目录

[1. 可观察性（observability） 2](#_Toc13767094)

[1.1 Increasing Class-Component Testability （提高可观察性） 2](#_Toc13767095)

[1.2 On testable object-oriented programming 3](#_Toc13767096)

[1.3 On the Testability of Distributed Real-Time Systems W. Schutz 1991 3](#_Toc13767097)

[1.4 Practicing Testability in the Real World 3](#_Toc13767098)

[1.5 Putting Assertions in Their Place 4](#_Toc13767099)

[1.6 Software Testability: The New Verification the most cited paper 6](#_Toc13767100)

[2. 可控制性（controllability） 7](#_Toc13767101)

[2.1 An Analysis Technique to Increase Testability of Object‐Oriented Components 7](#_Toc13767102)

[2.2 Communications Software Design for Testability: Specification Transformations and Testability Measures 7](#_Toc13767103)

[2.3 Constructing Self-Testable Software Component 8](#_Toc13767104)

[2.4 Heuristics of software testability 8](#_Toc13767105)

[2.5 Testability Analysis Applied to Embedded Data-Flow Software 8](#_Toc13767106)

[2.6 Testability Assessment of Object Oriented Software Using Internal & External Factor Model and Analytic Hierarchy Process \*\*\*\*\* measure 9](#_Toc13767107)

[3. 复杂度Complexity(and simplicity) 14](#_Toc13767108)

[3.2 An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity 14](#_Toc13767109)

[3.3 An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented system 15](#_Toc13767110)

[4. 依赖性Dependency (Cohesion / coupling) 15](#_Toc13767111)

[4.1 A Preliminary Testability Model for Object-Oriented Software \*\*measure 15](#_Toc13767112)

[4.2 A Measurement Framework for Object-Oriented Software Testability 16](#_Toc13767113)

[4.3 An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented system 16](#_Toc13767114)

[5. 可理解性Understandability 17](#_Toc13767115)

[5.1 A Component Testability Model for Verification and Measurement \*\*\*measure 17](#_Toc13767116)

[6. 继承 Inheritance 17](#_Toc13767117)

[7. 测试的可靠性Reliability of tests 18](#_Toc13767118)

[8. 可用性Availability 18](#_Toc13767119)

[8.1 A Component Testability Model for Verification And Measurement 18](#_Toc13767120)

[9. 灵活性Flexibility 19](#_Toc13767121)

[10. 测试套件的可重用性Reusability of test suite 20](#_Toc13767122)

[10.1 Testability of software components 20](#_Toc13767123)

[10.2 An Empirical Study into Class Testability \*\*\*\*\*\*measure 20](#_Toc13767124)

[11. 影响可测试性的其他因素Other factors affecting testability 21](#_Toc13767125)

# 1. 可观察性（observability）

### 1.1 Increasing Class-Component Testability （提高可观察性）

为了提高可观察性，在进行集成测试时，[P90]建议在方法中的最后一个定义和其他方法中的第一个定义之间收集定义 - 使用对。

Abstract: 本文介绍了一种提高组件可测试性的方法。此方法可帮助用户测试在集成期间重用组件的时间。首先，我们分析一个组件来收集定义并使用有关方法和类变量的信息。然后，此信息用于提高组件可测试性以支持组件测试。增强的可测试性有助于检测错误，并帮助测试人员观察状态变量并生成测试输入。本文使用一个例子来报告努力（在测试用例方面）和有效性（就杀死的突变体而言）。

软件组件的一个重要特性是实现透明性，这意味着实现不可用。这意味着测试人员几乎没有关于组件内部状态的信息。缺乏信息使得难以直接应用传统的白盒技术[15]并且难以完全运用软件（缺乏可控性）并且难以知道执行结果（缺乏可观察性）。

测试人员可以通过以下几种方式提高可测试性：

- 从没有源代码的组件收集信息

- 提高可观察性以监控输出

- 增加可控性以支持输入

- 选择测试标准并根据标准生成组件测试的测试用例

该研究项目特别假设该组件是面向对象的。此外，无法访问源代码。首先，我们的方法分析编译的组件以提取定义和使用信息（def-use）。然后，收集的信息用于增加组件可测试性。该信息提供了检测某些错误，观察状态变量和标准的方法[9]。此方法不考虑继承和多态关系。

Background：可测试性的几个定义在第1节中给出。我们的工作将组件可测试性定义为组件支持检测，可观察性和可控性的程度。检测侧重于检测故障的简便性。可观察性侧重于观察输出的容易程度。这意味着该组件支持观察或监视测试结果的方法。此外，可观察的组件不仅允许观察测试的输出，而且还允许观察中间值。可控性侧重于易于控制组件的输入。这意味着组件支持提供在必要时运行组件的输入的方法。

该研究提出了为黑盒组件创建和提供测试的技术，并应用它通过使用方法之间的数据流关系来指导测试选择。 Beizer [1]将集成测试定义为侧重于测试方法之间的接口。方法之间的耦合通过方法之间的数据流和控制流依赖关系来度量。因此，本节给出了数据流分析和基于耦合的测试的简要概述。

… …

coupling-based testing (CBT)。因此，CBT只关注调用（last-defs）之前的变量定义以及调用之后变量的使用（first-uses）。该条件基于以下定义：

•Coupling-def是包含last-def的语句，该last-def可以在至少一个执行路径上的另一个方法中首次使用

•Coupling-use是包含第一个的语句在至少一个执行路径上通过另一个方法的last-def可以达到的使用

•耦合路径是从耦合def到耦合ues的路径

### 1.2 On testable object-oriented programming

[P110]建议可以通过在代码中插入诸如写入或打印之类的探针仪器来实现被测对象的完全可观察性。

本文介绍了**一种有助于设计可测试的面向对象（OO）软件的新理念**。传统OO软件的测试侧重于为现有对象和系统生成测试;可测试的面向对象编程（TOOP）方法提请注意在编码或编译期间将可测试性构建到对象和系统中，以便可以简化测试生成和实现中的后续过程。开发了一种新的TOOP方法来提高OO软件的可测试性。对象级别和系统级别的软件可测试性是定量建模的。构建了一组面向对象中基本控制结构的基本内置可测试机制，以便在测试可控性和可观察性方面提高OO软件的可测试性。 TOOP获得的最有趣的特性是，任何对象中的内置测试都可以以与传统OO软件中的代码或函数相同的方式继承和重用。

### 1.3 On the Testability of Distributed Real-Time Systems W. Schutz 1991

[P116]侧重于分布式实时系统的可测试性。在这样的背景下，由于两个方面，可观察性对于确定SUT是否正确执行很重要：（1）测试工程师必须能够观察环境产生的每个重要事件，并确定正确的排序和时间事件，（2）有必要在测试期间观察系统的动作和输出，但不要干扰其时间行为。

**Abstract:** 事件触发和时间触发系统在可测试性方面进行了比较。为了便于比较，引入了可测试性的三个方面：测试覆盖率，可观察性和可控性。在简要回顾了测试并发程序的问题之后，作者将这两个系统架构与这三个方面进行了比较。比较结果有利于时间触发系统的所有三个方面，最显着的是可控性和测试覆盖率。因此，时间触发系统本质上比事件触发系统更可测试。总结了一系列实验，表明很容易实现时间触发系统的可重复测试。

