软件测试与维护

Introduction to Software Testing

[Reading assignment: Chapter 1, pp. 9-22]

Bug

- Defect
- Fault
- Problem
- Error
- Incident
- Anomaly异常
- Variance偏差

- Failure
- Inconsistency矛盾
- Product Anomaly
- Product Incidence
- Feature :-)

A software bug occurs when at least one of these rules is true

- The software does not do something that the specification says it should do.
- The software does something that the specification says it should not do.
- The software does something that the specification does not mention.
- The software does not do something that the product specification does not mention but should.
- The software is difficult to understand, hard to use, slow ...

Most bugs are not because of mistakes in the code ...

- Specification (~= 55%)
- Design (~= 25%)
- Code (~= 15%)
- Other (~= 5%)

Goal of a software tester

- ... to *find* bugs
- ... as *early* in the software development processes as possible
- ... and make sure they get *fixed*.

- **Advice**: Be careful not to get caught in the dangerous spiral of unattainable perfection.千万不要在无法达到的完美上兜圈子

You now know ...

- ... what is a bug
- ... the relationship between specification and bugs
- ... the cost of a bug relative to when it is found
- ..... the goal of the software tester
- ... valuable attributes of a software tester

The Software Development Process

[Reading assignment: Chapter 2, pp. 23-36]

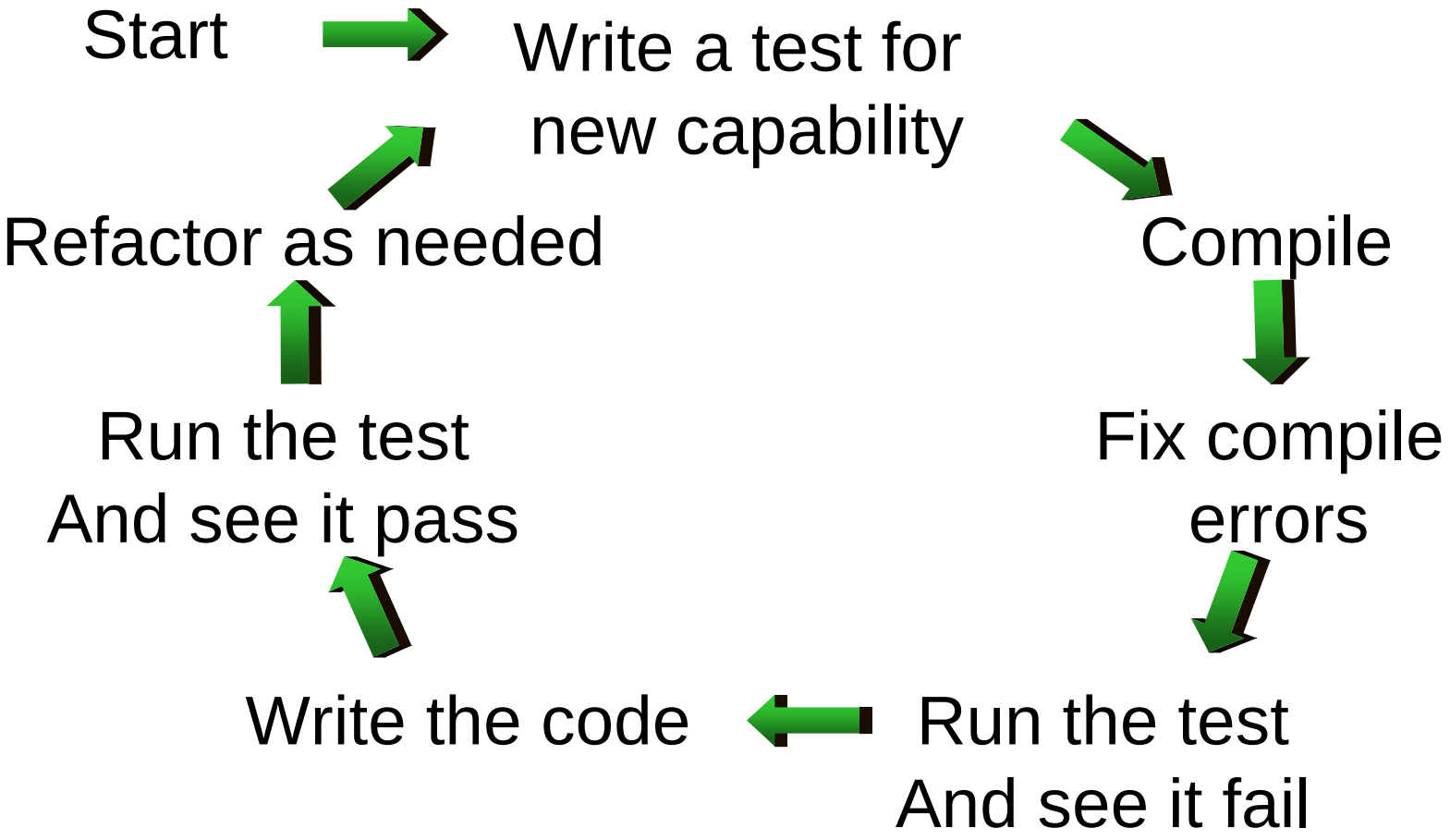
Software is ...

- requirements specification documents
- design documents
- source code
- test suites and test plans
- interfaces to hardware and software operating environment
- internal and external documentation
- executable programs and their persistent data

Discussion ...

- What is software engineering?
- Software engineering is an engineering discipline which is concerned with all aspects of software production，既包括技术，也包括管理
- What is CASE (Computer-Aided Software Engineering)
  - Software systems which are intended to provide automated support for software process activities. CASE systems are often used for method support
- Where does testing occur in the software development process?

