目录

[1. Requirement 1](#_Toc13165632)

[2. Design 3](#_Toc13165633)

[3. Measurement 5](#_Toc13165634)

[3.1 A Model for the Measurement of the Runtime Testability of Component–Based Systems 5](#_Toc13165635)

[3.2 An Empirical Analysis of a Testability Model for Object-Oriented Programs 13](#_Toc13165636)

[3.3 An Estimation of Software Testability Using Fuzzy Logic 14](#_Toc13165637)

[3.4 A Study of the Relationship Between Class Testability and Runtime Properties 15](#_Toc13165638)

[3.5 Metric Based Testability Model for Object Oriented Design (MTMOOD) 21](#_Toc13165639)

[4. Improving 23](#_Toc13165640)

[5. Other 23](#_Toc13165641)

# Requirement

**令人惊讶的是，很少有研究关注需求可测性问题，尽管它也被IEEE软件需求规范830 1998[1]推荐在实践时需要考虑。例如，七个选定的指标在可读性文献[2]中是众所周知的，并且是：(1) Average Grade Level, (2) Flesch Kincaid Reading Ease, (3) Flesch Kincaid Grade Level, (4) Gunning Fog Score, (5) Simple Measure of Gobbledygook (SMOG Index), (6) Coleman Liau Index, 和(7) Automated Readability Index。 然后，他们使用机器学习和统计分析开发了需求可测试性模型，并评估了是否可以学习需求可测试性模型并将其应用于其他需求。**

**1.1** Measuring design testability of a UML class diagram

它提出了一种测量需求质量以预测可测试性的方法。 作者选择了一组文档可读性指标，用于表征需求的可理解性和质量，并评估它们是否表征了可测试的需求。

具体的，该文献描述了测量UML类图的设计可测试性方法（针对面向对象系统的UML类图）。

为了指导测试任务，并且类图具有继承和动态绑定关系，因此类图可作为检测和掌握广泛的隐式控制依赖关系的良好基础。

**1.2** Requirement Decomposition and Testability in Development of Safety-Critical Automotive Components

它侧重于安全关键汽车软件背景下的需求可测试性。 作者将需求Ri定义为逻辑表达式Li：<Object X>shall<Action Y>[apply to]<Subject Z>。需求被映射到对象X上，对象X将动作Y应用到对象Z上。 Ri的可测试性是Ri的一个属性，它可以验证该逻辑表达式Li。 他们建议，为了实现可测试性，需求必须包括Object，Action和Subject以及Object，Action和Subject必须在系统内可识别并存在。 在这些条件有效的情况下，可以验证该需求，并且需求是可测试的。

* 1. Sizing Software with Testable Requirements

标题为“具有可测试需求的大小调整软件”是另一项有趣的研究，由一名从业者撰写。 该工作提出了可测试需求作为一种新的软件测量范例。 通过在工业环境中应用它，该论文指出，可测试需求是一种直观，灵活的措施，是向用户和管理层传达问题的有用工具。

**1.4** Testable Requirements for Offshore Outsourcing

它考虑了离岸外包（全球软件工程）背景下的需求可测试性。 作者认为，离岸外包需要一套可测试的需求，这是承包商和客户之间法律协议质量保证的核心。 他们为该环境中可测试需求的模型提供了语义。

* 1. Testable Use Cases in the Abstract State Machine Language

它通过使用抽象状态机语言（ASML）指定它们来检查可测试性用例，ASML是Microsoft Research开发的可执行规范语言。 然后，作者通过描述如何从ASML中的用例规范生成测试用例和测试oracles来演示该方法的优点

* 1. Using the Testability Analysis Methodology for the Validation of AIRBUS Systems

它提供了Airbus系统验证需求可测试性的案例研究。 实验表明，可测试性分析可以简化系统验证活动。

# Design

**设计一个便于测试的系统非常重要。 可测试性设计（DfT）通常在SDLC（Software Development Life Cycle）的设计和编码阶段进行。**

**2.1** Improving Web Application Testing Using Testability Measures

糟糕的设计（例如，陷入设计反模式的陷阱）可能会对测试阶段的系统或其单元的可测试性产生负面影响。

**2.2** Quantitative Analysis of Testability Anti-patterns on Open Source Java Applications

可测试性反模式也是另一项工作的焦点，它将可测试性反模式定义为“已知使测试变得困难和/或增加要执行的测试用例数量的设计方法”。

**2.3** A Study of the Relationship between System Testability and Modularity

它的作者使用system-dynamics建模来研究模块性和可测试性之间的关系。 作者关注模块化的成本和收益相关的可测试性。 使用来自system-dynamics的模拟数据，作者发现采用适当的模块化和适当的测试架构可以提高系统测试的效率和有效性。