### 1.4 Practicing Testability in the Real World

[P120]题为“在现实世界中实践可测试性”，是微软的一篇论文。该研究的动机是公司缺少要求更改功能的错误，因为测试它会变得更加简单。作者在应用可测试性概念时提供了经验，并提供了指导以确保在特征规划和设计期间考虑可测试性。它基于名为SOCK的模型（简单性，可观察性，控制和预期结果的知识）。该论文认为，组件越简单，测试成本越低。与暴露状态（可见性和透明度）相关的可观察性。与测试人员是否可以激活组件的每个方面相关的控制。对预期结果的了解也会影响测试。

**Abstract:** 已经研究了可测试性的概念以及如何实现它，例如使用测试钩。尽管它有明显的好处，但我们仍然发现可测试性概念的采用率很低。例如，我们没有看到很多错误要求更改功能，因为测试它会变得更简单。在本文中，我们将介绍我们在应用可测试性概念时的经验，并制定指南以确保在功能规划和设计期间考虑可测性。我们通过简要介绍可测试性概念奠定了基础。我们讨论了测试开发人员思维中的典型思维过程，并提出了为什么练习可测试性很难的关键见解。然后，我们展示了我们在职业生涯中遇到的现实生活中的例子，展示了可测性因素如何使我们的测试变得更简单。我们讨论可测试性对设计的影响。在这些示例的基础上，我们提供了每个可测试性原则的清单。查看此可测试性检查表有助于确保在测试特定功能时考虑核心可测试性原则，并最终简化测试并降低总体成本。我们最后提出了一个练习，我们可以将检查表应用于复杂的事件处理组件，并实现应用可测试性概念的好处。

**可观察性**是指可以有效地分析被测特征组件和整个系统的执行结果的程度，以便准确地确定测试是成功还是失败。通过编程方式查询和确定系统状态越容易和确定，测试的分析阶段就越准确和灵活。此外，自动化测试用例的成本随着可观察性的降低而降低。判断可观察商的主要标准是：“可以通过编程方式轻松访问所有输出，以确定性地知道组件是否通过了测试吗？”

### 1.5 Putting Assertions in Their Place

[P127]题为“将断言置于其位置”主张在完全没有断言（一种常见做法）和每个地点断言的理论理想之间的中间立场。建议的折衷方案是仅在测试不可能发现软件故障的位置进行断言。本文中使用的可测试性测量能够确定测试不太可能有效的位置。本文将断言定义为“通过增加输出空间的维度来提高软件可观察性的方法”。

放置在程序中每个语句的断言可以自动监视程序执行的内部计算。 但是，通用断言的优点是有代价的。 具有如此广泛的内部仪器的程序将比没有仪器的相同程序慢。 一些断言可能是多余的。 在每个位置使用正确的断言检测代码的任务是繁重的，并且不能保证断言本身是正确的。 我们主张在任何断言（最常见的做法）与每个地点断言的理论理想之间存在中间立场。 我们的妥协是仅在传统测试不可能发现软件故障的位置进行断言。 一种可测试性测量，灵敏度分析，确定测试不太可能有效的位置。

我们的研究重点是故障如何“隐藏”测试;最隐蔽的故障是最危险的。在本文中，我们应用一种称为“灵敏度分析”的动态可测试性技术来识别在测试过程中可能隐藏的故障（如果存在）[9]。这些位置被称为“低可测试性”位置。

断言是对执行程序的状态或对程序状态的某些部分的测试的测试。通常，软件测试仅在输出值后才检查值的正确性。相反，断言检查中间值，这些值通常不是由规范定义为输出。内部检查值的好处是我们尽快知道程序是否已进入错误状态。

我们假设所有断言在内部状态满意时评估为TRUE，否则为FALSE。在这种方法中，如果断言评估为FALSE，那么我们认为程序的执行导致失败，即使最终输出根据规范是正确的。因此，我们“人为地”修改我们认为输出空间中的失败：如果输出不正确或断言失败，则会发生故障。实际上，这不仅会修改被视为失败的内容，还会修改被视为输出的内容。

“可观察性”是硬件设计中使用的术语，意味着我们可以检测芯片的内部工作原理。断言是一种通过增加输出空间的维度来提高软件可观察性的方法。然后在我们测试程序N次并且没有观察到失败（使用断言）之后，我们确信错误不会隐藏在代码中。然后我们可以应用“Squeeze Play”模型[6]（首先在ISSRE92和ISSTA93 [lo]中引入）。测试完成后，如果开发团队选择，则可以删除断言。 （断言的删除通常是出于在生产运行期间更有效执行的愿望所致。）

**Fault Simulation in Sensitivity Analysis(敏感性分析中的故障模拟):**

我们将软件可测试性定义为代码在随机测试期间揭示现有故障的趋势。本文建议在质量评估过程中考虑软件可测性预测。

通过灵敏度分析测量的软件可测试性是（程序，输入选择标准）对的函数。选择输入的方法是测试策略的参数：输入可以根据输入分布随机选择，'可以根据程序结构选择，也可以根据测试人员的直觉选择。输入选择标准是敏感性分析产生的信息的基础;应选择标准以满足特定的测试目标。虽然我们仅限于本文中的随机黑盒测试，但其他类型的测试可能有自己的可测试性测量。 [3]

为了通过该定义评估软件具有“良好”可测试性，每当存在故障时，必须可能在测试期间发生故障。要了解这种可能性，有必要了解导致软件故障的事件顺序。 （由于软件故障，我们的意思是由程序中的缺陷引起的错误输出或失败的断言，而不是由程序执行环境的问题引起的。）软件故障仅在以下三个必要和充分的情况下发生条件按顺序发生：

1. 输入必须导致执行故障;
2. 一旦执行了故障，后续数据状态必须包含数据状态错误;
3. 一旦创建了数据状态错误，数据状态错误必须传播到输出状态（我们在这里包括失败的假设作为输出状态。）

为了预测在测试期间发现现有故障的概率，可测试性分析必须预测是否将执行故障，是否会感染后续数据状态，从而产生数据状态错误，以及数据状态错误是否会发生将其不正确性传播到输出中。当现有数据状态错误未传播到任何输出变量时，我们说数据状态错误已被取消。当取消执行期间创建的所有数据状态错误时，触发数据状态错误的故障的存在仍然无法进行测试，从而导致软件可测试性降低。这些条件为预测软件的可测试性提供了一种合理的方法，该方法与故障/故障计算模型紧密耦合。

灵敏度分析[9]可用于评估软件可测试性;它基于故障/故障模型。灵敏度分析基于三个子过程，每个子过程负责估计故障/故障模型的一个条件：执行分析估计根据特定输入分布执行语句的概率;感染分析估计句法突变体影响数据状态的概率;和传播分析估计在更改的数据状态上恢复执行后，已更改的数据状态影响程序输出的概率。

敏感性分析通过估计（1）输入分布，（2）句法突变体和（3）数据状态中的变化值对当前程序行为的影响来预测未来程序行为。更具体地说，当以下程序全部以相同的输入分布运行时，该技术首先观察程序的行为（1）原始程序（2）程序的副本，其中一次一个位置注入语法变体，（3）程序的副本，其中数据状态[由某个输入的程序位置动态创建]的一个值被改变。