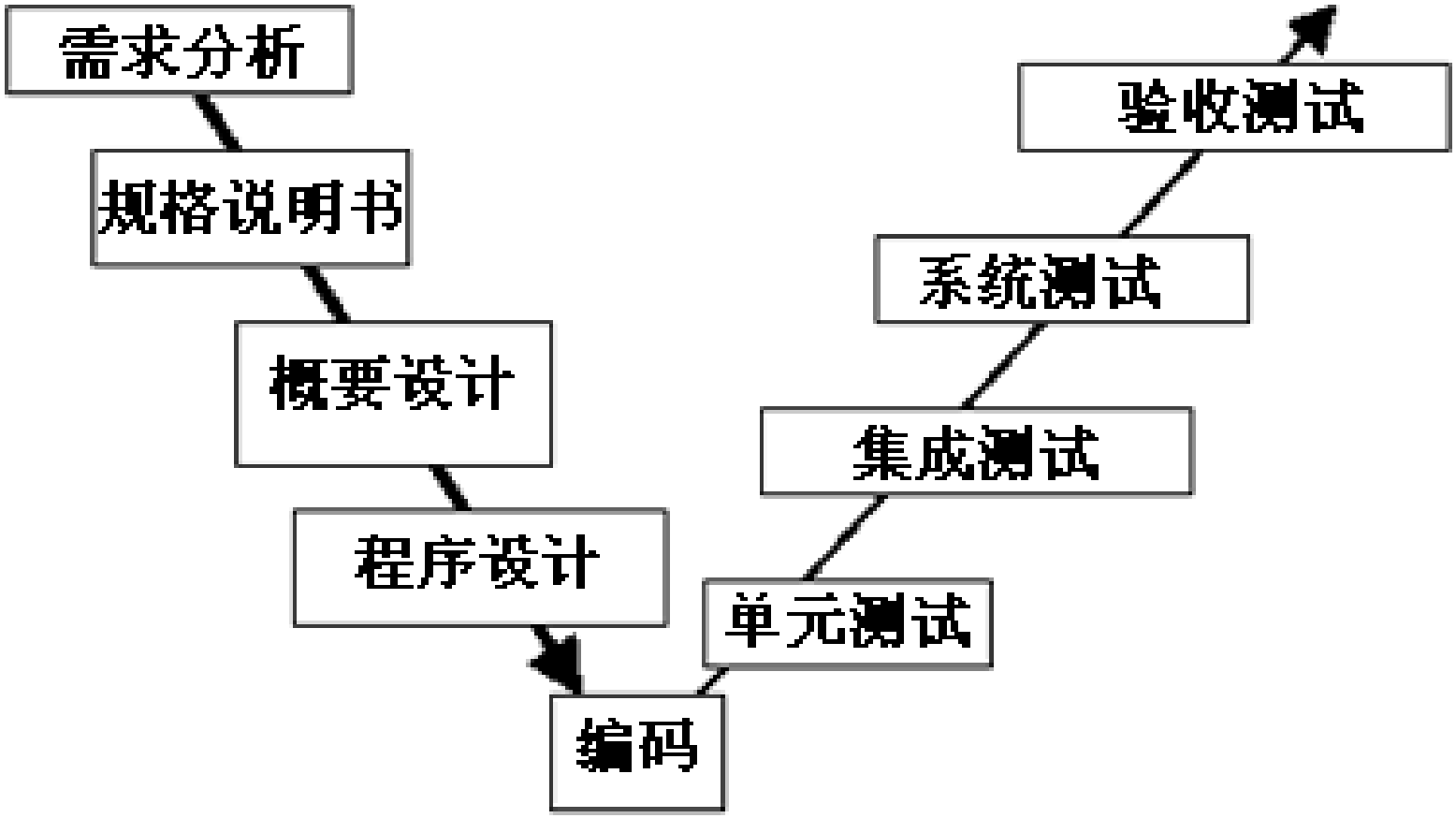
TDD – Test-Driven Development测试驱动开发



1.2.3与测试相关的过程模型

- V模型
- W模型
- H模型
- 其他模型
- 模型的意义

V模型

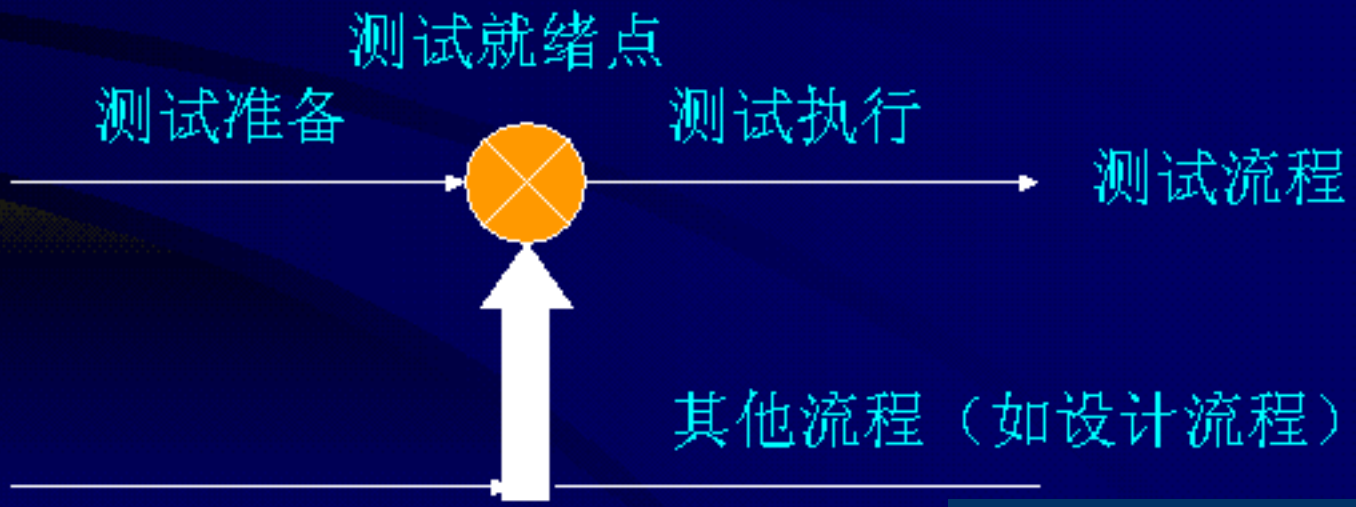


W模型


需求分析
需求测试
交付
验收测试
概要设计
概要设计测试
系统确认
系统测试
详细设计
详细设计测试
集成
集成测试
编码
单元测试

H模型

- 测试流程：
  - 测试准备活动：测试计划、测试设计、测试开发。
  - 测试执行活动：测试运行、测试评估。



The Realities of Software Testing

[Reading assignment: Chapter 3, pp. 37-50]

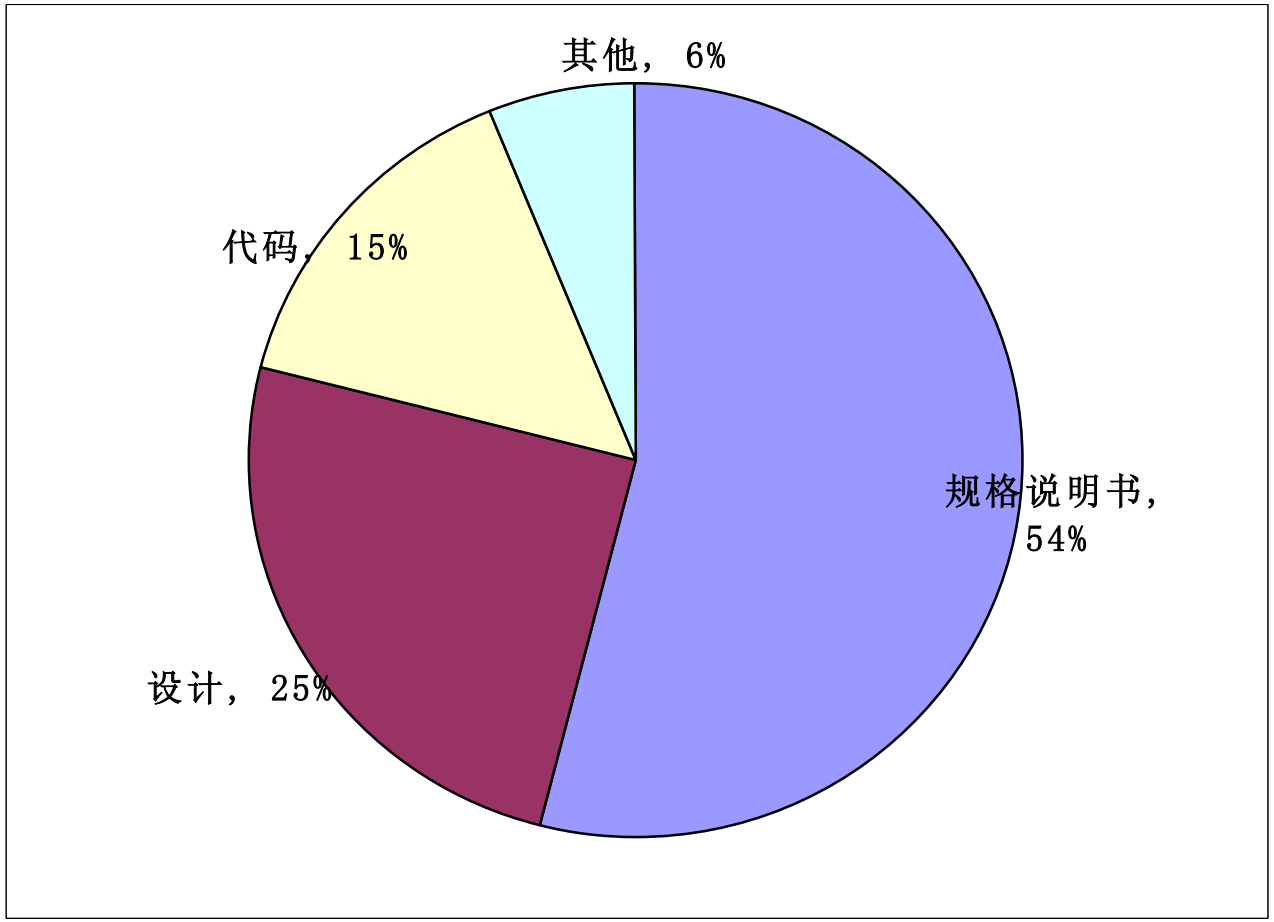
Software testing axioms

1. It is impossible to test a program completely.
2. Software testing is a risk-based exercise.
3. Testing cannot show the absence of bugs.
4. The more bugs you find, the more bugs there are.
5. Not all bugs found will be fixed.
6. It is difficult to say when a bug is indeed a bug.
7. Specifications are never final.
8. Software testers are not the most popular members of a project.
9. Software testing is a disciplined and technical profession.

