精神在测试岗位上的样子。

过渡：到这里，2.3 节的三块内容——产品评审、静态分析、验证与确认——全部讲完，静态测试与动态测试这条主线也就闭合了。下一页，我们把视角从测试方法转到测试工作本身，看看软件测试到底分哪些层次、有哪些岗位分工。

【2.53】验证和确认（6 分钟）

这一页的标题只有四个字——验证和确认，配了一张图。别看字少，它是整个测试行业的两个基本坐标，我们后面讲的每一个测试活动，归根到底都在回答这两个问题里的一个。

>

先说验证。验证回答的是：我们是不是把产品做对了。说得完整一点，就是检查产品有没有按照事先定好的规格说明、设计文档、需求基线去实现。它的参照物是文档，是"我们当初说好要这么做"。审阅需求、审查设计、代码走查，单元测试里按语句和分支设计用例，这些都属于验证。

>

再说确认。确认回答的是：我们是不是做了对的产品。它的参照物不是文档，而是用户和真实需求——用户拿到这个东西，能不能解决他的问题。验收测试、用户试用、在线试运行，这些都属于确认。

>

为什么必须把这两件事分开？因为它们会各自出错。只做验证，可能把一份本来就写歪的规格说明百分之百地实现出来——程序一行没错，用户却用不了。只做确认，可能用户说"能用"，但内部实现全是临时补丁，

> 一改就崩。两个都做，才既有过程质量又有结果质量。

>

> 生活里的类比很好记：盖房子，验证是拿图纸对尺寸、对钢筋标号，一条一条核对；确认是搬进去住几天，看采光、看漏水、看一家人住得顺不顺。图纸全对，不代表住得舒服。

>

> 落到我们的学生选课系统：需求写"支持按学号查询选课结果"，验证就是检查代码有没有按设计实现这个查询；确认就是请教务老师真的用一遍，看她能不能顺手找到她要的那个学生的记录。

>

> **板书：**验证 Verification → 做对了吗（符合规格）→ 参照物＝文档／设计｜确认 Validation → 做对的产品吗（满足需求）→ 参照物＝用户／真实需求｜口诀：验证对内看文档，确认对外看用户

提问：我们的单元测试里，按设计文档检查一个函数有没有实现对非法输入的校验，这是验证还是确认？为什么？

> **预设回答：**①说是确认——因为校验的是用户数据；②说是验证——因为对照的是设计与规格；③说两个都算。教师点评：标准答案是验证，因为手里的参照物是文档。但如果"非法输入必须拦截"这条规则最初来自用户投诉，那这个活动其实已经带上了确认的成分——这说明验证与确认在实践中经常发生在同一个活动里。判断的关键就靠一句话：你手里的参照物是文档，还是用户？追问一句：那验收测试的参照物是什么？

易错点 / 考点：①把"验证"和"确认"当同义词，中英文都容易混，要注意 verification 与 validation 是两个词；②以为两者严格分阶段、先验证后确认——实际上在需求阶段两者就可以同时开始，这正是测试左移。判断三步法：一看参照物（文档／用户）→ 二看问题（做对了吗／做对的产品吗）→ 三看结论（符合规格／满足需求）。考试常出情景选择题："开发人员按详细设计评审代码"属验证；"邀请用户试用新版选课系统"属确认。

> **过渡：**有了验证和确认这两个坐标，我们再换一个维度——测试人员到底出不出手？下一页讲主动测试和被动测试。

【2.54】2.4 主动测试 vs. 被动测试（7 分钟）

> 这一页正式进入 2.4 节，按"谁在驱动被测对象"把测试分成两类。

>

> 课件给的定义很清楚。主动测试，是测试人员主动操作被测对象，比如输入数据、发送请求等来驱动被测对象，从而验证被测对象的响应或者输出结果。被动测试，是测试人员不干预产品的运行，而是被动地监控产品在实际环境中运行而获得系统运行的数据，然后进行分析。

>

> 主动测试我们最熟：点按钮、填表单、发接口请求、写自动化脚本跑上千条用例，全是主动测试。它的好处是能构造场景，能把边界值、非法值、极端组合都喂进去，而且有明确的预期结果可以比对。它的短板也在这儿——你只能测到你想得到的路径，真实用户那些"想不到"的操作序列，你构造不出来。

>

> 被动测试正好补上这一块。它不打扰系统，去监控线上真实的流量、日志、崩溃上报、响应时间，从海量真实数据里找异常模式。它最大的好处是数据真、场景全、对用户零打扰；最大的短板是没有标准答案——你看到一个响应时间是 1.8 秒，这算慢还是正常？没有需求基线、没有预期输出，只能靠趋势和对比来判断，也就是我们常说的"预言问题"。

>

> 打个比方：主动测试像体检，医生让你抬手、深呼吸，按项目逐条测；被动测试像给你戴一个 24 小时心率手环，你该干嘛干嘛，它默默记录。体检能查出专项指标，手环能发现你自己都没察觉的日常异常，两者谁也替代不了谁。

>

> 在工程里，主动测试主要用于研发和测试阶段，被动测试主要用于线上运行阶段，也就是我们下一页要讲的在线测试。考试常这样出：给一段活动描述，让你判断属于哪一类，判据就是"有没有人为注入输入"。

>

> **板书：**主动测试 → 测试人员注入输入／发请求 → 驱动对象 → 比对预期输出（可构造场景、有预言）｜被动测试 → 不干预 → 监控真实运行数据 → 事后分析（数据真实、无预言）｜判据一句话：有没有人为注入输入

> **提问：**我们上线后接了崩溃上报和 APM 监控，运维同事每天看错误率曲线。这属于主动测试还是被动测试？它能不能替代功能测试？

> **预设回答：**①"被动测试，不能替代"——正确；②"这是运维不是测试"——需要纠正：按教材口径，它正是被动测试／在线测试的一类，属于测试工作范畴；③"有了线上监控就不用写用例了"——典型错误。追问：如果没有一条功能用例说清"按学号查询应该返回什么"，监控发现接口报 500，你能不能判断它错在哪儿？借此讲清被动测试缺少预期结果这个先天短板。

> **易错点 / 考点：**①把"被动"理解成"不测"——被动测试依然是测试，只是不注入输入；②把"自动化测试"等同于主动测试，其实线上拨测也是自动化的；③误以为被动测试只能发现性能问题，它同样能发现崩溃、数据异常和业务流程断点。记忆口诀："主动动手，被动动眼"。判断题常见陷阱："被动测试不需要设计"——错，监控点、判定阈值、对比组同样是设计出来的。

> **过渡：**讲了被动测试，自然要问：产品在真实环境里跑、真实用户在用的那种测试，具体长什么样？下一页给了一个名字——在线测试。

【2.55】实例：在线测试（Product-in Testing）（6 分钟）

> 这一页只有一个标题加一张图，讲的是一个非常现代的测试形态：Product-in Testing，直译是"产品内测试"，我们课堂上就叫它在线测试。

>

> 它的核心思想一句话：把测试从实验室搬到真实的生产环境，让真实用户、真实数据、真实并发来当测试的输入源。传统做法是我们在测试环境里模拟用户，模拟得再像也只是模拟；在线测试是直接把新版本用灰度、用功能开关的方式放给一小部分真实用户，一边跑一边采集数据，用数据判断它到底行不行。

>

> 课件这张图，要大家看出三件事。第一，被测对象不再是代码分支，而是跑在真实环境里、有真实数据的产品。第二，测试数据不再是我们编的，而是真实用户产生的流量、点击、报错和反馈。第三，判断标准从"用例通过没通过"变成"线上指标有没有变坏"——错误率、响应时间、崩溃率、业务转化率。

>

> 为什么企业越来越依赖它？因为有些问题只在真实规模下才出现：真实的并发量、真实的数据脏乱程度、真

> 实机型的兼容性、真实的网络抖动。这些东西在测试环境里复现不出来，或者复现成本高得离谱。

>

> 但代价必须讲清楚：在线测试的对象是真实用户，一旦出问题就是真事故。所以它有一整套安全措施——先小比例灰度，观察再放量；关键功能挂功能开关，随时可以关掉；预设止损阈值，指标一超标立刻回滚；同时留对比组，才能区分"是版本变坏了"还是"今天流量本来就大"。

>

> 回到我们的课程项目：选课系统正式开放前，先让一个班试用，盯住接口错误率和响应时间，没问题再放开给全院。这就是最小规模的在线测试。

>

> **板书：**在线测试 = 生产环境 + 真实用户 + 真实数据 → 采集指标 → 决策（放量／回滚）｜三道安全阀：灰度放量 → 功能开关 → 止损回滚，另加对比组

> **提问：**我们把新版本放给 5% 的用户做在线测试，结果错误率比对照组高了。这时候第一件事是继续观察还是立刻回滚？依据是什么？

> **预设回答：**①"立刻回滚"——止损优先，正确；②"再观察一会儿看是不是偶然"——要区分场景：如果影响的是关键业务路径、影响面还在扩大，先回滚再复盘；如果只是轻微波动且有完善监控，可以缩短观察窗，但必须设死线；③"回滚了就说明这个功能不能上"——纠正：回滚只是止血，问题定位修好之后可以再灰度。

> **易错点 / 考点：**①把在线测试等同于"不做测试直接上线"——恰恰相反，它需要更完善的监控和更清晰的判定基线；②忽略对比组，把大盘波动误判成版本质量问题；③只讲收益不讲风险。考点多为案例判断："灰度发布加埋点监控"属于哪种测试？答：被动测试／在线测试。记忆口诀："先小后大、可关可回"。

> **过渡：**主动与被动，说的是在实验室还是在线上的差别。接下来换个维度：我们看不看得见程序的内部结构？这就是 2.5 节——黑盒与白盒。

【2.56】2.5 黑盒测试方法和白盒测试（7 分钟）

> 2.5 节，黑盒测试方法和白盒测试。这一页的关键词有两组，请大家对照着记。

>

> 第一组，黑盒对应的是"基于需求的测试"，也叫"数据驱动测试"。看图上那个示意：左边是客户需求，中间是输入，右边是输出，中间那个盒子是黑的不透明。黑盒测试就是不看程序内部怎么写的，只从需求和界面出发，准备输入数据，看输出对不对。它为什么叫数据驱动？因为驱动你测试的就是那一组组输入数据以及它们对应的期望输出——用例表就是它的核心资产。事件驱动也是同一个意思：你按一个按钮触发一个事件，然后观察系统响应。

>

> 第二组，白盒对应的是"结构化测试"，也叫"逻辑驱动测试"。这里盒子是透明的，你盯着代码里的语句、分支、条件、循环、路径去设计用例，追求的是"覆盖"——语句覆盖、判定覆盖、条件覆盖、路径覆盖。覆盖得够不够，要靠工具统计覆盖率来说话。

>

> 为什么这两种方法都必须有？因为它们能发现的问题不一样。黑盒站在用户角度，能发现"功能没实现、需求理解错了、界面不友好、业务流程走不通"；白盒站在实现角度，能发现"某个分支写错、边界判断漏了、异常没处理、代码里有死代码"。同一段代码，用两种视角去测，能找出完全不同的缺陷。

>

> 还有一个中间地带叫灰盒测试，既知道一点内部结构——比如知道接口协议、知道数据库表结构，又主要从外部去验证。它是现在接口测试和集成测试的主要形态。

>

> 企业里的粗略分工是这样：单元测试多用白盒，因为代码就在手边；系统测试、验收测试多用黑盒，因为测试人员关心的是业务；接口层常用灰盒。

>

> **板书：**黑盒 = 基于需求／数据驱动 → 输入—[黑盒]—输出｜白盒 = 结构化／逻辑驱动 → 语句・分支・条件・路径覆盖｜灰盒 = 知接口不知实现｜选择依据：信息可得性 + 测试阶段 + 测试目标

> **【思政落点 1】**同一段代码，用黑盒视角和白盒视角去测，能发现完全不同的缺陷——这正是科学精神里"多视角求证、不轻易下结论"的训练。要特别提醒学生：发现缺陷是团队的共同资产。发现后如实记录，不因为"这是我写的模块"就轻描淡写，也不因为"不是我负责的"就随手关掉，这是测试从业者最基本的诚信底线。

> **提问：**一个登录功能，输入框什么都不填直接点登录，页面白屏。用黑盒视角你能发现什么？用白盒视角又

能发现什么？

> **预设回答：**①"黑盒发现白屏这个现象，白盒发现空指针判断缺失"——很好，这正是标准答案；②"白盒测试是开发人员做的，测试人员不用管"——纠正：白盒是方法，谁做取决于阶段和技能，测试人员完全可以做接口级白盒；③"黑盒就是手工测试"——错，黑盒照样能自动化，关键看有没有利用内部结构信息。

> **易错点 / 考点：**①把黑盒／白盒当成"手工／自动化"的分类——两码事，分类维度是"是否利用程序内部结构信息"；②误以为白盒只测代码、黑盒只测界面；③考试爱考对应关系："数据驱动测试"对应黑盒，"逻辑驱动测试"对应白盒，选择题常把两者对调。判断三步法：一看有没有利用内部结构信息 → 二看用例依据是需求还是代码逻辑 → 三看覆盖度量是需求覆盖还是结构覆盖。

> **过渡：**到这里，2.4 和 2.5 两节的主要概念就讲完了。下一页是一张小结图，我们把它整理成两张对照表。

【2.57】小结（6 分钟）

> 这一页是小结，一张图，把 2.4 和 2.5 两节的知识点收在一起。我们不重复念标题，直接把它整理成两张对照表，方便大家抄下来复习。

>

> 第一张表，按"谁驱动被测对象"来分。主动测试，测试人员注入输入、发请求、驱动对象、验证响应；特点是有明确预期、能构造边界和异常，缺点是想得到的才测得到。被动测试，测试人员不干预，监控生产环境的运行数据再分析；特点是数据真实、场景完整、零打扰，缺点是没有标准答案、难判定对错。工程上，前者主要在研发测试阶段用，后者主要在线上运行阶段用。

>

> 第二张表，按"看不看得见内部结构"来分。黑盒，基于需求和数据，做输入输出比对，覆盖的是功能与业务路径；白盒，基于代码逻辑，追求语句、分支、条件、路径覆盖；中间还有灰盒，知道接口和数据结构，但主要从外部验证。

>

> 两张表放在一起，其实构成了四个象限。同一个测试活动可以同时落在两个维度上——比如我们写一段自动化接口脚本，发请求去验证返回，同时我们清楚这个接口会读哪张表：那它既是主动测试，又是灰盒测试。再比如线上的灰度监控，是被动测试，同时是黑盒测试，因为你不看代码只看指标。

>

> 给大家一个口诀收尾："主动被动看谁驱动，黑盒白盒看透不透。"再加一个判断方法：拿到任何一个测试活动，先问"输入是谁给的"，再问"我看得见里面吗"，两次追问，分类就清楚了。

>

> 这部分内容是后面所有具体方法的地基。等价类、边界值、判定表、因果图，属于黑盒方法族；语句覆盖、判定覆盖、基本路径测试，属于白盒方法族。今天先把族谱认清楚，后面才不会把方法用错地方。