在我们的研究中，我们将“位置”定义为单个赋值语句，可以更改控制流，输入语句或输出语句的语句。我们对句法单元构成位置的定义是基于需要隔离程序的不同部分以检查代码部分出错的可能性。此外，我们认为在某个位置的数据状态中发生的情况会影响用户定义的变量或数据状态的程序计数器。即使在某个位置存在副作用，我们也可以将该位置视为两个位置，其中两个不同的变量受到影响。

我们预测位置中的故障将改变程序的数据状态的概率。对于每个位置，该过程重复若干次：对所讨论的位置进行一阶句法语法变异。然后，根据程序的输入分布随机选择输入，运行具有此变异位置的程序若干次。对于执行变异位置的所有时间，我们记录具有变异位置的程序产生与原始位置不同的数据状态的时间的比例;这个比例是我们对这个位置的故障感染概率的估计。通常，针对单个位置制作许多不同的句法突变体，每个都以这种方式产生概率估计。然后，可以使用该位置的概率估计（称为感染估计）以及软件中其他位置的概率估计来预测软件的可测试性。尽管灵敏度分析仅进行简单的代码修改，但耦合效应的证据表明，对于复杂的缺陷，简单突变体观察到的特征可能经常相似[a]。

最后一组预测涉及数据状态错误将传播到输出的概率，从而导致可识别的故障。对于每个位置，该过程重复若干次（在一组程序输入上）：使用从输入分布中随机选择的输入来执行程序。程序执行在执行位置后立即停止，随机生成的数据值被注入某个变量，程序执行恢复。如果位置在循环中，我们通常在每次连续迭代时将另一个随机选择的值注入到同一变量中。有关如何执行此过程的具体细节可在[9]中找到。此过程模拟在执行期间创建数据状态错误。我们将此过程称为“扰乱”数据状态，因为在执行期间某个点处的变量值表示数据状态的一部分。在恢复执行之后，我们自动记录扰动数据状态的任何后续传播到后继输出状态。该过程重复固定次数，每个被扰动的数据状态在执行的同一点影响相同的变量。重复该过程，一次变量就会扰乱不同的变量。使用扰动数据状态发现的概率估计可用于预测程序的哪些区域可能以及哪些区域不可能传播由正版软件故障引起的数据状态错误。然后，可以使用该位置的概率估计（称为传播估计）以及软件中其他位置的概率估计来预测软件的可测试性;这样做的模型见[9]。

… …

Hecht [1]指出，很少遇到的故障可能是安全和关键任务失败的主要原因。即使在系统正在进行旨在提高可靠性的测试和调试时，很少遇到的故障也会长时间隐藏。如果只有一个罕见故障有可能导致灾难性故障，那么我们就会有一个可靠，不安全的系统。因此，这种断言放置模型的目标是通过使用断言攻击程序在其“最弱”点来降低系统中任何故障隐藏的可能性。请注意，使用断言来提高对安全关键应用程序的信心并不是唯一的;该方法的独特之处在于使用“基于可测试性”的断言，这些断言仅放置在代码中预测错误掩蔽程度高的那些位置。在[2]中，Guiho＆Hennebert表示，SACEM是一种部分嵌入式系统的验证，该系统控制着法国巴黎RER A线上所有列车的速度，需要大约100人年的努力;正式证明需要断言。断言是验证工作的关键部分;这些断言也可以在测试期间放在代码中，但显然不是。这里有趣的结果是，开发人员认为他们已经达到了每小时设备失败的概率，即，这个新的SACEM系统和它取代的旧系统一样安全，而流量吞吐量要低得多。因此，断言是验证期间必不可少的实体，无论它们是用于正式证明（验证安全代码）还是用于测试期间的内部自检。这个计划有成本;这些成本包括：（1）测试期间性能下降，（2）执行可测试性分析的成本，以及（3）从规范中获得断言的成本。此外，如果在部署代码之前删除了断言，则会产生轻微的额外成本。但对于关键系统，如果可以相对于方案的成本证明增值收益，则该方案不能自动解除。通过可测试性分析的白盒分析，通过测试进行黑盒分析，以及通过断言进行基于规范的测试，我们可以实际评估测试期间故障未隐藏的高可信度。

### 1.6 Software Testability: The New Verification the most cited paper

[P151]题为“软件可测试性：新验证”是该池中被引用次数最多的论文（截至本文撰写时为392次）。 [P151]将软件可观察性与IC（集成电路）设计的背景下的硬件可观察性联系起来。据报道，IC设计工程师经常使用与软件可观察性密切相关的可观察性概念。在硬件域中，可观察性是能够查看嵌入在电路中的特定节点的值。在这种情况下，测试大规模集成电路的主要障碍是内部信号的不可访问性。在这种情况下增加可观察性的一种方法是增加芯片的引脚数，让额外的引脚执行额外的内部信号，这些信号可以在测试期间进行检查。对于软件，当模块包含局部变量时，在功能测试期间失去查看局部变量信息的能力，这可能成为面向对象系统的主要问题。为了解决这个问题，测试人员可以通过增加单元测试期间检查的数据状态信息量来应用类似于增加芯片中引脚数的概念。

我们的可测试性与另一种意义上的传统观点不同：在过去，软件可测性已被非正式地用于讨论在测试期间可以容易地满足某些输入选择标准。例如，如果测试人员在测试期间需要完全分支覆盖并且发现难以选择覆盖超过一半分支的输入，则该软件将被归类为具有差的可测试性。相反，我们的可测试性不仅仅涉及找到满足覆盖目标的输入集;它可以量化特定类型的测试在测试期间导致现有故障失败的概率。我们将可测试性的定义集中在软件的语义上，以及它在包含故障时的行为方式。这与询问是否有利于覆盖或是否正确不同。

我们对软件可测试性的定义着重于软件包含故障时，一个软件在测试期间的下一次执行失败的概率（使用假设的输入分布）。相比之下，标准的IEEE定义侧重于评估I/O对是否正确。

# 2. 可控制性（controllability）

对于可控性有几种略微不同的定义。它是可以根据测试要求控制被测组件状态的程度。另一个定义是，它确定了在价值，操作和行为方面为程序提供所需输入以执行某个条件或路径是多么容易。如果受测试的系统或组件支持各种提供输入的方式来运用它，它往往提供更好的可控性，从而提高可测试性。 Binder [P64]定义了可控性的重要性如下：“如果用户无法控制输入，他们无法确定导致给定输出的原因”。可控性直接影响可测试性，因为如果不容易控制测试所需的被测组件的状态，并且因此测试各种条件，路径，操作和行为将是具有挑战性的。

### 2.1 An Analysis Technique to Increase Testability of Object‐Oriented Components

[P25]提出了一种分析技术，以提高面向对象组件的可测试性。分析组件的字节码以开发控制和数据流图，然后通过增加可控性和可观察性来改善组件的可测试性。

为了提高可控性，应用定义和使用信息来收集最后定义和首次使用之间的定义 - 使用变量对（即definition – use pairs）。通过支持测试用例生成来覆盖所有必要的测试，应用定义 - 使用变量对来增加可控性。

