软件测试研究①

薛冲冲，陈　坚

(西南大学 计算机与信息科学学院，重庆 400715)

摘　要：软件测试是保证软件质量和提高软件可靠性的重要手段。随着软件程序量和复杂度不断地增加，人们对软件质量的要求也在不断提高，软件测试在软件开发过程中占据的位置也变的越来越重要了，同时软件测试的工作量也显得越加艰巨。系统介绍了软件测试的概述包括软件测试的定义、阶段、过程模型和目的，并分析总结了软件测试的策略、方法和自动化。

关键词：软件测试软；件测试的策略；测试用例设计方法；软件测试自动化

Software Testing

XUE Chong-Chong, CHEN Jian

(College of Computer and Information Science & Software, Southwest University, Chongqing 400715, China)

Abstract: Software testing is an important means to ensure quality of software and to improve reliability of software. But with the volume and complexity of software continues to increase, people constantly improve the software quality requirements. Software testing occupies an more and more important position in software development. At the same time, software testing is becoming more and more difficult. This article introduces the outline of software testing including the definition, phase, process model and purpose of software testing, then analyses and summarizes the strategies, methods and automation.

Keywords: software testing; software testing strategies; test case design means; software test automation.

1　引言

随着软件规模和复杂度不断地增加，软件中的错误使得软件开发在成本、进度和质量上失控的可能性增大，导致了软件危机的出现。而软件测试出现可以使程序中的错误密度达到尽可能低的程度，是保证软件质量和提高软件可靠性的重要手段，是解决软件危机的重要途径。在这样的背景下，软件测试越来越受到软件领域的关注。

2　软件测试概述

2.1 软件测试的定义

1983 年 IEEE 提出的软件工程标准术语中给软件测试下的定义是:“使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别”[1]。软件测试是为了发现软件中的错误而执行程序的过程；是为了证明软件中存在错误，而不是证明没有错误；是为了证明软件能够满足用户的需求。

2.2 软件测试的阶段及过程模型

软件测试从工程阶段的角度来看，可以分为组件测试(或是单元测试)、系统测试、接收测试等三个阶段。如图 1 所示软件测试的阶段图。

图 1　软件测试的阶段图

2.2.1 组件测试阶段

又称为单元测试，组件测试就是对单独的组件或者复合组件进行测试以确保其操作的正确性。组件是

① 收稿时间:2010-05-17;收到修改稿时间:2010-06-27

在不受其他组件或系统的影响下进行的。组件测试阶段的目的是通过对单个程序组件如(函数、对象或是复合组件)的测试发现逻辑结构上的错误，在这个测试阶段多用到是结构化测试即白盒测试。

2.2.2 系统测试阶段

系统测试就要将组件集成为系统，进行系统测试。系统测试与组件测试相比更关注找出组件间非预期的交互行为和组件接口问题，同时也要测试系统是否在功能上和非功能上能满足需求，并在此后会对系统进行总体特征的测试[1]。系统测试阶段的目的是确定系统能满足其功能性和非功能性需求上，以及系统不会以未预料到的方式执行。

2.2.3 接收测试阶段

接收测试就是让软件接受来自用户的真实数据的测试，因为真实数据会以不同的方式来测试系统，能暴露出系统需求定义中的错误和遗漏。接收测试阶段的目的是通过用户提供的真实数据测试系统，暴露出系统需求定义中得错误和遗漏。

2.2.4 软件测试过程模型

对于不同阶段的测试，测试过程大致是相似的。首先根据各个阶段的特征选择和设计测试用例，然后就是由测试用例导出测试的数据，接着就是将这些数据作为输入运行程序，将通过测试得到的结果与事先预测的结果进行比较，最后就是将两者的比较结果写成测试报告，开发者根据测试报告再决定对软件如何处理。如图 2 所示软件测试过程模型图。

图 2 软件测试过程模型图

2.3 软件测试的目的

根据软件测试各个阶段的目的可以对软件测试的目的解释如下[2]：

(1) 向开发者和用户展示软件满足了它的需求。同时,这个目标导致了有效性测试即测试系统能够满足用户对它的需求。

(2) 为了找出软件中的缺陷和不足,即软件的活动是不正确的、所不希望的或是不符合它的描述的。同时，这一目标也导致了缺陷测试即使用测试用例来暴露系统中的缺陷。

3 软件测试的策略

由软件测试的三个阶段可以得出，软件测试的策略应该也包括三个步骤进行，即组件测试(或是单元测试)、系统测试(包括集成测试和性能测试)、接收测试。图 3 所示为软件测试的四个步骤。