**2.4** A Study on Design for Testability in Component-Based Embedded Software

它在欧洲电信行业的两家大型公司的基于组件的嵌入式软件环境中报告了一项关于DfT的研究。 根据访谈和技术文档，讨论了不同DfT方法（侧重于可观察性和可控性）的差异和益处。 该论文提出了几个关于DfT的建议，例如，（1）特别是在嵌入式系统的情况下，良好的主机测试环境能够实现高可测试性。 当此环境尽可能与目标系统匹配时，可以进行有效的主机测试; （2）在系统中包含测试支持功能可以进行更有效的测试，包括分析长时间运行的测试和部署的系统。

**2.5** Design for Testability

由从业者撰写的论文为DfT提供了实用的建议。 根据作者的经验，它提供了用于测试各种软件产品的可测试性功能的示例，包括事件记录，断言，诊断，资源监视，测试点，故障注入钩子和支持软件安装和配置的功能。 它还探讨了可测性问题如何影响GUI测试自动化以及测试人员在缺乏可测试性时必须提供哪些规定。 本文最后讨论了测试人员如何与产品开发人员合作，以获得内置于软件产品中的重要可测试性功能。

**2.6** Design for Testability in Object-Oriented Systems

这是一篇有影响力的1994年论文题为“面向对象系统的可测试性设计”（撰写本文时引用次数为378次），该论文的专家罗伯特·宾德尔（Robert Binder）。 作者介绍了DfT的重要概念和不同架构。 它最后说：“几乎所有用于实现高可测试性的技术和技术都已确立，但需要财务承诺，计划和有意识的努力”。

**2.7** On Building Testable Software Components

它专注于构建可测试的软件组件，并引入了可测试（Java）“bean”的概念，并提出了一种基于可测试体系结构和良好定义的内置接口构建可测试bean的方法。

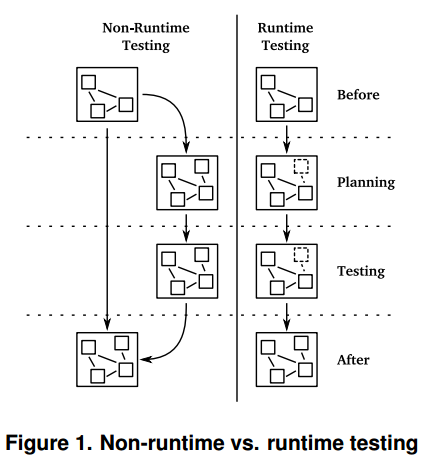
# Measurement

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

在[P10]中提出了运行时可测试性度量。测试基于组件的系统，运行时测试，分为两个阶段：（1）deployment-testing（部署测试）；（2）in-service-testing（服务/运行时测试） 。

具体的，（1）部署时测试的动机是因为系统的许多方面在系统部署到真实环境之前无法验证。 服务或组件集成商无法完全控制他或她正在集成的组件。此外，在许多情况下，这些组件将是远程的，第三方服务，他或她将完全无法控制，更不用说控制访问系统的第二个实例以进行测试。

（2）基于组件的系统（例如系统和面向服务的体系结构）可以具有变化的结构。构成系统的组件可能不可用，甚至不可预先知道。每次添加，删除或更新新组件时，都会有许多测试，其结果将不再有效，必须重新进行。验证。因此，测试的唯一可能性是在系统被修改并且缺少的组件可用之后验证和验证系统。这也与自动管理自主系统相关，它动态地改变可能已经在不可预测的环境中运行的组件的结构或行为。

