《软件测试课程设计》讲稿（第 1 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 1 章　引论

第 1 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 1 章 讲稿（第 1 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 1 章 引论 —— 为什么必须做软件测试?
>
> 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 61 页（`lecture1/`，云端 http://111.231.0.226:2345/lecture1/index.html）
>
> 本讲重点：软件缺陷的代价与质量成本观、软件测试的必要性（真实事故案例）、软件测试的定义与学科形成
>
> 本讲难点：区分测试/调试/质量保证三者的边界；理解测试左移与成本非线性增长
>
> **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 能说出软件测试的定义（在指定条件下运行或评审系统与组件、观察记录结果、对照预期作出评价）；② 能列举至少 5 个真实缺陷事故并归纳其教训；③ 能说清测试与 SQA、与开发的关系，以及测试左移的含义；④ 能复述 TDD"测试在前、开发在后"的基本思想
能力	① 能用"是否运行程序"判断静态测试与动态测试；② 能区分测试、调试、质量保证三者的边界；③ 能针对给定事故案例指出缺失了哪一类测试（集成/兼容/性能/安全等）
素质（思政）	① 通过 Therac-25、波音星际客机、骑士资本等事故建立"质量即责任"的职业底线；② 通过《血狮》与国产软件案例理解工程诚信与工匠精神；③ 通过测试左移与缺陷成本曲线体会严谨求实、精益求精

本讲要建立的一个总印象：软件缺陷客观存在且代价高昂，**测试是发布前的质检关卡**，也是测试人员的职业责任。

二、时间分配总表

> ⚠ **容量提示**：本讲稿为**素材全量版**，口播合计约 3 万字量级（≈2 小时纯朗读）。90 分钟课堂扣除互动、练习、课间约 20 分钟后，纯讲授容量约 70 分钟。
>
> **请按以下方式取用**：① **重点页全文慢讲（2–3 分钟/页）**：1.11、1.13、1.14、1.15、1.17、1.23、1.31、1.32、1.35、1.37、1.49、1.51、1.54；② **常规页快速带过（0.5–1 分钟/页）**：只讲口播第一段＋板书＋1 个提问；③ **可课后阅读页**：★1.5、★1.6、★1.7、★1.8、★1.9、★1.60。

时段	页码	内容	分钟	备注
第 1 节	1.1–1.10	封面、课程导学、教材与三篇结构	10	讲清考核口径与课程设计任务
第 1	1.11–	案例导入：真实缺陷事故（迪士尼/Intel/火星/波音/骑士资本	16	1.13–1.15、1.17 重点

节	1.21	/AWS/Uber 等）		讲
第 1 节	1.22–1.34	1.2 为什么要测试；1.3.1 学科形成；1.3.2 正反思维	14	1.23、1.31、1.32 是考点
——	——	**课间休息 10 分钟**	——	——
第 2 节	1.35–1.45	1.3.3 定义（IEEE/ISO 29119）；1.3.4 其它视角（质量/风险/经济/Oracle/批判性思维）	16	1.35、1.37 必讲
第 2 节	1.46–1.52	1.4 测试与 SQA；1.5 测试与开发（并行协作、测试左移）	12	1.49、1.51 是难点
第 2 节	1.53–1.61	1.6 TDD 思想；问题、小结、思考与练习、学习资源、结束	12	1.54、1.57、1.59 必做

> **压缩优先级（时间不够按序压）**：★1.5–1.9（课程导学细节）→ ★1.60（学习资源改为课后阅读）→ 1.3–1.4（就业市场与敏捷时代合并为一句带过）→ 1.40–1.45（其它视角只讲风险与经济两个）。**1.13–1.15、1.17、1.23、1.31、1.32、1.35、1.37、1.49、1.51、1.54 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。

【1.1】软件测试方法和技术（2 分钟）

同学们好，欢迎来到《软件测试课程设计》。我是这门课的任课老师周宇文。这门课的教材是朱少民老师主编、清华大学出版社 2022 年出版的《软件测试方法和技术（第 4 版）》。先交个底：这是专业选修课，32 学时、2 学分，理论 16 学时加实验 16 学时，一半时间讲原理，一半时间在机房动手。考核方式是考查，看平时的过程性表现，加上最后的课程设计作品与答辩。今天先不急着学工具，我们回答一个最根本的问题——软件为什么要测？不测会怎样？把这个问题想明白，后面所有方法才有落脚点。

> **板书：**课程名 → 教材＝朱少民《软件测试方法和技术（第 4 版）》清华 2022 → 32 学时（理论 16＋实验 16）→ 考查（过程性＋作品答辩）

提问：你们觉得，一个软件完全不测试就上线，最坏会发生什么？

预设回答：①"大不了不好用，用户骂两句"——这是最典型的低估，我会追问：如果它管的是钱、是电、是医疗设备呢？②"会崩溃、会赔钱"——方向对但太笼统，要求他说清"哪一类损失、谁来承担"。③"现在都敏捷了，边开发边测，不用专门测"——这句话先记在黑板上，讲到敏捷那一页再回来算账。

易错点 / 考点：最常见的错，是把软件测试等同于"点点鼠标看有没有报错"。记三句话：测试是有理论、有方法、有流程的学科；测试的目标是发现缺陷、提供质量信息，不是证明软件是对的；测试不是开发之后的收尾工序。判断三步法：一问测什么（对象与范围），二问谁来测、何时测（主体与时机），三问测到什么程度算完（准则）。

过渡：坐标定下来了。接下来看一张图，它会告诉我们：软件测试这座山，到底是什么形状。

【1.2】软件测试全景图（3 分钟）

这张图叫"软件测试全景图"。它不是让大家背的，而是要在脑子里建一张地图。所谓全景图，就是把和软件测试有关的东西按几个维度铺开。横着看，是软件的生命周期：需求、设计、编码、集成、部署、运维，测试在每个阶段都有自己的活儿。竖着看，是测试的不同层次：单元测试、集成测试、系统测试、验收测试。再往里看，是测试的类型与方法：功能、性能、安全、兼容性、易用性，以及黑盒方法、白盒方法。最后还有一列是支撑体系：测试流程与规范、测试管理、缺陷跟踪、自动化工具。为什么先看这张图？因为新手最常犯的错，是手里只有一把锤子，就把所有东西都当钉子——只学一种方法，什么问题都拿它去套。有了全景图，你才知道自己站在哪儿，手上这把工具适合解决哪一类问题。打个比方，这就像去医院：全景图告诉你有内科、外科、影像科、检验科，你挂号之前得先知道自己大概是什么毛病。软件也一样，性能问题拿功能用例去测，是永远测不出来的。

板书：画一张四层图 → ①生命周期（需求→设计→编码→集成→部署→运维）②测试层次（单元→集成→系统→验收）③类型方法（功能/性能/安全/兼容/易用＋黑盒/白盒）④支撑（流程规范→管理→缺陷跟踪→自动化工具）

提问：如果用户投诉"这个 App 一到晚上八点就卡"，按全景图，你该去哪个格子找工具？

预设回答：①"找功能测试"——错，功能测试看的是"做得对不对"，不看"快不快"，我会引导他往性能格子走；②"找性能测试"——对，但要追问：是响应时间、吞吐量还是并发用户数的问题？③还有同学会说"先看看是不是网络问题"——这个思路很好，说明他有了分层排查的意识，可以表扬并补一句：先用监控确认瓶

颈在客户端、网络还是服务端，再选工具。

易错点 / 考点：容易混的是"层次"和"类型"——层次回答"在哪一级测"，类型回答"测什么属性"。判断题常考"性能测试属于系统测试阶段"这类说法，要按"层次＋类型"两个坐标分别作答。

过渡：地图看完了，我们把镜头拉近：这张地图上的那些词，彼此是什么关系？

【1.3】系统地理解软件测试（4 分钟）

这一页请大家跟着我一起看，标题叫"系统地理解软件测试"。图上有二十来个词：软件、质量、缺陷、测试、目标、测试用例、方法、设计、实施、发现、清除、阶段、管理、思想、源泉、确定、寻求、指导。看着散，其实是一条完整的逻辑链。第一，测试的对象是"软件"，追求的是"质量"，所以质量是整套活动的目标；没有质量这个目标，测试就变成"为测而测"。第二，质量的敌人是"缺陷"，缺陷就是测试直接要对付的东西——做测试，本质上就是找缺陷、报缺陷、推动清除。第三，"源泉"和"确定"回答的是测试从哪里开始：测试的源泉是需求与用户期望，需求不确定，测试就无从下手，所以测试要尽早介入，先把"测什么"确定下来。第四，"寻求""设计""方法""测试用例"是一条技术链：先寻求怎么测，再设计测试用例，再选合适的方法，用例是这件工作的核心产物。第五，"实施""发现""清除"是执行链：把用例跑起来，发现缺陷，推动清除。第六，"阶段""管理""思想""指导"是保障链：测试要分阶段推进，要靠管理组织资源、进度和缺陷，还要靠测试思想来指导决策。再打个比方，这整张图就像看病：目的是健康，也就是质量；症状是发烧，也就是缺陷；你不能看一眼就开药，得先问诊、验血、拍片，这就是分析与用例设计；确诊后治疗是清除缺陷，康复靠作息管理就是阶段与管理。放到软件行业里，拿一个登录功能说：需求写着"手机号加验证码可以登录"，这是源泉；用例要覆盖正常登录、验证码错误、账号被锁、网络中断、重复点击，这是设计；每个异常都要记成缺陷单、跟踪到关闭，这是管理与执行。所以一句话口诀：目标管质量，对象抓缺陷，源泉看需求，核心是用例，落地靠执行，保障靠管理和思想。

板书：质量（目标）→ 缺陷（对象）→ 需求（源泉）→ 用例（核心）→ 方法/设计（怎么测）→ 实施/发现/清除（执行）→ 阶段/管理/思想（保障）

提问：请用一句话说说，"缺陷"和"测试用例"是什么关系？

预设回答：①"用例是用来找缺陷的"——对，但不够深，我会补充：用例是把"希望发现的缺陷"提前翻译成可执行步骤；②"用例就是操作步骤"——典型的窄化，要提醒用例还必须写明前置条件、输入数据、预期结果和判定准则；③有同学会说"没有预期结果就不算用例"——这句很到位，可以顺势强调：没有预期结果，就无法判断实际结果是"对"还是"错"。

易错点 / 考点：这一页最爱考"关系"而不是"定义"。记住一条主线：需求是源泉，用例是核心，缺陷是对象，质量是目标。选择题常把"测试的目的是证明软件没有缺陷"当干扰项，这句话是错的——测试只能证明缺陷存在，不能证明缺陷不存在。

【思政落点 1】：这二十来个词里，"清除"两个字最见职业底线。找缺陷是技术活，报缺陷是良心活。行业里有真实教训：发现缺陷却因为怕影响绩效、怕得罪人而选择隐瞒，最后问题流到用户手上，代价由公众承担。请同学们记住——测试人员手里握着的是别人看不见的风险，如实记录、不隐去不利结果，是这个岗位的第一条职业底线，也是一名工程师对国家、对社会负责的起点。

过渡：原理讲完了，大家可能更关心一个现实问题：学这个，将来干什么？我们看两张招聘市场的图。

【1.4】从就业市场看软件测试（3 分钟）

这一页是两张图，都来自招聘市场。第一张是岗位与需求分布，第二张是岗位要求里的技能关键词。据课件数据，招聘平台上与软件测试相关的岗位长期保持稳定需求，测试工程师、自动化测试工程师、测试开发工程师是三类最常见的职位名称。请大家重点看第二张图里的技能词：功能测试、用例设计、缺陷管理、数据库、接口测试、性能测试、自动化脚本、Linux、Python 或 Java。有没有发现？这些词跟咱们这门课的大纲几乎一一对应——用例设计、缺陷管理、接口测试、性能测试、自动化工具，后面都会讲到。为什么要先看就业？因为学习最怕"不知道学了有什么用"。这张图告诉大家：测试不是"点鼠标的岗位"，而是一个有明确能力栈的岗位。招聘方要的不是"你会不会用某个工具"，而是"你能不能分析需求、设计用例、定位问题、说清缺陷并推动它被修掉"。把这张图记在心里，以后每学一个方法，都问自己一句：这一条能不能写进简历，能不能在面试里讲出一个具体例子？

板书：三类岗位（测试工程师／自动化测试／测试开发）→ 技能栈：功能测试・用例设计・缺陷管理・数据库・接口测试・性能测试・自动化脚本・Linux・Python/Java

提问：招聘要求里为什么不直接写"会用某某工具"，而写"用例设计能力"？

预设回答：①"工具学得快，能力难练"——很准，工具两星期能上手，设计思维要一学期；②"工具会过时"——也对，方向一致；③"因为公司用的工具各不相同"——这是最实在的回答，正好引出结论：工具是载体，方法是本事；方法迁移得走，工具只是脚手架。

易错点 / 考点：别把"软件测试"理解成"软件测试工具"。考试如果问"测试工程师最核心的能力是什么"，落在"测试分析与用例设计"上，而不是某个工具的操作步骤。

过渡：既然岗位在变，那今天的行业跟十年前比，到底变了什么？我们看敏捷时代这一页。

【1.5】在敏捷时代，如何看待软件测试？（3 分钟）

这一页标题是个问题：敏捷时代，怎么看待软件测试？课件给了五条判断，我逐条讲。第一条，敏捷开发模式提倡整个团队对质量、对测试负责。注意这句话的分量——测试不再是"测试组的事"，产品、开发、测试都得为质量签字。第二条，开发与测试越来越融合，比如课件提到的微软测试人员转型：过去微软有庞大的专职测试岗，后来大量测试岗位被并入开发角色，测试能力变成开发者的基本功。第三条，持续交付倒逼持续测试，但测试最容易成为瓶颈。为什么？因为代码一天能集成十次，自动化测试要是跑一次要两个小时，交付就被卡住了。这就像高速公路：路修得再宽，收费站只有一个窗口，车照样堵。第四条，测试左移、测试前移，让开发做更多的测试——把缺陷拦在需求和设计阶段，越早越便宜。第五条，测试驱动开发，英文缩写 TDD：测试在前、开发在后，先写测试再写实现。这里再补一个生活化的类比：敏捷时代的质量，就像一支球队的防守——不是后卫一个人的事，前锋也要回防，教练也要布阵；如果大家都觉得"丢球是后卫的问题"，这支球队一定丢球。软件行业里还有一个很典型的现象：很多团队每天多次把代码合并到主干，每次合并都会自动触发一轮测试，谁把测试跑红了，谁就得马上修，绝不允许红色状态过夜。这就是"持续测试"的真实样子——测试不是一道关卡，而是开发流水线上的传送带。所以课件才提醒我们：这条路最怕的就是瓶颈。一旦自动化测试跑得比开发还慢，团队就会开始"绕过测试"，而一旦绕过，质量就会以看不见的方式持续下滑。大家把这一页记成一句：质量是全员责任，测试要与开发并行，而不是排在开发后面。

板书：①全员质量责任 ②开发测试融合（微软测试转型）③持续交付→持续测试（瓶颈风险）④测试左移/前移 ⑤ TDD：测试在前、开发在后

提问：为什么说"测试最容易成为瓶颈"？如果你们团队的自动化测试要跑两个小时，你会怎么改？

预设回答：①"因为测试最慢"——太表面，我会追问慢在哪里；②"因为每次提交都要跑全量测试"——这就说到点子上了：可以分层，提交时跑冒烟与单元测试，夜间跑全量与性能；③"可以并行、可以分布式跑"——也对，属于工程手段；④有同学说"那就少测一点吧"——这是危险回答，绝不能靠减少测试来提速度，要换成"提高测试的性价比"，说清哪一层该测什么。

易错点 / 考点：TDD 的顺序一定要记准——先写测试，再写实现，最后重构；顺序反了就变成"补测试"。判断题常把"测试左移"错误理解为"把测试人员提前招进来"，左移的核心是测试活动在时间轴上向前移，测试人员更早介入需求与设计。

过渡：讲到这里，有位同学可能会想：那我这门课到底是要背概念，还是要练本事？下面两页，一页讲能力，

一页讲方法。

【1.6】如何借助这门课培养学生的分析问题和解决问题的能力？（3 分钟）

这一页的问题很直接：这门课怎么培养大家分析问题和解决问题的能力？课件给了三个抓手。第一个抓手是"强调分析能力，测试分析是基础"。请注意，测试分析是基础——拿到一个功能，先要分析它有哪些输入、哪些分支、哪些边界、哪些异常，分析不清楚，用例就只能靠拍脑袋。第二个抓手是"批判性思维，比如探索式测试"。探索式测试要求你一边学习系统、一边设计用例、一边执行、一边根据结果调整下一步——它不相信"需求写完就万事大吉"，而是主动去质疑："这里是不是有问题？"第三个抓手是"工程思维，找到不同的解决方案，从中选出最优的"。这句话是这门课的灵魂：现实中的测试问题，从来没有唯一解。时间有限、人手有限，你是全量回归，还是按风险挑重点回归？这不是聪明不聪明的问题，是工程权衡的问题。举个生活中的类比：装修房子，预算和工期都有限，你是全屋精装还是先把水电做好？会算账的人，才是好工程师。那批判性思维怎么练？就是刚才说的探索式测试，它不是"随便点点"，而是带着问题去试探系统：先花二十分钟熟悉功能，再列出一批"我怀疑会出问题"的点，比如改价格后再改数量、优惠券和积分同时用、页面刷新一半就断网，然后一边试一边记录，试完复盘的发现又变成新的怀疑。这个过程就像侦探查案——不是等案情自己送上门，而是主动去找矛盾之处。工程思维又怎么练？还是拿购物车举例：如果测试时间只够覆盖一半功能，你是按页面从头测到尾，还是先把"下单、支付、退款"这条主链路测透？成熟的答案一定是后者，因为主链路出错的代价最大。这就是风险驱动：把有限的测试资源，压在出错代价最高的地方。我们这门课的实验和作业，都会逼着大家做这种权衡，并说明你权衡的理由。

板书：分析能力（测试分析是基础）→ 批判性思维（探索式测试）→ 工程思维（多方案比较→选最优，并说明理由）

提问：同一条需求，张三设计了 8 条用例，李四设计了 30 条。谁的测试更好？

预设回答：①"李四更全面"——这是最常见的直觉，我会反问：如果上线时间只剩两小时，30 条跑得完吗？②"张三更好，因为效率高"——也未必，如果那 8 条漏掉了核心分支，效率就是假效率；③比较理想的回答是"要看覆盖了什么，而不是看数量"——这就讲到关键：用覆盖率和风险来评价，而不是用条数。顺势提醒：用例数量不是绩效，漏测才是事故。

易错点 / 考点：别把"探索式测试"当成"随便点点"。它是有方法、有记录、有章程的：先明确测试目标与时间盒，边测边记，事后复盘并沉淀为可复用的用例。简答题若问"分析能力与工程思维的区别"，答：分析能力解决"有哪些问题要测"，工程思维解决"在约束下先测哪些、怎么测最划算"。

过渡：能力怎么练出来？光靠听讲不行，得动手。下一页讲实验教学。

【1.7】如何通过实验教学培养学生的实际能力？（3 分钟）

这一页讲实验。课件第一句就说：软件测试是一门实践性很强的课程。这句话不是套话——你可以把等价类划分的定义背得滚瓜烂熟，但面对一个真实的登录页面，你还是会漏掉手机号带空格、密码含中文、验证码过期这些情况。所以这门课的方法是"做中学"：听一遍，不如自己做一遍。课件还给了两条方法：问题驱动教学与实验；实验就是分析问题、解决问题的过程。什么是问题驱动？就是先给你一个"跑不通""算不对""会崩"的现象，然后你自己去定位：是环境问题、数据问题，还是程序逻辑问题。这个过程跟将来上班是一模一样的——领导给你一个 bug 单，上面只有一句"用户反馈下不了单"，剩下的全靠你自己查。其实道理跟学游泳、学开车一样：教练讲一百遍换气和打方向盘都没用，你得自己下水、自己上路；而实验课就是你们的泳池和练车场，在这里犯错不要钱，在工作里犯错很贵。软件行业里有个很朴素的经验：同一个缺陷，在需求评审时发现，改一句话；在设计阶段发现，改一段设计；在编码阶段发现，改几行代码；上线之后才发现，就要改代码、改数据、发版本、安抚用户，甚至上门道歉——越往后越贵。所以我们做实验，练的不只是"会不会用工具"，而是在练"我能多早发现它"。我给大家一个忠告：实验课别抄步骤、别只截图。真正长本事的是三件事——第一，把现象记录清楚（什么环境、什么数据、怎么操作、出现什么）；第二，把猜想和验证过程写下来，哪怕猜错了也写；第三，把结论沉淀成可复用的用例或检查清单。这门课的平时分，很大一部分就看这三件事做得实不实。

板书：做中学 → 问题驱动（给现象，找原因）→ 实验＝分析问题＋解决问题 → 记录三件事：现象/环境/数据、猜想与验证、沉淀用例清单

提问：实验报告里只写"运行成功，结果正确"，为什么拿不到高分？

预设回答：①"因为没分析"——对；②"因为没有记录过程"——也对；③"因为不知道他到底测了什么数据"——这正是关键：没有输入数据和预期结果，这份报告无法被复现，也就无法被信任。补一句：可复现是测试报告的生命线。

易错点 / 考点：常考"缺陷报告必备要素"：环境、版本、前置条件、操作步骤、实际结果、预期结果、复现概率、附件证据。判断三步法：别人按你写的步骤能不能复现？缺陷能不能被定位？不写预期结果的报告，等于没写。

过渡：方法有了，那这门课用什么教材、学哪几块内容？我们看下一页。

【1.8】《软件测试方法和技术》（3 分钟）

介绍一下我们手里的这本教材。据课件数据，这本书 2005 年出版，现在是第 4 版，是"十一五""十二五"国家级规划教材，也是上海市普通高校优秀教材；有 300 多所大学在使用，销量超过 20 万册，并连续三年获得清华大学出版社畅销书奖。这几个数字说明什么？说明它是一本被长期验证过的教材，不是速成手册。再看结构，本书共分三篇：第一篇"软件测试的原理与方法"，第二篇"软件测试技术"，第三篇"软件测试项目实践"。这个三篇结构其实对应了我们学习的三个阶段：先懂原理（知道为什么这么测），再学技术（知道用什么工具、什么方法测），最后做项目（在真实的约束下把前面学的用起来）。所以大家在读这本书的时候，不要一上来就跳到工具章节，工具是学得最快的，原理才是决定你上限的东西。这个三篇结构，很像医学院的培养路径：第一篇是基础医学，教你人体是怎么运转的；第二篇是临床技术，教你怎么用各种仪器去查、去治；第三篇是临床实习，把你扔到真实的病人面前做判断。没有第一篇，你看到症状只会乱开药；没有第三篇，你背再多书也不敢上手术台。我们这门课同样如此：第 3 章的方法如果脱离了原理，就会变成"照着公式套用例"；而课程设计如果不动手，你也永远不知道真实的软件有多不听话。所以我的建议是，读教材的时候手里要有三支笔：第一支划概念，第二支记工具怎么用，第三支写"这个方法在什么场景下会失效"。第三支笔最重要，因为任何测试方法都有它测不到的东西，知道边界，才知道什么时候该换方法。也提醒一句：教材第 4 版跟我们这门课的课件是对齐的，课后作业里出现的概念，都能在书上找到出处，遇到不懂的先翻书，再问老师。

板书：2005 年首版 → 第 4 版（2022）→ 国家级规划教材 · 上海市优秀教材 → 300+ 所高校使用 · 销量 20 万册+ · 连续三年清华畅销书奖 → 三篇：原理与方法 | 技术 | 项目实践

提问：为什么教材要分成"原理与方法""技术""项目实践"三篇，而不是直接把工具从第一页讲到最后一页？

预设回答：①"因为要先懂原理"——对；②"因为技术会过时，原理不会"——这个回答很成熟，可以表扬；③"因为最后要做项目"——也对：知识只有在项目里被串起来才会变成能力。可以把三种回答合成一句：原理决定你会不会想，技术决定你会不会做，项目决定你能不能交付。

易错点 / 考点：注意区分"教材版本"与"课程课时"这类事实性信息，别把第 4 版记成第 3 版，也别把 32 学时记成 64 学时。客观题常考教材结构与出版信息，记住"三篇"和"2022 年第 4 版"。

过渡：三篇里我们最先进入的是第一篇。第一篇包含哪四章？看下一页。

【1.9】第 1 篇 软件测试的原理与方法（2 分钟）

这是第一篇的目录页，包含四章。第 1 章"引论"，也就是我们今天开始讲的内容，回答的是"为什么必须测、什么是测试"这些根本问题；第 2 章"软件测试的基本概念"，把测试的分类、级别、静态与动态、黑盒与白盒这些概念系统化；第 3 章"软件测试方法"，进入具体方法：等价类划分、边界值分析、判定表、因果图、路径覆盖等等；第 4 章"软件测试流程和规范"，讲测试怎么在团队里组织起来，从测试计划、测试设计到缺陷管理、测试报告。大家注意这个顺序：先讲必要性（为什么），再讲概念（是什么），再讲方法（怎么做），最后讲流程（怎么组织起来做）。这是一条从"想明白"到"做得对"再到"做得稳"的路线。很多同学做课程设计时最痛苦的不是不会用工具，而是不知道从哪儿下手，根源就是跳过了前两章。所以第一篇这四章，请务必跟着走扎实。

板书：第 1 篇 = 第 1 章 引论（为什么）→ 第 2 章 基本概念（是什么）→ 第 3 章 测试方法（怎么做）→ 第 4 章 流程和规范（怎么组织）

提问：为什么"流程和规范"要放在"方法"之后讲，而不是一开始就讲？

预设回答：①"因为要先会方法才知道流程管什么"——很到位；②"因为流程是给团队用的"——也有道理，个人可以先会方法，团队才需要流程；③"因为流程不重要，可以最后看"——要纠正：流程不是不重要，而是它依赖于方法；没有方法的流程只是一堆表格。

易错点 / 考点：常考"四章的顺序与各自主题"的对应关系。记忆口诀：一论（引论）二概（概念）三法（方法）四流程。

过渡：好，教材结构清楚了，我们正式进入第 1 章。

【1.10】软件测试方法和技术（2 分钟）

翻到这一页，我们正式进入第 1 章"引论"。这一章在整个课程里的位置，相当于盖房子的地基。它不教你某个具体的用例设计技巧，而是回答三个问题：第一，为什么必须做软件测试——用真实事故说话；第二，什么是软件测试——把它作为一个学科来定义；第三，测试和质量保证、和开发到底是什么关系。这三问答不清楚，后面学得越多越乱：你会把测试当成开发的附属品，会把质量保证和测试混为一谈，也会在执行时失去判断标准。所以接下来这一章，我会带着大家看一批真实案例，每个案例都问一句：如果当时有规范的测试，哪些问题本来可以被提前发现？请大家带着这个问题听。