>

> **板书：**维度一：谁驱动 → 主动（注入输入）／被动（监控数据）｜维度二：是否利用内部结构 → 黑盒（需求・数据驱动）／白盒（逻辑・结构驱动）／灰盒（知接口）｜四象限举例：自动化接口脚本 = 主动＋灰盒；线上灰度监控 = 被动＋黑盒｜口诀：主动被动看谁驱动，黑盒白盒看透不透

> **提问：**请把这个活动归到两个维度上：测试人员用工具向支付接口重复发送 1000 笔请求，观察返回码和金额，同时他知道这个接口背后会调用第三方通道。这是什么测试？

> **预设回答：**①"主动测试加灰盒测试"——标准答案；②"压力测试"——说的是测试类型，不是分类维度，要引导他分清"分类维度"和"测试类型"；③"黑盒测试"——可以追问：他知道会调用第三方通道这条信息，到底用上了没有？如果没用上，说黑盒也不算错。这正好说明黑盒与灰盒的边界，取决于你实际利用了哪些信息。

> **易错点 / 考点：**①混淆分类维度——"主动／被动"按驱动方式分，"黑盒／白盒"按信息可见性分，"功能／性能"按测试目标分，不能混在一层里答；②误以为被动测试不用设计，实际上监控点、阈值、对比组都是设计出来的；③小结页最容易出简答题："简述主动测试与被动测试的区别""黑盒测试与白盒测试的适用场景"。答题模板：定义 → 特点 → 适用阶段 → 局限。

> **过渡：**分清楚了"怎么测"，下面就要问"分几层测"。2.6 节，软件测试层次。

【2.58】2.6 软件测试层次（5 分钟）

> 这一页是 2.6 节的开篇，标题是软件测试层次，配了一张图。它是一页课时要点页，课件备注里其实写明了这节课的讲法：先讲概念，再详细讲解，然后举例，接着分享工作中的经验、特别是理论和实际有差异的地方，最后给学习建议和实用工具。我们就按这个路子走。

>

> 先讲概念。测试层次，指的是按被测对象的规模和组装程度，把测试分成若干级别。为什么非要分层？两个理由。

>

> 第一，缺陷发现得越晚，代价越大。一个在编码阶段改一行代码就能解决的错误，如果拖到系统测试阶段才暴露，可能要重新设计、重新联调，成本成倍往上翻；拖到上线以后，就是真实用户买单。分层就是为了让不同性质的问题，在各自最合适的阶段暴露出来。

>

> 第二，不同层次的问题性质本来就不一样。函数内部的逻辑错误、模块之间的接口不匹配、整个系统在真实业务下的表现、用户能不能接受，这是四类完全不同的风险，用同一种方式去测是测不出来的。

>

> 这里有一条工程上特别重要的经验，也是理论跟实际最容易分叉的地方：教科书上层次之间是清清楚楚的，实际项目里往往是混着走的。敏捷团队可能今天写一个函数、明天就联调、下周就灰度，四个层次被压缩在几天之内。但注意——压缩的是时间，不是内容。该问的问题一个都不能少：这段逻辑对吗？接口对得上吗？系统在业务场景下行吗？用户认吗？

>

> 所以这一页图，大家要看出的是一个"由内到外、由小到大"的递进：从最小的代码单元，到单元之间的接口，再到单元组成的系统，最后到系统承载的用户业务。下一页我们就把这四层具体展开。

>

> **板书：**测试层次 = 按被测对象规模与组装程度分级｜为什么分层：① 缺陷越晚发现代价越大 ② 不同层次问题性质不同｜讲义六步：概念 → 讲解 → 举例 → 经验 → 建议 → 工具｜关键提醒：时间可压缩，问题不能省

【思政落点 2】分层背后是一种"把大问题拆开、逐层把关"的工程思维，也是工匠精神里"每道工序都有责任人"的体现。可以引导学生反思：很多质量事故并不是没人测，而是四层里某一层被"赶进度"跳过了。守规矩、按流程做事，本身就是质量责任感的体现。

提问：项目工期紧，领导说"单元测试和集成测试先跳过，直接上系统测试，反正最后测通就行"。你同意吗？

怎么反驳？

> **预设回答：**①"不同意，越晚发现越贵"——方向对，但要补上具体理由：单元层没测干净的逻辑错误，在系统层会被业务场景掩盖，定位成本极高；②"跳过也能发现所有问题"——典型错误，因为系统测试用黑盒方法，覆盖不到内部结构，很多分支错误永远暴露不出来；③"可以压缩但不能取消"——这是比较成熟的回答，可以据此展开"时间可压缩、内容不可省"。

> **易错点 / 考点：**①把"测试层次"和"测试类型"混为一谈——层次是纵向分级，类型是横向分类；②误以为层次越高越重要——每层目标不同，缺一层就是漏洞；③考点多问"V 模型中每个层次对应哪个开发阶段"，记住对应关系：需求↔验收、设计↔系统与集成、编码↔单元。判断口诀："由小到大看对象，由内到外看信息"。

> **过渡：**好，四层具体是哪四层、每层测什么，下一页那张层次图说得很清楚。

【2.59】软件测试 4 个层次（7 分钟）

这一页是软件测试的四个层次，图上标得很齐，我们一层一层看。

>

> 第一层，单元测试。被测对象是组件、模块、类或者函数，也就是程序里最小的可测单位。目标是确认这个最小单元的功能实现了没有、有没有编码错误。执行主体以开发人员为主。因为信息最全，代码就在手边，所以主要用白盒方法。

>

> 第二层，集成测试。被测对象是单元之间的接口。为什么单独立一层？因为每个单元自己测通了，合在一起不一定能工作。参数顺序写反了、数据格式一个是字符串一个是数字、调用时序不对、异常没有正确传递，这些都属于接口问题，单元测试发现不了，系统测试又太粗、定位不到。

>

> 第三层，系统测试。被测对象是由单元构成的整个系统，关注的是系统功能、安全性、健壮性、效率这些整体质量属性，是在接近真实的环境里，对完整产品做端到端的验证。

>

> 第四层，验收测试。被测对象是系统承载的用户业务，站在用户和业务的角度，确认产品能不能被接受。它

> 问的不是"程序对不对"，而是"用户愿不愿意用、能不能用这个系统把活干完"。

>

> 把这四层串起来看，它是一条从内到外、从小到大的链：组件 → 单元之间的接口 → 由单元构成的系统 → 系统承载的用户业务。对应的信息可见性也在变化：单元层能看到代码，集成层看得到接口协议，系统层基本是黑盒，验收层完全站在用户一侧。

>

> 我特别提醒一个常见误解：不是"层次越高越高级"。这四层是互补关系，不是替代关系。你系统测试做得再漂亮，也测不出一个分支条件写反的逻辑缺陷，因为它藏在黑盒里；反过来，单元测试覆盖率做到百分之百，也不能保证两个模块拼起来就能跑通。

>

> 我们的课程设计就按这条链走：先给功能写单元测试，再集成联调，最后拼成完整的选课系统做系统测试和验收演示。

>

> **板书：**单元（组件／模块／类／函数）→ 集成（单元之间接口）→ 系统（由单元构成的系统）→ 验收（系统承载的用户业务）｜由内到外、由小到大｜互补关系，不是替代关系

提问：我们小组三个人各写一个模块，每个人自己的模块都测过、都能单独跑通。把三个模块拼起来，一登录就报"用户不存在"，登录模块和数据库模块都觉得自己没错。这个问题该在哪个层次发现？为什么单元测试没抓到？

> **预设回答：**①"集成测试，因为这是接口问题"——标准答案；②"在系统测试发现就行"——可以追问定位难度：系统层只能看到"登录失败"，要在三个模块里定位，得靠日志一点点排，成本高得多；③"单元测试没抓到说明单元测试没用"——纠正：单元测试测的是模块内部逻辑，接口契约本来就不属于它的范围，这正是分层存在的意义。

易错点 / 考点：①把集成测试理解为"把单元测试再跑一遍"——错，它的目标是接口；②把验收测试等同于系统测试——区别在视角和主体：系统测试由测试团队在模拟环境做，验收测试由用户在其真实业务场景中确认；③考试常考层次与被测对象的对应连线题，也考"某缺陷属于哪一层"的案例题。判断三步法：一看被测对象是什么（代码／接口／系统／业务）→ 二看谁在做（开发／测试／用户）→ 三看有没有利用内部结构信

息。

> **过渡：**四个层次分好了，那每一层到底要完成哪些任务？下一页用一张表把任务列得明明白白。

【2.60】不同测试层次的任务（7 分钟）

这一页是一张表，把四个测试层次各自要完成的任务列出来了，信息量很密，我们一行一行过。

>

第一行，单元测试有三项任务：组件的功能、健壮性、效率。功能好理解，就是这段代码该算对的算对了没有。健壮性指的是异常情况下的容错能力——输入为空、输入超长、除数为零、文件打不开，这时候程序是优雅地报错还是直接崩掉。效率则是这个最小单位上的性能，比如一个函数的时间复杂度和资源占用。

>

第二行，集成测试的任务只有一项：组件之间的接口。任务少不等于简单，恰恰说明这一层的目标非常聚焦——它不管单个组件的功能，专门盯接口契约：参数个数和类型对不对、数据格式一致不一致、调用顺序有没有依赖、异常能不能正确传递、共享数据会不会打架。

>

第三行，系统测试的任务有四项：系统功能、安全性、健壮性、效率。跟单元测试比，多了安全性；而且这里的功能、健壮性、效率都是系统级的——系统功能是端到端的业务流程能不能走通；安全性是权限、注入、越权访问、敏感数据保护；健壮性是高并发、异常流量、部分服务不可用时的表现；效率是响应时间、吞吐量、资源使用。

>

第四行，验收测试的任务是：功能及用户界面、安全性、效率、用户的可接受性。注意最后一项——用户的可接受性，这是验收测试独有的，最主观也最终极：功能都能用，但界面乱、流程绕、术语看不懂，用户就是不接受，那这个系统在验收这一环就是不通过。

>

竖着看还有一条规律：越往上，测的维度越多、越贴近整体和用户；而"接口"只在集成这一层出现，它是这一层专属的靶心。

>

> 记忆方式：别死背四行，记三个关键词——组件三项、接口一项、系统四项、验收四项加可接受性。考试里最常见的题型，是给你一个缺陷现象，问你它属于哪一层的哪项任务。

>

> **板书：**单元 → 组件功能・健壮性・效率｜集成 → 组件之间的接口｜系统 → 系统功能・安全性・健壮性・效率｜验收 → 功能及用户界面・安全性・效率・用户可接受性｜竖看规律：越往上维度越多、越贴近用户

> **提问：**三个现象，请分别归到层次和任务上。①某函数传入空字符串时抛未捕获异常，导致进程退出；②订单模块传给库存模块的数量是字符串"3"，库存模块按数字处理，扣减结果错误；③选课系统在 2000 人同时抢课时，响应时间从 0.5 秒涨到 30 秒。

> **预设回答：**①单元测试的健壮性——注意不是功能，因为它没算错，是异常处理缺失；②集成测试的接口——典型的数据格式不一致；③系统测试的效率，也可以说性能测试。如果学生答成"都是 bug"，要追问他：这三个问题分别在哪个阶段最容易发现、由谁负责修，让他体会分层的实际意义。如果学生把③答成"验收测试的效率"，可以让他比较两种答法的差别：谁在测、在什么环境测。

> **易错点 / 考点：**①把"健壮性"和"功能"混起来——判据是：有没有产生错误结果。产生错误结果是功能问题，异常场景下扛不住是健壮性问题；②忘记"用户的可接受性"只在验收层，看到界面问题就答系统测试；③安全性出现在系统和验收两层，容易漏答一层。记忆口诀："组件三、接口一、系统四、验收四加一"。案例题答题结构：先定层次，再定任务项，最后给一句话理由。

> **过渡：**表格里四层都有了，接下来我们把最底层的单元测试单独拎出来细讲，它的做法和分工很有讲究。

【2.61】单元测试（8 分钟）

> 这一页专门讲单元测试，课件上这段话几乎是定义级的标准表述，我们逐句拆开看。

>

> 第一句：单元测试针对程序系统中的最小单元——模块或组件进行测试，一般和编码同步进行。有两个关键词。"最小单元"，在实际的面向对象开发里，通常就是一个类或者一个方法；"和编码同步进行"，意思是不要等整个模块写完再补测试，而是写一个函数就测一个函数，边写边测，问题在几分钟内就被发现并修掉。

>

> 第二句：主要采用白盒测试方法，从程序的内部结构出发设计测试用例，检查程序模块或组件的已实现功能与定义的功能是否一致，以及编码中是否存在错误。请注意，它同时验两件事：一是功能对不对，二是代码本身有没有写错，比如分支条件写反、变量没初始化、循环边界差一。

>

> 第三句，这条最容易被忽略：通常要编写驱动模块和桩模块。为什么？因为你要测的函数往往不是独立的——它要调用别人，也要被别人调用。桩模块用来替代"它调用的下层模块"，比如你的函数要查数据库，测试时就用一个假函数直接返回固定数据；驱动模块用来"调用它"，负责传参数、接收返回值、打印结果。有了桩和驱动，一个函数就能脱离整个系统单独跑起来，这才叫单元测试。

>

> 第四句：单元测试一般由编程人员和测试人员共同完成，以开发人员为主。原因是他们最了解自己的代码，而且刚写完就测，修复成本最低。

>

> 最后一句是个很硬的数据：单元测试包括代码评审，代码评审可以发现程序 50% 到 70% 代码的缺陷。这个比例来自教材，说明很多缺陷根本不用运行程序，人眼看代码就能发现，这是性价比最高的一道防线。

>

> 落到课程里：你写一个"根据选课学分判断能否毕业"的函数，用桩挡掉数据库，用驱动喂几组数据再看返回结果，这就是一次完整的单元测试。

>

> **板书：**对象：最小单元（模块／组件）｜时机：与编码同步｜方法：白盒为主｜目标：已实现功能 = 定义功能 + 无编码错误｜工具件：桩模块（替代被调用的下层）+ 驱动模块（调用本单元）｜分工：开发为主、测试参与｜补充：代码评审可发现 50%～70% 的缺陷（教材数据）

【思政落点 3】代码评审能发现 50%～70% 缺陷这个数据，正好用来说明"同行评审不是挑刺，而是互相保命"。要讲清两点职业素养：评审别人的代码时对事不对人，指出问题要给依据；被别人评审时坦然接受、如实修改。反过来，明知有问题却因为怕丢面子而不说，最后承担后果的是用户——这就是我们这门课反复强调的质量诚信。

> **提问：**我要测一个"计算订单总价"的方法，它内部会去数据库查商品单价，还会调用优惠券服务算折扣。为了做单元测试，我需要写哪两个东西？它们分别替代谁？

> **预设回答：**①"桩模块替代数据库和优惠券服务，驱动模块负责调用计算总价的方法"——标准答案；②"直接把数据库连上测就行了"——要追问：数据库里有脏数据、优惠券服务今天正好挂了，你的测试结果还稳定吗？借此讲"单元测试要隔离外部依赖"；③把桩和驱动说反了——这是最常见的错误，纠正口诀："桩在下被调用，驱动在上调用它"。

