**构件可测试性测量方法**

目录

[概述 1](#_Toc14965236)

[介绍 1](#_Toc14965237)

[构件可测试性测量 2](#_Toc14965238)

[面向对象类的可测试性测量模型 3](#_Toc14965239)

[基于运行时类信息的可测试性测量方法 3](#_Toc14965240)

[基于控制流的可测试性测量模型 4](#_Toc14965241)

[面向程序包级别的可测试性测量方法 7](#_Toc14965242)

[面向构件运行时的可测试性测量方法 9](#_Toc14965243)

[面向开发过程中构件的可测试性测量模型 11](#_Toc14965244)

[参考文献 14](#_Toc14965245)

## 概述

由于构件是基于构件系统的主要构建块，因此开发高质量构件对于基于构件的软件工程来说变得非常关键。要生成高质量的构件，我们必须注意构件的可测试性，以确保可重用组件不仅可以由构件供应商进行测试，而且可以由构件用户轻松验证。因此，构件可测试性分析，验证和测量是测试构件和基于构件的系统中非常重要的研究问题。本文调研了国内外近二十年的可测试性方法研究，并给出了6种可用于构件程序可测试性分析的验证方法和具体模型。工程师可以使用这些模型在构件开发过程中验证和测量构件程序可测试性。

## 介绍

在基于构件的软件工程中，可重用软件构件是构建基于构件系统的构建部件。可重用构件的任何缺陷都会对基于构件系统的质量产生严重的影响，这些系统是基于它们构建的。因此，可重用构件的质量控制和验证对于构件供应商和用户构建基于构件的软件非常重要。根据已有研究表明[13]，今天的工程师在测试基于构件的软件及其重用构件（包括COTS组件）时遇到了许多新问题。其中一个问题是可重用构件的可测试性差。这不仅增加了构件用户的验证成本，而且在构件验证中也造成了很大的困难。

什么是软件可测试性？根据IEEE标准，“可测试性”一词是指“the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met.”。这个定义表明软件可测试性是一个可衡量的质量指标，可用于衡量测试的难易程度，用于测试人员在测试过程中实现特定的测试目标，例如测试覆盖率标准，以及对给定软件进行测试操作的容易程度。也可以理解为一个软件工件（软件系统、构件、需求文件或设计文件等）在给定的测试环境下，可支援测试的程度。

概括地说，软件可测试性是六个因素[15]的结果：（1）表示的特征（例如，需求），（2）实现的特征;（3）内置测试功能；（4）测试套件（测试用例）；（5）测试支持环境；（6）进行测试的软件过程。

为了预测和提高软件可测试性，为企业和测试工程师节省大量的时间和金钱，过去几十年来，研究人员和从业人员已经提出了大量的技术，指标和框架，这些技术解决了软件开发生命周期（需求，设计，编码，测试，运维）中的各个阶段的可测试性分析。这使得审查和概述构件可测试性领域的整个最新技术和实践状态变得具有挑战性。此外，我们观察到没有关于如何衡量构件或系统可测试性以及如何处理问题的明确指导。特别是在我们的工业软件测试项目以及过去15年中与行业的互动中[16-20]，我们发现许多公司都在努力预测和改进软件可测试性，因为没有对该领域已经存在的内容进行充分的概述和缺乏系统性的通用支持。

为了解决构件的可测试性测量方法，我们调研了最近二十年的软件可测试性研究文献，并给出一些可用于构件的可测试性测量方法或模型。这些方法主要是面向已经开发完成的软件程序。

本文认为，软件可测试性不仅可以衡量测试过程的有效性，还可以衡量软件开发过程的质量。因此，它与测试工作量减少和软件质量直接相关。本文通过引入现有适用于构件可测试性分析的模型和方法来解决构件可测试性问题。工程师可以使用这些模型和方法在构件开发过程中执行构件的可测试性验证和测量。

## 构件可测试性测量

本章节描述了6种主要面向Java构件程序的可测试性测量方法或模型，这些方法或模型指标均为可量化的，可实现自动化的测量。工程师们可以根据自己的项目情况选择合适的构件可测试性测量方法。

关于测量指标的评估，由于现有的软件可测试性研究只是提供了测量方法或模型，并没有提供可测试性程度的标准值，也即是没有提供一个量化的可测指标数值（用于评判软件可测或不可测的数值）。因此，我们认为，本文给出的可测试性方法或模型的指标值可使用对比方法来评判软件的可测试性。换句话说，我们可以使用这些模型或方法计算现有我们认为可测的（或容易测试的）构件或系统，并获取它们的可测试性的指标数值。然后以现有可测试（或容易测试）软件的指标值为标准，对比新的构件或系统的指标值，来评估新的构件或系统的可测试性程度（在构件或系统规模相似的情况下）。

### 面向对象类的可测试性测量模型

Aymen Kout[3]等人提出了代码层面的面向对象类的可测试性测量模型（MTMOOD, Metric-Based Testability Model for Object Oriented Design）。他们通过实证研究证实了面向对象程序类的封装，继承和耦合程度能够影响程序类的可测试性。该模型包括三个面向对象的设计属性：封装（Encapsulation），继承（Inheritance）和耦合（Coupling）。封装被定义为一种抽象，它强制在对象的外部接口与其内部实现之间进行清晰的分离。继承定义为类之间is-a关系的度量。耦合被定义为在设计上对象与其他对象的相互依赖性。