板书：第 1 章 引论 → 三问：为什么必须测／什么是测试／测试与质保、开发的关系

提问：如果让你只用一个词概括第 1 章的主题，你选哪个词？

预设回答：①"定义"——只答对了一半，定义只是其中一问；②"案例"——案例是手段不是主题；③"必要性"——这个最贴切，第 1 章的主线就是回答"为什么非测不可"。

易错点 / 考点：第 1 章的考点集中在"测试的必要性""测试的定义""测试与调试、质量保证的区别"三处。注意"测试≠调试"：调试是定位并修复缺陷的开发活动，测试是发现缺陷、提供质量信息的活动。

过渡：第 1 章要讲哪六节？我们看全章目录。

【1.11】1.1 软件测试的必要性（3 分钟）

这是第 1 章的目录，一共六节：1.1 软件测试的必要性；1.2 为什么要进行软件测试；1.3 什么是软件测试；1.4 测试和质量保证的关系；1.5 测试和开发的关系；1.6 测试驱动开发的思想。请特别注意 1.1，它下面挂着一串案例：迪斯尼的多媒体游戏、一个缺陷造成数亿美元损失、火星探测飞船坠毁、软件测试走捷径导致灾难再次发生、错误指令造成骑士资本集团损失（课件数据为 4.4 亿美元）、AWS 宕机整整 4 个小时、预定的酒店住不进去露宿街头、Uber 泄漏个人隐私导致用户要求赔偿 3 亿多元，还有更多悲剧。我先把这一串名字报出来，大家不要急着记数字，先记住一件事：这些不是故事会，而是我们这门课的"病历本"。每看完一个案例，我都要问你们一句——这件事如果放在今天，用我们后面要学的方法，能不能提前发现？

板书：1.1 必要性｜1.2 为什么测｜1.3 什么是测试｜1.4 测试与质量保证｜1.5 测试与开发｜1.6 TDD → 1.1 案例串：迪斯尼・奔腾缺陷・火星飞船・捷径重演・骑士资本・AWS 宕机・酒店预订・Uber 隐私

提问：这些事故来自游戏、芯片、航天、金融、云计算、出行、酒店等完全不同的行业，它们的共同点是什么？

预设回答：①"都是软件出问题"——太笼统；②"都造成了很大损失"——对，但还要再进一步；③"共同点是：缺陷已经流到了用户端才被发现"——这个回答最好，正好点出测试的价值：把发现缺陷的时点往前移。

易错点 / 考点：容易把"缺陷"和"故障"混着用。缺陷是软件中存在的问题，故障是缺陷被触发后表现出来的异常。案例题答题模板：现象→影响范围→根因（哪一类缺陷）→如果做了哪一级测试可以拦截。

过渡：案例从哪里讲起？我们先看 1.1 这一节的开篇。

【1.12】1.1 软件测试的必要性（2 分钟）

这是 1.1 节的开篇页。按课件的说明，这一节要先列写本课时的学习要点，再依次深入讲解；如果有学习要求，

就写在要点之后。课件还给了我们讲每一节要走的六个动作：讲概念——这个概念是什么，工作中什么场景会用到；讲讲解——要胜任这个能力，需要掌握哪些知识和技术；举例子——用一个案例或一个演示帮大家真正看懂怎么用；分享经验——特别是理论和实际有差异的地方；给学习建议——告诉大家在真实学习或工作场景里可以怎么用；提供实用工具——这一节对应的工具或模板。请大家把这六个动作记成一条学习路径：概念→知识→案例→经验→建议→工具。这也是我要求你们做课程设计时的顺序：先弄明白要解决什么问题，再选方法，再动手，最后沉淀成模板。接下来的 1.1.1，我们就从第一个案例开始。

板书：六步学习路径：概念（什么场景用）→ 知识技术（要会什么）→ 案例/demo（怎么用）→ 经验分享（理论与实际的差）→ 学习建议 → 实用工具/模板

提问：为什么课件要把"经验分享——理论和实际的差异"单独列成一个动作？

预设回答：①"因为实际比理论复杂"——对，这是核心；②"因为书上写的做不到"——过于绝对，应该说"现实中要受时间、人力、环境的约束"；③"因为老师有工作经验"——也算一个原因，但重点不在老师，而在你们将来要面对的取舍。

易错点 / 考点：常考"学习方法论"类简答题，答"概念—知识—案例—经验—建议—工具"六步即可拿分；若问"理论与实际的差异体现在哪"，答：约束条件（时间、成本、环境、人手）导致的取舍与优先级排序。

过渡：好，第一个案例来了——圣诞节的迪斯尼。

【1.13】迪斯尼并不总是带来笑声（5 分钟）

这个案例的标题叫"迪斯尼并不总是带来笑声"。据课件数据，1994 年圣诞节前夕，迪斯尼公司发布了第一个面向儿童的多媒体光盘游戏"狮子王童话"。圣诞节后的第一天，客户支持部的电话就开始响个不停，不断有人咨询、抱怨：为什么游戏总是安装不成功？为什么装上了也没法正常使用？事后发现，这个游戏软件只能在少数系统上正常运行，课件把它归为一类：兼容性问题。圣诞节前发布，是典型的"节日档期"发布：发布时间定死，留给测试的窗口被压缩。买游戏的是家长，圣诞早晨孩子拆开礼物等着玩，结果装不上、进不去、没声音，家长的第一反应不是"这是兼容性缺陷"，而是"迪斯尼的东西怎么这么差"。这就是质量问题的真实代价：它不只是返工成本，更是品牌信誉的损失，而信誉的修复成本远高于当初把测试做完整的成本。兼容性问题为什么难躲？因为它本质是组合爆炸：操作系统版本、显卡与声卡驱动、光驱型号、解码器、区域语言、分辨率，每一项都可能不同。你不可能把所有组合都买齐了测，所以兼容性测试的核心不是"全都测"，而是先分析用户群，找出主流配置与高风险组合，按优先级覆盖，这正是我们后面要学的等价类划分和风险驱动测试的思想雏形。如果当年有规范的测试流程，哪些问题可能被提前发现？至少三件事：第一，需求与

设计阶段就明确"支持的运行环境清单"，并写进验收条件，不在清单里的系统就明确不支持、在包装上写清楚；第二，系统测试阶段针对主流配置矩阵做真实环境实测，而不是只在开发机上验证；第三，发布前让没参与开发的人从"拆包装"开始走一遍真实安装体验。这类缺陷放到今天的手机 App 上就是"某个机型闪退"，放到网页上就是"某个浏览器排版错乱"——本质没变。

板书：1994 圣诞前夕发布"狮子王童话"→ 节后客服电话不断（装不上/用不了）→ 根因：兼容性问题（只能在少数系统正常运行）→ 代价：用户流失＋品牌受损 → 对策：环境清单进需求 → 配置矩阵按风险覆盖 → 第三方真实安装体验

提问：如果你是迪斯尼当时的测试负责人，只有一周时间，你会怎么安排兼容性测试？

预设回答：①"把所有系统都买来测"——时间成本都不允许，我会追问：一周够吗？钱够吗？②"按市场占有率排，先测主流的三个系统"——这是很实际的答案，可以表扬，并补充还要看"用户群特征"：买儿童游戏的家长，家里电脑往往偏旧或偏杂牌；③"测一下开发机不就行了"——要立刻纠正，这正是案例翻车的根源：只在开发环境验证，等于没测兼容性；④"先做需求评审，把支持范围写清楚"——这个回答境界最高，说明他把测试往左移了，可以不测的组合就不测，但不能装作支持。

易错点 / 考点：这一页最常考"缺陷类型的判断"：安装不上、运行不了、只在少数系统正常 → 兼容性缺陷，而不是功能性缺陷。判断三步法：一，换环境问题就消失吗？是→兼容性；二，功能在支持的环境里是否按规格实现？是→不是功能缺陷；三，有没有明确的"支持环境清单"？没有→需求与文档缺陷，也要记一笔。简答题如果问"兼容性测试怎么做"，答题要点：确定支持矩阵→分析用户配置分布→识别高风险组合→按优先级与等价类覆盖→形成清单并写入发布标准。

【思政落点 2】：迪斯尼这个案例里，损失最大的是什么？是那些圣诞节早晨失望的孩子和他们父母的信任。测试人员的岗位之所以重要，是因为我们守着产品交给公众前的最后一道关。课件后面还会讲到放疗仪致死、航天器坠毁这类更沉重的案例，也会有国产游戏《血狮》的教训——把没有经过充分验证的产品当作成熟产品去卖，透支的是用户的信任，也是行业的信誉。请把"质量责任"四个字写进职业观：你签下的每一个"测试通过"，都可能关系到别人的时间、财产，甚至安全。

过渡：迪斯尼的代价是口碑，下一页我们要看的这个案例，代价直接用亿美元来计算——一个缺陷，究竟能贵到什么程度？

【1.14】一个缺陷造成了数亿美元损失（3 分钟）

我们接着往下看。屏幕上这道算式很奇怪，大家跟我一起读一遍：4195835 除以 3145727，再乘以 3145727，最后减去 4195835。学过小学数学的同学都知道，一个数先除以某个数、再乘回来，等于什么都没做，所以结果应该是 0。可是当年有一批计算机，算出来的偏偏不是 0。这就是著名的奔腾浮点除法缺陷——CPU 在做浮点除法的时候，某些特定数值的组合会算错。

大家注意，这不是小毛病。CPU 是计算机的心脏，它算错，就意味着所有跑在它上面的软件都可能算错。如果是我做作业算错一道题，最多扣两分；可如果这个 CPU 装在银行系统里、装在飞机的飞控里、装在医院的放疗设备里，那就是钱算错、飞机失控、剂量算错。

那么 Intel 是怎么发现的？不是自己测出来的，是一位数学教授在做研究的时候发现自己的计算结果对不上，一路排查才定位到 CPU。这就要命了——它说明这个问题在出厂前没有被测出来。最后的代价是什么？课件写得很清楚：Intel 公司付出很大代价，回收 CPU，造成 4 亿美元损失。这个数字是按课件给的，4 亿美元，而且请注意，这是上世纪九十年代中期的 4 亿美元。

我们打个生活里的比方：这就像一家奶粉厂，出厂检验只抽检了几罐，其余全凭信心发货，等消费者吃出问题才召回。召回的成本，远远大于出厂前把每一批都检一遍。

那这一页要带走什么？第一，缺陷的代价不是线性的，越往后越贵；第二，一个底层的小数点错误，可以放大成几亿美元的商业灾难。这也正是我们这门课反复要讲的一件事——测试不是给开发挑毛病，测试是替用户和公司把风险挡在出厂之前。

> **板书：**奔腾浮点除法缺陷 → （a÷b）×b−a ≠ 0 → 出厂前未发现 → 回收 CPU → 4 亿美元损失（据课件数据）→ 缺陷代价：越晚发现越贵

提问：如果这道算式在你自己的电脑上算出来不是 0，你判断问题出在软件、硬件还是编译器？为什么？

> **预设回答：**① 多数同学第一反应是"软件 BUG"——点评：思路可以理解，但这次错在 CPU 的浮点运算单元。这说明缺陷不只藏在代码里，硬件、固件、编译器都可能带缺陷，我们讲的"被测对象"边界比想象中宽。② 有同学说"编译器优化错了"——很好，这可以追问他怎么验证：如果是编译器问题，换一个编译器结果就该对；而当年的现象是换语言、换程序都错，所以定位指向硬件。③ 有同学说"不知道从哪下手"——正好给他一个判断三步法：先看能否稳定复现 → 再看换软件环境是否还错 → 最后把范围缩到硬件或固件。

易错点 / 考点：一是数字要记清，4 亿美元是"回收 CPU 造成的损失"，不是"赔偿用户的金额"。二是概念，

本案例对应的是浮点计算（精度/计算类）缺陷。三是常考结论："缺陷发现得越晚，修复代价越大"，给一个"早期改一行代码 vs 上线后召回"的情境让你判断成本高低。口诀：**先除后乘，回不到原点就是缺陷。**

【思政落点 1】这个案例最扎心的地方在于：问题不是测不出来，而是一开始选择了"先发货，再说"。企业算的是召回成本，用户承担的是算错的后果。要让学生记住一条职业底线——测试人员手里拿的不是一张验收单，而是用户的钱包、健康甚至生命。将来到了企业做测试，如实记录、如实上报是最基本的操守：发现缺陷不报，比没发现更严重。

过渡：奔腾的缺陷好歹只是算错数。接下来这一页，代价从"算错钱"变成了"船毁人亡"——我们看火星探测飞船。

【1.15】火星探测飞船坠毁（3 分钟）

上一页的损失是钱，这一页的损失是人命和几十亿的投入。课件讲的是"火星探测飞船坠毁"。课件给出的原因链是这样的：飞船上的着地开关，本应该在真正触地的时候才被触发，可是机械震动在大多数情况下也会触发这个开关，于是就设置了一个错误的数据位。设想一下，飞船开始着陆的时候，计算机看到这个错误的数据位，极有可能判断"我已经落地了"，于是关闭推进器。而实际上飞船还在半空中，距离地面还有 1800 米。没有反推进器的帮助，飞船从 1800 米的高度冲向地面，必然会撞成碎片。这里的"着地开关""错误的数据位""1800 米"都是课件原文给的数字和描述。

但这一页真正值得我们记住的，是课件的最后一句话："两个小组本身的工作都没什么问题，就是没有合在一起测试，其接口没有被测，而问题就在这里。"课件把这一页的缺陷类型标成四个字——集成测试不足。这才是这一页的教学落点，飞船只是代价。

什么叫做集成？我打一个比方：一个乐队，鼓手自己练得很好，小提琴手自己也练得很好，可是从来没有合排过一次。上台那天，各弹各的，节奏对不上，曲子就崩了。软件也一样：A 小组写的模块对外输出的是"米"，B 小组接收的时候以为是"英尺"，各自单元测试全绿，一合起来就出事。

这就是单元测试和集成测试的区别：单元测试回答"我这个零件好不好"，集成测试回答"这些零件接在一起转不转得动"。很多同学写程序，测试就是把自己那几个函数跑一遍，这叫单元测试；真正的风险往往藏在接口、参数、时序、数据格式这些"接缝"上。所以记一个判断三步法：一看模块本身对不对，二看接口约定一致不一致，三看连起来跑通没跑通——第三步不做，前面两步都是自娱自乐。

板书：火星探测飞船坠毁 → 着地开关被机械震动误触发 → 写入错误数据位 → 计算机误判"已着陆"、关闭

推进器 → 从 1800 米坠毁（据课件数据）→ 病根：**集成测试不足（接口没测）**

提问：两个小组各自测试都通过了，为什么合起来还会崩？请举一个你在做课程作业时遇到过的"接口对不上"的例子。

预设回答：① "他们没沟通好参数含义"——对，这就是接口约定问题。追问一句：这个约定该在什么时候定下来？答：设计阶段就该冻结，测试用例要围绕接口来设计。② "这不是测试的问题，是设计的问题"——这是很好的回答，要点评：缺陷的根因可能出在设计，但拦截它靠的是集成测试，测试也要前移去评审设计。③ 典型的错误回答："只要每个模块都测好了，系统就不会有问题"——这是最高频的错误认知，模块正确推不出系统正确。

易错点 / 考点：最易混的是单元测试与集成测试。"每个模块都正确"推不出"系统正确"，接口未测是集成测试缺位的典型症状。考试常给"分工开发、单独测试全通过、联调失败"的情境，问缺了哪个级别的测试——答集成测试。口诀：**零件都好，装起来未必好；接口不测，等于没测。**

过渡：如果说火星飞船是"没合起来测"，那下一页波音的故事更值得琢磨——它不是没时间测，而是主动把测试拆碎了，走了捷径。

【1.16】软件测试走了捷径导致灾难再次发生（3 分钟）

这一页的标题很刺眼——"软件测试走了捷径导致灾难再次发生"。主角是波音公司的载人飞船"星际客机"。课件的原文是：波音公司测试载人飞船星际客机软件系统的程序存在严重缺陷，现在计划对测试程序进行修改；造成程序存在严重缺陷的主要原因就是软件测试走了捷径——该公司缩短了对该飞行器软件的一次关键测试，他们把整个飞行过程分成了几个小单元分别进行测试，但最后却没有做完整的、端到端的集成测试，也就是没有进行时长为 25 个小时的整体测试。这里的"25 个小时"是课件给的数字，注意它是那一次整体测试的时长，别记成飞行时长。

请特别注意"缩短"这两个字。它不是"忘了测"，而是"时间不够，先把关键测试砍一段"。这在工程上非常常见：进度一紧，第一个被砍的往往是测试；测试里第一个被砍的，往往是那个最花时间、最不"出彩"的端到端整体测试。而且理由听起来还挺合理——"每个小单元我们都测过了呀"。可是把所有小单元各测一遍，加起来不等于整体跑一遍。就像一部电影，每个镜头单独看都没问题，但没有人把整部片子连着放一遍，剪出来的节奏、音画同步、字幕时间轴，照样能出大问题。

课件在这一页同样标了四个字：集成测试不足。上一页是"没合起来测"，这一页是"合不起来测"。上一页偏

组织之间的接口没测，这一页偏过程与过程之间的衔接没测——分段测试覆盖的只是每一段内部的逻辑，段与段之间的状态传递、时序、异常恢复，全都没人管。

再往深一层想，这是一个决策问题，不是技术问题。"走捷径"是人在进度压力下做出的取舍。所以课件用了很重的说法："灾难再次发生"。火星飞船的教训早就写进教科书了，为什么还会再来一次？因为每一个项目都有它自己的"这次不一样"。而测试人员的专业价值，恰恰就是在大家都说"这次来不及了"的时候，把风险讲清楚、把取舍记录清楚。

板书：波音 星际客机 → 关键测试被"缩短" → 把飞行过程拆成小单元分别测 → 未做端到端集成测试 → 缺失：25 小时整体测试（据课件数据）→ 本质：进度压力下的决策失误 → 集成测试不足

提问：假设你是这个项目的测试负责人，只剩三天，团队说"整体测试要 25 小时，来不及了，能不能先跳过"，你怎么回答？

预设回答：① "必须测，风险太大"——态度对，但光表态不够。追问：你怎么让项目经理接受？要能说出"不测的后果"和"替代方案"，比如先跑关键路径的短时冒烟版本。② "听项目经理的，出了事他负责"——这是典型的错误回答：测试人员要把风险如实呈现、把决策记录在案，而不是把责任一推了之。③ "只测最关键的几段"——这是比较专业的回答：范围可以裁剪，但必须显式声明裁掉了什么、风险有多大、由谁签字确认。

易错点 / 考点：常考判断："为赶进度缩短测试时间属于合理的进度管理"——错；正确的说法是"在风险可控、显式记录的前提下裁剪测试范围"。第二个混淆点是把"25 小时整体测试"记成飞行时长。口诀：**分段测完 ≠ 端到端跑通；砍测试不能砍接口。**

【思政落点 2】波音这个案例讲的不是技术不够，而是把质量控制让位给了进度。可以让学生代入一个具体处境：如果你是那个被要求"先跳过整体测试"的工程师，你会怎么做？正确的做法既不是硬顶，也不是随大流，而是把风险讲清楚、把替代方案给出来、把决定留痕。这就是工程人员的质量责任——**按时交付很重要，但交付一个自己知道有问题、却没说出来的东西，不可原谅。**

过渡：上面两个案例的根子都在"没合起来测"。下一页换一种味道——代码没写错、测试也没少做，只是一条指令发错了，45 分钟，几亿美元。

【1.17】错误指令造成骑士资本集团损失 4.4 亿美元（3 分钟）

这一页的主角是美国的骑士资本集团。课件给的时间线非常精确：2012 年 8 月 1 日上午 9 点，纽约证券交易

所开盘交易，骑士资本的第一位散户投资者发出了买卖其投资头寸的指令；仅仅 45 分钟后，骑士资本的服务器就执行了 400 万笔交易，使公司损失了 4.6 亿美元，濒临破产。45 分钟、400 万笔、4.6 亿美元，这三个数字是这一页的核心。另外提醒一句：这一页的标题写的是"4.4 亿美元"，正文写的是"4.6 亿美元"，两个数字不一致，我们以正文的 4.6 亿为准，后面讲这个案例时也统一按正文口径。

45 分钟能发生什么？我们平时点个外卖，45 分钟刚够送达。可是在一家做高频交易的机构那里，45 分钟足够让服务器把几亿美元打进市场，让一家公司走到破产边缘。这就是软件缺陷在金融领域的放大效应：它的运行速度不是人按鼠标的速度，而是机器每秒成千上万笔的速度，缺陷一旦上线，损失是按秒计的。

课件给这一页标的缺陷类型是"功能容错性问题"。什么叫容错？通俗说，就是"出了意外，系统还能不能兜住"。这一页的起因描述是"错误指令"——也就是说，系统面对一条非预期的、异常的指令时，没有拒绝、没有拦截、没有熔断，而是照单执行，还执行了 400 万次。生活里的类比很好找：家里的电路如果只有一个总开关、没有保险丝，短路一次就可能烧掉整栋楼。保险丝平时一点用都没有，出事那一秒，它替你保住了整栋楼。

所以这一页要带走两个词：**容错**和**止损**。容错是让系统在异常输入、异常状态下不崩溃、不放大错误；止损是万一真的错了，能不能快速停下来。做测试的时候，我们不仅要测"正常流程走得通吗"，更要测"给它异常输入，它会怎么样"——这就是后面要讲的异常测试、边界测试、失效恢复测试的来由。

板书：骑士资本 2012-08-01 09:00 开盘 → 45 分钟 → 400 万笔交易 → 4.6 亿美元（据课件数据；标题写 4.4 亿，以正文为准）→ 濒临破产 → 缺陷类型：功能容错性 → 关键词：熔断 · 止损 · 异常输入

提问："错误指令"是操作者发出来的，为什么这笔账要算在软件缺陷头上？

预设回答：① "因为软件应该能识别出这是错误指令"——对，这正是容错的核心：软件不能假设输入永远正确。② "是客户操作失误，跟软件无关"——要纠正：操作失误是常态而不是例外，软件设计的前提就是"人会犯错"，所以要有校验、限额、二次确认和熔断。③ "不知道算谁的责任"——顺势引导学生区分两件事：责任可能在人，但缺陷是系统缺少对异常的保护，二者不能相互抵消。

易错点 / 考点：一是数字，45 分钟、400 万笔、4.6 亿美元，并注意标题与正文不一致。二是概念，容错不等于可靠：可靠性讲"不出故障"，容错讲"出了故障也不造成严重后果"。三是案例题常问"这个案例暴露了哪类测试缺失"，答案指向异常/容错测试以及上线前的风控验证。口诀：**正常流程测不出胆子，异常输入才见真章。**

【思政落点 3】骑士资本这类案例里，真正的风险往往不是"没人知道有毛病"，而是"有人知道了但心存侥幸，

觉得不会那么巧"。测试人员最可贵的一种品质，就是不做这种侥幸者：缺陷等级怎么定就怎么报，风险多大就说多大，不粉饰、不缩水。**如实记录、如实上报、不隐瞒不利结果，这是诚信，也是这个职业能被信任的根基。**

过渡：2012 年的券商如此，2017 年的云计算巨头也没能幸免——下一页，AWS 宕机整整 4 个小时。

【1.18】AWS 宕机整整 4 个小时（2.5 分钟）

前面几个案例，都是某一家公司自己出事。这一页不一样，出事的是一家"替别人保管系统"的公司——亚马逊云，也就是 AWS。课件写得很清楚：2017 年 3 月 2 号，亚马逊云出现严重的宕机；亚马逊云之前也出现过宕机，但一般在一个小时之内解决问题，而这次非常严重，宕机整整四个小时。缺陷类型标注是"性能、稳定性问题"。

这里要请同学们注意，出问题的是云上的基础存储服务。说白了，它就是云上的"硬盘"——很多网站、APP 的图片、视频、备份数据都放在里面。它一停，不是一家网站打不开，而是很多网站图片显示不出来、很多 APP 直接报错。这就是云时代的放大效应：以前一个软件缺陷影响一个用户、一家公司；现在一个底层服务的缺陷，会沿着依赖链传到成千上万个系统上，这也叫级联失效。

再看时间。课件说，以前一小时能解决，这次四个小时。为什么这次这么久？因为它出问题的地方太底层了——修一个跑在最上面的应用很快，可当你发现是地基出了问题，能做的事情就非常有限，只能等它慢慢恢复。这是稳定性问题的一个典型特征：**越底层的组件，故障影响面越大、恢复时间越长。**

那我们做测试的同学从这里要学到什么？第一，性能与稳定性是独立的测试维度，跟功能测试不是一回事：功能测试回答"这个按钮点了有没有反应"，性能稳定性测试回答"一万人同时点会怎么样、连续跑一整天会怎么样"。第二，测试要关注依赖：你的系统再健壮，如果它依赖的那个云服务挂了，你也会挂。所以现代测试里有依赖分析和故障注入这类专门的工作，就是去模拟依赖失效，看系统还能不能撑住。

板书：AWS · 2017-03-02 → 宕机 4 小时（以往 1 小时内解决，据课件数据）→ 缺陷类型：性能 / 稳定性 → 云时代放大效应：级联失效 → 越底层，影响面越大、恢复越慢

提问：一个"存储服务"坏掉，为什么会让很多看起来毫不相关的网站一起打不开？

预设回答：① "因为大家的数据都存在里面"——对，这是依赖关系。追问：如果你的系统要抗住这种风险，测试阶段应该做什么？答：做依赖分析、故障注入，验证降级预案。② "说明他们测试做得不好"——要引导得更准确：更可能是稳定性与容灾能力的不足，而不只是"没测"；四小时里他们要处理恢复流程、依赖启动

顺序、流量限制，这些都需要提前演练。③ "云计算就是不可靠"——纠正：云不是不可靠，而是把风险集中了，管理风险的前提是承认依赖、度量依赖。

易错点 / 考点：数字记 2017 年 3 月 2 日、4 小时、以往 1 小时内。概念上，性能问题与稳定性问题常被混为一谈——性能看"快不快"，稳定性看"长时间、高负载下稳不稳"。案例题常考"云服务故障对下游系统的影响属于什么风险"，答依赖风险（级联失效）。

过渡：宕机影响的是"用不了"；下一页 Uber 这个案例，影响的是更敏感的东西——你的隐私。

【1.19】Uber 泄漏个人隐私，导致用户要求赔偿 3 亿多元（3 分钟）

这一页讲的是隐私。课件的原文是：2017 年 2 月，根据法国《费加罗报》报道，一名法国商人起诉 Uber，要求该公司赔偿 4500 万欧元；按当时汇率 7.4 计算，合计人民币 3.33 亿，以弥补隐私漏洞对自己婚姻造成的伤害。他的理由是，Uber 专车 App 中存在这样的安全性漏洞。标题说"3 亿多元"，跟正文的 3.33 亿是一致的。

