ⓘ 没有整理不会考的内容

0 软件全方位缺陷检测技术

1 引言

什么是软件（内涵）

- 软件是一系列按照特定顺序组织的计算机指令和数据的集合。
- 软件 = 程序 + 数据 + 文档

软件的三种类型（外延）

1. 编程语言：用于编写其他软件的软件
2. 基础软件：用于开发其他软件或者支撑其他软件运维的软件
3. 应用软件（特定领域软件）：用于处理和管理应用领域业务的软件

核心关注点

- 用户和开发人员关心的永远是这几个方面：
  - 软件产品能力（capability）：能做什么，是否有新功能
  - 软件产品质量（quality）：有多好，性能如何，是否易维护
  - 软件开发和运维成本（cost）：需要多少钱，多少人力
  - 软件开发效率（efficiency）：开发时间需要多久，使用有多方便
- 研究新语言、新方法、新技术、新模型、以及如何培养新人才，便于我们更好地满足用户和开发人员需求
  - 软件产品能力需求：数据处理、文档处理、语音处理、图像处理、视频处理……
  - 软件产品质量需求：功能正确性和完整性、性能、安全、可靠、可信……
  - 软件开发效率需求：开发效率、维护效率、管理效率、测试效率……
  - 软件运维成本需求：开发成本、运行成本、维护成本、更新换代成本……

什么是软件质量?

- **内涵**：
  1. 概括地：软件与明确地和隐含地定义的需求相一致的程度。
  2. 具体地：软件与明确地叙述的功能和性能需求、文档中明确描述的开发标准以及任何专业开发的软件产品都应该具有的隐含特征相一致的程度。
- **外延**：
  1. 软件的各种质量：产品质量（数据、代码、文档、模型、算法等）、过程质量、客户满意度等；
  2. 各种软件的质量：基础软件质量（编程语言、操作系统、数据库、工具软件、平台软件、中间件等）、应用软件质量（嵌入式软件、智能控制软件、管理软件……）。

如何评估软件质量?

质量评估模型 + 质量标准

2 软件全生命周期质量保障问题

何谓全生命周期软件质量保障?

不能过于依赖后期测试

- V：软件生命周期后期阶段的质量保障
- W：单个生命周期中每个阶段的质量保障
- 螺旋：多生命周期中每个生命周期的每个阶段的质量保障

全生命周期软件质量如何保障?

Two Legs/Ways

- 先验方法：Software development management
  1. Formal Specification
  2. SPI
  3. Architecture
  4. CMM（软件能力成熟度模型，Capability Maturity Model for Software）
  5. TQM（全面质量管理，Total Quality Management）
- 后验方法：Software defect / Fault detection
  - 测试、仿真、分析、度量等

早期阶段质量保障为什么重要

1. 降低风险
2. 软件缺陷越早发现，修复代价越小

何为早期阶段?

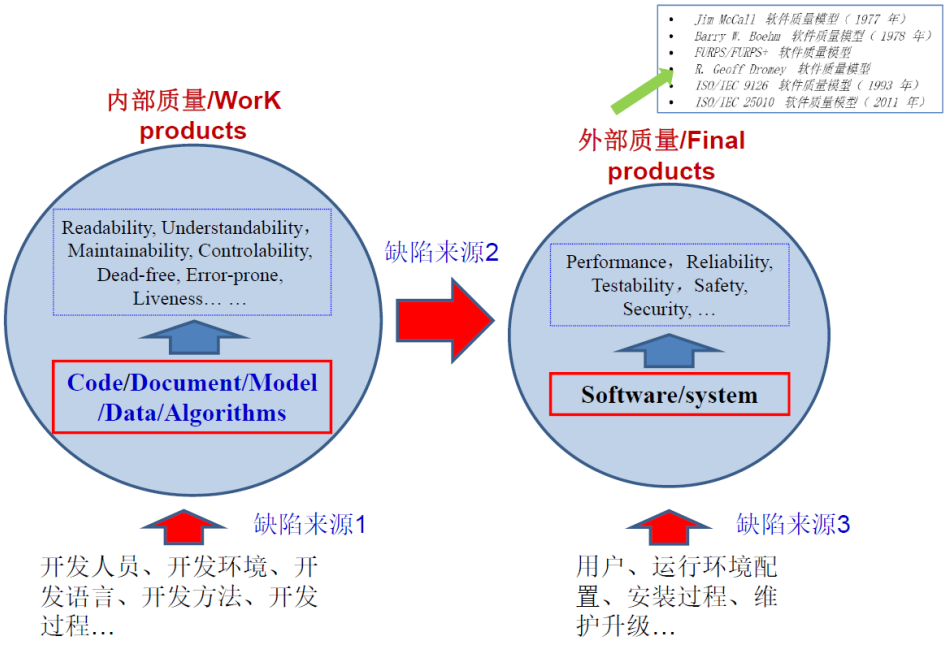
- 分析阶段
  1. 应用需求分析阶段（现实世界）：草图、非结构化文档
  2. 软件需求分析阶段（计算机世界）：结构化文档、UC、SRS
- 设计阶段：
  1. 架构设计
  2. 模块设计
  3. 数据库设计
  4. 接口设计
  5. 算法设计

早期软件质量保障方法

9 种方法，可验证如下：

1. 可靠性
2. 安全性
3. 正确性
4. 易用性
5. 可维护性
6. 互操作性

软件质量和软件缺陷的关系



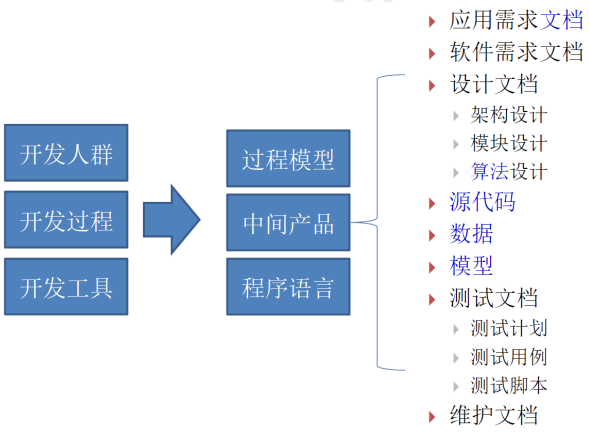
软件缺陷

- 软件缺陷（Defect），计算机软件或程序中存在的某种破坏正常运行能力的问题、错误，或者隐藏的功能缺陷。
- 缺陷的存在会导致软件产品在某种程度上不能满足用户的需要。
- IEEE729-1983 定义：从产品内部看，缺陷是软件产品开发或维护过程中存在的错误、毛病等各种问题；从产品外部看，缺陷使得系统所需要实现的某种功能的失效或违背，可能使软件发生故障（Faults）。

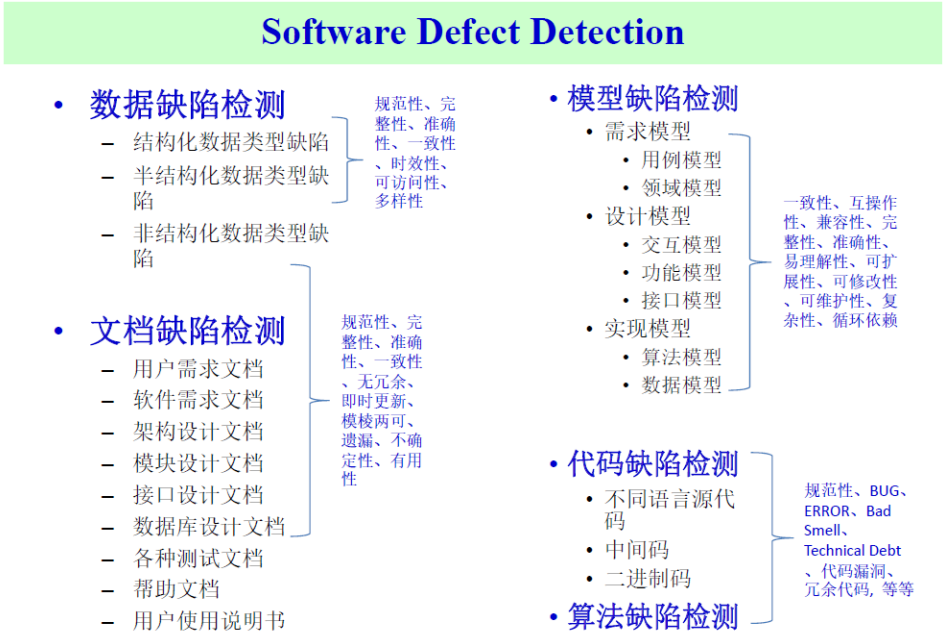
软件缺陷表现

- 错误：错误算法、模型、代码、公式、单位……
- 复杂度高：难以理解、不易修改、不易维护
- Bad Smell：一些不好的设计、代码
  - Defect, Bug, Flaw, Error, Fault, Bad smell, Technical debt, Weakness, Vulnerability……
- 代码冗余：多余的、重复的代码
- 不可行路径：从来不会被执行的程序路径

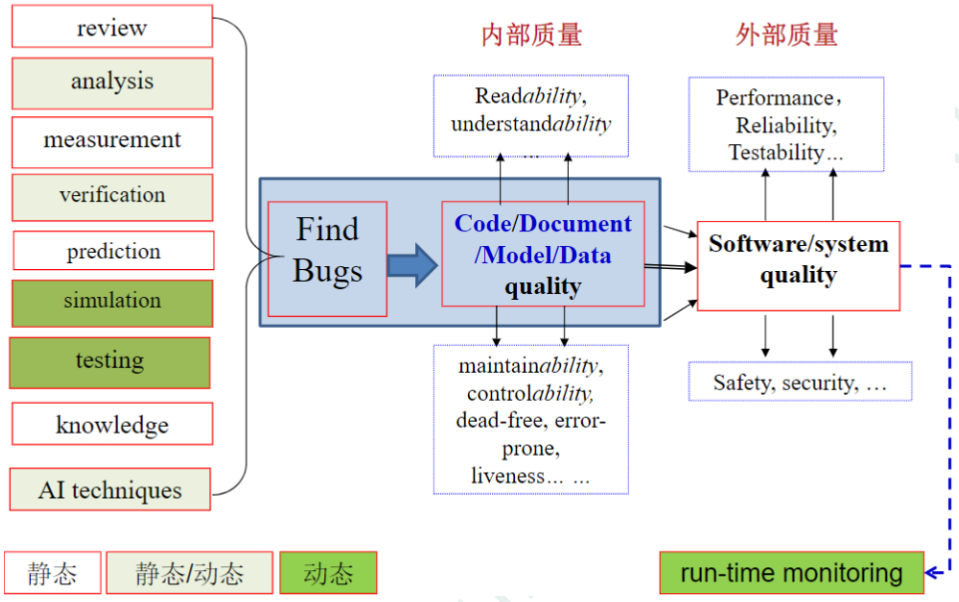
软件缺陷分布



Software Defect Detection



3 软件全方位缺陷检测的主流方法



1. 评审方法： 利用走查、检查单、审计、代码阅读等方式进行人工或自动评审，发现描述规范性、完整性、一致性、冗余等方面的缺陷。

2. 分析方法： 从控制流分析、数据流分析、代码坏味道检测、修改影响分析、路径剖析的角度进行代码层面的缺陷检测。
3. 度量方法： 度量设计和代码的好坏，发现设计和代码的 BUG。
4. 验证方法：从模型检验的角度检测和定位系统的安全性、一致性等时态属性存在的问题。
5. 仿真方法： 通过仿真找出系统设计的性能缺陷。
6. 测试方法： 从软件功能测试和非功能测试进行软件缺陷检测。
7. 监测方法： 通过软件运行过程中各种数据的监测，可以发现软件中存在的问题。
8. 基于知识方法：利用领域知识检查设计和代码中存在的缺陷。
9. 智能化方法：利用机器学习的方法发现缺陷。

