密　　级：普通
文件编号：NO.2
文件类别：测试管理体系文件
发 放 号：1002

应用软件

测试用例设计指南

北京梅梅出品有限公司

版本说明

日期	版本号	发布说明	作者	批准人	
				签字	岗位

目录

1、引言

测试设计遵循与软件设计相同的工程原则。好的软件设计包含几个对测试设计进行精心描述的阶段。这些阶段是：

- 测试策略
- 测试计划
- 测试描述
- 测试过程

上述四个测试设计阶段适用于从单元测试到系统测试各个层面的测试。

测试设计由软件设计说明所驱动。单元测试用于验证模块单元实现了模块设计中定义的规格。一个完整的单元测试说明应该包含正面测试（Positive Testing）和负面的测试（Negative Testing)。正面测试验证程序应该执行的工作，负面测试验证程序不应该执行的工作。

设计富有创造性的测试用例是测试设计的关键。本文档介绍了测试说明的一般设计过程，描述了一些结构化程序设计单元测试中采用的用例设计技术，同时也增加了面向对象编程中对类进行单元测试所采用的测试用例设计技术，这些可作为软件测试人员的参考阅读资料。

2、设计单元测试说明

一旦模块单元设计完毕，下一个开发阶段就是设计单元测试。值得注意的是，如果在书写代码之前设计测试，测试设计就会显得更加灵活。一旦代码完成，对软件的测试可能会倾向于测试该段代码在做什么（这根本不是真正的测试），而不是测试其应该做什么。单元测试说明实际上由一系列单元测试用例组成，每个测试用例应该包含 4 个关键元素：

- 被测单元模块初始状态声明，即测试用例的开始状态（仅适用于被测单元维持了调用间状态的情况）；
- 被测单元的输入，包含由被测单元读入的任何外部数据值；
- 该测试用例实际测试的代码，用被测单元的功能和测试用例设计中使用的分析来说明，如：单元中哪一个决策条件被测试；
- 测试用例的期望输出结果，测试用例的期望输出结果总是应该在测试进行之前在测试说明中定义。

以下描述进行测试用例设计，书写测试说明的 6 步通用过程

2.1 测试用例设计步骤

2.1.1 步骤 1：首先使被测单元运行

任何单元测试说明的第一个测试用例应该是以一种可能的简单方法执行被测单元。看到被测单元第一个测试用例的运行成功可用增强人的自信心。如果不能正确执行，最好选择一

个尽可能简单的输入对被测单元进行测试/调试。

这个阶段适合的技术有：

- 模块设计导出的测试
- 对等区间划分

2.1.2 步骤 2：正面测试(Positive Testing)

正面测试的测试用例用于验证被测单元能够执行应该完成的工作。测试设计者应该查阅相关的设计说明；每个测试用例应该测试模块设计说明中一项或多项陈述。如果涉及多个设计说明，最好使测试用例的序列对应一个模块单元的主设计说明。

适合的技术：

- 设计说明导出的测试
- 对等区间划分
- 状态转换测试

2.1.3 步骤 3：负面测试(Negative Testing)

负面测试用于验证软件不执行其不应该完成的工作。这一步骤主要依赖于错误猜测，需要依靠测试设计者的经验判断可能出现问题的位置。

- 适合的技术有：
- 错误猜测
- 边界值分析
- 内部边界值测试
- 状态转换测试

2.1.4 步骤 4：设计需求中其它测试特性用例设计

如果需要，应该针对性能、余量、安全需要、保密需求等设计测试用例。

在有安全保密需求的情况下，重视安全保密分析和验证是方便的。针对安全保密问题的测试用例应该在测试说明中进行标注。同时应该加入更多的测试用例测试所有的保密和安全冒险问题。

适合的技术：

- 设计说明导出的测试

2.1.5 步骤 5：覆盖率测试用例设计

应该或已有测试用例所达到的代码覆盖率。应该增加更多的测试用例到单元测试说明中以达到特定测试的覆盖率目标。一旦覆盖测试设计好，就可以构造测试过程和执行测试。覆盖率测试一般要求语句覆盖率和判断覆盖率。

适合的技术：

- 分支测试
- 条件测试
- 数据定义－使用测试

- 状态转换测试

2.1.6 步骤 6：测试执行

使用上述 5 个步骤设计的测试说明在大多少情况下可以实现一个比较完整的单元测试。到这一步，就可以使用测试说明构造实际的测试过程和用于执行测试的测试过程。该测试过程可能是特定测试工具的一个测试脚本。

测试过程的执行可以查出模块单元的错误，然后进行修复和重新测试。在测试过程中的动态分析可以产生代码覆盖率测量值，以指示覆盖目标已经达到。因此需要在测试设计说明中需要增加一个完善代码覆盖率的步骤。

2.1.7 步骤 7：完善代码覆盖

由于模块单元的设计文档规范不一，测试设计中可能引入人为的错误，测试执行后，复杂的决策条件、循环和分支的覆盖率目标可能并没有达到，这时需要进行分析找出原因，导致一些重要执行路径没有被覆盖的可能原因有：

- 不可行路径或条件 —— 应该标注测试说明证明该路径或条件没有测试的原因。
- 不可到达或冗余代码 —— 正确处理方法是删除这种代码。这种分析容易出错，特别是使用防卫式程序设计技术（Defensive Programming Techniques）时，如有疑义，这些防卫性程序代码就不要删除。
- 测试用例不足 —— 应该重新提炼测试用例，设计更多的测试用例添加到测试说明中以覆盖没有执行过的路径

理想情况下，覆盖完善阶段应该在不阅读实际代码的情况下进行。然而，实际上，为达到覆盖率目标，看一下实际代码也是需要的。覆盖完善步骤的重要程度相对小一些。最有效的测试来自于分析和说明，而不是来自于试验，依赖覆盖完善步骤补充一份不好的测试设计。

适合的技术：

- 分支测试
- 条件测试
- 设计定义——试验测试
- 状态转换测试

2.2 用例设计的一般原则

注意到前面产生测试说明的 5 个步骤可以用下面的方法完成：

- 通常应该避免依赖先前测试用例的输出，测试用例的执行序列早期发现的错误可能导致其他的错误而减少测试执行时实际测试的代码量；
- 测试用例设计过程中，包括作为试验执行这些测试用例时，常常可以在软件构建前就发现 BUG。还有可能在测试设计阶段比测试执行阶段发现更多的 BUG。
- 在整个单元测试设计中，主要的输入应该是被测单元的设计文档。在某些情况下，需要将试验实际代码作为测试设计过程的输入，测试设计者必须意识到不是在测试代码本身。从代码构建出来的测试说明只能证明代码执行代码完成的工作，而不是代码应该完成的工作。