我们先把数字换算一遍，大家跟着算：4500 万欧元乘以 7.4，等于 3.33 亿人民币。这个数字很有冲击力，但更值得琢磨的是一个细节——**索赔的理由不是"我的钱被偷了"，而是"我的隐私漏洞伤害了我的婚姻"。** 这说明泄露出去的绝不是一串没用的字符，而是能定位一个人、能还原一个人生活轨迹的信息：他什么时候叫车、从哪里上车、到哪里下车、多久去一次某个地址。这些数据拼起来，就是一张生活地图。

什么叫隐私保护问题？一句话：系统收集了用户的个人信息，却没有能力保证这些信息只被该看的人、在该看的时候看到。它属于安全性缺陷的一个分支。生活里的类比就是酒店的房卡：你住 301，房卡只能开 301；如果前台把一张万能卡随手放在大堂，问题不在于"卡坏了"，而在于权限管理失效了。这一页的问题，就是这把万能卡没管好。

从测试角度看，安全性测试跟前面讲的性能、功能都不太一样。它要回答三个问题：谁能看到什么、谁能改什么、谁能以别人的身份做什么。这类测试要求一种特别的思维方式——"攻击者思维"，站在坏人的角度想：如果我想拿到别人的数据，我会从哪里入手？后面讲安全性测试时，我们会专门讲越权访问、权限校验、传输加密、日志与审计这些检查点。还要提醒一句：隐私问题的后果往往是慢性的，泄露那一天没感觉，某一天你才发现有人知道你的全部行踪。

板书：Uber · 2017-02 · 法国《费加罗报》 → 索赔 4500 万欧元 ≈ 3.33 亿人民币（汇率 7.4，据课件数据）→ 缺陷类型：安全性 / 隐私保护 → 核心问题：权限与访问控制 → 攻击者思维：看得到 · 改得动 · 冒

> 充得了

> **提问：**一个打车 App 记录你的上车点和下车点，为什么这可能构成隐私漏洞？

> **预设回答：**① "因为这些数据能暴露我的住址"——好回答，追问：除了住址还能推出什么？工作地、作息规律、社交关系、甚至健康状况。② "只要有密码就安全了"——典型误区。密码只解决"是不是本人登录"，解决不了后台有没有越权看数据、接口有没有绕过校验。③ "数据在我手机里，别人拿不到"——提醒他：这些数据实际存在服务端，还要经过网络传输，服务端、传输、终端三个环节都可能出问题。

> **易错点 / 考点：**换算必考——4500 万欧元 × 7.4 ≈ 3.33 亿人民币。概念上，安全性缺陷不等于功能缺陷，它的特点是"功能都正常，但数据被不该看的人看到了"。口诀：**看得到、改得动、冒充得了——三个方向查安全。**

> **过渡：**大公司出大事故，普通人的遭遇往往更具体。下一页是 2019 年国庆假期，很多人订了酒店，却住不进去。

【1.20】预定的酒店住不进去、露宿街头（3 分钟）

这一页是一个发生在我们身边的故事。课件写的是：正值 2019 年国庆假期出行高峰期，通过某知名旅行网预定酒店的不少网友遭遇"人在囧途"——到达酒店却无法入住，已支付的订单却显示未支付或者订单不存在；与此同时，客服电话打不通，想退房也退不掉；有些旅客想重新预定，在这样的出行高峰期又很难订到房。最后的结果，就是标题里写的：预定的酒店住不进去、露宿街头。

这一页课件没有标注缺陷类型，所以下面这个判断是我们根据现象做的分析，不是课件原文：从"已支付的订单却显示未支付或订单不存在"来看，这是一个典型的功能与数据一致性缺陷——钱付掉了，订单状态没同步；再叠加客服打不通、退不掉，就是可用性和服务响应能力的问题。

我想请大家特别体会"假期高峰期"这四个字。为什么这类缺陷平时不出现，一到高峰期就爆发？因为高峰期的本质是**并发**：平时每分钟十个人下单，服务器轻轻松松；高峰期每分钟几万人下单，系统在压力下就暴露出一堆平时藏着的毛病——请求超时不处理、消息丢了、状态写了一半、重试把订单写重了。这跟上页讲的稳定性问题是同一家人，只不过这次被击穿的是业务数据的一致性。

再算一下这次事故的"代价"。课件没有给乘客的经济损失数字，我们也不编。但可以算另外一种账：一家平台，用户在寒夜里站在酒店门口却住不进去，第二天他还会用这个平台吗？**这就是信任成本。** 功能缺陷可以靠一个补丁修好，用户对你的信任一旦掉了，补丁修不回来。生活里的类比就是餐厅订位：你打电话订了位，

到店里说没有你的记录，经理也找不到人核实，旁边又没有空桌——这不只是"系统出故障"，这是让顾客站在门口。所以测试用例不能只在"一个人、一条数据、正常流程"下设计，必须要有高并发、状态流转、异常中断这些场景。

板书：2019 国庆 · 某知名旅行网 → 已支付订单显示"未支付 / 不存在" → 无法入住、退款无门、客服不通 → 判断（非课件原文）：功能与数据一致性缺陷 + 可用性不足 → 触发条件：高峰期并发 → 代价：信任成本

提问：为什么这类"订了房住不进去"的缺陷，平时的测试很难发现？

预设回答：① "因为平时没那么多人用"——对，这是并发场景缺失。追问：测试时能不能造出高并发？答：能，压力测试工具就是干这个的。② "因为订单系统太复杂"——部分成立，要顺势引导到可测性：越复杂越要把状态机拆开，把"待支付→已支付→已确认→已完成"每一次状态转移都测到。③ 典型的错误回答："这只是小概率事件，不用管"——要驳回去：涉及钱和承诺的业务，小概率乘以大用户量就等于必然发生。

易错点 / 考点：注意课件这一页**没有**给缺陷类型标签，答题时不要一句"性能问题"就完事，更准确的说法是数据一致性问题叠加可用性问题。考点常以"给现象判断缺陷类型"的形式出现。口诀：**钱付了、单没了——先查状态同步，再查并发。**

【思政落点 4】这个故事没什么高科技含量，但它最能让学生理解"用户视角"。对我们来说，那只是一个订单状态字段错了；对那位旅客来说，那是全家人半夜拖着行李站在街头。做测试要有一种把用户请到心里的能力：**你在设计用例时多想的那个异常场景，可能就是某个真实用户免于受困的那一晚。** 质量责任不是抽象口号，它落在每一条用例的细节里。

过渡：讲到这里，我们已经看了六个案例。下一页，课件把镜头拉开，告诉我们——这样的悲剧还有很多。

【1.21】更多的悲剧（3 分钟）

这一页的标题只有四个字："更多的悲剧"。课件给了两个例子，我念一遍，请大家把数字记准。

第一个是放射性治疗仪 Therac-25。它的软件存在缺陷，导致几个癌症病人受到非常严重的过量放射性治疗，其中 4 个人因此死亡。关键在"过量"两个字——放疗设备的剂量是精确控制的，软件出错让它给出的剂量远远超出安全范围。这不是"仪器不好用"，这是把治疗变成了伤害。

第二个是爱国者导弹防御系统。课件写的是：当这个防御系统的时钟累计运行超过 14 小时之后，系统的跟踪系统就不准确了，从而导致拦截伊拉克"飞毛腿"导弹的几次失败，其中一枚在沙特阿拉伯的多哈爆炸的"飞毛

腿"导弹造成 28 名美国士兵死亡。这一页有两个数字必须记住：**14 小时**和 **28 人**。

我们把这两个案例分开看，它们各代表一种非常典型、也非常可怕的缺陷。Therac-25 代表的是"安全关键系统中的软件缺陷"：在这类系统里，软件不是辅助工具，软件直接控制物理世界——剂量、速度、开关。软件缺陷不再是"少算了几毛钱"，而是直接指向人的身体。所以在医疗、航空、核电、汽车这类行业，测试的标准和强度跟普通应用完全不是一个量级，通常还有专门的标准和认证要求。

爱国者导弹代表的是"跟时间有关的缺陷"，也就是我们后面会专门讲的时间/日期类缺陷。为什么运行 14 小时之后才不准？因为系统内部用了跟时间相关的量，运行时间一长，精度就丢掉了，跟踪随之出现偏差。这类缺陷最阴险的地方在于：**它不在刚上线的时候出现，而是在系统跑了很久之后才出现。** 如果我们的测试只跑十分钟就收工，这种缺陷永远发现不了。这两件事合起来，给了这门课一个最硬的理由：软件测试不是锦上添花的工序，在某些行业它就是人命关天的最后一道闸门。

板书：Therac-25 放疗仪 → 过量辐射 → **4 人**死亡（据课件数据）｜爱国者导弹 → 累计运行 **> 14 小时**后跟踪不准 → 拦截飞毛腿失败 → **28 名**美国士兵死亡（据课件数据）→ 两类典型缺陷：安全关键系统缺陷 · 时间/精度相关缺陷

提问：爱国者导弹的缺陷"跑 14 小时才出现"，这说明测试设计上缺了什么？

预设回答：① "缺了长时间运行的测试"——准确，这叫耐久性/稳定性测试。追问：那要跑多久才够？答：按业务可能的最长连续运行时间来定，不是拍脑袋，要看真实的连续工作场景。② "应该一发现就重启系统"——这是运维上的权宜之计，不是测试；而且这种"重启掩盖"会让缺陷更难被发现，可以顺势讨论"用临时办法遮住缺陷"的危害。③ "这是硬件问题吧"——纠正：时钟累计运行导致的精度丢失与时间处理有关，账要算在软件上。

易错点 / 考点：必背数字——Therac-25 致 **4 人**死亡；爱国者系统累计运行 **14 小时**后跟踪不准，造成 **28 名**美国士兵死亡。常见混淆：把 Therac-25 说成"操作人员失误"，课件明确写的是软件存在缺陷。判断题高频："只有功能错误才会造成严重后果"——错，时间、精度、安全类缺陷同样致命。口诀：**剂量、时钟，两条命门。**

【思政落点 5】讲到 28 名士兵、4 位病人的时候，我们要停一下。这些数字背后，是一个个再也回不来的家庭。软件测试在有些行业是一份"沉默的守护"——做得好，没有人会记得你；做不好，代价由别人承担。要让学生明白：**我们将来提交的每一份测试报告，都可能成为某条产品线的安全底线。** 认真，是这个职业最高级的品质。

> **过渡：**六个案例、至少几十条人命、十几亿美元的损失，都指向同一个问题——我们为什么要做软件测试？下一页，课件正式给出这一节的标题。

【1.22】1.2 为什么要进行软件测试？（1.5 分钟）

前面六页案例看下来，我相信大家心里已经有答案了。不过我们还是要正式地把这个问题提出来：**1.2，为什么要进行软件测试？**这一页是本节的开场，课件上没有堆结论，只留了一个标题，等着我们往下回答。

在往下走之前，我先交代一下我们这门课"讲义"的用法，课件的备注里也专门强调了这一点：讲义不是语音讲解的辅助材料，它本身就是学习的主体。也就是说，你课后复习靠的是这份讲义，而不是靠回忆我上课说了什么。所以每一页我都会尽量把话写完整，你单独看讲义也能学得下去。

我们这门课讲每一个学习要点，都会按这样一条线索走：第一，先讲概念——这个知识点是什么，在工作中什么场景会用到它；第二，再展开知识和技术——要胜任这个能力，你需要掌握什么，我会讲细；第三，举例子，用一个案例或一个演示帮你理解怎么用；第四，分享经验，讲讲理论和实际工作的差别在哪里，这些往往是踩过坑才知道的；第五，给你们学习建议；第六，如果有对应的工具和模板，我也会一并给出来。

那么这一节到底要解决什么问题？就是一句话：把"测试"从一个"工序"变成一种"投资判断"。学完这一节，你们要能回答三件事：第一，测试为什么是保证质量的必要手段；第二，不做测试的代价是怎么一层层放大的；第三，测试这笔钱为什么值得花。

所以接下来这一节，我们就要正面回答"为什么要测试"。请大家带着前面那几个案例去想：如果当初多做一道测试，哪些悲剧是可以避免的？

> **板书：**1.2 为什么要进行软件测试？ → 讲义＝学习主体 → 学习六步：概念 → 知识/技术 → 示例或演示 → 经验分享 → 学习建议 → 实用工具

> **提问：**请你用一句话回答——为什么必须做软件测试？（不许用"为了找 BUG"这一句）

> **预设回答：**① "为了保证质量"——直接命中下一节课件的第一句话，可以表扬并追问：质量和缺陷是什么关系？② "为了发现缺陷、把缺陷清出去"——也对，追问：那发现了不修会怎样？（引出劣质成本）③ "因为软件一定会出错，用户会骂人"——很实在，把它接到"用户期望"上：质量就是满足用户规定和隐含期望的程度。对"为了交差""公司规定要测"这类回答，要明确纠偏：那是形式，不是目的。

> **易错点 / 考点：**这一页本身是过渡页，不产考点，但"讲义是学习主体"这句话要记住——课后复习以讲义为准，考试范围也以讲义和教材为界。另一个提醒：不要用"为了找 BUG"这种手段性表述来回答目的性问题。

过渡：好，下一页，我们把这个问题正式回答一遍。

【1.23】为什么要进行软件测试？（3 分钟）

我们正式回答这个问题。课件的答案很朴实，第一句话就是："答案很简单，就是为了保证软件质量。"注意"简单"这两个字——测试的目的不是"证明程序没错"，更不是"让开发难看"，而是为了保证质量。

第二句话是：软件总存在缺陷。只有通过测试，才可以发现软件缺陷；也只有发现了缺陷，才可以把软件缺陷从软件产品或软件系统中清理出去。请大家把这条逻辑链捋一下，它有三个环节：**软件一定有缺陷 → 测试才能发现缺陷 → 发现之后才能清除缺陷。**这条链子断在任何一环，结果都一样：缺陷留在产品里。如果不承认"软件总存在缺陷"，就会觉得测试是多此一举；如果发现了缺陷却不清理，测试就成了走过场。

第三句话讲的是成本：软件中存在的缺陷给我们带来的损失是巨大的，软件测试是软件质量保证的关键步骤；测试作为一种"预防和评估成本"的投入，从而降低缺陷造成的劣质成本。这里出现了三个成本概念，是本章的一个重点，我把它写在黑板上：**预防成本和评估成本，是我们主动花的钱；劣质成本，是缺陷漏出去之后被迫花的钱**——召回、赔偿、返工、客户流失，全算劣质成本。前面几页案例里那些几亿美元的损失，全都是劣质成本。测试的逻辑就是：花一笔小小的预防与评估成本，去换掉一大笔劣质成本。

那为什么不干脆不测、把钱省下来？因为这两笔钱不对等。业界的经验规律是：缺陷发现得越晚，修复它的代价就越高。为什么？因为越晚发现，要连带改动的东西越多——代码写完了、集成进去了、文档写了、客户培训做了，甚至已经上线了；这时候改一个缺陷，等于返工一整套东西。

课件最后一句是：软件测试在产品开发中占据着相当重要的位置，这也是软件行业几十年的实践所证明的一个道理。

板书：为什么测试？→ 保证质量 → 软件总存在缺陷 → 测才能发现 → 发现才能清除 → 缺陷损失巨大 → 测试＝预防成本＋评估成本 → 降低劣质成本 → 几十年工程实践证明

提问："测试是一种成本"——那是不是测试投入越多越好？

预设回答：① "越多越好，测到没有缺陷为止"——要引导：测试有边际效益递减，无限测会把项目拖死；正确说法是在质量目标与成本之间找平衡。② "投太多不划算"——对了一半，追问：那怎么判断该投多少？答：看缺陷后果的严重程度、看质量目标、看风险等级。③ "看项目大小"——补充：还要看行业，医疗、航空的测试强度远高于普通应用，因为他们漏出去的缺陷代价完全不同。

易错点 / 考点：三类成本的名字必须记准——预防成本、评估成本、劣质成本，测试属于前两类的投入。这是

选择题高频点。案例分析常给"某公司为省测试费直接上线，结果赔偿若干万元"，问赔偿属于哪类成本——答劣质成本。口诀：**预防＋评估是投资，劣质成本是学费。**

过渡：既然测试是为了保证质量，那"软件测试"这件事本身，到底该怎么定义？我们进入 1.3 节。

【1.24】1.3 什么是软件测试？（2 分钟）

接下来这一节，题目叫"什么是软件测试"。别看只有五个字，这是本章、也是这门课的地基。如果连"什么是测试"都说不清楚，后面讲用例设计、讲测试级别、讲自动化工具，全都是空中楼阁。

这一节我们分四个小节走，课件和教材的安排是这样：第一小节 1.3.1 软件测试学科的形成，教材在第 11 页；第二小节 1.3.2 正反两方面的争辩，教材在第 12 页；第三小节 1.3.3 软件测试的定义，教材在第 13 页；第四小节 1.3.4 软件测试的其它观点。这四个小节是一条线：先看历史，再看争论，然后才落定义，最后补充其它视角。

这个顺序本身就有讲究。为什么不直接甩一个定义出来？因为软件测试的定义不是天上掉下来的，它是几十年的实践和争论沉淀出来的。你先看争论，才会明白为什么定义里要有某些限定；你先看历史，才会明白为什么测试从"程序写完之后的事"变成了"贯穿开发全过程的事"。所以我希望大家听这一节的时候不要急着抄定义，先跟着走一遍"为什么会有不同的说法"。

这一节学完，你们应该能回答三个问题：测试是什么？测试和调试有什么区别？"测试是为了证明程序是正确的"这句话，错在哪里？

最后提醒一句学习方式：这一节的概念会成为后面所有章节的"公理"。比如后面讲测试用例设计，为什么要设计异常用例？根子就在"测试是发现错误"这个认识上；再比如讲测试级别，为什么测试要贯穿单元、集成、系统、验收？根子就在"测试有不同视角和目标"。所以这一节看起来最简单，实际上最不能打折扣。

板书：1.3 什么是软件测试？→ 1.3.1 学科的形成（教材 p11）→ 1.3.2 正反两方面的争辩（p12）→ 1.3.3 定义（p13）→ 1.3.4 其它观点 → 路线：历史 → 争论 → 定义 → 补充

提问：凭直觉说一句，你认为"软件测试"是什么？

预设回答：① "就是找 BUG"——最常见。点评：方向不错，但"找 BUG"是手段不是目的，追问一句"找到 BUG 之后呢？"，把话题引到"提供质量信息、清除缺陷"。② "就是运行程序看看能不能用"——这属于动态测试里的功能验证；提醒他测试还包括静态测试，不运行程序也能测，比如需求评审、代码走查。③ "测试就是保证软件没有错"——典型错误，要坚决纠正：测试不能证明软件没有错，只能发现其中存在的错。

易错点 / 考点：本节最大的误区是"测试是为了证明程序是对的"，先把这个误区立住，后面正式定义时再来拆。另外记住四小节与教材页码的对应（11/12/13）。口诀：**测试是发现错误，不是证明正确。**

过渡：那我们就从第一小节开始，看看"软件测试"这门学科是怎么一步步形成的。

【1.25】1.3.1 软件测试学科的形成（1.5 分钟）

这一页是把 1.3 节的四个小节摊开，先给大家一张地图。我们刚才已经点过一次，这里再明确一遍：1.3.1 软件测试学科的形成，1.3.2 正反两方面的争辩，1.3.3 软件测试的定义，1.3.4 软件测试的其它观点。

这四块内容，知识密度是递增的，但认知难度是先降后升的，我解释一下。1.3.1 讲学科怎么形成，是故事，最好懂，可它决定了你后面所有判断的底线；1.3.2 讲正反两方面的争辩，也就是正向思维和反向思维的对立，这块最容易被忽略，但它恰恰最有用——因为它决定了你测的时候是"顺着验证"还是"故意找茬"；1.3.3 给出定义，这是要背、要考、要能准确表述的；1.3.4 是其它观点，告诉你定义不只一家之言，帮你建立更完整的认识。

所以我给大家一个学习建议：这一节的四小节，不要只背 1.3.3 那一句定义。真到了考试或者面试问你"你怎么理解软件测试"，光有一句定义是撑不住场面的；你要能说出测试的历史演变、正反两种思维、以及不同观点的差异，这才叫理解。

另外提醒一句，页码我也写在这里了：教材上分别是第 11 页、第 12 页、第 13 页。课后读完这一节，第一次作业里"用自己的话写出软件测试的定义"那道题，你们就能答得更像样。

顺便说一下这四个小节在整门课里的位置：它们是"地图的图例"。后面几周我们讲的测试级别、测试类型、静态与动态、黑盒与白盒，全都可以挂到这张地图上。所以现在多花几分钟认清路，后面就少走弯路。

板书：1.3 学习地图 → 形成（p11）→ 争辩（p12）→ 定义（p13）→ 其它观点｜学习建议：不要只背定义，要能说出演变、思维与观点差异

提问：这四个小节里，你觉得哪一个最可能成为考点？为什么？

预设回答：① "定义，因为要背"——承认它是考点，但追问：如果只考定义，教材为什么还要花两节讲历史和争辩？引导学生理解知识之间的支撑关系。② "争辩，因为难"——好回答，可以表扬：正反思维是测试思维方式的根，案例分析题经常考。③ "其它观点，因为听起来高级"——提醒他观点要多、但主线要清，回答"你如何理解软件测试"时主线仍然是定义。

易错点 / 考点：四小节的顺序与教材页码（11 / 12 / 13）容易记混。记忆口诀：**先看形成（历史），再看争辩（思路），然后定名（定义），最后广角（其它观点）。**

过渡：那我们从第一小节开始——软件测试学科是怎么形成的。

【1.26】1.3.1 软件测试学科的形成（3 分钟）

我们进入 1.3.1，软件测试学科的形成。这一页课件上只有标题，但它要回答的问题很重要：测试这件事古已有之，可测试作为一个学科、一个专业，是很晚才出现的。为什么？这就跟它的形成过程有关。

这条形成过程，大致可以理解成三个阶段。第一个阶段，测试还不是一件独立的事，它藏在"调试"里面。程序写完，运行一下，看看哪里不对，改一改，再运行——这是开发人员自己做的事，没有专门的岗位，也没有系统的方法。这个阶段最大的问题是：测试的范围完全等于"开发者能想到的范围"。

第二个阶段，到了上世纪七十年代前后，业界开始把"测试"和"调试"分开来看。这里有一个理念上的大转弯：以前大家觉得测试是为了"证明程序是对的"，后来慢慢认识到，测试的本意是"发现程序里的错误"。教材 1.3.1 节里讲了这段演变的过程和相关的人物与年代，具体以教材为准，我这里只讲主线。这个转弯为什么重要？因为它决定了你测试的姿势——如果你想着"证明它是对的"，你就会顺着写程序的人的路子走，很容易漏；如果你想着"找出它的错"，你就会去挑边界、挑异常、挑组合。

第三个阶段，是测试的工程化和规范化。有了独立的测试角色、独立的测试团队、专门的流程和标准，测试从一门"手艺"变成了一门"工程"。今天我们看到的各种测试级别、测试类型、测试标准与认证，都是这个阶段的产物。再往前一步，就是我们当下说的"测试左移"和"全程测试"——测试不再是开发完成之后的最后一道工序，而是从需求阶段就开始介入。这个概念后面还会专门讲。

所以这一节给我们的整体印象是：软件测试不是某个人拍脑袋发明的，它是被一次次事故、一次次返工逼出来的一个专业。

板书：1.3.1 学科的形成 → ① 藏在调试里（开发自测，范围＝开发者的想象）→ ② 与调试分离（从"证明正确"到"发现错误"）→ ③ 工程化 / 规范化（独立团队 · 流程 · 标准）→ ④ 当代：测试左移、全程测试（详见教材 p11）

提问：为什么早期的"测试"会藏在调试里，而不是一开始就独立出来？

预设回答：① "因为那时候程序小，一个人就能搞定"——对，规模决定分工，这是最根本的原因。② "因为大家觉得测试不重要"——部分成立，但可以引导得更准确：不是不重要，而是当时没有人意识到它有独立

> 的价值。③ "因为那时候没有测试工具"——有道理，工具也是被需求催生出来的，可以顺带提一句"先有分工，才有工具"。

> **易错点 / 考点：**这一节最容易考的是"测试与调试的区别"（下一节展开）：调试是发现错误之后定位并修复的过程，属于开发活动；测试是发现错误、评价质量的过程。另一个易错点是仍把"测试是为了证明程序正确"当成对的。具体的年代、人物与事件以教材 1.3.1 节（教材 p11）为准。

> **过渡：**学科形成之后，关于"测试到底是干什么的"，业界的争论一直没有停。下一页我们就进入 1.3.2——正反两方面的争辩。

【1.27】从狭义的软件测试到广义的软件测试（7 分钟）

> 同学们，前面我们讲了软件缺陷带来的惨痛代价。从这一页开始，我们要回答一个更基础的问题：软件测试到底测什么？先看一个很常见的误解——很多人一说测试，脑子里出现的画面就是：程序写完了，运行起来点一点，看看有没有报错。这就是最狭义的软件测试，一句话概括：软件测试等于程序测试。但请注意，软件不只是可执行的程序。按照软件工程的观点，需求、设计、代码这三样东西，都属于软件的组成部分。需求说清楚要做什么，设计决定怎么做，代码才是最后的实现。既然这三样都是软件，那它们出了问题，是不是软件缺陷？当然是。而检查这三样东西有没有问题，靠的不是把程序跑起来，靠的是评审——需求评审、设计评审、代码审查。这些评审工作有一个统一的名字，叫静态测试。

>

> 那么问题来了：为什么过去我们不觉得评审算测试？因为早期观念里，只有"运行程序、看到结果"才算测试。可是大家想一想，如果在需求阶段就发现"这个功能压根没写清楚"，你还需要等代码写完再跑一遍去发现吗？完全不需要。所以，当业界正式提出"动态测试"这个词的时候，其实已经从反面承认了：还有一类不需要运行程序的测试，也就是静态测试。于是软件测试的范围扩大了——软件测试既包含静态测试，也包含动态测试，这就是广义的软件测试。

>

> 为什么一定要强调"广义"？因为它直接决定成本。有一句话请大家记牢：测试进行得越早，成本越低；反过来，缺陷发现得越迟，研发成本越高。你早一步在评审桌上指出需求里的二义性，代价可能只是改一句话；等系统上线了才发现，改的就是代码、接口、文档，甚至用户的信任。这笔账具体怎么算，我们看下一页的成本曲线。

> **板书：**狭义：软件测试＝程序测试 → 广义：软件＝需求＋设计＋代码 → 对需求/设计/代码的评审＝静态测试 → 软件测试＝静态测试＋动态测试 → 测试越早，成本越低