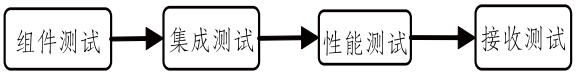

图 3 软件测试的策略

3.1 组件测试

组件测试又称单元测试，主要是发现组件内部逻辑和数据结构的错误。组件测试就是对单个组件的测试过程，从程序的内部结构出发设计测试用例。组件测试也是一个缺陷测试的过程，对组件测试大多数使用结构化测试即白盒测试。因为组件是软件中最小的单元，所以可以对多个组件进行组件测试。

3.1.1 组件测试的类型

在组件测试阶段，需要对不同类型的组件测试。1)对象内的单个函数或方法，2)有多个属性和方法的对象类，3)有不同对象或是函数组成的复合组件。从测试的类型可以看到，对单个函数或方法得测试是要容易些的，但当对多个组件组成的复合组件进行测试时，我们首先要关心得就是测试组件间的接口是否能正确执行，对复合组件的接口测试是这个测试中的重头戏。

3.1.2 接口测试

组件测试中最常用的一种测试即接口测试。接口测试就是对复合组件间的接口进行测试。因为复合组件之间存在了大量的、复杂的交互行为。在我们访问复合组件中的单个组件时，就要通过接口来调用他们。所以对接口的测试也就成为组件测试中的重中之重。

接口错误是复杂系统中最常见的错误形式之一，其包括接口误用、接口误解、计时错误等。因为接口的缺陷往往出现在不寻常的条件下、交互双方都有缺陷的情况下所以很难被发现，所以进行接口测试是十分有必要的。

组件的开发和测试是交叉进行的，因为组件的设计者对组件是十分熟悉的，所以组件的测试有其设计者进行是很自然的事了。

3.2 集成测试

集成测试将组件集成为系统进行的测试。集成测

试是检查组件在一起确实工作，组件与组件间的交互能够正确及时进行。集成测试就是发现与接口相关的错误，以保证在添加新的功能模块没有传播不期望得副作用，保证由新模块的添加引起的变更不引入需求外的行为或是增加额外的错误[3]。

3.2.1 集成系统的策略

不同的集成顺序对集成测试也将会产生不同影响。集成系统的策略理论上分为自上而下的集成和自底而上的集成。但在实际的情况下，对很多的系统来说，集成的策略是混合型的。自上而下的集成就是先开发出系统的框架，然后再将组件添加进去。最普遍的问题就是在测试下层时还要对上层进行测试。自底而上得集成就是从组件开始集成和测试。无论使用那一种策略，都需要开发额外的代码来仿真其他组件从而是系统运行起来。

集成测试中的主要问题是对错误的定位。因为组件之间又存在着复杂的交互行为，所以当我们发现一个错误时就很难准确的找到出错的位置。

3.2.2 集成测试的方法

为了解决集成测试中的主要问题。集成测试通常采用增量法来进行测试。增量法的基本思路就是开始时我们先组建个最小的系统并测试它；然后再将其他功能模块逐一的添加到这个已经测试了的系统上来；最后再对新组建的系统进行测试。这种方法易于分离和纠正错误，更易于对接口进行彻底测试。显而易见，在测试的过程中会出现回归测试即重复已经测试过的测试。

3.2.3 回归测试

回归测试就是重复已经测试过的测试。可以保证新增加块后不会给原来的系统组件间的行为产生不期望的变更。为了保障软件的正确运行回归测试是必不可少的，对测试过的要重新测试又会产生不必要的浪费。

要使得回归测试即减少对系统资源的浪费，又变得更有效。

(1) 引进自动化测试。使得测试可以自动得重复运行，减少回归测试对系统资源的浪费。

(2) 对增加块增加顺序提出要求，先增加最常用的功能块。使得最常用的组件得到最多的测试。确保常用功能的在各种情况下都能实现。

(3) 采用混合的集成策略，先采用自底向上得集成，等到距离顶层两三个层次时，采用自上而下的集成。

集成测试包括许多程序员的工作集合，这时的测试如果再交给组件的设计者显然是不合适的，这就要求有一个独立的测试团队按照先前拟定的测试计划来进行测试。

3.3 性能测试

性能测试就是测试系统对在非正常条件下的运行情况。它的任务是验证软件的功能和性能及其他特性是否达到了需求规格书上的要求。一般情况下包括恢复测试、安全测试、压力测试、性能测试。恢复测试是通过各种方式强制地让系统发现故障并验证其能够重新恢复的一种系统测试。安全测试是验证在系统内的保护机制是否能够实际保护系统不受非法入侵。压力测试是以一种要求反常数量、频率或容量的方式执行系统的测试方法[3]。