3、测试用例设计技术

广义地分为两类：

- 黑盒测试：使用单元接口和功能描述，不需了解被测单元的内部结构
- 白盒测试：使用被测单元内部如何工作的信息
- 其他技术：不属于以上 2 类

Black box (functional)	White box (structural)	Other
Specification derived tests Equivalence partitioning Boundary value analysis State-transition testing	Branch testing Condition testing Data definition-use testing Internal boundary value testing	Error guessing

测试设计最重要的因素是经验和常识。测试设计者不应该让某种测试技术阻碍经验和常识的运用。

白盒测试用例设计：使用程序设计的控制结构导出测试用例。

采用白盒测试的目的主要是：

- 保证一个模块中的所有独立路径至少被执行一次；
- 对所有的逻辑值均需要测试真、假两个分支；
- 在上下边界及可操作范围内运行所有循环；
- 检查内部数据结构以确保其有效性。

黑盒测试用例设计：使用详细设计导出测试用例。

采用黑盒测试的目的主要是：

- 检查功能是否实现或遗漏；
- 检查人机界户是否错误；
- 数据结构或外部数据库访问错误；
- 性能等其它特性要求是否满足；
- 初始化盒终止错误。

3.1 软件设计说明导出的测试

测试用例通过根据相关的软件设计说明文档进行设计。每个测试用例测试设计说明中一项或多项陈述。通常为被测单元设计说明的一系列陈述建立一系列对应的设计用例。

例 1：考虑下面计算实数平方根的函数的设计说明：

输入：实数

输出：实数

处理：当输入0或大于0时，返回输入数的平方根；当输入小于0时，显示：“ Square root error - illegal negative input"，并返回0；库函数Print_Line用于显示出错信息。

设计说明有3个陈述，可以2个测试用例来对应。

Test Case 1：输入4，返回2。 //执行第一个陈述

Test Case 2：输入－10，返回0，显示“ Square root error - illegal negative input ”
//对应第二个和第三个陈述

设计说明导出的测试用例提供了与被测单元设计说明陈述序列很好的对应关系，增强了测试说明的可读性和可维护性。但有软件设计说明导出测试是正面的测试用例设计技术。软件设计说明导出的测试应该用负面测试用例进行补充，以提供一个完整的单元测试说明。

设计说明导出的测试设计技术还可用于安全分析、保密分析、软件冒险分析和其他给单元设计的其他补充文档。

3．2 基本路径测试

基本路径测试是一种白盒测试技术。测试用例设计者导出一个过程设计的逻辑复杂性测度，并使用改测度作为指南来定义执行路径的基本集，从该基本集导出的测试用例保证对程序中的每一条执行语句至少执行一次。

基本路径测试的方法步骤如下：

3.2.1 画出控制流图

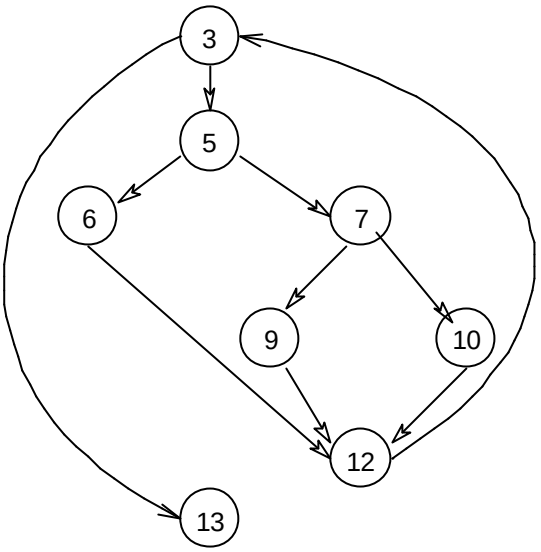
C/C++语句中的控制语句表示如下：

图中的每一个圆称为流图的节点，代表一条或多条语句。流图中的箭头称为边或连接，代表控制流。

任何过程设计都要被翻译成控制流图。如下面的 C 函数：

```
   void   Sort(int  iRecordNum,int  iType)
0 {
1     int  x=0;
2     int  y=0;
3     while  (iRecordNum--)
4      {
5         if(0= =iType)
6           x=y+2;
7         else
8            if(1=  =iType)
9                    x=y+10;
10                else
11                   x=y+20;
12        }
13 }
```

画出其对应的控制流图如下：

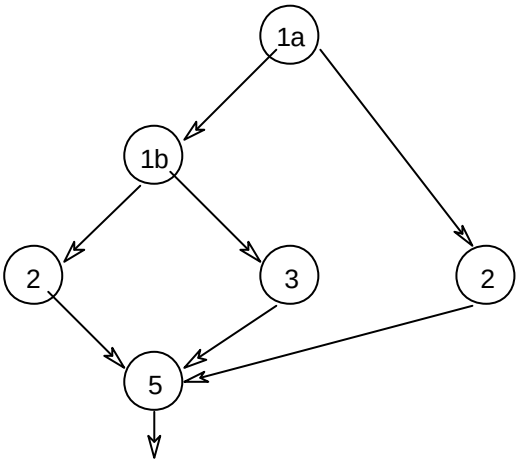


3.2.1 图一： 控制流图

注意 :如果在程序中遇到复合条件，例如条件语句中的多个布尔运算符（逻辑 OR、AND）时，为每一个条件创建一个独立的节点，包含条件的节点称为判定节点，从每一个判定节点发出两条或多条边。例如：

```
1 if    ( a or b)
2       x
3 else
4       y
5   …
```

对应的逻辑为：



3.2.2 计算圈复杂度

圈复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。独立路径必须包含一条在定义之前不曾用到的边。

有以下三种方法计算圈复杂度：

- 流图中区域的数量对应于环型的复杂性；
- 给定流图 G 的圈复杂度－V(G)，定义为 V(G)=E-N+2，E 是流图中边的数量，N 是流图中节点的数量；
- 给定流图 G 的圈复杂度－V(G)，定义为 V(G)=P+1，P 是流图 G 中判定节点的数量。