面向对象的组件工程越来越多地用于系统开发，部分原因在于它强调可移植性和可重用性。每次使用组件时，都必须在新环境中重新测试。不幸的是，组件通常使用的数据抽象导致可测试性低。首先，不能直接设置内部变量。其次，即使测试输入可能触发故障，故障也不会传播到输出。本文介绍了一种增加面向对象组件可测试性的技术，从而更容易检测故障。组件通常是密封的，因此源代码不可用。程序分析在Java组件字节码级别执行。分析组件的字节码以创建控制和数据流图，然后通过增加可控性和可观察性来增加组件的可测试性。我们已经实现了这种技术并将其应用于多个组件。实验结果表明，通过使用我们不断增加的可测试性过程可以增加故障检测。

### 2.2 Communications Software Design for Testability: Specification Transformations and Testability Measures

[P52]侧重于在通信软件环境中使用的有限状态机（FSM）。通过使用形式化方法，它定义了基于可达性集的FSM的可控性度量度量。

在本文中，我们在规范级别上讨论DFT。我们首先定义通信软件设计的通用框架，以便在协议开发生命周期活动中集成可测试性。然后，我们识别并定义各种可测试性度量，例如可控性，模糊性，状态表征和抽象度，它们共同定义了统一的可测性度量。最后，我们确定并定义了一些可以增强给定规范的统一可测性度量的转换。这些变换与三个可测试性增强因素有关，即规范的指定性，最小性和确定性。

### 2.3 Constructing Self-Testable Software Component

[P56]专注于构建可自行测试的软件组件。其中一个想法是使用插桩，这是在测试类中引入的额外软件，以提高其可控性和可观察性。该工具，也称为内置测试（BIT）功能，包括：断言，报告方法和BIT访问控制。

基于组件的软件工程技术因其具有提高新软件应用的生产率和降低开发成本的潜力而备受关注，同时满足高可靠性要求。第一步IO解决如此高的可靠性要求包括重用可靠的组件。为了使属性“可靠”，应该广泛验证组件。至于测试是最常用于验证的技术，这意味着可以对可重用组件进行良好测试。对于测试IO应该有效且及时地应用，组件应该是可测试的。本文介绍了一种通过将测试资源集成到其中来改进组件测试性的方法，从而获得n个自测稳定组件。开发了一个原型工具Concat，WAS以支持所提出的方法。该工具适用于Ci中实现的00个组件。还讨论了所提出的测试方法的故障检测有效性的实证评估的Sotne初步结果。

在本文中，我们提出了一种构建和使用自检组件的方法。除了实现之外，可自我测试的组件还包含可以从中导出测试用例的规范。我们在这项研究中关注的是00个独特类别的组件。开发了一个原型工具，以支持拟议方法的某些活动，以显示其可行性。这个名为Concat [33]的原型工具支持构建和使用C++语言实现的自测试组件。

### 2.4 Heuristics of software testability

[P81]讨论了两种不同背景下的可控性：（1）环境可控性，在项目相关可测性的背景下，这是人们可以控制我们测试周围环境中所有可能相关的实验变量的程度; （2）在内在可测试性的范围内的内在可控性，其涉及能够容易且立即地提供任何可能的输入并且根据需要调用任何可能的状态，状态组合或状态序列。

### 2.5 Testability Analysis Applied to Embedded Data-Flow Software

[P160]提出了基于数据流路径F的模块M的可控性的定量测量，如下：CO（M）= T（I）/ C（I），其中T（M）是模块的最大信息量M从流F的输入接收，而C（M）是模块M在隔离时将接收的总信息量。

可测试性是软件的重要品质因素，尤其是诸如航空电子软件之类的嵌入式数据流软件。此类软件缺乏可测试性会严重影响测试成本和软件可靠性。可测试性分析可用于识别难以测试的软件部分。在本文中，我们建议使用静态单一分配（SSA）表单将从数据流设计生成的源代码转换为数据流表示，然后我们描述一些算法以自动将SSA表单转换为可测试性模型。因此，可以将一些度量应用于可测试性模型，以便定位导致可测试性弱点的软件部分。

在本文中，我们关注从数据流设计生成的源代码的可测试性测量。这些措施可用于1）识别导致缺乏可测试性的设计部分，2）比较设计可测试性和生成的代码可测试性以评估代码生成过程，以及3）提高软件可靠性。可测试性分析对于航空电子软件等嵌入式数据流软件至关重要;由于这些软件部件通常采用数据流方法设计，因此本文主要关注数据流软件。

在之前的工作中，Le Traon和Robach [6,7]提出了一种可测试性模型，该模型在SATAN工具（系统的自动可测性分析）中实现，用于分析数据流软件的可测试性。所提出的可测试性模型是一个二分图，它被证明适用于模型数据流设计。但是，它不能用于分析从设计中使用的设计或组件代码生成的源代码：因为源代码通常是在命令式语言中生成或描述的。因此，我们最近提出使用静态单一赋值（SSA）形式将此模型应用于组件代码的可测试性分析[10]。从数据流设计生成的代码确实是设计中使用的组件代码的集成。

因此，可以使用SSA表单作为源代码的中间表示来分析代码可测试性。然后，我们提出了一些算法来自动将SSA表单转换为可测试性模型，该模型用于计算源代码的可测试性度量。

本文介绍了一种从数据流设计自动生成的代码的可测试性分析方法。使用SSA表单可以使用我们的SATAN工具分析代码可测试性。代码级别的分析比设计级别更复杂，但它可以提供更准确的结果。特别是，这种方法有助于在使用之前分析某些嵌入式数据流软件代码的可测试性，这是验证机构所要求的。由于SATAN方法可以在设计级别和代码级别分析数据流软件的可测试性，因此可以在整个软件开发项目中考虑可测试性分析。

[12] C. Robach and S. Guibert. Information based testability measures. In Proceedings of Silicon Design Conference, pages 429–438, Wembley, G.B., 1986.

[13] C. Robach and S. Guibert. Testability measures: a review. International Journal of Computer Systems Science and Engineering, 3:117–126, 1988.

### 2.6 Testability Assessment of Object Oriented Software Using Internal & External Factor Model and Analytic Hierarchy Process \*\*\*\*\* measure

[P171]如下将可控性与若干内部OO特征（封装，耦合内聚和多态）联系起来。封装促进了可控性。耦合使得可控性变得困难。凝聚力有助于改善它。多态性进一步降低了可控性。

可测性研究实际上是从减少测试工作量和测试成本的前景中完成的，这超过了任何软件总开发成本的40％[3]。尽管如此，可测试性领域的研究还没有详细进行。正如我们之前关于可测试性和可测试性指标[4]，[5]的工作中所讨论的，已经发现可测性研究仅在过去几年中加速了。很难，大部分工作都是使用各种面向对象的特色指标完成的。在本文中，我们提出了一种用于在设计时评估的可测试性模型，并使用AHP技术对其进行评估。

**Proposed Model:**

我们提出的可测试性模型基于Dromey的软件质量模型[36]，该模型已成为迄今为止用于各种质量特征以及许多可测试性模型的基准。我们按照下面提到的步骤来正式化模型：

