根据用例估计工时

John Smith

Rational Software 白皮书

TP 171, 10/99

Rational
the software development company

目录

问题

从直觉上，似乎可以根据用例模型的特征估计开发需要的大小和工时。毕竟，用例模型捕获功能需求，因此所依据的用例是否应该与功能点有所不同呢？有几点困难：

- 由于存在很多种类的用例规范风格和形式，这就使得我们难以定义度量，例如，测量用例长度的度量。
- 用例应该代表系统外部参与者的观点，因此如果有一个为 500,000 软件代码行（sloc）的系统编写的用例一个为 5,000 sloc 的子系统编写的用例，则这两个用例所处的***级别***相差很大（Cockburn97 讨论了级别和目标的表示法）。
- 用例的复杂性可能不同，显式的是书面表达的复杂性不同，隐含的是所需实现的复杂性不同。
- 用例应该从参与者的视点来描述行为，但这可能非常复杂。尤其当系统具有状态时（因为大多数情况是这样）。要描述这种行为可能需要一个系统模型（在任何实现完成以前）。这会导致为了捕获行为的本质，在很多层次上分解功能并细化。

因此，需要实现某种用例来使估计变成可能？也许关于直接根据用例进行估计的期望太高了，这错误地造成了将功能点和用例点表示法等同起来的想法。无论如何，对功能点计数的计算需要系统模型。从用例描述派生功能点将要求统一用例表达的级别，并且仅在实现开始时进行合并用例表达级别，功能点计数才可信。 Fetcke97 描述了从用例到功能点的映射，但是用例的级别必须适当才能使映射有效。其他方法使用基于类对象的度量作为源，例如，价格对象点（Minkiewicz96）。

其他工作

在描述和表单化用例方面有一大堆工作 － Hurlbut97 作了较好的调查。从用例派生估计度量的工作已有较大程度的减少。Graham95 和 Graham98 包含对用例非常严厉的批评（但我不完全明白他为什么迄今为止还不赞同用例），提出将“任务脚本”作为解决用例问题（包括用例可变的长度和复杂性）的方法的想法。 Graham 的“原子任务脚本” 是收集“任务点”度量的基础。原子任务脚本的问题是它的级别非常低：根据 Graham, 理想中它应该是一个简单的句子，且如果只使用域术语的话不再可分解。Graham 的“根任务” 包含一个或多个原子任务脚本，每个根任务对应于“仅一个系统操作（在启动计划的类中）。”（Graham98）。在我看来，这些根任务与低级用例非常相似，且原子任务脚本类似用例中的各个步骤。级别问题仍然存在。

其他工作已由 Karner（Karner93）、Major（Major98）、Armour 和 Catherwood（Armour96）以及 Thomson（Thomson94）完成。Karner 在其论文中提出了计算用例点的方法，但重新假设用例以类（即，在比子系统更详细的级别）能实现的方式进行表达。

因此，我们是否应该避开用来估计的用例并且依赖分析和出现的设计实现？这个方法存在的问题是它将作出估计的时间延后，这使选择此技术的项目经理无法满意 － 此时***会***需要早期估计并且不得不使用一些其他方法。对于项目经理来说，最好能及早获得对用途规划的估计，然后以嵌套迭代的方式进行***优化***，而不是延迟估计并在没有计划的方式下继续。

本白皮书描述了一个框架，在该框架中可以使用任意级别的用例估计工时。要表达想法，需要根据经验描述一些具有相关维和大小的简单规范结构。本白皮书是一个大胆的猜想，因为在该领域缺乏工作和数据的情况下，我找不到其他方法。 我在表述中提出了“互联系统的系统”的想法。

接下来，我撇开主题来简单讲讲一些背景想法，它使得我在这条路上走下去。

避开功能分解？

功能分解的想法对许多从事软件开发的人来说似乎是一道咒语。我个人极端采用功能分解的经历（在一个非常大的数据流图中完成 3000 个 5 到 6 层深的原始变换除基础结构层以外，不考虑用体系结构完成）也使我不赞成这种做法。该案例中的问题不仅在于功能分解，还在于直到到达功能原始级别（在该点上规范在长度上应该少于一页）才描述流程的这一观念。

结果很难理解 － 难以了解如何从这些原始变换中产生在更高级别需要的行为。另外一点不清楚的是，功能结构应该如何映射到满足性能和其他质量需求的物理结构。所以荒谬的事情是我们不断分解直至达到我们能“解决问题”的级别（原始级别），关于原始的东西在一起工作能否在更高级别达到目标，这点还不是很清楚，也无法表示出来。此方法中不能考虑非功能需求。体系结构就其总体来说，不仅仅是基础结构（通信、操作系统等。），应该在分解时进行了演进并且应该彼此互相影响。