简而言之，Aymen Kout提出了一个面向对象程序类可测试性模型，可以用来评估面向对象程序类的可测试性。该模型是

Testability = -0.08 \* Encapsulation +1.12 \* Inheritance + 0.97 \* Coupling

即，类可测试性MTMOOD = -0.08 \* 封装 +1.12 \* 继承 + 0.97 \* 耦合。

其中，具体的数值是影响可测试性因素的权重，封装被量化为计算类中定义的所有方法的数量，继承被量化为计算类设计中继承树的深度，耦合被量化为计算与类相关的类的数量。这些影响可测试性因素的数据均可使用静态分析源码的手段获取。

根据模型可以算出类的可测试性量化数值（指标），类的可测试性指标值越低，测试该类所需的测试用例数会越少，类的可测试性也就越高。相反，类的可测试性值越高，测试类所需的测试用例数就会越多，类的可测试性也就越低。在这里我们使用实证结果的平均系数值（Spearman相关系数，95％置信水平下存在相关性）来表示指标与测试类（例如JUnit）的关系，如下，

|  |  |  |  |
| --- | --- | --- | --- |
|  | TLOC | TAssert | TNOO |
| MTMOOD | 0.304 | 0.133 | 0.331 |

其中，TLOC是单元测试代码行数，TAssert是单元测试类中断言方法调用的次数，TNOO是测试类中测试方法的数量。

### 基于运行时类信息的可测试性测量方法

Amjed Tahir等人[4]提出了关于类可测试性和运行时属性之间关系的研究。这项工作表明运行时属性动态耦合和关键类（ dynamic coupling and key classes）与类可测试性之间存在正相关的关系，并认为使用动态指标收集的数据比使用静态指标收集的数据更广泛，更精确。

动态耦合（DC, Dynamic Coupling）指标。动态耦合旨在以两种形式进行测量 ，当一个类在运行时被另一个类访问时，以及一个类在运行时访问其他类（即，考虑到调用者和被调用者）。通过测量这些耦合水平，我们选择了先前定义的导入耦合（IC，Import Coupling）和输出耦合（EC, Export Coupling）指标[5]。IC测量系统中类接收的来自其他类的方法调用的数量（即，类调用了其他类的方法数量）。EC测量类发送到系统中的其他类的方法调用的数量（即，类的方法被其他类调用的数量）。话句话说，即IC是类动态调用了多少其它类的方法数量，EC是类的方法被多少其它类动态调用。

关键类（Key Classes）指标。即具有更高执行频率的那些类与类可测试性具有显着关系。我们定义执行频率（EF, Execution Frequency）动态度量以标识那些关键类。类C的EF计算是类中方法的执行次数。考虑类C，方法为 。设为类C方法m的执行次数，然后：

其中n是类C的方法的数量。动态测量数据的采集可以以各种方式完成。最常见（也是最准确）的方法是通过在软件执行期间使用动态分析技术获取跟踪信息来收集数据。这种方法在本研究中得到了实现，并通过使用AspectJ框架收集指标来实现，AspectJ框架是面向切面编程（AOP）的成熟Java实现。

实证研究结果表明，动态耦合和关键类指标与单元测试用例之间存在正相关关系，动态耦合和关键类在某种程度上可以作为类可测试性的补充指标。这里讨论的是，紧密耦合或频繁执行的类需要大的相应测试类，即，需要更高数量的测试代码行数（TLOC, Test Lines of Code）和测试用例（例如JUnit）数量（NTC, Number of Test Cases）。在这里我们使用实证结果的平均系数值（Spearman相关系数，95％置信水平下存在相关性）来表示指标与测试用例的关系，如下，

， ， 。

### 基于控制流的可测试性测量模型

Mourad Badri等人[6]进行了一个实证分析，表明控制流依赖性对类的可测试性具有一定的影响。他们在类的单元级测试(JUnit)中提供了证据表明控制流依赖性和类的可测试性之间存在着显著关系。

此方法引入了一个指标，质量保证指标（Qi，Quality Assurance Indicator），用以测量类的可测试性程度。Qi度量基于控制调用图，它是传统控制流图的简化形式。控制调用图是控制流图，从该控制流图中移除语句不包含对方法的调用的指令或顺序指令的基本块的节点。Qi度量值标准化在区间[0,1]中给出值。类的Qi的低值反映了类（是高风险类）并且需要更多测试工作来确保其质量，而高值表示（类是低风险类）在类上有效投入的测试工作量很高（与其复杂程度成正比）。一个类的Qi依赖于其协作类的Qi（被调用的类）。如下图Fig 1，给出了一种方法及其相应的控制调用示例图。