> **易错点 / 考点：**①桩模块与驱动模块混淆——判断方法是看它在调用链上的位置：被被测单元调用的是桩，调用被测单元的是驱动；②误以为单元测试必须写代码，其实人工代码评审也是单元测试的一部分；③误以为单元测试是测试人员的事——教材明确以开发人员为主。考点形式：名词解释、简答"单元测试为什么需要桩和驱动"、判断"单元测试一定要在全系统集成之后做"（错）。口诀："同步做、白盒为主、桩下驱动上、开发挑大梁"。

> **过渡：**光说概念还是抽象，下一页课件给了示例图，我们看着图把驱动和桩怎么搭，具体走一遍。

【2.62】示例（6 分钟）

> 这一页是示例，课件放了两张图。我不照着图念，而是带大家把单元测试的完整套路走一遍。你们看图的时候，就对着这三步走：搭架子、设计用例、看结果。

>

> 第一步，搭架子。看图上的调用关系：最外面是驱动模块，它负责准备输入数据、调用被测函数、接收返回值。被测函数内部如果要访问数据库、要读文件、要调用别的服务，就在那个位置换成桩模块。你要能一眼在图上指出：哪个框是被测单元，哪个框是被调用方替换出来的假货，哪个框是调用方。这是读这类图最核心的能力。

>

> 第二步，设计用例。单元测试主要用白盒方法，所以用例不是随便挑几个数字，而是盯着代码结构去设计：每条语句至少走到一次；每个判断的真假两个方向都要走到；多个条件组合的时候要覆盖关键组合；循环要测零次、一次、多次。用这些用例去覆盖，才能回答"我到底测到了多少"这个问题。

>

> 第三步，看结果、看覆盖。跑完以后不只看通过没通过，还要看覆盖率报告：哪些分支没被走到。没被走到的分支只有两种可能——要么是测试用例不够，要么是这段代码根本多余、或者永远进不去。两种情况都值得处理。

>

> 两张图放在一起，通常是一张讲被测单元与外部依赖的替换关系，一张讲用例的输入与预期输出。你们读图的时候注意一个细节：图里有"预期输出"这一列，这说明单元测试的判定标准是客观的——输入、预期输出、实际输出三者比对，不靠人主观判断，这也是单元测试能够自动化的前提。

>

> 最后再提醒一条工程上的经验：单元测试的用例应该和被测代码放在一起，跟着版本走，每次提交都跑。写一次只跑一次，那不叫单元测试，那只是一次性的手工验证。

>

> **板书：**读图三步：① 找被测单元 ② 找被替换的依赖（桩）与调用方（驱动）③ 找输入／预期／实际三列｜白盒用例设计：语句 → 分支真假 → 条件组合 → 循环 0／1／多次｜结果判定：通过与覆盖双看｜落地要求：用例随代码入库、每次提交都跑

提问：覆盖率报告显示某个 if 分支从来没被执行过。请你给出两种可能的原因，以及分别该怎么处理。

> **预设回答：**①"用例没覆盖到"——处理是补一条能走进去的输入；②"这段代码进不去，是死代码或者防御性代码"——处理是确认后删除，或者加注释说明；③"覆盖率不重要，能跑通就行"——典型误解，要纠正：覆盖率不是目的，但它能暴露"我根本没测过这里"这个事实，是发现测试盲区的手段。可以追问：覆盖率 100% 是不是就没有缺陷了？引导学生自己说出"覆盖的是结构，不是需求"。

易错点 / 考点：①把覆盖率当成质量结论——覆盖率 100% 只说明结构走到了，不代表需求全实现、边界全对；②设计用例时只按代码结构，忽略规格说明里的边界——白盒也要结合需求；③把示例图里的"预期输出"当摆设，实际工作中最常见的错误就是只看程序跑没跑通、不比对预期值。考点："单元测试用例设计的主要依据是什么"——答：程序内部结构（白盒），并结合模块的功能说明。

> **过渡：**单元自测通过以后，下一步就是把这些单元装到一起——这就是集成测试。

【2.63】集成测试（7 分钟）

这一页讲集成测试。课件给的定义是：集成测试，也称联调，在单元测试的基础上，将模块按照设计要求组装起来同时进行测试，主要目标是发现与接口有关的模块之间的问题。现在提倡持续集成测试。

>

把这个定义里的三个要点抠出来讲。

第一，前提是单元测试已经做过。集成测试不是拿一堆没测过的模块来试，而是在单个模块已可用的基础上，专门看"组装"这件事。顺序反了，出问题就分不清是模块内部错还是接口错。

>

第二，被测对象是接口，由此决定了集成测试用例长什么样：不深挖模块内部逻辑，而是构造跨模块的数据流和调用序列，检查参数传递、数据格式、字段含义、调用时序、错误码约定、异常传播、全局数据和资源竞争。举几个场景：A 模块传时间戳用秒，B 模块按毫秒解析，差了 1000 倍；订单和库存模块的调用顺序写反，出现超卖；一个模块抛的异常另一个模块接不住，流程静默中断。

>

第三，为什么叫"联调"。工程里，集成测试往事故意是开发人员坐在一起，把各自模块接上调通——它天然带一点协同色彩，需要接口契约先谈清楚。

>

集成策略顺便记一下：一次性组装，也就是"大爆炸"，风险最大；自顶向下，从主控模块往下接，需要桩；自底向上，从底层往上拼，需要驱动；三明治，两头往中间合。这四种策略的名字和优缺点，是考试常客。

>

"现在提倡持续集成测试"这句话很关键：以前集成是阶段性的动作，等所有模块写完才拼；现在每完成一个小功能就自动构建、自动跑集成用例，让接口问题在几小时内暴露。

>

第一章的火星探测器事故就是接口问题的极端版本：两个团队各自算得都没错，但一个用公制、一个用英制，接口没对齐，最终整机坠毁。它讲的不是"谁写错了代码"，而是"接口没约定清楚，谁都没错，系统却错了"。

>

> **板书：**集成测试 = 联调｜前提：单元测试完成｜目标：与接口有关的模块之间的问题｜检查清单：参数／格式／含义／时序／错误码／异常传播／全局数据／资源｜策略：大爆炸 · 自顶向下（需桩） · 自底向上（需驱动） · 三明治｜趋势：持续集成测试

> **提问：**单元测试全部通过，集成的时候发现下单流程偶发失败，日志显示库存扣减成功了但订单没生成。这最可能是哪类问题？集成测试该怎么设计用例才能抓到它？

> **预设回答：**①"接口时序或者事务问题"——对，可能是下单与扣库存的顺序问题，也可能是事务边界没包住两个操作，还可能是异步调用的超时与重试；②"再跑一遍单元测试"——要指出单元测试测不出跨模块的事务边界，跑一百遍也没用；③"这是偶发问题，先不管"——典型的危险回答，要追问：偶发意味着与条件有关，通常和并发、时序相关，集成测试恰恰要用并发和异常注入的用例去复现它。

> **易错点 / 考点：**①把集成测试与系统测试混同——区别在范围与目标：集成测接口、范围是模块组合；系统测整体、范围是完整产品；②以为集成测试只能等所有模块写完才能做，这与持续集成理念相悖；③策略名称与所需辅助模块混淆，记住口诀："自顶向下要桩，自底向上要驱动"。考点常见：简答"集成测试的主要目标"、案例题"该缺陷属于集成还是系统"。

> **过渡：**定义里最后那半句"现在提倡持续集成测试"，值得单独用一页讲清楚，因为它是现代软件交付的基本功。

【2.64】持续集成、持续测试（7 分钟）

> 这一页讲持续集成和持续测试，配了一张流程图。我们先把这个流程顺着图讲一遍：开发人员把代码提交到版本库，集成服务器检测到变更，自动拉取代码、自动构建、自动部署到测试环境、自动跑测试，然后把结果反馈给所有人。整个过程不需要人手工点，几十秒到几分钟就出结果。

>

> 为什么要这么做？先说不这么做会怎样。传统模式下每个人在自己机器上写代码，各自都认为自己是对的，等到集成日那天一起合并，冲突、编译不过、接口对不上，一下子全炸出来，业内管这叫"集成地狱"。最要命的是问题堆在一起，你分不清是谁引起的，排错时间比写代码时间还长。

>

> 持续集成的核心其实不是工具，而是习惯：小步、频繁地提交，每次提交都保证可构建，主干始终可用。这

> 样每次集成的差异都很小，一出问题，一眼就能定位到刚提交的那几行。

>

> 持续测试是持续集成的升级：把测试嵌进这条流水线。提交代码后自动跑单元测试；构建成功后自动跑集成测试和接口测试；部署到测试环境后跑冒烟测试；再往前是自动化回归和部分性能测试。它的价值在于反馈速度——缺陷在提交后几分钟就被发现，而不是等到测试阶段才被发现。这正好呼应我们前面讲的"越晚发现越贵"。

>

> 这里必须讲两个代价，不然就是只讲好处了。第一，自动化用例本身需要维护，接口一变用例就红，用例资产和产品代码一样需要人管。第二，会出现"不稳定用例"，一会儿过一会儿不过，大家慢慢就不看红灯了——一旦团队开始忽略红灯，这条流水线就形同虚设。所以工程上有一条铁律：流水线红了先修，绝不带着红灯往主干部署。

>

> 落到我们课程设计：你们小组可以把代码放到 Git 仓库，约定每次提交前必须自己先跑通测试，提交后互相检查。这就是最小规模的持续集成。

>

> **板书：**提交 → 自动拉取 → 自动构建 → 自动部署 → 自动测试（单元／集成／接口／冒烟／回归）→ 反馈｜核心习惯：小步提交、主干常绿、失败即停｜两大代价：用例维护成本、不稳定用例｜铁律：红灯不放过

【思政落点 4】持续集成这条流水线最考验的其实是诚信和责任感——测试红了却直接合入、失败用例被注释掉、报喜不报忧地宣布"构建通过"，这些做法短期看像是"效率"，长期看是给用户埋雷。要引导学生建立"红灯就是命令，谁打破谁负责修"的职业习惯，这正是质量文化落地的具体表现。

提问：小组里有个同学说："只要我本地测试通过，就可以提交了。"这句话有没有问题？

> **预设回答：**①"有问题，本地环境和集成环境不一样"——对，还要补一句：持续集成要求提交后整条流水线是绿的，个人通过不等于整体通过；②"有问题，应该先拉最新代码再测"——这是很成熟的回答，可以顺势讲代码同步与冲突；③"没问题啊"——典型错误，追问：如果你的改动让别人的模块挂了，而你没跑别人的测试，这个问题该由谁来发现？引出"主干常绿是团队共同责任"。

易错点 / 考点：①把持续集成等同于"装个构建工具跑一下"——它首先是开发习惯和纪律；②把持续测试等同

于"测试自动化"——自动化是用例的执行方式，持续测试强调的是嵌入流程、每次变更都跑、快速反馈；③忽略不稳定用例的危害。考点多为简答："持续集成的核心实践有哪些""为什么说持续测试能降低质量成本"。

过渡：讲完了理念，我们看一个真实项目的例子——下一页的示例。

【2.65】示例（6 分钟）

这一页又是示例，配了一张图，课件备注给了一个具体的项目：DotNetNuke，一套非常优秀的基于 ASP.NET 的开源门户网站程序，网址是 www.dotnetnuke.com，我们可以课后点开去看。

>

这个示例为什么选一个开源项目？因为开源项目的质量过程是透明的，你能看到很多平时在公司里看不到的东西。你去观察这样一类项目，我建议盯四个地方。

>

第一，看它的代码仓库怎么组织。有没有专门的测试工程目录，测试代码和产品代码是不是放在一起、一起提交。如果一个开源项目的测试代码和产品代码同步演进，说明它把测试当资产；如果测试目录很久没更新，那它的质量基本只能靠用户报 bug。

>

第二，看它的持续集成状态。这类项目通常会在仓库首页挂一个构建状态的标识，每次提交都自动构建、自动跑测试。绿色代表主干可用，红色代表有人刚把主干弄坏了——这就是我们上一页讲的"主干常绿"。

>

第三，看它的缺陷跟踪。提一个 bug 要求写清楚：复现步骤、期望结果、实际结果、环境版本。修完之后要求补一条回归用例，防止同一个问题再次出现。这个"缺陷—用例"的闭环，是质量持续改善的关键机制。

>

第四，看它的代码提交评审。合并请求要有别人审过、要有测试跟着进。这就是我们前面讲的"代码评审能发现 50% 到 70% 缺陷"，在工程里的落地方式。

>

这四点合起来，其实就是开源社区用流程和自动化，把一个没有统一管理者的大型项目维持住的秘诀。大家

做课程设计的时候完全可以照这个套路来：建仓库、写测试、把测试和代码一起提交、用缺陷清单跟踪问题、让组内同学互相评审。你们这个学期要交付的不只是一个能跑的系统，还有一套能说明"你是怎么保证它是对的"的过程材料。

>

板书：示例：DotNetNuke（基于 ASP.NET 的开源门户网站，www.dotnetnuke.com）｜观察四点：① 测试代码是否随产品代码一起提交 ② CI 构建状态是否常绿 ③ 缺陷跟踪是否闭环（复现 → 修复 → 回归用例）④ 合并是否经过评审并带测试｜迁移到课程设计：仓库 + 测试 + 缺陷清单 + 组内互评

提问：你到开源项目仓库里一看，发现它的测试目录最近三个月一次都没改过，而产品代码这三个月提交了两百多次。这说明什么？

预设回答：①"测试没跟上，质量只能靠用户报 bug"——标准答案；②"可能这个项目不需要测试"——要纠正：功能在持续变化，没有测试跟着变，等于回归验证缺位，老功能被改坏的几率很高；③"提交次数多说明项目活跃、质量好"——引导他区分"活跃度"和"质量保障强度"这两个不同的指标。

易错点 / 考点：①把"示例"当故事听，不提炼可迁移的做法——考试和答辩里，能不能把案例讲成"做法 + 效果 + 启示"才是得分点；②误以为持续集成、代码评审是大公司的奢侈品——开源项目用免费工具就能做；③混淆"缺陷多"和"质量差"：缺陷发现得多、跟踪得规范，反而是质量过程健康的信号，真正危险的是没人报缺陷。考点：案例题常要求"结合开源项目的做法，说明如何在小组项目中落地持续集成与测试"。

过渡：到这里，第 2 章关于测试分类、测试层次与黑盒白盒方法的主要内容就讲完了。下一片我们继续往下走，进入测试过程与更多具体方法的内容。

【2.66】系统功能测试（6 分钟）

前面几次课，我们一路走过了单元测试、集成测试，现在把镜头拉远，站到用户那一边。系统测试是软件测试四个层次里的第三层，它上面承着集成测试，下面接着验收测试。这一页讲的是系统测试里的第一块内容——系统功能测试。