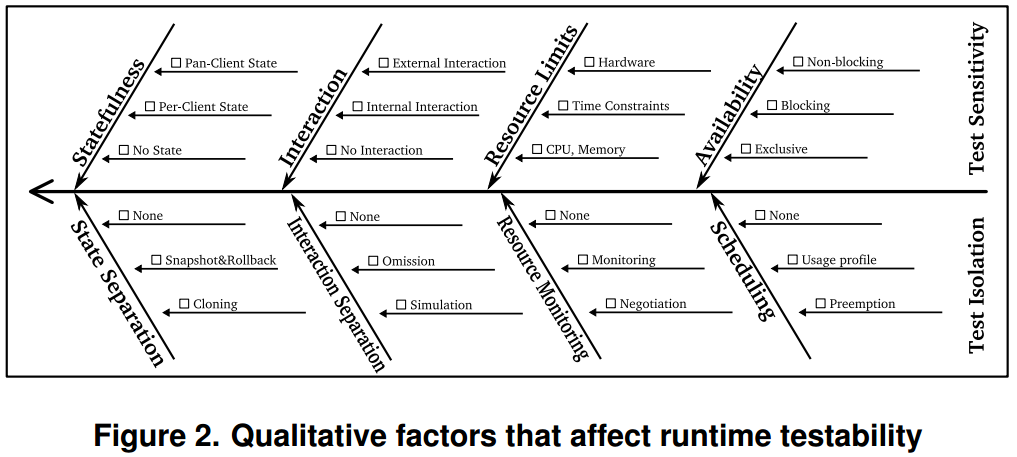


**运行时可测试性：**

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的同时发射导弹。

定义1 运行时可测试性是（1）系统或组件促进运行时测试的程度，而不会受到广泛影响; （2）允许在运行期间执行哪些测试而不会对运行系统产生广泛影响的规范。

该定义考虑了（1）系统的特性和运行时测试所需的额外基础设施，以及（2）从所有可能的测试中识别哪些测试用例。运行时可测性基于两个主要支柱：测试灵敏度，并测试隔离。我们将介绍对它们都有影响的主要因素。图2描绘了它们的鱼骨图。



**一、测试灵敏度**：

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。

1) Component State（组件状态）

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。

2) Component Interactions(组件交互)

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。这些交互有可能启动其他交互，等等。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。

3) Resource Limitations(资源限制)

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。

4) Availability(可用性)

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。

**二、测试隔离：**

测试隔离技术是测试工程师防止测试操作干扰系统的状态或环境，以及系统的生产状态和交互影响测试结果的手段。因此，它们代表了系统通过为其测试灵敏度提供对策来启用运行时测试的能力。 应用测试隔离技术的先决条件是先前对测试灵敏度的研究，以便解决将使运行时测试（部分）失败的运行时可测试性问题。在以下段落中，我们通过在可测试性增益和实现工作方面使用我们的测量，提出了可以在进一步工作中评估的测试隔离技术开发的一些想法。

1. State Separation(状态隔离)

状态分离技术与状态敏感性相对应。它们旨在将组件的使用中状态与测试状态分开。如果该组件具有可观察的状态，Suliman等人。 [21]提出了一个基于三个复杂程度的解决方案。第一级包括在测试时阻止组件操作。第二级建议使用来自运行时环境的specialsupport克隆组件。最后一级依赖于被测组件支持的特殊测试会话，以便在无法轻松克隆的组件上提供状态隔离

2）Interaction Separation（交互隔离）

交互分离可以应用于通过系统传播并影响其他组件的组件交互，特别是系统的外部环境。当交互跨越系统边界时，可以预见两种可能的隔离解决方案：省略输出或模拟。省略输出包括压缩输出，就像从未发生过一样。只有当测试的其余部分不依赖于输出可能产生的影响时，这才有可能。如果期望响应，以输入或外部事件的形式，则外部系统必须由asimulator或模拟组件替换

3）Resource Monitoring（资源监控）

为了防止测试用例耗尽系统资源，可以应用资源监控技术。如果需要比当前可用资源更多的资源，一个简单的监控解决方案就是拒绝或推迟测试用例的执行，例如，如果系统负载超过某个阈值。更高级的可能性是允许组件和测试协商特定测试所需的资源。

4) Scheduling(调度)

可以安排测试以保持组件的可用性，旨在控制方式，数量以及在什么时刻允许在系统中执行测试用例。例如，某些测试用例只会在组件需求较少时执行，类似于[20]中重新配置阶段的建议。如果某个组件被测试阻塞，但是某个服务操作无论如何都无法保留，那么可以从系统中排除测试用例以满足服务调用