1.识别面向对象软件可测试性评估的内部设计功能。

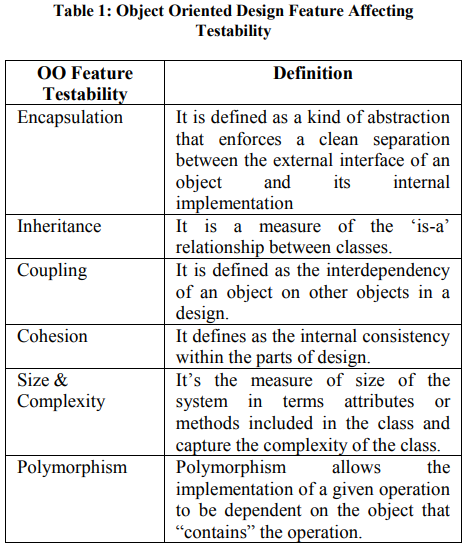
2.从每个流行指标中识别静态指标。 

3.在可测试性和这些已识别的因素之间建立联系。 

4.遵循使用AHP技术的模型评估。

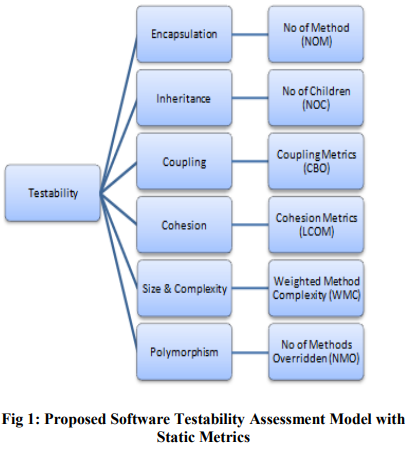
在我们之前的研究工作和调查的基础上，我们确定了六个因素来评估面向对象软件在设计层面的可测性[4]，[5]。所有这些都是内部质量特征 - 封装，继承，耦合，内聚，多态和尺寸和复杂性，如表1所示。在六个已确定的特征中，MTMOOD可测性模型[10]中提出了四个特征，其中未涵盖多态性大小和复杂性特征，也被许多研究人员在可测性研究中发现作为必要的内部特征[15]，[22]，[36]，[37]。

这六个面向对象的特征直接或间接地在可测试性即兴创作中起着非常重要的作用。这种关系是在对出版物[2]，[20]，[35]，[38]，[39]等进行彻底研究的基础上建立的。

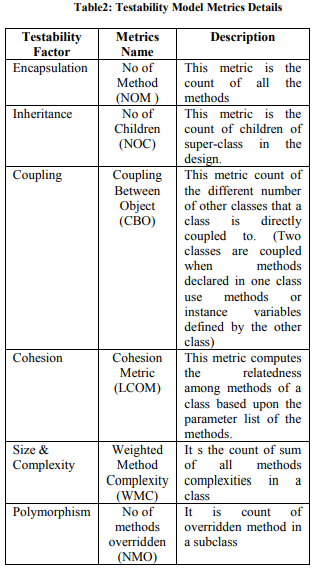


研究表明封装提高了效率和复杂性。继承对效率，复杂性，可重用性和可测试性或可维护性具有重要影响。虽然低耦合被认为有利于可理解性，复杂性，可重用性和可测试性或可维护性，而较高的耦合度被认为对这些质量属性产生不利影响。凝聚力被认为对设计的可理解性和可重用性有显着影响。大小和复杂性对可理解性，可测试性或可维护性具有重大影响。多态性降低了复杂性并提高了可重用性。虽然这些功能可以通过前面讨论过的许多度量选项来衡量[5]。大多数这些指标都被从业者接受“重用率和受欢迎程度”以及经验（后期开发）验证的学术专家。但是为了使AHP评估方面的研究变得简单，我们在可测性研究人员中选择了一些基本但很受欢迎的指标。

因此，所提出的关于使用静态度量的内部设计特征的可测试性评估模型如图1所示。它基于上面提到的六个面向对象特征，从可测性角度来看，如Binders研究中所指出的那样[6]。在我们之前的工作[41]中讨论的所有流行度量套件中，已经确定了下面表2中解释的六个静态度量，用于评估这些特征中的每一个及其对设计时任何面向对象的软件可测试性的影响。



正如下面的表2所述，对于封装评估，许多研究人员正在选择方法指标（NOM）的数量，以确定信息隐藏对可测试性的影响[10]，[38]。所以我们也将它保留给我们模型的封装评估。使用Number of Children度量（NOC）评估继承，NOM是最流行和最有效的继承度量之一[22]，[36]，[41]，[42]。对于耦合，我们选择了对象之间的耦合（CBO），对于Cohesion，我们选择了内聚度量（Li和Henry版本）（LCOM）。根据文献研究和行业从业者的普及，这两个是最受追捧和无与伦比的指标，可用于评估耦合和凝聚力对可测试性的影响[10]，[20]，[22]，[24]，[37]，[43]虽然尺寸和复杂度可以通过此类别中的其他指标轻松测量，但我们选择加权方法复杂度（WMC）指标，因为它在测试用例指示数量方面具有重要作用和关联[6]，[22]，[42] ]。多态性是影响可测试性的潜在因素之一，但是像Binder等早期研究人员强调的那样[6]，[45]因为它导致可测性降低，我们选择多态性因子指标（POF / PF）进行可测性评估。



**Material And Methodology:**

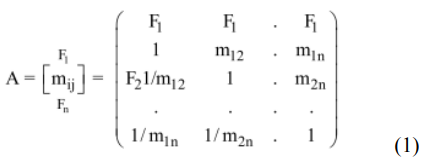
1. **AHP Methodology**

它最初要求将目标目标划分为因子和子因素的层次结构，这可以很容易地单独分析。一旦构建了层次结构，决策者的工作就是按如下方式评估问题：

步骤1

倒数矩阵形成：首先，基于这些因素构建了成对比较矩阵。每个因素都需要与下一个因素进行比较。 Saaty的常用比例如下表3所示。

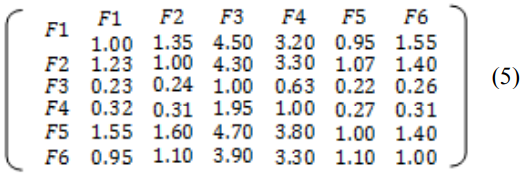
这样形成的矩阵在某种程度上看起来像这样。假设对于n个因子，考虑F1，F2 ...... .Fn，它们将被比较。与Fj相关的Fi的相对权重表示为mij，并且正方形矩阵A = [mij]的n阶将形成如下面的等式（1）中给出的。



这里，mij = 1 / mji，对于所有i，i不等于j和mii = 1。因此，计算的矩阵称为倒数矩阵。

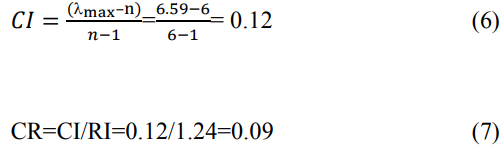
**Proposed Model Evaluation:**

如上所述，向10位专家发送6X6的方阵以用于成对的权重填充。使用这10个样品在六个可测性因子上形成的平均矩阵如下：



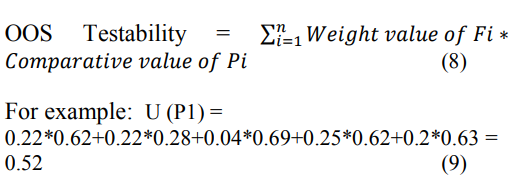
有许多计算特征向量的方法。我们使用基于电子表格的近似计算来确定标准的本地优先级。如此计算的本征值如下表5所示。 F1，F2，F3，F4，F5和F6的相对重要性的特征向量是（0.22,0.22,0.04,0.06,0.25,0.20），其在表-5中给出。**这些值是主要因素的权重，即封装性（0.22），遗传（0.22），偶联（0.04），内聚力（0.06），尺寸和复杂性（0.25）以及可测性评估中的多态性（0.20）。**