>

先给定义：系统功能测试，一般是在完成集成测试之后进行的，它依据的是产品功能说明书，针对产品所实现的功能，从用户的角度来做功能验证，确认每一个功能是不是都能正常使用。

>

> 这段话里有三个词要抠住。第一个是"集成测试之后"——顺序不能乱。第二个是"产品功能说明书"——测试依据是它，不是代码，也不是开发同事口头说的那句"这块没问题"。第三个最关键，是"从用户角度"。同一个功能，开发人员想的是"我是怎么实现的"，用户想的是"我要拿它办成什么事"。

>

> 为什么一定要强调用户角度？因为集成测试查的是模块之间的接口通不通，它关心的是"零件装得上装不上"；系统功能测试关心的是"这台机器能不能干活"。举个生活里的例子：一辆车在总装线上，每个零件都装好了、螺丝都拧紧了，这叫集成没问题；可这车到底能不能开、空调冷不冷、刹车灵不灵，得把它开到路上去跑一跑——这就是系统功能测试。

>

> 再看软件行业的实例。一个电商网站的购物车模块，集成测试关心的是购物车和库存、订单模块之间的接口参数对不对；系统功能测试关心的是一个真实用户能不能把商品放进购物车、改数量、用优惠券、下单、再取消订单，每一步的结果是不是符合功能说明书的描述。

>

> 还有一点必须讲清楚：这一层用的是黑盒方法。你不去管内部怎么实现，只通过界面和接口去驱动它，看输出对不对。这也意味着，功能说明书要是写得不清不楚，这一层测试就注定是糊的——所以我们在测试需求分析里反复强调"可测试性"。
>
> **板书：**系统功能测试 → 位置：完成集成测试之后（系统测试层）→ 依据：产品功能说明书 → 视角：用户角度 → 目标：每个功能都能正常使用 → 方法：黑盒。

提问：购物车"删除商品"功能，集成测试已经通过，系统功能测试还要测哪些点？

> **预设回答：**第一种回答："点删除，商品没了，就算通过。"这是典型错误——只测正常路径，还默认"接口通了功能就没问题"。第二种回答："还要看数量、总价有没有刷新，优惠券要不要重算，库存有没有释放，删除后能不能恢复。"这是正确方向。第三种回答："这些是单元测试该管的。"说明层次没分清。教师点评：先肯定第二种，再带大家把"接口通"和"功能对"这两个概念分开；然后追问库存、优惠券的联动为什么要放到系统层来看——因为它跨了多个模块，只有在系统这一层才观察得到。

易错点 / 考点：最容易和集成测试混淆。判断三步法：一看依据——功能说明书还是设计/接口约定；二看视角——用户视角还是模块之间；三看对象——每个功能可用与否还是接口问题。考试常以选择题问"系统功能测试在哪个阶段之后进行"，或给一个场景让你判断属于哪一测试层次。口诀：功能说明为依据，用户视角验

功能，集成之后才登场。

> **过渡：**功能都正常了，用户就满意了吗？如果点一下要等十秒、一千人同时用就崩了，用户照样跑掉。下一页我们看系统测试的另一半——系统非功能性测试。

【2.67】系统非功能性测试（6 分钟）

上一页我们说功能测试回答的是"能不能用"，这一页要回答的是"好不好用、扛不扛得住"。

>

> 先看定义：系统非功能性测试，是把软件放在整个计算机环境下——注意这个"整个计算机环境"包括软硬件平台、某些支持软件和数据等——在实际运行环境下验证系统的非功能性。

>

> 这句话里有两个地方要圈出来。第一个是"实际运行环境"。功能测试可以在你的开发机上跑，非功能测试不行。为什么？因为性能、稳定性这类特性，跟机器的 CPU、内存、磁盘、网络、数据库版本，甚至同一台机器上还跑了什么别的东西都有关系。你在开发机上测出来响应五十毫秒，用户那台装了三个杀毒软件的老电脑上可能是五秒。

>

> 第二个是"软硬件平台、支持软件和数据"。这几样凑在一起才叫环境，少一样，测出来的结论就不算数。

>

> 那它包含哪些内容？课件列了四项，后面还跟着省略号：性能测试、安全性测试、稳定性测试、兼容性测试。省略号意味着不止这些，还有易用性、可靠性、可维护性等等，也就是我们前面讲 ISO/IEC 25010 产品质量模型时提到的那些使用质量特性。

>

> 为什么非功能测试越来越重要？因为今天软件之间在功能上往往拉不开差距——你有的按钮，别人也有。真正让用户流失的是：双十一零点支付一直转圈、上传的照片被人拖走、系统连着跑三天就内存泄漏重启。非功能缺陷的代价，往往比功能缺陷大得多，因为它一坏就是全局性的，而且很难靠改几行代码补回来——它本质上是架构问题。

>

> 打个生活里的比方：一家餐厅，菜品齐全、每道菜都做得对，这是功能合格；可要是这家店只有十个座位，中午来一百个人就排到马路上，空调还坏了，你下次还去吗？
>
> **板书：**系统非功能性测试 → 环境＝整个计算机环境（软硬件平台＋支持软件＋数据）→ 前提＝实际运行环境 → 内容：性能 / 安全 / 稳定 / 兼容 / ……（省略号＝ISO/IEC 25010 其他质量特性）→ 特征：多为架构层面问题，代价全局化。

提问：同一款 App，在开发同事的高配手机上测响应很快，能不能据此判断"性能没问题"？

> **预设回答：**第一种："能啊，实测就是快。"这是错误回答——忽略了环境代表性和负载条件。第二种："不能，要约定目标机型、网络条件、并发量和数据量。"这是正确回答。第三种："性能就是没有卡顿。"理解过于片面。教师点评：由第二种引出"环境＋负载＋指标"三要素；然后强调，非功能测试如果不先把这三样定义清楚，结论就是不可复现、不可比较的，写进报告也没人信。

易错点 / 考点：常见混淆是把功能与非功能混为一谈，以及把"稳定性"和"可靠性"随手混用；最容易漏掉的是"实际运行环境"这个前提条件。考试常考选择题"下列哪项不属于非功能性测试"，或案例题要求为某系统列出非功能测试项。判断三步法：能不能用→功能；好不好用、扛不扛得住→非功能；换台机器、换个负载结论就变→非功能测试必须写清环境与指标。

> **过渡：**系统测试把功能和非功能都验完了，最后谁说了算？是掏钱的用户。下一页，我们讲交付前的最后一道闸门——验收测试。

【2.68】验收测试（5 分钟）

验收测试是软件测试四个层次里的最后一层，也是交付前的最后一道闸门。

>

> 它的目的说白了就一句话：向未来的用户表明，系统能够像预定要求那样工作。注意"表明"这两个字——验收测试带有很强的证明色彩，你要拿出证据告诉用户：你要的功能都在，性能也和你的合理期待一致。

>

> 定义里还有两个词要抠。第一个是"未来的用户"，也就是验收测试站在真正掏钱、真正要用这套系统的人这一边。第二个是"合理期待"。为什么不说"达到需求规格说明书"就够了？因为用户不会拿着你的说明书逐条比对，他心里的标准是——我上次用的系统是这样，我同事说那个功能应该这样，宣传册上写的是这样。这些没说出口的东西，就是隐含需求。而验收测试要管的，恰恰是这些"没写下来、但用户认为理所当然"的部

> 分。

>

> 验收测试还有一项很具体、也最容易被忽略的工作——安装测试，也叫部署验证。它是按照软件产品的安装手册或相应文档，在一个和用户使用该产品完全一样的环境中，或者相当于用户使用环境中，一步一步地做安装操作测试。为什么强调"一步一步"？因为用户就是照着手册一步步做的。手册里少写一句"先停服务"，用户就能给你装崩。所以安装测试不只测软件，也在测文档。

>

> 打个比方：你网购了一套衣柜，图纸上写"先装底板再装侧板"，你照着做，结果发现孔位对不上——这不是你笨，是这套图纸和产品没有经过"安装测试"。软件也一样。
>
> **板书：**验收测试 → 目的：向未来的用户表明系统能像预定要求那样工作 → 验证：功能＋性能＝用户的合理期待（含隐含需求）→ 安装测试（部署验证）：按安装手册/文档，在与用户环境完全一样或相当的环境中，一步一步做安装操作 → 一句话：一层管"用户认可"，一层管"装得上"。

> **提问：**需求规格说明书上写的功能全部测通过了，验收测试是不是就一定能过？

> **预设回答：**第一种："能，都按需求验过了。"这是典型错误——把"符合规格"当成了"用户满意"。第二种："不一定，还要看隐含需求、真实数据与业务量、实际环境和用户的使用习惯。"这是正确方向。第三种："验收测试是我们测试组自己跑一遍。"这是把主体搞错了，验收的主体是用户或客户。教师点评：借第二种回答讲清验证与确认的区别——验证是"把东西做对了"（符合规格），确认是"做了对的东西"（满足用户）；验收测试偏向确认，所以规格之外的合理期待同样要核。

> **【思政落点 1】**讲什么：定义里那句"如同用户所合理期待的那样"。怎么联系知识点：验收测试要管隐含需求，说明质量责任不只在纸面上那几行条款。落到什么价值观：诚实交付、对用户负责，不做"说明书上没写就不算我的事"的推诿——把用户没想到的风险替用户想到，这才是工程人员的职业良心。

> **易错点 / 考点：**第一，验收测试与系统测试混淆——判断点在主体与视角：验收由用户/客户主导、看"能否接受"，系统测试由测试团队主导、看"系统是否符合规格"。第二，漏掉安装测试其实属于验收测试。第三，把"合理期待"只理解成需求文档。考试高频点：α、β 测试都属于验收测试的形式（下一页面）。案例题常问"上线验收前应做哪些准备"。

> **过渡：**用户到现场来验，是一回事；用户不在现场，甚至成千上万个用户分散在全国各地，又该怎么验？下一页我们看 α 测试、β 测试，以及一个更有争议的做法——在线测试。

【2.69】α、β 测试 &在线测试（6 分钟）

这一页有三个概念：α 测试、β 测试，还有在线测试。课件用的是英文原文，我们一条一条来读。

>

> 第一段，α 测试。原文是：Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site，后面跟了一句，Is a form of internal acceptance testing。翻过来就是——α 测试是由潜在用户、客户，或者一支独立的测试团队，在开发方的场所进行的模拟的或实际的操作性测试；它属于一种内部验收测试形式。三个要点：谁来做（潜在用户、客户或独立测试团队）、在哪做（开发方场所）、什么性质（内部验收测试）；而且它既可以是模拟的，也可以是实际操作的。

>

> 第二段，β 测试。原文说：Beta testing comes after α testing，Versions of the software, known as beta versions, are released to a limited users outside of the programming team。翻过来——β 测试在 α 测试之后进行；把软件的某个版本，也就是我们说的 β 版本，发布给编程团队之外的有限用户使用。三个关键词：α 之后、有限用户、编程团队之外。说白了，α 是在自家院子里试车，β 是把车交给一批志愿车主开上路，让真实世界帮你挑毛病。

>

> 第三个概念是 Testing in production，在线测试，也常译成生产环境测试。它的思路是：软件已经在真实环境里跑了，那就直接在真实环境里测试、观测、比对，而不是另搭一套测试环境。为什么会出现这种做法？因为再像的测试环境也不是生产环境——真实的用户量、真实的数据分布、真实的地域网络差异，你复制不出来。

>

> 课件还给了两个延伸阅读的链接：一个是微软 MSDN 上 seliot 的博客，另一个是 thetestingplanet 在 2011 年 11 月的那篇《The Future of Software Testing Part One: Testing in Production》。从这两条链接能看出，这个做法讨论得很早，不是新潮概念。

>

> 但要注意分寸：在线测试能做，前提是有兜底——灰度发布、功能开关、实时监控、随时能回滚。这几样缺

了，在线测试就是拿用户当小白鼠。

板书：α 测试：开发方场所 / 潜在用户・客户・独立测试团队 / 模拟或实际操作 / ＝内部验收测试。β 测试：在 α 之后 / β 版本 / 编程团队之外的有限用户 / ＝外部真实环境试运行。在线测试（Testing in production）：在真实生产环境测试与观测；配套：灰度、开关、监控、回滚。延伸阅读：MSDN seliot 博客；thetestingplanet 2011.11《The Future of Software Testing Part One: Testing in Production》。

提问：一个团队想在正式上线后做在线测试，于是直接把新版本全量发布给所有用户使用，行不行？

预设回答：第一种："行，真实用户测出来的问题最真实。"这是错误回答——只看到真实性，忽略了风险与兜底手段。第二种："不行，要先小流量灰度、加功能开关、布好监控和回滚预案，用少数用户先验证。"这是正确回答。第三种："在线测试就是让用户帮忙报 bug。"理解片面。教师点评：由第二种讲"真实"与"可控"这对矛盾——在线测试的价值在真实性，底线在可控性，两者靠工程手段同时满足；再提醒一句：没有回滚能力的全量发布，不叫测试，叫赌博。

易错点 / 考点：第一，α 与 β 的顺序和场所最容易搞反，口诀是"α 在自家、β 在门外，α 先 β 后"。第二，把 α、β 排除在验收测试之外——课件明确 α 属于内部验收测试形式，β 是交给外部有限用户。第三，把在线测试理解成"上线了就不用测试"。考试常考选择题"下列关于 β 测试的说法正确的是"，干扰项多在场所、参与人、先后顺序上做文章。

过渡：到这里，测试层次这条线就走完了：单元、集成、系统、验收。接下来我们要换一个视角——不看被测对象，而是看测试这份工作本身：测试人员一年到头到底在忙什么？这就是 2.7 软件测试工作范畴。

【2.70】2.7 软件测试工作范畴（4 分钟）

从这一页开始，我们进入 2.7 节——软件测试工作范畴。前面 2.1 到 2.6，讲的是概念、方式、层次；从这一节起，我们把视角从"测什么、怎么测"转到"测试这份工作到底由哪些环节构成"。

>

课件在这一节的开头放了一张结构图，另外配了一段讲义编写说明。这段说明本身很值得念一遍，因为它规定了接下来每一个知识点要从六个角度展开。第一是概念：这个概念是什么，在工作中什么场景会用到。第二是讲解：要胜任这个能力，需要掌握哪些知识和技术。第三是举例：用一个案例或 demo 帮大家理解，并且知道怎么用。第四是分享经验：特别是理论和实际有差异的地方。第五是学习建议：在相关学习和工作场景中怎么用。第六是实用工具：这个知识点对应的工具和模板。

>

> 为什么要按这六步走？因为测试是一个工程实践性极强的岗位。你光记住"测试计划包含七个要素"没有用，你得知道在你们公司这份计划是给谁看的、评审会上谁最可能挑刺、写完之后靠什么跟踪。所以后面每一节，我都会尽量把这六件事补齐。

>

> 那么测试工作一共有哪些环节？按教材的划分，软件测试的工作范畴包括六块：测试需求分析、测试策略制定、测试计划、测试设计、测试执行，以及测试结果和过程评估。请大家先记在心里，这是一个闭环——先想清楚测什么，再定怎么测，再排计划，再做设计，然后执行，最后评估并反馈回需求。