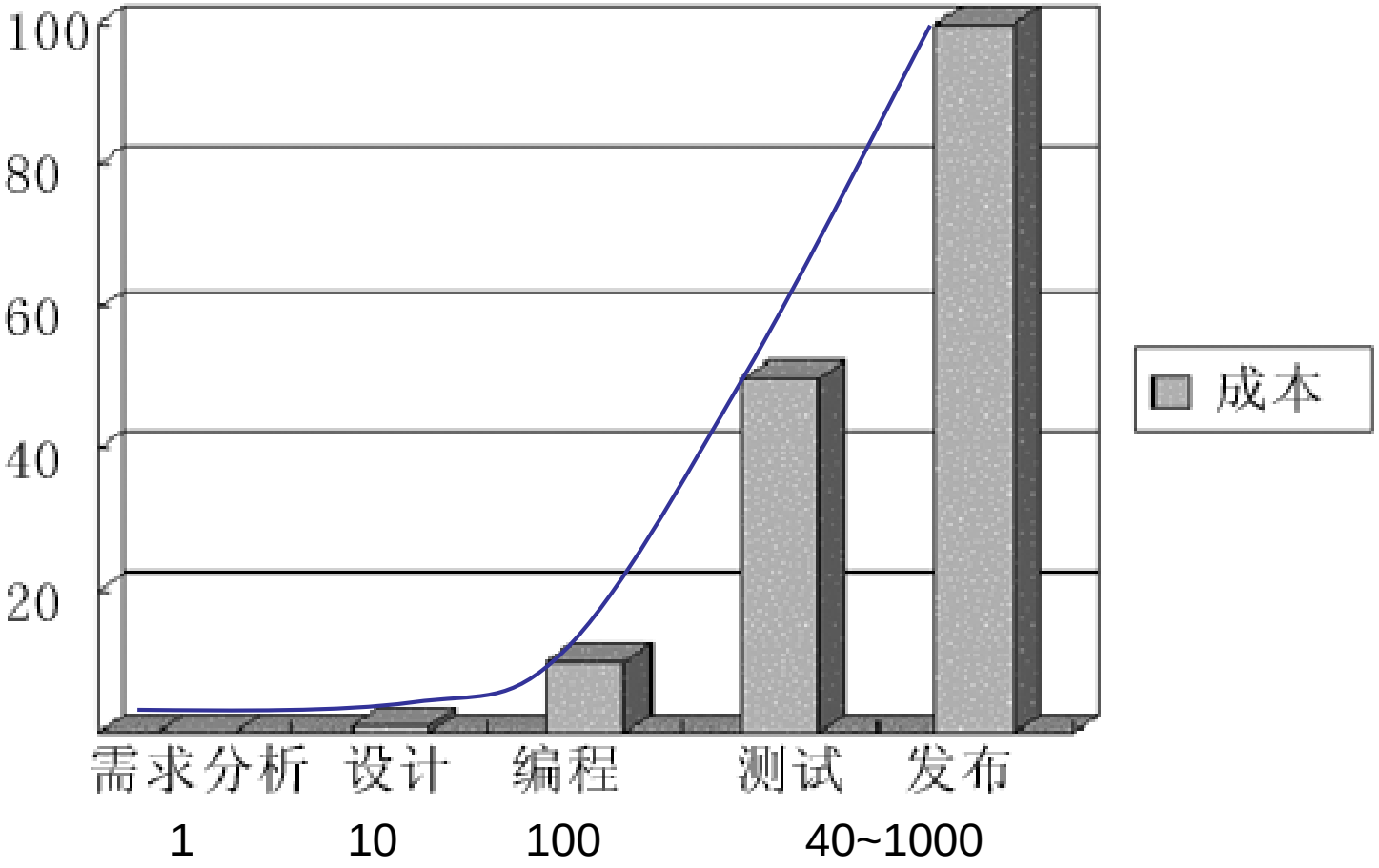
You now know …

- … the 9 axioms of software testing
- 软件测试的原则
- 软件测试的误区

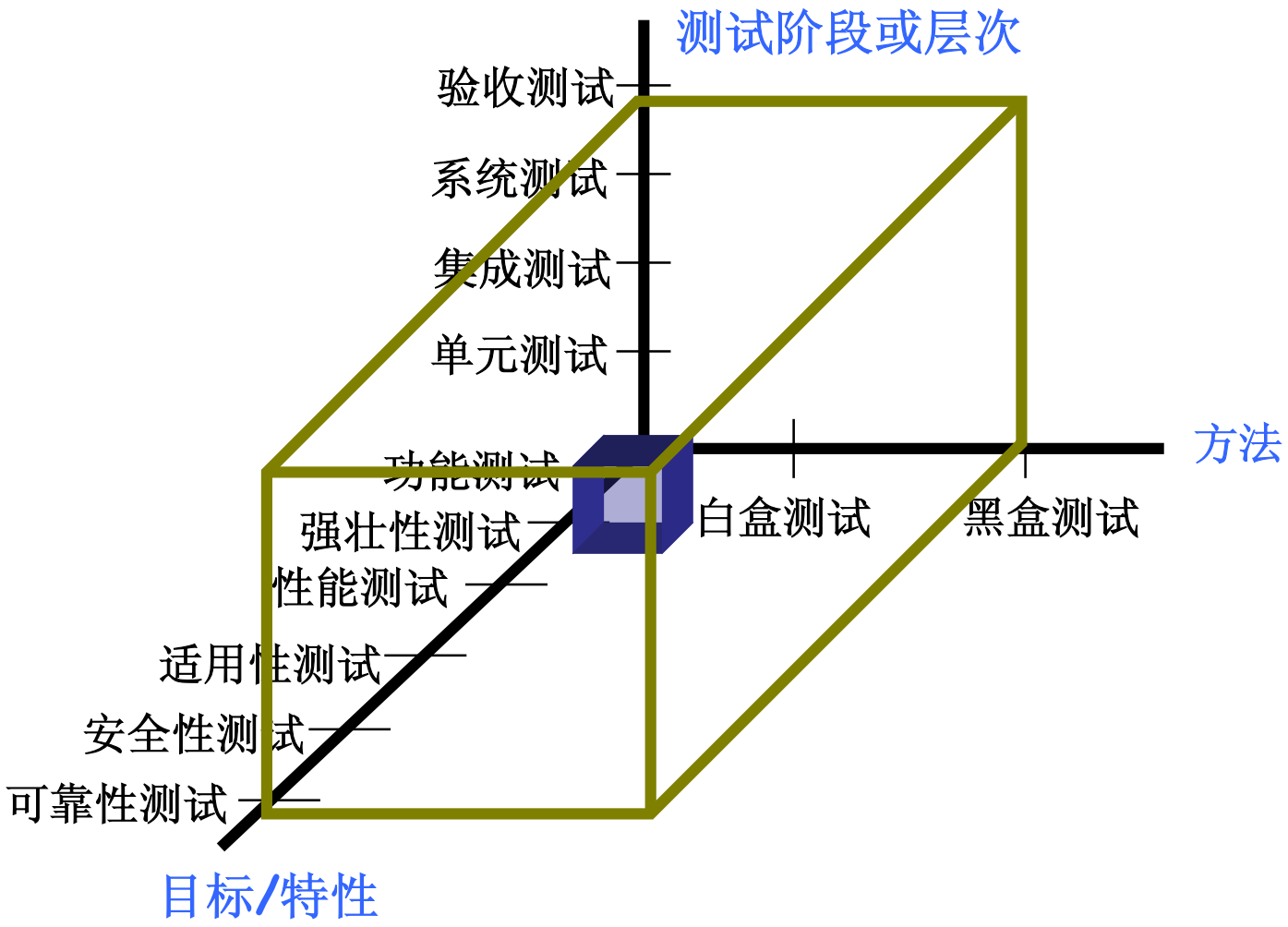
2. 3软件缺陷构成



2. 3缺陷成本

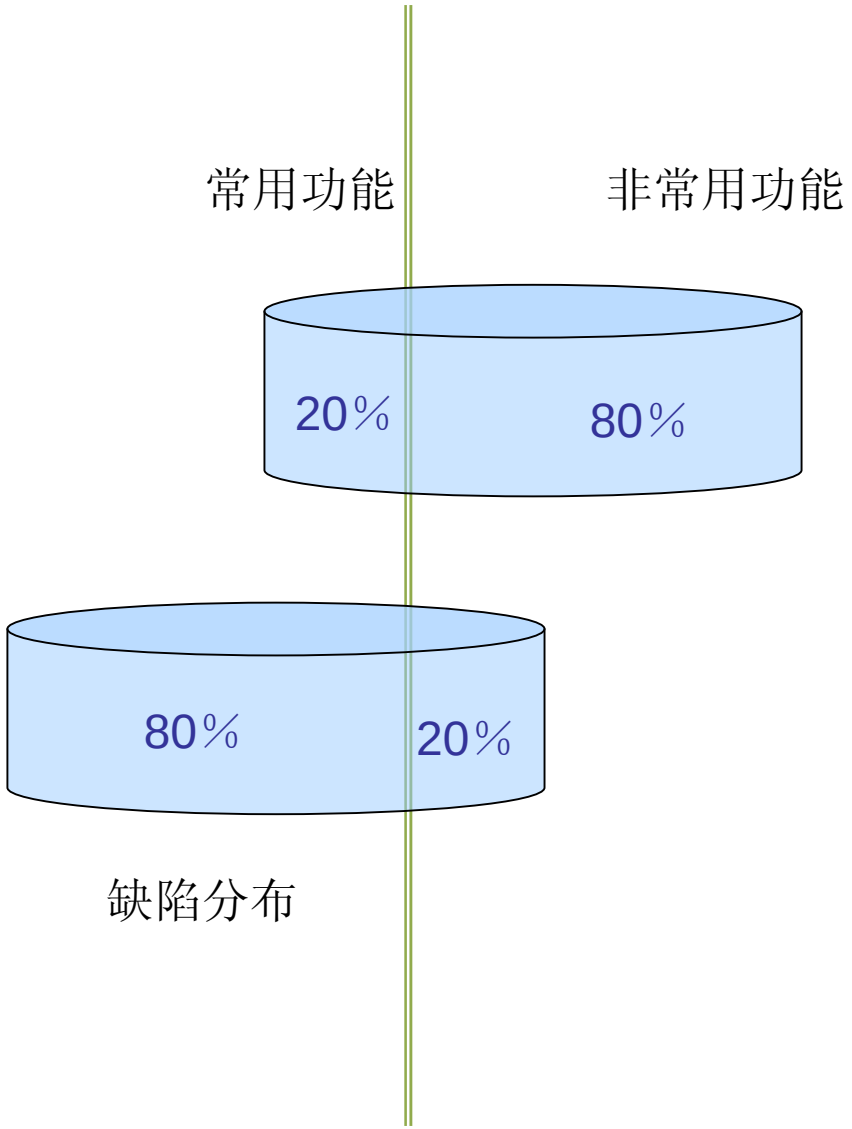


2.4软件测试分类


测试阶段或层次
验收测试
系统测试
集成测试
单元测试
功能测试
强壮性测试
性能测试
适用性测试
安全性测试
可靠性测试
白盒测试
黑盒测试
方法
目标/特性

2.5 ALAC（act-like-a-customer）

二八原则



2.5 确认、系统与验收测试

测试类型	对象	目的	依据	方法
单元测试	模块内部程序错误	消除局部模块的逻辑和功能上的错误和缺陷	详细设计	白盒为主黑盒为辅
集成测试	模块间的集成和调用关系	找出与软件设计相关的程序结构，模块调用关系，模块间接口方面的问题	概要设计	白盒与黑盒结合
系统与验收测试	整个系统中的软硬件	对整个系统进行一系列的整体、有效性测试	需求规格说明书	黑盒

测试与调试

- 测试发展的初期，测试就是调试，而现在测试是一个系统化工程化的概念，调试的范畴更小一些
- 调试不属于测试，是编码阶段的工作，由程序员完成；调试与测试的对象及采用的方法有很大程度上的相似，调试还用到断点控制等排错方法，但其目的却完全不同
- 而测试由测试员或程序员完成
- 测试是为了找出软件中存在的缺陷；　而调试是为了解决存在的缺陷
- 成功的测试发现了错误的症状，从而引起调试的进行

The Realities of Software Testing

[Reading assignment: Chapter 3, pp. 37-50]

Some Terminology

- Verification
  - “Are we building the product right?”
- Validation
  - “Are we building the right product?”

Verification vs validation

- Verification:
  "Are we building the product right"
- The software should conform to its specification(各阶段的规范，过程)
- Validation:
  "Are we building the right product"
- The software should do what the user really requires

什么是 SQA

Software Quality Assurance ?

软件质量保证是通过对软件产品和活动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活动.

SQA与软件测试的关系

- **SQA** 是管理工作、审查对象是流程、强调以预防为主
- **测试**是技术实施工作、测试对象是产品、主要是以事后检查（文档、程序）为主

- SQA指导测试、监控测试
- 测试为SQA提供依据
- 测试是SQA的一个环节、一个手段

ISO9000与CMM的区别

- ISO9001是通用的国际标准,适用于各类组织。
- CMM是美国军方为评价软件供应商的质量水平,委托SEI开发的一个评价模型,只用于软件业。
- CMM更详细,更专业。
- ISO9001只建立了一个可接受水平，而CMM是一个具有五个水平的评估工具。
- ISO9001聚焦于供应商和用户间的关系，而CMM更关注软件的开发过程。

CMMI

- CMM的成功促使其他学科也相继开发类似的过程改进模型，例如系统工程、需求工程、人力资源、集成产品开发、软件采购等等，从CMM衍生出了一些改善模型，比如：SW－CMM,SE-CMM,IPD-CMM等。不过，在同一个组织中多个过程改进模型的存在可能会引起冲突和混淆。
- CMMI就是为了解决这些模型之间的协调。 由业界、美国政府和卡内基·梅隆大学软件工程研究所率先倡导的能力成熟度模型集成（CMMI, Capability Maturity Model Integration ）项目致力於帮助企业缓解这种困境。CMMI为改进一个组织的各种过程提供了一个单一的集成化框架，新的集成模型框架消除了各个模型的不一致性，减少了模型间的重复，增加透明度和理解，建立了一个自动的、可扩展的框架。因而能够从总体上改进组织的质量和效率。CMMI主要关注点就是成本效益、明确重点、过程集中和灵活性四个方面。