对应 3.2.1 图一中代码的圈复杂度，计算如下：

- 流图中有四个区域；
- V(G)=11 条边-9 节点+2=4;
- V(G)=3 个判定节点+1=4。

3.2.3 导出测试用例

根据上面的计算方法，可得出四个独立的路径：

路径 1：3-13

路径 2：3-5-6-12-3-13

路径 3：3-5-7-9-12-3-13

路径 4：3-5-7-10-12-3-13

根据上面的独立路径，去设计输入数据，使程序分别执行到上面四条路径。

3.2 对等区间划分

对等区间划分是一种黑盒测试方法，该方法也成为等价类划分；

对等区间划分是测试用例设计的非常形式化的方法。它将被测软件的输入输出划分成一些区间，被测软件对一个特定区间的任何值都是等价的。形成测试区间的数据不只是函数/过程的参数，也可以是软件可以访问的全局变量，系统资源等，这些变量或资源可以是以时间形式存在的数据，或以状态形式存在的输入输出序列。

对等区间划分假定位于单个区间的所有值对测试都是对等的，应该为每个区间的一个值设计一个测试用例。

考虑前面的平方根函数的测试用例区间，有 2 个输入区间和 2 个输出区间，表示如下：

输入分区		输出分区	
i	<0	a	>=0
ii	>=0	b	Error

可以用 2 个测试用例测试 4 个区间：

测试用例 1：输入 4，返回 2　　//区间 ii 和 a

测试用例 2：输入－10，返回0，输出"Square root error - illegal negative input"

　　//区间 i 和 b

上例的对等区间划分是非常简单的。当软件变得更加复杂，对等区间的确定和区间之间的相互依赖就越难，使用对等区间划分设计测试用例技术难度会增加。对 等区间划分基本上

还是移植正面测试技术，需要使用负面测试进行补充。

对等区间划分的原则：

- 如果输入条件规定了取值范围，或者值的个数，则可以确定一个有效等价类和两个无效等价类；
- 如果输入条件规定了输入值的集合，或者是规定了“必须如何”的条件，这时可以确立一个有效等价类和一个无效等价类；
- 如果输入条件是一个布尔量，则可以确立一个有效等价类和一个无效等价类；
- 如果规定了输入数据的一组值，而且程序要对每一个输入值分别进行处理，这时要对每一个规定的输入值确立一个等价类，而对于这组值之外的所有值确立一个等价类；
- 如果规定了输入数据必须遵守的规则，则可以确立一个有效等件类（即遵守规则的数据）和若干无效等价类（从不同角度违反规则的数据）；
- 如果确知以划分的等价类中的各元素在程序中的处理方式不同，则应进一步划分成更小的等价类。

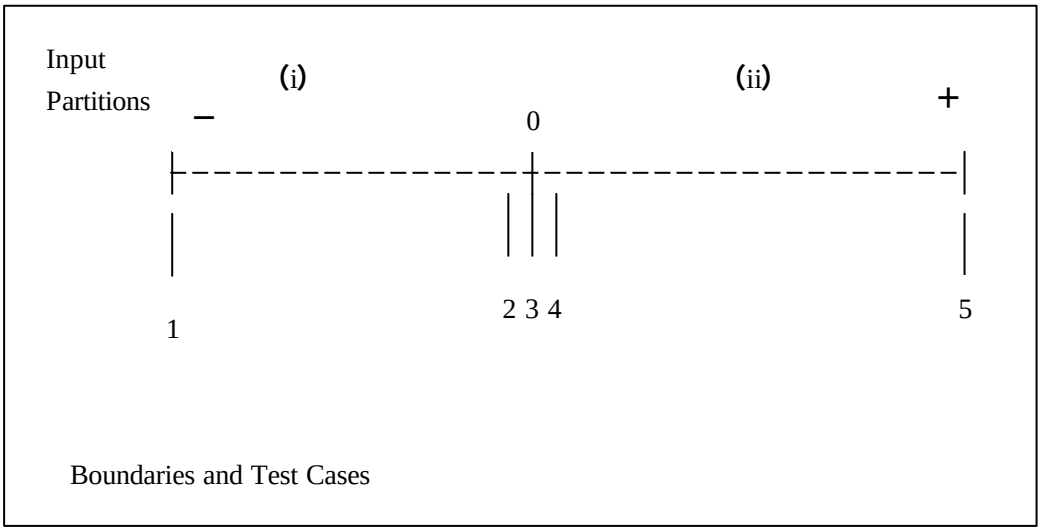
利用对等区间划分选择测试用例：

- 为每一个等价类规定一个唯一的编号；
- 设计一个新的测试用例，使其尽可能多的覆盖尚未覆盖的有效等价类；重复这一步骤，直到所有的有效等价类都被覆盖为止；
- 设计一个新的测试用例，使其仅覆盖一个无效等价类，重复这一步骤，直到所有的无效等价类都被覆盖为止。

3.3 边界值分析

边界值分析是一种黑盒测试方法。

边界值分析使用对等区间划分相同的分析。但是，边界值分析假定错误最有可能出现在区间之间的边界。边界值分析将一定程度的负面测试加入到测试设计中，期望错误会在区间边界发生，对边界值的两边都需设计测试用例。考虑平方根函数的 2 个输入区间，



0 和大于 0 区间的边界是 0 和最大实数，小于 0 区间的边界是 0 和最大负实数。输出区间的

边界是 0 和最大正实数。根据边界值分析可以设计 5 个测试用例：

Test Case 1：输入最大负实数，返回0，使用Print_Line输出“ Square root error － illegal negative input ”　　//区间（i）的下边界

Test Case 2：输入仅比0小的数，返回0，使用Print_Line输出“ Square root error － illegal negative input ”　　//区间（i）的上边界

Test Case 3：输入0，返回0

//区间（i）的上边界外，区间（ii）的下边界和区间（a）//的下边界

Test Case 4：输入仅比0大的数，返回输入的正数平方根

//区间（ii）的下边界外

Test Case 5：最大正实数，返回输入的正平方根

//区间（ii）的上边界和区间（a）的上边界

对于复杂的软件，使用对等区间划分就不太实际了，对于枚举型等非标量数据也不能使用对等区间划分。如区间（b）并没有实际的边界。边界值分析还需了解数的底层表示。一种经验方法是使用任何高于或低于边界的小值和合适的正数和负数。

选择测试用例的原则：

- 如果输入条件规定了值的范围，则应该取刚达到这个范围的边界值，以 及刚刚超过这个范围边界的值作为测试输入数据；
- 如果输入条件规定了值的个数，则用最大个数、最小个数、比最大个数多 1 格、比最小个数少 1 个的数做为测试数据；
- 根据规格说明的每一个输出条件，使用规则一；
- 根据规格说明的每一个输出条件，使用规则二；
- 如果程序的规格说明给出的输入域或输出域是有序集合（如有序表、顺序文件等），则应选取集合的第一个和最后一个元素作为测试用例；
- 如果程序用了一个内部结构，应该选取这个内部数据结构的边界值作为测试用例；
- 分析规格说明，找出其他可能的边界条件。