>

> 先看这张全局结构图，下一页我们把测试自身的工作流程展开。
>
> **板书：**2.7 软件测试工作范畴 → 六环节闭环：需求分析 → 策略 → 计划 → 设计 → 执行 → 结果与过程评估 →（反馈回需求）。讲义六步法：概念 → 讲解 → 举例 → 分享经验 → 学习建议 → 实用工具。

提问：这六个环节里，你觉得哪一个最容易被团队跳过或者敷衍？为什么？

> **预设回答：**第一种："测试执行，因为最累。"这说明只把执行当成产出环节，忽略了前面的设计决定执行价值。第二种："测试需求分析和策略，赶进度的时候容易被跳过，直接开始写用例。"这是正确方向。第三种："评估，项目一结束大家就散了。"也有道理。教师点评：肯定第二种，指出需求分析与策略是"地基"，跳过它们，用例写得再多也可能是无效覆盖——地基没打，砖砌得再多也白搭；同时表扬第三种观察，说明评估环节确实常被形式化。

易错点 / 考点：六环节的名称与顺序（常以多选题或排序题出现）；最容易混的是"测试策略"和"测试计划"——策略回答"怎么测"，计划回答"谁、在什么时候、用什么资源做什么"；另一个常见遗漏是只记到"执行"，把"评估"丢了。口诀：需、策、计、设、执、评。

> **过渡：**知道了有哪六块，还要看它们是怎么串起来的、各自的产出物是什么——下一页是软件测试自身的工作流程。

【2.71】软件测试自身工作流程（3 分钟）

这一页是一张流程图，页面上没有文字说明，所以我要带着大家把这张图读出来。

>

> 图的左边是入口，起点是测试需求——从需求文档、设计文档、产品功能说明里提炼出"要测什么"。往右走第一站是测试策略：决定手工还是自动化、黑盒还是白盒、先测哪块后测哪块。第二站是测试计划：把范围、测试项与优先级、风险、进度、资源、跟踪控制机制定下来，落到文档和排期上。第三站是测试设计：先做总体设计，把测试方案和测试结构定下来；再做详细设计，把一条条测试用例写出来。再往右是测试执行：手工的、自动化的，按用例跑，也允许探索式地去试。最后一站是测试结果与过程评估：看覆盖率够不够、缺陷是不是在收敛、当前版本质量如何，同时把计划里说的活动和实际做的事对一对。

>

> 评估的结果要往回反馈：缺陷多的模块，下一轮要加测；覆盖不到的路径，要回到设计补用例；需求变了，要回到需求分析重新界定范围。所谓"测试自身的工作流程"，重点在"自身"两个字——它和开发流程是两条并行的线，但两者要互相咬合。

>

> 读这张图，请大家盯住三样东西：起点在哪里、每一站的产出物是什么、哪些箭头是回头的。
>
> **板书：**流程主线：测试需求 → 测试策略 → 测试计划 → 测试设计（总体/详细）→ 测试执行（手工/自动化/探索）→ 测试结果与过程评估 →↩️ 反馈回需求与设计。读图三问：起点在哪？每站产出物？哪些箭头回头？

提问：照着这张图，如果你手上只有一份需求文档，下一步应该产出什么？再下一步呢？

> **预设回答：**第一种："直接开始写测试用例。"这是典型错误——跳过了需求分析、策略和计划。第二种："先做测试需求分析，明确范围与优先级；再定策略、排计划，然后才设计用例。"这是正确回答。第三种："先搭测试环境。"这是把手段当成了流程起点。教师点评：重点强调"产出物"意识——每一步都要有可评审、可交接的产物，没有产出物的流程等于没走；然后追问一句：测试用例是这张图里哪一站的产品？把答案落在"测试详细设计"上。

易错点 / 考点：流程顺序与各阶段产出物的对应关系（考试爱考"测试用例属于哪个阶段的产物"→测试详细设计；"测试计划属于哪一站"→第三站）；另一个易错点是把测试环境搭建当成流程起点。判断口诀：先想测什么（需求）→ 再想怎么测（策略）→ 再排怎么做（计划）→ 再写下来（设计）→ 再动手（执行）→ 最后回头看看（评估）。

> **过渡：**流程的第一站，也是最容易被敷衍的一站，就是测试需求分析——下一页我们把它讲透。

【2.72】测试需求分析（5 分钟）

测试需求分析，是整个测试工作的第一站。它要回答的是最原始的那个问题：到底测什么。

>

课件给了三件事。第一件，明确测试范围——哪些功能点要测试、哪些功能点不需要测试。注意后半句，"哪些不需要测"同样重要。为什么？因为测试资源永远是有限的，把范围写清楚，既是对测试负责，也是对项目和用户负责：白纸黑字说明这个版本不测这个模块，出了问题是大家都知道的风险，而不是测试漏了。

>

第二件，知道哪些测试目标优先级高、哪些优先级低。范围定了，还要排序。核心流程、用户高频使用的功能、涉及资金和隐私的部分，优先级最高；边缘的、内部的管理功能，可以往后排。为什么排序这么重要？因为现实是，测试时间永远会被压缩。会压缩到什么程度？压缩到只能测最重要的那几条。所以优先级不是锦上添花，它是你在最后关头决定"先测什么"的依据。

>

第三件，要完成哪些相应的测试任务，才能确保目标的实现。这句话是从目标倒推任务：要实现"核心支付流程可用"这个目标，需要哪些任务？准备测试数据、搭建环境、设计多少条用例、执行几轮回归、分别由谁来做？这些想清楚了，后面的计划才有东西可排。

>

我给大家一个判断标准：测试需求分析做得好不好，就看它能不能回答三个问句——测什么？不测什么？先测什么？

>

打个生活里的比方：装修房子的时候，设计师上来不会先问你要买什么牌子的油漆，他会先问几口人住、要不要书房、预算多少、什么时候入住。范围、优先级、条件先对齐，后面才不会返工。测试需求分析就是这个"先对齐"的环节。

>

做这一步时还要同步考虑一个词——可测试性。需求里写"系统应具有良好的用户体验"，这条你没法测。遇到这种情况，正确做法是回到需求评审把二义性问清楚，而不是假装它不存在。

> **板书：**测试需求分析＝测什么 → ①范围：要测的功能点 / 不测的功能点 → ②优先级：高 / 低 → ③从目标倒推测试任务 → 检验三问：测什么？不测什么？先测什么？ → 同步识别"可测试性"（含糊需求回评审澄清）。

> **提问：**某条需求写着"系统应具有良好的用户体验"，你怎么把它变成可测试的测试需求？

> **预设回答：**第一种："照抄进测试需求，列成一条测试项。"这是典型错误——无法判定通过与否，等于没测。第二种："追问具体场景与指标，比如首页首屏加载时间、完成关键操作的步数、错误提示是否看得懂。"这是正确方向。第三种："直接删掉这条需求。"做法太粗暴，应当回评审确认，而不是单方面删除。教师点评：把"二义性"当成一类缺陷来处理，说明测试需求分析的实质是把需求翻译成可验证的语句；同时强调必须找需求方确认，测试人员不能自己给需求加戏。

> **易错点 / 考点：**第一，把"测试需求"等同于"软件需求"——测试需求是测试视角的转化结果。第二，只列要测的，不列不测的。第三，漏掉优先级。考试常见题型：给出需求文档片段，要求写出测试范围与优先级，或判断"该需求是否可测"。口诀：范围清、优先级明、任务能落地。

> **过渡：**知道自己要测什么、先测什么之后，下一个问题就是怎么测才最划算——这就是软件测试策略。

【2.73】软件测试策略（6 分钟）

> 测试策略，是这一章里最能体现"测试是一门工程"的一页。它要做的决定，是"怎么测最划算"。

>

> 课件把要考虑的因素分成三类。第一类，测试方式。这里面有几组权衡：手工方式还是自动化方式，静态方式还是动态方式；用探索式测试还是基于脚本的测试；是自己团队测，还是众测、外包。第二类，测试方法。是黑盒方法还是白盒方法，是基于数据流还是基于控制流，是做完全组合测试还是做组合优化测试。第三类，测试过程。先测什么、后测什么，测试阶段怎么划分。

>

> 我们挑几组讲透。第一组，手工还是自动化。大家容易有个误解，觉得自动化高级、手工落后。其实不是。自动化适合那些稳定、重复、回归性的工作——每次发版都要跑一遍的登录、下单、支付主流程；手工适合一次性的、需要人的判断和感受的工作——界面好不好看、提示语读起来别扭不别扭、新手会不会迷路。拿自动化去测"这句话说得顺不顺"，那是拿锤子拧螺丝。

>

> 第二组，完全组合还是组合优化。假设有十个参数，每个参数三个取值，完全组合就是三的十次方，将近六万种组合，这还没算操作顺序。你根本跑不完。所以工程上常用正交试验、成对组合这类组合优化方法，用很少的用例覆盖大部分缺陷。这里的关键认知是：测试不可能穷尽，策略就是在"测不完的世界"里做取舍。

>

> 第三组，自己团队测还是众测、外包。自己团队了解产品，但视角容易固化、存在盲区；众测人多面广、贴近真实用户，但结果质量参差、保密风险要评估；外包省人力，但沟通成本和知识沉淀是代价。

>

> 课件最后给了一句目的性很强的话：制定或选择更合适、更有效的测试方式、方法和技术，目的是以最低的时间或人力成本，达到最大程度地揭示产品的质量风险，尽快完成测试。请注意，这句话里有三个诉求同时存在——成本要低、风险要露出来、时间要快。它们在互相打架，策略就是在打架的三者之间找平衡点。

>

> 所以请记住：策略不是"选最先进的技术"，策略是"为这个项目、这个阶段、这批人做出最合适的取舍"。
>
> **板书：**测试策略 → 三类因素：①测试方式（手工↔自动化 / 静态↔动态 / 探索式↔基于脚本 / 自测↔众测・外包）；②测试方法（黑盒↔白盒 / 数据流↔控制流 / 完全组合↔组合优化）；③测试过程（先测什么后测什么 / 测试阶段划分）。目标：最低时间人力成本 ＋ 最大程度揭示质量风险 ＋ 尽快完成（三角平衡）。

> **提问：**一个每月发版四次的产品，主流程回归每次都靠两个人手工跑两天，你觉得策略该怎么改？为什么不是"全都自动化"？

> **预设回答：**第一种："全都自动化，手工不要了。"这是错误回答——忽略了一次性测试、体验类测试，也没算脚本的开发和维护成本。第二种："把稳定的回归主流程自动化，探索式和体验类仍然手工，同时评估脚本的开发与维护成本。"这是正确方向。第三种："直接加人扛。"这是在回避问题。教师点评：由第二种引出"自动化成本＝开发＋维护＋失效后修复"，强调自动化是有投资回报门槛的；选择标准就三句——稳定、高频、结果可判定。

> **易错点 / 考点：**第一，把测试策略和测试计划混为一谈——策略回答"怎么测"，计划回答"谁、何时、用什么资源做"。第二，把"组合优化"理解成"少测几个功能"——它其实是方法层面的覆盖设计。第三，以为自动化一定优于手工。考试常以案例题出现："请为该产品制定测试策略，并说明你选择这种测试方式的理由。"

> **过渡：**策略定了，接下来要把它变成可执行的安排——排期、分人、定资源。这就是测试计划。

【2.74】测试计划（5 分钟）

> 测试计划，是把策略落到纸面和排期表上的那一步。课件列了七项内容，我们一项一项看，每一项都问一句：不写会怎样？

>

> 第一，目标和范围。这份计划要达成什么，管到哪儿为止。不写，测试做完了没人知道算不算完成。

>

> 第二，测试项及其优先级。这次要测哪些功能、哪些模块、哪些非功能特性，谁先谁后。不写优先级，一遇到延期就只能慌。

>

> 第三，测试风险识别与分析。这一条最容易被写成空话。我给大家几个常问的风险方向：需求变更频繁、测试环境不稳定、开发提测延迟、人手不足、第三方接口不可用。写风险不是凑数，是要连着写应对措施——比如提测延迟，就要写明每延迟一天，测试时间怎么压缩、优先保哪些测试项。

>

> 第四，指定测试策略。也就是把上一页讲的取舍写进来，作为本次测试的方法依据。

>

> 第五，进度安排。什么时候测哪个阶段，里程碑是什么，什么时候能给出验收结论。进度要跟开发进度对齐，关键节点最好倒排。

>

> 第六，资源配置。人力、环境、设备、工具、测试数据。这里"测试数据"经常被遗漏，可它常常是最大的瓶颈——真实数据拿不到、脱敏规则没定，执行阶段就卡住了。

>

> 第七，跟踪和控制机制。进度怎么跟踪，日报还是看板？谁来汇总？偏差到什么程度要启动调整？测试计划不是写完就归档的文档，它是活的：需求变了要改，进度偏了要调，所以必须有跟踪和控制机制兜住。

>

> 最后强调一点：测试计划不是测试组长一个人的作业。它要跟项目计划对齐，要经过评审，要让开发、产品、

运维都知道里面关于他们的承诺。

板书：测试计划七要素 → 目标与范围 → 测试项及优先级 → 风险识别与分析（必须带应对措施）→ 指定测试策略 → 进度安排 → 资源配置（人/环境/设备/工具/数据）→ 跟踪与控制机制。口诀：目标范围、项与序、风险、策略、进度、资源、跟踪控。

提问：如果你们的测试计划只能删掉一项，删哪项损失最小？如果只能保留一项，留哪项？

预设回答：第一种："删跟踪控制机制，文档写完就完了。"这是错误回答——计划会因此失去活性，没人跟踪就等于没有计划。第二种："保留目标与范围，因为其他各项都围绕它展开。"这个判断比较合理。第三种："删资源配置，人不够是项目经理的事。"这是错误回答——资源需求不写进计划，测试就没法争取到人。教师点评：强调"目标是纲"，其余六项都是它的支撑；同时点出跟踪控制是计划能落地的关键，也是考试里最常被漏掉的一项。

易错点 / 考点：七项内容与"策略 vs 计划"的边界；常考多选题"以下属于测试计划内容的有"，干扰项多为测试用例细节（那属于测试设计）。另一处易错：风险评估只列风险清单、不写应对措施。判断口诀：目标定方向、范围划边界、项序排先后、风险带对策、策略定方法、进度定时间、资源定投入、跟踪保落地。

过渡：计划里写着"要做哪些测试"，可具体一条一条到底怎么测出来？这就进入测试设计。

【2.75】测试设计（6 分钟）

测试设计，解决的是"如何测"这个问题。前面测试需求分析回答了"测什么"，策略和计划回答了"用什么方式、什么时间、多少资源测"，到了测试设计，就要把它落成一条条具体的、可执行的测试。