现在，针对这些因子中的每一个计算的六个本征值是（6.62,6.61,6.58,6.47,6.63,6.60），其中max= 6.59，其> = 6（因子总数），这是一致的。使用此我们计算CI和CR值如下：



… …

现在最后我们必须构建三个选定项目P1，P2和P3的特征向量矩阵和六个可测性评估因子权重F1，F2，F3，F4，F5和F6，如下所述。每个项目的总体全球效用是使用OO项目权重的乘积的总和，参考每个因素，通过相应因子的权重得出每个OO项目的全局效用。



最好的OO项目是具有最高整体可测试性指标值的项目。因此，完成了OO项目的排名，如表13所示，并且P1被认为是最佳选择，因为其可测试性指数值在三个中最高。

**Result and Findings:**

**上述技术表明封装（22％），遗传（22％），耦合（4％），凝聚力（6％），尺寸和复杂度（25％）和多态性（20％）在任何整体可测性评估中的作用OO项目基于AHP技术的抽样调查。此处的结果用于三个中型项目，用于整体可测性指数（TI,** **testability index）计算。在实际情况中，可以从使用面向对象技术开发的运行项目中收集特征的比较值。虽然这里比较的项目是中等规模的项目，但仍然足以支持该模型。然而，我们的动机是展示所提出的方案对面向对象软件的可测性评估的适用性。提议的方案可以基于所识别的六个因素的值应用于现实生活软件，并且将确定所考虑的软件的可测试性指数（TI）。它可以应用于每个模块（方法，类，包，模块等）以了解它们的可测试性，或者它也可以应用于整个开发的系统以了解其整体可测试性。**

# 3. 复杂度Complexity(and simplicity)

**3.1 A Software Testing Assessment to Manage Project Testability**

软件复杂性是一个术语，它包含一个软件的众多属性，所有这些属性都会影响内部交互。简单性被视为复杂性的对立面，复杂性被定义为软件工件具有单一，明确定义的责任的程度。 [P17]发现简单性对可测试性的设计有积极影响。

对测试服务的需求在很大程度上是“衍生需求”直接受到先前开发活动的方式的影响。结构化软件开发生命周期（SDLC）项目的早期阶段通常可能落后于计划，缩短了执行适当测试的可用时间，尤其是在必须满足软件发布期限时。这种情况促使需要影响预测试活动并有效地管理测试工作。我们的研究探讨了如何在测试开始之前测量SDLC项目的可测试性。它建立在“可测试性设计”的基础上，引入了“可管理性”的视角。软件可测试性侧重于SDLC流程的活动是否正在以能够使测试团队找到软件产品缺陷（如果存在）的方式发展。为了应对这一挑战，我们开发了一个软件测试评估。此评估旨在为测试经理提供以下信息：（1）以最终提高测试效率和有效性的方式影响预测试活动，以及（2）规划测试资源以优化高效和有效的测试。我们通过与关键信息提供者的访谈制定了具体的软件测试评估措我们提供了针对大型结构化软件开发项目的度量收集的数据，以说明评估的有用性和应用。

### 3.2 An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity

[P31]报告了动态软件可测性度量与静态圈复杂度的经验比较，发现它们之间存在正相关关系。

### 3.3 An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented system

此外，[P37]还发现，高度复杂的类可能需要比低复杂度类更多的单元测试工作。如果一个类非常复杂（例如，它的方法具有高度复杂的控制流结构和/或方法和属性之间的交互模式非常复杂），则可能需要更多的努力。用于检查测试用例执行是否成功的代码也可能更复杂。因此，可能需要更多的努力来对高度复杂的类进行单元测试。

# 4. 依赖性Dependency (Cohesion / coupling)

内聚指的是模块内的元素属于一起的程度。它衡量一个类的方法和数据之间的关系强度，以及该类所服务的一些统一目的或概念。耦合是软件模块之间的相互依赖程度;衡量两个例程或模块之间的紧密联系的度量;模块之间关系的强度。

提到内聚和耦合都影响可测试性。已发现一般的内聚力可增强可测试性。一方面，类中方法之间的内聚性是可取的，因为它通常意味着减少方法的数量，类的大小，从而减少测试工作[P15]。另一方面，低内聚通常意味着设计不良或类的组织不良，从而增加了类的复杂性和错误的可能性。因此，类中方法的内聚性越强，该类的可测试性就越高。方法缺乏凝聚力导致降低该类的可测试性[P15]。

### 4.1 A Preliminary Testability Model for Object-Oriented Software \*\*measure

在软件质量保证中，已经使用两种方法来提高故障检测的有效性。人们试图找到更有效的测试策略;另一个尝试使用软件本身的设计特征来增加故障暴露的可能性（如果确实存在的话）。后者结合了故障预防和故障检测的最佳特性，也被称为软件质量保证的可测试性方法。本文研究了影响面向对象软件中软件可测试性的因素，并提出了评估软件可测试性度量的初步框架。最终目标是在面向对象设计中制定一套指南，通过提高可测试性来提高软件质量。

在本文中，我们使用Voas＆Miller [3]提出的定义，该定义侧重于软件设计，以增加“如果软件包含故障，它将在测试期间下次执行时失败的概率”。因此，可测试性是程序显示其潜在故障的容易程度。程序的可测试性越高，程序中潜在的错误就越容易显现出来;可测试性越低，故障被检测的可能性越大。可测试性方法结合了故障预防和故障检测方法的最佳方面。一方面，它需要仔细设计软件，从而改善故障预防;另一方面，它增加了故障自我暴露的可能性，从而使软件故障检测更容易，成本更低。

### 4.2 A Measurement Framework for Object-Oriented Software Testability

[P9]提到，只要测试或集成的单元依赖于尚未测试或甚至编码的其他单元，单元测试和集成测试就需要测试存根。这些单元之间的耦合驱动了短截线工作，因为增加耦合可能会导致短截线中的附加特征，并因此降低可测试性。但是，鼓励某种类型的耦合，即在类之间使用继承的耦合，因为它增强了可测试性。

在本研究中，我们遵守IEEE [24]和ISO [25]的定义。我们的目标是提供一个全面的框架，以实用的方式帮助测量和评估可测试性，重点是面向对象开发的分析和设计阶段。我们的主要动机是，在分析和设计阶段，可测试性分析可以产生最高的回报：可以在实施开始之前做出设计决策以提高可测试性。第2节中对现有技术的回顾用于证明本研究的动机以及我们在分析和设计层面定义和构建软件可测试性测量框架的目标（第3节）。在第4部分，部分基于我们对相关工作的审查，我们列出了可能对软件可测试性产生影响的许多属性。我们根据这些属性与测试活动的关系对这些属性进行分类，并确定它们影响可测试性的已知机制