3.4 状态转换测试

状态转换测试对于软件被设计成一个状态机或实现了一种被建模成一种状态机的情况。可以设计测试用例测试状态间转换，测试用例创建引起转换的事件。

可以设计负面测试的测试用例用于测试状态与事件的非法组合。

3.5 分支测试

在分支测试中，测试用例用于测试单元的控制流分支或决策点。通常用于实现决策覆盖（Decision Coverage）的测试目标。如果只有模块单元的函数功能设计说明，“ 黑盒 ”形式的分支测试对分支代码进行猜测，并设计测试用例执行该分支。如果拥有模块单元的结构设计说明，在单元中指明控制流，就可以设计测试用例执行各个分支。

3.6 条件测试

Conditions Testing：测试程序模块中的所有逻辑条件，测试案例的设计策略包括：

- Branch testing：执行每个分支至少一次；
- Domain Testing：每个关系运算使用三个或四个测试；
- Branch and relational operator testing：使用条件约束，覆盖约束集。

3.7 数据定义－使用测试

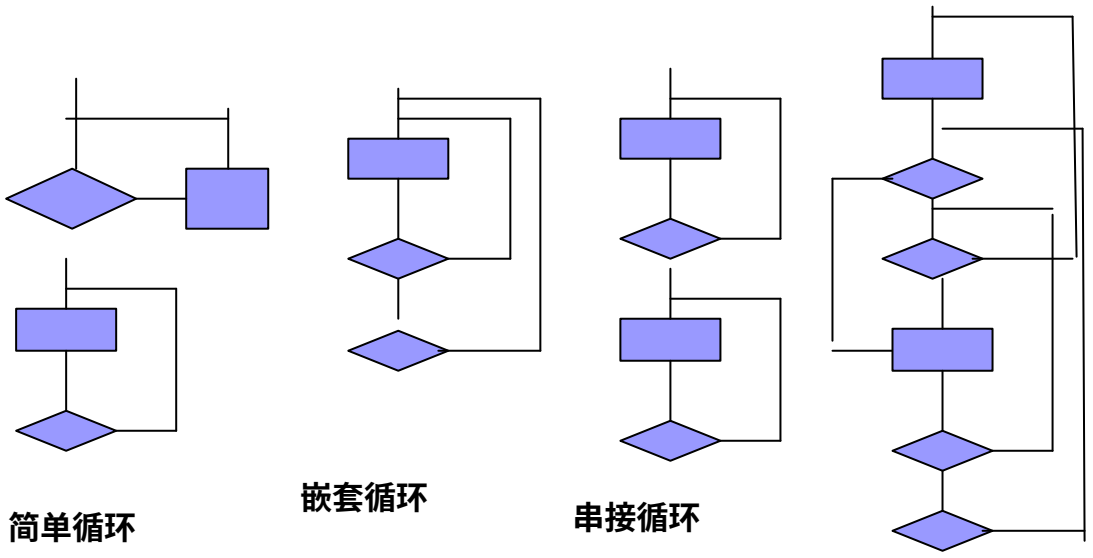
Data Flow Testing：根据变量定义和变量引用位置设置测试路径

3.8 循环测试

循环测试是一种白盒测试技术，注重于循环构造的有效性。

- 循环结构测试用例的设计

循环可以划分为以下几种模式：



可以安装如下方法设计循环测试用例：

- Simple Loops of size n:
  - Skip loop entirely
  - Only one pass through loop
  - Two passes through loop
  - m passes through loop where m<n
  - (n-1), n, and (n+1) passes through the loop
- Concatenated Loops

- If independent loops, use simple loop testing
- If dependent, treat as nested loops.

■ Nested Loops

- Start with inner loop. Set all other loops to minimum values
- Conduct simple loop testing on inner loop
- Work outwards
- Continue until all loops tested

■ Unstructured loops

- Don't test – redesign

简单循环：

下列测试集用于简单循环，其中 n 是允许通过循环的最大次数。

- 整个跳过循环；
- 只有一次通过循环；
- 两次通过循环；
- m 次通过循环，其中 m<n；
- n-1，n+1 次通过循环。

嵌套循环：

如果将简单循环的测试方法用于嵌套循环，可能的测试数就会随嵌套层数成几何级增加，这会导致不实际的测试数目，下面是一种减少测试数的方法：

- 从最内层循环开始，将其它循环设置为最小值；
- 对最内层循环使用简单循环，而使外层循环的跌代参数（即循环计数）最小，并为范围外或排除的值增加其它测试；
- 由内向外构造下几个循环的测试，但其它的外层循环为最小值，并使其它的嵌套循环为“典型”值；
- 继续直到测试所有的循环。

串接循环：

如果串接循环的循环都彼此独立，可是使用嵌套的策略测试。但是如果两个循环串接起来，而第一个循环是第二个循环的初始值，则这两个循环并不是独立的。如果循环不独立，则推荐使用的嵌套循环的方法进行测试。

无结构循环：不能测试，尽量重新设计给结构化的程序结构后再进行测试。

3.9 内部边界值分析

在多数情况下，测试用例区间的边界值可以从模块单元的功能说明中获得，可以用上述方法进行对等区间划分。在某些情况下，模块单元的内部边界只能从模块的结构说明中获得，这就需要使用内部边界值分析方法进行对等区间划分。

3.10 错误猜测

错误猜测大多基于经验，需要从边界值分析等其他技术获得帮助。这种技术猜测特定软件类型可能发生的错误类型，并且设计测试用例查出这些错误。对有经验的工程师来说，错误猜测有时是唯一最有效发现 BUG 的测试设计方法。

为了最好地利用现成的经验，可以列出一个错误类型的检查列表，帮助猜测错误可能发

生在单元中的位置，提高错误猜测的有效性。

4、面向对象的单元测试

4.1 面向对象测试的特点

自 80 年代中后期以来，面向对象软件开发技术发展迅速，获得了越来越广泛的应用，在面向对象的分析、设计技术以及面向对象的程序设计语言方面，均获得了很丰富的研究成果。与之相比，面向对象软件测试技术的研究还相对薄弱。例如，对面向对象的程序测试应当分为多少级尚未达成共识。基于结构的传统集成策略并不完全适于面向对象的程序。这是因为面向对象的程序的执行实际上是执行一个由消息连接起来的方法序列，而这个方法序列往往是由外部事件驱动的。在面向对象语言中，虽然信息隐藏和封装使得类具有较好的独立性，有利于提高软件的易测试性和保证软件的质量，但是，这些机制与继承机制和动态绑定给软件测试带来了新的课题。尤其是面向对象软件中类与类之间的集成测试和类中各个方法之间的集成测试具有特别重要的意义，与传统语言书写的软件相比，集成测试的方法和策略也应该有所不同。