>

课件把测试设计分成两层。第一层，测试总体设计。它主要指测试方案的设计和测试结构的设计。测试结构是什么意思？就是把整个测试对象组织成一个层次——测哪几个大块，块与块之间是什么关系，每个块下面再分哪些功能点、哪些测试项，形成一棵树。有了这棵树，你才知道覆盖率该怎么算：叶子节点覆盖了多少。

>

测试方案的设计，范围比较大。课件列了五件事：选择测试方法、明确测试策略、设计测试技术路线、选择测试工具、规划测试环境。我把它们串成一句话：用什么方法、走什么技术路线、拿什么工具、在什么环境里、按什么策略去测。测试环境这一项常常被低估——环境规划包括要几套环境、硬件配置、数据库与中间件版本、测试数据从哪来、环境之间怎么隔离和复用。环境不对，前面设计得再好也白搭。

>

> 第二层，测试详细设计，主要指测试用例的设计。这是最接地气的一层：一条用例要有编号、所属模块、前置条件、输入数据、操作步骤、预期结果、优先级，必要时还要关联需求编号。

>

> 这里有个常见的认知误区：很多人以为"测试设计"就是"写用例"。其实写用例只是详细设计，前面还有总体设计。为什么总体设计不能省？因为它决定了覆盖的思路——如果连测试结构都没搭起来，用例就是一堆散装条目，覆盖率没法算，哪一块遗漏了也看不出来。这就是我们课程设计里为什么要求先画测试项结构，再写用例。

>

> 打个比方，测试设计就像装修的施工图：总体设计相当于平面布局图——哪里是客厅、哪里是卧室、水电怎么走；详细设计相当于每个柜子的加工尺寸图。只有尺寸图、没有布局图，装出来的房子是住不了人的。
>
> **板书：**测试设计＝解决"如何测" → ① 总体设计：测试方案设计（选测试方法 / 明确测试策略 / 设计技术路线 / 选测试工具 / 规划测试环境）＋测试结构设计（测试项树形分解→可算覆盖率）；②详细设计：测试用例设计（编号/模块/前置条件/输入/步骤/预期/优先级/关联需求）。类比：布局图 vs 加工图。

提问：给"用户登录"功能做测试设计，总体设计和详细设计分别要产出什么？

> **预设回答：**第一种："上来就写十条用例。"这是典型错误——跳过了总体设计。第二种："总体设计产出登录相关的测试项结构（正常登录、异常登录、安全、兼容、性能等）、方法选择（等价类、边界值）、环境与数据准备；详细设计产出一条条用例，写清账号密码取值、预期结果和优先级。"这是正确回答。第三种："只画结构不写用例。"这是另一个极端。教师点评：强调两层缺一不可；再联系本课的实践任务"用等价类和边界值为登录功能设计十条用例"——那十条属于详细设计，而它前面的分析属于总体设计。

易错点 / 考点：第一，把"测试设计"等同于"写用例"，漏掉总体设计。第二，总体设计与详细设计各自包含什么（高频选择题）。第三，把测试环境规划当成运维的事而不写进测试方案。口诀：总体定方案与结构，详细出用例；先有结构后有例，覆盖率才算得清。

> **过渡：**详细设计的产物就是测试用例。下一页我们专门说说，为什么测试用例这么值得花时间去写。

【2.76】测试用例（5 分钟）

> 测试用例，是测试详细设计的产物，也是测试人员手里最实在的东西。课件给了它四层价值，我们一层一层来看。

>

> 第一层，它是测试人员在测试过程中的重要参考依据。什么依据？执行的时候照着它做：前置条件是什么、输入什么、怎么操作、期望看到什么。有了它，测试就不再依赖某个人的记忆。这一点在多轮测试里尤其重要——第一轮谁测的、第二轮换个人来测，结论必须一致。

>

> 第二层，它有助于节约测试时间、提高测试效率。这里要澄清一个误解：写用例看起来费时间，怎么反而省时间？因为它把"想"和"做"分开了。设计的时候集中精力想清楚测什么、怎么判断对错；执行的时候就是照做、记录、比对。如果边想边做，一个人一上午可能只推进两三个功能点，而且判断标准飘忽不定。

>

> 第三层，良好的测试用例会不断被重复使用，使得测试过程事半功倍。这句话的关键词是"重复使用"。一个版本发一次，回归一轮；发十次，就回归十轮。同一批用例被用了十遍，它的开发成本就被摊得很薄。这也是自动化脚本能够落地的前提——没有稳定、描述清晰的用例，自动化无从谈起。

>

> 第四层，测试用例是一个知识积累的过程。一位老测试人员的价值，很大一部分就沉淀在他们积累的用例库里：那些踩过的坑、遇到过的边界、闻所未闻的异常输入，都写在用例里。新人来了，读用例往往比读需求文档学得快。用例库越厚，团队的整体能力就越不依赖个别人。所以用例不仅要写，还要评审、维护、归档、复用。

>

> 一句话总结这一页：用例是测试的"规格说明书＋知识库＋回归资产"。

> **【思政落点 2】**讲什么：用例库是团队一点一点积累起来的知识资产。怎么联系知识点：正因为良好用例会不断被重复使用、成为回归与自动化的基础，它才必须被认真评审、长期维护。落到什么价值观：踏实记录、精益求精的工匠精神——反对"能用就行、随手一测"的应付态度，也不藏私，让个人经验变成团队能力。

> **板书：**测试用例四层价值 → ①执行的参考依据（统一标准、不依赖记忆）→ ②节约时间、提高效率

（把"想"与"做"分离）→③不断重复使用、事半功倍（回归资产）→④知识积累（团队能力沉淀，要评审、可维护、可复用）。

提问：为什么说"没有用例也能测"，但长期看仍然必须写用例？

预设回答：第一种："因为公司要求写文档。"这是表面回答。第二种："为了统一标准、支撑回归复用、沉淀团队知识，还能作为自动化的基础。"这是正确回答。第三种："有了用例就不需要探索式测试了。"这是把两者对立起来，错误。教师点评：借第二种承接探索式测试与基于脚本的测试的关系——临场探索可以没有用例，但要把探索的发现固化成团队资产，就必须回写用例；最后强调"可重复使用"才是用例最大的价值。

易错点 / 考点：第一，把"节约时间"理解成"写用例很快"，它省的是执行与回归的时间。第二，漏掉"知识积累"这一层（简答题常考这个要点）。第三，用例要素记不全。判断口诀：依据—效率—复用—沉淀。考试常考：测试用例的作用（简答/多选）；以及判断题"测试用例是测试详细设计的产物"。

过渡：用例写好了，接下来就是照着它去跑——测试执行。这里面有两条路：手工执行和自动化执行。

【2.77】测试执行（5 分钟）

测试执行，终于到了动手的环节。课件把它分成两种：手工执行和自动化执行。

>

先看手工执行。定义是：基于详细设计的测试用例来完成测试，也可以在没有测试用例的情况下进行探索式测试。这两句话要拆开看。前半句是我们最熟悉的——拿到用例，按步骤操作，比对预期结果与实际结果，记录通过还是失败。这里我要强调执行的质量：一个负责任的执行，不只是"照着点"。你要如实记录实际结果，包括那些"虽然过了、但总觉得怪怪的"现象；遇到失败，要留下可复现的步骤、截图、日志和环境信息。为什么？因为缺陷报告的质量直接决定开发能不能一次修对。很多扯皮，都源于一句"反正就是不行"。

>

后半句是探索式测试：在没有测试用例的情况下进行的测试。注意，"没有用例"不等于"没有方法"。探索式测试靠的是测试人员的经验、对产品的理解和即时的判断，边学、边设计、边执行、边调整。

>

再看自动化执行。定义是：采用测试工具来完成，一般都需要开发自动化测试脚本，然后由工具执行脚本。这意味着自动化执行的前期投入在脚本开发上——所以它适合反复执行的场景，只用一次的场景不值得为它

> 写脚本。

>

> 课件还提示我们，关于自动化的更多内容，会在后面"单元测试与集成测试""系统测试"和"自动化测试框架"等章节详细讨论，这一页先把概念建立起来。

>

> 执行阶段还有几件配套的事要记住：一是执行的纪律——用例没跑完不能报"测试完成"，跳过的用例要写明原因；二是缺陷的记录与跟踪——发现、提交、跟踪到关闭；三是执行证据的留存——日志、截图、测试数据。我们课程设计里要求大家"发现缺陷并记录、完成一个非规范的测试报告"，练的就是这一段功夫。

【思政落点 3】讲什么：执行时发现了一个对自己不利的结果——是如实记录，还是轻轻放过。怎么联系知识点：测试执行要求如实记录实际结果、保留失败证据、跟踪缺陷到关闭。落到什么价值观：诚信是测试人员的职业底线；隐去不利结果，等于把风险转嫁给用户，也辜负了团队对测试的信任。

> **板书：**测试执行 → 手工执行：基于详细设计的用例（操作—比对—记录）＋无用例的探索式测试（靠经验与判断，不是靠蛮点）→ 自动化执行：先开发脚本 → 再由工具执行 → 后续章节详述 → 执行三配套：纪律（不跳不瞒）、缺陷记录与跟踪、证据留存。

> **提问：**一位测试同事跑用例时发现某条用例失败，但重跑又通过了，他觉得"重跑是好的，就记通过"——你怎么看？

> **预设回答：**第一种："同意，重跑通过说明没问题。"这是典型错误——把间歇性缺陷掩盖掉了。第二种："应如实记录，标注为偶现，补充日志、时间、环境、数据、出现次数等信息，作为缺陷提交并跟踪。"这是正确回答。第三种："直接报严重缺陷。"方向对，但不留证据，开发无法定位。教师点评：把"偶现缺陷"讲成典型案例——这类缺陷往往最危险，上线后就是偶发故障，且极难复现；用"如实记录＋留证据＋标偶现"三步法收口。

易错点 / 考点：第一，把自动化执行理解成"不用写脚本、工具自己会测"。第二，把探索式测试等同于随机乱点。第三，用"重跑通过即通过"处理偶现失败。考试常考：手工执行与自动化执行的适用场景对比；或"下列属于测试执行阶段工作的是"，干扰项常常是设计用例、编写测试计划这类其他阶段的工作。

> **过渡：**执行里的手工那条路，本质上就是照着脚本走的；自动化那条路，脚本更是核心。下一页我们把"基于脚本的测试"这个传统但基础的方式讲清楚。

【2.78】基于脚本的测试（6 分钟）

这一页讲基于脚本的测试，英文是 Scripted Testing，简称 ST。它是我们理解后面探索式测试的参照物。

>

先看它的核心特征，课件写得很简洁：先设计后执行。也就是说，测试动作不是临场发挥，而是先把要测什么、怎么测、怎么判断结果写清楚，写完了再去执行。这个"先"字，就是 ST 与探索式测试最大的分野。

>

第二个要点，Script 到底是什么？课件给了两种情况：在手工测试里，Script 就是 Test case，测试用例；在自动化测试里，Script 就是 Test Script，测试脚本。所以同一个词，在两种执行方式下的产物不一样——一个是给人看的步骤，一个是给工具跑的代码。

>

第三个要点，它的阶段很明显。课件列了四步：分析、设计、执行、报告。分析阶段理解需求、界定测试项；设计阶段把用例写出来；执行阶段按用例跑；报告阶段汇总结果、写测试报告。四个阶段一个接一个，边界清楚，所以课件说它属于较传统的测试方式。但传统不等于过时——今天大量的测试工作，尤其是回归测试和对合规性要求高的行业，仍然是基于脚本的。

>

第四个要点，课件写了一句很传神的话："有什么开发就有什么测试。"什么意思？开发的产物是需求、设计、代码，ST 就对应地有需求分析、测试设计、测试执行，一一映射。开发走瀑布，测试也走瀑布；开发按模块划分，测试就按模块划分用例。它的好处是结构清晰、可管理、可度量、可交接；它的代价是——需求一变，前面写好的用例就要返工，响应变化比较慢。

>

顺带说一下，课件在这一页的最后还附了一句 WebEx Meeting Center 的客户收益描述："使用 WebEx Meeting Center，可以加快业务决策，同时减少差旅成本和时间。"这句话本身是产品文档里的宣传语，不是测试内容。我把它摆在这里是想提醒大家：产品文档、宣传材料里对用户的承诺，恰恰是验收测试要逐条对照核验的依据。文档写了却做不到，测试就要把它标出来。

>

把这一页收成一句话：ST 是"先想清楚、写下来、再照着做"，它最大的优点是可控和可复用，最大的短板

是应对变化慢。

板书：基于脚本的测试 ST → 先设计后执行 → Script：手工测试＝Test case（用例）／自动化＝Test Script（脚本）→ 阶段明显：分析 → 设计 → 执行 → 报告 → 特点：较传统、结构清晰、可管理可度量可交接；响应变化慢 → 口诀"有什么开发就有什么测试"。附：WebEx 客户收益语（产品承诺＝验收核验依据）。

提问：一个需求频繁变更的项目，团队坚持全流程严格的基于脚本的测试，会遇到什么问题？该怎么调整？

预设回答：第一种："没问题，规范总是好的。"这是忽略成本的典型错误。第二种："用例维护成本高、返工多、执行进度被拖累；应该分层——核心稳定的部分保持脚本化，变化快的部分用探索式测试加轻量用例，并缩短设计与执行之间的间隔。"这是正确方向。第三种："干脆全部改成探索式测试。"这是走到另一个极端。教师点评：借第二种提前铺垫 ST 与探索式测试的对比关系，强调"不是选边站，而是按场景配比"；同时把下一节的问题抛出来。

易错点 / 考点：第一，把 Script 只理解成自动化脚本，漏掉手工测试的 Test case。第二，把 ST 的四个阶段记混——是分析、设计、执行、报告，不是需求、计划、设计、执行。第三，认为 ST 已经被淘汰。考试常考：ST 的特征（先设计后执行、阶段性明显），以及 ST 与探索式测试的对比（下一节展开）。记忆口诀：先设后行、四步分明、开发测试一一对应。

过渡：ST 要求先设计后执行，可如果项目节奏快到没时间先写用例呢？下一节我们看另一种风格——探索式测试，它是怎么把学习、设计、执行并行起来做的。

【2.79】探索式测试定义（3 分钟）

前面几页我们把静态测试、动态测试、手工测试、自动化测试这些"测试方式"逐一过了一遍，现在来到一个很有意思、也特别容易被误解的概念——探索式测试。这一页给出的定义，来自 Cem Kaner 在 2006 年 6 月 6 日软件测试协会会议上的主题演讲《Exploratory Testing After 23 Years》。标题里的"23 年"往前一推，就是 1983 年——也就是说，这个概念从出现到被系统总结，走了二十多年。

定义是这样说的：探索式测试是一种软件测试风格（Style），它强调独立测试人员（Individual Tester）的个人自由和职责（Personal Freedom and Responsibility），为了持续优化其工作的价值（Value），将测试相关学习（Test-related Learning）、测试设计（Test Design）、测试执行（Execution）和测试结果分析（Analysis）作为相互支持的活动，在整个项目过程中并行地执行。