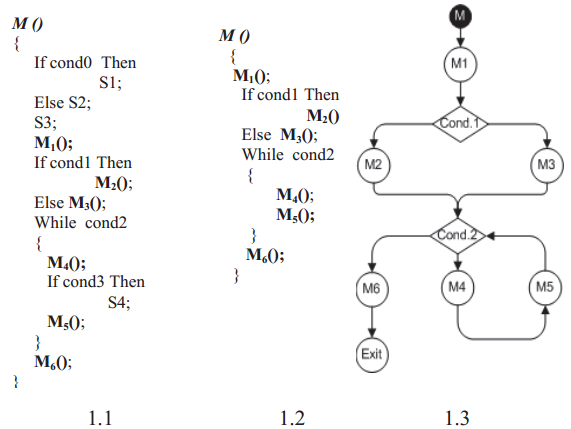


Fig 一种方法及其相应的控制调用图

让我们考虑图Fig 1中子图1.1给出的方法M的例子。Si表示不包含对方法的调用的指令集合。子图1.2给出了减少语句后的方法M的控制调用流图，即从方法M的原始代码中删除了不包含对方法的调用的指令（指令集）。子图1.3给出了相应的控制调用图。与传统的调用图不同，控制调用图是更精确的模型。它们抓取了和控制相关的调用结构。

质量保证指标。我们将方法Mi的Qi定义为一种估算控制流将通过该方法而没有任何失败的概率。它可以被视为与方法（以及高级别的类）相关的风险指标。事实上，方法Mi的Qi取决于方法本身的各种内在特征，例如其单元测试覆盖率（实际投入到方法上的测试工作量）及其圈复杂度，以及所有被Mi调用的方法的Qi。事实上，我们假设方法的质量，特别是在可靠性方面，也取决于它与执行任务的方法的质量。在面向对象系统中（OOS）中，对象协作以实现各自的职责。质量差（低测试）的方法可能（直接或间接）对使用它的方法产生负面影响。这里存在一种传播，取决于系统中控制流的分布，以及那些需要被抓取的方法调用。

特别是在大而复杂的OOS的情况下，直观地识别类之间的这种类型的干扰是不明显的。方法Mi的Qi由下式给出：

其中，

分配概率，即路径可能被执行的概率P的计算。方法的控制调用图可以看作控制流通过的一组路径。事实上，通过特定路径取决于控制结构中的条件状态。为了抓取控制流的这种概率特性，我们为控制调用图的每个路径分配一个概率，如下所示：

其中是组成路径的有向弧。通过假设，为了简化分析和计算，控制结构中的条件是独立的，为退出控制结构时有向弧覆盖的概率，也即是控制语句指向下一个语句的弧的概率。然后将简化为控制结构中条件状态概率的乘积。为了便于计算，我们可根据图表Fig 2中给出的规则将概率分配给Java程序的不同控制结构。在生成Qi模型时，在程序源代码的静态分析期间自动分配这些值。作为替代方式，概率值也可以通过动态分析获得，或者由程序员分配（知道代码的情况下）。例如，对于if条件语句，条件不论为真还是假，退出控制结构时，指向下一个语句（或指令）的弧的概率均为0.5。

内在质量保证指标。方法Mi的内在质量保证指标，标记为。其取决于其圈复杂度以及其单元测试覆盖率（作为测试工作的指标）。它可以用如下等式表示，

其中： ，为方法的圈复杂度，以及为方法的单元测试覆盖率,，。

圈复杂度可以帮助软件工程师确定程序的固有风险。一些研究提供了经验证据，证明圈复杂度与容易出错之间存在显着关系[7][8]。圈复杂度也被认为是可测试性的良好指标[9][10]。程序的圈复杂度越高，其测试工作量就越高。测试活动将降低复杂程序的风险并实现其质量。此外，测试覆盖率提供了关于测试过程有效性的客观测量。

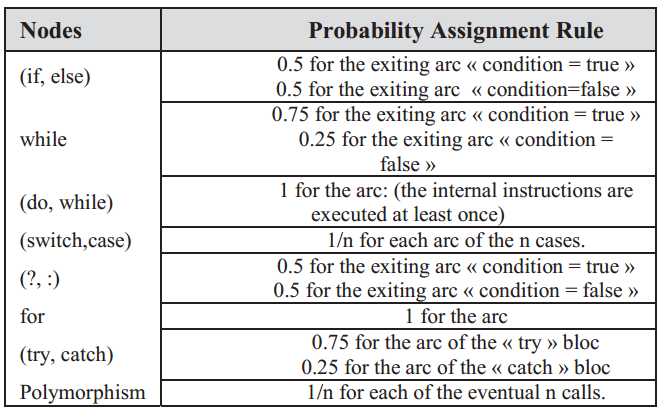


Fig 概率的分配规则

我们评估了Qi和Q\*指标与单元测试用例（JUnit）指标的相关性：TLOC、TAssert、THEff（the effort necessary to implement or understand a test class）。分别是测试一个类需要的测试代码行数，测试用例调用的断言方法数量，实现或理解一个测试类需要花费的工作量。

在这里我们使用实证结果的平均相关系数（Spearman相关系数，95％置信水平下存在相关性）来表示指标与测试用例指标的相关系，如下，

|  |  |  |  |
| --- | --- | --- | --- |
|  | TLOC | TAssert | THEff |
| Qi | -0.493 | -0.346 | -0.401 |
| Q\* | -0.517 | -0.426 | -0.481 |