A 评审方法

- 规范性检查
  - 文档规范性检查：完整性、一致性、冗余描述、错误描述、模糊描述
  - 模型规范性检查：完整性、一致性
  - 代码规范性检查：完整性、一致性、正确性
  - 数据规范性检查：完整性、一致性
  - 算法规范性检查：输入/输出、确定性、有限步骤
  - 编码规则检查：编码规则冲突
- 技术评审（technical review）
  - Walk through：技术合理性、可行性
  - Inspection：特定方面的合理性、可行性、正确性
  - Auditing：标准符合度、政治文化、安全可靠、可信
  - Code reading：人工/自动阅读代码

B 分析方法

- 静态分析方法（static analysis）
  - 控制流分析（CFA）
  - 数据流分析（DFA）
  - 关联关系分析
  - 调用依赖关系分析（CRA）
  - 层次依赖关系分析（HRA）
  - 数据依赖关系分析（dependence relationship）
  - 控制依赖关系分析（dependence relationship）
  - 传递依赖关系分析（dependence relationship）
  - 循环依赖关系分析（dependence relationship）
  - 静态程序切片
  - 修改影响分析（change impact analysis）
- 动态分析方法（dynamic analysis）
  - 修改传播分析（change propagation analysis）
  - 路径剖析（path profiling）
  - 执行轨迹跟踪
  - 动态切片
- 混合分析方法（hybrid analysis）
  - 符号执行

- 半静态程序切片
- 条件程序切片

控制流分析

- Control-flow analysis discovers the flow of control within a procedure (e.g., builds a CFG) or between procedures (ICFG)
- Representing Control-Flow
  - Implicit in AST
  - Control-flow graph
  - Control dependences in program dependence graph
- Why do we perform Control-flow Analysis
  - Loops are important to optimize
  - Programmers organize code using structured control-flow (if-then-else, for-loops etc.)
- Measurement：
  - 入度/出度 fan-in/fan-out
  - 耦合度 coupling
  - 圈复杂度 McCabe's Cyclomatic Complexity: CC=number of basis paths

数据流分析

- Data-flow analysis provides information for these and other tasks by computing the flow of different types of data to points in the program
- For structured programs, data-flow analysis can be performed on an AST; in general, intra-procedural (global) data-flow analysis performed on the CFG
- Exact solutions to most problems are un-decidable
  - May depend on input
  - May depend on outcome of a conditional statement
  - May depend on termination of loop

Data-flow analysis is approximate

- Approximate analysis can overestimate the solution:
  - Solution contains actual information plus some spurious information but does not omit any actual information
  - This type of information is safe or conservative
- Approximate analysis can underestimate the solution:
  - Solution may not contains all information in the actual solution
  - This type of information in unsafe
- For optimization, need conservative, safe analysis
- For software engineering tasks, may be able to use unsafe analysis information
- Biggest challenge for data-flow analysis: how to provide safe but precise (i.e., minimize the spurious information) information in an efficient way

Two kinds of DFA

- Reaching Definitions

    - for each basic block (program statements) s find which of all definitions in the program reach the boundaries of s.
- Liveness analysis
    - a variable is live at a particular point in the program if its value at that point will be used in the future (dead, otherwise) ,so, to compute liveness at a given point, need to look into the future

Applications of Data Flow

- Find useless variables
- Find key variables
- Tracing variables
- Data flow diagram construction
- Useful for data-flow testing

程序切片

给定某个程序的兴趣点和兴趣变量（或变量集合），即二元组 (S, V)，观察程序中哪些语句对 S 处的变量 V 有影响，或者观察 S 处的变量 V 对程序中哪些语句有影响。

修改影响分析

当对软件进行修改时，肯定会对软件的其他部分造成一些潜在影响，从而带来软件的不一致性，如果实施该修改所需的成本比较高（或者影响范围比较广泛），甚至超过重新开发该软件所需的成本（几乎影响整个系统），那么就需要考虑其他代替修改方案或者重新开发软件，而如果接受了修改，我们需要准确地预测修改带来的波动效应，而准确地对波动效应进行预测既可以提高维护人员实施修改的信心，又可以帮助维护人员准确地找到需要进行二次修改的程序代码，从而节省了维护时间。

路径剖析

- 动态程序分析是基于程序执行的分析技术，所以收集程序的执行信息是动态分析方法不可缺少的一部分。
- 路径剖析是收集程序执行信息的重要手段。与程序追踪相比，其缺少对数据流信息的记录，但是耗费低廉；与边的剖析相比，其耗费略高，但是提供的信息远多于边的剖析。

C 度量方法

- 软件产品度量/软件质量度量
    - 软件质量度量模型：ISO 9126、CMM、
- 软件过程度量/过程质量度量
    - 软件过程度量主要包括三大方面的内容：
        1. 成熟度度量（maturity metrics）：主要包括组织度量、资源度量、培训度量、文档标准化度量、数据管理与分析度量、过程质量度量等等；
        2. 管理度量（management metrics）：主要包括项目管理度量（如里程碑管理度量、风险度量、作业流程度量、控制度量、管理数据库度量等）、质量管理度量（如质量审查度量、质量测试度量、质量保证度量等）、配置管理度量 (如式样变更控制度量、版本管理控制度量等）；
        3. 生命周期度量（life cycle metrics）：主要包括问题定义度量、需求分析度量、设计度量、制造度量、维护度量等。
    - 一般流程：确认过程问题；收集过程数据；分析过程数据；解释过程数据；汇报过程分析；提出过程建议；实施过程行动；实施监督和控制。
- 复杂度度量
    - 结构复杂度

    - Cyclomatic Complexity(CCN)： $V(G)=e-n+2$
    - Halstead 复杂度
- 模块内聚度量 TCC 和 LCC
- 扇入扇出度 FFC
- 模块间耦合度 CBO
- 模块间响应度 RFC
- 软件架构静态成熟度 SSAM
- 软件架构动态成熟度 DSAM
- 软件架构综合成熟度 ISAM

Cyclomatic Complexity: (McCabe's)

1. Computed in several ways:
    1. Edges – nodes +2;
    2. Number of regions in CFG;
    3. Number of decision statements + 1 (if structured)
2. Indication of number of test cases needed
3. indication of difficulty of maintaining

D 验证方法

- **定理证明**：定理证明方法是一种将模型抽象为逻辑公式，然后使用自动的逻辑推理技术来验证电路是否正确的技术，定理证明方法十分严格，跟数理逻辑结合十分紧密，一般使用高阶逻辑（Higher-Order Logic，HOL）系统来进行证明。
- **模型检验**：主要是检查 RTL 代码是否满足规范中规定的一些特性。
- **等效性检验**：主要是验证在一个设计经过变换之后，穷尽地检验变化前后的功能的一致性。

形式化验证

- 形式化验证包括模型检验、定理证明和逻辑推理等。
- 形式化方法的实质是以逻辑、自动机、代数和图论等数学理论为基础，用一套特定的符号和技术对软件系统进行描述和分析，以期提高软件可靠性。
- 形式化方法的研究意义主要有以下几个方面：提供描述手段，以精确、无二义地描述系统赋予程序意义；提供分析手段，以证明系统正确性，或帮助开发人员找出系统出错原因提供开发方法和工具，以实现软件开发过程中全部或部分开发活动的自动或半自动化。

E 仿真方法

1. 架构仿真
    1. 性能仿真
    2. 可维护性仿真
    3. 可靠性仿真
2. 系统仿真
    1. 性能仿真
    2. 可维护性仿真
    3. 可靠性仿真

F 测试方法

1. 结构化测试

2. 功能测试
3. 性能测试
4. 安全测试

G 监控方法

- Runtime Monitoring：
  - Instrument
  - Tracing Program Execution
  - Aspect Programming
  - Logging
  - Observer 模式

H 基于知识方法

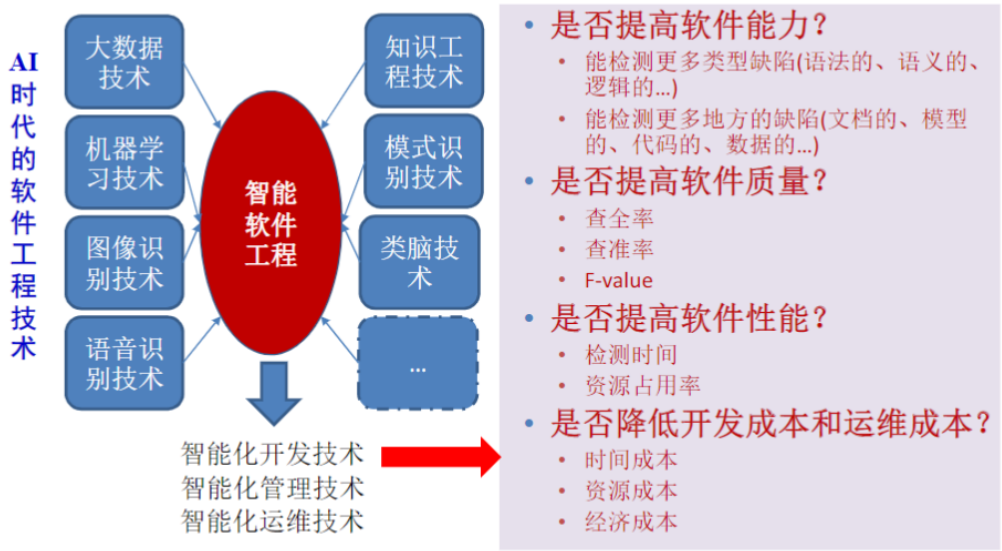
核心思想：利用知识图谱描述领域知识及关系，获得领域知识图谱，再构建问题知识图谱（例如程序的知识图谱），检测其中关系、实体是否一致。

- 例如，可以检查领域 KG 和问题 KG 的三元组（实体、关系、实体）之间是否存在不一致。

I 智能化方法

利用机器学习的各种模型

- 数据集/训练模型：项目 - 测试需求，测试需求 - TC，测试需求 - Tscript，项目 - 测试报告…
- 测试用例自动生成、选择和优先排序
- 测试脚本自动生成
- 缺陷预测：数量、出错倾向性、缺陷密度、缺陷严重性、缺陷分布情况



4 软件全方位缺陷预测的主流方法

预测技术的应用范围，或者说：待预测项目的范围。

1. 版本内预测
2. 跨版本预测
3. 跨项目预测
4. 跨组织/企业预测
5. 跨领域/行业预测
6. 跨国家/地区预测

缺陷预测的主流方法

1. 基于统计：统计抽样
2. 基于软件度量：圈复杂度、Halstead、LOC、功能点
3. 基于数据：大数据、演化数据、开发过程数据
4. 基于学习：集成、深度、半监督、标签
5. 基于特征：特征选择、特征迁移
6. 基于网络：神经、贝叶斯、复杂
7. 基于相似性：相似性/不相似性度量
8. 基于切片：静态、动态
9. 基于信息：信息熵、信号量
10. 基于向量机：SVM、向量回归
11. 基于 XX 算法：粒子群、蚁群、鱼群

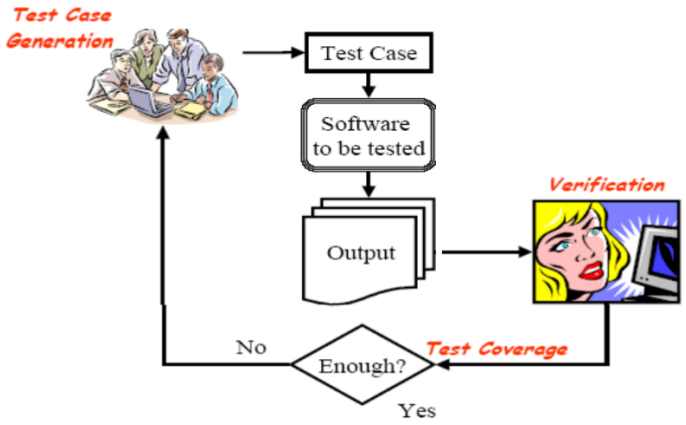
1 软件测试

1. 软件测试是什么？测试概念：静态测试、动态测试；白盒测试、黑盒测试、灰盒测试；单元测试、集成测试、系统测试、验收测试，回归测试等。
2. 如何进行测试？测试过程：测试需求、测试计划、测试用例生成、测试执行、测试结果与预期结果（test oracle）比较、测试充分性评估、测试报告。
3. 有哪些测试方法？测试方法：蜕变测试、随机测试、组合测试、基于模型测试……
4. 都能测试什么系统？测试应用：构件软件测试、OO 测试，Web 测试、服务测试、云测试…
5. 软件测试难点有哪些？测试挑战：1）测试用例自动生成问题；2）测试预言（test oracle）问题；3）测试充分性问题；

- 其他知识点
  - 测试模型
  - 测试工具
  - 测试自动化
  - 测试智能化
  - 测试脚本
  - 测试优先级
  - 集成测试序

What is testing?