You now know …

- … the 9 axioms of software testing
- … what is software verification
- … what is software validation
- … the relationship between testing and quality assurance
- 软件测试的分类，基本的定义
- 标准

软件测试与维护

组建测试队伍

12.1.1 软件测试团队的任务与责任

① 基本任务：测试计划、测试用例设计、执行测试、评估测试结果、递交测试报告

② 发现软件程序、文档、系统或产品中所有的问题；

③ 尽早地发现问题；

④ 督促相关人员尽快地解决产品中的缺陷；

⑤ 帮助项目管理人员制定合理的开发计划；

⑥ 并对问题进行分析、分类总结和跟踪

⑦ 帮助改善开发流程、提高产品开发效率；

⑧ 提高程序、文档编写的规范性、易读性、可维护性等。

12.2.1 测试团队的基本构成

- **QA/测试经理：** 人员管理，资源调配、测试方法改进等；
- **实验室管理人员：** 设置、配置和维护实验室的测试环境
- **内审员：** 审查流程，建立测试模板，跟踪缺陷测试报告的质量等；
- **测试组长：** 负责项目的管理、测试计划、任务安排等；
- **测试设计人员/资深测试工程师，** 产品规格说明书、设计的审查、测试用例的设计、技术难题的解决、培训和指导、实际测试任务的执行；
- **一般（初级）测试工程师，** 执行测试用例和相关的测试任务。
- **发布工程师：** 负责测试后产品的上载、打包、发布

12.5优秀软件测试工程师的必备素质

1、沟通能力

2. 技术能力

3、信心

4. 外交能力和幽默感

5. 耐心

6、很强的记忆力

7. 怀疑精神

8. 洞察力

9. 适度的好奇心

10. 反向思维和发散思维能力

Test Planning

CSSE 376, Software Quality Assurance

Rose-Hulman Institute of Technology

March 9, 2007

测试策略的概念

测试策略通常是描述测试工程的总体方法和目标。描述目前在进行哪一阶段的测试（如单元测试、集成测试、系统测试）以及每个阶段内进行的测试种类（如功能测试、性能测试、压力测试等）和方法，以确定合理的测试方案使得测试更有效。

Components of a Test Plan

1. Test plan identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item pass/fail criteria
8. Suspension criteria and resumption requirements
9. Test deliverables
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Schedule
15. Risks and contingencies
16. Approvals

Test Cases

CSSE 376, Software Quality Assurance

Rose-Hulman Institute of Technology

March 13, 2007

什么是测试用例

- 测试用例的定义
  - 满足特定目的的测试数据、测试代码、测试规程的集合
  - 是发现软件缺陷的最小测试执行单元
  - 有特殊的书写标准和基本原则

书写格式

字段名称	描述	
标志符	0001	
测试项	学生选课系统登录功能	
测试环境	测试环境要求	实际测试环境
	硬件： 软件： Window 2000 server, IE5.0以上浏览器 Web server 网络： Internet 工具： Rational robot2003 等等	
输入	输入标准	实际输入
	1. 输入正确的用户名密码 2. 输入错误的用户名密码	1. user1，pwd1 2. errorusr,pwd2
输出	输出标准	实际输出
	1. 正确进入系统 2. 错误提示	1. 正确进入系统 2. 错误提示
测试用例间的关联		