正如上表所示，Qi与测试用例相关指标负相关，也正如之前所言，Qi的值越高，类可测试性也就越高，相反，Qi的值越低，类的可测试性越低。

### 面向程序包级别的可测试性测量方法

Y Singh等人[11]以Eclipse项目为例，提出了面向对象程序包（package）级别的可测试性指标，比Magiel Bruntink[12]早前提出的可测试性指标更加详细。他评估了14个能够影响程序包级别的可测试性程度的源码属性（指标），并推导出相关源码属性与单元级测试用例（JUnit）的Spearman相关系数（置信度为95%）。

Table 1展示了具有影响可测试性程度的面向对象程序的指标集。这些指标在类级别（Class level）计算，然后在包级别（Package level）计算并分析（每个指标的包级别计算方法参照Table 1的第三列）。这些指标通过大尺寸（size）、耦合（coupling）、内聚（cohesion）、继承（inheritance）和多态（polymorphism）将指标划分为5类。

Table 源码指标集

|  |  |  |
| --- | --- | --- |
| Level | Metric name | Package level |
| Class | Line of code per class(LOC) | Max of class level metric |
| No. of Attributes per Class(NOA) | Total of class level metric |
| No. of Methods per Class(NOM) |
| Weighted Methods per Class(WMC) |
| Response for Class(RFC) |
| Coupling between Object(CBO) |
| Data Abstraction Coupling (DAC) |
| Message Passing Cohesion (MPC) |
| Tight Class Cohesion (TCC) |
| Information based Cohesion (ICH) |
| Lack of Cohesion (LCOM) |
| Depth of Inheritance (DIT) |
| No. of Children (NOC) |
| No. of Methods Overridden by a subclass (NMO) |
| Class | Test Lines of code per Class (TLOC) | Total of class level metric |
| Test methods per class (TM) |
| Test Cases per class (TA) |
| Package | NClass (No. of Classes) | Value |
| NTClass (No. of Test Classes) |

尺寸指标（Size Metrics）。这些指标通过类中代码的行数，属性数量，方法数量来度量系统的大小。由于这些指标抓取了类的复杂性，因此它们可以被用来评估类的可测试性。

Lines of Code per Class (LOC)，它计算类中的代码总行数（不包含空白行和注释行）。

Number of Attributes per Class (NOA)，它计算每个类的属性（字段）数量。

Number of Methods per Class (NOM)，它计算类中的方法（函数）数量。

Weighted Methods per Class (WMC)，它计算类中所有方法圈复杂度的总和。

Number of Source Classes per package (NSClass)，它计算给定包中Java类的数量。

内聚指标（Cohesion Metrics）。内聚是用来衡量一个模块(方法)中元素的功能相关程度。一个强的内聚模块很少或不与其它模块交互并且只实现软件的一个功能特性。这里考虑如下三个内聚指标。

Lack of Cohesion in Methods (LCOM)，它用来衡量类的内聚性，它被定义为，设M为类的方法集合，A为类的属性集合。是访问了属性a的方法数量。Mean是对A中属性访问的平均值。即Mean是A中平均一个属性的方法访问数量。LCOM可以使用如下表示，LCOM = (Mean -|M|)/(1-|M|)。

Information Flow Based Cohesion (ICH)，ICH被定义为同一个类中方法被其它方法调用的数量，使用被调用方法的参数数量作为方法的权值。

Tight Class Cohesion (TCC)，度量TCC定义为具有共同属性用法的类的公共方法对的百分比。

耦合指标（Coupling Metrics）。耦合关系增加复杂度并降低封装性。

Coupling Between Objects (CBO)，类的CBO即是计算类中使用其它类的数量。两个类时耦合的，当一个类的方法使用了另一个类的方法或实例变量。

Data Abstraction Coupling (DAC)，在这里被定义为类中定义的抽象类型数据（Abstract Data Types, ADT）的数量。

Message Passing Coupling (MPC)，即是类中发送语句（Send Statements）的数量。如果在一个类C中的两个方法同时访问类D中同一个方法，则MPC=2。

Response for a Class (RFC)，RFC被定义为可以响应该类对象接收的消息而执行的一组方法。

继承指标（Inheritance Metrics）。这里将讨论两个不同的继承指标。

Depth of Inheritance Tree (DIT)，继承层次结构中类的深度是从类节点到树根的最大步数，并由祖先类的数量来衡量。

Number of Children (NOC)，在类继承层次结构中，类直接的孩子的数量。

多态指标（Polymorphism Metrics）。多态是面向对象软件的特性，即是类重新实现了父类包含的方法。

Number of Methods Overridden by a Subclass (NMO)，当子类和父类有相同签名的方法时，这个在父类中的方法被称作为被子类重写的方法。

测试指标（Test Metrics）。这里使用4个测试指标来衡量软件可测试性程度。它们是TLOC (Lines of Code for Test Class)，TM（Number of Test Methods），TA(Number of asserts), 和NTClass（Number of test classes per test package）。这些指标是单元级测试用例的属性。前3个指标是类级别指标。它们最后被用于包级别的可测试性分析。更具体的如下。