从目前的研究现状来看，研究较多地集中在类和对象状态的测试方面。面向对象程序设计的继承和动态联编所带来的多态性对软件测试的影响，虽然有所论及，但是不仅缺乏针对这一特点的测试方法，而且还有许多问题有待进一步的研究。

软件测试中的另一个重要问题是测试的充分性问题，充分性准则对软件测试的揭错能力具有重要影响。对传统语言的软件测试已经存在多种充分性准则，但对面向对象的软件测试，目前尚无普遍接受的充分性准则。对这些方面的深入研究将会产生真正对软件测试的理论与实践有指导意义、有影响的成果。

对 OO 软件的类测试相当于传统软件的单元测试。和传统软件的单元测试不同，他往往关注模块的算法细节和模块接口间流动的数据，OO 软件的类测试是由封装在类中的操作和类的状态行为所驱动的。因为属性和操作是被封装的，对类之外操作的测试通常是徒劳的。封装使对对象的状态快照难于获得，继承也给测试带来了难度，即使是彻底复用的，对每个新的使用语境也需要重新测试。多重继承更增加了需要测试的语境的数量，使测试进一步复杂化。如果从超类导出的测试用例被用于相同的问题域，有可能对超类导出的测试用例集可以用于子类的测试，然而，如果子类被用于完全不同的语境，则超类的测试用例将没有多大用途，必须设计新的测试用例集。

4.2 类的功能性测试和结构性测试

在类的生命周期中，类测试只是一个初始的测试阶段。类作为独立的成分可以多次在不同的应用系统中重复使用，这些成分的用户可要求每个类是可靠的，并无须了解其实现细节。

这样的类要尽可能多地进行测试，因为我们关心的是类单元本身，而不是它所处的上下文，如类库中的 List、Stack 等基本类。

类的测试用例可以先根据其中的方法设计，然后扩展到方法之间的调用关系。如果类中的方法都已定义了前置／后置条件，则可以此作为开发对各方法进行测试所用的测试用例的参考。一般情况下，我们根据方法的前置、后置条件以及关于类的约束条件，利用一些传统的测试方法，也能设计出较完善的测试用例。

类测试一般也采用传统的两种测试方式：功能性测试和结构性测试，即黑盒测试和白盒测试。功能性测试以类的规格说明为基础，它主要检查类是否符合其规格说明的要求。例如，对于 Stack 类，即检查它的操作是否满足 LIFO 规则；结构性测试则从程序出发，它需要考虑其中的代码是否正确，同样是 Stack 类，就要检查其中代码是否动作正确且至少执行过一次。

4.2.1 功能性测试

功能性测试包括两个层次：类的规格说明和方法的规格说明。

(1) 类的规格说明：类的规格说明是各方法规格说明的组合及对类所表示概念的广义描述。例如，Stack 类的规格说明中包括了方法 push 和 pop 的规格说明，但 push 和 pop 中都没有说明这两个操作在一个类中同时工作的情况，即 push 的规格说明只要求把其参数值加入到栈顶上，但对删除不加任何说明，而 pop 也同样不对被删除项的加入作任何描述，仅在类这一层的规格描述中才表达 LIFO 的要求限制。对于数据类型的形式化描述也可以用来对类进行定义，但类比类型的含义更广泛，具有更确切的语义，尤其是类之间的继承关系也被表示出来了。

一个 C＋＋类的规格说明具有多层性，但对它的用户说来，它只包括了在类定义公共区中方法的说明，子类所能见到的父类是其 public 和 protected 区域中的内容，一个类中所定义的方法可分为三个存取层次：public，protected 和 private。这些方法可以各自分开独立考虑，一个类是所有这些的综合。

(2) 方法的规格说明：每个独立方法的规格说明可以用其前置／后置条件描述。根据前置条件选择相应的测试用例，就可以检查其产生的输出是否满足后置条件而完成对独立方法的测试，对独立方法的测试与对独立过程的测试方法类似。

4.2.2 结构性测试

结构性测试对类中的方法进行测试，它把类作为一个单元来进行测试。测试分为两层：第一层考虑类中各独立方法的代码；第二层考虑方法之间的相互作用。每个方法的测试要求能针对其所有的输入情况，但这样还不够，只有对这些方法之间的界面也做同样测试，才能认为测试是完整的。对于一个类的测试要保证类在其状态的代表集上能够正确工作，构造函数的参数选择以及消息序列的选择都要满足这一准则。因此，在这两个不同的测试层次上应分别做到：

(1) 方法的单独测试：结构性测试的第一层是考虑各独立的方法，这可以与过程的测试采用同样的方法，两者之间最大的差别在于方法改变了它所在实例的状态，这就要取得隐藏的状态信息来估算测试的结果，传给其它对象的消息被忽略，而以桩来代替，并根据所传的消息返回相应的值，测试数据要求能完全覆盖类中代码，可以用传统的测试技术来获取。

(2) 方法的综合测试：第二层要考虑一个方法调用本对象类中的其它方法和从一个类向其它类发送信息的情况。单独测试一个方法时，只考虑其本身执行的情况。而没有考虑动

作的顺序问题，测试用例中加入了激发这些调用的信息，以检查它们是否正确运行了。对于同一类中方法之间的调用，一般只需要极少甚至不用附加数据，因为方法都是对类进行存取，故这一类测试的准则是要求遍历类的所有主要状态。

4.3 基于对象—状态转移图的面向对象软件测试

面向对象设计方法通常采用状态转移图建立对象的动态行为模型。状态转移图用于刻画对象响应各种事件时状态发生转移的情况，图中结点表示对象的某个可能状态，结点之间的有向边通常用“事件／动作”标出。

如图 4.3 的示例中，表示当对象处于状态 A 时，若接收到事件 e 则执行相应的操作 a 且转移到状态 B。因此，对象的状态随各种外来事件发生怎样的变化，是考察对象行为的一个重要方面。

图 4.3 对象-状态转换图

基于状态的测试是通过检查对象的状态在执行某个方法后是否会转移到预期状态的一种测试技术。使用该技术能够检验类中的方法能否正确地交互，即类中的方法是否能通过对象的状态正确地通信。因为对象的状态是通过对象的数据成员的值反映出来，所以检查对象的状态实际上就是跟踪监视对象数据成员的值的变化。如果某个方法执行后对象的状态未能按预期的方式改变，则说明该方法含有错误。