- Software testing is a set of processes aimed at investigating, evaluating and ascertaining the completeness and quality of computer software. Software testing ensures the compliance of a software product in relation with regulatory, business, technical, functional and user requirements.
- 软件测试是一组用来促进鉴定软件正确性、完整性、安全性和质量的过程。换句话说，软件测试是一种实际输出与预期输出之间的审核或者比较过程。
- 软件测试的经典定义是：在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足需求进行评估的过程。

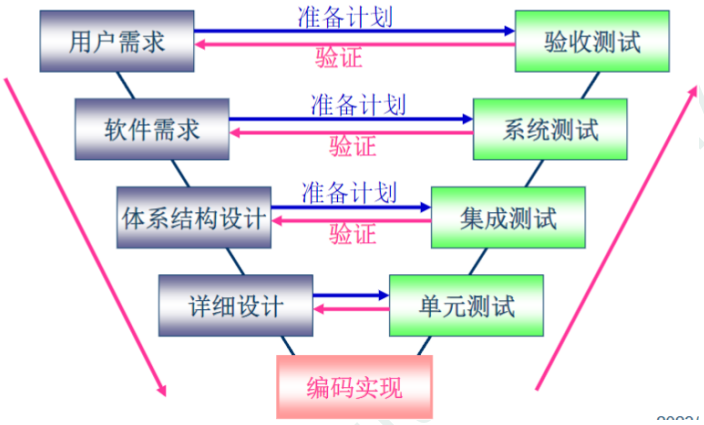


Basic Problems of Testing

- Adequacy of Test Suite
  - Test Criteria
  - Test case generation & selection
- Oracle
  - Expert knowledge
  - Metamorphic relation

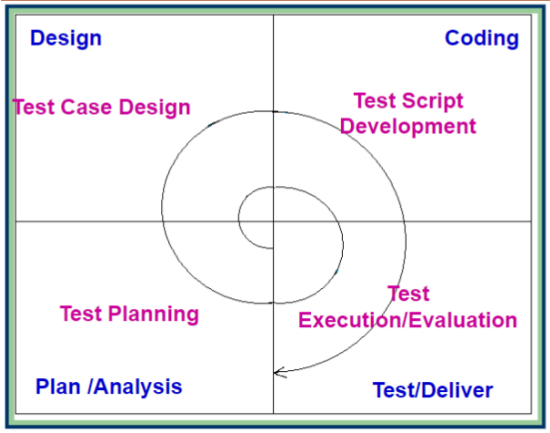
Software & Testing

1. Testing with V-Model



2. Test constantly and as early as possible

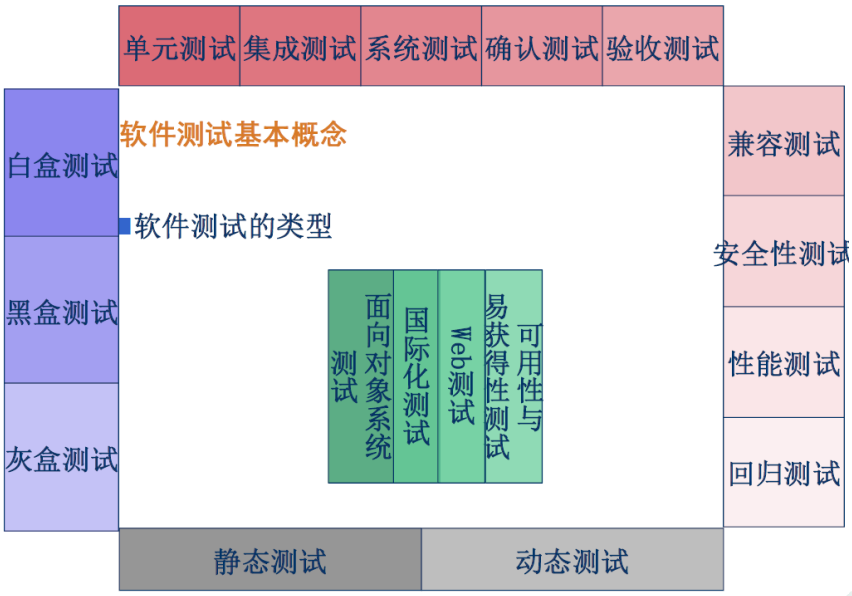
经常测试、尽早测试



3. Service Testing

- WSDL → Test cases
- Function Testing
- Volume/Load Testing
- Regression Testing
- Reliability testing

4. Cloud Testing: testing cloud



2 经典软件测试理论

- 软件测试就是确定合适的测试用例进行测试；
- 测试过程贯穿在整个软件生命周期中，不断迭代；
- 测试阶段：单元测试、集成测试、系统测试、验收测试
- 测试方法：动态、静态、黑盒、白盒等
- 测试类型：可靠性、功能、性能等
- 测试策略：评估标准 + 测试方法
  软件测试可以发现代码中绝大多数 BUG，但是软件测试没有发现 BUG，并不表示代码中不存在 BUG！

1. 开发者测试

开发者测试（Developer Testing，DT），是指开发者所做的测试，有别于专职测试人员（来自测评机构）进行的测试活动。DT 目标是在软件交付转验收测试前，发现和解决绝大多数代码缺陷，而其理论依据是业界研究反复揭示的“前端发现问题的代价远小于后端”。

单元测试

- 侧重于核实软件的最小可测试元素
- 通常应用于实现模型中的模块（函数、类、WSDL 服务等）
- 主要做：功能正确的前提下的控制流和数据流的覆盖测试
- 主要针对单元的内部结构
- 更注重白盒测试

单元测试内容

- 算法和逻辑（不能直接测试到）

- 单元接口
- 数据结构
- 边界条件
- 独立执行
- 错误处理

例子

1. 模块接口测试：检查进出模块的数据是否正确
2. 模块局部数据结构测试：检查局部数据结构能否保持完整性
3. 模块边界条件测试：检查临界数据是否正确处理
4. 模块独立执行通路（路径）测试：检查由于计算错误、判定错误、控制流错误导致的程序错误
5. 模块内部错误处理测试：检查内部错误处理设施是否有效

集成测试

为什么进行集成测试?

1. 一个模块可能对另一个模块产生不利的影响。
2. 将子功能合成时不一定产生所期望的主功能。
3. 独立可接受的误差，在组装后可能会超过可接受的误差限度。
4. 可能会发现单元测试中未发现的接口方面的错误。
5. 在单元测试中无法发现时序问题（实时系统）。
6. 在单元测试中无法发现资源竞争问题。

集成是错误的高发阶段

- 集成的单元能够正确的执行用例。
- 测试对象是实现模型中的一个包或一组包。
- 要集成的包通常来自于不同的开发组织。
- 集成测试将揭示包接口规约中不够完全或有错误的地方。

集成测试的方法

- 非增式集成测试：采用一步到位的方法来构造测试对象：对所有模块进行个别的单元测试后，按程序结构图将各模块联接起来，把联接后的程序当作一个整体进行测试。
- 增式集成测试 ： 把下一个要测试的模块同已经测试好的模块结合起来进行测试，一次增加一个要测试模块。
  - 自顶向下结合
  - 自底向上结合

自顶向下增式集成

1. 主控模块作为测试驱动，所有与主控模块直接相连的模块作为桩模块；
2. 根据集成的方式（深度或广度），每次用一个替换从属的桩模块；
3. 在每个模块被集成时，都必须已经进行了单元测试；
4. 进行回归测试以确定集成新模块后没有引入错误
5. 上述过程从第 2 步重复进行，直到整个系统结构被集成完成。

自底向上增式集成

1. 组装从最底层的模块开始，组合成一个构件，用以完成指定的软件子功能；

2. 编制驱动程序，协调测试用例的输入与输出；
3. 测试集成后的构件；
4. 按程序结构向上组装测试后的构件，同时除掉驱动程序。

两种增式集成测试方法的比较

	优点	缺点
自顶向下集成	可以自然地做到逐步求精，一开始便能让测试者看到系统的框架	▪ 需要提供桩模块 ▪ 在输入/输出模块接入系统以前，在桩模块中表示测试数据有一定困难 ▪ 由于桩模块不能模拟数据，如果模块间的数据流不能构成有向的非环状图，一些模块的测试数据难于生成； ▪ 观察和解释测试输出往往也是困难的
自底向上集成	▪ 由于驱动模块模拟了所有调用参数，即使数据流并未构成有向的非环状图，生成测试数据也没有困难 ▪ 特别适合于关键模块在结构图底部的情况	▪ 直到最后一个模块被加进去之后才能看到整个程序（系统）的框架 ▪ 只有到测试过程的后期才能发现时序问题和资源竞争问题

	非增式集成	**增式集成**
工作量	大	小
接口错误	发现错误较晚	发现错误早
错误定位	难	易
测试程度	不彻底	彻底
需要的机器量	少	多
测试的并行性	好	差

系统测试

- 所有的集成测试完成
- 软件系统之间的联合测试
- 软件、硬件等之间的联合测试
- 模拟真实运行环境的测试

验收测试

- V 模型中测试的最后一道工序
- 用户在场或者直接测试
- 用户可能自定义测试用例

α测试和β测试

通常由用户或其他人（非开发人员和测试人员）来完成

- **α测试**是在开发即将完成时对应用进行的测试，此时仍然允许对设计作微小的变动。用户在开发环境下进行，或开发机构内部用户在模拟实际操作环境下进行。这是在受控的环境下进行的测试。开发者坐在用户旁边，随时记录下错误情况和使用中的问题。