TLOC（Test Lines of Code），它被定义为在JUnit测试类中的代码行数（不包括空白行和注释行）。

TM（Number of Test Methods），它被定义为在JUnit测试类中的测试方法数量。

TA（Number of Asserts），它被定义为测试类中调用的断言（Assert）数量。

NTClass（Number of Test Classes），它被应以为测试包中的测试类的数量。

上述三种测试指标（TLOC，TM，和TA）首先需要在类级别计算，然后在包级别计算。

Table 2展示了面向对象软件的属性和测试用例属性之间的相关系数（Spearman相关系数，95％置信水平下存在相关性），根据这些可影响软件可测试性的指标可以衡量软件的可测试程度。表中的系数值均为正数，意味着源码指标与测试用例指标呈正相关，也即是源码属性指标值越大，测试这个源码所需要的JUnit测试用例就越多，测试工作量也就越大，进而软件的可测试性就越低。

Table 指标关系系数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sum | TLOC | TM | TA | NTClass |
| LOC | 0.515 | 0.408 | 0.439 | 0.440 |
| NOA | 0.502 | 0.425 | 0.478 | 0.432 |
| NOM | 0.566 | 0.465 | 0.505 | 0.453 |
| WMC | 0.548 | 0.450 | 0.482 | 0.448 |
| LCOM | 0.602 | 0.494 | 0.586 | 0.448 |
| ICH | 0.601 | 0.457 | 0.486 | 0.540 |
| MPC | 0.535 | 0.455 | 0.496 | 0.554 |
| TCC | 0.586 | 0.508 | 0.495 | 0.641 |
| CBO | 0.500 | 0.421 | 0.447 | 0.586 |
| RFC | 0.597 | 0.504 | 0.531 | 0.533 |
| DAC | 0.419 | 0.327 | 0.352 | 0.478 |
| DIT | 0.485 | 0.428 | 0.458 | 0.450 |
| NOC | 0.511 | 0.448 | 0.447 | 0.528 |
| NMO | 0.631 | 0.553 | 0.557 | 0.505 |

### 面向构件运行时的可测试性测量方法

本节描述了构件的运行时可测试性测量方法[1]。

我们知道运行时测试会以意想不到的方式干扰系统状态或资源可用性，因为系统的生产状态和数据将与测试混合。更糟糕的是，测试操作可能触发系统边界之外的事件，可能以难以控制或无法恢复的关键方式影响系统环境，例如在测试部分战斗系统时发射导弹。

运行时测试中存在干扰的事实需要一个指示器来指示系统相对于运行时测试的弹性有多大，或者换句话说，运行系统上的测试可能导致的不利影响。 IEEE的可测试性标准定义可以改写为反映这些要求，如下：

运行时可测试性定义：运行时可测试性是（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。

下面我们将介绍影响运行时可测试性的主要因素。

测试灵敏度（Test Sensitivity），它表征了作为测试的一部分执行的哪些操作以不可接受的方式干扰正在运行的系统或其环境的状态。在本节中，我们将描述影响组件测试灵敏度的四个主要因素：具有内部状态的组件，组件的内部/外部交互，资源限制和系统可用性。

组件状态（Component State）。知道组件是否具有某种外部状态（即输入的结果不仅取决于输入本身的值，而且还取决于过去输入的值）是测试灵敏度的重要因素。在传统的“离线”测试中，这很重要，因为调用顺序会对测试的预期结果产生影响。在运行时测试的情况下，了解组件是否具有状态非常重要，原因有两个。首先，因为如果处理不当，运行时测试的结果将受到系统状态的影响，其次，因为系统的状态可能会随着测试调用的结果而改变。

组件交互（Component Interactions）。在许多情况下，组件将使用系统中的其他组件，或与系统边界之外的外部参与者交互。这些交互可能启动其他交互等。这意味着组件的运行时可测试性取决于它在测试期间与之交互的组件的运行时可测试性。

所有这些交互都可能通过改变协作中任何组件的状态来干扰正在运行的系统的状态。在某些情况下，这些交互中的一些将跨越系统的边界并影响其他系统的状态，这可能难以预防和修复。在最坏的情况下，相互作用将通过发送一些能够实现无法撤消的物理输出的输出来达到“外部世界”，例如发射导弹。

资源限制（Resource Limitations）。之前的两个灵敏度因素主要影响系统的功能要求。但是，运行时测试也可能会影响非功能性需求。由于运行时测试将在正在运行的系统上执行，因此这些测试的负载将被添加到由系统正常运行引起的负载中。在某些情况下，它将超过系统的可用资源，例如处理器或内存使用，时序限制，超出功耗限制。

可用性（Availability）。要进行测试的系统的可用性要求也是一个因素。存在两种可能性：如果组件仅用于测试目的（独占使用），或者用于测试和正常服务（共享使用）。在共享配置中，可以做出两个区别：阻塞和非阻塞。第一个意味着在执行测试时生产操作将被阻止或拒绝，从而削弱组件提供的服务的可用性。如果组件具有高可用性要求，则无法在此情况下执行运行时测试。在第二种情况下，测试调用可以与生产调用交错，并且组件能够区分测试和生产请求。

运行时可测试性度量。

最终，阻碍运行时测试的所有测试灵敏度因素将阻止测试工程师评估某些特性或需求，否则这些特性或需求可在无限资源的理想条件下执行并完全控制运行系统。这是测量标准使用的主要思想，以获得系统的运行时可测性测量（RTM）的数值测量。