三、**Runtime Testability Measurement（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要求，否则这些特性或要求可在无限资源的理想条件下执行并完全控制运行系统。这是本节中使用的主要思想，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要求的度量，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要求的实际数量，使用Mr≤M\*。系统的运行时可测性测量（RTM）定义为M \*和Mr之间的商



虽然通用，RTM的简单性允许工程师根据他们的特定需求定制它，适用于系统的任何抽象，他们希望通过运行时测试来评估他们的功能。在本文中，我们将根据测试覆盖率进一步实例化RTM，以估计测试工程师在运行时测试条件下能够达到的最大测试数据。这种运行时可测试性测量可用于预测测试覆盖范围内的缺陷，即使在完全没有测试用例的情况下，也可以通过在解决导致无法测试的问题时显示可测试性改进来纠正这种情况。

四、Case Study

在基于双组件的系统上进行运行时可测试性研究：系统取自海上安全和安全域的案例研究，以及机场的无线接入点系统。我们的实验目的是证明我们的模型可以帮助识别不适合运行时测试的系统，并决定如何解决这种情况，最终目标是提高集成系统的质量和可靠性。

基于组件的系统由通过其服务接口绑定在一起的组件形成，可以提供（组件提供服务）或者需要（组件需要其他组件来提供服务）。在测试期间，可以在组件中调用组件的任何服务，但需要付出代价。在运行时可测试性分析的情况下，我们感兴趣的成本是测试调用对运行系统或其环境的影响成本，从第4节中组件的灵敏度因子得出。这些成本可以多个量级呈现（计算）成本，时间或金钱等）。

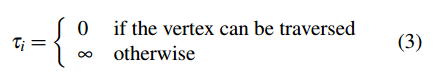
必须避免影响成本过高的操作，将其指定为不可测试的。在本文中，我们将从识别成本来源及其大小的过程中抽象出来，并假设所有操作都具有可测试（无成本）或不可测试（无限成本）。

1. Model of the System（系统模型）

出于本文的目的，系统将使用称为组件交互图（CIG，Component Interaction Graph）的定向组件依赖图进行建模。一方面，它足够详细，可以将关键运行时可测试性问题识别到导致它们的组件的各个操作中。另一方面，它很简单，因此它的推导很容易，并且它的计算是一个容易处理的问题。

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中的边缘是从顶点创建的，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。

每个顶点vi∈V都用测试信息τi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：



边缘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。在没有可用于某些顶点的信息的情况下，应采取保守的方法，为其分配无限权重。

2. Coverage Criteria（覆盖准则）

我们将根据现有的准则中提出的两个充分性标准来衡量系统的运行可测性：

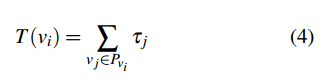
1. 全顶点充分性标准要求在组件的所有提供的和所需的接口中执行每个方法，这转换为遍历我们模型的顶点vi∈V，至少一次。
2. 全上下文依赖标准要求在每个可能的上下文之间测试顶点的调用。如果存在来自vi的到达vj的调用序列，则顶点vj是依赖于vi的上下文。对于这些依赖性中的每一个，所有可能的路径（vi，vi + 1，...，vj）被认为是可行的，并且需要进行测试。

3. Value of RTM(RTM的值)

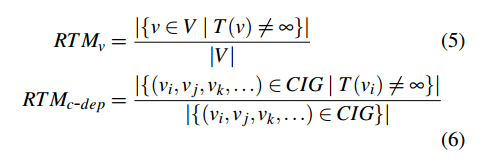
我们将假设交互开始于顶点（对于所有顶点覆盖）或路径的第一个顶点（对于所有上下文依赖性覆盖）我们想要覆盖。因为CIG中的边表示可能发生或不发生的交互（没有任何控制流信息），我们不能假设在尝试覆盖路径时，只会遍历路径中的顶点。在最坏的情况下，交互可以通过从交互开始的顶点可到达的所有顶点传播。

因此，为了估计覆盖从顶点vi开始的顶点vi或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，v j，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2，