状态转移图中的结点代表对象的逻辑状态，而非所有可能的实际状态。

理论上讲，对象的状态空间是对象所有数据成员定义域的笛卡尔乘积。当对象含有多个数据成员时，要把对象所有的可能状态进行测试是不现实的，这就需要对对象的状态空间进行简化，同时又不失对数据成员取值的“覆盖面”。简化对象状态空间的基本思想类似于黑盒测试中常用的等分区间的方法。依据软件设计规范或分析程序源代码，可以从对象数据成员的取值域中找到一些特殊值和一般性的区间。特殊值是设计规范里说明有特殊意义，在程序源代码里逻辑上需特殊处理的取值。位于一般性区间中的值不需要区别各个值的差别，在逻辑上以同样方式处理。例如下面的类定义：

```
class Account{
    char *name;
    int  accNum;
    int  balance;
}
```

依据常识可知，特殊值情况：name=NULL，accNum=0，balance=0；一般区间内：name！＝NULL，accNum<0 或 accNum>0，balance<0 或 balance>0。

进行基于状态的测试时，首先要对被测试的类进行扩充定义，即增加一些用于设置和检查对象状态的方法。通常是对每一个数据成员设置一个改变其取值的方法。另一项重要工作是编写作为主控的测试驱动程序，如果被测试的对象在执行某个方法时还要调用其他对象的方法，则需编写桩程序代替其他对象的方法。测试过程为：首先生成对象，接着向对象发送消息把对象状态设置到测试实例指定的状态，再发送消息调用对象的方法，最后检查对象的状态是否按预期的方式发生变化。

下面给出基于状态测试的主要步骤：

- 依据设计文档，或者通过分析对象数据成员的取值情况空间，得到被测试类的状态转移图；

- 给被测试的类加入用于设置和检查对象状态的新方法,导出对象的逻辑状态；
- 对于状态转移图中的每个状态,确定该状态是哪些方法的合法起始状态,即在该状态时,对象允许执行哪些操作；
- 在每个状态，从类中方法的调用关系图最下层开始，逐一测试类中的方法；
- 测试每个方法时，根据对象当前状态确定出对方法的执行路径有特殊影响的参数值，将各种可能组合作为参数进行测试。

4.4 类的数据流测试

数据流测试是一种白盒测试方法，它利用程序的数据流之间的关系来指导测试的选择。现有的数据流测试技术能够用于类的单个方法测试及类中通过消息相互作用的方法的测试中,但这些技术没有考虑到当类的用户以随机的顺序激发一系列的方法时而引起的数据流交互关系。为了解决这个问题，我们提出了一个新的数据流测试方法，这个方法支持各种类的数据流交互关系。对于类中的单个方法及类中相互作用的方法，我们的方法类似于一般的数据流测试方法；对于可以从外部访问类的方法，以及以任何顺序调用类时，我们计算数据流信息，并利用它来测试这些方法之间可能的交互关系。这个方法的最大好处是我们可以利用数据流测试方法来测试整个类,且这个技术对于在测试类时决定哪一系列的方法应该测试时非常有用，另外,象其它的基于代码的测试技术一样，这种技术的大部分都可以实现自动化。

4.4.1 数据流分析

当数据流测试用于单个过程的单元测试时，定义-引用对可利用传统的迭代的数据流分析方法来计算，这种方法利用一个控制流图(control flow graph)来表示程序，其中的节点表示程序语句,边表示不同语句的控制流,且每一个控制流图都加上了一个入口和一个出口。为了将数据流测试技术应用到交互式过程中，需要有更精确的计算。过程间数据流分析(interprocedural dataflow analysis)可以计算定义在一个过程中，而引用又在另一个过程中的定义-引用对，这种技术可以计算全局变量的定义-引用对，另外它在计算定义-引用对时还考虑指针变量及别名的影响。

利用上面的算法为程序建立一个过程间控制流图,它把单个的过程和控制流图结合在一起，并把每一个调用点用一个调用节点和一个返回节点代替，通过加入从调用节点到输入节点的边及从输出节点到返回节点的边表示过程的调用，从而把整个控制流图联系在一起。

4.4.2 类及类测试

类是个独立的程序单位，它应该有一个类名并包括属性说明和服务说明两个主要部分，对象是类的一个实例。不失一般性，我们这里构造一个类的模型，它只包括两种访问类变量及方法的方式：public 和 private；类在实例化时一般要执行构造方法，在完成时要析构。如图 4．2 所示一个队列 Queue 的例子，它包括公开的方法如构造函数 Queue 和析构函数～Queue，AddtoQueue，GetfromQueue，DeletefromQueue，私有的方法如 Lookup，Hash 等。

```
1   #include “queue.h”
2
3   class Queue {
4       private
```

```
5             int  *queue;
6             int   numentries, queuemax;
7             int   *Lookup (int,int) ;
8       public:
9             Queue (int n) {
10            queuemax = n;
11            numentries = 0;
12            queue = new int [queuemax]; };
13             ~Queue() { delete queue;  } ;
14            int AddtoQueue (int key);
15            int GetfromQueue (int key);
16    }
17
18    int Queue :: Lookup (int key, int index){
19       int saveindex;
20       int Hash (int);
21       saveindex = index = Hash(key);
22       while ( queue[ index] != key){
23             index ++;
24             if(index == queuemax )
25                index = 0;
26            if( queue[ index] == 0 || index == saveindex )
27                return FALSE;
28           }
29           return TRUE;
30     }
31
32  int Queue :: AddtoQueue (int key) {
33     int index;
34     if( numentries < queuemax ) {
35        if ( Lookup (key, index)==TRUE)
36            return NOTOK;
37        AddKey (key, index);
38        numentries ++;
39        return OK;
40     }
41     return NOTOK;
42  }
43
44   int Queue :: GetfromQueue (int key) {
45     int index;
46     if ( Lookup ( key, index) == FALSE)
47          return NOTOK;
48     return OK;
```

```
49    }
50
51   void Queue :: DeletefromQueue (int key) {
52      int index;
53      if ( Lookup ( key, index)==FALSE)
54            return NOTOK;
55      queue[key] = 0;
56   }
```

图 4.4.2 一： 类 Queue 的部分代码

我们用类调用图（class call graph）来表示类的调用结构，在图中，结点表示方法，边表示方法间的过程调用，图 4.4.4.2 二 为类 Queue 的类调用图，AddtoQueue 调用 Lookup，则有一条边由 AddtoQueue 指向 Loopup；图中还有一些虚线，它表示从类外部发给这些公开方法的消息。