> **提问：**如果一个功能的需求文档里写"系统应快速响应用户请求"，这条需求本身有没有问题？它属于静态测试还是动态测试的检查对象？

> **预设回答：**①答"没问题，写得挺清楚"——要追问：多快算快？1 秒还是 10 秒？谁来判断达标？指出"快速"是二义性需求，测试无法据此判定通过与否。②答"这是需求问题，应该让开发去改"——要强调这正是测试人员的活：需求评审就是静态测试的战场，测试要在需求阶段就把不可测的词挑出来。③答"属于动态测试"——纠偏：需求文档不运行程序，属于静态测试。点评要点：能在需求阶段发现二义性，就是最便宜的缺陷修复。

> **易错点 / 考点：**常见混淆是把"软件测试"等同于"程序测试"（狭义），漏掉对需求、设计的静态测试。考点多以判断题出现："只有运行程序才能叫测试"——错。判断三步法：一看对象（需求／设计／代码／程序）→ 二看是否运行程序（不运行即静态测试，运行即动态测试）→ 三看时机（越早越便宜）。

> **过渡：**既然早发现能省钱，那到底能省多少？我们把成本曲线摊开来看。

【1.28】广义的软件测试可以极大地降低研发成本（8 分钟）

> 大家看这张图。横轴是软件开发的时间轴——需求、设计、编码、测试、上线运维；纵轴是修复一个缺陷所要付出的成本。图上这条曲线，请特别注意它的形状：它不是一条斜着往上的直线，而是一条越往右越陡的曲线。这就是我们反复强调的四个字：非线性增长。

>

> 什么叫非线性？打个比方。装修房子的时候，你在图纸阶段跟设计师说"这面墙我想敲掉"，设计师改一张图纸，几分钟的事，成本几乎为零。等到水电都铺好了你才说要敲墙，得重新布线、重新贴砖，又花钱又费时。要是房子已经住进去三年了才想起来要敲，那要搬家具、要临时找住处，成本翻好几倍。软件是一模一样的道理，而且软件更狠，因为软件是"牵一发而动全身"的。

>

> 具体怎么个牵动法？一个缺陷在需求阶段被发现，你改的是文档上的一句话，成本就是一次讨论。到了设计阶段，你得改设计文档、改接口定义，上下游同事都要跟着对齐。到了编码阶段，你要改代码，还要改相应

> 的单元测试。到了测试阶段才发现，代码要改、已经跑过的测试要重跑、相关模块要回归。要是等产品上线、用户已经用起来了才爆出来，那就不只是改代码的问题了：要紧急发版，要写公告，要处理用户投诉，严重的话还要赔偿、丢客户、丢口碑。这就是为什么这条曲线在后期会陡得吓人。

>

> 所以我们说，广义的软件测试之所以能极大地降低研发成本，道理就在这里：它把测试这件事从"最后一道关"变成了"贯穿始终的一条线"。需求评审、设计评审、代码审查这些静态测试，做的是最左边、最便宜的那一段；动态测试在中间和右边兜底。左边的成本压下去了，整个项目的总成本才降得下来。
>
> **板书：**缺陷发现越迟 → 修复成本越高，且非线性增长（曲线左低右陡）→ 成本构成：改文档／改设计／改代码＋回归／发版＋公告＋投诉＋赔偿 → 广义测试＝把测试铺满全过程，压住最左端

> **提问：**同样一个"登录后跳转错误"的缺陷，在需求评审时发现，和在产品上线后被用户发现，修复成本差在哪里？请至少说出三项差异。

> **预设回答：**①只说"花的钱不一样多"——要追问具体差在哪，引导说出：改动对象（文档／设计／代码／已上线系统）、连带工作（重测与回归范围）、外部影响（用户投诉、公告、口碑与赔偿）。②答"反正都要改代码，成本差不多"——典型错误，纠偏：越早发现，越可能只改文档或设计，根本不进代码。③答"上线后发现问题更严重所以贵"——肯定方向，补充"非线性"这一层：不是贵一点，而是成倍地贵，因为回归范围随系统规模扩大。点评：把三者串成一句——发现越迟，要动的东西越多、要重做的验证越多、要面对的外部人越多。

> **易错点 / 考点：**易错是把成本增长理解成线性，认为"晚发现无非多花一点时间"。考点：给一个事故情境问"该缺陷本应在哪个阶段被发现，能省下哪些成本"（案例分析）。记忆口诀：早一步在文档上改一句话，晚一步在系统里改一圈。

> **【思政落点 1】**：讲国外真实事故——Therac-25 放疗仪因软件缺陷导致患者受到过量辐射致死，骑士资本因交易系统缺陷在 45 分钟内亏损约 4.6 亿美元。这些代价背后共同的一点是：缺陷在最贵的那一段才被发现。要让学生先估算损失量级，再对照真实数据，体会"没有测试把关就交付"意味着什么。落到职业观：测试人员的质量责任不是口号，你手里那张用例表，可能对应着使用者的生命安全与公众利益。

> **过渡：**知道了成本曲线，我们再把镜头拉远，看看软件测试这门学科是怎么一步步走到今天的。

【1.29】软件测试学科的发展（8 分钟）

这一页讲软件测试学科的发展脉络，四个阶段、四个关键词，请大家把年份和"导向"对应起来记。

>

第一个阶段，1957 到 1978 年，以功能验证为导向。这个时期大家对测试的理解是：测试就是证明软件是正确的。注意这句话背后的思维方式，是正向思维——我先相信软件是对的，测试的任务是把这个"对"证明出来。所以做法通常是跑一遍功能，看看结果跟预期是否一致，一致就算通过。

>

第二个阶段，1978 到 1983 年，以破坏性检测为导向。这是一个观念上的转折点。人们发现，你按说明书跑一遍功能、跑通了，并不能说明软件没问题——你只是没找到问题而已。于是测试的目的变了：测试是为了找到软件中的错误。这是逆向思维：先假定软件有错，然后想尽办法把它找出来。

>

第三个阶段，1983 到 1987 年，以质量评估为导向。这一时期，测试的意义又被抬高了：它不只是找错，还要提供产品的评估和质量度量。也就是说，测试要回答"这个产品现在到底处于什么质量水平"——缺陷有多少、严重程度如何、哪些模块风险高。测试开始从"找 bug 的手艺"变成"提供质量信息的工作"。

>

第四个阶段，1988 年起，以缺陷预防为导向。测试是为了展示软件符合设计要求，发现缺陷、预防缺陷。请注意最后两个字：预防。发现缺陷是治已病，预防缺陷是治未病——通过分析缺陷的分布和模式，回头去改需求和开发的流程，让同一类缺陷不再反复出现。

>

四个阶段连起来，是一条很清晰的线：证明对 → 找出错 → 评估质量 → 预防缺陷。测试的地位，也从"事后挑毛病"一路升到"过程改进的引擎"。

板书：1957–1978 功能验证（正向：证明正确）→ 1978–1983 破坏性检测（逆向：找错误）→ 1983–1987 质量评估（提供质量度量）→ 1988– 缺陷预防（发现＋预防缺陷）

提问：这四个阶段里，哪些阶段的测试目标已经超出了"找缺陷"本身？它们分别多做了什么？

预设回答：①只答"第四个，1988 年起"——部分正确，要追问：第三个阶段其实也超了，它提供的是"质量

> 度量"，回答的是产品处于什么质量水平。②答"第二阶段最重要，因为开始找错了"——肯定其转折意义，但提醒：找错仍是"治已病"，第四个阶段才引入预防。③答"都差不多，就是找 bug"——纠偏：四个阶段的目标从"证明正确"到"预防缺陷"，思维方式和工作产出都不一样，后期还要出质量度量和缺陷模式分析。点评：答案要落在"质量评估（提供信息）"和"缺陷预防（改流程）"这两级上。

> **易错点 / 考点：**易错是把 1957–1978 的"正向思维"与后面 Myers 的反向思维混为一谈；把"质量评估"当成"质量保证"。考点：年份与导向的匹配（选择题）、"测试目的的演变"（简答）。记忆口诀：证明—找错—度量—预防，四步走。

> **过渡：**这是一种划分方法。同样这段历史，换一个观察角度，还能切成三个阶段。

【1.30】不同的阶段划分（7 分钟）

> 刚才我们用"导向"把软件测试的历史切成了四段，这一页换成另一种切法，按学科成熟度切成三个阶段。两种切法讲的其实是同一段历史，只是观察角度不同，考试里两种都要认得。

>

> 初级阶段，1957 到 1971 年。这个阶段最典型的特征是一句话：测试通常被认为是对产品进行事后检验。什么叫事后检验？就是产品做完了，拉出来检一遍，合格就交付。它跟工厂里质检员拿卡尺量零件是一个思路。而且这个阶段缺乏有效的测试方法——没有系统的用例设计方法，没有覆盖率的说法，靠的是个人经验和直觉。所以这个阶段的测试，地位低、效果差，还经常背锅。

>

> 发展阶段，1972 到 1982 年。这十年里有一个标志性事件：1972 年召开了第一次关于软件测试的正式会议。为什么一次会议这么重要？因为在这之前，测试是每个程序员各自的手艺活，没有统一的术语、没有公认的方法、也没有交流的平台。有了正式会议，大家开始把测试当成一个值得研究的对象：怎么设计用例、怎么评价测试是否充分、测试和开发怎么分工。这些问题被摊到桌面上讨论，软件测试才真正开始"长学问"，也促进了它的发展。

>

> 成熟阶段，1983 年到现在。标志是国际标准 Std 829-1983 的发布。这个标准把测试文档规范了下来——测试计划、测试说明、测试用例、测试报告该写什么、怎么写，都有了统一要求。有了标准，测试就从"手

> 艺"变成了"专业"：它可以被教学、被检查、被审计、被认证。于是软件测试形成一门独立的学科和专业，成为软件工程学科中的一个重要组成部分。

>

> 三个阶段连起来：事后检验的手艺 → 有交流、有方法的研究对象 → 有标准、有分工的独立专业。这也解释了为什么今天的企业会设专门的测试部门、测试岗位和测试职级。
>
> **板书：**初级阶段 1957–1971：事后检验，缺有效方法 → 发展阶段 1972–1982：1972 年第一次软件测试正式会议，促进发展 → 成熟阶段 1983–：国际标准 Std 829-1983，独立学科与专业，软件工程重要组成部分

提问：为什么说 1972 年的第一次软件测试正式会议，是测试从"手艺"走向"学科"的关键一步？

> **预设回答：**①答"因为开会很重要"——空话，要追问：会议到底提供了什么？引导说出"统一术语、交流方法、形成研究共同体"。②答"因为这个会议发布了标准"——纠偏：Std 829-1983 是成熟阶段的标志，1983 年才发布，不要张冠李戴。③答"因为从此测试有了专门的人做"——肯定，补充：有了交流平台，个人经验才能沉淀成可传授的方法，方法才能变成标准和专业分工。点评：一次会议的价值，在于把个人经验变成公共知识。

易错点 / 考点：易错是把两种阶段划分的年份和标志事件搞混（尤其 1972 年会议与 1983 年标准），以及把初级阶段的"事后检验"误记成"缺陷预防"。考点：阶段—年份—标志事件三连线（选择／填空）。记忆三步：先定时段（1957–1971／1972–1982／1983–）→ 再定标志（事后检验／首次正式会议／Std 829-1983）→ 最后定地位（缺方法／促发展／独立学科）。

> **过渡：**历史的脉络清楚了，接下来进入 1.3.2，看一个至今还在争论的问题：测试到底是为了证明对，还是为了找出错?

【1.31】1.3.2 正反两方面的争辩（4 分钟）

我们进入 1.3.2 节，正反两方面的争辩。

>

> 先解释一下这个标题。软件测试从诞生到现在，围绕"测试的目的到底是什么"，一直有两种针锋相对的观点。一种说：测试是为了证明软件能正常工作，让用户和开发团队对它建立信心。另一种说：测试是为了证明程

> 序有错，你要是没找到错，那说明这次测试做得还不够狠。这就是所谓的正向思维和反向思维。

>

> 大家可能会问：这不就是个定义之争吗？咬文嚼字有什么意义？意义很大。因为你怎么理解测试的目的，直接决定你怎么设计用例、怎么判断测试做完了没有。

>

> 如果按正向思维，你的动作是：把需求说明书上的功能一条一条跑一遍，全都通过了，就认为测试完成、可以交付。如果按反向思维，你的动作完全不同：你要专门去想"这个功能在什么情况下会出错"——边界值、异常输入、并发、断网、极端数据，你要主动去攻击这个系统，直到想不出新的攻击方式为止。同样一个登录功能，两种思维写出来的用例，数量和质量都天差地别。

>

> 所以这一节我们不是死记概念，而是要搞清楚三件事：两种思维方式各自长什么样、各自出自谁的观点、它们的价值分别在哪里。最后我们还要落到一个更实际的结论——它们不是二选一，而是要在不同场景下配合使用。接下来两页，我们先分别认识两位代表人物。
>
> **板书：** 1.3.2 正反两方面的争辩 → 正向思维：验证软件"能工作"，跑遍功能直至全部通过 → 反向思维：假定软件有错，"找出错"，攻击薄弱点直至找不出问题 → 认知决定用例怎么设计、何时算测完

提问： 如果只允许你用一句话向新人解释"测试的目的"，你会站正向还是反向？为什么？

> **预设回答：** ①站正向："测试就是确认功能都对"——追问：功能都通过了，能保证没有隐藏问题吗？引导认识"通过≠无错"。②站反向："测试就是找 bug"——肯定这是 Myers 的经典立场，再追问：一个查不出 bug 的测试就等于失败吗？引导到"测试的价值也在于提供质量信息"。③答"看情况"——这个答案最有前途，顺势引导到下一页的结论：两种思维各有适用场景，工程上要结合使用。点评：先让学生亮明立场，再用后面的内容让他们自己修正立场。

易错点 / 考点： 易错是把"正向思维"理解成"积极的态度"、把"反向思维"理解成"消极的态度"——两者说的是方法论立场，不是工作态度。考点：给出一段测试行为描述，判断属于正向还是反向思维。

> **过渡：** 先看正向思维的代表人物和他的原话。

【1.32】软件测试的正向思维（8 分钟）

这一页是正向思维的代表，Bill Hetzel 博士。请大家把他的名字和"正向思维"绑在一起记，这是考点。

>

> 先看他最核心的一句话：软件测试就是为程序或系统能够按预期设想运行而建立信心的过程。请注意这句话的落点——"建立信心"。在 Hetzel 看来，测试的产出不只是"发现了几个问题"，更是"让相关的人相信这个系统能按预期跑起来"。如果测试做完了，大家对这个系统能不能上线还是心里没底，那测试就没达到目的。

>

> 他还有一句更书面、更像定义的表述：软件测试是一系列活动，以评价一个程序或系统的特性或能力，并确定是否达到预期的结果。这句话里有三个动作，请大家拆开来记：第一，它是"一系列活动"，不是随手点几下；第二，它的对象是"特性或能力"，比如性能、可靠性、易用性；第三，它的结论是"是否达到预期的结果"——也就是说，测试要给出一个有依据的判断。

>

> 接着看第三句：测试是为了验证软件是否符合用户需求，即验证软件产品是否能正常工作。这句话把"预期"的来源点清楚了——预期不是开发人员自己拍脑袋定的，而是用户需求。所以正向思维的本质，是拿需求当尺子去量产品。

>

> 我们用一个生活场景来体会。你去餐厅点了一份牛排，要求七分熟。菜端上来，你切开看一眼，确实是七分熟，于是你满意了。这个"看一眼、确认符合要求"的过程，就是正向思维。它回答的问题是：该做的，做到了吗？

>

> 但大家马上能想到一个问题：我看这一眼，只能证明这一块牛排是七分熟，不能证明后厨没有别的隐患——比如食材新不新鲜、有没有交叉污染。也就是说，正向思维能验证"符合要求"，但它天然不擅长发现"没想到的问题"。这个短板，就是下一页反向思维要补的。
>
> **板书：**正向思维代表：Bill Hetzel → 测试＝为"按预期运行"建立信心的过程 → 测试＝一系列活动，评价特性/能力，确定是否达到预期结果 → 测试＝验证软件是否符合用户需求 → 回答：该做的，做到了吗？

提问：按 Hetzel 的说法，"测试是一系列活动以评价特性或能力并确定是否达到预期结果"。这句话里的"预期

结果"应该由谁提供?

> **预设回答：**①答"开发人员"——纠偏：预期来自用户需求，开发人员自己的假设不能当尺子。②答"测试人员自己判断"——追问：测试人员的个人判断如果和需求不一致怎么办？引导到"以需求文档、规格说明为依据，需求不清就回到需求阶段解决"。③答"用户需求"——正确，追问：如果需求文档本身就写得含糊呢？正好呼应前面讲的静态测试——需求二义性要在评审阶段挑出来。点评：测试的尺子是需求，尺子本身不准，量出来的结论就没有意义。

易错点 / 考点：易错是把"建立信心"理解成"测试就是走个过场让人放心"。考点：Hetzel 与 Myers 各自观点与人名的对应（选择题／连线题）；"测试是为了验证软件是否符合用户需求"出自谁的观点。判断三步法：看落点——落在"符合预期／建立信心"是正向，落在"发现错误／证伪"是反向。

> **过渡：**正向思维把测试当"确认"，反向思维则把测试当"进攻"。我们看 Myers 怎么说。

【1.33】软件测试的反向思维（8 分钟）

这一页是反向思维的代表，Glenford J. Myers。如果说 Hetzel 的落点是"建立信心"，Myers 的落点就是四个字：证明有错。

>

> 先看他的第一句话，也是最有冲击力的一句：测试是为了证明程序有错，而不是证明程序无错误。按常识，我们做测试不就是为了确认软件没毛病吗？Myers 说不。他的逻辑是：你把软件跑一遍没发现问题，只能说明这一次没找到，不能证明软件没有问题；而只要你能找到一个错误，你就确凿地证明了"这个程序有错"这件事。所以从逻辑上讲，测试能够证明的是"有错"，不能证明的是"无错"。

>

> 再看第二句：一个好的测试用例在于它能发现至今未发现的错误。这一句把"好用例"的标准给改了。过去我们评价用例，看的是它覆盖了哪个功能、写得规不规范；Myers 说，这些都不是关键，关键是你这条用例有没有真的抓到过问题。一条跑一万遍都通过的用例，和一条抓出严重缺陷的用例，价值完全不一样。这个观点直接催生了后来的一整套用例设计方法——等价类、边界值、错误推测，全都是为了让用例"更容易抓到错"。

>

> 第三句更彻底：一个成功的测试，是发现了至今未发现的错误的测试。注意"成功"这两个字的位置——测试的成功不在于"全部通过"，恰恰在于"逮到了问题"。这就带来一个推论：如果一轮测试下来一个缺陷都没发现，这轮测试到底算成功还是失败？按 Myers 的立场，得先怀疑是不是做得太软了。

>

> 生活里也有类似的场景：安检员一天下来什么都没查出来，我们不会觉得他工作很成功，反而会担心他是没认真查。所以反向思维的本质是：先假定系统有问题，再带着"我要把它找出来"的心态设计测试。正向和反向到底谁对？下一页我们用一张表把两者摆在一起看。
>
> **板书：**反向思维代表：Glenford J. Myers → 测试是为了证明程序有错，而不是证明程序无错误 → 好的测试用例＝能发现至今未发现的错误 → 成功的测试＝发现了至今未发现的错误 → 回答：怎么能把它弄坏？

提问：按 Myers 的观点，如果一轮测试执行完毕，一个缺陷也没有发现，你该如何评价这轮测试？

> **预设回答：**①答"说明软件质量很好"——最典型的错误，纠偏：也可能说明用例设计得太弱、只跑了主流程、没有攻击边界和异常。②答"说明这轮测试不合格"——太绝对，追问：如果用例已覆盖边界、异常、组合，仍然没发现缺陷，那它的价值在哪里？（提供质量信息，说明该范围的缺陷密度低。）③答"要看用例是怎么设计的"——最成熟的回答，肯定并总结：先看测试的充分性，再谈软件的质量。点评：把"没发现缺陷"当成需要解释的现象，而不是结论。

易错点 / 考点：易错是把"Myers 说测试是为了证明程序有错"误解成"测试人员的工作就是挑刺、和开发对立"——它说的是方法论，不是人际关系。考点：Myers 三句名言的填空与理解；"成功的测试"的判断（易出判断题，选项常写成"测试全部通过即成功"——错）。记忆口诀：Hetzel 建信心，Myers 找错误。

> **过渡：**两种思维摆在一起，差别就一目了然了。看这张对比表。

【1.34】认知决定着行为（9 分钟）

> 这一页的标题叫"认知决定着行为"，它把正向和反向两种思维放在一张表里对照。为什么要对照？因为这张表要证明一件事：你怎么认知测试，就直接决定你手上敲出来的用例长什么样。

>

> 我们先看表的上半部分。左边一路是正向思维：它的前提是验证软件正常工作；对应的行为描述是——在设

> 计规定的环境下运行软件的所有功能，直至全部通过。请注意这里面的两个关键词："所有功能"和"全部通过"。正向思维追求的是覆盖面加通过率，它是一张清单式的做法：需求上有多少功能，我就跑多少功能，跑完都绿了，收工。右边一路是逆向思维：它的前提是假定软件有错误；对应的行为描述是——寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统，直至找不出问题。

>

> 两句话一对比，差别非常清楚。正向是"遍历清单"，逆向是"攻击弱点"：一个按需求文档往下走，一个按风险和经验去找。正向问"这个功能正常吗"，逆向问"我怎么能把它弄坏"。

>

> 再往上看一点，这里还并列了两句关于测试的说法：一句是"评价一个程序或系统的特性或能力并确定是否达到预期的结果"，另一句是"测试是为发现错误而针对某个程序或系统的执行过程"。这两句话各自代言一种思维，最后都指向同一个词——软件测试。所以这一页真正想告诉大家的是：软件测试这个概念本身就同时包含两层含义，既有度量的成分，也有证伪的成分。它不是非此即彼，而是一体两面。

>

> 那工程上应该怎么用？我的建议是分场景：对安全攸关的系统，比如医疗设备、金融交易，要更多用逆向思维——漏掉一个缺陷的代价不可承受；对需求明确、迭代快速的普通业务系统，正向思维能保证基本功能可靠，但也不能只做正向，否则线上一定会被"没想到的路径"教育。
>
> **板书：**

认知（前提）	行为（做法）	终止条件
正向：验证软件正常工作	在设计规定的环境下运行软件的所有功能	直至全部通过
逆向：假定软件有错误	寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统	直至找不出问题

→ 结论：软件测试＝度量＋证伪，一体两面

提问：同样测一个"转账"功能，正向思维和逆向思维分别会写出什么样的用例？请各举一条。

> **预设回答：**①正向举例："输入正确账号和金额 100 元，点击转账，检查余额扣减 100 元"——典型的按规格验证，肯定它。②逆向举例："余额刚好等于转账金额""金额输入 0 或负数""转账过程中断网""并发两笔同时扣款"——这正是对薄弱环节的攻击，表扬。③答"逆向就是随便乱输"——纠偏：乱输不叫逆向思维，逆向

> 是有目的地找易错处（边界、异常、并发、状态），错误推测法也是讲方法的。追问：这两条用例哪一条更可能在真实项目里抓到线上事故？引导学生体会逆向用例的价值。

> **易错点 / 考点：**易错是把正向思维等同于某种用例设计方法（如等价类）、把逆向思维等同于错误推测法——两两相关但不等同，正向／逆向是目的层面的立场。考点：给出行为描述判断思维类型；简答"为什么说软件测试是一体两面"。判断三步法：看前提（假设软件对／错）→ 看动作（遍历功能／攻击薄弱点）→ 看终止条件（全部通过／找不出问题）。

> **过渡：**认知讲完了，接下来要落到最关键、也是考得最多的一件事：给软件测试下一个严谨的定义。进入 1.3.3。

【1.35】1.3.3 软件测试的定义（4 分钟）

> 我们进入 1.3.3，软件测试的定义。

>

> 可能有同学会想：前面都讲了两节课了，怎么到现在才讲定义？这恰恰是有意安排的。因为"软件测试"这四个字在不同人嘴里含义差别很大——有人指跑一遍程序，有人指找 bug，有人指评估质量，还有人把它和质量保证混在一起。如果不在前面把历史、把静态动态、把正反思维讲清楚，直接甩一个定义给大家，大家只会背下来，不会真正理解。

>

> 所以这一节的任务是：先看看大家心里默认的定义是什么，再用国际标准给出的定义把这个概念框住，最后用"软件测试的价值"来收口，回答"它到底给项目带来什么"。

>

> 我先给大家透个底：这一节里的定义来自 IEEE 和 ISO/IEC 29119 相关标准，都是英文原文加中文翻译。看到英文不要怕，我们逐句拆，重点是把定义里的每个要素抠出来——在什么条件下、对什么东西、做什么动作、得到什么结果。定义之所以是定义，就是因为这些要素缺一不可。以后你们写测试计划，实际上就是在回答这几个要素：条件是什么、对象是什么、动作是什么、结果怎么评价。

>

> 另外提醒一句，这一节后面还有"软件测试的价值"，那四点请大家务必记牢。它不仅是考试内容，更是你们在课程设计答辩时被问到"你这个测试有什么意义"时的标准答案。待会儿看到英文原文也别跳过——考试里定义要素的识别，就藏在那句话的每一个词里面。
>
> **板书：**1.3.3 软件测试的定义 → 为什么放到现在讲：先有历史／静态动态／正反思维 → 依据：IEEE、ISO/IEC 29119（对应 ISO/IEC 24765 词汇）→ 收口：软件测试的价值

提问：在看标准定义之前，请你用自己的话说一句"软件测试是什么"。你觉得你的说法里，缺了哪个要素？

> **预设回答：**①"软件测试就是运行程序找 bug"——缺"在特定条件下""观察或记录结果""做出评价"这几个要素。②"软件测试就是确认软件没问题"——这是正向思维的口径，缺"评价"和"发现差别"的成分。③"软件测试就是保证质量"——把测试和质量保证混了：测试提供质量信息，不"保证"质量。点评：把学生的答案先写在黑板上，讲完标准定义再回头对照，让他们看到自己漏了哪几个词，这是最有效的定义教学法。