### 3.2 An Empirical Analysis of a Testability Model for Object-Oriented Programs

**[P29]报告了用于预测OO类可测试性的缺乏内聚度（LCOM）的经验分析。**

本文提出的研究旨在根据经验分析模型在代码级别预测类的可测试性的能力。我们从单元测试和需要的测试工作的角度研究可测试性。我们使用从两个开源Java软件系统收集的数据设计并进行了一项实证研究，其中存在JUnit测试用例。为了根据所需的测试工作量捕获类的可测试性，我们使用不同的度量来量化相应的JUnit测试用例。为了评估模型预测类的可测性（相应测试类的特征）的能力，我们根据经验调查了模型在代码级（来自软件类的源代码）给出的值与开发的测量特征之间的关系。 JUnittest案例。我们使用相关性进行统计检验。本文提出并讨论了取得的成果。

我们在这项工作中使用了Khan等人提出的面向对象设计的基于度量的可测试性模型（MTMOOD）。 [20]。该模型的开发是为了通过分析软件项目类图中的几个面向对象特征来预测类在设计层的可测试性。该模型包括三个面向对象的设计属性：封装，继承和耦合。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。耦合被定义为对象在设计中的其他对象上的相互依赖性。

每个设计属性的相对重要性是加权的。作者使用多元线性回归分析来获得系数。使用从各种中型工业项目的类图中收集的数据，验证了模型在预测设计可测性方面的有效性。事实上，作者表明，使用MTMOOD模型估计的总体可测性与独立评估者确定的评估具有统计学上显着的相关性。用于评估设计级别的可测试性的所用计算公式为（MTMOOD）：可测性= -0.08 \*封装+1.12 \*继承+ 0.97 \*耦合

为了测量所考虑的设计性能，Khan等人。 [20]使用了以下三个类图级别度量：度量ENM（作为封装度量），它计算类中定义的所有方法的数量，度量REM（作为重用继承度量），它计算类设计中继承树的深度和度量CPM（作为耦合度量），它计算与类相关的类的数量。这些指标是根据类图计算的。

由于我们的工作本质上是探索性的，我们使用以下源代码度量标准将此模型调整到代码级别：NOO [15]它给出了类中的方法数，DIT [8]给出了类的继承树的深度和CBO [8]给出了给定类所耦合的类的数量。下面的等式给出了我们用于评估源代码级别（MTMOOP）的可测试性的计算公式：可测性= -0.08 \* NOO + 1.12 \* DIT + 0.97 \* CBO。

### 3.3 An Estimation of Software Testability Using Fuzzy Logic

**在[P36]中提出了一些与可能影响可测试性的软件开发人员相关的因素和度量，例如，多年的编码经验和类似项目的先前经验开发。**

如IEEE [6]所示，软件可测试性是软件工件（即软件模块，软件系统，设计或要求文档）在给定测试环境中进行测试的程度。在软件工件的高可测性的情况下，通过测试方法找到软件中的错误（如果有的话）将更容易。可测性取决于各种因素，如可控性，可观察性，可分性，关注点分离，可理解性，自动化，异构性。本文的目的是介绍一些影响可测性的其他因素。

1. **Fuzzy Logic**

模糊逻辑（FL）是一种计算方法，它基于大多数计算机所基于的典型0或1，真或假布尔逻辑的真实或意见程度构建。模糊逻辑（FL）是一种与人类推理相似的推理技巧。模糊逻辑的方法模仿人类做出决策的方式，其包括数字值1和0之间的所有过渡选择。

使用模糊逻辑（FL），因为它可能无法提供精确的推理，但提供了可接受的推理。

在本节中，我们介绍了影响面向对象软件可测试性的一些因素。可测性是指代码在随机测试中显示其错误的简单性。可测试性的重要性在以前的每一项工作中得到认可，以及如何在早期工作中得到关注软件开发生命周期的各个阶段。软件是由开发人员的巨大努力开发的，那么他们怎么能不影响可测性。我们提出了可能影响可测试性的严格相关的软件开发人员因素。