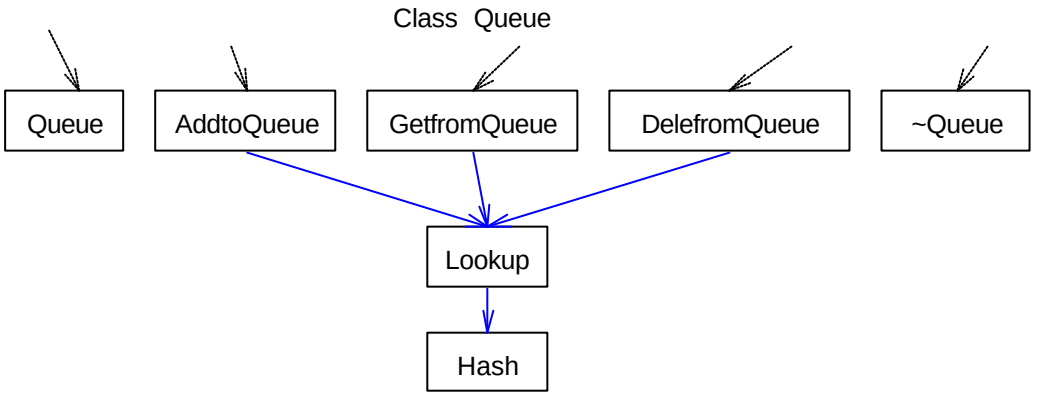


图 4.4.2 二：Queue 的类调用图

我们对类进行三级测试，定义如下：

方法内部测试(Intra-method testing)：测试单个方法，这级测试相当于单元测试；

方法间测试(Inter-method testing)：在类中与其它方法一起测试一个直接或间接调用的公开方法，这级测试相当于集成测试；

类内部测试(Intra-class testing)：测试公开方法在各种调用顺序时的相互作用关系，由于类的调用能够激发一系列不同顺序的方法，我们可以用类内部测试来确定类的相互作用关系顺序，但由于公开方法的调用顺序是无限的，我们只能测试其中一个子集。

为了说明这些级别的测试，我们结合图 4.4.2 二 所示的 Queue 类来进行描述。我们在类 Queue 进行方法内部测试来分别地测试每一个方法(共有 7 个)，在对 AddtoQueue 进行方法间测试时则要把 Addtoqueue、Lookup，Hash 方法集成起来。类似地，对 GeffromQueue 进行方法间测试时则要把 GetfromQueue、Lookup，Hash 方法集成起来。由于 Queue 和~Queue 方法不调用其它的方法，则只要测试它本身即可。对于类内部测试，我们可以选择这样的测试序列：<Queue：AddtoQueue，GetfromQueue>或<Queue：AddtoQueue，AddtoQueue>，在任何一种情况下，我们需要有桩模块和驱动模块来执行测试。

4.4.3 数据流测试

为了支持现有的类内部测试（Intra-class testing）技术，我们需要一个基于代码的测试技术来识别需要测试的类的部件，这种技术就是数据流测试，它考虑所有的类变量及程序点说明的定义-引用对（def－use pairs)。在类中共有三种定义-引用对需要测试，这三种类型分别与前一节所定义的相对应，在下面的定义中，设 C 为需要测试的类，d 为表示一个包含定义(definition)的状态，u 为包含引用(use)的状态。

方法内部定义-引用对(Intra-method def-use pairs)：设 M 为类 C 中的一个方法，如果 d 和 u 都在 M 中，且存在一个程序 P 调用 M，则在 P 中当 M 激发时，(d，u)为一个引用对，那么(d，u)为一个方法内部定义-引用对。

方法间定义-引用对(Inter-method def-use pairs)：当 Mo 被激发时，设 Mo 为 C 中的一个公开方法，{M1，M2，…，Mn}为 C 直接或间接调用的方法集。设 d 在 Mi 中，u 在 Mj 中，且 Mi、Mj 都在{M1，M2，…，Mn}中，如果存在一个程序 P 调用 Mo，则在 P 中当 Mo 激发且 Mi≠Mj 或 Mi、Mj 被同一个方法分别激发时，(d，u)为一个引用对，那么(d，u)为一个方法间定义-引用对。

类内部定义-引用对(Intra-class def-use pairs)：当 Mo 被激发时，设 Mo 为 C 中的一个公开方法，{M1，M2，…，Mn}为 C 直接或间接调用的方法集。当 No 被激发时，设 No 为 C 中的一个公开方法(可能与方法 Mo 相同)，{N1，N2，…，Nn}为 C 直接或间接调用的方法集，设 d 在{M1，M2，…，Mn}的某个方法中，u 在{N1，N2，…，Nn}中，如果存在一个程序 P 调用 Mo 和 No，且在 P 中(d，u)为一个引用对，并且在 d 执行之后，u 执行之前，Mo 的调用就中止了，那么(d，u)为一个类内部定义-引用对。

一般来说，方法内部定义-引用对出现在单个的方法中，且测试定义-引用对的相互作用时也限于这些方法中。例如，在 Queue 类中，Lookup 方法中包含一个关于 index 的方法内部定义-引用对(23，24)，即变量 index 在 23 行中定义，index 的引用则在 24 行中。

方法间定义-引用对出现单个公开方法被调用后方法之间相互作用之中，它们定义出现在一个方法中，引用则出现在通过公开方法直接或间接调用这个方法的另一个方法中。例如在类 Queue 中，公开方法 AddtoQueue 调用 Lookup 方法，接收 index 的值并使用在 AddKey 方法中，定义引用对(25，37)是一个方法间定义-引用对，即 index 的定义出现在方法 Lookup 中(25 行)而 index 的使用出现在方法 AddtoQueue 中(37 行)。

类内部定义-引用对出现在一系列公开方法被激发时。例如，在方法序列<AddtoQueue，AddtoQueue>，在第一次调用 AddtoQueue 时，如果有一个数据加入到队列中，则第 38 行 numentries 的值被设置，在第二次调用 AddtoQueue 时，第 34 行取得 numentries 的值，那么(38，34)是一个类内部定义-引用对。

上面所提及的三种定义-引用对对于类的测试是非常有用的。例如，当我们使用“all-uses"数据流覆盖准则时，方法内部定义-引用对(23，24)测试 Lookup 方法当 index 指向队列的末尾时是否将 index 重新指向队列的开始。方法间定义-引用对(25，37)测试程序是否能将一个数据加入到表的第 0 个位置，类内部定义-引用对(38，34)测试当队列满时是否能正确地处理。

另外，类内部定义-引用对还可以指导测试者选择合适的方法序列，判断某些方法序列是否需要执行。下面我们举一个例子来进一步说明类内部定义-引用对的优点，设类 Queue 中第 34 行改为如下的语句：

```
if(numentries<=queuemax)
```