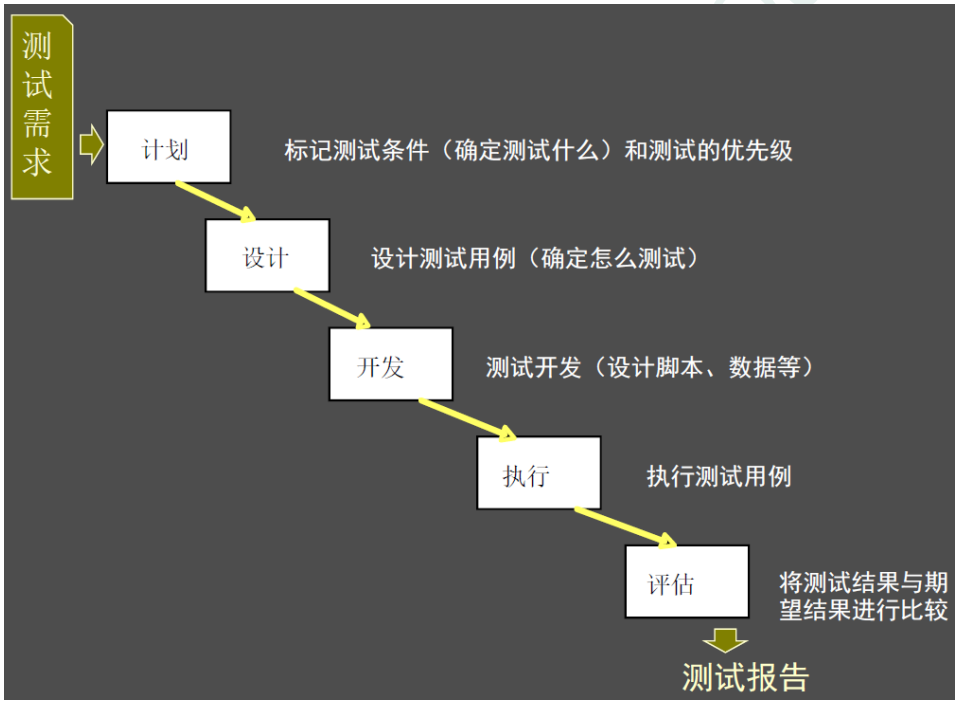
- **β测试**是一种软件发布之前的验收测试。一般根据产品规格说明书严格检查产品，对照说明书上对软件产品所做出的各方面要求，确保所开发的软件产品符合用户的各项要求。软件的多个用户在实际使用环境下进行测试。开发者通常不在测试现场，不能由程序员或测试员完成。

回归测试

- 目标：
  1. 修改的或增加的部分是正确的
  2. 没有引起其他部分产生错误
- 应用：
  1. 增量开发
  2. 版本控制
  3. 软件维护
- 方法举例：
  - 全部再测试 (Retest All)
  - 按风险再测试 (Retest Risky Use Case)：rerun by risk
  - 按频率再测试 (Retest by Profile)：rerun by allocating time in proportion to operational profile
  - 按修改再测试 (Retest Changed Segments)：rerun by comparing code changes
  - 按依赖再测试 (Retest Within Firewall)

2. 测试过程 (面向测试任务)

包括：测试需求描述、测试计划、测试用例生成、测试执行、测试结果与预期结果 （test oracle） 比较、测试充分性评估、测试报告生成等。



测试需求

在项目中要测试什么。需要明确测试需求（What），才能决定怎么测（How）、测试时间（When）、需要多少人（Who）、测试的环境是什么（Where），测试中需要的技能、工具以及相应的背景知识，测试中可能遇到的风险等等，以上所有的内容结合起来就构成了测试计划的基本要素。

计划

测试目标

- 单元测试：Unit Testing
  - 检验程序最小单元有无错误
    - 接口、数据结构、边界、覆盖、逻辑
  - 检验单元编码与设计是否吻合
- 集成测试：Integration Testing
  - 检验组成系统的模块接口有无错误
  - 代码实现的系统设计与需求定义是否吻合
- 系统测试：System Testing
  - 检验组成整个系统的代码、以及系统的软硬件配合有无错误
  - 代码实现的系统与用户需求是否吻合
  - 检验系统的各种文档等是否完整、有效
  - 模拟验收测试的要求，检查系统是否符合用户的验收标准
- 验收测试：Acceptance Testing
  - 使客户验收签字
  - 系统是否符合事先约定的验收标准
- 回归测试：Regression Testing
  - 验证程序修改或者版本更新以后，以前正确的功能和其他指标仍旧正确。

测试范围

- 接口测试：哪些子系统的接口需要测试
- 性能测试，负载测试等：哪些功能要做性能测试

测试项目

- 数据和数据库集成测试：验证×记录并发访问…
- 功能测试：依据×文档，×.×节的“特征”
- GUI 测试：依据规格说明书的相关叙述
- 性能测试：测试访问某功能的响应时间
- 安全测试
- 压力测试
- 负载测试
- 容量测试
- 失效/恢复测试
- 安装测试
- 配置测试

测试策略/手段

- Code coverage strategies, e.g.
  - Decision coverage
  - Data-flow testing (Defines -> Uses)
- Specification-based testing, e.g.
  - Equivalence partitioning
  - Boundary-value analysis
  - Combination strategies
- State-based testing

测试工具

工具的类型、名称、版本等规定

- 测试管理
- 测试设计
- 缺陷跟踪
- 数据库工具
- 性能测试工具

测试资源

- 角色
  - 角色名称：经理，设计人，测试人等
  - 角色安排：
  - 责任
- 系统资源
  - 软件
  - 硬件

交付/产出物件

- 测试计划
- 测试环境
- 测试包
- 测试日志
- 缺陷报告

设计

- 测试用例设计：确认和详细描述测试用例
- 测试脚本设计：确认和设计测试脚本
- 覆盖准则设计：评估测试覆盖

测试脚本设计

- 测试脚本（Testing script），一般指的是一个特定测试的一系列指令，这些指令可以被自动化测试工具执行。
- 测试脚本可以被创建（记录）或使用测试自动化工具自动生成，或用编程语言编程来完成，也可综合前三种方法来完成。
- 测试脚本语言（test scripting language）

测试覆盖准则设计

几种白盒覆盖率

开发

1. 建立测试环境
2. 录制或编写测试脚本
3. 开发测试驱动器 (drivers)

4. 开发桩模块（stubs）
5. 建立外部数据集

执行

1. 执行测试脚本
2. 测试执行情况分析
3. 结果验证和确认
4. 研究未预期的结果
5. 写日志（logging）
6. Bug 报告/Bug 跟踪

评估

1. 分析测试用例覆盖
2. 分析代码覆盖
3. 分析缺陷
4. 分析是否达到测试停止、成功标准
5. 写测试分析报告

- 测试用例覆盖
  - Ratio Test Cases Performed = 110/120 = 92%
  - Ratio Test Cases Successful = 95/110 = 87%
- 代码覆盖
  - Ratio LOC executed = 94,399 / 102,000 = 93%
- 缺陷分析
  - 缺陷分布（严重性，源代码，时间长度）
  - 缺陷趋势
  - 缺陷状态（记录，提交，修改，测试，关闭）
- 图表

3. 测试方法分类

静态测试和动态测试

- 静态测试：用计算机测试源程序时，计算机并不真正运行被测试的程序，只对被测程序进行特性分析。常称为“静态分析”，是对被测程序进行特性分析的一些方法的总称。
- 动态测试：计算机真正运行被测试的程序，通过输入测试用例，对其运行情况（输入/输出的对应关系）进行检查和分析。通常意义上的测试。

区别：

1. 被测部分不同
   - 静态测试是指测试不运行的部分,只是检查和审阅,如规范测试、软件模型测试、文档测试等；动态测试是通常意义上的测试，也就是运行和使用软件。
2. 测试方式和方法不同
   - 静态测试通过文档评审、代码阅读等方式测试软件；动态测试通过运行程序的测试软件。
   - 静态测试不用执行程序，它主要采用代码走查、技术评审、代码审查的方法对软件产品进行测试；动态测试主要通过构造测试实例、执行程序、分析程序输出结果来对软件进行测试。

静态测试（又称静态分析）

- 定义：
  - 不实际运行程序，而是通过检查和阅读等手段来发现错误并评估代码质量的软件测试技术。也称为静态测试技术。
- 方法：
  - 走查：WalkThrough
  - 审查：Inspection
  - 评审：Review
  - 审计：Auditing

黑盒测试和白盒测试

- 黑盒测试（Black-box Testing）又称功能测试、数据驱动测试或基于规格说明的测试，是一种从用户观点出发的测试。用来证实软件功能的正确性和可操作性。
- 白盒测试（White-box Testing）又称结构测试、逻辑驱动测试或基于程序的测试。用来分析程序的内部结构

		黑 盒 测 试	白 盒 测 试
测 试 规 划		根据用户的规格说明，即针对命令、信息、报表等用户界面及体现它们的输入数据与输出数据之间的对应关系，特别是针对功能进行测试。	根据程序的内部结构，比如语句的控制结构，模块间的控制结构以及内部数据结构等进行测试。
特点	优 点	能站在用户立场上进行测试。	能够对程序内部的特定部位进行覆盖测试。
	缺 点	• 不能测试程序内部特定部位。 • 如果规格说明有误，则无法发现。	• 无法检验程序的外部特性。 • 无法对未实现规格说明的程序内部欠缺部分进行测试。
方 法 举 例		基于图的测试 等价类划分 边值分析 比较测试	语句覆盖 判定覆盖 条件覆盖 判定/条件覆盖 基本路径覆盖 循环覆盖

模拟用户操作的测试方法

- 基于对用户如何使用被测试软件的了解来进行测试的方法。
- 经验告诉我们，复杂的软件产品有许多错误，但用户一般只能找出这些错误中很少的一部分。
- 为给用户带来最大利益，要着重对那些用户可能发现的错误进行测试和修改工作。

4. 测试类型

1. 完整性测试：侧重于评估测试对象的强壮性（防止失败的能力），语言、语法的技术兼容性以及资源利用率的测试。该测试针对不同的测试对象实施和执行，包括单元和已集成单元。
2. 结构测试：侧重于评估测试目标是否符合其设计和构造的测试。通常对基于 Web 的应用程序执行该测试，以确保所有链接都已连接、显示正确的内容以及没有孤立的内容。
3. 配置测试：侧重于确保测试对象在不同的硬件和/或软件配置上按预期运行的测试。该测试还可以作为系统性能测试来实施。
4. 功能测试：侧重于核实测试对象按计划运行，提供需求的服务、方法或用例的测试。该测试针对不同的测试对象实施和执行，包括单元、已集成单元、应用程序和系统。
5. 安装测试：侧重于确保测试对象在不同的硬件和/或软件配置上，以及在不同的条件下（磁盘空间不足或电源中断）按预期安装的测试。该测试针对不同的应用程序和系统实施和执行。

6. 安全测试：侧重于确保只有预期的主角才可以访问测试对象、数据（或系统）的测试。该测试针对多种测试对象实施和执行。
7. 容量测试：侧重于核实测试对象对于大量数据（输入和输出或驻留在数据库内）的处理能力的测试。容量测试包括多种测试策略，如创建返回整个数据库内容的查询；或者对查询设置很多限制，以至不返回数据；或者返回每个字段中最大数据量的数据条目。
8. 基准测试：一种性能测试，该测试将比较（新的或未知的）测试对象与已知的参照负载和系统的性能。
9. 竞争测试：侧重于核实测试对象对于多个主角对相同资源（数据记录、内存等）的请求的处理是否可以接受的测试。
10. 负载测试：一种性能测试，用于在测试的系统保持不变的情况下，核实和评估系统在不同负载下操作极限的可接受性。 评测包括负载和响应时间的特征。如果系统结合了分布式构架或负载平衡方法，将执行特殊的测试以确保分布和负载平衡方法能够正常工作。
11. 性能测试：在该测试中，将监测测试对象的计时配置文件，包括执行流、数据访问、函数和系统调用，以确定并解决性能瓶颈和低效流程。
12. 强度测试：一种性能测试，侧重于确保系统可在遇到异常条件时按预期运行。系统面对的工作强度可以包括过大的工作量、不充足的内存、不可用的服务/硬件或过低的共享资源。
13. 恢复测试：（例如，硬件故障或用户不良数据引起的一些情况）
14. 兼容性测试： 对软硬件、操作系统、网络的兼容性测试
15. 比较测试： 与同类产品进行比较，列出优缺点
16. 算法测试： 确定是否已正确实现算法
17. 正向测试： 确定正确使用软件是软件某一特性产生的结果是否与需求一致
18. 逆向测试： 确定软件无效输入或非法操作的处理是否合理
19. 时序测试：测试软件执行的时间序列问题
20. 突变测试：（mutation testing） 判断测试用例是否有效的方法

5. 测试工具

- 测试用例设计工具
- 录制/回放工具
- 测试覆盖监视工具
- 测试结果比较工具
- 内存泄漏检查工具
- 缺陷跟踪工具
- 性能测试工具

测试工具在软件测试中的作用

1. 速度：自动化测试远高于手工执行测试
2. 效率和成本：推进软件开发进度、降低开发、测试和维护成本
3. 准确度和精度：工具能快速且无差错地执行测试用例或测试脚本

3 黑盒测试用例设计

1 黑盒测试概述

- 被测程序被当作一个黑盒，不考虑程序内部结构和内部特性，测试者只知道该程序输入和输出之间的关系或程序的功能，依靠能够反映这一关系和程序功能的需求规格说明书考虑确定测试用例并推断测试结果的正确性。

1.1 如何理解黑盒测试?

