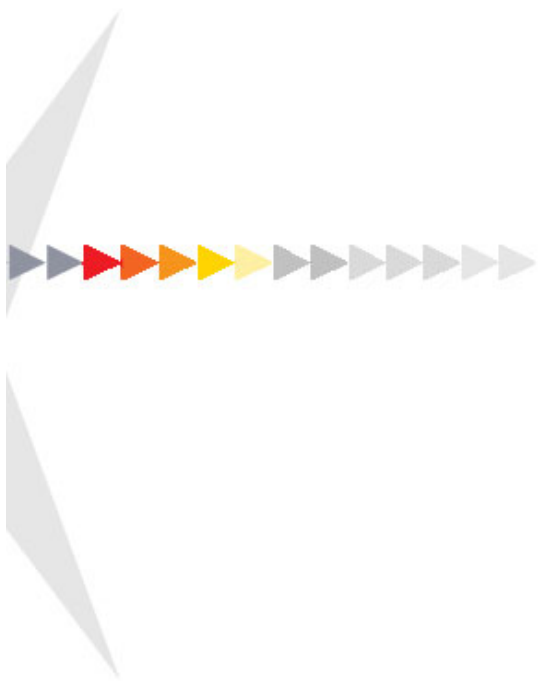


软件工程

大连理工大学软件学院



第11章 软件测试



- 质量保证：产品保证和过程保证
- 软件测试确保开发的软件满足用户需求
- 形式化验证方法，系统正确性验证的理论和技術
- 测试分类、测试策略、软件度量
- 等价类分析和基于控制流的测试
- 断言机制和测试框架是两种基本的测试实现技术
- 可测试性

形式化验证



- 软件测试的基本原理是根据用户需求的满足情况判断软件的质量情况。
- 为此可以根据不同的需求准备一系列的测试，让其以某种形式运行起来以实现检测。
- 测试的通过并不能用来证明整个系统是正确的，因为测试数量是有限的，测试内容的选择通常是对系统可能的缺陷进行分类并有针对性让其表现出来，并不能代表全部可能的情况。（穷举测试不可行）
- 可计算性理论中的停机问题告诉我们不可能用一个单独的程序来判定任意程序的执行是否能够终止。

证明系统



- 对任意的编程语言和逻辑进行正确性的证明。
- 通过对较小的已经被证明的程序及证明规则来推导出较大程序的正确性。
- 需求作为程序的一种性质以逻辑的方式体现。
- 如程序

p1

在具有性质**E**及其逻辑**B**满足的情况下，可实现性质**F**的要求；程序

P2

在满足性质**E**以及**not B**的逻辑下，也实现了性质**F**的要求；则规则：**If (B) then P1 else P2**满足性质**F**的要求。
- 模型检测如何在软件开发中应用仍是一个开放的问题，可以预见**OCL**是一个很好的切入点。

测试技术

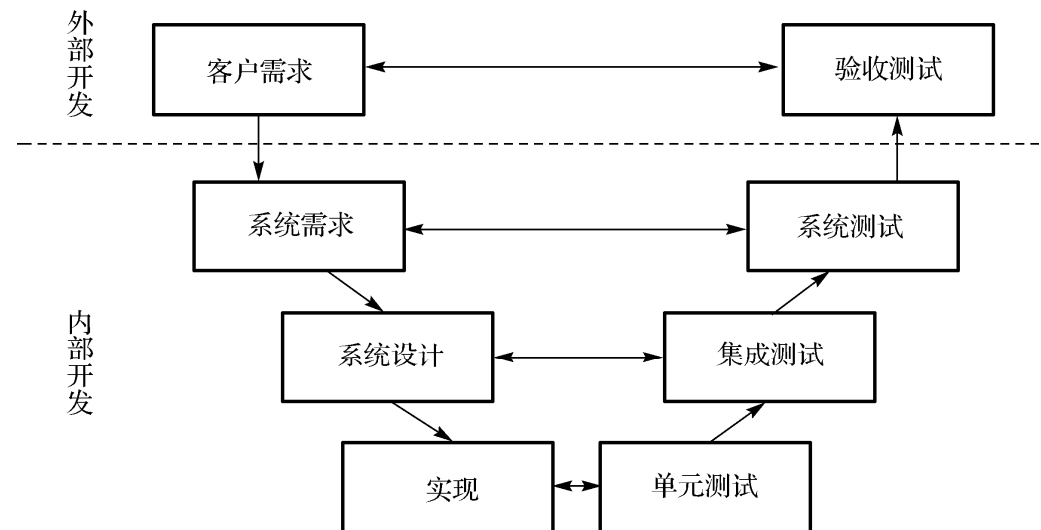


- 软件测试(**Software Testing**)的定义
 - 使用人工或自动的手段来运行或测定某个软件系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。
 - 软件测试通常被认为只是开发后期的一次性活动，但实际上，应该与整个开发流程紧密地融为一体。敏捷思想的普及使软件测试得到了前所未有的重视，它也是项目成功必不可少的基石。
- 测试用例(**Test Case**)是为某个特殊的测试目标而编制的一组测试输入、执行条件及预期结果，以便测试某个程序路径或核实软件是否满足某个特定需求。
 - 测试用例其实就是测试的具体内容，也是软件测试设计的主要工作。

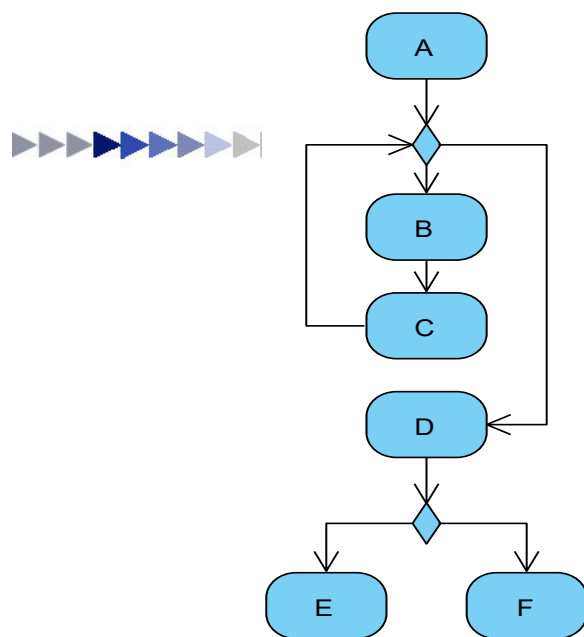
测试分类和测试V模型

- 在类测试（单元测试）阶段对每个单一的新开发的功能模块进行测试，并且多由开发者本人按照质量保证相关规范进行执行。
- 集成测试对经过单元测试的类逐步进行集成，构成最终的包或者系统的过程。重点关注每个类与其它相关类的交互过程。

- 系统测试将经过集成测试的构件或系统需要与其它通过预定义的接口进行通信的其它系统进行进一步的测试，从而构成最终完整的应用系统。
- 验收测试，一般由客户主导，主要是将系统测试中的工作在用户现场重复执行，并在此基础上加入一些客户自己的测试愿望，比如采用完全真实的现场数据。