### 4.3 An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented system

[P37]也报告了类似的发现。它指出，具有低内聚力的类可能需要比具有高内聚力的类更多的单元测试工作。弱的凝聚力表明一个阶级可能会服务于几个不相关的目标，这表明不恰当的设计。封装不相关的属性/方法可能会导致更复杂的测试。因此，可能需要更多的努力来对具有低内聚力的类进行单元测试。具有高耦合的类可能比具有低耦合的类需要更多的单元测试工作。一方面，具有高导入耦合的类取决于许多外部提供的服务。在对这样的类进行单元测试时，必须开发许多存根来模拟任何尚未提供的外部提供的服务的行为。另一方面，具有高导出耦合的类对系统有很大影响：许多其他类依赖于它。这样的课程在测试期间可能会受到额外的关注，因此也可能需要更大的测试套件。因此，对于具有高耦合的类进行单元测试可能需要更多的努力。

[P40]将对象/单元之间的依赖性（一种耦合类型）称为可测试性反模式。

-Testability是一种软件特性，旨在生成易于测试的系统。可测试性反模式是一个可能会对软件的可测试性产生负面影响的因素。在本文中，我们比较了源代码级别和不同抽象级别的反模式，以便了解它们在开发过程中引入的时间点。

# 5. 可理解性Understandability

### 5.1 A Component Testability Model for Verification and Measurement \*\*\*measure

可理解性是指软件工件记录或自我解释的程度。 [P2]讨论了组件可理解性，它指的是生成的组件用户手册，API规范如何易于用户操作和测试。因此，它影响可测试性。

由于组件是基于组件的系统的主要构建块，因此开发高质量组件对于基于组件的软件工程来说变得非常关键。要生成高质量的组件，我们必须注意组件的可测试性，以确保可重用组件不仅可以由组件供应商进行测试，而且可以由组件用户轻松验证。因此，组件可测试性分析，验证和测量成为测试组件和基于组件的系统中非常重要的研究课题。本文讨论了基于组件可测试性分析模型的可量化方法中的组件可测试性。工程师可以使用此模型在组件开发过程中验证和测量组件可测试性。基于该可测试性模型，本文讨论了组件可测试性验证，并提出了一种用于可测性测量的五角星模型。

# 6. 继承 Inheritance

6.1 A Measurement Framework for Object-Oriented Software Testability

通常，继承也会降低可测试性。 [P9]发现，继承层次结构的大小越大，由于子类之间的依赖性，测试的成本就越高。换句话说，由于继承，测试超类和子类之间的接口可能更昂贵，因为需要额外的测试用例，并且可能需要修改每个子类的测试oracles。

6.2 A Preliminary Testability Model for Object-Oriented Software

[P15]发现继承树中类的深度会影响类的可测试性。通常，类在继承树中越深，它将继承的方法的数量就越多。这使得测试和维护它变得更加复杂。如果一个类靠近继承树的根，那么每次测试用例通过它的后代时，它都有更多机会被测试，它也将通过类本身。因此，类越接近继承树的根，类的可测试性越高，而类在继承树中越深，其可测试性就越低。

6.3 An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems

[P37]报道了类似的发现。具有许多祖先，许多后代，深度继承层次结构，许多重写方法和/或许多重载方法的类可能需要比具有少量祖先，几个后代，影子继承层次结构，少数重写方法和少量重载的类更多的单元测试工作。方法。如果一个类有更多的祖先或位于继承层次结构的深处，它可能会从其祖先继承更多的属性和方法。因此，可能需要更多努力来覆盖继承的属性/方法，并且用于检查测试用例执行是否成功的测试代码也可能更复杂。

# 7. 测试的可靠性Reliability of tests

7.1 A Study of the Relationship between System Testability and Modularity

[P21]定义了可测试性的一个有趣方面：测试的可靠性，定义为测试结果的置信度，或测试的正确结果的概率准确性。

模块化是一种设计方法，除其他设计属性外，还有助于提高可测试性。然而，模块化类型与系统可测试性之间的关系尚未得到充分研究。在本文中，我们提出了一个可测试性的概率模型，使我们能够研究模块性和可测试性之间的关系。特别是，我们关注与为系统可测试性唯一目的而选择的特定模块化相关的成本和收益。我们表明，通过仔细的模块化采用适当的测试架构可以大大提高系统测试的效率和有效性。

本文研究了测试架构对测试效率和有效性的影响。该体系结构依次通过组件和接口建模来创建子系统或模块。在下一节中，我们将介绍一个简单的集成和测试模型。基于该模型，建立模拟以在概率意义上估计测试的预期时间/成本以及测试后的预期系统可靠性。上一节讨论了测试体系结构对这两种可测试性结果的作用。

在设计，装配，部署，运营，维护和退役时设计系统模块化时，使用模块化的原因有很多。在其他设计属性中，我们在本文中已经表明模块化也有助于可测试性。在本文中，我们提出了一个可测试性的概率模型，使我们能够研究模块性和可测试性之间的关系。特别是，我们关注与为系统可测试性唯一目的而选择的特定模块化相关的成本和收益。我们发现，尽管从架构到架构的预期时间/成本差别很大，但很明显，相对于非模块化架构（单层测试架构），模块化为两级架构可将测试成本降低50％以上。此外，时间/成本对测试的拓扑结构比对单元可靠性更敏感。特别地，在均匀的部件可靠性和测试可靠性条件下，较低的成本似乎导致在每个模块内部的部件/接口的数量方面的最平衡的模块化。

# 8. 可用性Availability

可用性是系统的一个特征，旨在确保达到一致的运营绩效水平，通常是正常运行时间，高于正常水平。

### 8.1 A Component Testability Model for Verification And Measurement

[P2]发现文档可用性（涉及组件工件的可用性，包括需求规范，API规范和用户参考手册）会影响可测试性。

由于组件是基于组件的系统的主要构建块，因此开发高质量组件对于基于组件的软件工程来说变得非常关键。要生成高质量的组件，我们必须注意组件的可测试性，以确保可重用组件不仅可以由组件供应商进行测试，而且可以由组件用户轻松验证。因此，组件可测试性分析，验证和测量成为测试组件和基于组件的系统中非常重要的研究课题。本文讨论了基于组件可测试性分析模型的可量化方法中的组件可测试性。工程师可以使用此模型在组件开发过程中验证和测量组件可测试性。基于该可测试性模型，本文讨论了组件可测试性验证，并提出了一种用于可测性测量的五角星模型。

8.2 A Model for the Measurement of the Runtime Testability of Component–Based Systems

[P10]提供了一种用于测量基于组件的系统的运行时可测试性的模型。它发现，如果组件具有高可用性要求，则无法执行诸如要求之类的运行时测试，因为它会降低高可用性。因此，它的运行时可测试性会降低。运行时可测试性在[P10]中以两种方式定义，即（1）系统或组件在不受到广泛影响的情况下促进运行时测试的程度; （2）允许在运行时执行哪些测试而不会对运行系统产生广泛影响的规范