•多年的编码经验

•开发时间

•类似项目的先前开发经验

•对主要使用编程语言的信心

•工作的努力程度

### 3.4 A Study of the Relationship Between Class Testability and Runtime Properties

**[P20]报告了一个关于类可测试性和运行时属性之间关系的案例研究，例如动态耦合度量（例如，导入耦合和导出耦合）。该案例研究是在四个开源项目中进行的。通过测试LOC的大小测量可测试性。**

**Abstract:**

已知软件测试昂贵，耗时且具有挑战性。虽然以前的研究调查了几个软件属性和软件可测试性之间的关系，但重点是静态软件属性。在这项工作中，我们提出了对运行时属性（动态耦合和关键类, dynamic coupling and key classes）与类可测试性之间可能关系的实证研究结果。我们使用动态指标测量这两个属性，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。基于统计分析，我们发现动态耦合和关键类与类可测性显着相关。因此，我们建议这些属性可以用作类可测试性的有用指标。

提高软件可测试性应有助于降低测试成本，工作量和资源需求。

如果组件难以测试，那么设计用于测试这些组件的测试用例的大小以及所需的测试工作量必然会更大[-]。当在开发过程的后期检测到问题时，可测试性差的组件也更昂贵。相比之下，具有良好可测试性的组件和软件可以显着提高软件质量并降低测试成本[-]。

大量工作已经解决了可能影响软件产品可测试性的各种设计和代码特征。例如，关系是面向对象（OO）系统中的内部类属性，并且在先前的几项研究中已经研究了相应单元测试的特征（例如，[-,-]）。在这些研究中，已经使用了几个OO设计指标（主要来自C＆K套件[-]）来研究类/系统结构和测试复杂性之间的关系。在这样的研究中，已经发现了几个复杂性和与规模相关的生产代码指标和内部测试代码属性之间的一些强大且重要的关系[-]。

**Testability Concepts:**

**1)** **Dynamic Coupling**

在这项研究中，动态耦合被选为系统特征之一，以测量和研究其与可测试性的关系。耦合已经在以前的工作中显示出对软件质量有直接影响，并且还与复杂性和可维护性的软件质量特征有关[-]。已经表明，在所有其他条件相同的情况下，耦合水平越高，复杂性越大，维护系统的难度越大[-]。这表明期望耦合与可测试性相关是合理的。我们的研究选择了动态而非静态耦合来解决传统静态耦合测量的一些缺点。多年来，基于软件的有限结构特性，已经对静态耦合进行了测量[-]。这会错过运行时在不同级别（类，对象，包等）的不同组件之间的耦合，这应该捕获更完整的图像，因此可测试性更接近。在新的软件工程研究文献中，这种测量动态耦合的概念非常普遍。在我们最近对动态度量的系统映射研究中，动态耦合被发现是最广泛研究的系统特征，用作动态分析的基础[-]。

为了这项工作的目的，遵循[19]采用的方法，并使用捕获对象级别的耦合的动态耦合度量。如果两个物体中至少有一个作用于另一个物体，则它们会被耦合[11]。用于耦合的度量基于运行时方法调用/调用：如果来自类A（调用者）的方法调用来自类B（被调用者）的方法，则反复使用两个类，类A和类B，以进行耦合。 Sect4.2中提供了用于衡量这种耦合形式的具体指标的详细信息。

**2) Key Classes**

本研究中引入了关键类的概念，作为一种新的生产代码属性进行测量，并研究其与类可测性的关系。 OO系统围绕类的组形成，其中一些类链接在一起。随着软件系统规模的扩大，这些系统中使用的类数量也在增加。分析和理解一个程序或系统，它是如何工作的以及它的潜在可能性，重要的是要知道从哪里开始以及哪些方面应该被赋予优先权。从维护的角度来看，理解类的作用及其对系统的相对重要性至关重要。在这方面，有些类可以比其他类更具影响力，扮演更重要的角色。这组类在这里被称为“关键类”。我们将Key Class定义为在系统的典型使用剖面中经常执行的类。

关键类的概念可以在文献的其他地方看到，但在本研究的意义和用法中具有重要的意义。例如，在[24]中，作为关键类的分类基于类的耦合程度。因此，Key Classes是紧密耦合的类。相反，我们的定义基于这些类的用法：关键类是那些在运行时具有高执行频率的类。用于衡量关键类的度量在Sect2.4中进行了解释。这些类的潜在用途之一是确定测试活动的优先级 - 测试人员可以优先考虑他们的工作，首先关注测试这些关键类，同时考虑其他因素，如风险和关键性信息。