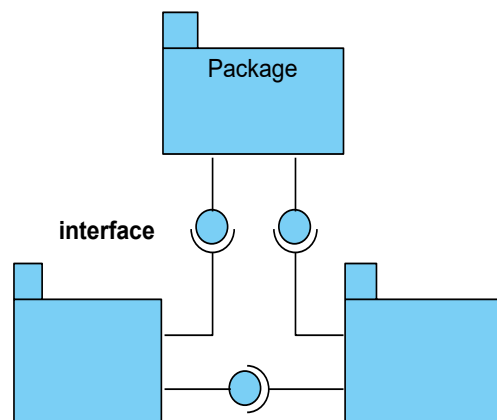


白盒测试



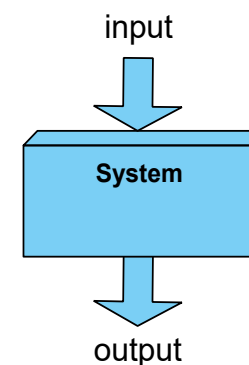
方法和类测试

灰盒测试



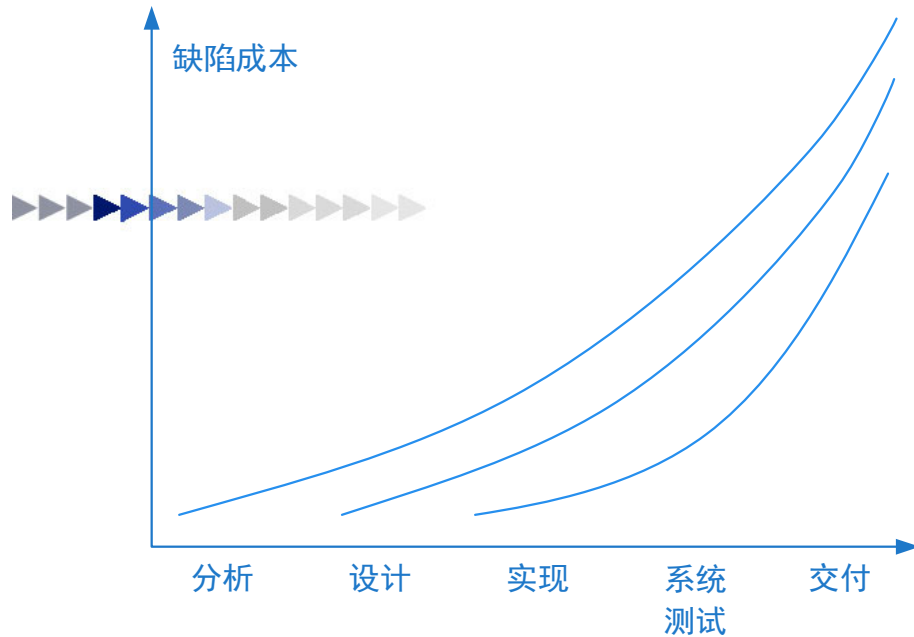
集成测试

黑盒测试

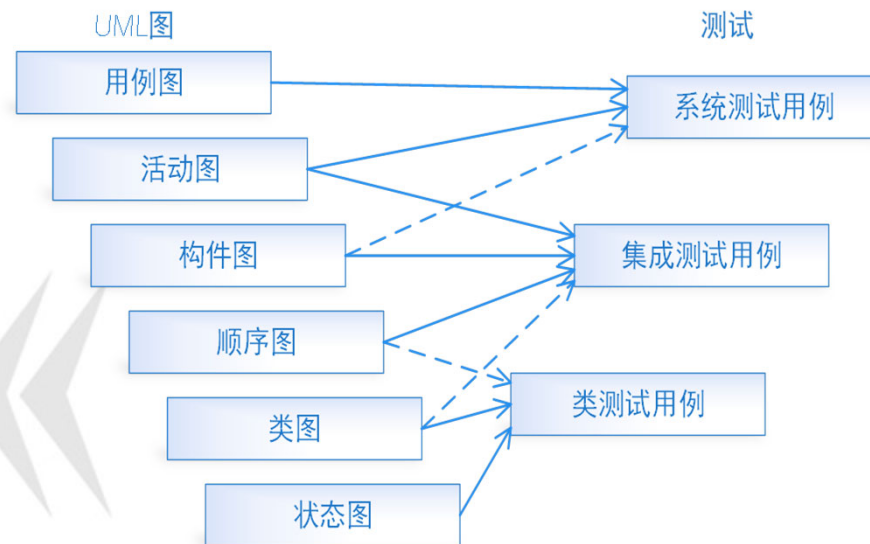


系统测试

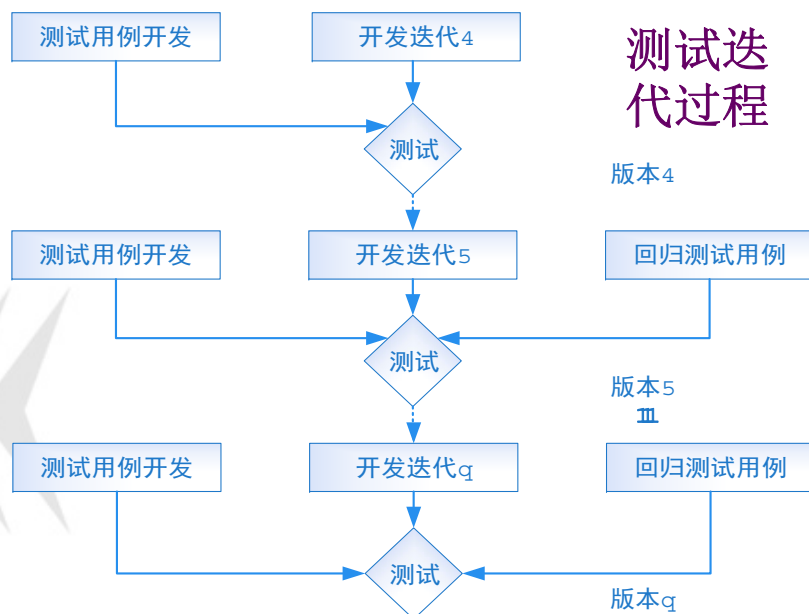
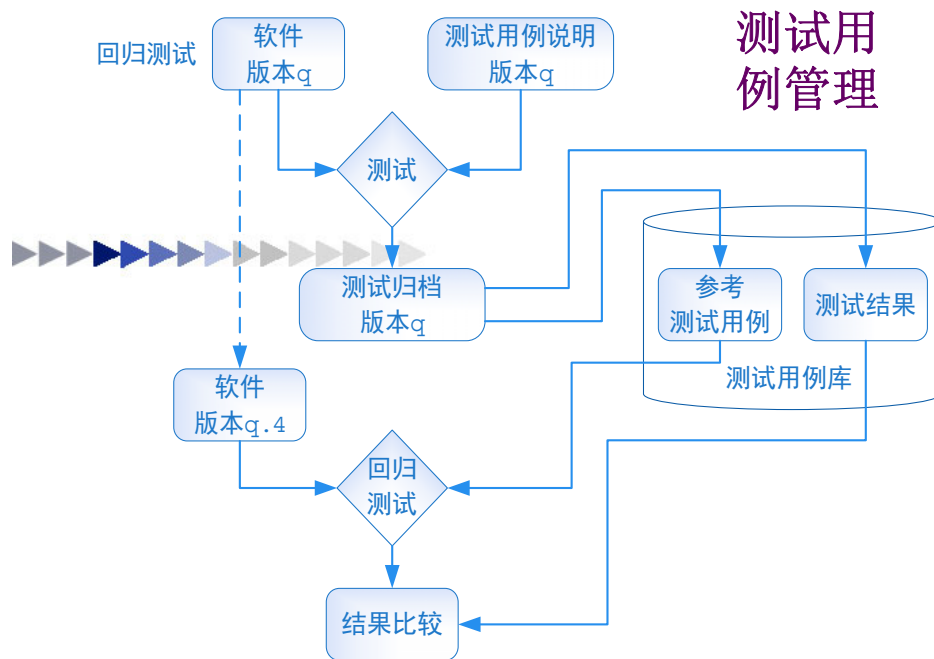
- 白盒测试关注的是被测对象的内部构成细节，比如算法的结构和流程，所以多在类测试阶段采用。
- 灰盒测试一般对应在集成测试阶段中使用，因为这个过程关注的是类、包等程序单元之间的关系，而不是类内部的细节。
- 在系统或验收测试阶段一般使用的是黑盒的测试方法，这里系统内部的细节不再重要，重要的是系统的外部行为。