Topics in Software Dynamic White-box Testing

[Reading assignment: Chapter 7, pp. 105-122 plus many things in slides that are not in the book …]

黑盒测试VS白盒测试

黑盒测试	白盒测试
不涉及程序结构	考查程序逻辑结构
用软件规格说明生成测试用例	用程序结构信息生成测试用例
功能测试和系统测试	单元测试和集成测试
某些代码段得不到测试	对所有逻辑路径进行测试

白盒测试分类

- ControlFlow-testing
  - 逻辑分支覆盖法
    - 语句覆盖
    - 判定覆盖
    - 条件覆盖
    - 判定/条件覆盖
    - 条件组合覆盖
  - 路径法
    - 路径覆盖
    - 基本（独立）路径测试法
- DataFlow-testing

14.2.1逻辑覆盖

- 逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。这一方法要求测试人员对程序的逻辑结构有清楚的了解，甚至要能掌握源程序的所有细节。
- 由于覆盖测试的目标不同，逻辑覆盖又可分为：
  - 语句覆盖
  - 判定覆盖
  - 条件覆盖
  - 判定/条件覆盖
  - 条件组合覆盖

14.2.1.1语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。
- 这种覆盖又称为点覆盖，它使得程序中每个可执行语句都得到执行，但它是最弱的逻辑覆盖，效果有限，必须与其它方法交互使用。

To be continue…

14.2.1.2判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。判定覆盖又称为分支覆盖。

To be continue…

判定覆盖

- 判定覆盖只比语句覆盖稍强一些，但实际效果表明，只是判定覆盖，还不能保证一定能查出在判断的条件中存在的错误。因此，还需要更强的逻辑覆盖准则去检验判断内部条件。

14.2.1.3条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。

To be continue…

条件覆盖

- 条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却关心整个判定表达式的值。
- 但也可能有相反的情况：虽然每个条件都取到了不同值，但判定表达式却始终只取一个值。
- 条件覆盖不一定包含判定覆盖
- 判定覆盖也不一定包含条件覆盖。
- 为解决这一矛盾，需要对条件和判定兼顾。

To be continue…

判定-条件覆盖

- 判定—条件覆盖要求设计足够的测试用例，使得判定中每个条件的所有可能（真/假）至少出现一次，并且每个判定本身的判定结果（真/假）也至少出现一次。

14.2.1.5条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得每个判断的所有可能的条件取值组合至少执行一次。
- 显然，满足“条件组合覆盖”的测试用例是一定满足“判定覆盖”、“条件覆盖”和“判定—条件覆盖”的。

条件组合覆盖

- 这是一种相当强的覆盖准则，可以有效地检查各种可能的条件取值的组合是否正确。它不但可覆盖所有条件的可能取值的组合，还可覆盖所有判断的可取分支，但可能有的路径会遗漏掉。测试还不完全。

白盒测试技术

逻辑分支覆盖法

- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定/条件覆盖
- 条件组合覆盖

路径法

- 路径覆盖
- 基本（独立）路径测试法

白盒测试技术

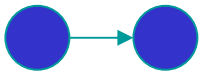
- 路径法
  - 路径覆盖
  - 基本（独立）路径测试法

14.2.1.6路径覆盖

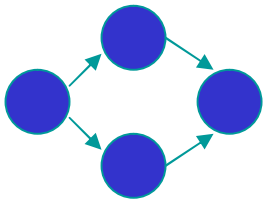
- 路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。这是最强的覆盖准则。但在路径数目很大时，真正做到完全覆盖是很困难的，必须把覆盖路径数目压缩到一定限度。

流图(flow graph)

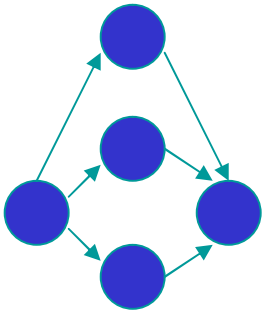
利用流图表示控制逻辑



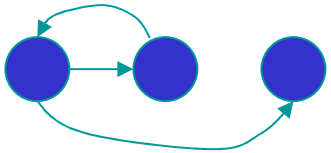
顺序结构



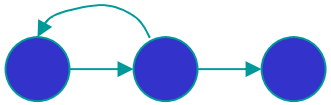
if 结构



Case 结构



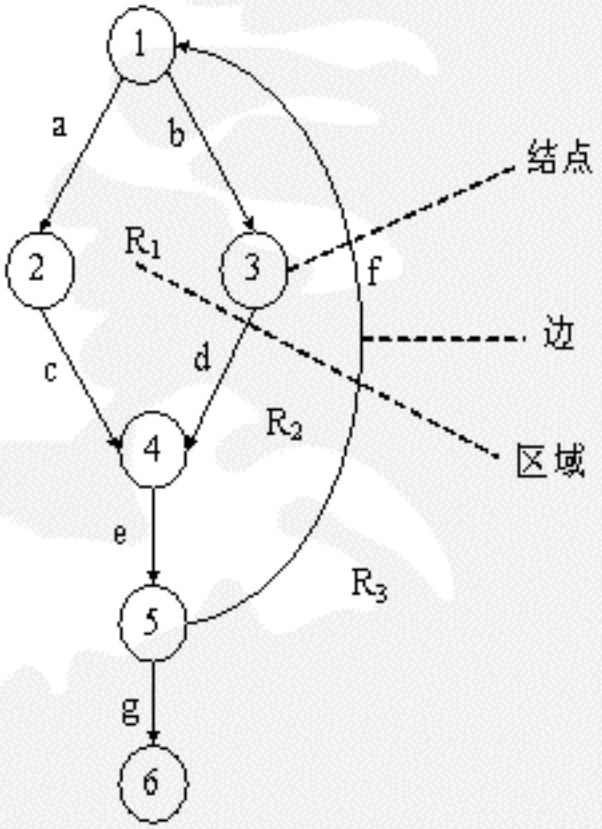
while 结构



until 结构

基本概念—控制流图

- 描述程序控制流的一种图示方式
- 结点：一个圆圈，表示单条或多条语句。
- 边：带箭头的线条，起始于一个结点，中止于一个结点。表示了控制流的方向。
- 区域：边和结点圈定的部分。



14.3.2.2 基本路径法

- 为解决难题只得把覆盖的路径数压缩到一定限度内。例如，只考虑循环的两个可能性：重复零次和多于零次的重复；对于do-while循环，两种，循环体执行一次和多于一次
- 使用线性独立路径

独立路径

- 定义
  - 从入口到出口的路径，至少经历一个从未走过的边。这样形成的路径叫独立路径。
- 优点
  - 减少路径数量
  - 包含所有的边和结点
- 缺点
  - 简化循环结构

基本路径测试步骤

①根据程序的逻辑结构画出程序框图
②根据程序框图导出流图
③计算流图G的环路复杂度V(G)
④确定只包含独立路径的基本路径集
⑤设计测试用例

程序框图 ==> 流图 ==> 基本路径 ==> 测试用例

基本路径

基本路径数（独立路径数）的计算

- 定义
  - 从入口到出口的路径，至少经历一个从未走过的边。这样形成的路径叫独立路径。

计算V(G)的不同方法

(1) V(G) =区域个数

(2) V(G) =边的条数-节点个数+2

(3) V(G) =判定节点个数+1

Topics in Software Dynamic White-box Testing
Part 2: Data-flow Testing

[Reading assignment: Chapter 7, pp. 105-122 plus many things in slides that are not in the book …]

Data-Flow Testing

- **Data-flow testing** uses the control flowgraph to explore the unreasonable things that can happen to data (*i.e.*, anomalies).
- Consideration of data-flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch or statement testing.

Data-Flow Testing (Cont' d)

- **Data-flow testing** is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- *E.g.*, Pick enough paths to assure that:
  - Every data object has been initialized prior to its use.
  - All defined objects have been used at least once.

Data Object Categories

- (d) Defined, Created, Initialized
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate

Example: Definition and Uses

```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4        w = x + 1;
     else
5.       y = y + 1;
6.   print (x, y, w, z);
```

Def	*C-use*	*P-use*
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

du Path Segments

- A **du Path** is a path segment such that if the last link has a use of **X**, then the path is simple and definition clear.