3.4 接收测试

接收测试是对将要分给客户的系统版本的测试过程。通常是一个黑盒测试过程，将软件与计算机的其他系统元素结合在一起，在实际运行环境下，由用户通过真实的数据测试对软件在内的计算机系统，以求能发现系统需求定义中得错误和遗漏。同时也期望能发现因系统的设施不能满足用户的需求或系统的性能无法接受的错误[2]。

3.4.1 接收测试的方法

接收测试的最好测试方法是基于脚本的测试，先设计出多个脚本然后再从脚本中开发错测试用例。一个完全的脚本测试必需要考虑到异常情况，保证对象能够正确处理异常。但是脚本的维护工作量大。且脚本的复用对实际运行环境有特别的要求。

3.4.2 接收测试的分类

接收测试根据用户的不同可以分为 α 测试和 β 测试。无论是 α 测试还是 β 测试都是在实际的使用环境下进行的。 α 测试是测试为特定的用户开发的系统，且由最终用户在开发者在场的情况下进行的，开发者在旁边记录用户所遇到的错误和使用问题，a 测试要持续进行直到开发者和用户双方都对最后交付的系统满意。 β 测试的对象是一个将要在市场上销售的软件产品， β 测试是所有愿意使用该软件的用户在使用该软件是所遇到的所有问题，然后在反馈给开发者，开发者再根据反馈回来的信息决定对

软件做如何处理。通常情况下 β 测试能暴露开发者无法预见得错误。

4 软件测试的测试用例设计方法

测试用例将指导测试工作，可以有效而快捷的测试软件系统，所以测试用例的设计一直是软件测试的热点和重点。测试用例设计是系统和组件测试的一部分。测试用例设计的目的就是建立一组用例集合。这些用例有两个用途，分别对应软件测试的两个目的即有效的发现程序中的缺陷和证明系统能够满足用户的需求。设计测试用例的方法主要有三种：

4.1 基于需求的测试

基于需求的测试是对由多个组件组成的系统或子系统的测试；是一种系统的测试用例设计方法。基于需求的测试就是要测试系统的需求能否正确的被提供，所以它又是一种有效性的测试而不是缺陷测试。重点关注两个关键问题：第一：验证需求是否正确、完整、无二义性并且逻辑一致。这就要求在软件开发阶段时要严格按照软件需求规格书进行。就可以在一定程度上减少错误的出现。第二：要从“黑盒”的角度设计出充分并且必要的测试集合，以保证设计和代码都符合需求[4]。测试用例也就是为测试系统而设计的，在系统测试阶段最常用到。

4.1.1 黑盒测试

基于需求的测试最常用也是最重要的测试用例设计方法是黑盒测试。黑盒测试是指在软件接口处的执行测试，检查系统的基本方面。很少去关心软件的内部结构。黑盒测试一般应用于系统测试，他把系统封装成“看不见的黑盒”，看不到系统内部逻辑结构，测试时也不需要知道内部逻辑结构和特征，根据系统的需求说明书来确定系统的输入测试用例，再通过测试结果和预期的结果相比较来判断软件的功能是否实现。黑盒测试的好处是：完全不需要去考虑系统内部的逻辑结果和特征，只以用户的观点出发测试系统。黑盒测试的缺点有不可能覆盖所有的代码，覆盖率较低；自动化测试的复用性较低。

4.1.2 黑盒测试的方法

常用的黑盒测试方法又分为等价类划分、边界值分析、错误推测和因果图等方法其中等价划分法和边界值分析是最常用的[5]。

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等价类划分的理想测试用例是可以单独的发现一类错误[1]。

边界值分析法是将测试用例选择为边界值，可以查出更多的错误。边界值分析法的基本思想是在最大值，最小值，接近最大值、接近最小值作为测试用例。

4.2 结构化测试

结构化测试(白盒测试)是一种测试用例设计方法。白盒测试是基于过程细节的封闭检查。要通过对软件内部逻辑结构和特征来设计测试用例。进行白盒测试就必须要知道系统内部的结构和动作。白盒测试主要应用于组件测试阶段，在集成测试阶段也会用到但需要自动化测试工具来支持，否则起工作量是惊人的。白盒测试的直接好处就是知道所设计的测试用例在代码级上哪些地方被忽略掉；帮助软件测试人员增大代码的覆盖率，提高代码的质量，发现代码中隐藏的问题。缺点就是工作量大。白盒测试是对路径的测试，所以采用覆盖准则来设计测试用例。常用的方法有语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖、路径覆盖[5]。

路径测试是一种结构化测试策略。基本路径测试是由 Tom McCabc 首先提出的一种白盒测试技术，使用该方法生成的测试用例，可以保证程序中的每一条语句至少被执行一次。路径测试的流程是：根据软件的结构图画出相应的流图；再根据流图确定出程序的环复杂度；然后在确定出独立路径的基本集会；最后是设计测试用例，强制执行基本集会中的每条路径[3]。