1. 有时无法获取程序代码
2. 可以发现其它测试遗漏的逻辑缺陷（如因果关系等）
3. 尽早进行黑盒测试可以尽早发现软件功能缺陷
4. 适用于各个测试阶段：单元测试，集成测试，系统测试，回归测试

定义

- 黑盒测试：一种基于规格说明（Spec），不要求考察代码，以用户视角进行的测试
- 其它称谓：功能测试、基于规格说明的测试

意义

- 黑盒测试有助于软件产品的总体功能验证：
  1. 检查明确需求和隐含需求
  2. 采用有效输入和无效输入
  3. 包含用户视角

实施者

- 专门的软件测试部门
- 有经验的测试人员

步骤

1. 熟悉规格说明书，理解测试需求;
2. 生成测试用例;
3. 执行测试;
4. 判定测试结果。

1.2 黑盒测试有什么特点?

黑盒测试特点：一种基于需求的测试

- 目的
  - 确认（V&V）软件需求规格说明书列出的需求是否都正确实现
- 前提
  1. 软件需求规格说明书已经过仔细评审
  2. 隐含需求已经明确化
- 需求跟踪矩阵（RTM）
- 优点
  1. 黑盒测试与软件具体实现无关，所以如果软件实现发生了变化，测试用例仍然可以使用;
  2. 设计黑盒测试用例可以和软件实现同时进行，因此可以压缩项目总的开发时间。

1.3 如何实施黑盒测试?

黑盒测试实施：正面测试和负面测试

正面测试

- 正面测试和正面测试用例：通过正面测试用例产生一组预期输出验证产品需求
- 目的：证明软件对于每条规格说明和期望都能通过

Example

需求标识	输入 1	输入 2	当前状态	预期状态
BR-01	号码为 123 的钥匙	顺时针转动	开锁	上锁
BR-01	号码为 123 的钥匙	顺时针转动	上锁	不变
BR-02	号码为 123 的钥匙	逆时针转动	开锁	不变
BR-02	号码为 123 的钥匙	逆时针转动	上锁	开锁

负面测试

- 负面测试和负面测试用例：展示当输入非预期输入时，产品没有失败（fail）
- 目的：产品没有设计、没有预想到的场景，尝试使系统垮掉

Example

序号	输入 1	输入 2	当前状态	预期状态
1	某个其它锁的钥匙	顺时针转动	上锁	不变
2	某个其它锁的钥匙	逆时针转动	开锁	不变
3	铁丝	逆时针转动	开锁	不变
4	用石头打击		上锁	不变

正面测试和负面测试比较

	正面测试	负面测试
测试条件	根据需求说明规格	测试条件都在需求规格说明之外
目的	验证需求规格说明书的需求是否都得到满足	通过非法输入尝试搞垮软件
覆盖率计算	覆盖需求和条件	没有覆盖率
挑战性	根据规格说明设计测试用例	需要高度创造性产生尽可能多的非法输入

2 黑盒测试用例设计和生成方法

做例题！！

2.1 等价类划分法

Summary

1. 什么是等价类?
2. 如何进行等价类划分?
3. 如何使用等价类生成测试用例?

用尽可能少的测试用例发现尽可能多的缺陷。

- 等价类划分—equivalence partitioning
- 等价类也可以是：测试相同目标或暴露相同软件缺陷的一组测试。
- 等价类是指输入域的某个子集，该子集中的每个输入数据对揭露软件中的错误都是等效的，测试等价类的某个代表值就等价于对这一类其他值的测试。

等价类划分方法的基础

- 原理：
  - 将程序的输入域划分为等价类，以便导出测试用例；
  - 它试图通过设计一个测试用例来尽可能发现多个错误，从而减少测试用例数目，降低测试工作量。
- 等价类（划分）：
  - 如果软件行为对一组值来说是相同的，则称这组值为等价类；
  - 产生相同预期输出的一组输入值叫一个划分

有效等价类和无效等价类

- 有效等价类：完全满足产品规格说明的输入数据，即有效的、有意义的输入数据构成的集合。
  - 利用有效等价类可以检验程序是否满足规格说明书。
- 无效等价类：不满足程序输入要求或者无效的输入数据构成的集合。
  - 利用无效等价类可以检验程序是否能够处理非法输入。

等价类划分准则

输入数据类型	划分等价类规则	
布尔值	1个有效等价类：TRUE	1个无效等价类：FALSE
取值范围	1个有效等价类：正确取值范围	2个无效等价类：大于和小于取值范围
数据个数	1个有效等价类：正确数据个数	2个无效等价类：大于和小于数据个数
集合	1个有效等价类：正确的集合取值	1个或多个无效等价类
需分别处理的一组输入数据	多个有效等价类：每个输入数据为1个等价类	1个无效等价类
符合某些规则的输入	多个有效等价类：符合某个规则的输入数据为1个等价类	若干个无效等价类

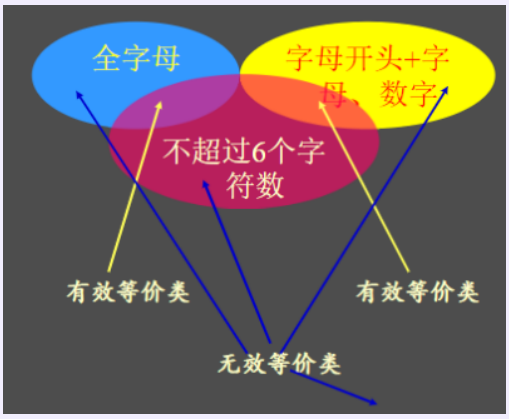
课件上有具体的例子

等价类划分方法的步骤

1. 选择划分准则（范围、取值、布尔、集合…）
2. 根据准则确定有效等价类和无效等价类
3. 从等价类中选取样本数据覆盖所有等价类
4. 根据需求写预期结果
5. 执行测试
   每个有效等价类都要覆盖，每个无效等价类都要单独覆盖

人寿保险

注册用户名



等价类划分考虑输入条件还是输出条件?

判断三角形类型

11 有效，34 无效，共 45；有效可用 5 个覆盖，故一共 39 个。

2.2. 因果图（判定表）法

Summary

1. 什么是因果图?
2. 如何根据软件需求画出因果图?
3. 如何生成判定表以及测试用例?

因果分析法背景

- 测试挑战：多个输入条件的关联问题
- 组合测试面临处理大量无效测试用例的现实

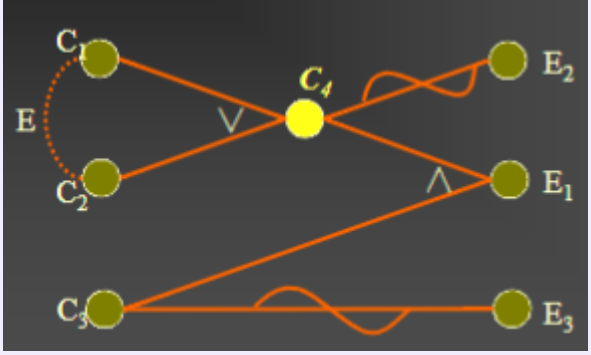
因果分析法基础

1. 软件的输入和输出之间存在因果逻辑关系，可以用因果图（cause-effect diagram）刻画；
2. 因果图可从规格说明书中获得

因果图的表示

- 关系：恒等（—）、非（~）、或（∨）、与（∧）
- 输入约束：互斥（E）、包含（I）、唯一（O）、要求（R）
- 输出约束：屏蔽（M）

文件管理系统



因果图列表

原因的所有组合及相应的结果组合
注意：某些原因的组合不存在

文件管理系统

因果图列表示例

		1	2	3	4	5	6	7	8
原因（可能输入）	C1	0	0	0	0	1	1	1	1
	C2	0	0	1	1	0	0	1	1
	C3	0	1	0	1	0	1	0	1
结果（预期输出）	E1	0	0	0	1	0	1		
	E2	1	1	0	0	0	0		
	E3	1	0	1	0	1	0		

生成判定表

方法：原因——判定表中的条件； 结果——判定表中的行动；
判定规则——原因与结果的组合。

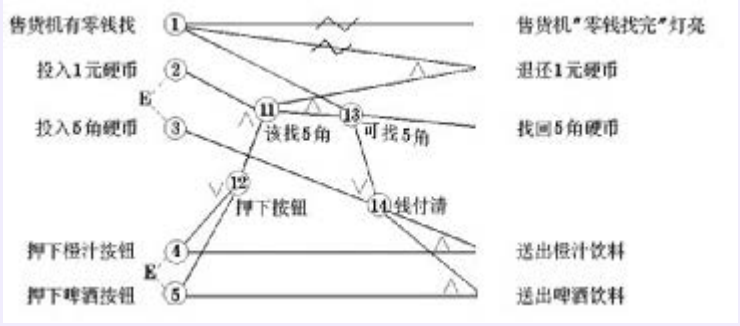
生成测试用例

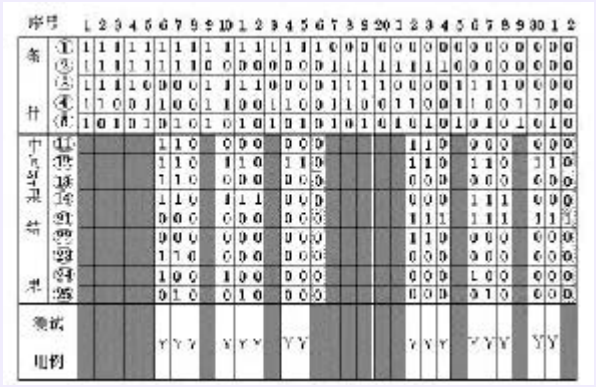
1. 判定表中的条件——测试用例的输入条件
2. 判定表中的行动——测试用例的期望输出
3. 一条判定规则——一个测试用例

文件管理系统

		1	2	3	4	5	6
条件（合法输入）	C1	0	0	0	0	1	1
	C2	0	0	1	1	0	0
	C3	0	1	0	1	0	1
行动（预期输出）	A1	0	0	0	1	0	1
	A2	1	1	0	0	0	0
	A3	1	0	1	0	1	0
测试用例	测试输入	DY	C2	BN	B5	AM	A3
	测试输出						

饮料自动售货机





因果分析法总结

1. 分析规格说明书，识别原因和结果
2. 在因果图连接原因和结果
3. 标明原因之间以及结果之间的约束条件
4. 因果图转换为因果图列表进而生成判定表
5. 判定表的规则转换为测试用例