假设有 queuemax+1 个不同的数据，在插入 queuemax 个数据后再调用 AddtoQueue 方法，在这次调用中，numentries 的值为 queuemax，符合判断语句的条件，因此调用 Lookup 方法，

由于各个数据都不同，且队列中已没有空的地址可存放，则执行到第 26 行由于满足后一个条件而返回 FALSE，AddtoQueue 则将这个数据加入到队列中去，将当前位置的数据覆盖掉。要检查出这个错误，我们必须对 AddtoQueue 调用 queuemax+l 次。

4.4.4 计算类的数据流信息

为了支持类的数据流测试，我们必须计算类的各种定义-引用对。前面描述的算法对于计算方法内部及方法间的定义-引用对是有用的，但由于它需要一个完整的程序来构造一个控制流图，因此不能直接用于计算类内部定义-引用对。为了计算类内部定义-引用对，我们必须考虑当一系列的公开方法被调用时的相互作用。可以考虑建立一个图来描述这些相互作用，然后用类似的算法来计算它。

为了计算类的三种定义-引用对，我们可以构造一个类控制流图(class control flow graph—CCFG)，其算法如下：

1. 为类构造类调用图，作为类控制流图的初值；
2. 把框架(frame)加入到类调用图中；
3. 根据相应的控制流图替换类调用图中的每一个调用节点，具体实现方法：对于类 C 中的每一个方法 M，在类调用图中用方法 M 的控制流图替代方法 M 的调用结点，并更新相应的边；
4. 用调用节点和返回节点替换调用点，具体实现方法：对于类调用图中的每一个表示类 C 中调用方法 M 的调用点 S，用一个调用节点和返回节点代替调用点 S；
5. 把单个的控制流图连接起来，具体实现方法：对于类控制流图中的每一个方法 M，加上一条从框架调用节点到输入节点的边和一条从输出节点到框架返回节点的边，其中输入节点和输出节点都在方法 M 的控制流图中；
6. 返回完整的类控制流图。

第一步，我们为类 C 构造一个类调用图，并且把它作为类控制流图的原形。

第二步，我们用一个框架把类调用图包围起来，这个框架有助于数据流分析，它是类的一个驱动模块，可以让我们模拟调用公开方法的随机序列。一个框架包含五个节点：框架输入(frams entry)和框架输出(frame exit)，它们分别表示从框架输入和输出，框架循环(frame loop)，它促进方法的序列化，框架调用(frame call)和框架返回(frame return)，它们分别表示从任何公开方法进行的调用和返回。框架还包括四条边：(框架输入，框架循环)、(框架循环，框架调用)、(框架循环，框架输出)和(框架返回，框架循环)。

图 4.4.4 为类 Queue 包含框架的类调用图，在此图中，框架节点用虚线框表示。此时构造的类控制流图中框架和类调用图并没有联系。

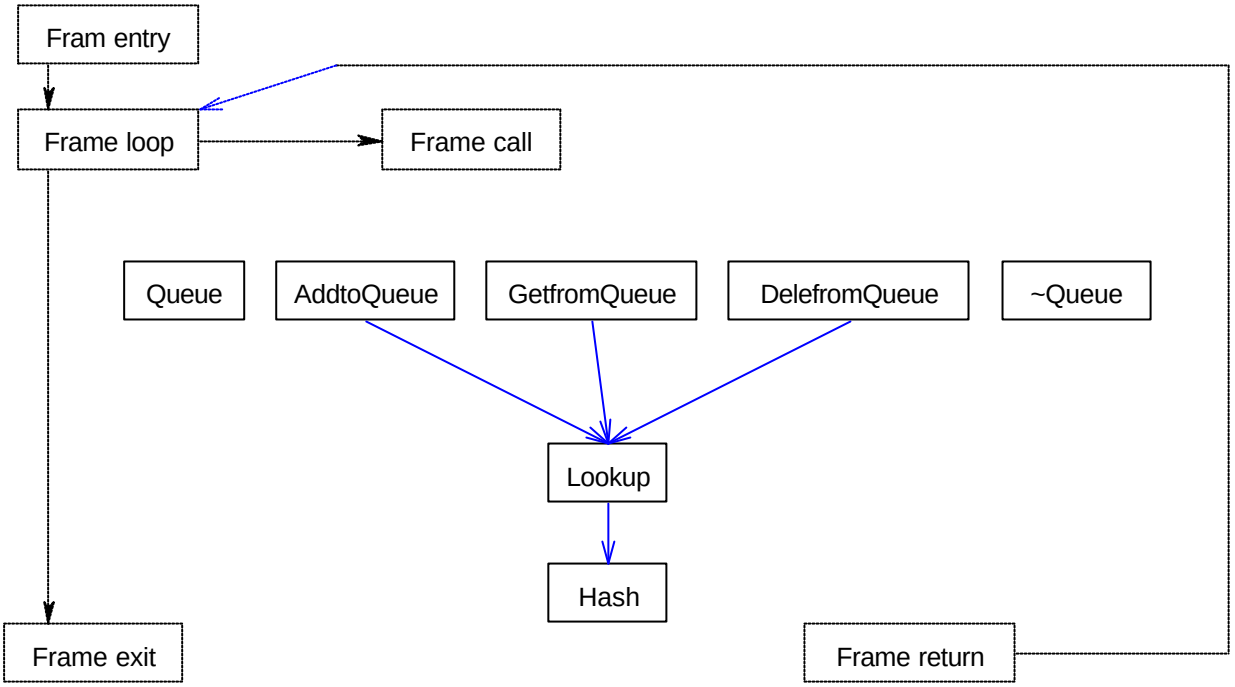


图 4.4.4　Queue 包含框架的类调用图

我们介绍了类的数据流测试技术，定义了三种数据流测试：1)方法内部测试，用于测试单个类方法；2)方法间测试，用于测试一个类中不同方法通过过程调用的相互作用：3)类内部测试，用于测试一系列方法的调用。为了区分这三种测试的定义-引用对，我们把类作为一个单输入、单输出的程序，并为这个程序构造一个类控制流图。

5 编后语

软件测试是一个计算机发展中出现的一门崭新的学科，虽然目前有很多的测试技术和测试方法，但还没有完全形成一个统一的行业标准，有很多的测试方法很难在实践工程中进行应用，很多测试工具也不能很好的与实际工作结合，所以说，在具体的测试活动中还需要根据工程本身的特点，灵活的选择测试方法和测试工具，实践与理论相结合才能取得时间、资源等的合理投入和最大产品效益的获得。