构件可测试性测量方法

本章节描述了几种构件的可测试性测量方法。

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。

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，

， ， 。

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。

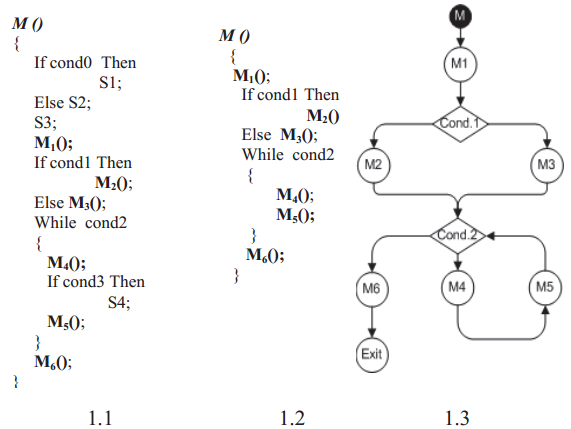


Fig 一种方法及其相应的控制调用图

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。

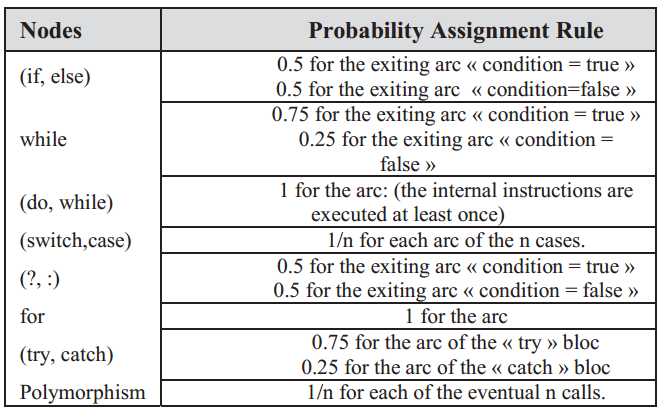


Fig 概率的分配规则

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。

Y Singh等人[11]以Eclipse项目为例，进行了面向对象程序包（package）级别的可测试性案例研究，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。这些指标通过大小（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) |

大小（Size）指标。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）。

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)，

References

[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.