# 9. 灵活性Flexibility

题为“灵活性：可测试性的关键因素”，[P78，P177]强调了单元或系统灵活性的重要性及其对可测试性的影响。此外，[P173]中探讨的主要研究问题是：灵活性如何影响基于框架的应用程序的可测试性？

9.1 Flexibility: a key factor to testability

[78] 可测试性是一个重要的软件质量因素，如果在开发生命周期的早期阶段无法获得，则无效。在面向对象设计的情况下，它变得更加重要。灵活性是可测试性分析和测量的重要关键因素，可提供高级可测试和可维护的软件。灵活性是软件开发人员，设计人员和质量控制人员至关重要的标准。它不断引导和支持，以避免浪费资源，并使设计人员能够在开发过程中不断改进。灵活性涉及在成本和时间的约束下构建高质量和可靠的软件。它极大地影响了软件演化过程中的成本，质量和可靠性。尽管灵活性对软件开发过程至关重要且非常重要，但管理不善。本文重点关注设计阶段早期灵活性的必要性和重要性。通过建立多元线性回归，提出了一种用于面向对象设计的灵活性测量的模型。最后，使用实验试验验证了所提出的模型。

灵活性是衡量面向对象软件设计可测试性的最重要因素之一。研究开发了灵活性测量模型，该模型建立了灵活性，面向对象设计属性和面向对象度量之间的相关性本文展示了灵活性作为可测试性的关键因素及其与各种面向对象设计属性的关系的重要性。**设计阶段**的灵活性测量模型已经在理论上和经验上使用实验试验开发和验证。对于实验验证，已经使用了几个大型商业项目。对灵活性模型的应用验证得出结论，开发的模型具有高可靠性，符合标准和重要性。最后论文得出结论，灵活性和可测试性之间存在很高的相关性，此外统计结果表明，灵活性模型在统计上非常重要和可靠。

9.2Testability measurement model for object oriented design

[177]在本文中，确定了软件可测试性关键因素，并分析了它们在设计阶段对可测性测量和改进的影响。 “可修改性和灵活性”是影响面向对象设计可测试性的两个关键因素。考虑到两个主要因素，已经开发了面向对象设计的可测试性测量模型（TMMOOD），并且统计推断被验证以获得高水平的更好的可接受性。用于测量面向对象软件设计可测试性的开发模型非常值得信赖，并且与面向对象的设计结构相关。可测性测量模型已经在理论上和经验上使用实验试验进行了验证。对可测试性模型的应用验证得出结论，所提出的模型高度一致，可接受且可靠。

9.3 Testability Estimation of Framework Based Applications

在当今的情况下，高度需要软件系统的质量。高度可测试的系统也提高了可靠性。超过50％的测试工作时间和成本用于生产高度可测试的系统。因此，需要设计可测试性来减少测试工作量。为了提高软件组织的质量，生产力和降低成本，正在促进生产面向重用的产品。将重用技术整合到软件开发过程和测试过程这两个方面可能会带来很多好处。保持这种观点，我们的研究重点关注面向对象框架的软件系统的可测试性，并确定面向对象框架的变量点的灵活性，选择用于框架实例化，极大地影响面向对象框架的软件的可测试性。每个级别的测试。在本文中，我们提出了一个可测试性模型，考虑可变点的灵活方面来估计基于框架的软件的早期阶段，需求分析阶段和开发过程的可测试性。所提出的模型有助于在实施开始之前提高软件的可测试性，从而降低总体开发成本。

# 10. 测试套件的可重用性Reusability of test suite

[P32]和[P182]提到测试套件的高可重用性和结构对可测试性产生积极影响。

### 10.1 Testability of software components

[p182]通过将可观察性和可控性的概念应用于软件来定义软件的域可测试性的概念。结果表明，可域可测试的程序不会出现任何输入输出不一致性，并且支持小测试集，其中很容易理解测试输出。讨论了可用于评估修改程序以使其可以进行域可测试所需工作量的度量标准。还讨论了从程序规范和实验中评估可测试性，该实验表明，构建和测试从域可测试规范开发的程序所花费的时间少于从非域可测试规范开发的类似程序。

### 10.2 An Empirical Study into Class Testability \*\*\*\*\*\*measure

P[32]在本文中，我们研究面向对象软件系统的可测试性因素。起点是通过对文献的研究给出的，以获得可测试性的初始模型和与可测试性相关的现有面向对象度量。随后，通过对存在JUnit测试用例的商业和开源Java系统的五个案例研究来评估这些度量。本文的目标是识别和评估一组可用于评估Java系统类可测试性的指标。

# 11. 影响可测试性的其他因素Other factors affecting testability

此外，56篇论文提到了其他其他因素。以下是这些因素的一些例子：单位大小[P9];有状态[P10];可分性，测试自动化[P17];软件过程能力[P29];模块化[P33];内置测试功能和测试支持环境[P64];容易出错[P76];可管理性和可支持性[P120]测试套件的质量[P122];和自我记录[P169]。

其他尝试对可测性和影响它的因素进行分类：我们还应该注意到，其他尝试对可测试性进行分类以及影响它的因素已经在社区中呈现。例如，当提出软件可测试性的启发式时，一位名叫James Bach的从业者将可测试性分为五类（www.satisfice.com/tools/testable.pdf）:( 1）内在可测性：SUT的内部特征影响可测试性（例如，可观察性和可控性）; （2）与项目相关的可测试性：信息可用性等问题（我们是否获得了我们想要或需要测试的所有信息？）或工具可用性（我们是否提供了测试所需的所有工具？）; （3）主观可测试性：测试技能（我们的测试能力显然使测试更容易）和产品知识等问题（对产品有很多了解，包括内部如何工作，深刻地提高了我们测试它的能力）; （4）与价值相关的可测试性：我们对产品的期望，例如oracle可靠性：我们受益于可以信任在不同的测试执行和许多条件下工作的神谕; （5）认知可测试性：我们所知道的与我们需要了解的产品状态之间的差距。例如，测试自动化非常适合为我们提供增加认知可测试性的假象，例如：“每晚，我们在不到一个小时内运行10,000次测试！”虽然它实际上可以降低可测试性：“Bob每天花费四个小时处理测试自动化的结果！“

Kedemo使用James Bach的分类来开发扩展模型，并将其称为可测试性的三个维度[65] :( 1）产品维度：代码和环境，团队和愿景; （2）测试者维度：技能和知识，心理状态; （3）背景驱动因素，如：风险，资源，发展范式。这些驱动程序中的每一个都对可测试性具有正面或负面影响，例如，更多资源通常会导致更好的可测试性。由测试活动或故障引起的更多项目风险通常会导致可测试性低。前面讨论的内在可测试性因素对应于该模型中“产品”组下的因素。

最后，其他一些作者提供了清单。例如，题为“在现实世界中练习可测试性”，[P120]提出了一个可测试性检查表，其中包括以下问题：

•在测试实际组件和稍后“清理”之前，是否需要花费大量时间和代码来“设置”？

•由于您正在测试的组件外部的组件，您的测试是否会间歇性地失败？

•通过直接使用组件，您是否可以使测试执行更简单，更容易？

•您的组件可以自行初始化还是单独启动？

•你能准确指出失败的原因吗？

•您的组件是否遵循既定的设计模式？