- 一是缺陷修正成本与缺陷的类型有关，比如需求阶段的成本要比实现阶段高很多；
- 二是缺陷越早被识别并处理付出的成本也越低。

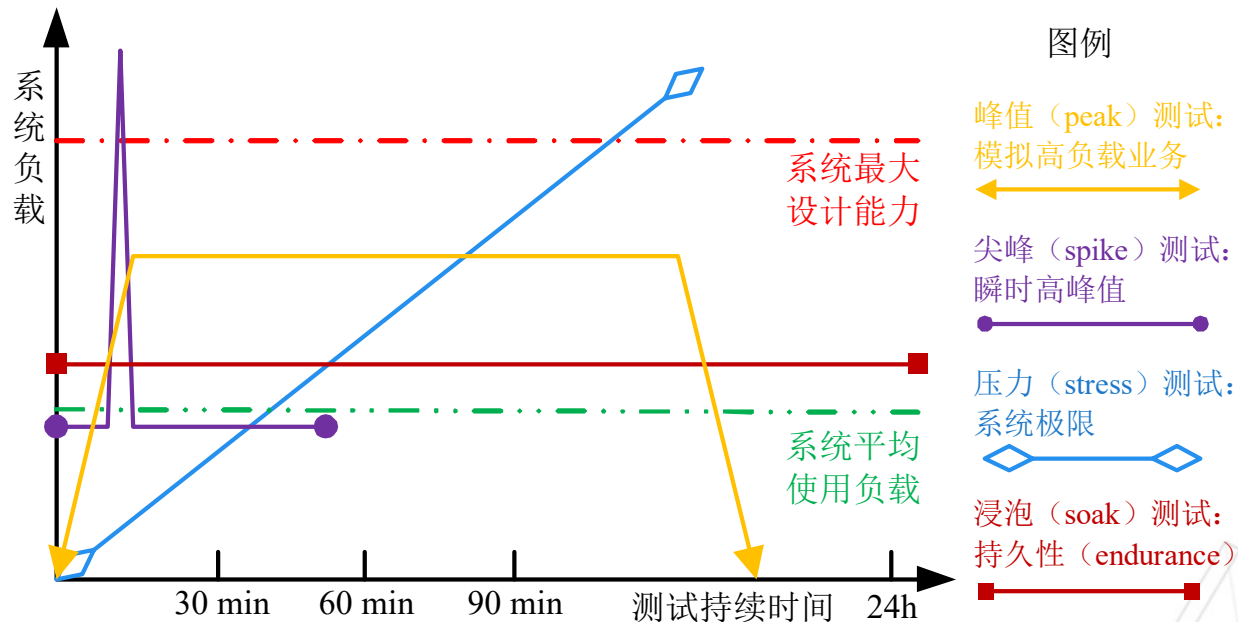


- 不同开发阶段中的**UML**模型和测试阶段之间在总体上的对应关系
- 在开发的早期就应该考虑到它们在后续测试中的可测试性以及如何对它们进行测试。



- 所有的测试用例都应该集中管理，需要为每个测试具体指定执行的条件以及预期的结果等。
- 这在增量式的开发过程中是非常必要的，因为上个迭代周期中的测试用例需要在本次的开发迭代测试中重新执行以确保对原有功能没有引入新的缺陷，这是一种回归测试的方式。
- 测试数据库在每次迭代中会加入新的测试用例并且要求其中所有的测试都要成功通过。
- 每次迭代的测试用例数量在逐渐增加，并且需要新旧测试用例共同执行完成本轮测试。

非功能测试



- 除功能性测试，对软件系统的测试还包括非功能性测试，如性能测试、安全测试、安装测试、配置测试、界面测试、容量测试等。
- 性能测试需要模拟实际用户负载来测试系统，包括反应速度、最大用户数、系统最优配置、软硬件性能、处理精度等。
- 性能测试一般结合压力测试、负载测试等手段。

界面测试



- 界面测试比较适合使用等价类的方法来建立对应的测试类。
- 界面测试常采用“捕捉和回放”（**Capture-and-Replay**）工具。部分工具支持对输出屏幕的比较。
- 测试用例的生成需要使用工具对被测界面通过一组标准的操作进行录制，然后在每个新的发布后对用例脚本进行回放，从而实现测试自动化的执行。

软件度量

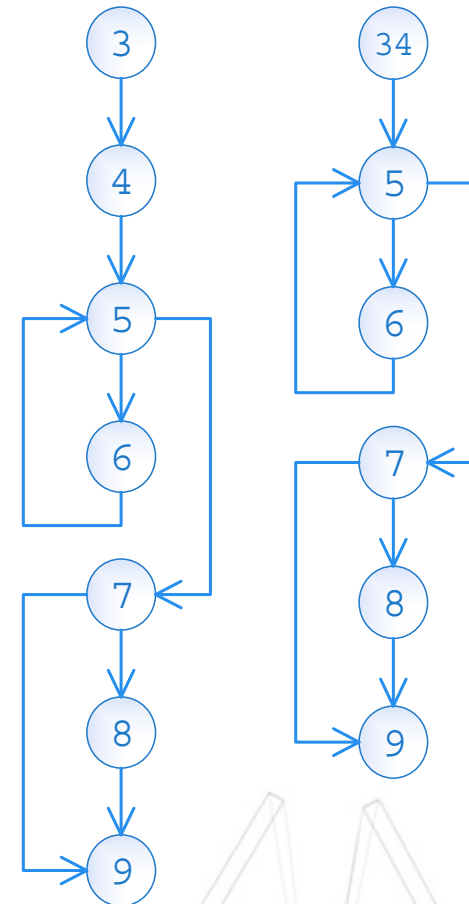


- 度量是确定软件质量的一种有价值的辅助手段。
- 一些质量度量相关的指标及作用：
 - 注释行数与代码行数的比例反映注释强度
 - 利用统计方法计算代码行数与方法数的比值确定出方法的平均长度
 - 保持变量和方法名适当的长度以提高程序的可读性
 - 方法中参数个数反映了方法的复杂程度
 - 类中实例变量的数目决定了该类信息的丰富程度
 - 继承深度提供了对继承使用是否恰当的信息，过多的继承对应深度的增加，并使得重用变得困难
- 方法复杂程度的度量：McCabe指标
- 类的内聚性的度量：LCOM*指标

```

public int test (int input) {
    int result = -input/2;           //0
    int i=input;                     //1
    while (i>0) {                     //2
        result = result + (i--);     //3
    }
    if (input < 0 || input%2 == 1) { //4
        result = 0;                  //5
    }
    return result*2;                  //6
} //正偶数求平方，其它返回零