设M\*是我们想要测试的所有那些特征或需求的数量，并且Mr是相同的测量，但是减少到可以在运行时测试的特征或需求的实际数量，即Mr ≤ M\*。系统的运行时可测性测量（RTM，Runtime Testability Measurement）定义为M \*和Mr之间的商

更具体的，我们需要量化Mr和M\*。

本测量标准引入组件交互图（CIG，Component Interaction Graph）作为量化基础，即组件的接口调用图。CIG被定义为有向图CIG =（V，E）。顶点集V =Vp U Vr，由提供的和所需顶点的集合形成，其中每个顶点表示某个组件的接口的方法。E中的边是从顶点创建的，这些顶点对应于用于组件间依赖性的接口所提供的顶点的所需接口，和从提供到用于组件内依赖性的所需接口。即Vp代表组件提供的接口，Vr代表组件需要（或依赖）的接口，E代表接口间的依赖关系。

其中，每个顶点都用测试信息进行注释，这意味着在执行运行时测试时是否可以遍历这样的顶点，如下所示：

(2)

其中，如果该接口可以被运行时调用，也即是该接口运行时测试不会对系统产生影响，标记为0，否则标记为∞。

接口和边信息可以通过对组件源代码的静态分析，或通过提供某种模型来获得，例如状态或序列图。对于某些顶点无法进行信息注释的情况，应采取保守的方法，为其分配无限权重∞。

覆盖标准（Coverage Criteria）。我们将根据现有的准则中提出的两个充分性标准[2]来衡量系统的运行可测性：

1）全顶点充分性标准要求在组件的所有提供的和所需的接口中执行每个方法，这转换为遍历我们模型的顶点vi∈V，至少一次。

2）全上下文依赖标准要求在每个可能的上下文之间测试顶点的调用。如果存在来自vi的到达vj的调用序列，则顶点vj是依赖于vi的上下文。对于这些依赖性中的每一个，所有可能的路径（vi，vi + 1，...，vj）被认为是可行的，并且需要进行测试。

RTM的值(Value of RTM)。

我们将假设交互开始于我们想要覆盖的顶点（对于所有顶点覆盖）或路径的第一个顶点（对于所有上下文依赖性覆盖）。因为CIG中的边表示可能发生或不发生的交互（没有任何控制流信息），我们不能假设在尝试覆盖路径时，只会遍历路径中的顶点。在最坏的情况下，交互可以通过从交互开始的顶点可到达的所有顶点传播。

因此，为了估计覆盖从顶点vi开始的顶点或vi上下文相关路径的最坏情况惩罚，计算必须考虑从vi可到达的所有顶点，我们将其表示为Pvi。

对于我们想要覆盖的每个顶点vi或路径（vi，vj，vk，...），我们计算一个损失值T（vi），类似于单个顶点的损失值：

通过将那些特征为T（vi）≠∞的特征考虑为可测试，可以分别为所有顶点和全上下文依赖覆盖重写等式(1)，

### 面向开发过程中构件的可测试性测量模型

本方法提出了一个组件可测试性模型[14]，以便在组件开发过程中进行可测试性验证和测量。与现有的研究工作不同，这里的目的是开发一个组件可测试性模型，以帮助工程师测量构件的构造程度，以便于从需求到测试的构件验证。

本方法是一个需要人主观参与评判的一种用于可测性测量的五角星模型，见图Fig 3。

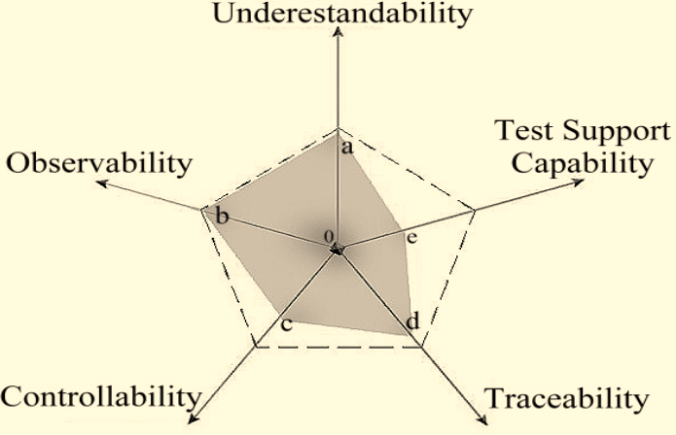


Fig 可测试性五角星模型

本方法认为，软件可测试性主要与以下5种因素有关，可理解性（Understandability），可观察性（Observability），可控制性（Controllability），可追踪性（Traceability）和测试支持能力（Test Support Capability）。具体的解释见下文，详细解释请参考引用论文[14]。

1）组件可理解性（Understandability）是组件可测试性的五个因素之一。它指的是指定和设计组件的程度，以便于组件用户和工程师的理解，以便他们可以轻松定义组件测试和组件验证标准。此因素侧重于生成的组件工件的质量（例如需求规范，API规范和用户参考手册），以了解它们的生成情况，以便于用户和工程师进行组件验证。它可以进一步细化为以下五个因素：