**3)Study Design**

我们研究了两个因素，原则上与系统可测试性有关：动态耦合和关键类。为此，我们有两个研究项目要回答：

**RQ1**: 类的动态耦合是否与其相应的测试类/单元的类可测性显着相关？

**RQ2**: 关键类是否与其相应测试类/单元的类可测试性显着相关？

调查以下两个研究假设来回答研究问题：

**H0**: 动态耦合与类可测性具有显着相关性。

**H1**: 关键类与类可测试性有显着的相关性。

相应的空假设是：

**H2**：动态耦合与类可测性度量没有显着的相关性.

**H3**：关键类与类可测性没有显着的相关性。

**3.1)** **Measurements**

**a)** **Dynamic Coupling Metrics**

如Sect 3.1中所述。，动态耦合旨在以两种形式进行测量 - 当一个类在运行时由另一个类访问时，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[19]。 IC测量系统中来自其他类（调用者）的类（被调用者）接收的方法调用的数量（即，一个类调用了其他类的方法数量）。 EC测量从类（调用者）发送到系统中的其他类（被调用者）的方法调用的数量（即，一个类的方法被其他类调用的数量）。请注意，两个度量都是基于方法调用/调用收集的。 [19]中提供了对这些指标的更详细解释。

**b) Key Classes Metrics.**

关键类的概念在Sect3.2中有解释。目标是检查那些关键类（即具有更高执行频率的那些类）是否与类可测试性具有显着关系（如下一小节中所定义）。我们定义执行频率（EF，Execution Frequency）动态度量以标识那些关键类。 C类的EF计算是C类中方法的执行次数。考虑C类，方法为m1，m2，...... mn。设EF（mi）为C类方法m的执行次数，然后：



其中n是C类中执行的方法的数量。

**c) Class Testability Measures.**

这里考虑了类的可测试性。在这项工作中，我们利用两个静态指标来衡量单位测试特征：代码测试行（TLOC）和测试用例数（NTC）。这些指标受[9]建议的测试套件指标的影响。 TLOC源自经典的代码行（LOC）度量，是一种大小度量，用于计算一个或多个测试类中代码的物理行总数。 NTC是一个测试设计指标，用于计算测试类中测试用例的总数。因此，我们的假设反映了一种期望，即生产代码类的动态耦合和关键类与其相关测试类的大小和范围相关。

**4)Data Collection.**

动态度量数据的集合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框架是面向方面编程（AOP）的成熟Java实现。以前的工作（包括[23,27,28]）已经表明，AOP是一种有效且实用的方法，用于客观地收集动态度量数据，因为它可以执行完整的运行时自动源代码检测。

使用CodePro Analytix1工具收集可测试性度量数据，包括LOC，TLOC和类数（NOC），稍后使用Eclipse Metrics Plugin2检查和验证这些值。从JUnit框架收集NTC度量标准的值，并由第一作者手动验证这些值。。。。。。

**5)Disscusion**

**RQ1**: 类的动态耦合是否与其相应的测试类/单元的类可测性显着相关？

**RQ2**: 关键类是否与其相应测试类/单元的类可测试性显着相关？

调查以下两个研究假设来回答研究问题：

**H0**: 动态耦合与类可测性具有显着相关性。 **√**

**H1**: 关键类与类可测试性有显着的相关性。 **√**

相应的空假设是：

**H2**：动态耦合与类可测性度量没有显着的相关性. **×**

**H3**：关键类与类可测性没有显着的相关性。 **×**

在重新审视调查研究问题的列表时，动态耦合被发现具有显着（尽管不是非常强）与可测试性（RQ1）的直接关联。在关键类（即，频繁执行的类）和类可测试性度量之间发现了更显着的相关性。**通过回答RQ1和RQ2，这表明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**（即单元测试大小）。这里争论的是，紧密耦合或频繁执行的类需要大的相应测试类（即，更高数量的TLOC和NTC）。已经发现这种期望在所检查的四种系统中至少有三种中存在。

**6）Conclusion**

在这项工作中，我们开始研究两个运行时代码属性（即动态耦合和键类）之间的任何关联的存在和重要性，以及四个开源OO系统中类的可测试性。测试可测性基于系统的生产类及其相关的单元测试。使用两种不同的度量来测量类可测性，即TLOC和NTC。由于我们对运行时系统特性之间的关系感兴趣，因此使用通过AOP收集的动态软件度量来测量动态耦合和关键类。然后使用Spearman相关系数测试对结果进行统计分析以研究关联。