请大家抓住四个关键词。第一是"风格"，它不是一种具体技术，也不是一个工具，而是一种做事的方式；第

二是"独立测试人员"，强调个人的自由和职责——自由给你了，责任也一起给你，你不能说"用例没写我就不会测"；第三是四个活动：学习、设计、执行、分析，它们是相互支持、并行进行的，不是先学完再设计、设计完再执行那种串行；第四是"持续优化工作的价值"，这句是灵魂——探索式测试不是乱测，它每一步都在问"我这一轮测下来，对项目到底有什么价值"。

生活里最好的类比是"老医师问诊"。新手照着检查单一项一项勾，老医师一边听你说、一边追问、一边当场决定下一个该查什么，学、想、查、判断是同时发生的。软件行业里也一样，一个刚上线、需求天天改的移动端产品，就特别适合用这种风格去探。

课件备注里还给了两个词：Interpretation 是"解释、说明"，Mutually 是"互为、相互地"。后者正好对应定义里那句"相互支持的活动"，请大家顺手记住。

板书：探索式测试（ET）→ 风格 Style ≠ 技术 → 独立测试人员：个人自由＋职责 → 学习→设计→执行→分析（相互支持、并行）→ 持续优化价值｜出处：Cem Kaner 2006.6.6 演讲（概念源头 1983 年）

提问：定义里说四个活动是"并行执行"的，那它跟"想到哪测到哪"的乱测，差别到底在哪里？

预设回答：①"没差别，就是随机点一点"——典型错误。要追一句：随机点的人有没有在记录、分析、调整下一轮策略？没有分析就没有探索，那只是碰运气。②"差别在于有没有用例"——部分对，探索式测试也可能有松散的用例甚至测试章程，关键是学习和设计在执行现场实时发生。③"差别在于测试人员的能力和责任心"——说得好，这正是定义里"个人自由和职责"的落点：自由越大，越依赖自律。

易错点 / 考点：①把探索式测试等同于 Ad-hoc（即兴）乱测——错，它是"有结构的即兴"。②把"并行"理解成"不做设计"——错，设计还在，只是挪到执行过程中去做。③选择、判断题常考这个定义的出处：Cem Kaner，2006 年主题演讲，概念源头可追到 1983 年。记忆口诀：**风格、独立、四活动并行、价值驱动**。

过渡：知道了探索式测试是什么，我们自然会问：它跟传统的、按用例一条条执行的测试，到底该怎么摆？下一页课件给出一条谱系。

【2.80】基于脚本的测试 vs. 探索式测试（3 分钟）

这一页是课件里第一张"对照图"，它把各种测试做法排成了一条从"规范"到"自由"的连续谱。谱的最左端是 Fully Scripted Testing，完全脚本化的测试，用例写得满满的，一步一步照着执行；往右走是 Automated Tests，自动化测试；再往右进入中段，出现了 Test Design、Test Execution，以及松散的测试用例和测试

场景；接着是 Bug Hunting，缺陷猎捕；再往右是 Exploratory Testing 探索式测试、Product Exploration 产品探索；最右端是 Ad-hoc Testing，完全自由的探索，想到哪测到哪。课件还摆出了"角色扮演测试"这一类做法，提示我们中段存在大量介于规范与自由之间的测试形式。

这张图最想告诉大家的一句话是：**这不是一道二选一的判断题，而是一条光谱**。左端追求可控、可重复、可度量；右端追求灵活、应变、靠人的判断。真实项目里你几乎不会只用一端，而是看阶段、看风险、看被测对象，来决定自己站在谱上的哪一段。回归测试、验收测试、需要反复执行的冒烟测试，就往左边站，越自动化越好；一个全新的功能、一份含糊的需求、一个还没被任何人碰过的模块，就往右边站，靠人去探。

为什么会有这么一条谱？因为左右两端的成本结构不一样。规范脚本前期投入大：要分析需求、要写用例、要评审，但一旦写好就能反复执行，边际成本很低；自由探索前期几乎不用投入，立刻就能开始，可它高度依赖执行者的水平和状态——今天这个人能发现的问题，换个人可能一个都发现不了。

所以判断标准很朴素：**这段测试会不会被重复执行很多次？需求是不是已经稳定？**两个都"是"，就往规范一侧走；两个都"否"，就往自由一侧走。

顺带说一句，课件把 Ad-hoc 放在最右端，也是在提醒我们：探索式测试虽然位于谱的右侧，但不等于最右端那个"完全自由的探索"，它是有方法、有记录、有分析的——这一点上一页讲定义时已经埋了伏笔。

板书：规范 ←————————————→ 自由

Fully Scripted Testing → Automated Tests → Test Design / Test Execution / 松散的测试用例、测试场景 → Bug Hunting → Exploratory Testing → Product Exploration → Ad-hoc Testing（完全自由的探索）｜中段还有角色扮演测试等形式

提问：同一个登录功能，如果它是长期维护的核心模块，你把它放在谱上的哪一段？如果它是本周才加的一个实验性开关呢？

预设回答：①"核心模块放左边，实验性功能放右边"——标准答案。追问为什么：核心模块回归次数多、需求稳定，脚本化和自动化的收益最高。②"都放右边，人工测发现的问题更多"——要纠正：人工发现的问题"看起来多"，但不可重复、不可度量，回归时无法保证不漏。③"都放左边，全写成自动化"——也要纠正：需求都没定的实验功能写脚本是浪费，需求一改脚本全废。

易错点 / 考点：①最易错：以为探索式测试就等于 Ad-hoc 测试。课件把两者分列在谱上的不同位置，Ad-hoc 是"完全自由的探索"，ET 是右侧但有方法的一档。②选择题常考两端的名称：最规范端是 Fully Scripted Testing，最自由端是 Ad-hoc Testing。③简答题常问"什么时候用脚本化、什么时候用探索式"，答

题三步法：**看重复次数 → 看需求稳定性 → 看风险与人员水平**。

过渡：谱系图讲完了，下一页课件把两端拉成一张逐条对照表，我们一条一条看。

【2.81】ET vs. ST -2（3 分钟）

这一页是探索式测试（ET）和基于脚本的测试（ST）的第二张对比页。上一页看的是它们在谱系上的位置，这一页看的是一条一条的工作方式差异。先看左边的 Scripted Testing，脚本化测试：第一，先设计、后执行，顺序不能倒；第二，强调逻辑分析，靠文档和推理把用例推出来；第三，关注需求和测试文档，文档是它的主战场；第四，有明确的测试标准——什么算通过、什么算失败，事先定死；第五，强调评审、可控，用例要评审，执行要可控；第六，总体上严谨、规范。

再看右边的 Exploratory Testing：第一，学习、设计和执行并行，边学边设计边测；第二，上下文驱动，测什么、怎么测，由当前这个产品、这个版本、这段时间的上下文决定，而不是照搬模板；第三，强调个人能力，同样的时间，高手和新人的产出差距极大；第四，Test Oracle，测试判定准则，这是重点——脚本化测试的判定标准写在用例里，而探索式测试常常要测试人员现场判断"我看到的这个现象算不算缺陷"；第五，关注与产品的交互，测试人员是真的像用户一样在用；第六，拥抱变化、乐趣，需求变了不慌，反而觉得有新东西可探。

我们把它压成一句话：**脚本化测试是把不确定性提前消灭在文档里，探索式测试是把不确定性留到现场、用人的判断去消化。**左边赌的是"我事先想得够全"，右边赌的是"我现场反应够快"。所以左边最怕需求变更，一变，用例全废；右边最怕人员不行，人一弱，测试就变成瞎点。

页面上标的是"ET vs. ST -2"，说明这是对比的第二页——第一页看谱系位置，第二页看工作方式，两页合起来才是完整的一对。

板书：ST：先设计→后执行｜逻辑分析｜重需求和测试文档｜标准明确｜强调评审、可控｜严谨规范

ET：学习＋设计＋执行并行｜上下文驱动｜靠个人能力｜Test Oracle 现场判定｜关注与产品交互｜拥抱变化、乐趣

一句话：ST 把不确定消灭在文档里；ET 把不确定留到现场消化

提问：右边提到探索式测试要有 Test Oracle，那如果一个测试人员说"我觉得这里不对劲，但我一时说不清为什么"，这算不算发现了缺陷？

预设回答：①"不算，说不清就不是缺陷"——要纠正：发现异常只是第一步，接下来要做的是最小化复现、对照需求文档和同类产品的行为，把它变成可描述、可举证的缺陷，说不清只是起点。②"算，凭直觉就行"——也要纠正：缺陷必须能复现、能说明影响，否则开发无法修、也无法验证。③"要看能不能稳定复现、是否偏离需求和用户期望"——好答案，这正好落到缺陷报告的要素上。

易错点 / 考点：① Test Oracle 常被译错成"测试预言"，标准说法是"测试判定准则/预期结果来源"，考试要能解释它回答的是"凭什么判断这次测试通过了"。②简答高频：ST 与 ET 的差别，至少要答出"是否先设计后执行""是否上下文驱动""对文档和标准的依赖程度"三条。③易错：把"拥抱变化"当成"不做计划"——ET 同样要有测试章程和时间盒。

过渡：测试做完了，结果就摆在那里，那我们怎么判断"测够了没有、这个版本能不能发"？下一页讲测试结果与过程评估。

【2.82】测试结果和过程评估（3 分钟）

前面几页讲的是怎么测，这一页讲的是测完之后怎么看。课件把它分成两件事：测试结果评估和测试过程评估。这两件事经常被混为一谈，其实一个看的是产品，一个看的是我们自己的工作。

先看测试结果评估，课件给了三个分析角度。第一，分析测试覆盖率，目的是了解测试是否充分——覆盖率高不代表质量一定好，但覆盖率低一定说明还有大片区域没人碰过，这是"测没测到"的问题。第二，做缺陷的趋势分析，看缺陷是越来越多还是越来越少，从而了解缺陷是否已经收敛——如果每天发现的缺陷还在往上走，说明这个版本远没到稳定的时候。第三，做缺陷的分布分析，看缺陷集中在哪几个模块、集中在哪几类严重程度，再基于缺陷来评估当前被测试版本的质量。注意这里的落脚点是"版本质量"，也就是为"这个版本能不能发布"提供依据。

再看测试过程评估。课件讲得很清楚：结合测试计划来进行评审，相当于把计划的测试活动和实际执行的活动做比较，了解测试计划执行的情况和效果。说白了就是"计划 vs 实际"的偏差分析：计划里要执行的用例，实际只跑了一部分，差在哪里？计划的回归时间被压缩了，是人力不够还是环境卡住了？过程评估的目的不是追责，而是发现流程里的堵点，让下一轮计划定得更靠谱。

记住这三层顺序：**先看结果够不够（覆盖率），再看问题收敛没有（趋势与分布），最后回过头看自己干得怎么样（过程偏差）。**很多同学写测试总结只写"测了多少条用例、发现多少缺陷"，那只是结果的一半；把覆盖率、趋势、分布和计划偏差补齐，才叫会评估。

> **板书：**测试结果评估 → 覆盖率（是否充分）→ 缺陷趋势（是否收敛）→ 缺陷分布（集中在哪里）→ 版本质量判断

测试过程评估 → 计划的活动 vs 实际的活动 → 偏差与效果 → 改进下一轮计划

三层顺序：测没测到 → 问题收没收敛 → 自己干得怎么样

> **提问：**如果 A 版本发现 100 个缺陷，B 版本发现 30 个缺陷，能不能直接说 A 的质量比 B 差？

> **预设回答：**①"当然更差，缺陷多"——典型错误。追问：这 100 个是随便点出来的，还是照着系统化的用例测出来的？测试强度不同，发现数就不可比。②"要看缺陷是否收敛、分布在哪里、测试投入有多少"——好答案，三个角度全用上了。③"要看严重程度"——对，同样是 100 个，90 个界面文案错误和 90 个数据计算错误，结论完全相反。

> **易错点 / 考点：**①最大误区：把"缺陷总数"直接等同于"质量水平"，必须结合测试充分性和投入来看。②"收敛"要会解释：新增缺陷数随时间下降，并且严重缺陷已基本关闭。③选择题常考两种评估的区分：结果评估看产品与版本，过程评估看计划执行。口诀：**覆盖看充分，趋势看收敛，分布看集中，过程看偏差。**

> **过渡：**到这里第 2 章的概念就全部讲完了，我们花几分钟把整章串成一张全景图。

【2.83】本章小结（3 分钟）

这一页是全章"收口"的地方，一共五条，我们一条一条过。

第一条：缺陷是质量的对立面。这句话的意思是，谈缺陷之前必须先谈质量——质量是什么？是软件满足规定需求和隐含需求、也就是满足用户期望的程度。所以课件提醒我们，要借助产品质量模型和使用质量模型来理解质量：产品质量模型看的是软件本身的属性，使用质量模型看的是软件被真正使用起来以后，对用户任务产生了什么效果。

第二条讲缺陷的因果链条：缺陷是内部错误，它在外部表现为失效。这一点非常关键——用户看不到你代码里那行写错的条件判断，用户只看到"点结算没反应"。而缺陷的来源很多，课件点了三个大源头：需求定义、设计和代码。需求写歪了，后面怎么努力都歪；设计漏了边界，代码再漂亮也补不回来。课件还强调一句：缺陷要尽早发现、尽早修正，否则带来的劣质成本越大——这就是我们前面讲过的成本放大效应，越往后修越贵。

第三条列了软件测试方式的几组对比：静态 vs 动态、主动 vs 被动、手工测试 vs 自动化测试、基于脚本的测试 vs 探索式测试。这一条是"横向"的，讲的是同一件事可以用哪些不同方式去做。

第四条列了单元测试、集成测试、系统测试和验收测试。课件这里字面写的是"软件测试方式"，但列出来的这四项其实是"测试级别"，也有教材叫测试阶段。易错点就在这里：方式回答"怎么测"，级别回答"在哪一层测、由谁来测"，这是两条不同的轴线。后面讲单元测试与集成测试的章节会专门展开这四级。

第五条讲软件测试的工作范畴，一共六件事：测试需求分析、测试策略制定、测试计划、测试设计、测试执行、测试结果和过程评估。这六件事就是大家做课程设计时要走完的完整链路，也是本课程第 3 篇各章的主题。

课件的落脚点写得很清楚：帮助同学们建立软件测试的整体全景图。这张图请务必装进脑子里——纵向是级别，横向是方式，中间贯穿的是六项工作，最上面压着的是质量与缺陷这条主线。

【思政落点 1】科学精神・质量观・诚信。质量模型和缺陷分类不是背概念，而是在训练一种"凡事讲证据、讲分类、讲因果"的科学态度：说质量要有指标，说缺陷要有复现和影响。再往前一步就是职业诚信——测试人员发现了缺陷却隐去不报，等于把风险留给用户、留给同事。所以这一章真正要立的规矩是：数据如实记录，不利结果不隐瞒。

板书：①缺陷＝质量的对立面（产品质量模型＋使用质量模型）②缺陷：内部错误→外部失效；来源＝需求/设计/代码；越早修越便宜 ③方式：静态 vs 动态、主动 vs 被动、手工 vs 自动化、脚本 vs 探索 ④级别：单元→集成→系统→验收 ⑤工作范畴：需求分析→策略→计划→设计→执行→结果与过程评估