判定表组成

4 种成份：

1. 条件桩：列出所有可能问题（条件）
2. 条件项：列出条件所有可能取值
3. 动作桩：列出可能采取的操作
4. 动作项：指出在条件项的各种取值情况下应采取的动作
   判定规则：贯穿条件项和动作项的一列

判定表简化

- 简化目标：合并相似规则
- 相似规则判断：
  - 有两条或以上规则具有相同动作，并且在条件项之间存在极大相似，便可以合并
- “—”：表示合并后该条件项与取值无关，称“无关条件”

机器维修

初始化判定表									
		1	2	3	4	5	6	7	8
条件	功率大于100马力	Y	Y	Y	Y	N	N	N	N
	维修记录不全	Y	Y	N	N	Y	Y	N	N
	运行时间超过10年	Y	N	Y	N	Y	N	Y	N
动作	优先维修	√	√	√	√	√	√	√	
	正常维修								√

初始化判定表化简					
		1 2 3 4	5 6	7	8
条件	功率大于100马力	Y	N	N	N
	维修记录不全	—	Y	N	N
	运行时间超过10年	—	—	Y	N
动作	优先维修	√	√	√	
	正常维修				√

优惠卡

		1	2	3	4	5	6	7	8	9	10	11
输入/条件	是否曾话费拖欠	Y	Y	Y	Y	Y	Y	N	N	N	N	N
	是否按时结欠	N	Y	Y	Y	Y	Y	—	—	—	—	—
	通话费用>=500	—	Y	N	N	N	N	Y	N	N	N	N
	通话费用[300，500)	—	N	Y	N	N	N	N	Y	N	N	N
	通话费用[200，300)	—	N	N	Y	N	N	N	N	Y	N	N
	通话费用[100，200)	—	N	N	N	Y	N	N	N	N	Y	N
	通话费用<100	—	N	N	N	N	Y	N	N	N	N	Y
输出/动作	钻石卡							√				
	金卡		√	√					√	√		
	银卡				√						√	
	无卡	√				√	√					√

		1	2，3	4	5，6	7	8，9	10	11
输入/条件	是否曾话费拖欠	Y	Y	Y	Y	N	N	N	N
	是否按时结欠	N	Y	Y	Y	—	—	—	—
	通话费用>=500	—	—	N	N	Y	N	N	N
	通话费用>=300	—	Y	N	N	N	—	N	N
	通话费用>=200	—	N	Y	N	N	Y	N	N
	通话费用[100，200)	—	N	N	—	N	N	Y	N
	通话费用<100	—	N	N	—	N	N	N	Y
输出/动作	钻石卡					√			
	金卡		√				√		
	银卡			√				√	
	无卡	√			√				√
用例		欠费	YY300	YY200	YY100	N500	N200	N100	N90
结果		无卡	金卡	银卡	无卡	钻卡	金卡	银卡	无卡

2.3 边界值分析法

什么是边界值分析法

边界值分析法就是对输入或输出的边界值进行测试的一种黑盒测试方法。通常边界值分析法是作为对等价类划分法的补充，这种情况下，其测试用例来自等价类的边界。

为什么需要进行边界值分析?

- 软件的两个主要缺陷源：
  - 条件：变量取值需要满足的约束
  - 边界：变量的各种“极限取值”
- 边界值分析：能有效捕获出现在边界处缺陷的一种测试方法；利用并扩展了缺陷更容易出现在边界处的理念
- 缺陷容易出现在边界处的原因：
  1. 使用比较操作符时未仔细分析
  2. 多种循环和条件检查方法引起的困惑
  3. 对边界附近需求的理解不够

如何使用边界值设计测试用例?

使用边界值分析方法设计测试用例，首先应确定边界情况。通常输入和输出等价类的边界，就是应重点测试的边界情况。应当选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。
测试临近边界的有效数据，测试最后一个可能有效的数据，同时测试刚超过边界的无效数据

	边界值附近数据	测试用例设计思路
字符	Min-1, Min, Min+1 Max-1,Max,Max+1	若文本输入框允许输入1-255个字符。 则边界值测试用例：0,1,2,254,255,256
数值范围	Min-1, Min, Min+1 Max-1,Max,Max+1	输入数据允许1-999。 0，1，2，998，999，1000
集合、空间	0，Min, Min+1 Max-1,Max,Max+1	图片大小不能超过50k 0, 1k, 49k, 50k, 51k

测试用例（数据）确认方法

边界值附近数据的确认方法
n: 存在边界值的参数个数
m: 边界值条件数
Paul Jorgensen 公式（3 种方法的测试粒度依次增强）：

1. $4n+1$：基本边界测试。每个参数取 min，min+1， max-1，max 各一次，同时其他参数取典型值 nom。最后全部参数取典型值 nom 一次。
2. $6n+1\ (+x)$：健壮性边界测试。每个参数取 min-1，min，min+1， max-1，max，max+1 各一次，同时其他参数取典型值 nom。最后全部参数取典型值 nom 一次。
3. $3m$：边界条件测试。每个条件取 -1，自身，+1 各一次。

边界条件测试

确定边界条件：

1. 每次只考虑一个参数的边界，固定其它参数
2. 补充确定的关联边界值

边界值分析原理

次边界条件/隐性边界：产品说明书中没有，外部用户看不到

1. 2 的幂：各种数据类型的极限值；

2. ASCII 表、Unicode 字符总数；
3. 空白、空值、零值、无输入等条件

边界值分析总结

1. 边界值需要彻底测试
2. 考虑资源极限，如有限的缓存
3. 需求规格中对硬件资源的限制
4. 输出值的边界也需要考虑

边界值分析和等价类划分的关系

- 等价类划分时，往往先要确定边界值。
- 边界值分析是等价类划分方法的补充。
- 测试中需要将两者结合起来使用。

4 白盒测试及测试用例设计

1 白盒测试概念

白盒测试=结构化测试

- 白盒测试两个特点：（1）基于代码；（2）尽可能覆盖实现的行为。
- 黑盒测试两个特点：（1）基于规约；（2）尽可能覆盖定义的行为。

实施白盒测试的原因：

1. 确保每段代码都被执行，避免相应的缺陷；
2. 是黑盒测试/功能测试的补充
3. 能覆盖高层规范说明中的忽视的底层细节

白盒测试基本概念

- 依赖于对程序细节的严密检验，针对特定条件和/与循环集设计测试用例，对软件的逻辑路经进行测试。
- 在程序的不同点检验“程序的状态”以判定其实际情况是否和预期的状态相一致。
- 要求对某些程序的结构特性做到一定程度的覆盖，或者说是“基于覆盖的测试”。

定义

- 白盒测试：一种基于**源程序**或**代码**的测试方法。依据**源程序**或**代码结构与逻辑**生成测试用例，以尽可能多地发现并修改源程序错误。
- 分为**静态**和**动态**两种类型。
- 其它称谓：结构测试、逻辑驱动测试、基于程序的测试

意义

主要的单元测试方法，保证软件质量的基础

实施者

单元测试阶段：一般由开发人员进行
集成测试阶段：一般由测试人员和开发人员共同完成

步骤（动态）

1. 程序图 (CFG,PDG);
2. 生成测试用例;
3. 执行测试;
4. 分析覆盖标准;
5. 判定测试结果

进入和退出条件

- 进入条件： 编码开始阶段
- 退出条件:
    1. 完成测试计划（满足一定覆盖率）
    2. 发现并修正了错误
    3. 预算和开发时间

2 静态白盒测试

静态白盒测试

定义

在不执行软件的条件下有条理地仔细审查软件设计、体系结构和代码，从而找出软件缺陷的过程，有时称为结构化分析。

理由

1. 尽早发现软件缺陷;
2. 为后继测试中设计测试用例提供思路。
3. 测试人员通过旁听审查评论，可对软件有一定理解，更好确定软件缺陷的范围。

优点

1. 可发现某些机器发现不了的错误。
2. 利用不同人对代码的不同观点。
3. 对照设计，确保程序能完成预期功能。
4. 不但能检测出错误，还可以尝试确定错误根源。
5. 节约计算机资源，但以增加人工成本为代价。
6. 尽早发现缺陷，避免后期缺陷修复造成的巨大压力。

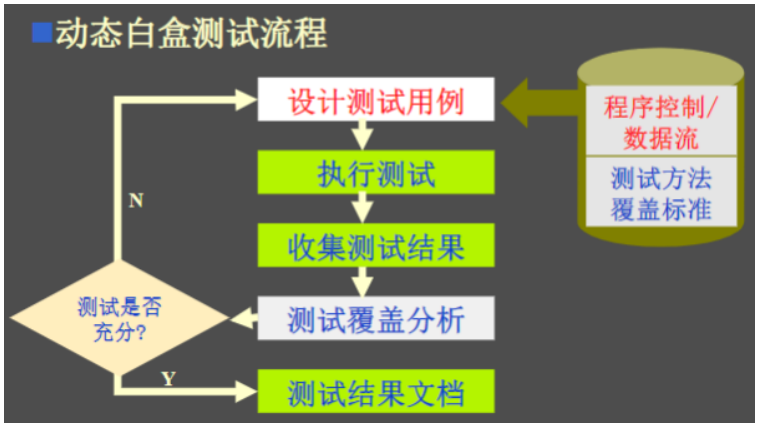
3 动态白盒测试

动态白盒测试简介

特点

1. 不但要提供软件源代码，还要提供可执行程序，测试过程需要在计算机上执行程序.
2. 对程序模块中的所有独立执行路径至少执行一次
3. 对所有逻辑判定的取值（“真”与“假”）都至少测试一次
4. 在上下边界及可操作范围内运行所有循环
5. 测试内部数据结构的有效性

测试流程



覆盖准则简介

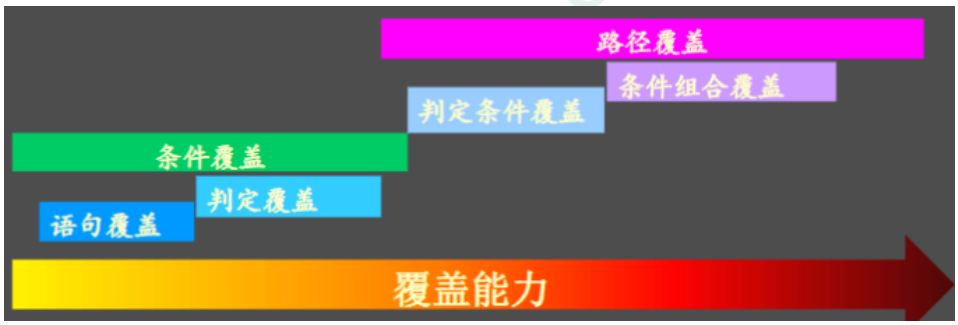
- 意义： 对“测试执行到何时才是充分的？”的定量回答。
- 作用： 测试软件的一种度量标准，描述程序源代码被测试的程度。
- 一种测试技术通常有一种对应的覆盖准则。

基于控制流的测试

白盒测试要求对被测程序结构特性做到一定程度的覆盖
常用覆盖准则

1. 语句覆盖：保证程序中的每条语句都执行一遍
2. 判定覆盖：保证每个判断取 True 和 False 至少一次
3. 条件覆盖：保证每个判断中的每个条件的取值至少满足一次
4. 判定条件覆盖：保证每个条件和由条件组成的判断的取值
5. 条件组合覆盖：保证每个条件的取值组合至少出现一次
6. 路径覆盖：覆盖程序中所有可能路径