由此产生的证据表明，动态耦合和内部类可测性之间存在显着关联。我们发现动态耦合度量，特别是出口耦合度量与TLOC有显着的直接关联。动态导入耦合和NTC之间的关联性较小。同样，在所检查的四个系统中至少有三个系统中，Key Classes也显示与我们的测试套件指标显着相关。

这项工作的结果有助于我们理解生产特征与测试代码之间的关系性质。动态度量的使用可以提供单独使用静态度量不可用的洞察力。这些关系可以作为内部类级可测试性的指标，并且应该有助于通知维护和重新设计任务。

本研究的结果表明了未来的几个方向。通过检查更广泛的系统（例如闭源系统）可以扩展这项工作，以进一步评估结果。另一个研究方向是研究动态耦合和关键类信息是否可以共同用于预测测试类的大小和结构。预测类级可测试性应该改进对测试活动所需工作的早期估计和评估。这项工作还可以扩展到使用运行时信息调查其他源代码因素和可测试性之间的关联。将当前关于类可测试性的信息与其他测试信息（例如测试覆盖率和测试策略）结合起来也是有益的。

### 3.5 Metric Based Testability Model for Object Oriented Design (MTMOOD)

**[P29]的作者使用基于度量的可测试性模型进行了案例研究，该模型用于面向对象程序，在[P101]中有所介绍。该研究根据经验探索了模型在代码级别评估类的可测试性的能力。作者从单元测试和所需的测试工作的角度研究了可测试性。实证研究基于从两个Java软件系统中收集的JUnit测试用例。为了评估模型预测类的可测性（相应测试类的特征）的能力，作者使用相关性进行统计测试。**

**[P101]提出了一种基于度量的可测试性模型，用于面向对象的设计，名为MTMOOD。基于经验数据和回归分析，该研究报告了以下定量公式：可测性= -0.08 \*封装+ 1.12 \*继承+ 0.97 \*耦合。**

3.6 Empirical Analysis for Investigating the Effect of Control Flow Dependencies on Testability of Classes

[P72]报告了一个实证分析，用于研究控制流依赖性对类的可测试性的影响，在单元级测试中。结果提供了证据表明控制流依赖性和类的可测试性之间存在显着关系。

3.7 Improving the Testability of Object Oriented Software Through Software Contracts

[P85]提出了一种通过软件合同提高面向对象软件可测试性的方法。软件合同在一个类中进行了测试，并且使用路径测试技术为该类设计了测试用例，然后将其与类进行比较，而没有使用软件合同。该研究发现，软件合同的使用减少了测试用例的数量，从而提高了可测试性。

3.8 Method to analyzing software testability affecting factors based on testability tree

[P100]提出了一组Web应用程序可测试性评估的度量标准。该集包括这些度量：（1）IDP（具有ID属性的元素的数量），（2）TWI（工作流中断的数量）：当人类用户必须在SUT外执行操作时的情况的数量，（3）WIR（工作流程中断比率），和（4）TDI（困难元素的数量）：SUT中各种类型的元素的数量很难通过自动化测试来定位和处理。

3.9 MuAspectJ: Mutant Generation to Support Measuring the Testability of AspectJ Programs

[P106]提出了一个名为MuAspectJ的工具，用于生成用于测量AspectJ程序可测试性的突变体。还可以通过该工具测量可测试性。

3.10 On Testable Object-Oriented Programming

[P110]提出了一种名为TestableObject-Oriented Programming（TOOP）的方法，用于在编码或编译期间将可测试性构建到对象中，从而可以简化生成和实现的后续过程。

3.11 Predicting Testability of Eclipse: A Case Study

[P123]报告了一个关于测量Eclipse项目可测试性的案例研究。

3.12 Quality Plans for Measuring Testability of Models

[P129]的作者认为，对于基于模型的测试中使用的模型，评估其可测试性是一个重要问题。本文提出了一种质量管理方法，用于基于目标问题度量（GQM）和质量模型的组合评估软件模型。该方法还在模型中使用了“信息需求”的概念。质量模型将模型可测性分为三个方面：句法质量（软件模型与语言定义之间的对应关系），语义质量（软件模型与域之间的对应关系），可理解性（软件模型的理解能力）。

# Improving

# Other