易错点 / 考点：易错是把"软件测试"和"质量保证"当成同义词。考点：标准定义中要素的识别；"测试能不能保证质量"（判断题：不能，测试只能提供质量信息）。

> **过渡：**那我们先来看，大家心里想的那些答案，究竟哪个才算是软件测试。

【1.36】什么是软件测试？（8 分钟）

> 这一页是一张图，图上的问题很直白：什么是软件测试？然后列了一串大家脑子里的答案，我们一个一个过，看看哪些对、哪些不全对。

>

> 第一个答案：检查、检验，Check。这个说法很朴素，就是"看一眼有没有毛病"。它对，但太笼统——检查什么？怎么检查？检查到什么程度算够？都没回答。

>

> 第二个：验证软件能否正常运行。这就是我们前面说的正向思维口径，它的短板也很清楚：正常运行只说明主流程没问题，异常路径、边界情况都没碰到。

>

> 第三个：发现问题，Detect error。这是反向思维口径，抓问题、找错误。但它只讲了测试的一个动作，没

> 讲测试还要给结论。

>

> 第四个：证明是对的，Correction proof。这个说法最危险。因为软件测试在逻辑上做不到"证明软件无错"——你测过的路径是有限的，没测过的路径是无限的。所以严格讲，测试不能证明正确，只能提供证据。

>

> 第五个：质量评估，Quality evaluation。这个说法往前迈了一大步：测试要给出对质量的判断，比如缺陷密度、严重程度分布、风险模块。这是 1983 到 1987 年那个阶段的成果。

>

> 第六个：质量保证，Quality Assurance。注意，这个最容易混淆。质量保证管的是过程——规范、评审、度量、流程改进，从源头上让缺陷少产生；测试管的是产品——验证和确认，提供质量信息。测试是质量保证的重要手段之一，但测试不等于质量保证。

>

> 所以你看，这六个答案从"看一眼"一路走到"保证质量"，本身就是一个认知不断升级的过程。它们不是互相否定，而是层层包含：检查是动作，找错是目标，评价是结论，而质量保证是更大的范畴。下一步，我们就用国际标准的定义，把这些零散的认知收成一个完整、严谨的说法。
>
> **板书：**什么是软件测试？→ 检查/检验 Check？→ 验证能否正常运行？→ 发现问题 Detect error？→ 证明是对的 Correction proof？→ 质量评估 Quality evaluation？→ 质量保证 Quality Assurance？→ 六者层层包含，最后用标准定义收口

> **提问：**这六个答案里，哪一个说法在逻辑上最站不住脚？为什么？

> **预设回答：**①选"证明是对的"——正确，追问为什么：测试只能执行有限条路径，无法穷尽，所以只能提供证据，不能给出证明。②选"质量保证"——也抓住了要点，但要说清它不是"站不住脚"而是"范畴混淆"：质量保证包含测试，不能说测试就是质量保证。③答"检查、检验"——可以讨论：它不算错，只是太笼统。点评：要区分两类问题——逻辑错误（证明正确）与概念混用（测试 vs 质量保证）。

> **易错点 / 考点：**易错是认为"测试通过"就等于"软件没有缺陷"；把质量评估与质量保证混为一谈。考点：给出一句关于测试的说法，判断是否正确并说明理由（高频简答）。记忆口诀：查→找→评→保，四层递进；但"证明正确"这一层永远做不到。

> **过渡：**大家心里的答案过完了，下面看权威标准怎么定义。

【1.37】软件测试 IEEE/ISO29119 的定义（8 分钟）

这一页是权威定义，出自 ISO/IEC 24765《系统和软件工程词汇》，也是 IEEE 与 ISO/IEC 29119 采用的口径。英文原文我先读一遍，再逐要素拆。

>

> 原文是：An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

>

> 中文译文是：在特定的条件下运行系统或组件，观察或记录结果，对系统或组件的某个方面做出评价。

>

> 这个定义每个词都有讲究，我们拆成四个要素。

>

> 第一，它是一项活动。这提醒我们：测试是有组织、有计划的过程，不是随手点两下；是活动，就意味着有输入、有步骤、有产出。

>

> 第二，在特定的条件下运行。这个"特定条件"太重要了。测试绝不是"随便跑跑"，你必须先想清楚：用什么数据、什么环境、什么配置、什么前置状态。条件不明确，结论就没有意义。反过来说，测试用例的本质，就是把"特定条件"固化下来。

>

> 第三，观察或记录结果。这里的关键词是"记录"。光看见不行，要有可追溯的记录：实际结果是什么、截图在哪、日志在哪。这也是为什么我们课程实验要求大家截图留证。

>

> 第四，对系统的某个方面做出评价。注意两点：一是"做出评价"，测试必须给判断，不能只报现象；二是"某个方面"，说明测试是有侧重的，一次测试通常只针对功能、性能、兼容性这类特定方面，不可能一口气覆盖全部。

>

> 四个要素连起来，就是完整的测试活动：明确条件下运行 → 观察并记录结果 → 对特定方面做出评价。以后写用例、写测试报告，都可以拿它来检查有没有漏。
>
> **板书：**ISO/IEC 24765（IEEE、ISO/IEC 29119 口径）→ 四要素：① Activity 一项活动 ② executed under specified conditions 特定条件下运行 ③ results observed or recorded 观察或记录结果 ④ evaluation of some aspect 对某个方面做出评价

提问：这个定义里，哪个要素是很多同学做实验时最容易忽略的？漏了它会有什么后果？

> **预设回答：**①答"记录结果"——最常见，追问后果：没有记录就无法复现、无法证明缺陷存在，开发一句"我这边没问题"就能把缺陷打回去。②答"特定条件"——也很好，追问：条件不写清楚，别人照你的用例跑不出同样结果，用例就没有可重复性。③答"做出评价"——提醒：只报"点了一下没反应"是现象，不是评价，要说清"这违反了哪条需求、严重程度如何"。点评：把要素落到课程设计的具体动作上——用例要有前置条件、步骤、预期结果、实际结果、截图和结论。

易错点 / 考点：易错是把定义背成"运行软件找错误"，漏掉"特定条件""记录""评价"三个要素。考点：定义的填空与要素识别；判断某段工作是否属于测试活动（如"只运行不记录、不评价"——不完全符合定义）。记忆四要素口诀：有条件、要运行、留记录、给评价。

> **过渡：**这条定义讲的是"运行和评价"，下一页的续讲，补上定义的另一半——测试到底在比较什么。

【1.38】软件测试 IEEE/ISO29119 的定义 – 续（8 分钟）

这一页是上一条定义的续，它回答了一个更本质的问题：测试在做比较的时候，到底拿什么跟什么比？

>

> 先看英文：Testing is comparing what the test item does with what it is expected to do. 翻译过来就是：测试就是把测试项实际做的事情，和它被期望做的事情进行比较。这句话是全章最该记住的一句话之一。它

> 把测试的动作抽象成了一个"比较"。

>

> 这个比较有两边。左边是"实际表现"——被测对象真实跑出来的行为，由运行环境、输入数据和代码逻辑决定，是客观、可观察的。右边是"期望表现"——需求规格、设计文档、接口契约、用户预期所规定它应该有的行为。测试要做的，就是设法让这两边对齐，然后找出它们的差。

>

> 第二条表述更正式：分析某个软件项，以发现现存的和要求的条件之差别（即错误），并评价此软件项的特性。请注意这里的括号——"差别（即错误）"。也就是说，缺陷的定义可以直接从这句话推出来：所谓错误，就是实际的条件和要求的条件之间的差别。这一下就把"缺陷"的判定标准讲清楚了：有差别才有缺陷，没有要求就没有差别，也就无从判定缺陷。这就是为什么需求不清楚的项目没法测——你连比较的右边都画不出来。

>

> 还有一个词不能漏：评价此软件项的特性。前面说找差别，这里说评特性。说明测试的输出有两类：一类是具体缺陷，一类是对特性水平的整体评价，比如性能达到什么量级、可靠性处于什么水平。它们对应两种报告：缺陷单和质量报告。

>

> 最后强调一点：既然是比较，测试的严谨程度就取决于两块——测得够不够全（左边取样够不够），拿来比的标准对不对（右边尺子准不准）。这两条正是后面要展开的测试用例设计与测试需求分析。
>
> **板书：**Testing＝comparing what the test item does（实际）with what it is expected to do（期望）→ 分析软件项，发现现存的条件与要求的条件之差别（即错误），并评价此软件项的特性 → 缺陷＝实际与要求的差别 → 输出：缺陷单＋质量报告

> **提问：**按照"测试是实际与期望的比较"这一定义，如果一个项目的需求文档写得很笼统，会直接导致什么后果？

> **预设回答：**①答"测试做不了"——方向对，要具体化：没有明确的"要求条件"，就没有比较的右边，无法判定通过或失败，测试结论只能靠个人感觉。②答"那就按开发说的算"——纠偏：拿开发的口头说明当标准，等于让被测方给自己定标准，独立性丧失。③答"先测着，发现问题再说"——追问风险：漏掉的问题无从发

> 现，也无法度量测试充分性。点评：这正是测试人员必须尽早参与需求评审的原因——把不可判定、二义性的需求在评审阶段挑出来。

> **易错点 / 考点：**易错是把"缺陷"理解成"程序崩溃"这类明显故障，漏掉"与要求不符"这个判定核心（界面与设计稿差两像素、日志格式不符要求，同样是缺陷）。考点：由定义推导缺陷定义（简答）；"没有需求文档能不能测"（判断／案例）。判断三步法：找到"要求"→ 找到"实际"→ 两者有差别即缺陷。

> **过渡：**定义和比较都清楚了，最后我们回答一个务实的问题：投入这么多做测试，究竟换回了什么？

【1.39】软件测试的价值（8 分钟）

> 这一页是本节前一部分的收口：软件测试的价值。课件给了四条，我一条一条讲，并且告诉你们每一条在实际项目里对应什么动作。

>

> 第一条，全面评估产品质量，获得有关产品质量的全面、客观的信息。注意"全面"和"客观"这两个词。全面，是说测试不能只盯功能，还要看性能、兼容性、易用性、可靠性这些方面；客观，是说结论要有数据支撑，不能凭感觉说"我觉得差不多"。这一条对应的产出，是测试报告里的质量评估部分。

>

> 第二条，发现问题，督促问题解决，提高产品质量。这一条强调"督促"。测试人员找到缺陷只是第一步，缺陷不修，测试的价值就等于零。所以测试工作里有一大块是缺陷跟踪：确认缺陷被受理、被修复、被验证关闭。这也是我们课程设计要求大家写缺陷单、跟踪缺陷状态的原因。

>

> 第三条，持续提供质量反馈、及时揭示质量风险，有助于控制项目风险，提高构建的质量。这一条最容易被忽略，但项目经理最看重。软件测试像什么？像汽车仪表盘。它不能替你开车，但它告诉你油量、水温、车速，让你在抛锚之前做出决策。测试就是在每个版本、每个迭代，把"当前质量到了什么程度、哪些模块风险高"这个信息交给决策者。没有这条信息，项目管理者就是在盲开。

>

> 第四条，通过缺陷分析，获得缺陷模式，有助于缺陷预防。这是最高层次的价值：把已经发生的缺陷归类、

统计、找规律——比如某一类缺陷反复出现在某几个模块，或者集中在某个开发阶段——然后回头去改流程，让这一类缺陷不再产生。发现缺陷是治已病，缺陷预防是治未病。

>

四条连起来看，测试的价值是层层递进的：给信息 → 促修复 → 控风险 → 防缺陷。这也正好呼应了我们前面讲的学科发展四个阶段，说明测试早就不只是"最后一道检验工序"了。

板书：软件测试的价值：①全面评估产品质量，获得全面、客观的质量信息 ②发现问题、督促解决、提高质量 ③持续质量反馈、及时揭示风险、控制项目风险、提高构建质量 ④缺陷分析→缺陷模式→缺陷预防 → 层层递进：给信息→促修复→控风险→防缺陷

提问：这四条价值里，哪一条最容易被团队忽略，但对项目风险控制最关键？为什么？

预设回答：①答"第三条，持续提供质量反馈"——正确，追问：如果测试只在最后出一份报告会怎样？项目管理者在中间做不了决策，风险只能到最后一起爆。②答"第二条，督促问题解决"——可以讨论，补充：督促偏"事后"，而第三条是"过程性"的，价值在于及时。③答"第一条，评估质量"——提醒：评估是基础，但如果没有持续反馈，评估结果来得太晚，只能"验尸"。点评：测试真正的价值是让风险早暴露，而不是让报告更漂亮。

易错点 / 考点：易错是把"测试的价值"说成"保证软件没有缺陷"——测试不保证无缺陷，它提供质量信息、降低风险。考点：四条价值的记忆与排序（简答）；结合案例说明缺陷预防的价值。记忆口诀：评质量、促修复、控风险、防缺陷。

【思政落点 2】：讲国产游戏《血狮》（1997）——宣传声势远超其实际被验证的质量，交付后口碑崩塌；再对照波音星际客机因测试未做到端到端而出问题。两个方向一个共同点：把"没测到"当成了"没问题"。联系知识点：测试的价值在于四点，而缺陷预防是最高一层。落到价值观：测试是替用户把最后一道关，如实记录缺陷、不隐瞒不利结果，是从业者的职业底线与质量责任；家国情怀要靠过硬的质量来承载。

过渡：到这里，"测试是什么、为什么值得做"已经讲清楚了。带着这四条价值，我们从"为什么测"转向"怎么测"，继续沿着教材的思路往下展开。

【1.40】1.3.4 软件测试的其它观点（2 分钟）

前面几页我们走了三步：1.3.1 看软件测试这门学科是怎么形成的，1.3.2 看围绕测试的正反两方面的争辩，1.3.3 给出软件测试的定义。到了这一页，1.3.4，我们要接着追一个问题：定义有了，那能不能从更多角度把

测试看得更立体一点？答案是能。这一节给大家五个视角，等于从五个不同的坡面去爬同一座山。第一，质量视角，测试是对软件质量进行全面评估；第二，风险视角，测试是对潜在的各种质量风险进行评估；第三，经济视角，测试是用最小的代价换最高的软件产品质量；第四，基于 Test Oracle 的认知，判断输出结果对不对，得有判断准则；第五，基于批判性思维的认知，测试是借助观察、经验、反思、推理和沟通来收集信息、做出结论的不断探索的过程。有同学会问：定义不就够了吗，为什么要费劲讲五个视角？因为定义是一句话，观点是一整套看问题的方式。你在企业里跟人讨论"这个版本要不要多测一轮"，只背定义是说服不了人的：跟项目经理要讲风险，跟老板要讲成本，跟开发要讲判断准则，跟客户要讲质量。不同角色的语言不一样，测试人员必须会切换语言。打个比方，同一辆车，司机关心好不好开，修车师傅关心哪里容易坏，保险公司关心出事故的概率，二手车商关心值多少钱——车是同一辆，问题不同，结论自然不同。还要提醒一句：这五个视角不是五套互相打架的说法，而是互补的。质量视角回答"测什么"，风险视角回答"先测什么"，经济视角回答"值不值得测"，Test Oracle 回答"怎么判定对错"，批判性思维回答"用什么态度去测"。考试爱考"观点与核心主张"的对应，给你一句话让你判断它属于哪个视角，所以接下来每一页，请重点抓那句话的关键词。

板书：1.3.4 其它观点 → 质量视角｜风险视角｜经济视角｜Test Oracle｜批判性思维 → 互补、不冲突

提问：如果项目经理说"这个版本时间很紧，少测一点"，五个视角里你更可能用哪一个去跟他沟通？为什么？

预设回答：①答"风险视角"——正确，且能说出"按风险排序，砍掉低风险项而不是平均砍"。教师点评：方向对，要追问一句"那你怎么证明哪些是低风险"，逼学生给出风险判断的依据。②答"质量视角"——只对了一半。教师追问：你讲质量，项目经理会说"质量我当然要，但时间不够"，这句话没有说服力；质量视角讲的是"测什么"，不是"砍多少"。③答"经济视角也行"——可以接受，但要提醒：经济视角解决的是"这笔测试投入值不值"，是用成本收益算账，跟"先测哪块"是两个问题。把三种回答并排写在黑板上，正好演示五个视角各管一段，不是随便挑一个都行。

易错点 / 考点：最典型的错，是把五个视角背成五句口号，却分不清谁回答哪个问题。判断三步法：先看这句话在谈"测什么"还是"先测什么"还是"值不值"；再看它有没有提到判断准则；最后看它是不是在讲收集信息、做结论的过程。选择题常考"从经济视角看软件测试的核心是什么"这类对应题，判断题常把"测试就是找Bug"当正确项来设置陷阱。

过渡：五个视角，我们从最容易被挂在嘴边、也最容易被理解偏的一个开始——质量视角。

【1.41】从质量视角认知软件测试（4 分钟）

这一页进入质量视角。课件上那句话是：软件测试被认为是对软件质量进行全面评估的活动，给出质量信息，

从而确定质量是否满足设计和用户的需求。请抓住三个关键词：全面评估、给出质量信息、满足需求。第一层，测试是一项评估活动，不是"找茬活动"。大家看这张图，中间画着"发现 Bug？"，后面专门带了一个问号，这就是在提醒我们：测试的目的不只是找 Bug，而是给出质量信息，让决策者知道这个软件现在到底能不能交付。你把 Bug 报上去，开发改完，测试再验，这一整套动作的最终产出，是一份关于质量的判断，而不是一堆缺陷单。第二层，为什么会有缺陷？看图的下半部分：左边是用户，带着"要求/期望"；右边是质量，也就是实际做出来的东西；中间标着"矛盾/对立"。软件缺陷的本质，就是做出来的东西和用户要求的、期望的东西之间出现了落差。这句话很重要，它解释了后面为什么要做需求评审、为什么要写验收标准——因为只有当"要求和期望"被写清楚，落差才可判定。第三层，靠什么评估？图里写了"质量模型"。质量模型就是一把尺子。国际上常用的软件质量模型，会把质量拆成功能适合性、性能效率、兼容性、易用性、可靠性、安全性、可维护性、可移植性这些特性。没有这把尺子，"质量好"就是一句空话，A 说挺好，B 说不行，吵到最后谁嗓门大谁赢。生活里的道理是一样的：体检报告不会只写"你身体不错"，它给出血压、血糖、血脂这些指标，让医生和你都能对比、能追踪。测试报告也一样，要给出可对比、可追踪的质量信息。为什么这个视角重要？因为太多团队的测试报告只写一句"测试通过，未发现严重问题"，这句话对决策几乎没有价值——它是结论，不是信息。真正有用的写法是：测了哪些质量特性、覆盖了多少条需求、剩余风险集中在哪里。这一页还埋下了后面反复要用的三个概念：需求是测试的依据，质量特性是测试的维度，评估结论必须建立在证据之上。

板书：质量视角 → 全面评估 + 给出质量信息 → 用户"要求/期望" ↔ 质量（矛盾/对立）→ 评估靠"质量模型" → 发现 Bug？（不只是找 Bug）→ 需求是依据、质量特性是维度、结论靠证据

提问：课件的图里，"发现 Bug"后面为什么要加一个问号？如果测试只以"找到 Bug"为目的，会漏掉什么？

预设回答：①答"因为测试的目的不只是找 Bug"——正确。教师追问：那除了找 Bug，测试还产出什么？引导学生说出"质量信息、能否交付的判断、剩余风险"。②答"因为不是每次都能找到 Bug"——这是典型错误，把目的和结果搞混了。教师点评：找不到 Bug 的那一轮测试一样有价值，它提供了"当前版本在这些范围内没有发现问题"的信息，这是发版决策的证据；要找 Bug 才做测试，等于把测试当成抽奖。③答"因为 Bug 这个词有歧义，需求理解错了也算 Bug"——有深度，可以顺势引到下一句"矛盾/对立"：需求本身有歧义时，判断对错的标准就不牢靠，所以要提前评审需求。三种回答按"结论—证据—标准"三层排一下，学生就明白为什么测试报告要写信息而不是只写结论。

易错点 / 考点：两个混淆点。第一，把"测试"等同于"找 Bug"，忽略"给出质量信息"，这是最常见的选择题陷阱。第二，把质量模型当成考试名词硬背，却不会用它组织测试：考案例题时，要求你针对某个系统列出该

测哪些质量特性，只写"功能"两个字是拿不到分的。记忆口诀：要评估、给信息、比需求、靠模型。

过渡：质量视角回答的是"测什么"，但它回答不了"先测什么"。要回答这个问题，我们得换到风险视角去看。

【1.42】从风险视角认知软件测试（4 分钟）

这一页讲风险视角。课件上写着：软件测试被认为是对软件系统中潜在的各种质量风险进行评估的活动。后面还有一句非常关键的话：测试是样本实验而不能穷尽，其风险总是存在的。再往下，是"基于风险的测试"这个说法——它强调对软件开发的全过程进行检测，随时发现问题、报告问题，减少对客户不利影响的风险。我们先拆"测试是样本实验而不能穷尽"。这句话的意思是：你不可能把用户所有可能的输入、所有可能的操作顺序、所有可能的运行环境都试一遍。输入组合是爆炸式增长的，一个几百行的小程序，穷举路径都做不到，何况一个几十万行的系统。既然测不完，那就必须选择测什么——这就是样本实验。既然是抽样，就一定会有漏网之鱼，所以风险永远存在，测试的目标不是"消灭风险"，而是把风险压到可以接受的水平。这就引出基于风险的测试：哪里出事概率高、出了事后果重，就优先测哪里。风险大致等于"发生的可能性"乘以"一旦发生的损失"。举个身边的例子：一个电商系统，支付和退款模块一旦出错就是真金白银的损失，而"关于我们"页面的错别字损失极小。时间只够测一半的时候，先测支付，这叫按风险排序，不叫偷懒。再看图上这条链：上面是需求、设计、代码，中间是功能和非功能特性，往下是测试，最后到交付；左边站着产品经理、项目经理和开发人员，图里向下的箭头旁边写着"持续反馈质量风险"，而且是两处，意思是从头到尾一直在反馈。这一点是理解本页的钥匙：风险不是等到测试阶段才暴露，需求写得含糊、设计接口对不上、代码没有异常处理，这些都是风险源，测试人员要尽早介入，随时把风险信息反馈回去。所以风险视角下的测试，是贯穿开发全过程的一项检测活动，不等同于最后阶段的验证。对客户来说，提前发现一个高风险问题，可能就避免了一次线上事故、一次口碑崩塌。

板书：风险视角 → 测试＝样本实验，不能穷尽 → 风险＝可能性 × 影响 → 基于风险的测试：全过程检测、随时发现、随时报告 → 需求→设计→代码→（功能/非功能特性）→测试→交付，向上"持续反馈质量风险"

提问：同样的测试时间，为什么"平均分配给每个功能模块"是一种不负责任的做法？请举一个你熟悉的系统来说明。

预设回答：①答"因为有的模块出问题损失大，平均分配等于高风险的地方没测够"——正确，能说出概率和影响两个维度更好。教师点评：让他具体说出"哪个模块、什么后果"，把抽象原则落到场景。②答"因为测试时间不够，从来测不完"——只答对了一半，这只解释了为什么要选择，没解释为什么不能平均。教师追问：那如果某个模块又简单又不重要，你给它同样的人力，损失的是什么？引导学生说出"机会成本"。③

答"平均分配没什么不好，覆盖面广"——这是很典型的错误直觉。教师点评：覆盖面广不等于风险低，覆盖了十个不重要功能，不如覆盖一个支付功能；把这句话写在黑板上，作为"风险排序"的反面教材，再让学生把它改写成"按风险×影响排序"。

易错点 / 考点：第一，把"基于风险的测试"误解成"高风险模块多测几轮"这么简单，漏掉"全过程检测、持续反馈"这一半。第二，误以为测试可以做到零风险、穷尽验证，判断题常拿"只要测试足够充分就能保证软件无缺陷"来设坑，答案是错的。判断三步法：先问"能不能穷尽"（不能），再问"先测哪里"（风险高的），最后问"风险从哪来"（需求、设计、代码全过程）。

过渡：既然测不完，就必须做取舍；而只要谈取舍，就绕不开钱。接下来我们从经济视角算一算这笔账。

【1.43】从经济视角认知软件测试（3 分钟）

这一页从经济视角看测试。课件原话是这样四句：测试的经济观点就是以最小的代价获得最高的软件产品质量；经济观点也要求软件测试尽早开展工作；发现缺陷越早，返工的工作量就越小，所造成的损失就越小；还有一句是最扎心的——测试的成本要小于缺陷造成的损失，测试才有意义。我们一句一句来。第一句，"以最小的代价获得最高的质量"，注意不是"不计代价追求零缺陷"。测试是要花钱的：要人、要时间、要环境、要机器。如果为了一百万分之一的概率去投入巨大成本，那这笔钱花得不值。所以测试从来不是"测得越多越好"，而是"花得值"。第二句和第三句连起来看，是这一页的核心：为什么要尽早测？因为发现得越早，返工的工作量越小，损失越小。缺陷像滚雪球，需求阶段一句话的含糊，到了设计就变成接口定义错误，到了编码就变成一片实现，到了测试甚至上线才暴露，这时候要改的不只是一个字，而是设计、代码、文档、数据，甚至已经交付给用户的产品。生活里的类比很好懂：牙上有个小黑点，早去看，补一下就好；拖到牙神经坏了，就要根管治疗加牙冠，钱和罪都翻好几倍。汽车刚有异响就去检查，也许只是换个垫片；拖到高速上抛锚，损失的就是拖车费、误工费甚至安全。软件行业最典型的例子，就是英特尔当年奔腾处理器的浮点除法缺陷，据本课程案例素材，最后召回换货付出的代价在四亿美元这个量级。一个除法算错的细节，最后变成了一场大规模的召回。第四句给了我们一把决策尺子：测试的成本小于缺陷造成的损失，测试才有意义。反过来，如果某类缺陷造成的损失远小于测试它的成本，那就要重新考虑测到什么程度。这也解释了为什么测试要有优先级、要做风险排序——钱要花在刀刃上。同学们以后写测试计划，里面一定会有"范围和取舍"，那时你要能说清：我为什么测这些，为什么这一段先测，为什么那一段只做抽样。

板书：经济视角 → 最小代价 × 最高质量 → 尽早测试：发现越早，返工越小、损失越小 → 决策尺子：测试成本 < 缺陷损失，测试才有意义 → 测试不是"越多越好"，是"花得值"

提问：有人说"质量是免费的"。结合这一页的经济视角，你同意吗？如果不完全同意，请把这句话补完整。

预设回答：①答"不完全同意，预防的投入是必要的，但质量不是零成本"——很好。教师点评：这话的原始含义是"把缺陷挡在前面比事后返工便宜"，而不是"一分钱不花就有高质量"。②答"同意，做好质量就不需要返工"——典型错误。教师追问：那测试人员、测试环境、回归测试的机器要不要钱？引导学生区分"预防成本低"和"成本为零"。③答"只有测试成本小于损失才值得测"——抓住了课件的尺子。教师点评：这句话是对的，但要追问边界——损失怎么估？让学生意识到"损失估算"本身就是一件需要判断的事，不能拍脑袋。

