# 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。