无论哪种覆盖方法，都无法绝对保证程序的正确性



循环处理方法

循环分为 4 种不同类型：

1. 简单循环
2. 嵌套循环
3. 串接循环
4. 非结构循环

有各自的构造方法

基于数据流的测试

- 数据流测试（data-flow testing）是一种面向变量 定义 - 使用位置覆盖的基于程序结构的测试方法。

    - 该测试方法重点关注变量的定义与使用。在选定的一组代码中搜索某个变量所有的定义位置和使用位置，并检查在程序运行时该变量的值将会如何变化，从而分析是否是 Bug 的产生原因。
- 数据流测试与路径测试的区别在于：
    - 路径测试基本上是从控制流（逻辑覆盖）角度来分析的。
    - 而数据流测试则是数据流（定义 - 使用对）的角度来分析的，利用变量之间的关系，通过定义 - 使用路径设计一系列的覆盖标准用于衡量测试的覆盖率。
- 数据流是什么？
    - 数据流是变量的定义或使用顺序和变量的可能状态的一种抽象表示。
        - 变量的状态可以是：创建/定义 (Creation/ Defined)、使用 (Use)、清除/销毁 (Killed/Destruction)。
        - 变量存在从创建、使用到销毁的一个完整状态变化过程。
        - 例如：变量赋值错误检查就是发现代码缺陷或错误的一种有效方法。
    - 实际上该方法可认为是路径测试的“真实性”检查，是对基于路径测试的一种改良。
- 数据流覆盖
    - 程序是对数据的加工处理过程，对程序的测试可从数据处理流程的角度进行考虑。
    - 数据的处理流程对应数据流图 （DFG：data flow graph）
        - 数据流图类似于控制流图（CFG：control flow graph），描述了测试对象代码的处理过程。同时也详细描 述了代码中变量的创建、使用和销毁的状态。通过检查数据流图来验证测试对象代码中每个变量的状态组合是否正确。
    - 受 Debugging 过程的启发： 寻找关于某变量的定义和使用位置，思考程序在运行时该变量的值会如何变化，从而分析 Bug 产生的原因。

数据流测试的作用是用来测试变量定义点和使用点之间的路径。这些路径也 称为“定义 - 使用对” （define-use pairs 或 du-pairs） 或“定义 - 使用路径 （define-use paths）”。通过数据流分析而生成的测试集 （test suite）可用来获得针对每个变量的“定义 - 使用对”的 100% 覆盖。
要追踪整个程序代码中的每个变量的定义和使用时，并不需在测试时考虑被测对象的控制流。

数据流测试原理和基础

根据程序中变量定义和其后变量使用的位置来选择程序的测试路径
基本定义：

1. P——程序
2. G(P)——程序数据流图
3. V——变量集合
4. PATH(P)——P 的所有路径集合
5. 定义节点 DEF(v,n)——在节点 n 定义了变量 v，
    - 即变量赋值语句，例如：`input x; x = 2;`
6. 使用节点 USE(v,n)——在节点 n 使用了变量 v
    - 例如：`print x; a=2+x;`
7. 谓词使用 P-use——USE(v,n) 位于一个谓词中，即条件判断语句中
    - 例如：`if b>6`
8. 计算使用 C-use——USE(v,n) 位于一个计算中，即计算表达式中
    - 例如：`x=3+b`
9. 输出使用 O-use—— 变量值被输出到屏幕/打印机
10. 定位使用 L-use—— 变量值用于定位数组位置
    - 例如，`a[b]` 中的 `b`
11. 迭代使用 I-use—— 变量值用于控制循环次数
12. 变量 v 的定义 - 使用路径 (du-path)

- 给定 PATH(P) 中的某条路径，如果定义节点 DEF(v,m) 为该路径的起始节点，使用节点 USE(v,n) 为该路径的终止节点，则该路径是 v 的一条 du-path。

13. 变量 v 的定义 - 清洁路径 (define-clear-path/dc-path)
    - 如果变量 v 的某个定义 - 使用路径，除起始节点外没有其它定义节点，则该路径是变量 v 的定义 - 清洁路径。

数据流覆盖测试步骤

1. 对于给定的程序，构造相应的程序数据流图
2. 找出所有变量的 du 路径
3. 考察测试用例对这些路径的覆盖程度，即可作为衡量测试效果的度量值

小结

- 数据流测试是在程序代码经过的路径上检查变量用法是否正确的一种方法。
- 可以发现“定义 - 使用”异常的缺陷。这里异常是指可能会导致程序失效的情形。如发现的数据流异常有：没有初始化就读取了变量的值，或根本没有使用变量的值。
- 异常可能会触发程序运行的风险，但并非数据流异常都会导致错误的程序行为发生 (即失效)。因此对发现的数据流问题，需深入检查确定是否存在定义 - 使用问题。
- 优点
    1. 揭示隐藏在代码变量定义和使用中的各种错误
    2. 可以覆盖所有语句、所有分支、所有路径
    3. 对代码的测试比较彻底
- 缺点
    1. 无法检测代码中不可达路径
    2. 不验证需求规格

5 面向对象软件的测试

1 面向对象测试基础

面向对象技术产生更好的系统结构，更规范的编程风格，极大的优化了数据使用的安全性，提高了程序代码的重用。

面向对象概念

对象

- 是一个可操作的实体，既包括了特定的数据，又包含了操作这些数据的代码。如：某个银行帐户的数据和操纵这些数据的代码；
- 对象是软件开发期间测试的直接目标：对象的行为是否符合它的说明规定？该对象与它相关的对象能否协同工作？
- 对象的生命周期：一个对象被创建时生命周期开始，这个过程贯穿于对象的一系列状态，当一个对象被销毁/删除时生命周期就结束。

消息

- 消息是对象的操作将要执行的一种请求，包含：名字、实参、类型等
- 面向对象的程序是通过一系列对象的协同工作来实现功能/服务的，这一协作是通过对象之间的互相传递消息来完成的。

接口

- 接口是对象行为声明的集合。行为被集中在一起，并通过单个的概念定义一些相关动作；
- 接口由一些规范组成。规范定义了类的一套完整的公共行为。

类

- 是一些具有共性的对象集合；
- 面向对象程序运行的基本元素是对象，类则是用来定义对象这一基本元素的。
- 创建对象的过程称为“实例化”，创建的结果称为“实例”
- 类声明：定义了类的每个对象能做什么
- 类实现：定义类的每个对象如何做他们能做的事情。

封装

- 定义类结构
- 接口由公用方法定义
- 行为由在其实例数据上操作的方法定义
- 有助于强制信息隐藏

继承

- 继承是类的一种联系，允许新类在一个已有类的基础上进行定义。一个类对另一个类的依赖，使得已有类的说明和实现可以被复用。
- 优势：已有类不会被改变。

继承修改

- 只继承父类属性不增加新的属性
- 增加新的属性
- 重新定义父类的属性

多态

- 一个属性可能不止一组值
- 一个操作可能有不止一个方法实现
- 重载 (overloaded，变量的类型或数目)
- 动态绑定 (Dynamic binding)
- 多态提供了将对象看作是一种或多种类型的能力，类型机制可以支持许多不同的类型适应策略。
- 类型的完全匹配非常安全，多态支持灵活的设计，又易于维护。

传统 vs 面向对象方法语境下的测试

传统开发方法与测试

- 需求规约 → 系统测试
  - 验证软件满足需求
- 设计规约 → 集成测试
  - 基于结构设计

    - 自顶向下或自底向上
- 编码 → 单元测试
    - 封装功能

面向对象开发与测试

- 责任驱动（职责分配 Responsibility Assignment）
- 对象协作（collaboration）
- 迭代、增量方法
- 传统的测试层次在面向对象语境下发生变化
- 系统测试
    - 仍然基于需求规约
- 单元测试
    - 两个常用的基本元素：method、class
- 集成测试
    - 主程序最小化
    - 集成测试是 OO 测试最复杂的部分
    - 基于合成（composition）
    - 使用类簇（class cluster），含继承
    - 对象关系图 (ORD: object relation diagram) – 类间依赖和方法依赖关系的有向图

2 面向对象软件的特点对软件测试的影响

2.1 类的使用对测试的影响

- 基本可测单元不再是子程序，而是类或对象；
- 在对每个类进行了单元测试后还要对类簇进行测试，这样逐步完成整个系统的测试。

2.2 封装对测试的影响

- 信息隐藏使对象的部分不可访问，减少了波动影响
- 修改封装会导致大量的回归测试：CIA
- 测试顺序相当重要（可以减少工作量）：集成测试序
- 要测试一个对象的方法，测试人员在激活一个方法前后必须检查这个对象的状态（Pre-Condition and Post-Condition，前后置条件），就需要访问对象的内部状态，而这些信息正是信息隐藏所不让测试人员知道的。

2.3 继承对测试的影响

- 继承允许子类（衍类）重用父类（基类）的实现，要确定衍类中从基类继承的已测试的功能是否需要再测试，衍类的测试受对基类重新测试的量的影响，
- 有时依赖于具体的面向对象语言。

2.4 多态和动态绑定对测试的影响

- 引入了不可判定性问题，很难甚至不可能静态地确定在一给定的测试用例中哪个方法被激活（在做一个对多态方法的调用或一个方法有多态参数时就会出现这个问题）。
- 对象中没有状态报告机制，状态的控制分布在整个系统中，使得难以对每个状态进行单一的测试。

- 一个多态组件的每一个可能的绑定都需要一个独立的测试，而实际上又难以找到所有的这样的绑定，特别是非预期的绑定，这也使得集成计划复杂化。

2.5 抽象对测试的影响

- 本意是降低复杂性，把对象的本质行为与实现分开。
- 使测试变得困难，特别是要执行内部的结构测试时，降低了软件的可测试性。

3 测试与面向对象软件开发过程集成

3.1 面向对象的软件开发过程模型

- 面向对象的开发模型突破了传统的瀑布模型，采用迭代式增量开发过程模型，每次迭代又包含面向对象分析（OOA）、面向对象设计（OOD）、和面向对象编程（OOP）三个阶段。
- OOA 阶段产生整个问题空间的抽象描述，OOD 阶段进一步设计成适用于面向对象编程语言的类和类结构，OOP 阶段形成代码。
- 由于面向对象的特点，采用这种开发模型能有效的将分析设计的文本或图表代码化，不断适应用户需求的变动。

Spiral Life Cycle；RUP: Phases and Iterations

3.2 测试过程与面向对象软件开发过程集成

- 针对面向对象的软件开发过程模型，结合传统的测试步骤的划分，在整个软件开发生命周期全过程中不断测试，使开发阶段的测试与编码完成后的单元测试、集成测试、系统测试成为一个整体。
- 分析一点、测试一点，设计一点、测试一点，编码一点、测试一点。这样一来，不仅能在软件开发早期发现错误，也避免由已有的错误产生新的错误，在开发过程中每一步都迭代式增量方法，软件就在不断精化中开发。

3.3 面向对象测试模型 (OOTM)

- OOA Test 和 OOD Test 是对分析结果和设计结果的测试，主要是对分析、设计产生的模型图和文档资料进行复审，验证每个模型元素的正确性，由建模专家审查语法正确性，领域专家审查语义正确性，软件专家审查每个类的结构、方法的算法、行为与需求的一致性，是软件开发前期的关键测试。
- OOP Test 主要针对编程风格和程序代码实现进行测试，其主要的测试内容在面向对象单元测试和面向对象集成测试中体现。
- OO Unit Test 是对程序内部具体单一的功能模块的测试，如果程序是用 C++ 语言实现，主要就是对类成员函数的测试。面向对象单元测试是进行面向对象集成测试的基础。
- OO Integrate Test 主要对系统内部的相互服务进行测试，如成员函数间的相互作用、类间的消息传递等。面向对象集成测试不但要基于面向对象单元测试，更要参见 OOD 或 OOD Test 结果。
- OO System Test 是基于面向对象集成测试的最后阶段的测试，主要以用户需求为测试标准，需要借鉴 OOA 或 OOA Test 结果。