易错点 / 考点：常见错误有两个。一是把经济视角读成"少测省钱"，其实它的落脚点是"尽早测"和"算总账"。二是忽略"测试也是有成本的"，答案例题时只写"要测得更充分"，不提成本与收益的平衡。考试常考判断题："缺陷发现得越晚，修复成本越高"——正确；"测试投入越多，质量一定越高"——错误。记忆口诀：早发现、小返工、算总账。

【思政落点 43】讲什么故事：课件这句"测试成本要小于缺陷造成的损失，测试才有意义"，背后是企业实实在在的钱，也是用户实实在在的安全。怎么联系知识点：经济视角不是教大家省钱省测试，恰恰相反，它告诉我们缺陷一旦流到用户手里，代价会成倍放大——英特尔奔腾的浮点除法错误，最后是四亿美元量级的召回。落到什么价值观：工程人员的每一次"这个先不测了，赶进度"，都不是一句轻飘飘的话，最终买单的是企业和用户；把缺陷挡在交付之前，既是职业能力，也是质量责任的底线。

过渡：经济视角告诉我们要算总账，可要算账，前提是能判断"这一段结果到底对不对"。这就引出了 Test Oracle 的认知。

【1.44】基于 Test Oracle 的认知（3 分钟）

这一页很短，字也很少，但它是整个测试活动的地基。课件上只有一句话：输出结果是否正确，需要判断准则。图也只有一张，画的就是这个意思。什么叫判断准则？就是当你把一组输入喂给程序、程序吐出一个结果的时候，你凭什么说这个结果是对的。这个"凭什么"，在测试领域有一个专门的术语，叫 Test Oracle，中文一般翻译成测试预言或测试判定准则。为什么它这么重要？因为没有 Oracle，测试就变成了"看着像对的就算对"。举个最简单的例子：一个计算器程序，你输入 2 加 3，它输出 5，你说对；如果它输出 6，你说错。你的判断准则来自小学数学，这就是 Oracle。再比如登录功能：输入正确的账号密码，期望结果是登录成功、跳转到首页、并且页面右上角显示用户名。这一串"期望结果"，才是你的判断准则。如果只写"登录成功"，那系统提示"登录成功"却把你导到一个空白页，你算不算通过？没有准则，就判不了。那 Oracle 从哪里来？常见的有几类：需求规格和验收标准，这是最正规的来源；业务规则和领域知识，比如财务系统里借贷必须

相等；旧系统的实际行为，做系统改造时拿老系统当参照；已有的参考实现或者同类产品的表现；还有测试人员的人工经验判断。这里要提醒一个现实困难，业内叫"Oracle 问题"：有些输出很难有唯一的正确标准。比如界面的美观程度、语音识别的结果质量、算法给用户推荐得合不合理，甚至是今天大量的 AI 输出——这些地方"对"的边界很模糊。遇到这种情况，通常的做法是把它转化成可判定的形式：把美观转成设计规范的检查项，把推荐质量转成一批标注好的样本和一条通过线，或者引入多人评审。所以这一页想让大家记住的是：测试不是"跑一遍看看"，测试是先有准则、再执行、最后比对。你在本课程的实验里写测试用例，表头一定会有"预期结果"这一列，那一列就是你的 Oracle，它不是形式主义，它是你能不能判定通过的依据。

板书：Test Oracle → 输出结果是否正确，需要判断准则 → 准则来源：需求规格/验收标准｜业务规则与领域知识｜旧系统行为｜参考实现｜人工经验 → 有准则才能判"通过/不通过" → 难判定的输出要转化为可判定形式

提问：测试用例表里"预期结果"这一列，如果空着或者写成"正常显示"，会带来什么后果？

预设回答：①答"没法判断测试通过还是失败，两个人可能得出两个结论"——正确。教师点评：再追一步，这种不一致在团队协作里会导致什么？引导到"缺陷争议、返工、回归标准不统一"。②答"没关系，执行的人自己知道对错"——典型错误。教师点评：写用例的人知道，不代表执行的人知道，更不代表三个月后回归测试的人知道；用例是给团队用的，不是给个人备忘的。③答"可以事后补上"——可接受但不是好做法。教师追问：如果开发改完代码，你又补了新的预期结果，那这次改动到底算通过还是没通过？引导学生理解：Oracle 是先于执行确定的，事后补等于给结果找解释。

易错点 / 考点：第一，把 Test Oracle 当成某个工具或某个测试类型，其实它是一个概念——判定准则。第二，写用例时把预期结果写成"结果正确""正常"这类无法判定的表述，这是实验和课程设计里最常见的扣分点。第三，误以为 Oracle 只能来自需求文档；事实上领域知识、旧系统、参考实现、人工经验都可以，但来源越客观，争议越小。判断口诀：先有准则、再执行、后比对。

过渡：有了准则，还差一样东西——态度。因为再好的准则，也挡不住"我觉得没问题"这种随意判断。

【1.45】基于批判性思维的认知（3 分钟）

这一页把测试和一种思维方式绑在一起。课件上是这么说的：软件测试就是借助观察、经验、反思、推理或沟通等收集信息，并对软件产品相关的质量信息进行分析，以此评估软件质量，并做出结论。后面还跟着一句短语：不断探索的过程。我们先看前半句里的五个手段：观察、经验、反思、推理、沟通。观察，是看现象——点下去有没有反应、返回的数据对不对、界面有没有错位；经验，是你见过类似的系统怎么出错，比

如日期、金额、并发；反思，是回头想"我刚才这个判断是不是太快了""这里没出问题是不是因为我根本没测到"；推理，是从已知推未知——这个输入框限制了长度，那粘贴超长文本会怎样；沟通，是找开发、产品、用户把需求问清楚。这五种手段合起来，是在收集信息，而不是在收集感觉。后半句"做出结论"，说明测试最终要给出判断，但判断必须建立在分析过的质量信息之上。那"批判性思维"体现在哪里？体现在一个根本立场：不轻信。程序说"保存成功"，不一定真保存了，要去数据库里核实；界面显示"操作已完成"，不一定真完成，要看状态有没有变；开发说"这个功能没问题"，那就要用证据来确认。这跟前面 risk 视角里那句"测试是样本实验"是一脉相承的。反过来，如果测试人员的思维方式是"我写的用例都跑过了，应该没问题"，那就从批判性思维滑回了自我安慰。这也是本页两个图要表达的对照：一个是不断提出疑问、不断探索的循环，一个是一旦停止怀疑就停止发现缺陷。生活里我们其实每天都在用批判性思维：买东西看配料表，不听广告听成分；看到"限时特价"先想是不是原价就那样。做测试就是把这种较真的习惯，变成一种有方法、有记录、有结论的工作方式。请记住这句话：测试人员不是软件的敌人，但一定是"未经证实的结论"的敌人。

板书：批判性思维 → 观察｜经验｜反思｜推理｜沟通＝收集信息 → 分析质量信息 → 评估质量、做出结论 → 不断探索的过程 → 核心立场：不轻信、要证据

提问：界面上弹出"保存成功"，为什么还不能直接判定这个功能测试通过？你会再去看什么？

预设回答：①答"要去数据库或列表里查一下数据真的写进去了"——正确。教师点评：这就是观察加推理，把界面提示当作线索而不是结论。②答"把提示截个图，用例就算过了"——典型错误，属于只验证了现象、没验证结果。教师追问：如果程序只弹了提示、实际什么都没存，你这条用例是不是漏掉了缺陷？③答"看开发怎么说"——不可取。教师点评：沟通是手段之一，但沟通用来澄清需求，不能用来替代验证；开发说存了，你仍然要自己看到证据。

易错点 / 考点：第一，把批判性思维理解成"跟开发抬杠""什么都怀疑"，其实它的落点是收集信息、分析信息、做出有依据的结论。第二，把"探索"当成漫无目的地乱点，忽略课件列的五个手段和后续的分析动作。第三，考试里常把"测试是证明程序没有错误"当正确项，这是反的——测试是带着怀疑去求证，不是去证明"我写对了"。记忆口诀：不轻信、找证据、做结论。

【思政落点 45】讲什么故事：批判性思维这一页，考验的其实是一个人的诚信与科学态度。怎么联系知识点：课件要求测试借助观察、经验、反思、推理、沟通去收集信息，再基于信息做出结论——这就是用证据说话。落到什么价值观：发现缺陷不报告、为了赶进度隐去不利结果、把"没测"写成"通过"，都是对事实的背叛；如实记录、如实评价，是测试人员最基本的职业操守，也是我们这门课过程性考核看重的态度。

过渡：到这里，软件的"其它观点"讲完了。接下来要处理一个在课堂上和工作中都极易混淆的关系——测试

和质量保证到底是什么关系。

【1.46】1.4 测试与质量保证的关系（2 分钟）

这一页是 1.4 节的开头，标题就是本节要回答的问题：测试与质量保证的关系。大家先看这张图，它把本节要建立的框架先摆出来了。为什么单独拿一节来讲这个关系？因为这是最容易被混为一谈的一对概念。很多同学一听说"质量保证"，脑子里想的就是"测试"；也有同学以为"测试"和"质量保证"是两个各自为政的部门。这两种理解都会出问题：前一种会导致你把质量保证做成了事后检测，后一种会导致两边互相推责任——测试说"你怎么没保证质量"，质量保证说"我提醒过你，是你没测出来"。我们先看课件备注里列的内容：1.3.1 软件测试学科的形成、1.3.2 正反两方面的争辩、1.3.3 软件测试的定义、1.3.4 软件测试的其它观点——这是刚才这一大段的回顾，说明我们前面的讨论都还在 1.3 节里。现在转进 1.4，接下来三页要依次回答三个问题：第一，什么是 SQA，也就是软件质量保证；第二，SQA 到底做哪些活动；第三，软件测试和 SQA 之间谁包含谁、谁指导谁、谁给谁提供数据。这三个问题的答案，都会在实验报告和课程设计文档里用到——比如你写测试计划，如果分不清"测试活动"和"SQA 活动"，范围就会写乱；你写质量报告，如果分不清"我提供数据"和"我负责整个流程"，职责就会越界。所以这一节虽然概念性强，但它是后面所有文档写作的语法。最后把本节最容易混的一点先点出来：以后做题，凡是问"谁负责流程与规范"的，答案指向质量保证；凡是问"谁给出产品的验证结论"的，答案指向测试。这两个答案千万不要对调。

板书：1.4 测试与质量保证的关系 → 三问：什么是 SQA？SQA 有哪些活动？测试与 SQA 谁包含谁？→ 前面回顾：1.3.1 学科形成｜1.3.2 正反争辩｜1.3.3 定义｜1.3.4 其它观点

提问：你们觉得"测试"和"质量保证"是一回事吗？如果是两回事，差别在哪一句话上？

预设回答：①答"不是一回事，测试是查产品，质量保证是管流程"——很接近正确答案。教师点评：这句直觉很好，先记在黑板上，等第 49 页再严格对照。②答"是一回事，测试就是保证质量"——最常见的错误。教师追问：如果测试就是质量保证，那一个没有独立测试团队的公司，是不是就没有质量保证了？引导学生发现两者活动范围不同。③答"质量保证更大，包含测试"——方向对了。教师点评：正确但不完整，要补上"包含"之外还有"指导和监督"这层关系，留到第 49 页揭晓。

易错点 / 考点：本节的混淆点集中在层级关系上——软件测试是 SQA 的活动之一，但 SQA 不等于测试；"测试是 SQA 的重要手段"和"SQA 指导监督测试"是两句话、两个方向，不能只答一半。判断三步法：先看它说的是流程还是产品，再看它是给数据还是提要求，最后落到对应的那一栏。选择题常把两者对调来设坑，务必看清主语。

过渡：要把这个关系讲清楚，先得知道 SQA 本身是什么——下一页我们就给它一个正式的定义。

【1.47】什么是 SQA？（4 分钟）

这一页回答什么是 SQA。SQA 是软件质量保证的英文缩写，全称 Software Quality Assurance。课件给出了一个完整定义：软件质量保证活动是通过对软件产品有计划的进行评审和审计来验证软件是否合乎标准的系统工程，通过协调、审查和跟踪以获取有用信息，形成分析结果以指导软件过程。这个定义里有三个动作请画下来：有计划的评审、审计、以及验证是否合乎标准；还有一条目的：指导软件过程。注意最后这四个字——指导软件过程，它说明 SQA 关心的不只是"这个产品行不行"，更是"我们造产品的这套流程行不行"。围绕这个定义，课件又给了三句展开。第一句：对软件工程各个阶段的进展、完成质量及出现的问题进行评审、跟踪。注意"各个阶段"，从需求到设计到编码到测试到上线，每个阶段都要有检查点，而不是最后一次性大检查。第二句：审查和验证软件产品是否遵守适用的标准、规程和要求，并最终确保符合标准、满足要求。这里说的是"合规"，标准可能是国际标准、行业标准，也可能是公司自己定的开发规范和编码规范。第三句：建立软件质量要素的度量机制，了解各种指标的量化信息，向管理者提供可视信息。这一句最容易被忽略，但它非常实在——质量保证要拿出数据，比如缺陷密度、评审发现率、需求变更次数、用例通过率，用这些量化信息让管理者看清楚现状。把三句合起来看，SQA 的工作方式可以概括成三个词：定标准、查执行、报数据。生活里最好懂的例子是食品安全检查：不是等吃坏肚子才去查，而是从原料采购、生产车间、包装、出厂检验，一路都有标准和抽查记录。软件也一样，SQA 是给软件生产过程装的一套"质量体系"，测试只是这套体系里的一环。所以如果你以后做测试工程师，会经常收到 SQA 发来的评审通知、度量表格和规范更新；而如果你做质量保证，就要学会从测试数据里读出流程的问题，而不是只盯着缺陷数。

板书：SQA＝Software Quality Assurance 软件质量保证 → 定义：有计划地评审、审计，验证是否合乎标准 → 目的：获取信息、分析结果、指导软件过程 → 三句展开：①各阶段进展/质量/问题 评审跟踪 ②审查验证是否遵守标准、规程、要求 ③建立度量机制，向管理者提供可视信息 → 三词概括：定标准、查执行、报数据

提问："向管理者提供可视信息"为什么也要算 SQA 的工作？只提交一份缺陷清单不行吗？

预设回答：①答"不行，管理者要做决策，需要看趋势和量化指标，不只是零散的问题"——正确。教师点评：顺势点出缺陷密度、通过率这类指标比单条缺陷更能反映流程状态。②答"因为管理者看不懂技术细节"——有点道理但角度偏了。教师点评：不是看不懂，而是管理决策需要的是"趋势、对比、风险"，单条缺陷回答不了"这个版本能不能发"。③答"这是形式主义，交清单就够"——典型错误。教师追问：如果缺陷有 200 条但都集中在同一个模块，清单能告诉你什么？引导学生说出"集中度、根因、流程改进方向"。

易错点 / 考点：第一，把 SQA 缩写成"质量检测"，忽略它是过程性的、计划性的系统工程。第二，只记住定义，答不出 SQA 的三个工作面向（评审跟踪、合规验证、度量报告）。第三，考试中常出现"以下哪项不属于 SQA 的工作"，要把 SQA 的度量、评审、审计、标准执行和"具体写代码"区分开。记忆口诀：定标准、查执行、报数据，目的是指导过程。

过渡：定义讲完了，那 SQA 具体要做哪些活动呢？下一页列出了七项。

【1.48】SQA 活动（3 分钟）

这一页列出 SQA 的具体活动，一共七项，我们一项一项过。第一，技术方法的应用。就是说开发过程要用对方法和工具，用什么开发方法、什么设计方法、什么建模手段，SQA 会关注这些方法有没有被正确使用，因为它直接影响质量。第二，正式技术评审的实施。注意"正式"两个字——它和随便找人看两眼完全不同，正式技术评审要有计划、有角色、有检查单、有记录、有结论。第三，软件测试。这一项请大家重点画下来。软件测试是 SQA 的活动之一，也就是说测试在质量保证的框架之内。第四，标准的执行。编码规范、文档模板、命名约定、配置管理要求，这些标准要落地执行，SQA 负责推动和检查。第五，修改的控制。也就是变更控制，代码、需求、文档的每一次修改都要走流程、要评估影响、要留痕。为什么这一项重要？因为大量缺陷不是"一开始就写错"，而是"改出来的"——修了 A 功能结果弄坏了 B 功能。第六，度量。要建立指标，收集数据，比如缺陷密度、评审效率、测试通过率。没有度量，改进就没有方向。第七，质量记录和记录保存。评审记录、测试报告、变更记录，这些都要归档保存，一方面是为了追溯，另一方面是为了体系审核时能提供证据。把这七项放在一起看，你会发现一个规律：它们大部分都不是"直接检查产品"，而是在检查"做事的方式"——方法对不对、评审做没做、标准守没守、变更管没管、数据有没有。只有第三项"软件测试"是直接对产品做验证和确认。这就解释了为什么测试是 SQA 的重要手段之一，但绝不是全部。同学们在做课程设计的时候，其实可以拿这七项当自查表：你的项目有没有走评审？代码有没有规范？需求变更有没有记录？测试数据有没有统计？测试报告有没有归档？这七问答下来，基本就知道一个团队的质量管理水平在哪里。也提醒一句，考试里非常喜欢问"软件测试是不是 SQA 的活动之一"，答案是肯定的，别答成"两回事"。

板书：SQA 活动七项 → ①技术方法的应用 ②正式技术评审的实施 ③软件测试 ④标准的执行 ⑤修改的控制 ⑥度量 ⑦质量记录和记录保存 → 多数查"做事方式"，第③项直接查产品 → 结论：测试是 SQA 的重要手段之一，不是全部

提问：这七项里，哪一项最容易被小团队省略？被省略之后，通常在什么时候付出代价？

预设回答：①答"质量记录和记录保存，或者度量"，也有人答"正式技术评审"——都合理。教师点评：请他

给出一个具体后果，比如没有记录，出了线上问题无法追溯是哪次改动引入的。②答"测试最容易被省"——不准确。教师点评：测试往往是小团队唯一保留的"质量动作"，恰恰是被裁到只剩它；真正的空白通常是评审、度量和记录。③答"七项都得做，一项都不能省"——态度可嘉但要引导到现实：资源有限时应该按风险取舍，但"不留记录"这种省法等于把可追溯性一起省掉了，代价最大。

易错点 / 考点：第一，漏项。七项里"修改的控制""度量""质量记录"最容易被忽略，考试多选常在这里设置正确项。第二，把"技术方法的应用"理解成"使用测试工具"，它的范围更宽，指开发全过程中技术方法的正确使用。第三，判断题常考"软件测试是 SQA 活动之一"——正确；"SQA 就是软件测试"——错误。记忆口诀：法（方法）评（评审）测（测试）标（标准）改（变更）度（度量）记（记录）。

过渡：七项活动里有测试，也有评审、标准、变更、度量。那测试在整个 SQA 里到底站在什么位置？下一页我们就正面对比。

【1.49】软件测试 vs. SQA（4 分钟）

这一页是本节的核心，把测试和 SQA 摆在一起对照。课件给了三层关系，我们逐层看。第一层：SQA 指导、监督软件测试的计划和执行，督促测试工作的结果客观、准确和有效，并协助测试流程的改进。请注意这三个词——客观、准确、有效。SQA 不替测试干活，它盯着测试干得对不对：你的测试计划有没有覆盖关键需求，你的测试结果是不是如实记录，你的测试流程有没有可以改进的地方。第二层：软件测试是 SQA 重要手段之一，为 SQA 提供所需的数据，作为质量评价的客观依据。也就是说，SQA 要判断"这批产品的质量行不行""这条流程改得对不对"，它靠什么下判断？靠测试给的数据。测试是 SQA 的眼睛和手。第三层是最容易考的对照：SQA 是一项管理工作，侧重于对流程的评审和监控；测试是一项技术性的工作，侧重对产品进行评估和验证。一个管过程，一个验产品；一个关注"我们是怎么做的"，一个关注"做出来的东西对不对"。把三层连起来记，就是一句话：SQA 给测试定规矩、看规矩有没有被守，测试给 SQA 交数据、当质量判断的依据。这里要澄清两个常见误解。第一个误解是"测试做得越多，SQA 就越好"。不对——测试发现了一大堆缺陷，恰恰说明前面的需求、设计、评审环节有问题，SQA 该做的是去改进流程，而不是庆祝测试团队发现的缺陷多。第二个误解是"有了 SQA 就不用测试，或者有了测试就不需要 SQA"。也不对——没有测试，SQA 的结论没有客观数据支撑，只能靠感觉；没有 SQA，测试容易变成自说自话，标准不统一、结果不可比、流程不改进。这一页的知识在你们的课程设计里会直接用到：写测试计划时，范围、准则、记录方式其实都受质量保证要求约束；写质量报告时，你要清楚自己提供的是"数据与验证结论"，流程评审和改进建议属于另一条线。分得清这两条线，文档就不会写串。

板书：软件测试 vs. SQA → SQA 指导、监督测试的计划与执行（客观、准确、有效）→ 测试是 SQA 重要手

段之一，为 SQA 提供数据 → SQA＝管理工作，侧重流程的评审与监控；测试＝技术工作，侧重产品的评估与验证 → 一句话：SQA 定规矩看规矩，测试交数据当依据

提问：如果某个版本测试发现缺陷特别多，是先表扬测试团队，还是先检查流程？为什么？

预设回答：①答"先检查流程，说明前面环节没拦住缺陷"——正确。教师点评：追一句"那测试团队该不该表扬"，引导学生区分"发现缺陷有价值"和"缺陷多说明前期失控"两件事。②答"表扬测试，说明测试认真"——情绪上合理，但漏了重点。教师追问：如果需求评审做扎实了，这批缺陷里有多少根本不会产生？让学生体会"质量是设计出来的，不是测出来的"。③答"两个都要查"——稳妥，可以接受。教师点评：这个回答不错，但要问清优先级：先看缺陷的引入阶段分布，才能判断是流程问题还是个别疏忽。

易错点 / 考点：第一，把两者的关系答成"包含关系"就停手，漏掉"指导、监督、提供数据"三层互动。第二，答选择题时把"SQA 侧重产品验证、测试侧重流程监控"选成正确项——这是把两者对调了，属于高频陷阱。第三，判断题"SQA 就是测试部门""测试人员就是 SQA 人员"都是错的。记忆口诀：SQA 管过程，测试验产品；SQA 给测试定规矩，测试给 SQA 交数据。

【思政落点 49】讲什么故事：课件特意强调 SQA 要"督促测试工作的结果客观、准确和有效"，这句话说的是监督，落点其实是诚信。怎么联系知识点：测试向 SQA 提供质量评价的客观依据，数据一旦被修饰，整个质量判断就失去意义。落到什么价值观：真实是质量工作的生命线。企业里为了过一个质量门而修改测试结论、遮掩失败用例，短期看是"顺利交付"，长期看是拿用户和公司信誉做赌注。我们做课程设计也一样，测出来的问题如实写、测不到的地方如实标明范围，这才是合格的工程素养。

过渡：说清了测试和质量保证的关系，还有一个关系同样重要，而且更贴近日常——测试和开发的关系。

【1.50】1.5 测试与开发的关系（2 分钟）

这一页进入 1.5 节，标题是测试与开发的关系。同样先放一张图，把这一节的问题摆出来。为什么这个关系也值得单独讲一节？因为它直接决定测试在项目里被摆在什么位置。在不少团队和不少人的直觉里，开发是"造东西的"，测试是"检查东西的"，造完了才轮到检查，所以测试天然排在开发后面。这种想法听起来很自然，实际代价非常大：测试时间被排在最后，一旦前面延期，被压缩的永远是测试；缺陷暴露得晚，返工成本高；测试成了"把关的唯一一道墙"，压力大、效果差。这一节要纠正的就是这个排序。课件备注里仍然是刚才 1.3 节的四个小节作为回顾，说明现在我们正式转入 1.5。本节接下来用两页来对照：第一页讲"测试不是开发下一道工序"，把传统的那条串联链摆出来，让大家看清它的危害；第二页讲"测试与开发是并行、协作的关系"，给出正确的组织方式。这两页合起来，也是本课程后面很多内容的思想基础——比如为什么要有测试左移，

为什么单元测试要由开发来写，为什么缺陷修复过程需要开发与测试反复沟通。请大家带着一个问题听：如果测试不是最后一道工序，那它在需求阶段、设计阶段应该做什么？这个问题在第 52 页会给出答案。最后还要纠一个态度：这一节不是要否定开发，也不是要把测试抬到开发之上，而是纠正一种排序。开发和测试是同一支队伍里的两种专业能力，没有谁比谁低一等，谁先谁后也不是身份问题，而是时机问题。

板书：1.5 测试与开发的关系 → 传统直觉：开发造、测试查，测试排最后 → 代价：时间被挤、缺陷晚暴露、返工大 → 本节两问：测试是下一道工序吗？测试与开发该是什么关系？→ 回顾：1.3.1—1.3.4

提问：回想一下你做过的课程项目或小组作业，测试是安排在哪一步做的？结果如何？

预设回答：①答"最后交之前随便点几下"——非常真实，也是本页要打的靶子。教师点评：不要批评，先问"那几次点出来的问题好改吗"，多数学生会说难改，因为结构已经定了。②答"边写边测"——好回答。教师点评：这正是第 52 页要讲的并行协作，可以先表扬并让他说明具体做法。③答"我们分工，有人专门测"——这是"流程上的顺序"和"人员上的分工"混在一起了。教师追问：专门测的那个人，是从需求阶段就参与，还是等着代码写完才开始？把混点拆开，学生才能分清"角色分工"和"阶段排序"是两回事。

易错点 / 考点：第一，把"并行协作"当成口号，答不出在哪些阶段交换哪些信息；第二，把"测试不是下一道工序"理解成"测试不必在开发之后执行"，把介入时机和执行时机混为一谈；第三，误以为测试左移就是要测试人员去替开发写代码。判断口诀：活动提前介入、执行按版本推进、责任团队共担。

过渡：先看最常见、也最需要纠正的那张图——把测试当成开发的下一道工序。

【1.51】测试不是开发下一道工序（3 分钟）

这一页只有一条链：需求定义、设计、编程、测试、交付，五个方框一字排开。大家注意，这条链本身没有错——从时间上看，需求确实在设计和编程之前，代码确实在测试之前。它错在哪？错在我们把它理解成"做完上一格才能进下一格，做完就没我的事了"。这条链最大的问题是可以被读成一条流水线：需求分析员交需求，设计师交设计，程序员交代码，测试员交测试报告，然后交付。每一格都只管自己那一环，前一格交出去就结束。这在软件里会带来三个后果。第一个后果，缺陷发现得晚。需求里的一处含糊、设计里的一处接口不一致，一路上没人质疑，一直走到测试阶段才被发现，这时候要改的东西已经扩散到设计、代码、数据、文档。第二个后果，返工量成倍放大。经济视角那一页我们讲过：发现越早，返工越小。第三个后果，最实际——测试时间被压到最短。因为它在链条末端，前面每一格延期都会吃掉它的时间，而它又是最后一格，没有地方可以再往后推，于是只能压缩用例、压缩回归、压缩环境准备，最后连"测完"都做不到。真实世界里的教训不少。据本课程案例素材，波音星际客机的软件问题，反映出的就是测试被拆散、端到端验证不足；