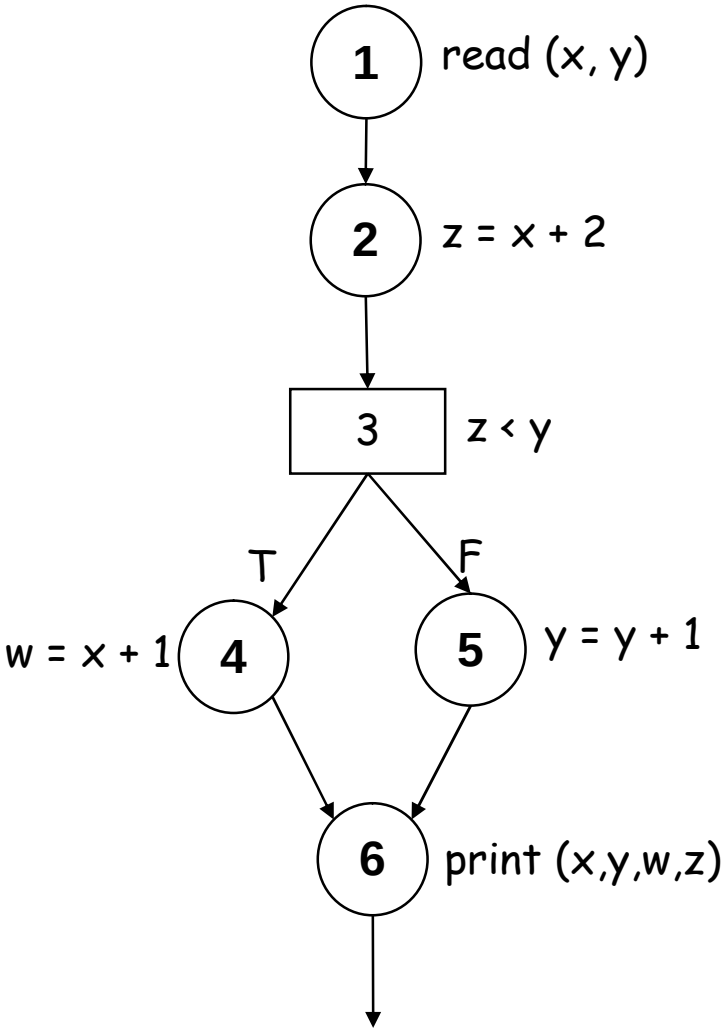
def-use Associations

- A def-use association is a triple (x, d, u,), where:

  x is a variable,
  d is a node containing a definition of x,
  u is either a statement or predicate node containing a use of x,

  and there is a sub-path in the flow graph from d to u
  with no other definition of x between d and u.

Example: Def-Use Associations



Some Def-Use Associations:

(x, 1, 2), (x, 1, 4), …

(y, 1, (3,t)), (y, 1, (3,f)), (y, 1, 5), …

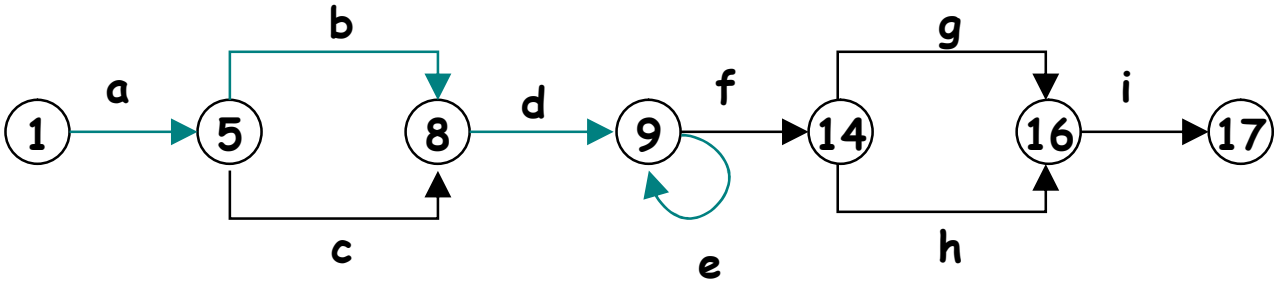
(z, 2, (3,t)),...

All du Paths Strategy (ADUP)

- **ADUP** is one of the strongest data-flow testing strategies.
- **ADUP** requires that every **du** path from every definition of every variable to every use of that definition be exercised under some test All **du** Paths Strategy (ADUP).

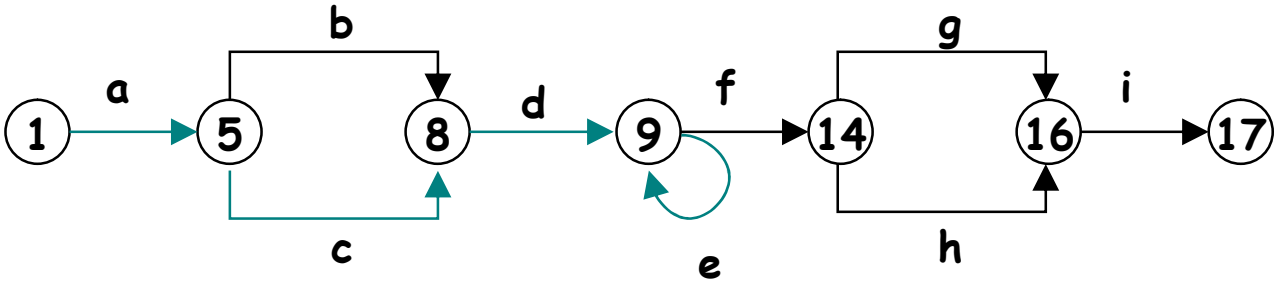
Example: pow(x,y)
du-Path for Variable x

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
*/
1       void pow (int x, y)
2       {
3       float z;
4       int p;
5       if (y < 0)
6          p = 0 – y;
7       else p = y;
8       z = 1.0;
9       while (p != 0)
10         {
11         z = z * x;
12         p = p – 1;
13         }
14      if (y < 0)
15         z = 1.0 / z;
16      printf(z);
17      }
```



Example: pow(x,y)
du-Path for Variable x