4.3 划分测试

划分测试就是将具有共同特征的一组数据作为一个划分，设计测试用例使系统对每个输入划分中的数据，生产相对应的输出划分。划分的基本思想就是如何能有效的识别出划分，一般使用程序描述或用户文档来识别划分，也可以根据经验知识来进行划分。划分测试是缺陷测试，是为了能找出组件或系统的错误[2]。划分测试要求对每个划分中的数据，程序的行为应该是相同的。基于系统和组件的所有划分的识别是测试用例设计的一个系统方法，只有对每个划分进行有效的识别后，程序运行时才能将输出的数据落

到相应的划分里。划分测试不仅要能识别出现有的各个划分中的数据，也要能识别出不在划分中的数据，他们是用来判断程序能否对无效的输入进行正确处理[3]。

5 软件测试自动化

用户对软件的需求，随着使用环境的变化，而对软件产生新的需求。这就要求软件总是在不断变化的。当开发人员将新的功能模块添加到原来的系统上后，测试人员不仅要测试新增加的模块，还要重新测试整个系统，这无疑增加了软件测试人员的工作量。

自动化测试的目的是弥补手工测试的不足，用自动的方法来实现和替代人工测试中的一些繁琐和机械重复的工作，减少测试人员的工作量，保证软件的质量，提高用户的满意度。

软件测试自动化是软件测试领域必须要经历的阶段，随着应用软件程序规模的不断扩大，复杂程度的不断提高，用户对软件质量的要求也不断在提高，在软件的测试活动里适当使用自动化测试是非常必要的。

软件测试自动化的好处[6]：

(1) 提高测试效率 自动化测试可以更有效，可重复的自动测试。能在更少的时间内完成更多的测试工作，同时也消耗更少的系统资源。从而能提高测试的工作效率。

(2) 降低回归测试的开销 回归测试中多数的测试工作是重复的，采用自动化测试来完成这些重复的工作，可以大大减少回归测试的开销。

(3) 高度可重复性 一个理想的自动测试系统能够让人随时、方便和迅速的运行大量的测试用例。

6 结束语

随着软件组件的可复用性程度不断提高，越来越多的软件开发都是对可复用组件进行集成，并配置和调整已经存在的软件来满足特定的需求。在这种情况下，软件测试大都是系统测试，单独的组件测试将会越来越少。所以软件测试将来的工作更多的集中在系统测试即集成测试和性能测试上面。同时，将自动化引入软件测试领域能有效地减轻测试人员的劳动强度,提高测试的效率和质量,从而节省软件开发的成本,提高软件的质量。软件组件的可复用性程度和软件测试自动化势必将会对软件测试领域产生重大影响。

参考文献

1 赵瑞莲.软件测试方法研究[博士学位论文].北京:中国中科院, 2001.
2 IAN Sommerville.软件工程.北京:机械工业出版社,2007.
3 ROGERS Pressman.软件工程:实践者的研究方法.北京:机械工业出版社,2007.
4 RICHARD Bender.基于需求的测试是软件测试的本质.程序员,2010:9.
5 许静,陈宏刚,王庆人.软件测试方法简述与展望.计算机工程与应用,2003:76－77.
6 应杭.软件自动化测试技术及应用研究[硕士学位论文].杭州:浙江大学,2006.

(上接第 172 页)

采用第一种优化方法，将整数运算的函数转化为 Python 的扩展，然后在 Python 中调用，取得性能测试结果如表 2 所示。

从上述结果可看出，将计算量大的运算剥离，改用 C 编写成为 Python 的扩展模块，能够取得相当于只比 C 语言代码的性能慢 10%的性能，较大程度改善了性能问题。

5 结束语

Python 是最流行的脚本语言之一，但性能较慢。本文以整数运算为例，通过实验对比评估了 Python 与 C 语言性能差异，并从虚拟机内部源代码的层面分析了性能差异产生的原因。本文还总结归纳几种常见的 Python 性能优化方法，并基于其中一种方法进行了优化对比测试，证明能够取得较好的性能优化效果。

参考文献

1 Python 官方网站.[2010-5-13]. http://python.org/about/index.html
2 罗霄,任勇,山秀明. 基于 Python 的混合语言编程及其实现. 计算机应用与软件, 2004,21(12):17－18.
3 陈儒. Python 源代码剖析.北京:电子工业出版社,2008.1－480.
4 丘恩,宋吉广译. Python 核心编程. 北京:人民邮电出版社, 2008.1－654.
5 赖勇浩. Python 性优化经验谈.程序员,2008,4:99－103.