```

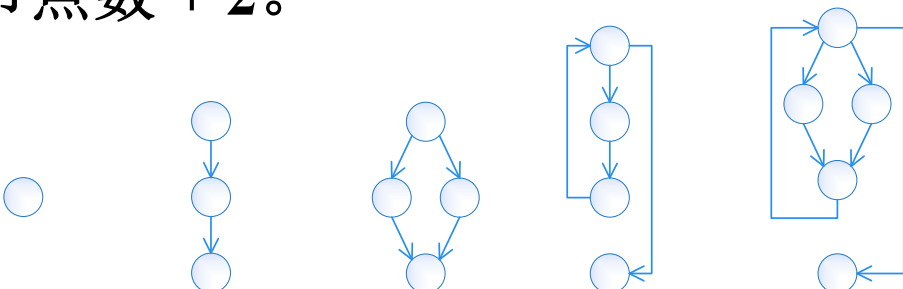


为尽量统一描述程序控制流程图，如果指令对应的节点 $k_1 \rightarrow k_2 \rightarrow \dots \rightarrow k_n$ 顺序出现，在以下情况将它们合并为一个节点进行处理：

1. 此序列的执行每次都是从 k_1 开始，除此之外没有边终止于 $k_2 \dots k_n$ ；
2. 此序列的执行每次都是以 k_n 结尾，除此之外没有边始于 $k_1 \dots k_{n-1}$ ；
3. 满足1和2的最长节点序列。

McCabe指标

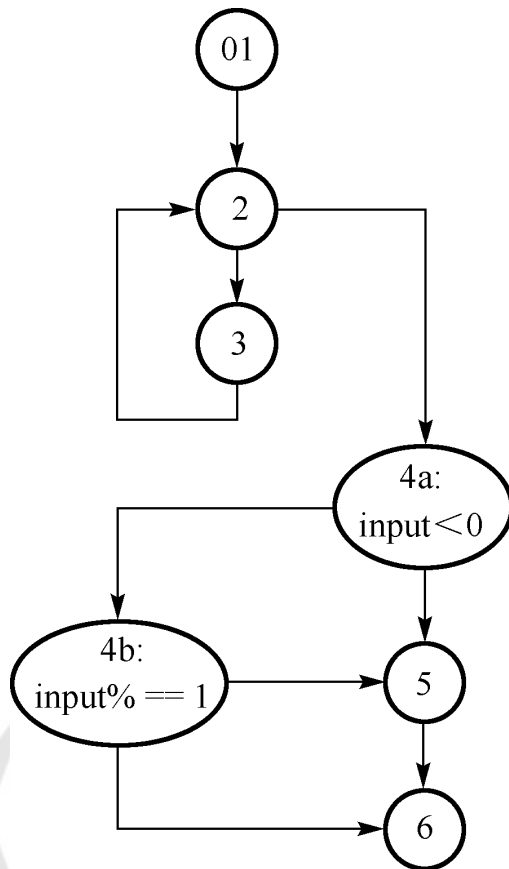
- **McCabe**环形复杂度，以方法的控制流程图结构为基础进行计算：边数 - 节点数 + 2。



边数	0	2	4	3	6
节点数	1	3	4	3	5
McCabe值	1	1	2	2	3

- 考虑到复合条件的情况，**McCabe**的计算实际上反映了方法中下列语句产生的分支结构：**if**语句、条件组合**&&**和**||**、**for**语句和**while** 语句。

McCabe计算



- 控制结构中分支或者循环，**McCabe**值越大，这也是程序可读性的一个反映指标。
- 公式中的“+2”的主要作用是对**McCabe**值的归一化，以保证其最小值为1。
- 在计算环形复杂度时需要注意，如果判断语句中含有多个原子谓词组合成的复合条件，那么需要将复合条件拆分成多个判定，并保证每个判定中只含有一个原子谓词。
- 边数为9，节点数为7，环形复杂度 $V(G)=9-7+2=4$ 。

```

// V(G) = 1
// +2 conditions, V(G) = 3:
if ((i > 13) || (i < 15)) {
▶▶▶▶▶ System.out.println("Hello, there!");
    // +3 conditions, V(G) = 6:
    while ((i > 0) || ((i > 100)
        && (i < 999))) {
        ...}
}
// +1 condition, V(G) = 7
i = (i==10) ? 0:1;
switch(a) {
    case 1: // +1, V(G)=8
        break;
    case 2: // +1, V(G)=9
    case 3: // +1, V(G)=10
        break;
    default:
        throw new
            RuntimeException("a = " + a);
}

```

2020/12/7

- 复杂度的计算过程的简化：
 - 每个代码段初始复杂度为1
 - 遇到每个原子条件加1
 - 每个switch中的case段加1
- 如果复杂度大于10，应考虑将该方法简化，而类中的方法McCabe值一般限制在5以下。
 - 优化算法结构，使其尽可能简单
 - 对if嵌套结构进行分解，将外层的if语句包含的程序部分移到另外一个局部方法中
 - 利用多态性使得对于分支的选择不再受开发者代码逻辑的控制，而是根据程序运行时的实际选择
- 程序结构越复杂越难于测试和理解，复杂的逻辑条件同样会使方法的可理解性降低。