```
/* pow(x,y)
   This program computes x to the power of y, where x and y are integers.
    INPUT:     The x and y values.
    OUTPUT: x raised to the power of y is printed to stdout.
*/
1       void pow (int x, y)
2       {
3       float z;
4       int p;
5       if (y < 0)
6          p = 0 – y;
7       else p = y;
8       z = 1.0;
9       while (p != 0)
10         {
11         z = z * x;
12         p = p – 1;
13         }
14      if (y < 0)
15         z = 1.0 / z;
16      printf(z);
17      }
```



Summary

- Data are as important as code.
- Data-flow testing strategies span the gap between **all paths** and **branch** testing.填补路径和分支测试的缝隙

黑盒测试技术
black-box-testing

Dynamic black-box testing

- Dynamic black-box testing is testing without having an insight into the details of the underlying code.
  - Dynamic, because the program is running
  - Black-box, because testing is done without knowledge of how the program is implemented.
- Sometimes referred to as *behavioral testing*.
- Requires an executable program and a specification (or at least a user manual).
- Test cases are formulated as a set of

Black-box testing

- Characteristics of Black-box testing:
  - Program is treated as a black box.
  - Implementation details do not matter.
  - Requires an end-user perspective.
  - Test planning can begin early.

黑盒测试

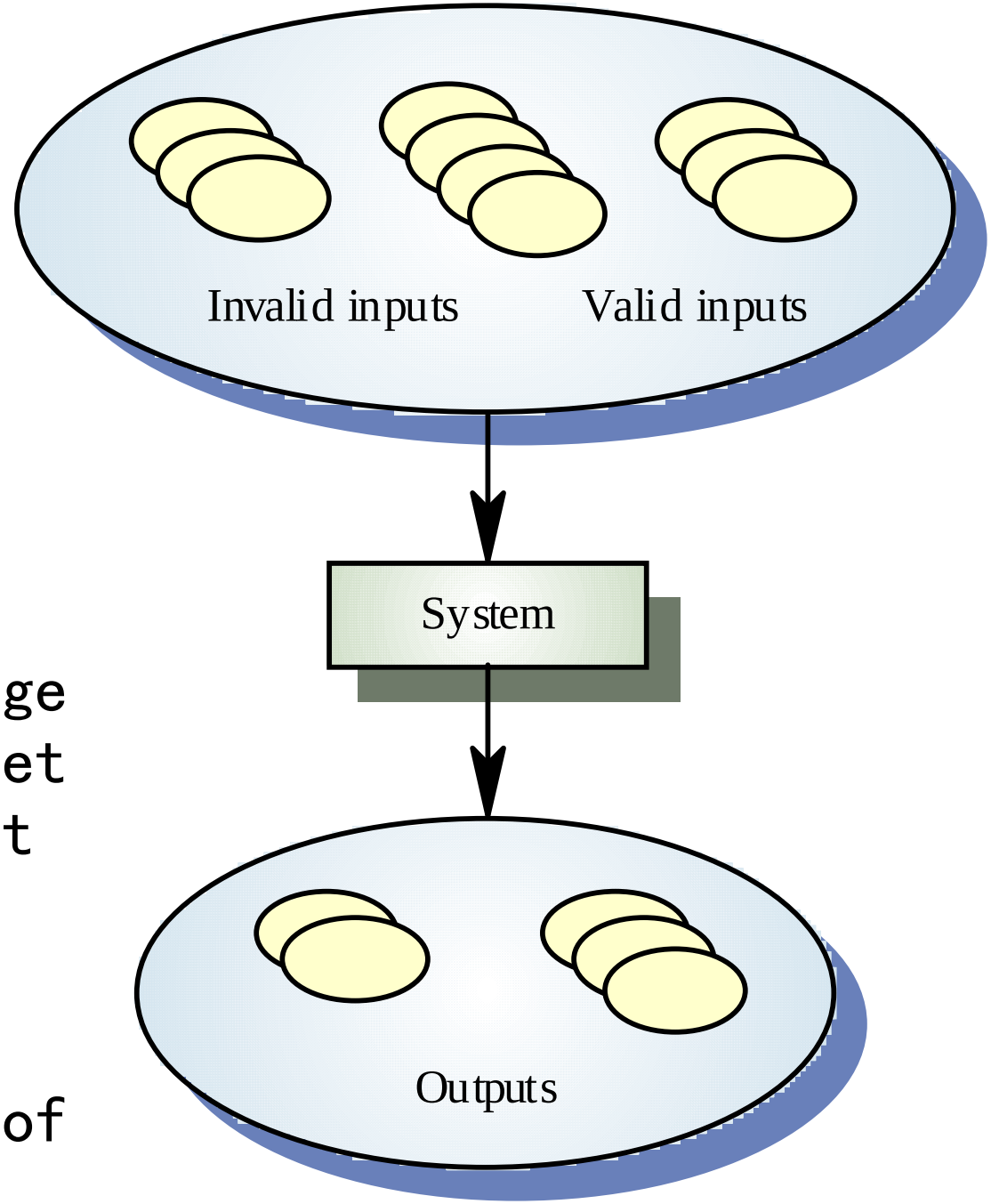
- **测试用例设计技术**
  - 等价类划分方法
  - 边界值分析方法
  - 错误推测方法
  - 判定表驱动分析方法
  - 因果图方法
  - 场景法

等价类划分法

- 等价类划分法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试用例。
- 每一类的代表性数据在测试中的作用等价于这一类中的其他值，也就是说，如果某一类中的一个例子发现了错误，这一等价类中的其他例子也能发现同样的错误；反之，如果某一类中的一个例子没有发现错误，则这一类中的其他例子也不会查出错误。

Equivalence Partitioning

- Equivalence partitioning is the process of methodically reducing the huge (or infinite) set of possible test cases into a small, but equally effective, set of test cases.



确定等价类的原则

- 1.在输入条件规定了取值范围或值的个数的情况下，则可以确立一个有效等价类和两个无效等价类。

确定等价类的原则

- 2.在输入条件规定了输入值的集合或者规定了“必须如何”的条件的情况下，可以确立一个有效等价类和一个无效等价类。

确定等价类的原则

- 3.在输入条件是一个布尔量的情况下，可确定一个有效等价类和一个无效等价类。

等价类划分

4. 在规定了输入数据的一组值 ，并且程序要对每一个输入值分别处理的情况下，可确立n个有效等价类和一个无效等价类

确定等价类的原则

- 5. 在规定了输入数据必须遵守的规则的情况下，可确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)。

确定等价类的原则

（6）如果确知，已划分的等价类中各个元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。

有效等价类：

无效等价类：

例如：if(i>=0&&i<=100)