* 文档可用性，指的是组件工件的可用性，包括需求规范，API规范和用户参考手册。
* 文档可读性，指的是编写给定组件规范文档的程度，以增强对组件用户和测试工程师的理解。
* 需求可测试性，指的是生成组件需求和API接口的程度，以确保它们是可测试的。在这里，组件要求可以分为非功能要求和功能要求。
* 需求可测量性，指的是生成组件需求和API接口的程度，以确保它们是可测量的。
* 组件可用性，指的是生成的组件用户手册，API规范对用户操作的简便程度。

2）组件可观察性（Observability）是组件可测试性的第二个因素。它指的是组件的设计程度，以便于监视和观察组件功能和组件测试的行为。为了使工程师能够验证和评估组件的可观察性，我们基于以下四个因素进一步完善它:

* GUI可观察性: 对于具有GUI界面的组件，它指的是组件支持观察其与用户界面中的用户交互的能力。它可以从三个方面进行检查和测量：用户输入，GUI事件和用户输出。
* 功能可观察性: 指在测试期间支持其功能以观察其输入和相应输出的组件能力。它可以进一步细化到两个层次：

1. API函数可观察性，仅关注API函数的域函数可观察性的验证和测量。
2. All-function observability，它指的是组件中定义的所有域的功能可观察性。

* 外部交互可观察性: 这是指支持与同一计算机中其他组件的外部交互观察的组件功能。
* 通信可观察性； 对于支持基于消息的通信的组件，它指的是组件支持观察传入消息和相应传出消息的能力。

3）组件可控性（Controllability）是组件可测试性的第三个因素。它指的是组件的设计程度，以便在验证期间控制其执行。可以将组件可控性进一步细化为以下五个因素

* 组件环境控制：它指的是支持组件环境安装，配置设置和部署的内置组件功能。
* 基于组件状态的行为控制：这是指内置功能，有助于控制基于组件状态的行为，例如重置组件状态，状态控制功能和转换控制功能。
* 组件执行控制：这是指内置组件执行控制功能，可以在不同模式下执行组件，例如测试模式，正常功能模式，控制模式等。通过此功能，用户和测试人员可以根据需要启动，重新启动，停止，暂停和中止程序。
* 组件测试控制：这是指内置功能，便于组件验证的组件控制。它包括：a）设置元数据，b）调用函数，c）激发事件，或d）触发传入消息。
* 组件域功能控制：这是指内置功能有助于组件功能的可控性。它可以分为两个不同的级别：a）所有已定义函数的全功能可控性，b）API函数可控性，仅涉及其API接口中定义的函数。

可能需要，设计和实施所有这些类型的组件控制能力以增强组件可控性。在组件开发过程的所有阶段，我们可以通过软件审查和检查来检查这些因素，以了解需要，设计和实施哪些类型的可控性。可以很容易地根据三个级别检查控制功能：自动，半自动和手动。要实现自动级别，给定组件必须将选择可控性列为内置功能要求的一部分。此外，必须根据给定的可控性要求执行其设计，实现和验证。

4）组件可跟踪性（Traceability）是组件可测试性的第四个因素。它指的是表示组件开发的程度，以便于监控不同类型的程序行为。可以从两个角度进一步检查：a）内置程序跟踪功能，b）不同类型程序可跟踪性的完整性。常见的需要追踪的是系统错误追踪，事件追踪，状态追踪，性能追踪，存储追踪，操作追踪等。

5) 组件测试支持功能（Test Support Capability）是组件可测试性的最后一个因素。与前四个不同，它只能在组件测试过程中进行验证和测量，并侧重于组件验证期间的测试操作支持功能。它可以进一步细化为四种类型。它们是：

i）测试生成：它指的是使用系统测试生成方法和工具生成组件测试和测试脚本的程度。有两种类型的测试生成：白盒和黑盒测试生成。白盒测试生成可以分为五种类型：基于路径，基于数据流，基于分支，基于语法和基于状态。黑盒测试生成也可以分为五种类型：基于边界值，基于等价划分，基于图形，基于随机和基于需求。

ii) 测试管理能力：它指的是提供系统解决方案以支持各种类型测试信息的管理的程度。有三种常见的测试管理，即问题，测试和套件管理。它们中的每一个都可以在三个级别进行评估：系统化工具，无系统化工具或临时工具。

iii) 测试覆盖率分析：它指的是可以根据选定的测试标准轻松测量，监控和报告组件测试覆盖范围的程度。为了提高这种能力，工程师需要两件事；第一个是一组明确定义的组件覆盖标准和标准，另一个是测试覆盖率分析功能，其中包含提供测量，监控和报告的测试工具。

vi) 组件测试执行能力：它指的是如何轻松执行组件测试。此功能对于用户的组件测试和评估至关重要。

可测试性测量模型。在图Fig 3中，字母a，b，c，d和e分别用于表示组件可测试性的五个因素，当测量每个因子时，可以在下面计算组件可测试性：

其中，在这里假设每个因子的测量结果是从0到1的值。值“1”表示每个因子的最大值，“0”表示最小值。五角星的面积用作组件可测试性的度量。显然，这个五角星区域的最小值是0，最大值大约是2.4。由于五角星由五个三角形组成。每个三角形的面积可以计算为：其中，表示三角形的边，并且表示两边之间的72度角。