LCOM*指标



- 度量LCOM*（**Lack of Cohesion in Methods**），分析每个类中方法与实例变量之间的关系，然后通过归一化公式进行计算。

$$LCOM^* = \frac{\left(\frac{1}{a} \sum_{j=1}^a \mu(A_j) \right) - m}{1 - m}$$

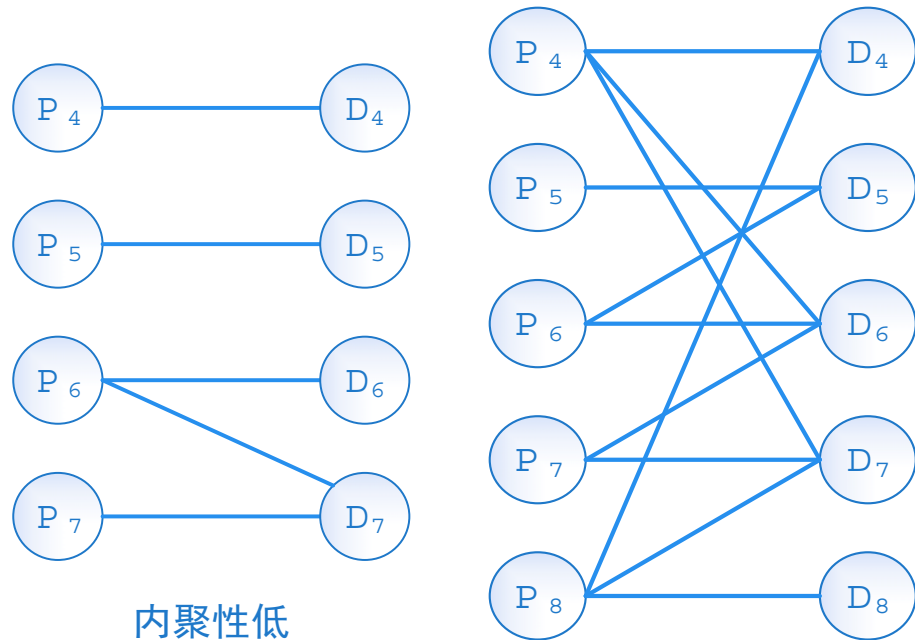
- **m**为方法数，**a**为所含的实例变量数， $\mu(A_j)$ 为访问每个实例变量的方法数。
- 当LCOM*为0时，该类的内聚性最佳，否则内聚性较差，需要考虑对其中的功能进行分解。当然如果该类只有一个唯一的实例变量，则不需要考虑它的LCOM*。

```

public class LCOMExample {
    private int a;
    private int b;
    private int c;
    public void do1(int x){
        a=a+x;
    }
    public void do2(int x){
        a=a+x;
        b=b-x;
    }
    public void do3(int x){
        a=a+x;
        b=b-x;
        c=c+x;
    }
}

```

$$LCOM^* = \frac{\frac{1}{3}(3+2+1) - 3}{1-3} = 0.5$$



内聚性低

内聚性高

- 由于get和set方法一般只对一个变量进行访问，为降低它们对LCOM*的影响，在计算时可不考虑类中的set和get方法。

等价类测试



- 等价类是离散数学中的一个概念，其基本思想是将一个集合按照一定的标准划分为若干个子集合，其中每个元素的归属依赖于指定功能下具体的行为。
- 对于数值型的集合，可根据输入变量的取值范围，产生一个有效等价类和两个无效等价类。
- 对于非数值类型的集合稍微复杂一些，比如对于枚举类型，假设合理的取值包括red、yellow和blue，则可以简单的将这些取值分别对应一个等价类，如果不允许其它值作为输入数据，则不存在它的无效等价类，例如Enumeration类型就是这样的情况。否则可以将所有其它输入对应一个无效等价类，例如包含任何符号的文本输入。

- 考查一个类的构造方法，用来创建某学生对象的数据，具有名字、出生年份和专业3个属性。名字属性要求不能为空，出生年份要求介于**1900**和**2000**之间；专业取值只能是枚举类型，包括“贸易（**TRADE**）、计算机（**CS**）、数学（**MATH**）”中的一个元素。则对于输入数据可产生如下的等价类：

E1) 名字非空（有效）

E2) 名字为空（无效）

E3) 出生年份小于**1900**（无效）

E4) 出生年份大于等于**1900**并且小于等于**2000**（有效）

E5) 出生年份大于**2000**（无效）

E6) 专业为国际贸易（有效）

E7) 专业为计算机科学（有效）

E8) 专业为工科数学（有效）

- 该方法有三个输入，进行调用时要同时指定这些参数，因此总的测试用例的数量与这些等价类的组合相关。
- 一种组合方式是对于有效等价类要尽可能采用少的测试用例进行覆盖，比如对于E1、E4和E6三个有效等价类可使用一个测试用例同时覆盖。
- 对于无效等价类则要慎重一些，其覆盖的规则是每个无效等价类必须与其它有效等价类组合测试，以此保证能够触发该无效值对应的专门处理过程。

测试用例	1	2	3	4	5	6
覆盖的等价类	E1 E4 E6	(E1) (E4) E7	(E1) (E4) E8	E2	E3	E5
名字	“杨楠”	“田亮”	“王东”	“”	“杨楠”	“杨楠”
出生年份	1987	1989	1985	1988	1892	2006
专业	TRADE	CS	MATH	TRADE	TRADE	TRADE
预期结果	ok	ok	ok	fail	fail	fail

- 在等价类的基础上还可以继续应用边界值分析的方法
- 数值类型的等价类上下边界容易确定，如E3的上边界1899和E5的下边界2001，E4的下边界1900和上边界2000。

- 还需要注意计算机中对数值的表达方式，不同的数值类型其能够表达的最小值和最大值的能力也不尽相同。

测试用例	1	2	3	4	5	6	7	8	9	10
覆盖的等价类	E1 E4D E6	(E1) E4D+(E6)	(E1) E4U E7	(E1) E4U- (E7)	(E1) E4 E8	(E1) E3U (E6)	(E1) E3 (E6)	(E1) E5D (E6)	(E1) E5 (E6)	E2 (E4) (E6)
名字	“杨楠”	“杨楠”	“田亮”	“田亮”	“王东”	“杨楠”	“杨楠”	“杨楠”	“杨楠”	“”
出生年份	1900	1901	2000	1999	1985	1899	1892	2001	2006	1988
专业	TRADE	TRADE	CS	CS	MATH	TRADE	TRADE	TRADE	TRADE	TRADE
预期结果	ok	ok	ok	ok	ok	fail	fail	fail	fail	fail

- 另外的组合方式是将3个变量具有的等价类分别进行组合，就会产生 $2*3*3=18$ 个测试用例，这将涵盖所有可能的输入情况，把这种组合方式称为**强等价类方法**。
- 这种测试用例的设计方法可以比较容易的实现，但前提条件是各个输入参数彼此独立并且随着等价类数目的增加所产生的测试用例数量也会急剧增加，因此在实际测试中这种方法并不很常用。
- 如果输入变量彼此间存在相互依赖的关系，则需要对业务规则进行仔细分析，才能达到最佳测试效果，比如对[这段程序的测试](#)。
- 这个简单的小例子说明了全面系统的设计出能够揭示所有问题的测试用例是一项复杂且耗时的工作。测试只是一种确认程序中在特定的形式下未存在缺陷的手段，但却不能用来作为程序正确性的保证，这种保证需要更为严谨和形式化的方法。

面向对象中的等价类



- 类作为一个整体在使用等价类方法进行测试时要将状态作为一种输入参数进行考虑。
- 类的状态是由其静态属性确定的，即实例变量。实例变量的数量决定了类的状态数量及其等价类的复杂程度。
- 在一个示例性的预定系统中，每个客户的信用等级通过一个专门的类**Credit**进行管理，该信用等级与客户的支付行为相关。
- 对超过一定金额的订单要求按照客户信用情况进行资金流动性的检查。
- 在对该类进行测试时要考虑到3个状态对应的3个有效等价类和订单金额的组合情况。