更早的火星探测器事故，则与模块接口没有做充分的集成测试有关。这些事故的共性不是"程序没写出来"，而是"没有人从整体上验证它"。所以这一页要建立的观念是：这条链描述的是**工作产品的依赖顺序**，不是**人员的篱笆**。你可以按顺序产出需求、设计、代码，但测试的思维和活动必须提前介入，从需求阶段就盯着每一样工作产品的质量。有同学会问：那测试到底放在哪？答案是：测试无处不在，下一张图会讲清楚。

板书：传统串联链：需求定义 → 设计 → 编程 → 测试 → 交付 → 误读＝"做完上一格就没我的事" → 三后果：①缺陷发现晚 ②返工成倍放大 ③测试时间被压缩（末位、无路可退）→ 案例：系统集成与端到端验证不足引发事故 → 正解：这是工作产品的依赖顺序，不是人员的篱笆

提问：为什么这条链上最容易"牺牲"的总是测试？如果不想被牺牲，测试应该在前面做哪些事？

预设回答：①答"因为它在最后，没有下游可以顺延"——准确。教师点评：这句话点出了排程上的本质，写进板书。②答"因为测试不重要"——错误但值得讨论。教师追问：如果测试不重要，为什么出了线上事故大家第一个问的是"测了没有"？引导学生发现：说它不重要，只是因为它"看起来不产出功能"。③答"前面做需求评审和设计评审，把问题提前拦下来"——很好的回答。教师点评：这正是测试左移，可以顺势让大家举一个"需求评审里发现的问题"的例子，把抽象概念落到自己的项目上。

易错点 / 考点：第一，把"测试不是下一道工序"误读成"测试不需要在开发之后进行"——测试的执行确实在产品可用之后，但测试的介入必须提前，这是两件事。第二，考试里常把"测试是开发完成后的最后一个阶段"当作正确说法，这是典型的错误选项。第三，写文档时把测试计划安排在编码结束后才启动，属于实践扣分点。判断三步法：问顺序（时间上有先后，正常）→ 问介入（活动是否提前，必须提前）→ 问人员（是不是各扫门前雪，不能）。

过渡：既然不该是一条单向流水线，那测试和开发到底该怎么排、怎么配合？最后一张图给出答案。

【1.52】测试与开发是并行、协作的关系（3 分钟）

这一页是 1.5 节的结论：测试与开发是并行、协作的关系。这张图要大家看出的，是两条轨道同时向前推进，并且在每个节点上都在交换信息。我们把"并行"和"协作"分开讲。先说并行。并行指的是测试活动和开发活动在时间上重叠，而不是一前一后排队。怎么重叠？需求阶段，测试人员就参与需求评审，专门盯那几个问题：这条需求能不能验证？验收标准写清楚了吗？有没有互相矛盾或者含糊的地方？这些正是后面写用例的依据。设计阶段，测试人员参与设计评审，关注接口约定、异常处理、数据边界，同时开始做测试需求分析和测试计划。编码阶段，开发人员自己写单元测试，测试人员准备测试数据和用例，双方对接口契约达成一致，甚至可以先把接口的测试用例冻结下来，等实现完成就能马上开跑。到了系统测试和验收阶段，测试人

员主导，开发人员待命修缺陷，缺陷修复后由测试人员回归验证，形成闭环。再说协作。协作的意思不是"客气"，而是双方对同一个目标负责。测试人员不是来挑刺的，他的产出是质量信息，帮团队在交付前发现问题；开发人员也不是把代码一丢就完事，他要提供可测的版本、清楚的变更说明、真实的环境和日志。落到具体做法上有几条特别实用：一是缺陷报告要写得能被复现——步骤、数据、环境、期望与实际，缺一样开发就要来回问；二是修复后要说明影响范围，方便测试判断回归哪些用例；三是遇到需求歧义，一起找产品确认，而不是互相赌气。这套并行协作的机制，业内常叫测试左移和持续反馈，前面 1.3.4 风险视角里那句"持续反馈质量风险"讲的就是它。给大家一句话总结：开发和测试不是接力赛的上下两棒，而是同一辆车上的两个仪表——一个负责把车开起来，一个负责告诉你车况怎么样；只有并排看着同一块路，才知道什么时候该踩刹车。这也正是我们课程设计里要求团队协作、要求测试与开发文档互相配套的原因。

板书：测试与开发＝并行 + 协作 → 并行：需求评审介入（可验证性/验收标准）→ 设计评审介入（接口、异常、边界）→ 编码期：单元测试 + 用例与数据准备 + 冻结接口契约 → 系统/验收：测试主导，开发修复，回归闭环 → 协作：缺陷可复现、变更讲影响、歧义一起找产品 → 关键词：测试左移、持续反馈

提问：测试左移最直接的好处是什么？请从"缺陷的引入阶段"和"修复成本"两个角度各说一句。

预设回答：①答"越早发现问题，改动越小、成本越低"——正确，覆盖了成本角度。教师点评：再补一句引入阶段的角度——在需求阶段拦下一个歧义，等于同时拦住了由这个歧义派生出的设计错误、代码错误和一批用例。②答"测试人员可以早点开始写用例，后面轻松点"——只说到进度，没说质量。教师追问：如果测试人员只是在旁边写用例，不提需求问题，算不算左移？引导学生区分"提前干活"和"提前发现问题"。③答"左移就是让开发多写代码少测试"——方向反了。教师点评：左移不但不减测试，反而让开发承担更多的验证责任（比如单元测试），是总量增加、位置前移。

易错点 / 考点：第一，把"并行"理解成"同时开始、各干各的"，漏掉"协作"和信息交换这一半。第二，认为测试左移就是"测试提前介入执行"，忽略了需求与设计评审阶段的介入形式。第三，判断题常见"开发和测试是两个独立的、互不干涉的阶段"——错误；"测试人员只对产品负责，不对流程负责"——也不准确，因为测试结果同时是 SQA 评价流程的依据。记忆口诀：并行不排队，协作不甩锅；左移一步，省下十倍。

【思政落点 52】讲什么故事：这一页讲协作，最能体现一个团队的质量文化。怎么联系知识点：测试与开发并行协作，意味着缺陷信息要如实、及时地在两边流动，谁的锅先放一边，先把问题解决。落到什么价值观：软件是集体作品，推责比缺陷更伤人。开发对测试隐瞒改动、测试对开发夸大问题，都会让团队失去互信；反过来，把"我们一起把这个缺陷解决掉"当成默认姿态，团队的质量水平和交付能力都会上一个台阶——这也是本课程团队协作与工程表达能力这一目标想训练的东西。

> **过渡：**到这里，第 1 章关于"测试是什么、为什么必须测、测试与质量保证是什么关系、测试与开发是什么关系"的讨论就讲完了。下面我们把镜头拉近，进入软件测试的基本概念——测试的分类与级别、静态测试与动态测试、黑盒白盒与灰盒，看看一次具体的测试工作到底是怎么分类、怎么分层展开的。

【1.53】1.6 测试驱动开发的思想（1.5 分钟）

好，同学们，我们进入本章的最后一节——**1.6 测试驱动开发的思想**。这一节不摆新概念，它承担的任务是**给前面五节收口，再把顺序翻转过来**。

大家回想一下，这一章我们一路走来，讲了测试的必要性、测试的价值、从五个视角认识测试、正向思维和逆向思维、测试与质量保证的分工、测试与开发的并行协作。你会发现，所有这些讨论都指向同一个结论：**测试的地位，必须从"事后把关"提到"事前定义"。**

那么问题来了，怎么在工程上把这个结论落地呢？靠喊口号吗？不是。靠的是一套人人都能执行的具体做法——**这就是测试驱动开发，英文缩写 TDD。**

我给大家先透一个底：这一节只有四页，但它是本章的压轴——**前面五节解决的是"测试重不重要"，这四页解决的是"测试放在哪里做"。**一个解决认识，一个解决动作。认识不对，动作就白做；动作不落地，认识就是空谈。

所以这一节回答的是本章最后那个问题：**测试这件事，到底该在什么时候开始？**

> **板书：**黑板左侧竖写"需求 → 设计 → 编码 → 测试 → 交付"这条传统顺序；右侧并排写"**1.6 TDD：测试前移、顺序反转**"；中间画一个反向箭头，从"测试"指回"需求"，箭头旁写"← 提前介入"。

> **提问：**TDD 这个名字里，"驱动"两个字最值得我们琢磨。它驱动的到底是什么——是驱动写测试的手，还是驱动写代码的手？换句话说，TDD 最有价值的产出，是那些测试用例本身，还是那条被反转过来的顺序？

> **预设回答：**第一种回答会说"最值钱的是测试用例，攒下来就是回归测试的资产"——这个回答只对了一半，要肯定它看到了**资产价值**，然后追问："如果只为了攒用例，那为什么不先写完代码再补用例呢？反正用例最后还是那些用例。"这一问就能把学生推到关键点上。第二种回答会说"最值钱的是那个先后顺序，顺序一变，人做事的思路就变了"——这是我们要的答案，要直接表扬："对，先写测试，你在心理上就被迫先回答'它应该做什么'，而不是'我做了什么'，这个心理转向才是 TDD 的命门。"第三种，也会有同学说"不就是把测试提前吗，我提前写就行了"——这是**典型的误读**，要点破：TDD 不只是时间上提前，而是要**先写一个会失败的测试**，看着它红、再让它绿，这个"红—绿"的节奏是有硬性要求的，下一两页我们就讲这个节奏。

易错点 / 考点：第一个混淆点，是把 TDD 当成"一种测试技术"——错，**TDD 是一种开发思想、一种设计方法**，它产出的东西是代码，不是测试报告。第二个混淆点，是把 TDD 和"先写测试再写代码"划等号——不够，还要求**测试先行、失败为先、小步快跑、随时可重构**。判断口诀记四个字：**"先、败、小、重"**——先写测试、允许失败、小步进行、随时重构。

过渡：名字听过了，位置也说清了，那 TDD 到底怎么个"测试在前、开发在后"？我们翻到下一页，看它那张最经典的循环图。

【1.54】测试驱动开发的思想（4.5 分钟）

这一页的标题还是"测试驱动开发的思想"，正文只有一行字，但这一行字是整节的定盘星——**TDD，test driven development，测试在前，开发在后。**请注意这八个字的分量：它不是"测试和开发同时进行"，更不是"开发完再补测试"，而是**测试必须排在整个动作序列的最前面。**

屏幕下半部分画的是经典的 TDD 循环图，我用大白话带大家走一圈，一共八步，一圈一圈地转。

起点叫"开始"，旁边写"**为新性能写一个测试**"。注意，还没有任何实现代码，先把"我期望它做到什么"写成一条测试。

第二步是"**编译**"，第三步是"**修订编译错误**"——这一步特别有意思：因为实现还不存在，测试根本编译不过。但大家注意，**这里的报错不是事故，是路标**，它明确告诉你"还差什么"。

第四步"**运行测试并发现错误**"，第五步"**编写代码**"——只写让这条测试通过所需要的最少代码。第六步"**运行测试并通过**"。第七步"**如果需要就进行重构**"。第八步，箭头回到开头，为下一个新功能再写一个测试，循环往复。

我把这个循环概括成四个字，大家记下来：**红、绿、重构。**测试失败是红，测试通过是绿，在绿灯的保护下整理内部结构，就是重构。

这里我必须讲透一个最容易讲错的地方：**为什么一定要先看到"红"？**因为如果一条测试你从没见过它失败，你就根本不知道它有没有在验证东西——它可能永远都是绿的，那你写了等于没写。这是 TDD 里最容易被跳过、也最不能跳过的一步。

再打个生活化的比方。你去体检，是先定好"血压低于 140 算正常"这个标准，还是等检查完了再回头说"哦，我这份报告看着还行"？当然是先定标准。**TDD 里那条测试，就是你先写下来的体检合格线。**写代码的过程，

就是照着这条合格线去做。

还有一个词要解释：**重构**。重构不是"重写"，它的定义是**在不改变外部行为的前提下，改善代码的内部结构**。那问题来了，谁敢在项目里随手改结构？答案是：**手里有测试网的人才敢。**测试全绿，你改完再跑一遍，绿的就还敢继续走；绿变红了，说明你改坏了，立刻退回。**测试网是重构的安全绳**，这就是"测试驱动"这四个字最实在的收益。

最后提醒大家看图上的循环形状——它是一个**闭合的圆**，不是一条直线。直线意味着"做完就完了"，圆意味着**每一小圈都留下了一层测试**。转一圈加一个功能，也加一层回归保护。这就是为什么 TDD 出来的项目，越往后越不怕改。

板书：画一个顺时针圆圈，标八步：**为新功能写测试 → 编译 → 修编译错误 → 运行测试、发现失败（红）→ 编写代码 → 运行测试、通过（绿）→ 需要则重构 → 回到起点**。圆圈中央写三个大字：**红 · 绿 · 重构**。右下角补一句：**测试 = 先定好的合格线**。

提问：TDD 循环的第四步是"运行测试并发现错误"。假如我图省事，跳过这一步，直接写代码、写完了再一次性运行测试，会丢掉什么？请大家具体说出丢掉的到底是什么。

预设回答：第一种回答会说"丢掉了一次运行而已，问题不大"。这个回答要当场纠正，因为它把"运行"当成了成本：丢掉的是**对测试本身有效性的验证**——一条从没红过的测试，你没有证据证明它在真的检查东西，它可能只是恒为真的空断言。这时可以追问："怎么证明一条测试不是摆样子？"答案是："看它红过。"第二种回答会说"丢掉的是对需求的确认，就是那个编译不过、测试失败的阶段，本来是在逼我想清楚接口长什么样"。这个回答层次很高，要表扬并放大：**TDD 的第一个产物其实不是测试，而是接口设计**，是先想清楚"外面怎么用我"，再想"我里面怎么写"。第三种回答会说"省了时间，效率更高"。这个要借势讲代价：省下的是几十秒的运行，赔上的是**没有回归网**——后面每次改代码你都不知道有没有改坏别处，最终会付出更大的返工代价，这跟我们前面讲过的"缺陷发现越迟、成本非线性增长"是同一条规律，只是缩短到了分钟级。

易错点 / 考点：三个高频错误。第一，**把"测试在前"理解成"测试文档在前"**——不是，是**可运行的测试代码在前**，必须是能编译、能执行的。第二，**忽略"红的"那一步**，直接写代码再补测试，这不是 TDD，这叫"测试稍后（test after）"，考试里经常拿它做干扰项。第三，**把重构和重写混为一谈**——重构**不改变外部行为**，重写会改变；多选题常把"重构=重写"设成错误选项。判断三步法：一看**有没有先写测试**，二看**有没有先看到失败**，三看**有没有在小步循环里积累回归测试**。

过渡：TDD 这条循环看着挺简单，但它不是哪个人凭空想出来的。它有明确的出身和出处——下一页我们

> 就去看看，它最早来自哪个开发方法。

【1.55】TDD 的实践最早来自极限编程（4.5 分钟）

这一页的标题是一句结论：**TDD 的实践最早来自极限编程。**极限编程的英文是 Extreme Programming，缩写 XP，它是 1990 年代末敏捷运动中非常有代表性的一种开发方法。TDD 并不是哪本教材坐在办公室里设计出来的，它是**在 XP 的工程实践中先做出来、后被总结成思想的**。这一点很值得让学软件的同学记住：**工程方法往往先有实践，后有理论。**

屏幕上这张图是课件里给出的 XP 实践流转图，信息量很大，我带着大家一步一步读。

我们从最左边开始。起点是"**下一个任务或失败的验收测试**"——注意，一上来就有"失败"两个字，这说明**这个流程是被"还没满足的东西"推着往前走的**，不是被"已经做完的东西"推着走，这个出发点就决定了它的性格。

接着箭头分成两层：**简单设计**指向 **CRC 卡**，**复杂问题**也指向 CRC 卡。CRC 就是 Class-Responsibility-Collaborator，类－职责－协作者卡片，是一种很轻量的设计工具。它要说明的是：**不管问题简单还是复杂，设计这一步都不能省，只不过用的工具可以很轻。**

再往右，从"下一个任务或失败的验收测试"出发，**结对**之后"**创建一个单元测试**"。这里"结对"两个字很关键——**两个人一起干，一个人写、一个人看**，这是 XP 的标志性做法，也正好回应了我们马上要讲的那个问题：**一个人测自己写的东西会有什么毛病。**

接下来进入中间那个核心循环，我们看它的每一步：**创建单元测试 → 修订代码 → 失败或通过的单元测试 → 不断的集成 → 运行所有单元测试 → 100% 单元测试通过。**

大家仔细看"修订代码"这个节点，它下面挂着一个词叫"**代码重构**"，而且重构旁边标了两条："**简单代码**"和"**复杂代码**"。这个细节很值得讲：**简单代码就直接改，复杂代码就要靠重构来梳理**——但两条路都通向同一个目标，就是让代码保持可维护。

图上还有几条反馈回路，我也点一下：一条向上写"**改变结构、对人员**"，一条写"**人员调整**"，再往上一条写"**我们需要帮助**"。这几条线说明 XP 不只是技术实践，它同时管人：**结构不合适就调结构，人干不动就调人，搞不定就喊帮助。**这是一种非常坦诚的团队文化。

再看最右边：**新的单元测试 → 新功能** 这条线，是"不断集成"时新功能带来的新测试；而最上方一条弧线写着"**运行失败的验收测试**"，从"100% 单元测试通过"绕过去，最后落到"**通过验收测试**"。这一段是整个流程的

出口——**单元测试全绿只是内部过关，验收测试通过才叫对外交付。**

> **板书：**左侧竖排写三条主线——**① 任务/失败的验收测试 ② 简单设计与复杂问题 → CRC 卡 ③ 结对 → 创建单元测试**；中间画核心循环：**创建单元测试 ⇄ 修订代码（重构：简单代码/复杂代码）→ 不断集成 → 运行所有单元测试 → 100% 通过**；右侧写出口：**100% 单元测试通过 → 运行失败的验收测试 → 通过验收测试**；最上方批注三条旁线：**人员调整、改变结构、我们需要帮助**。

> **提问：**这张流程图的入口写的是"下一个任务**或失败的验收测试**"。为什么一个开发流程，要用"失败"来当起点？换成"下一个要开发的功能"，不是更顺耳吗？

> **预设回答：**第一种回答会说"因为还没有实现，所以测试当然是失败的，这是正常状态"。这个回答已经摸到门了，要顺着往下压一句："对，但请你再说深一层——**把'失败'摆在起点，等于承认了'不满足'才是工作的常态**，这和你习惯的'我做完一个就勾一个勾'是完全不同的心态。"第二种回答会说"用失败来驱动，能保证每一步都是有目标、可验证的"。这个回答很好，把它升级成一句板书：“**从「还没满足」出发，每一步都有一个可以说清楚的判据。**”第三种是典型的抵触回答："这不就是先写个必然失败的测试，多此一举吗？"这时候要正面接住：这不是多此一举，恰恰是因为**人天生倾向于保护自己的成果**，先写失败测试，就能把"我写得对不对"这个主观判断，换成一个客观的、机器来判的绿灯/红灯。

> **易错点 / 考点：**第一，**极限编程（XP）** 这个名字的写法与归属要记准——考题常问"TDD 最早来自哪种开发方法"，答案就是**极限编程（Extreme Programming）**。第二，**CRC 卡（类－职责－协作者）** 是设计工具，容易被误当成编码工具或测试工具。第三，**"100% 单元测试通过"与"通过验收测试"是两个关卡**，不能合并——单元测试是内部的、盯代码的，验收测试是对外的、盯需求的，这个区别在第 56 页还要再往前推一步。

> **过渡：**讲到这儿大家应该发现了，TDD 在 XP 里是"一个动作"。后来它长大成了一种"思想"，还分出了两个不同的实践层次。下一页，我们就把这三个名字摆在一起说清楚：TDD、UTDD、ATDD。

【1.56】TDD 与 UTDD、ATDD（4.5 分钟）

> 这一页的标题是"**TDD 与 UTDD、ATDD**"，页面上还写着一行总结性的话，我先把这行话念清楚：**TDD 成为思想，UTDD 单元测试驱动开发、ATDD 验收测试驱动开发则成为实践。**请大家把这句话记在笔记最上面，因为考试里最爱考的就是这三个词的分工。
>
> 我们先看屏幕这张图。它是两个同心圆：**里面那个小圆标着 UTDD**，圆里画的是"**单元测试**"和"**代码**"两个方

块，两个方块之间是**双向箭头**；**外面那个大圆标着 ATDD**，大圆里面包含了整个小圆。

大圆左上方还有一个"**验收指标**"的椭圆，它用箭头指向"**验收测试**"；同时它还有一条虚线连到左下角那个**笑脸**图标上。另外还有一条虚线，从笑脸连向"验收测试"的方向。

这张图的读法，我教大家一句话：**思想在圆心，实践分两层；内圈管代码，外圈管指标。**

先说**内圈 UTDD**，Unit Test Driven Development，**单元测试驱动开发**。它的对象是"代码"这个最小的可测单元，做法是先用单元测试把"这个函数应该输入什么、输出什么"钉住，再去把代码写出来，然后让测试变绿。**注意那两个方块之间是双向箭头**——这说明单元测试和代码之间是**相互塑造**的关系：测试约束了代码的行为边界，代码的实现细节又会反过来催生新的测试。

再说**外圈 ATDD**，Acceptance Test Driven Development，**验收测试驱动开发**。它站在比单元更高的位置：从"**验收指标**"出发，先把"这个东西要满足什么条件才算交付合格"讲清楚，再把这些指标变成**验收测试**。大家看图上的位置关系——**ATDD 这个大圆把 UTDD 整个装在里面**，这个包含关系非常重要：它说明**验收测试驱动开发并不排斥单元测试，它是把单元测试包在自己的循环里**。业务上的验收条件往下分解，落到代码层面就变成一条条单元测试。

最后说**圆心上的 TDD**。为什么说 TDD 是"思想"而不是"实践"？因为思想只有一个内核——**测试先行、以测试来驱动设计与实现**；至于在哪个层次上先写测试，那是实践层面的选择：**在最细的代码层次上先写测试，就是 UTDD；在验收指标的层次上先写测试，就是 ATDD。**两个实践的出发点不同、颗粒度不同、面向的人也不同，但它们共享同一个思想内核。

图上那个笑脸，我理解课件想表达的是"**最终结果让使用者满意**"——验收指标是要对着"用户满意"这条线来定的，而不是对着开发自己方便来定的。这个视角一摆正，ATDD 和 UTDD 的分工就清楚了：**UTDD 保证里面是对的，ATDD 保证外面是用户要的。**

> **板书：**先画一个大圆标 **ATDD（验收测试驱动开发）**，圆内再画一个小圆标 **UTDD（单元测试驱动开发）**；小圆内并排写"**单元测试 ⇄ 代码**"；大圆内小圆外写"**验收指标 → 验收测试**"，并引一条线连到"**用户满意**"；大圆圆心处写一个大字："**TDD＝思想（测试先行）**"。旁边竖排口诀：**思想在圆心，实践分两层；内圈管代码，外圈管指标。**

> **提问：**假设一个团队说"我们做的是 TDD，但我们只写验收测试、不写单元测试"。按这一页的图来判断，他们的话站得住吗？为什么？

> **预设回答：**第一种回答会说"站得住，因为 ATDD 把 UTDD 包在里面了，做外圈自然就包含内圈"。这个回答

混淆了"包含"和"替代"，必须点破：包含关系说的是**范围**，不是说**自动等价**——外圈大并不等于内圈可以被跳过，图里小圆是**实打实画在里面的**，那意味着单元测试这一层仍要存在。第二种回答会说"站不住，ATDD 和 UTDD 是两个不同颗粒度的实践，一个管验收指标、一个管代码细节，验收测试过了不能说明每个单元的逻辑分支都对"。这是标准答案，要追一句把考点夯实："对，所以这两层是**互补**，不是**互替**。"第三种回答会说"只要产品能用，写不写单元测试无所谓"。这个回答在工程上很危险，正好用它讲代价：没有单元测试，缺陷一旦在集成阶段暴露，**定位成本会成倍上升**，因为你不知道是哪一块坏了——这又是我们第一章反复出现的那条规律：**越晚发现，代价越高。**

易错点 / 考点：第一，**三个缩写的全称必须能默写**：TDD＝Test Driven Development，UTDD＝Unit Test Driven Development（单元测试驱动开发），ATDD＝Acceptance Test Driven Development（验收测试驱动开发）。第二，**"TDD 是思想，UTDD 和 ATDD 是实践"**这句话是原句，判断题常把"TDD 是一种实践"设成错误选项。第三，**内圆外圆的关系是包含，不是并列，也不是替代**——画图题常考：请把 TDD、UTDD、ATDD 用一个图表示出来，标准答案就是**同心圆**。第四，别把 ATDD 和"验收测试"混为一谈：ATDD 强调**驱动**，也就是**在开发之前就用验收标准来牵引**，它是动作；验收测试是**最终执行的检验**，它是结果。

过渡：这一章讲了这么多，从测试的价值讲到 TDD 的思想，最后我要把一个最尖锐的问题摆在大家面前——**让开发人员测自己的产品，到底行不行？TDD 凭什么能解决它？**下一页就是这个问题。

【1.57】问题（5.5 分钟）

这一页没有别的，页面上就是两个字——"**问题**"。下面两行小字是两个问句：第一，**如果开发人员测试自己的产品，会有哪些障碍？**第二，**TDD 又是如何克服这些障碍的？**这一页是我们第 1 章的思考高点，也是考试、答辩都爱问的地方，我按参考答案的口径，把它一层一层讲透。

先说第一个问题，障碍分三层，我们逐层看。

第一层，是思维盲区。人写代码的时候，脑子里已经装着自己那套实现思路。等到回头测试，他会不自觉地**顺着自己的实现思路去测**。这一顺，问题就来了——**他只会去走自己"想到过"的路径，而缺陷恰恰最爱藏在"他没想到"的路径和输入组合里。**说白了，**用你自己的思路去找自己的错，就像一个从来没迷过路的人去画地图，他会把走过的那条路画得很细，把没走过的岔路口漏掉。**

第二层，是心态偏差。这一点更隐蔽、也更普遍。人在面对自己亲手做出来的东西时，**天然倾向于证明"我写的是对的"**。这不是道德问题，这是心理机制。所以同样一条失败的测试，别人跑出来会追问"代码哪里错了"，