```
switch(i){
    case 1:...;break;
    case 3:...;break;
    case 5:...;break;
    ...
    }
```

根据等价类创建测试用例的步骤

- 建立等价类表，列出所有划分出的等价类：

分析	有效等价类	无效等价类
…	…	…
…	…	…

·为每个等价类规定一个唯一的编号；

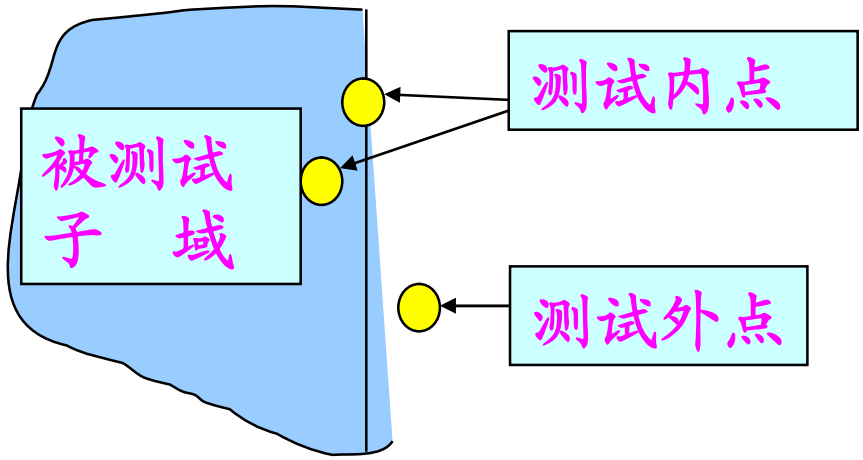
·设计一个新的测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类。重复这一步，最后使得所有有效等价类均被测试用例所覆盖；

·设计一个新的测试用例，使其只覆盖一个无效等价类。重复这一步使所有无效等价类均被覆盖。

14.3.2边界值分析法

（BVA，Boundary Value Analysis）

- 边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。
- 边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。
- 人们从长期的测试工作经验得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。因此针对各种边界情况设计测试用例，可以查出更多的错误。



边界值分析法

- 与等价划分的区别
  - 边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。
  - 边界值分析不仅考虑输入条件，还要考虑输出空间产生的测试情况。

边界值测试用例设计方法

- 边界值分析法：
  - 是等价划分法的一个补充
  - 程序的很多错误发生在输入或输出范围的边界上，因此针对各种边界情况设置测试用例，可以发现不少程序缺陷。
  - 设计方法：
    - 确定边界情况（输入或输出等价类的边界）
    - 选取正好等于、刚刚大于或刚刚小于边界值作为测试数据
    - 一个边界，加上左右边界，共三个测试用例

14.3.3错误推测法

- 通过经验和直觉推测出程序的错误所在；
- 主观、灵感、反向思维，难以复制等等；
- 不是一个系统的方法，用作辅助手段

策略

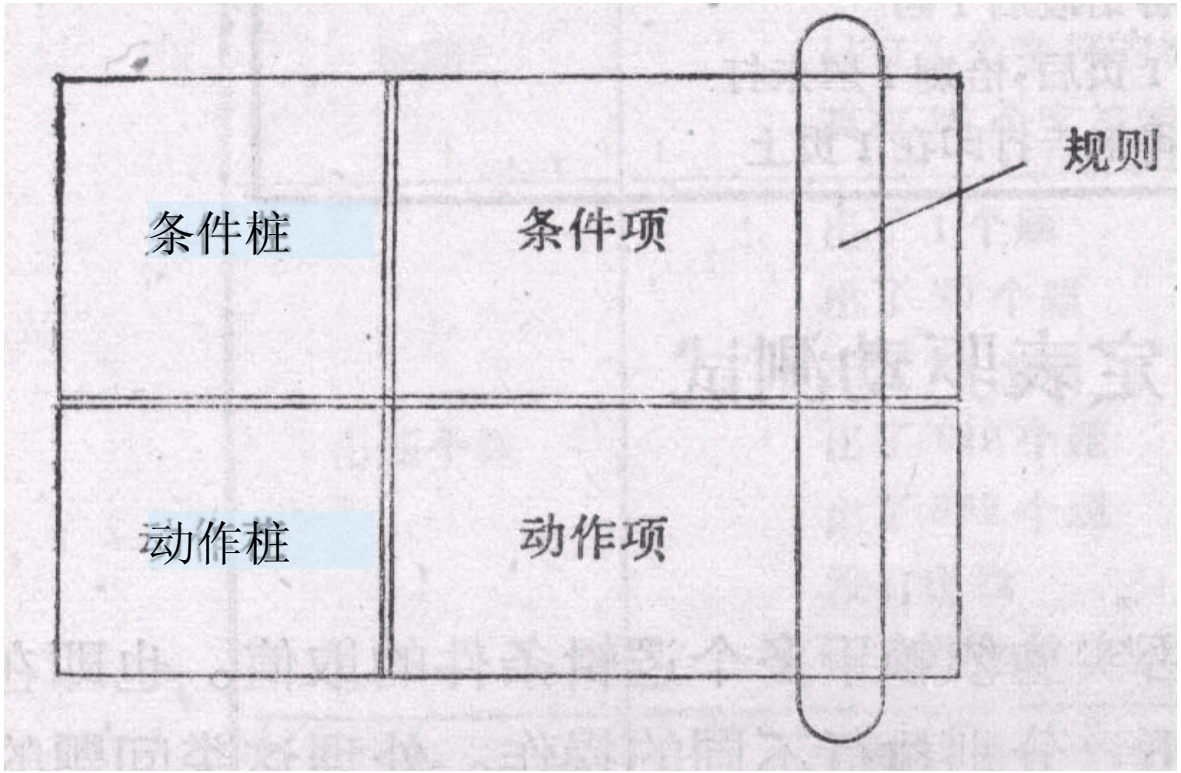
- 先等价类划分、再边界值分析、三错误推测法
- 不要搞倒了，实际中经常搞倒

14.3.4判定表驱动测试方法

- 后面的因果图方法中会到判定表。判定表（Decision Table）是分析和表达多逻辑条件下执行不同操作的工具。

判定表组成

- 判定表通常由五个部分组成：
  - 条件桩
  - 动作桩
  - 条件项
  - 动作项
  - 规则



判定表

下表是一张关于科技书阅读指南的判定驱动表：3个问题8种情况 2*2*2=8

		1	2	3	4	5	6	7	8
问题	你觉得疲倦吗？	Y	Y	Y	Y	N	N	N	N
	你对内容感兴趣吗？	Y	Y	N	N	Y	Y	N	N
	书中内容使你胡涂吗？	Y	N	Y	N	Y	N	Y	N
建议	请回到本章开头重读	x				x			
	继续读下去		x				x		
	跳到下一章去读							x	x
	停止阅读，请休息			x	x				

”读书指南”判定表

11.8判定表的建立步骤

- 判定表的建立步骤:（根据软件规格说明）

  ①确定规则的个数.假如有n个条件桩。每个条件桩有两个取值（Y,N），故有2^n种规则。

  ②列出所有的条件桩和动作桩。

  ③填入条件项。

  ④填入动作项。得到初始判定表。

  ⑤简化.合并相似规则（相同动作）。

判定表测试用例设计

- 测试用例设计：
  - 就是根据原始判定表的规则，设计测试用例，要求覆盖所有的原始判定表的规则

14.3.5因果图方法

- 前面介绍的等价类划分方法和边界值分析方法,都是着重考虑输入条件,但未考虑输入条件之间的联系，相互组合等。
- 考虑输入条件之间的相互组合,可能会产生一些新的情况.但要检查输入条件的组合不是一件容易的事情，即使把所有输入条件划分成等价类,他们之间的组合情况也相当多．因此必须考虑采用一种适合于描述对于多种条件的组合,相应产生多个动作的形式来考虑设计测试用例。这就需要利用因果图（ Cause—Effect Graphics ）方法。
- 采用因果图方法能够帮助我们按一定步骤，高效率地选择测试用例，同时还能为我们指出，程序规格说明描述中存在着什么问题。

因果图法

概念：

•借助图的方式，设计测试用例，被测程序有多种输入条件，输出结果依赖于输入条件的组合；

•着重分析输入条件的各种组合，每种组合就是一个“因”，它必然有一个输出的结果，这就是“果”；

•与其他的方法相比，更侧重于输入条件的组合

作用：

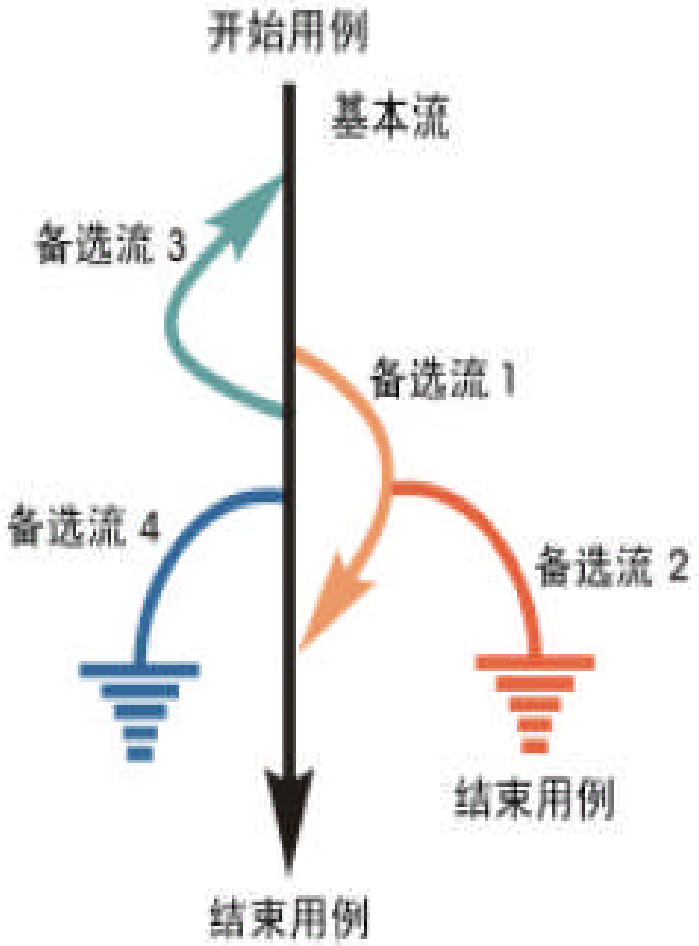
•能有效检测输入条件的各种组合可能引起的错误和结果；

•借助图进行测试用例的设计

14.3.6场景法

- 现在的软件几乎都是用事件触发来控制流程的，事件触发时的情景便形成了场景，
- 用例场景用来描述流经用例的路径，从用例开始到结束遍历这条路径上所有基本流和备选流。

基本流和备选流



基本流和备选流

- 按照上图中每个经过用例的路径，可以确定以下不同的用例场景：
  - 场景 1 基本流
  - 场景 2 基本流 备选流 1
  - 场景 3 基本流 备选流 1 备选流 2
  - 场景 4 基本流 备选流 3

基本流和备选流

- 场景 5 基本流 备选流 3 备选流 1
- 场景 6 基本流 备选流 3 备选流 1 备选流 2
- 场景 7 基本流 备选流 4
- 场景 8 基本流 备选流 3 备选流 4

- 注：为方便起见，场景 5、6 和 8 只考虑了备选流 3循环执行一次的情况。

黑盒测试策略

- 针对单个用例usecase，采用场景法，进行该用例的全方位测试；
- 后续针对单个功能进行细化测试，采用等价类、边界值、错误推测法，如果是多个条件的组合可以采用判定表或因果图法；

白盒法测试步骤：

- ⑴ 选择逻辑覆盖标准
- 显然只要程序不是太复杂，应该尽可能选择一种覆盖程度较深的标准，如条件组合覆盖。
- ⑵ 按照覆盖标准列出所有情况
- 所谓“情况”是指根据所选择的覆盖标准，列出能够满足此标准的可能的路径、条件等。

白盒法测试步骤：

- (3) 设计测试用例
- 能够满足覆盖标准的测试用例往往不止一个，应该设计选择高效的测试用例，即是用最少的用例能够满足覆盖标准。特别要注意：测试用例应由两部分组成：输入数据（测试用例）和预期的输出结果（正确结果），另外不同的覆盖方式，需要加入覆盖的判定、覆盖的条件、条件组合、覆盖的路径或基本路径等等。
- (4) 验证分析运行结果与预期结果
- 将运行结果与测试用例中的预期结果进行比较分析，可以帮助找出错误。

测试用例管理工具

- 测试用例管理工具：
  - 使用 TestLink 进行测试管理