基于控制流的测试



- 等价类测试方法关注被测对象向外界提供的功能，测试用例设计并不依赖程序内部结构，被称为是一种**黑盒测试**的方法。
- **白盒测试**的思想是充分利用程序的结构信息来设计测试用例，以实现每个程序块（代码）的覆盖。覆盖程度和指标在等价类方法中通常无法保证和度量。
- 基于控制流的测试将程序段使用一个有向控制流图进行描述。
 - 流图中的节点表示代码指令，节点间通过有向直线进行连接，表示这些指令执行的先后顺序。
 - **If**、**Switch**和循环指令会导致在图中对应节点处产生分支。

覆盖指标

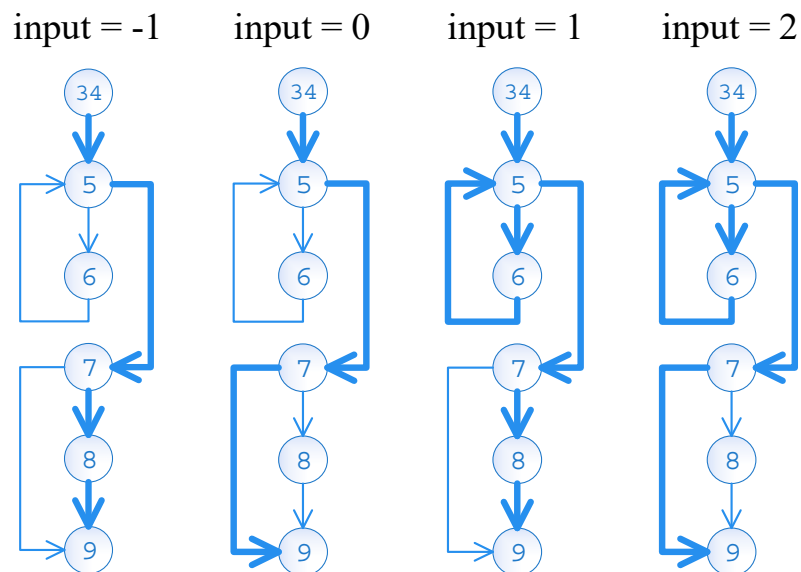


- 程序覆盖是提供一组测试用例尽可能使得覆盖率指标越大越好，或者说越接近1越好。
- 覆盖率指标有很多计算标准，其中较基础的有语句覆盖(**Statement Coverage**)、分支覆盖(**Branch Coverage**)、条件覆盖(**Condition Coverage**)、多条件组合覆盖(**Multiple Condition Coverage**)及路径覆盖(**Path Coverage**)等。

语句覆盖

- 语句覆盖表示在程序控制流图中测试经过的节点数与所有节点数的比例

$$\frac{\text{控制流图中测试经过的节点数}}{\text{所有节点数}}$$



语句覆盖是一种很粗略的度量，因为它主要关注的是控制流图中的节点而非执行路径

分支覆盖



- 分支覆盖的目标是尽可能覆盖控制流图中所有的边。

$$\frac{\text{控制流图中测试经过的边数}}{\text{所有的边数}}$$

- 分支覆盖的意义在于它要求对所有程序片段间的各种可能的连接至少执行一次，因此，满足分支覆盖要求一定会满足语句覆盖要求。

测试用例	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}
分支覆盖	4/7	3/7	6/7	5/7	5/7	7/7	7/7	6/7

条件覆盖

- 分支覆盖无法保证理论上所有可能的程序逻辑都会被测试到。
- 条件覆盖：要求每个原子谓词的真假两种取值都要取到。
- 条件覆盖不是根据程序的运行情况，而是根据出现的布尔条件进行测试用例的设计，条件覆盖与分支覆盖并没有直接的关系。

$$\frac{\text{取值为真的原子谓词} + \text{取值为假的原子谓词}}{2 \times \text{所有的原子谓词数}}$$

测试用例	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}	{-1,1,2}
i>0	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
input<0	t	f	f	f	t,f	f	f	t,f	t,f
input%2==1	-	f	t	f	f	f,t	t,f	t	t,f
条件覆盖	3/8	4/8	5/8	5/8	6/8	7/8	7/8	6/8	8/8

条件覆盖



- 高级编程语言中的短路（**short-circuit**）评估方式：
 - `if (x<4 || x/0==2) → if (x<4 | x/0==2) [java]`
- 条件覆盖与分支覆盖没有任何特别的联系，这表示分支的完全覆盖不能保证条件的完全覆盖，反之也成立，即完全的条件覆盖也不能保证分支的完全覆盖。
- 在分支覆盖和条件覆盖的基础上，还衍生出一种条件/分支覆盖标准，它是两种覆盖的混合，要求同时满足两种覆盖的要求。

多条件组合覆盖

- 所有的覆盖要求在多条件组合覆盖标准中得到了综合，它要求所有在条件中出现的原子谓词的组合都要覆盖到。

$$\frac{\text{取值为真的原子谓词数} + \text{取值为假的原子谓词数}}{2 \times \text{所有的原子谓词数}}$$

测试用例	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}	{-1,1,2}
i>0	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
input<0	t	f	f	f	t,f	f	f	t,f	t,f
input%2==1	-	f	t	f	f	f,t	t,f	t	t,f
input<0 input %2==1	t	f	t	f	t,f	f,t	t,f	t	t,f
多条件组合覆盖	3/8	4/8	5/8	5/8	6/8	7/8	7/8	6/8	8/8

路径覆盖



- 路径覆盖(Path Coverage)是度量一个方法中所有可能路径的覆盖情况。
- 路径是指从方法的入口开始到出口结束，由控制流图中若干条边构成的一条唯一的执行路线，也可以看成由若干可能的逻辑条件的组合。
- 由于循环结构会使得路径的数量剧烈地增长，因此路径覆盖只考虑有限循环次数的情况。
- 基本路径测试是路径覆盖的典型方法，它是在控制流图的基础上，通过分析控制结构的环环复杂度，导出基本可执行的路径集合，从而设计测试用例的方法。

基本路径测试

- 绘制程序的控制流图。
- 计算**McCabe**环形复杂度。从程序的环形复杂性可导出程序基本路径集合中的独立路径条数，独立路径要求在路径中至少含有一条未曾使用过的边。
- 导出测试用例。为每一条基本路径设计测试用例的数据输入和预期结果，并确保覆盖到基本路径集中的每一条路径，参照环形复杂度规定的上限路径条数。

用 例	路 径	输 入	预期输出
1	01→2→4a→5→6	{-1}	0
2	01→2→4a→4b→6	{0}	0
3	01→2→4a→4b→5→6	-	-
4	01→2→3→2→4a→5→6	-	-
5	01→2→3→2→4a→4b→6	{2}	4
6	01→2→3→2→4a→4b→5→6	{1}	0

断言 (Assertion)



- 很多高级编程语言提供断言指令，如Java、C#和C++中，一般形式： **Assert <Boolean condition>**
- 断言提供对异常进行检查的能力，但只能在开发阶段使用，并且不能替代常规的异常处理
- 断言规定了方法必须要满足的条件，即需求的程序特性。如对输入不能确定是否满足既定要求，比如在防御性程序设计（**defensive programming**）中要求方法总是可用，则必须要使用异常处理而不能使用断言
- **Java断言指令的扩展：**

assert <boolean condition>: <any object>

- 位于函数尾部的断言可以用来检验是否该方法的计算是期望的结果，或者说可以用来检验计算出的结果是否具有期望的某些属性。[对应代码](#)