这里给一个可测试性计算例子：

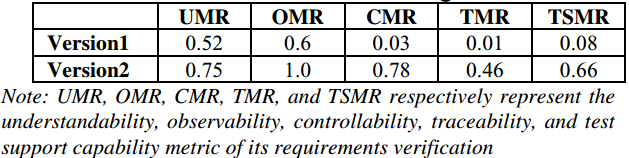


Fig 评估案例

图Fig 4 给出了一个程序的两个版本数据，其中UMR代表可理解性的得分，OMR代表可观察性的得分，CMR代表可控制性的得分，TMR代表可追踪性的得分，TSMP代表测试支持能力的得分（指标的取值范围在[0~1]）。这些得分主要依据工程师根据上述五种影响可测试性因子的影响因素来评估。

对五个因子应用可测性五角星模型，我们得到以下结果：

可测性（版本1）= 0.48（（0.52 \* 0.6）+（0.6 \* 0.03）+（0.52 \* 0.01）+（0.01 \* 0.08）+（0.03 \* 0.08））= 0.16

可测性（版本2）= 0.48 \*（0.75 \* 1 + 1 \* 0.78 + 0.75 \* 0.46 + 0.46 \* 0.66 + 0.78 \* 0.66）= 1.29

可以看出版本2的可测试性相比版本1有了很大的提升。

## 参考文献

[1] González A, Piel É, Gross H G. A model for the measurement of the runtime testability of component-based systems[C]//2009 International Conference on Software Testing, Verification, and Validation Workshops. IEEE, 2009: 19-28.

[2] Wu Y, Pan D, Chen M H. Techniques for testing component-based software[C]//Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems. IEEE, 2001: 222-232.

[3] Kout A, Toure F, Badri M. An empirical analysis of a testability model for object-oriented programs[J]. ACM SIGSOFT Software Engineering Notes, 2011, 36(4): 1-5.

[4] Tahir A, MacDonell S, Buchan J. A Study of the Relationship Between Class Testability and Runtime Properties[C]//International Conference on Evaluation of Novel Approaches to Software Engineering. Springer, Cham, 2014: 63-78.

[5] Arisholm E, Briand L C, Foyen A. Dynamic coupling measurement for object-oriented software[J]. IEEE Transactions on software engineering, 2004, 30(8): 491-506.

[6] Badri M, Toure F. Empirical Analysis for Investigating the Effect of Control Flow Dependencies on Testability of Classes[C]//SEKE. 2011: 475-480.

[7] Aggarwal K K, Singh Y, Kaur A, et al. Empirical analysis for investigating the effect of object‐oriented metrics on fault proneness: a replicated case study[J]. Software process: Improvement and practice, 2009, 14(1): 39-62.

[8] Zhou Y, Leung H. Empirical analysis of object-oriented design metrics for predicting high and low severity faults[J]. IEEE Transactions on software engineering, 2006, 32(10): 771-789.

[9] Bruntink M, Van Deursen A. Predicting class testability using object-oriented metrics[C]//Source Code Analysis and Manipulation, Fourth IEEE International Workshop on. IEEE, 2004: 136-145.

[10] Bruntink M, van Deursen A. An empirical study into class testability[J]. Journal of systems and software, 2006, 79(9): 1219-1232.

[11] Singh Y, Saha A. Predicting testability of eclipse: a case study[J]. Journal of Software Engineering, 2010, 4(2): 122-136.

[12] Bruntink M, van Deursen A. An empirical study into class testability[J]. Journal of systems and software, 2006, 79(9): 1219-1232.

[13] Gao J, Tsao H S J, Wu Y. Testing and quality assurance for component-based software[M]. Artech House, 2003.

[14] Gao J, Shih M C. A component testability model for verification and measurement[C]//29th Annual International Computer Software and Applications Conference (COMPSAC'05). IEEE, 2005, 2: 211-218.

[15] Garousi V, Felderer M, Kılıçaslan F N. A survey on software testability[J]. Information and Software Technology, 2018.

[16] V. Garousi, M.M. Eskandar, K. Herkiloğlu, Industry-academia collaborations in software testing: experience and success stories from Canada and Turkey, Softw. Qual.J. 25 (2016) 1–53. [17] V. Garousi, J. Zhi, A survey of software testing practices in Canada, J. Syst. Softw. 86 (5) (2013) 1354–1376.

[18] V. Garousi, A. Coşkunçay, A.B. Can, O. Demirörs, A survey of software testing practices in Turkey (Türkiye’deki yazılım test uygulamaları anketi), in: Proceedings of the Turkish National Software Engineering Symposium “Ulusal Yazılım Mühendisliği Sempozyumu” (UYMS), 2013. [19] M. Felderer, F. Auer, Software quality assurance during implementation: results of a survey in software houses from Germany, Austria and Switzerland, in: Proceedings of the International Conference on Software Quality, 2017, pp. 87–102. Complexity and Challenges of Software Engineering in Emerging Technologies.

[20] M. Felderer, R. Ramler, A multiple case study on risk-based testing in industry, Int. J. Softw. Tools Technol. Transf. 16 (5) (2014) 609–625.