4 面向对象单元测试

- OO 软件功能是由类通过消息传递来完成的，因此 OO 单元测试实际就是对类的测试。
- 类的测试策略包括：基于服务的测试、基于状态的测试、基于响应状态的测试。

4.1 基于服务的类测试策略

- 主要考察封装在类中的一个方法对数据进行的操作是否存在问题，可以采用传统的白盒测试方法。

- 为克服软件测试的盲目性和局限性，保证测试的质量，提高软件的可靠性，Kung 提出了块分支图法（BBD），从基于基本路径测试的结构测试出发对类中服务进行测试。

4.2 基于状态的类测试策略

- 考察类的实例在生命周期各个状态下的情况，采用从外界向对象发送特定消息序列的方法来测试对象（类的实例）的响应状态。
- 【前后置条件】执行前对象状态的变化，可能会使同样的一个成员方法执行完全不同的功能，另外用户对对象方法的调用又具有不确定性，所以这部分的测试变得非常复杂，超出了传统测试所覆盖的范围。因此构造 OSD（object state diagram）模型来进行类的状态测试。

5 面向对象集成测试和系统测试

5.1 集成测试

由于面向对象软件的特殊性，传统的集成测试策略无法应用到面向对象软件的测试，面向对象软件的集成测试需要在整个程序编译完成后进行，面向对象程序具有动态绑定（dynamic binding）特性，程序的控制流无法确定，只能对编译完成的程序做基于黑盒子的集成测试。

1．面向对象软件的集成测试策略

1. 基于线程的测试
   - 集成对响应系统的一个输入或事件所需的一组类，每个线程分别进行集成和测试，应用回归测试以保证没有产生副作用。
2. 基于使用的测试
   - 按分层来组装系统，可以先进行独立类的测试，然后用测试过的独立类对从属类进行测试，直到整个系统构造完成。

2．面向对象软件的集成测试过程

1. 静态测试
   - 针对程序结构进行，检测程序结构是否符合设计要求。通过使用测试软件的逆向工程功能，得出源程序的类系统图和函数功能调用关系图，与 OOD 结果相比较，检测程序结构和实现上是否有缺陷，检测 OOP 是否达到了设计要求。
2. 动态测试
   - 根据静态测试得出的函数调用关系图或类关系图作为参考，设计测试用例，使其达到一定的测试覆盖标准。

3．设计集成测试用例的步骤

1. 选定检测的类，参考 OOD 分析结果，确定出类的状态和相应的行为；
2. 确定覆盖标准，利用结构关系图确定待测类的所有关联；
3. 根据程序中类的对象构造测试用例，确认使用什么输入激发类的状态，使用类的服务和期望产生什么行为等，还要设计一些类禁止的例子，确认类是否有不合法的行为产生。

5.2 系统测试

测试软件与系统其他部分配套运行的表现，以保证在系统各部分协调工作的环境下也能进行工作。不仅是确认系统在实际运行时是否满足用户的需要，也是对软件开发设计的再确认。应该参考 OOA 的结果，对应描述的对象、属性和各种服务。

1. 功能测试

    - 以软件分析文档为标准，测试系统的功能是否达到要求，是否满足用户的需求。
2. 强度测试
    - 测试系统的负载情况和功能实现情况，比如信息系统能容纳多少人同时在线操作。
3. 性能测试
    - 与强度测试相结合，测试软件系统的运行性能。
4. 安全测试
    - 验证安装在系统内的保护机构确实能够对系统进行保护。
5. 恢复测试
    - 采用人工的干扰使软件出错，中断使用，检测系统的恢复能力。
6. 可用性测试
    - 测试用户是否能够满意使用，主要是指操作是否简便，操作界面是否符合使用习惯。
7. 安装/卸载测试
    - 测试用户是否能方便地安装和卸载。

总的来说，面向对象的集成测试和系统测试都是基于面向对象的分析和设计进行的，在分析阶段总结出的用例图、状态图、顺序图、协作图和活动图都可以作为集成测试和系统测试的依据。

6 软件性能测试

1. 软件性能

- 软件性能（Software Performance）是软件的一种非功能特性，它关注的不是软件是否能够完成特定的功能，而是在完成该功能时展示出来的及时性。【属于用户的非功能需求】
- 不同的人对同样软件有不同主观感受，对软件性能关心视角也不同。

性能指标

- 响应时间：系统对请求作出响应的时间。
- 系统响应时间和应用延迟时间：
    - 系统响应时间：计算机对用户的输入或请求作出反应的时间。
    - 应用延迟时间（latency）：应用接到指令后的处理时间。
- 吞吐量：系统在单位时间内处理请求的数量。
- 并发用户数：系统可以同时承载的正常使用系统功能的用户的数量。
- 资源利用率：反映一段时间内资源平均被占用情况。
- 广义上系统的性能包括执行效率、资源占用、稳定性、安全性、兼容性、可扩展性、可靠性等。

性能视角

- 用户视角：响应时间，不关心来源，有主客观因素（如心理）。
- 管理员视角：首先关注普通用户感受到的软件性能，其次关注如何利用管理功能进行性能调优。
- 开发人员视角：与管理员基本一致，但需要更深入地关注软件性能。在开发过程中，开发人员希望能够尽可能地开发出高性能的软件。

2. 性能测试概述

- 通过自动化的测试工具模拟多种正常、峰值以及异常负载条件来对系统的各项性能指标进行测试。
    - 负载压力测试工具
    - 功能回归测试工具
    - 测试管理工具

- 用来保证产品发布后系统的性能满足用户需求，保障软件质量。

性能测试的目的

验证软件系统是否能够达到用户提出的性能指标，同时发现软件系统中存在的性能瓶颈，优化软件和系统。

- **评估系统的能力**：测试中得到的负荷和响应时间数据可以被用于验证系统模型的能力，并帮助作出决策。
- **识别系统中的弱点**：受控的负荷可以被增加到一个极端的水平，并突破它，从而修复系统的瓶颈或薄弱的地方。
- **系统调优**：重复运行测试，验证调整系统的活动得到了预期的结果，从而改进性能。
- **检测系统的问题**：长时间的测试执行可导致程序发生由于内存泄露引起的失败，揭示程序中的隐含的问题或冲突。
- **验证系统稳定性（stability）和可靠性（reliability）**：在一个生产负荷下执行测试一定的时间是评估系统稳定性和可靠性是否满足要求的唯一方法。

3. 性能测试的步骤

3.1 熟悉应用

- 了解应用的架构、应用的功能逻辑。

3.2 测试需求

- 看按照目前的硬件性能和数量能不能支撑一定量的 UV，如果不能就找瓶颈，否则再加压。

3.3 测试准备

- 测试客户端机器准备、测试数据准备、测试脚本准备。
- 客户端机器：要足够，要和服务器保持网络通畅，保证瓶颈不在客户端或网络。包括：网络带宽要高于服务器吞吐量、网络带宽要稳定。

测试数据

- 如果被测功能涉及数据库和高速缓存，通常需要预设很大的数据量才能凸显性能瓶颈。
- 如果已经上线，数据可以从线上拷贝得到；否则需要构造类似线上的数据量。
- 测试数据准备的脚本，有时候比测试脚本本身还要多。
- 对于实在没有办法构造大数据量的情况，如果要测试高速缓存，有时会按数据量的比例减少高速缓存，以使测试结果尽量准确。

测试用例设计

- 在了解软件业务流程的基础上。
- 受最小的影响提供最多的测试信息，一次尽可能的包含多个测试要素。这些测试用例必须是测试工具可以实现的，不同的测试场景将测试不同的功能。
- 尽可能把性能测试用例设计得复杂，才有可能发现软件的性能瓶颈。

3.4 测试执行

- 需要监控测试客户端和服务器性能，监控服务器端应用情况：
  - 客户端的系统资源（CPU、IO、memory）情况
  - 服务端的系统资源（CPU、IO、memory）情况
  - 服务器的 JVM 运行情况

- 服务端的应用情况，看是否有异常
- 响应时间、吞吐量等指标

执行测试用例

- 通过性能测试工具运行测试用例
- 在不同的测试环境上运行
- 通过性能测试工具运行测试用例。同一环境下作的性能测试得到的测试结果是不准确的，所以在运行这些测试用例的时候，需要用不同的测试环境，不同的机器配置上运行。

3.5 测试结果分析

- 在测试执行过程中，用各种监控工具看系统运行的状态，及时发现问题。
- 常见的问题有：内存问题、有限资源竞争问题。

4. 性能测试的指标

- **响应时间**
  - 对请求作出响应所需要的时间。
- **吞吐量**
  - 单位时间内系统处理用户的请求数。
  - 对于交互式应用，反映服务器承受的压力，能够说明系统的负载能力。
- **并发用户数**
  - 系统用户数：系统额定的用户数量，如一个 OA 系统，可能使用该系统的用户总数是 5000 个。
  - 同时在线用户数：在一定的时间范围内，最大的同时在线用户数量。
- **资源利用率**
  - 系统各种资源的使用情况，如 cpu 占用率为 68%，内存占用率为 55%。
  - 内存（Memory）
  - 磁盘（Physical Disk）
  - 处理器（Processor）
  - 网络

5. 软件性能测试方法

- **基准测试**
  - 通过和基础标准对比发现系统的不同点与变化。
  - 应用场景：在制定的标准下通过基准测试建立性能基准，当系统的环境、参数发生变化后，再进行一次相同标准下的测试，可看出变化对性能的影响。
  - 可以在较早的阶段发现性能问题。
- **负载测试**
  - 在超负荷环境中运行，看程序是否能够承担。
  - 在被测系统上不断增加压力，直到性能极致。例如：响应时间已经超过预定指标或者某种资源使用已将达到饱和状态。
  - 测试在各种工作负载下系统的性能，即当负载逐渐增加时，系统各项性能指标的变化情况。
  - 找系统的负载极限，为系统调优提供数据。
- **压力测试**
  - 检测一个系统的瓶颈或者不能接受的性能点，测试系统能提供的最大级别服务（最大并发用户数、最大用户访问量、最大吞吐量）。

    - 找出高负载下系统的问题，例如资源竞争、同步问题、内存泄露等。
- **容量测试**
    - 确定测试对象在给定时间内能持续处理的最大负载或工作量，如系统可处理同时在线的最大用户数。
    - 通过测试预先分析出反映软件系统应用特征的某项指标的极限值（如最大并发用户数、数据库记录数等），系统在其极限状态下没有出现任何软件故障或还能保持主要功能正常运行。确定测试对象在给定时间内能够持续处理的最大负载或工作量。
    - 能让软件开发商或用户了解该软件系统的承载能力或提供服务的能力，如某个电子商务网站所能承受的、同时进行交易或结算的在线用户数。
    - 例如，在数据库中加入大量的数据，测试系统能否在规定的时间内处理这些数据。
- **配置测试**
    - 是通过被测系统软/硬件环境的调整，了解各种不同环境对系统性能影响的程度，从而找到各项资源的最优分配原则。
- **并发测试**
    - 模拟用户并发访问，测试多用户并发访问同一应用、同一模块或数据记录时是否存在死锁或者其他性能问题。 并发用户数和并发数不同。
- **可靠性测试**
    - 给系统加载一定的业务压力，让应用系统持续运行一段时间，测试系统在这种条件下是否能够稳定运行。
- **失效恢复测试**
    - 针对有备份和负载均衡的系统，可用来检验如果系统局部发生故障， 用户能否继续使用系统，以及如果这种情况发生，用户将受到多大程度的影响。
- **大数据量测试**：
    - 独立的数据量测试： 针对某些系统存储、传输、统计、查询等业务进行大数据量测试。
    - 综合数据量测试：压力测试、负载测试、并发测试、可靠性测试相结合。