提问：请用一句话说出"方式"和"级别"的区别，并举一个例子。

预设回答：①"方式是怎么测，级别是在哪一层测。比如对登录模块做单元测试——单元测试是级别；用白盒还是黑盒、手工还是自动化，是方式。"——标准答案。②"级别就是阶段，方式就是方法，差不多"——要纠正：混用会导致计划里写不清"谁在什么阶段用什么手段"。③"方式包含级别"——错误，正好用课件第四条那句表述来纠偏。

易错点 / 考点：①课件第四条文字写的是"软件测试方式"，但列的是单元/集成/系统/验收，实质是级别——考"测试级别有哪些"就答这四项。②顺序题：六项工作范畴的先后顺序（需求分析在计划之前）不能颠倒。③本章小结本身是简答题高发区，建议按"质量—缺陷—方式—级别—工作范畴"五段来背。

过渡：概念都清楚了，接下来是两件要动手做的事——课后作业和本章的思考题。

【2.84】作业（2 分钟）

这一页是本课程的第一次课后作业，我把要求念清楚，也讲清楚我到底想看什么。作业要求是：选择同类的两到三个产品，比如词典类、视频播放器这一类，对它们做一次质量比较。具体四件事：第一，详细地分析它们的外部质量和使用质量；第二，列出它们之间有明显差异的质量属性；第三，列出这类软件在需求评审时需要关注的功能点；第四，列出这类软件在设计评审时需要关注的非功能特性。

先解释两个词，不然大家容易写偏。外部质量，指的是从外面看得见、可以测量的质量特性，比如功能是否齐全、界面是否好用、运行是否流畅、在不同设备上是否兼容。使用质量，指的是放到真实使用场景里去看：它能不能帮用户把任务顺利完成、效率高不高、用户满不满意、有没有风险。同样一个词典软件，外部质量看它词库多大、查询多快；使用质量看你查一个生词到看懂例句，一共花了多少秒、中间有没有被广告打断。

第三条和第四条是把"比较"转换成"评审关注点"，这是本次作业最有价值的部分。需求评审关注功能点，就是在需求阶段要问清楚哪些功能、哪些流程、哪些边界；设计评审关注非功能特性，就是性能、安全、兼容、易用、可维护这些"不是功能却决定体验"的属性。为什么这么设计？因为这两个正是测试人员最应该提前介入的会议。

交作业时我提醒三点：一是写清你比较的是哪几个产品、什么版本、什么时候测的，这叫测试环境信息；二是质量属性的差异要给出证据，比如截图或具体操作步骤，不能只写"感觉 A 比 B 好"；三是把"我认为需要关注的点"和"它实际上做到了什么"分开写，前者是你的判断，后者是事实，混在一起就分不清哪是分析、哪是臆测。

这门课的考核方式是过程性考核加上课程设计作品与答辩，这份作业就是过程分的一部分，请认真做。

> **板书：**同类产品 2–3 个（词典、播放器…）→ ①外部质量＋使用质量分析 ②有明显差异的质量属性 ③需求评审关注的功能点 ④设计评审关注的非功能特性｜作业要求：写清版本与环境、差异要给证据、判断与事实分开写

提问：如果让你比较两款词典 App，你会用哪个指标来判断"使用质量"的差别？

> **预设回答：**①"看谁的词库大"——那是外部质量里的功能属性，还没到使用质量。②"查同一个生词，从输入到看完释义和例句的总耗时、总步骤数"——好答案，这就是任务级的效率指标。③"看应用商店评分"——可以当参考信号，但要追问：评分受价格、广告、运营影响，不能当作质量测量的主证据。

易错点 / 考点：①易错：把"使用质量"当成"用户体验好不好"这种纯主观感受，它其实是围绕任务效果、效率、

满意度、风险的可观察指标。②功能点与非功能特性容易写反：需求评审数功能，设计评审谈非功能。③这份作业的内容与期末课程设计里的"测试需求分析"直接相关，别当形式作业交。

过渡：作业之外，课件还留了三道思考题，这三道题就是本章的考点清单，我们把它讲透。

【2.85】思考题（4 分钟）

这一页的三道思考题，请大家一定当成复习提纲来用，因为本章期末要考的点基本都在里面。我把参考答案一题一题讲。

第一题：为什么说缺陷是质量的对立面？先明确质量的定义——质量是软件满足规定需求和隐含需求、也就是满足用户期望的程度。而缺陷恰恰是违背需求与期望、导致结果不正确的问题。所以两者是同一个坐标轴上的两端：质量是"满足期望"的度量，缺陷是"违背期望"的实证。缺陷越多、越严重，质量就越低；反过来，想让质量高，最直接的动作就是减少缺陷，尤其是减少严重缺陷。答题时一定要走三步：先给质量下定义，再讲缺陷是"违背"，最后落到"数量与严重程度共同决定质量水平"，三步缺一不可。

第二题：为什么静态测试和动态测试是一对对立统一体？"对立"讲的是手段：静态测试不运行程序，靠阅读、评审、检查文档和代码来发现问题；动态测试必须运行程序，通过输入数据、观察输出来发现问题。一个不动、一个要动，这是手段上的对立。"统一"讲的是目标和互补关系：两者的目标完全一致，都是发现缺陷、评价质量；而且它们能覆盖对方覆盖不到的地方——静态测试擅长抓需求里的二义性、设计上的缺陷、编码规范与逻辑问题，这些问题程序跑起来未必暴露；动态测试擅长验证运行时的真实行为，比如计算错误、性能瓶颈、并发问题，这些问题光看代码很难断定。所以标准答案是：手段对立，目标统一，功能互补，缺一不可。

第三题：为什么说测试需求分析是测试计划、测试设计的基础？一句话，测试需求分析回答的是"测什么"——测试项有哪些、范围到哪里、优先级怎么排、可测性如何。这些答案一旦有了，测试计划才有依据：范围定了才能估工作量和资源，优先级定了才能排进度、安排人力，风险点清楚了才能定应对措施。测试设计同样依赖它：只有明确了测试项和它的可测性，才能决定用哪些方法、覆盖到什么程度、判定准则是什么。反过来想，如果跳过测试需求分析直接写计划，就会出现"计划里排满了场次，却不知道要测什么"；直接写用例，就会出现"用例写得漂亮，却漏了最重要的业务场景"。这正是后面"测试需求分析与测试计划"那一章要专门解决的问题。

三道题串起来其实就是本章的主线：先有质量与缺陷的概念，再有静态与动态两类手段，最后用测试需求分

析把概念接进工程流程。

> **板书：**①质量＝满足规定需求＋隐含需求（期望）；缺陷＝违背需求与期望 → 同一坐标轴两端，缺陷的数量与严重程度决定质量水平

②静态（不运行程序：阅读、评审、检查；抓需求二义性、设计缺陷、规范问题）↔ 动态（运行程序：抓计算错误、性能、并发）→ 手段对立、目标统一、功能互补

③测试需求分析＝回答"测什么"（测试项、范围、优先级、可测性）→ 计划据此定范围、资源、进度、风险；设计据此定方法、覆盖与判定准则

提问：如果只允许你做静态测试或者只允许你做动态测试，你选哪一个？为什么？

> **预设回答：**①"选动态，能跑起来才算真测试"——要纠正：跑起来之前，需求二义性和设计缺陷已经在里面了，动态测试往往只能看到症状，说不出根因。②"选静态，成本低"——要纠正：不运行就永远无法确认运行时行为和系统级质量。③"两个都不能只选一个"——正确，正好回扣"对立统一"。

易错点 / 考点：①把静态测试等同于"看代码"——不完整，需求文档、设计文档、测试用例甚至用户手册的评审都属于静态测试。②只答"一个运行一个不运行"而漏掉"互补"，最多得一半分。③判断题高频：静态测试也能发现缺陷（对）；缺陷是内部错误、失效是外部表现（对）；测试需求分析在测试计划之后进行（错，必须在前）。口诀：**对立看手段，统一看目标，基础看"测什么"。**

> **过渡：**三道思考题之后，本章还留了一个动手实验，我们把它的要求说清楚。

【2.86】实验：完成一个简单的测试过程（3 分钟）

这是本章的实验，名字很朴素——完成一个简单的测试过程。注意"过程"两个字，它不是让你背概念，而是让你完整走一遍"分析—执行—记录—报告"的闭环。

第一步，选被测对象。课件给了两个来源：优先用你在软件工程课或其他课程里开发过的软件系统，从里面选定一到两个功能模块；如果手上没有系统，就用课件给的两个练习站点——saucedemo 和 automationpractice。这两个都是业界经典的练习用电商站点，功能完整、又没有真实业务风险，随便点、随便试，非常适合第一次做功能测试。

第二步，先做初步的功能测试分析。课件提了两个具体问题：一是了解功能操作的路径，也就是从哪进、点

几步能到；二是明确要输入哪些数据，以及有哪些特殊、异常的数据或操作。这一步就是"设计"，不要一上来就乱点，先把入口、路径、输入项写清楚。比如登录功能，你要问：输入框接受什么格式？空值、超长值、错误密码、特殊字符算不算异常？能不能只用键盘 Tab 和回车走完全流程？

第三步，执行手工测试。课件的要求很形象：像用户使用产品那样操作软件，进行手工测试，发现缺陷并记录。请务必做到"操作有路径、缺陷有步骤、现象有截图"。记录一条缺陷至少要有四样东西：怎么操作、期望什么结果、实际什么结果、能不能稳定复现。

第四步，完成一个非规范的测试报告。这里的"非规范"我要特别强调：它的意思是第一次不要求你套正式模板、不要求完整的测试计划与评审流程，而不是"可以随便写"。报告里至少要有被测对象与版本、测试的功能模块、用了哪些数据和异常输入、发现的缺陷清单，以及你自己的一句结论。

具体细节见教材第 40 到 41 页。这个实验是后面课程设计的基础动作，工具上先不做硬性要求，先把"发现缺陷并把它说清楚"这件事做到位。

板书：①选对象：自研系统 1–2 个功能模块／saucedemo、automationpractice ② 初步分析：操作路径＋输入数据（含特殊、异常） ③手工测试：像用户一样用，发现并记录缺陷 ④非规范报告（非规范 ≠ 随便写）｜详见教材 P40–P41｜一条缺陷四要素：操作步骤、期望结果、实际结果、可复现性

提问：在登录功能里，你会设计哪些"特殊、异常"的数据或操作？

预设回答：①"输入错误的密码"——对，但太基础，继续追问：空用户名、超长字符、前后空格、大小写、SQL 特殊字符呢？②"连续点五次登录按钮"——好例子，属于异常操作，考验防重复提交。③"点登录后再按浏览器后退，看状态对不对"——很好的思路，这是流程与状态类的异常路径。教师点评：特殊与异常输入通常从"格式、长度、边界、顺序、频率、状态"六个方向去想。

易错点 / 考点：①把"非规范报告"理解成可以只写结论——报告必须有证据链。②缺陷记录只写"这里有 bug"——工程和考试都要求给出复现步骤与期望/实际的对比。③容易漏掉"分析"这一步直接开测，导致测试没有针对性；而这一步恰恰是本章测试设计意识落到手上的地方。

过渡：这节课的内容就到这里，最后我们用一句话帮大家收个尾。

【2.87】感谢聆听（1 分钟）

好，以上就是本节课的内容。这一章讲到这里，我们把软件测试的基本概念过了一遍：缺陷是质量的对立面，

缺陷是内部错误、在外部表现为失效；测试方式有静态与动态、主动与被动、手工与自动化、基于脚本与探索式；测试级别有单元、集成、系统、验收；工作范畴从测试需求分析、测试策略、测试计划、测试设计、测试执行，一直到测试结果和过程评估。请回去把本章小结那张全景图和三道思考题结合起来复习，作业和实验按要求提交。

我是广东金融学院的周宇文，课件和资料会发到课程群，有问题随时找我。感谢聆听，下次课我们进入新的内容。

板书：质量与缺陷 → 方式（静态/动态、主动/被动、手工/自动化、脚本/探索）→ 级别（单元/集成/系统/验收）→ 工作范畴（需求分析→策略→计划→设计→执行→评估）｜下次课预告 · 作业与实验提交提醒 · 联系方式

提问：请用一句话说出本章你最有收获的一个概念。

预设回答：①"缺陷是内部错误，失效是外部表现"——能把这两层分开，说明因果链理解了。②"静态测试不运行程序、动态测试运行程序，两者互补"——分类思维到位。③"测试需求分析必须先于测试计划"——工程顺序意识有了，这正是后面课程设计的起点。

易错点 / 考点：收尾前再点一遍本章三个高频错点——探索式测试不等于 Ad-hoc 乱测；"软件测试方式"那一条列出的其实是测试级别；测试需求分析在计划与设计之前，顺序不能颠倒。

过渡：第 2 章到此结束，下次课我们进入新的一章，开始讲具体的测试方法与技术。

四、思政落点清单（共 16 处）

序号	所在页	落点主题
1	2.5	讲一个真实的工作场景：某团队上线后收到大量用户投诉，工程师在周报里写"功能符合需
2	2.13	讲一个课堂可讨论的情境：同一次测试里，你发现了三个缺陷，其中两个是界面小瑕疵，一
1	2.24	讲什么故事：霍珀团队把一只飞蛾贴进日志本并写明"这是第一个实际发现的 bug"。怎
1	2.28	讲"判断缺陷必须有依据"，可以联系工程中的标准意识：需求文档、设计规范、行业标准
2	2.30	缺陷构成分析里有一个职业底线问题：不能只统计"好修好说"的缺陷，把难看的、自己漏

3	2.32	成本曲线讲的是"欠下的质量债早晚要还"，这正好对应职业责任：明知道某个功能没有验
1	2.46	请记住评审席上的两条底线。第一条叫"对事不对人"：我们用证据和事实说话，指出的是
2	2.52	这一页其实藏着一个很硬的职业态度：不要满足于"文档上通过了"。规格说明书也是人写
1	2.56	同一段代码，用黑盒视角和白盒视角去测，能发现完全不同的缺陷——这正是科学精神里"
2	2.58	分层背后是一种"把大问题拆开、逐层把关"的工程思维，也是工匠精神里"每道工序都有
3	2.61	代码评审能发现 50%～70%缺陷这个数据，正好用来说明"同行评审不是挑刺，而是互
4	2.64	持续集成这条流水线最考验的其实是诚信和责任感——测试红了却直接合入、失败用例被注
1	2.68	讲什么：定义里那句"如同用户所合理期待的那样"。怎么联系知识点：验收测试要管隐含
2	2.76	讲什么：用例库是团队一点一点积累起来的知识资产。怎么联系知识点：正因为良好用例会
3	2.77	讲什么：执行时发现了一个对自己不利的结果——是如实记录，还是轻轻放过。怎么联系知
1	2.83	科学精神・质量观・诚信。质量模型和缺陷分类不是背概念，而是在训练一种"凡事讲证据