自己跑出来第一反应往往是"是不是测试写错了"。**这个倾向会让缺陷被解释掉、被绕过、被改成"设计如此"。**这里我要提醒一句：这才是"开发人员不适合测自己产品"这句话的真正含义——不是能力不够，而是**立场天然不中立**。

第三层，是独立性与时间不足。独立性说的是：**开发和测试最好不是同一个人**，这样评审视角才客观，这在测试组织里叫"独立性原则"。时间不足说的是现实里的排期——开发任务压得最紧的时候，最先被砍掉的往往就是自测和用例设计，因为**它们不直接产出新功能，交付压力一来，第一个被牺牲的就是它们。**

现在讲第二个问题，**TDD 是怎么把这三层障碍逐一化解的**。它一共给了我们三件武器，正好对着三个障碍。

第一件武器，对应思维盲区：先写测试、再写实现。这个顺序的意义在于，它**逼着你从需求与接口出发**——因为写测试的时候，实现代码一行都还没有，你手里只有"这个功能应该表现成什么样"，你只能从外部往里看。**这就把"顺着实现思路测"的路给堵死了。**

第二件武器，对应独立性与时间不足：测试先行形成自动回归网。测试是在写功能之前就落下来的，它不会等到交付前才被临时砍掉——**它已经变成了代码库的一部分。**而且这个回归网是自动的，随时能跑，这就把"没时间测"从一个主观借口，变成了一个客观事实：**你的测试覆盖了多少，跑一遍就知道。**

第三件武器，对应心态偏差：用例就是"可执行规格"，减少主观判断。这是三件武器里最漂亮的一件。原来"这算不算缺陷"是个**主观问题**，靠人吵；现在它变成了**客观问题**——测试通过还是不通过，机器说了算。**规格是可执行的，判断就不再有解释空间。**所以 TDD 不只是让测试更早，它是**把"质量由谁说了算"这个问题，从人手里交给了测试。**

最后我给大家一句总纲，请记牢：**障碍的根子是"自己人评自己"，TDD 的解法是"用先写下来的客观标准去评"。**先把期望写成测试，再把实现写出来——立场问题被顺序解决了，盲区问题被外部视角解决了，时间问题被自动化解决了。

板书：左侧写"**障碍三层**"：① **思维盲区**——顺着自己的实现思路测，漏掉没想到的路径与组合；② **心态偏差**——倾向证明"我写的是对的"；③ **独立性与时间不足**。右侧写"**TDD 三件武器**"，用箭头一一对应：① 先写测试再写实现 → **迫使从需求与接口出发**；② 测试先行 → **自动回归网，用例就是可执行规格**；③ 用例即规格 → **减少主观判断**。底部写总纲：**用先写下来的客观标准，去评自己写的东西。**

提问：三个障碍里，"心态偏差"是最难承认的一个。请你举一个你身上真实发生过的场景——你是怎么在"发现自己的东西不对"的那一瞬间，说服自己"其实这样也行"的？

预设回答：第一种回答是典型的："我写作业的时候跑出来结果不对，第一反应是检查输入数据，改了几次

数据凑出正确答案，就交了。"这个回答太真实了，要立刻接住并顺势把它变成教学素材："**注意他刚才做了什么——他改了输入去迁就输出。**这就是心态偏差最标准的表现形式：不是改代码，是改标准。而 TDD 的做法刚好反过来，**标准是事先锁死的，你只能改代码。**"第二种回答会说"我会跟自己说这个边界情况不常见，用户不会这么用"。这也是极好的素材，要追问："那你凭什么判断用户不会这么用？"答案往往就是"凭感觉"——那就点破：**"凭感觉"正是主观判断，而 TDD 就是把这条"凭感觉"的路用测试堵住。**第三种，也会有同学很硬气地说"我从来都要求自己严格自测，不存在这个偏差"。这时候不要否定他，要用提问引导："好，那我问你——**你怎么证明你测到了你没想到的地方？**"这个问题他答不上来，因为**你无法用"想到"去覆盖"没想到"**，这恰恰是外部视角和自动化回归网不可替代的原因。

易错点 / 考点：第一，**"开发人员不能测自己的产品"这句话的标准理由，是"独立性"与"客观性"，不是"水平不行"**——选择题常把"开发人员技术能力不足"设成错误选项。第二，**TDD 克服障碍的机制要能对上号**：先写测试对应"思维盲区"，自动回归网对应"时间与独立性"，可执行规格对应"心态偏差"，这三条是**一一对应**关系，不要交叉。第三，**别忘了"用例即规格"这个提法**——它把主客观判断的边界讲清楚了，简答题里是加分点。判断三步法：**谁在评？评什么？用什么标准评？**——TDD 的答案是"用事先锁死的外部标准评实现"。

过渡：这一章从"为什么要测试"一路讲到"TDD 怎么解决自测的障碍"，到这里，1.1 到 1.6 的六节内容就全部讲完了。下面我们用一页时间，把这一章的五个理解点收拢起来。

【1.58】本章小结（4 分钟）

好，这一页是**本章小结**。屏幕上列了五条，我就按这五条，一条一条把第 1 章的要点钉一遍。大家注意，**这五条里的每一条，都对应着我们这一章前面讲过的一整节**，所以我讲的时候，你们可以顺手在心里回放一下那一节的关键句。

第一条，理解测试的定义与价值。测试是什么？按国际标准的说法，是在特定条件下运行系统或组件，观察或记录结果，对它的某个方面做出评价。而测试的价值有四个层面：**全面评估产品质量，拿到客观的质量信息；发现问题、督促问题解决，从而提高质量；持续提供质量反馈、及时揭示质量风险；通过缺陷分析得到缺陷模式，进而预防缺陷。**大家看，从"评估"到"发现"到"反馈"到"预防"，这四个词其实是一条**由被动到主动的上升线**——测试走到最高层，不是抓错，而是**让错误不再发生**。

第二条，从不同视角认识软件测试。我们这一章一口气给了五个视角：**质量视角**，测试是对软件质量的全面评估，给出质量信息；**风险视角**，测试是对系统潜在质量风险的评估，而且因为测试是抽样实验、不可能穷

尽，风险总是存在的；**经济视角**，测试追求以最小代价获得最高质量，所以**测试成本必须小于缺陷造成的损失，测试才有意义**；还有 **Test Oracle 视角**，输出是否正确需要有判断准则；以及**批判性思维视角**，测试是一个借助观察、经验、反思、推理去收集信息、不断探索的过程。五个视角，其实是五把尺子——**同一件事，换个视角看，你关注的东西完全不同。**

第三条，正向思维、逆向思维，也会决定测试的行为。这一条是本章的思想核心。正向思维的代表是 Bill Hetzel 博士，认为测试是**为程序能按预期运行而建立信心的过程**；逆向思维的代表是 Glenford J. Myers，讲得更锋利——**测试是为了证明程序有错，而不是证明程序无错；一个好的测试用例在于它能发现至今未发现的错误；一个成功的测试是发现了至今未发现的错误的测试。**落到行为上就是一个对比：正向思维是"**在设计规定的环境下运行软件的所有功能，直至全部通过**"；逆向思维是"**寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统，直至找不出问题**"。**认知决定行为**——你心里假设它是对的，你就在走过场；你心里假设它有错，你才会去挑刺。

第四条，理解测试、SQA、质量、开发之间的关系。SQA 是一项**管理工作**，侧重于对流程的评审和监控，它指导、监督测试的计划与执行；测试是一项**技术性工作**，侧重对产品进行评估和验证，**测试是 SQA 的重要手段之一**，为 SQA 提供质量数据。至于开发，**测试不是开发的下一道工序，两者是并行、协作的关系**——需求、设计、编码的每一步，测试都在场。

第五条，理解先进的 TDD 思想。一句话：**测试在前，开发在后**；它最早来自极限编程的实践，TDD 是思想，UTDD 和 ATDD 是它的两种实践；它靠"先写测试、看到失败、再写实现、然后重构"的循环，把"自己人评自己"这个根子上的障碍给拆掉。

最后我给大家留一句本章的总结：**这一章我们学到的不是"怎么测"，而是"该在什么时候、以什么心态去测"。**答案是：**越早越好，并且先假设它有错。**

> **板书：**竖排五条：① **定义与价值**（评估 → 发现 → 反馈 → 预防）；② **五个视角**（质量 / 风险 / 经济 / Test Oracle / 批判性思维）；③ **正反思维决定行为**（Hetzel 正向 vs. Myers 逆向）；④ **测试—SQA—开发**（管理 vs. 技术；并行而非下道工序）；⑤ **TDD**（思想在内，UTDD/ATDD 在外）。黑板最下方写本章一句话：**越早越好，并先假设它有错。**

提问：这五条里如果要你只留一条带走，你留哪一条？请用一句话说出来，并说明你为什么选它。

> **预设回答：**第一种回答会选第①条"定义与价值"，理由是"这是基础，后面全都要用它"。这个回答稳妥，要肯定，但可以把它的层次往上推一层："基础确实重要，不过你再想想——**定义讲的是'测试是什么'，心态讲**

的是'测试的人是谁'，哪一个更决定成败？"第二种回答选第③条"正向逆向思维"，理由是"这一条最反直觉，也最能改变我的做事方式"。这就是我们要的答案，要直接落到板书上："对，**正向思维是「验证它没问题」，逆向思维是「证明它有问题」，这两种心态下写出来的用例，质量完全不是一个量级。**"第三种回答也会有人说"我选第⑤条 TDD，因为它最具体，能马上用"。这也很好，要顺势补一句：TDD 之所以能落地，正是因为它是把第③条那种逆向心态，**固化成了流程动作**——思想一旦变成动作，就不会因为人的惰性而丢掉。

易错点 / 考点：第一，**"测试是 SQA 的重要手段之一"与"测试就是 SQA"要分清**——这是判断题的高频陷阱，两者是**手段与体系**的关系，不是等同关系。第二，**正反思维的代表人物别归错**：正向是 **Bill Hetzel**，逆向是 **Glenford J. Myers**。第三，**经济视角的那句不等式要能默写**："**测试的成本 < 缺陷造成的损失，测试才有意义**"，考试里常把不等号方向设成错误项。第四，**"测试不是开发的下一道工序"**这个判断，是选择题的常客。记忆口诀对第③条尤其有用，四个字：**"假设有错"**——答题时先写这四个字，再展开正反两面的行为差异。

过渡：课本上的内容到这儿就总结完了，但学习不能停在"听懂了"。下一页我们留几道题，检验一下这一章是不是真的学进去了。

【1.59】提问、思考与练习（5.5 分钟）

好，最后一页练习——**提问、思考与练习**。屏幕上有四道题，我要提醒大家一句：**这四道题不是随便凑数的问题，它们的答案基本就是我们这一章所有考点的浓缩，期末和课程设计答辩里都可能以不同形式出现。**所以我不念题、等你们答，我直接把讲法和答案都讲透，你们对照着检查自己的思路。

第一题：在日常使用软件的过程中，遇到哪些软件质量问题？这道题看起来随便，其实考的是"能不能把生活现象对应到缺陷类型"。我按参考答案的口径归类：**闪退、卡顿这类，属于性能与可靠性问题**；**数据丢失或者计算结果错误，属于功能与数据缺陷**；**界面错位、乱码，属于兼容性与易用性问题**；还有一类特别常见——**更新之后老功能失效了，这叫回归缺陷**。请大家留意这四类的名字，**它们就是缺陷分类的入门口径**，也是你们第一次作业要用的标签。我建议你们回去建一个"质量问题随手记"，连续记一周，把遇到的每个现象都往这四类里归一次——**归不上类的，那本身就是个值得讨论的例子。**

第二题：软件测试的正反两方面观点，会如何影响测试工作？这道题考的是第 1.3.2 节。**正向思维是按规格去验证，"该做的都做到了"；反向思维是先假设程序有错，专门挑边界、异常输入和组合去证伪。**这两个思维落到具体工作上，差别非常具体：同样一个"登录"功能，正向思维会写"输入正确账号密码，登录成功"；反向思维会写"空密码怎么办、超长输入怎么办、特殊字符怎么办、连续错十次怎么办"。**一个是照着需求清单**

打勾，一个是照着'哪里可能出错'去挖。你们做实验的时候，我会明确要求**用例里必须有一半以上是反向用例**，就是这个道理。

第三题：软件测试和软件开发的关系是怎样的？如何更好地利用这种关系？这道题背后是我们讲的"**测试不是开发的下一道工序**"。**它们是并行、协作的关系**：需求的定义、设计、编程、交付，每一段测试都在场。那怎么"更好地利用"呢？答案就是**测试左移**——**在需求和设计阶段，测试就介入进来。**我举两个最实在的例子：需求评审的时候，测试人员专门负责挑**二义性**，一句话如果有两种读法，就当场问清楚；接口还没实现的时候，先把**接口契约和用例冻结下来**，让开发和测试照着同一份规格各干各的。**你越往前介入，你能拦住的问题就越多，返工就越少**——这正是我们第 1.2 节讲过的经济视角。

第四题：软件测试和质量保证之间的联系和区别？这一题最容易答糊，我用一句话把它钉死：**SQA 面向过程，测试面向产品。**SQA 关心的是"**做事的过程合不合规矩**"——规范有没有执行、评审有没有开、度量有没有做、过程有没有在改进；测试关心的是"**做出来的东西对不对**"——验证和确认，把质量信息交出来。**联系在于：测试是 SQA 的重要手段之一，测试跑出来的数据，是 SQA 判断质量、评价过程的客观依据；反过来，SQA 会指导、监督测试的计划和执行，督促测试的结果客观、准确、有效。**大家看，一边是管理体系，一边是技术手段，它们是**相互支撑**的。答题三步法也给大家：**先判属性（管理还是技术）→ 再比侧重（流程还是产品）→ 最后说产出（过程改进的建议，还是质量信息与缺陷数据）。**

四道题讲完了。屏幕备注里写着"重点学习内容"，确实如此——**这四道题，就是这一章的四根支柱。**

板书：左侧写四个题号；右侧写答题关键词：**① 质量问题四归**（性能与可靠性 / 功能与数据 / 兼容与易用性 / 回归）；**② 正反思维**（按规格验证 vs. 专挑边界与异常证伪）；**③ 并行协作 + 测试左移**（需求评审挑二义性、接口契约与用例先行）；**④ SQA 面向过程 vs. 测试面向产品**（测试是 SQA 的重要手段）。底部写答题三步法：**判属性 → 比侧重 → 说产出**。

提问：用第④题的框架考你们一下——如果我发现"这次迭代的缺陷特别多"，这个信息应该反馈给谁？是让测试把用例写得更狠一点，还是该去查一查过程和规范哪里出了漏洞？

预设回答：第一种回答会说"当然先让测试补用例，先把 bug 都抓出来"。这个回答只做了一半，要肯定它的紧迫性，同时提醒：**缺陷多是一个结果，不一定是一个原因**。追问一句："如果同一类缺陷反复出现，补用例能解决它第二次出现吗？"——这就是把话题引到 SQA 上。第二种回答会说"两边都要做，短期靠测试补齐覆盖，长期要靠 SQA 去查根因、改过程"。这是最完整的答案，要表扬并帮他说完："对，**测试解决这一次，SQA 解决下一次**，这正好就是 SQA 要做度量、要做过程改进的原因。"第三种回答会说"缺陷多是开发的问题，跟测试无关"。这个回答要温和地纠偏：**缺陷多也反映测试自己的覆盖与准入把关**——测试有没有

在需求阶段就介入、有没有把准入标准卡住。这也正好回应第③题"测试左移"的价值。

易错点 / 考点：第一，**第①题的分类口径要能对上号**——"闪退卡顿"对应**性能与可靠性**，不要答成功能缺陷；"更新后旧功能失效"对应**回归缺陷**，不要答成兼容性。第二，**第②题千万别把正反思维答成"好与坏"**——它们不是对错关系，是**两种互补的思维取向**。第三，**第③题的关键词是"并行""协作""测试左移"**，如果只答"测试在开发之后进行"，整题零分。第四，**第④题最忌把 SQA 和测试说成"一个东西的两种叫法"**，一定要落到"**过程 vs. 产品**"和"**手段 vs. 体系**"这两组对比上。

过渡：四道题答完，这一章的知识就闭环了。学完一门课，光靠课件不够，最后我给大家推荐三本可以继续读下去的书。

【1.60】学习资源推荐（3 分钟）

这一页是**学习资源推荐**，课件上列了三本。我一本一本说，告诉大家**什么时候该翻哪一本**，这样你们回去才用得上。

第一本，《软件测试方法和技术（第 4 版）》，清华大学出版社，2022 年。这就是我们这门课的教材，也是我们所有术语口径的来源。我提醒一句：**我们课上讲的每一个概念，最后都要回到这本教材的说法上**，尤其是定义、分类、流程这些容易有不同版本表述的地方。所以大家备考的时候，**以这本第 4 版为准**，不要去网上随便找一份说法就背下来。这本书的结构我们也已经在讲第一章的时候带大家看过一遍了——**原理与方法、技术、项目实践这三篇**，正好对应我们这门课"理论加实践"的安排。

第二本，《全程软件测试（第 3 版）》，人民邮电出版社，2019 年。这本书的名字里有两个字最关键——"**全程**"。它讲的不是"测试阶段要做什么"，而是**测试怎么贯穿软件开发的整个过程**。你们现在听完第 1 章应该已经有感觉了：我们这一章反复强调的就是**测试左移、测试与开发并行**。等你做课程设计的时候，要写测试计划、要安排测试介入的时机，这本书会给你很完整的参照。我把它定位成**"工程实践视角的补充读物"**。

第三本，《致命 Bug 软件缺陷的灾难与启示》，人民邮电出版社，2016 年。这本书是本**案例集**。前面两本讲"应该怎么做"，这一本讲"**没做好会怎样**"。我们课上提到过的那些真实事故——放疗仪致人死亡、处理器浮点除法错误导致大规模召回、航天探测器因为接口没有做集成测试而坠毁，这一类案例在这本书里有更完整的来龙去脉。我要专门说一句：**这一本对你们最有用的场景，是写课程设计报告里的"缺陷影响分析"，以及答辩时回答"这个缺陷如果不修会有什么后果"。**

三本书的分工，我给大家总结成一句：**教材用来定口径，全程测试用来学工程做法，致命 Bug 用来长教训、找案例。**

最后补一句自己的建议：书不用一次看完。**未来两周，请先把教材第 1 章重读一遍，边读边把你在生活里遇到的软件质量问题记进那个"随手记"里。**这一步做到了，这一章就算真的落地了。

板书：竖排三本，每本后面批注用途：①**《软件测试方法和技术（第 4 版）》（2022，清华）→ 定口径 / 备考依据**；②**《全程软件测试（第 3 版）》（2019，人邮）→ 全程视角 / 写测试计划的参照**；③**《致命 Bug：软件缺陷的灾难与启示》（2016，人邮）→ 案例库 / 缺陷影响分析与答辩素材**。旁边写一句：**教材定口径，全程学做法，Bug 书长教训。**

提问：这三本书里，如果你现在只能选一本、并且只有两周时间，你会选哪一本来支撑你的课程设计？请连理由一起说。

预设回答：第一种回答选第①本教材，理由是"考试以它为准，先把口径对齐"。这个选择很务实，要肯定，同时补一句："口径对齐之后，别忘了**测试计划是要写给别人看的**，所以第②本迟早要翻。"第二种回答选第②本《全程软件测试》，理由是"课程设计要写测试计划、要安排测试介入时机，这本最直接"。这个判断很到位，要顺势提醒一个风险："工程做法好学，但**术语口径如果跟教材不一致，答辩时容易被追问**，所以两本要配着看。"第三种回答选第③本《致命 Bug》，理由是"案例最好看、也最好用"。这个回答要鼓励但要纠偏：**案例的价值在于你说得出"启示"**，如果只会复述事故经过，报告里那一段是撑不起来的——**要把事故对应回本章的知识点**，比如"这正是缺少集成测试的代价""这正是经济视角下测试成本小于损失的反面案例"，这才叫把案例用活了。

易错点 / 考点：这里不是背书的出版社和年份，而是要建立"**什么场景查什么资料**"的判断力。唯一需要精确记忆的是**教材的版本信息**：《软件测试方法和技术（第 4 版）》，清华大学出版社，2022 年——**课程所有概念的最终口径以它为准**。另外提醒两点：一是**别把"全程软件测试"和"软件测试方法和技术"混为一谈**，前者强调测试贯穿全程的工程视角，后者是本课程的系统教材；二是**引用案例时一定要写出启示**，只写事故描述在报告里是不合格的。

过渡：这一章我们从"为什么要测试"一直讲到"测试该在什么时候做"，中间经过了测试的价值、五个视角、正反思维、SQA 的分工、开发的协作，最后落在 TDD 上。课就上到这里，我最后说两句收尾的话。

【1.61】感谢聆听（2分钟）

好，同学们，这一章到这里就讲完了。我用最后两分钟把第 1 章收个尾。

按课件备注里给的口径，我在这里应该说的话是："**以上是本节课的内容**""**这一章讲到这里**"——这两句话我今天就照着说：**以上是本节课的内容，第 1 章"引论"讲到这里。**

我们从头回顾一下这一章走的路：我们先问了"**为什么必须做软件测试**"，用真实事故说明**缺陷的代价是真实的、而且发现得越晚代价越高**；接着我们从**质量、风险、经济、Test Oracle、批判性思维**五个视角认识了测试；然后讲了**正向思维和逆向思维**——它们不是对错之分，而是两种会直接决定你测试行为的取向；再往下区分了**测试与 SQA**：一个是技术、一个是管理，一个面向产品、一个面向过程；又讲了**测试与开发**：不是上下道工序，而是并行协作，要**测试左移**；最后落在 **TDD 思想**上——**测试在前，开发在后**，用先写下来的客观标准，去评自己写的东西。

这一章的名字叫"引论"，我希望大家现在就记住这一章真正留下的东西，它不是某个定义，而是**一个态度**：**面对软件，不要假设它是对的；作为测试的人，你的价值不在于证明它对，而在于比别人更早发现它不对。**

课下的两件事我再说一遍：**第一，整理 5 个真实软件缺陷事故，标注类型；第二，用自己的话写出软件测试的定义，并说明它和调试、质量保证的区别。**同时别忘了装好实验环境，完成第一次测试体验——**找出给定小程序里至少两个缺陷，截图记录下来。**

我们下一次课讲**第 2 章 软件测试的基本概念**，会讲清楚测试的分类、测试的级别、静态测试和动态测试、黑盒白盒灰盒。**带着今天这个"先假设它有错"的心态来，第二章会更好懂。**

周宇文，广东金融学院。**以上就是本节课的全部内容，谢谢大家，下课。**

> **板书：**正中间写"**感谢聆听**"；右上角竖排写本章六节回放：**1.1 必要性 → 1.2 价值与成本 → 1.3 定义与正反思维 → 1.4 SQA → 1.5 测试与开发 → 1.6 TDD**；左下角写本章一句话态度：**不假设它对，只求更早发现它不对**；右下角写作业与预习：**作业：5 个事故 + 定义辨析；实验：找 2 个缺陷并截图；预习：第 2 章 测试的分类与级别。**

> **提问：**这一章我们从事故讲到了 TDD。我想请大家用一句话回答：如果只能从这一章带走一样东西，你希望带走的是哪一个判断——是"测试很重要"，还是"测试要早"，还是"测试要抱着怀疑的心态"？请说你选它的理由。

> **预设回答：**第一种回答会说"测试要早，因为成本是越晚越高的，早就是省钱"。这个回答很实在，要肯定它

> 的经济逻辑，同时补一层："**早**是效率层面的答案，但'早'是为了什么？是为了让发现缺陷的那一秒钟尽量靠前，所以还得配上怀疑的心态，早才有意义。"第二种回答会说"要抱着怀疑的心态，因为正向思维容易走过场"。这是最贴近本章立意的答案，要直接落到板书上："对，**这一章所有的方法论，最后都要靠这个心态来落地**——心态不对，再好的流程也是走形式。"第三种回答会说"测试很重要"。这个回答没错，但太笼统，可以善意地推一把："'重要'是一个结论，**这一章我们要的不是结论，而是可执行的态度**——所以请你再往前一步，把'重要'翻译成一个具体动作：比如，从下次写作业开始，先写一份'它应该怎样'的清单，再去实现。"这样一问，学生就能把"重要"落到"测试先行"上，正好和下一章的测试分类衔接起来。

易错点 / 考点：一是**本章标题是"引论"，考纲里的落点是"理解"层级**，重点在**概念、思维与关系**（测试与 SQA、测试与开发、正反思维），而不是具体技术，答题时不要跑到第 3 章之后的测试方法上去。二是**回顾本章的六节顺序**，别把 **1.4 测试与 SQA** 和 **1.5 测试与开发** 调换——考纲里的顺序就是教材顺序。三是**收尾语的要求**：按课件备注，每课结束时要用"**以上是本节课的内容**""**这一章讲到这里**"这两句话，这是课堂规范的固定动作，答辩或录课时不要漏。

> **过渡：**第 1 章到此全部结束。下一节课我们进入**第 2 章 软件测试的基本概念**，从"软件测试分哪几类、有几个测试级别"开始，把今天建立的这些认识落到具体的方法体系上。我们下次课见。

四、思政落点清单（共 13 处）

序号	所在页	落点主题
1	1.3	这二十来个词里，"清除"两个字最见职业底线。找缺陷是技术活，报缺陷是良心活。行业
2	1.13	迪斯尼这个案例里，损失最大的是什么？是那些圣诞节早晨失望的孩子和他们父母的信任。
1	1.14	这个案例最扎心的地方在于：问题不是测不出来，而是一开始选择了"先发货，再说"。企
2	1.16	波音这个案例讲的不是技术不够，而是把质量控制让位给了进度。可以让学生代入一个具体
3	1.17	骑士资本这类案例里，真正的风险往往不是"没人知道有毛病"，而是"有人知道了但心存
4	1.20	这个故事没什么高科技含量，但它最能让学生理解"用户视角"。对我们来说，那只是一个
5	1.21	讲到 28 名士兵、4 位病人的时候，我们要停一下。这些数字背后，是一个个再也回不来的

1	1.28	讲国外真实事故——Therac-25 放疗仪因软件缺陷导致患者受到过量辐射致死，骑
2	1.39	讲国产游戏《血狮》（1997）——宣传声势远超其实际被验证的质量，交付后口碑崩塌
43	1.43	讲什么故事：课件这句"测试成本要小于缺陷造成的损失，测试才有意义"，背后是企业实
45	1.45	讲什么故事：批判性思维这一页，考验的其实是一个人的诚信与科学态度。怎么联系知识点
49	1.49	讲什么故事：课件特意强调 SQA 要"督促测试工作的结果客观、准确和有效"，这句话说
52	1.52	讲什么故事：这一页讲协作，最能体现一个团队的质量文化。怎么联系知识点：测试与开发