- “**assert false;**” 这个断言表示永远都不会通过，可将其放置于不应该到达的地方，以确保此处不可达。[对应代码](#)

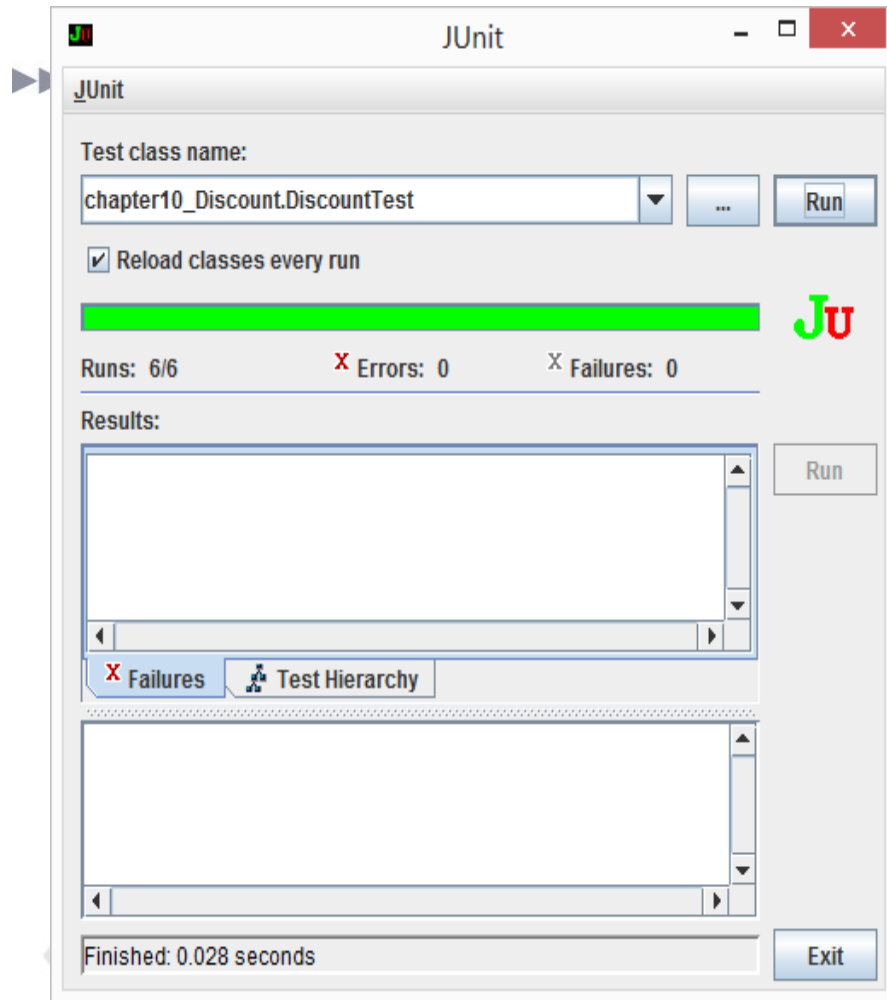
- 对断言的使用要确保不会带来任何的副作用，也就是说不会改变实际类的状态。
- 比如，对于以下迭代器**iter**的断言是不合适的，因为该断言检测后会导致迭代器状态的改变，使其指向了下一个对象。

assert iter.next() != null;

测试框架之Junit



- 首先由**Kent Beck**提出并完成的测试框架，允许使用各种程序设计语言来创建测试，而不是花费过多的精力在整个测试环境的创建上。
- 首先在**Smalltalk**平台——**SUnit**框架，后来又在不同的语言平台上进行移植，如**Java**、**C#**、**C++**等，称为单元测试框架。
- 基本思想是对不同的测试用例创建与其对应的测试方法，测试用例的执行和评价由**JUnit**接管。
- 负责所有测试用例执行，得到执行的测试数、通过的测试数以及失败的测试数等。



- **JUnit**可以使用反射机制在测试环境中调用传递过来的方法，所有某类的**test**测试方法都放在一个测试类中，这个类需要从**JUnit**提供的系统类**TestCase**继承而来。
- 方法**setup**在每个测试开始之前执行，其中可以对实例变量等环境相关的内容进行设置。
- 每个测试结束后会执行**tearDown**方法，比较适合做一些清理工作。
- 对某些特征进行检验，通过调用源自类**TestCase**的父类中**Assertion**的方法。

Junit开发实例



- 该例开发由一个[折扣类Discount](#)负责存储当前折扣情况，其中含有一个**discount**的实例变量。
- 另外该类提供一个对某个客户“锁定”的能力，对锁定客户的折扣计算将会导致一个异常**DiscountException**的抛出并被终止。
- 测试类对**set**和**get**方法，如果大多是由代码生成工具自动产生的，则可以略去对它们的检查。
- 测试过程经常对步骤进行分解，由多个独立的测试用例顺序执行，形式上多是若干**assert**指令先后出现，如果第一个产生错误，后续的**assert**将不会被继续执行。
- 有时测试用例并不独立，即测试过程需要两个测试用例紧密配合执行，比如[以下的例子](#)。

- 在很多大型的项目开发中，需要对测试用例进行收集和管理并在合适的时刻重复执行所有的测试用例或者根据情况重复执行部分的测试用例，这叫做回归测试。
- **JUnit**使用**TestSuite**的概念来支持对测试用例集合的管理。首先创建**TestSuite**对象，然后通过该对象实现对指定测试用例的组织，如[代码](#)所示。
- [JUnit 4的测试代码](#)示例。
- **JUnit**比较适合开发人员对自己创建的单个类进行测试。事实上，**JUnit**存在一个强有力的竞争对手，即**TestNG**框架。
- 两个框架的不同在于它们的核心设计。
 - **JUnit**一直是一个单元测试框架，也就是说，其构建目的是促进单个对象的测试；
 - 而**TestNG**则是用来解决更高级别的测试问题，它具有**JUnit**中所没有的一些特性，比如依赖性测试、参数化测试以及多线程测试等特性。

可测试性



- 测试的构建一般采用的方式是“自底向上（**Bottom Up**）”的方式，也就是新的测试总是在已经经过测试的类和方法的基础上进行构建，比如要为一个依赖（使用）**Discount**类的另外一个类创建测试，则无需在这个测试类中重新测试**Discount**的**price()**方法。
- 测试进行的过程中尤其是在多人并行开发的时候经常会出现彼此依赖的情况。为避免等待而影响开发进度，可以构建一个模拟程序（**mock**）或桩（**stub**）。
- 通过上面的例子和分析，在进行类开发的时候就要为其可测试性做好设计上的准备：可测试性构建的原则。

设计简单的方法



- 规模较小的方法的类的测试性要好于带有较少方法但每个方法的规模较大的类。
- 方法的规模越大，对其所有方面的测试就越困难。
- 方法的调用接口要尽量简单，这符合最少知识原则，它主张一个类只要知道它需要知道的那么多即可。

`getRent(car, customer)`

`getRent(car.getDays(), car.getPrice(), customer.getDiscount())`

避免私有方法



- 方法的可测试性是比较差的，由于封装性，普通的JUnit类无法直接对它们进行访问。
- JUnit 4之后的protected方法的情况将有所好转，因为一个测试类可以从任意类中继承，如可从测试的目标类继承。
- 由于private的特点，在设计相关的类方法时，应考虑尽量使用它们来完成特别简单的任务。

优先使用通用方法



- 静态代码无法应用多态的特性，也就意味着没有代码重用，无论是对被测程序还是测试程序。
- 这就可能会导致应用程序与测试代码的冗余，也无法实现测试代码与程序代码的轻松替换。
- 多态的另外一个好处是能够较容易的降低方法的复杂程度，避免使用较长的**switch**和**if**语句。

组合优于继承



- 类之间通过关联进行组合的关系更易于测试。
- 因为运行时，代码不能改变继承的层次结构，但可以组合不同的对象，使得对象易于从一种状态切换到另外一种状态，也就易于测试。



避免隐藏的依赖关系与全局状态



- 对全局状态一定要谨慎，因为全局状态使测试的构建变得复杂，如果全局状态没能共享或者遗漏，都会导致一些意外的后果。
- 隐藏的依赖关系也一样，这些都会使测试过程变得艰难。

```
public void testCreditCardCharge(){  
    Database.init();  
    OfflineQueue.init();  
    CreditCardProcessor.init();  
    CreditCard c = new CreditCard("1234 5678 9012 3456", 5, 2018);  
    c.charge(100);  
}
```

建设性质量保证

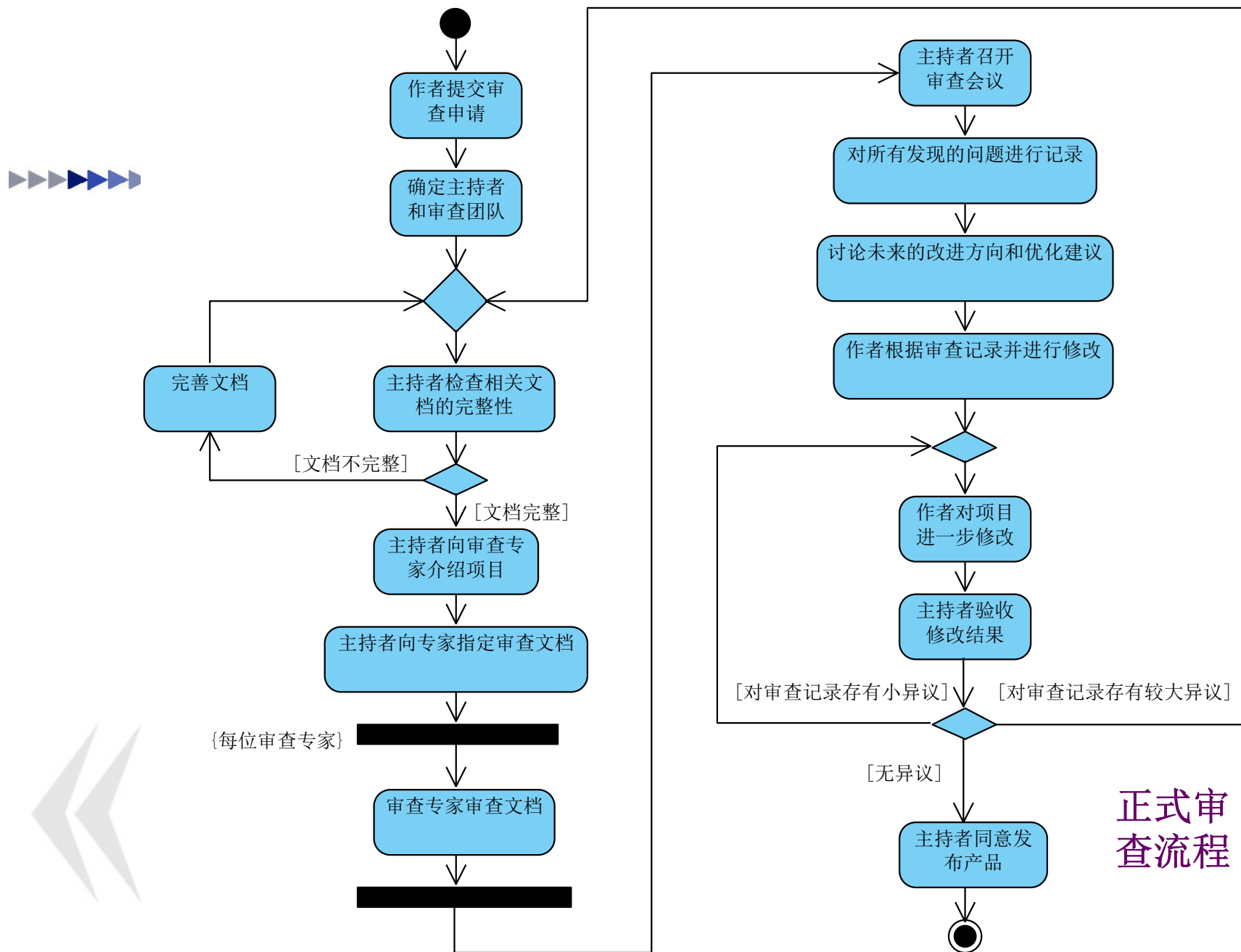


- 在软件开发后作用于被测软件从而发现缺陷的存在，通常被称为分析式的质量保证方法（**Analytical QA**）。
- 将在实际产品能够执行之前的质量活动称为是建设性的质量保证活动（**Constructive QA**）。
- 这些活动可能针对具体产品，如需求分析文档或者程序代码，也可能整个开发过程，包括：
 - 原则（**Principles**），技术（**technique**），方法（**methods**）和工具（**tools**）
 - 生命周期模型（**Life-cycle models**）
 - 文档（**Documentation**）
 - 需求（**Requiriements**）：系统化的（**systematic**）和正式的（**formal**）表示
 - 开发环境（**Development environments**）
 - 配置管理（**Configuration management**）
 - 人件（**Peopleware**）

人工测试



- 审查（**Inspection**）活动的重点在于整个过程对时间的计划，因为正式的分析过程需要耗费人员大量的时间和精力。
- 评审（**Review**）与审查比较起来主要的差别在于准备工作没有那么正式，同样需要所有评审专家对产品进行通篇的审阅，然后将发现的问题进行标注并给出评价意见。作者出席评审会议，可以针对专家的疑问进行回答，但一般不主动提出问题。作者在会后获得会议中所有评审专家的评审意见和他们的评论。
- 走查（**Walkthrough**）中作者作为主持人主持整个讨论过程，作为会议的主导介绍自己的工作产品。其它的参与者听取作者的讲解并提出一些批判的问题或改进的建议。为使大家的讨论更有针对性，每个参与者也应该在之前对产品进行一定的了解和分析，这对于产品质量的提高也会起到积极的作用。



正式审查流程

作业



- 习题1~3
- 习题4（选做）