Bauhaus 关于“表单遵循功能”的定义又怎么样呢？根据功能可以设计出很多好东西，但也会设计出坏东西，比如到处使用水平的屋顶。如果您只考虑将屋顶及其附属物的全部功能设计成屋顶是居住者的遮盖物，那么结果在某些地方肯定不会让人满意。这样的屋顶很难防水；它们会造成积雪。

现在这些问题就能解决了，***但是如果您选择了一个不同的设计则会增加开销。***因此尽管说这些显得有些老生常谈，但表单应该遵循需求 － 全部需求（功能需求和非功能需求，后者可能包括审美需求）。架构设计师的问题是非功能需求常常难以表述，而且在“事物该是什么”方面，很大程度上依赖于架构设计师的经验。因此，如果功能分解单独驱动结构体系（如果分解成几个级别并且原始功能使用“模块”来一对一映射）并定义它们的接口，则功能分解显得不尽如人意。

类似这样的考虑使我相信，在***结构体系发挥作用之前***，把用例分解到某些标准级别（这可以通过类的协作实现）没有任何意义。如果系统处于某种大小（请参阅 Jacobson97），则会发生分解。但分解的标准和工程流程是很重要的 － 专门进行功能分解并不够好。

系统注意事项

系统工程师们在综合一个设计时做功能分析、分解和分配，但功能不是结构体系的唯一驱动力，专业工程师小组会参与对备选设计的评估。IEEE 标准 1220（系统工程流程应用程序和管理标准）的 *6.3* 节『功能分析』的 *6.3.1* 子节『功能分解』中描述了功能分解的用法，在 *6.5* 节『综合』中描述了系统产品解决方案。需要特别注意的是 6.5.1 子节『分组和分配功能』与 6.5.2 子节『备选物理解决方案』。在 *6.3.1* 节中，讲述了执行分解来清楚地***理解*** 系统必须完成什么，并且***一般分解一个级别就足够了***。

注意：功能分解的目的不是构造系统（综合是构造系统），而是了解和交流系统必须做什么，功能模型是完成这一步的有效方法。在综合中，子功能分配到解决方案结构中，然后在考虑所有需求的情况下，对解决方案进行评估。此方法和多级功能分解之间的区别是：在每个级别，您试着描述所需的行为并且在决定是否需要在下一级别进一步细化行为并将它们分配到更低级别的组件之前，找到一个解决方案来实现行为。

从这得出一个结论：无需用数百个用例来描述同一级别的行为。将充分覆盖所描述事物（系统、子系统和类）的外部用例（和相关场景）的数量可能非常少。我应该说说我对外部用例的理解。拿一个由子系统（这些子系统由类组成）组成的系统作为例子。我将那些描述系统行为及其参与者的用例称为外部用例。子系统也可能有它们自己的用例 － 这些用例对系统来说是内部的，但是对子系统来说是外部的。最终用来构造非常大（例如超过 1,000,000 行代码）的系统的用例（***外部和内部***）总数会达几百个，因为那种大小的系统会构造为系统中的系统或者至少是子系统的系统。

有关结构和大小的假设

用例数

在 Rational« Software 中，我们概要讲述了用例的数量要少（比如 10-50）而且观察到大量的用例（超过 100 个）会导致功能分解失效，此时用例不向参与者传递任何有价值的东西。不过，我们确实发现在真实的项目中有很多用例，而且并非全都是坏的，它们覆盖了很多级别，例如，在 Rational 内部邮件中，作者引用了来自 Ericsson 的一个例子：

> Ericsson 正在为大部分新一代电话交换机建模，它估计这需要 600 多个工作年（顶峰时，3-400 名开发者）和 200 个用例（***使用一个以上级别的用例，请参阅"互联系统的系统"***（斜体字部分）

对于一个需要 600 多个工作年的系统（这有多大？ 1,500,000 行 C++ 代码？），我怀疑用例分析在子系统（这就是说，如果一个人定义了一个有 7000-10000 行代码的子系统）的上一个级别就停止了，否则计数仍将增加。

因此，我坚持我的想法，那就是少量外部用例就已足够。为了能符合我提出的结构和
规模，我断言 ***10 外部用例***，每个有 ***30*** 个***相关场景***[1] 就足以描述行为[2] 了。如果在真实例子中，用例数量超过 10 个，并且它们确实是在该级别的外部，那么所描述的系统大于相应的规格。我会尽量提供一些支持数据，因为本书后面会用得到这些数字。

系统层次结构

推荐的系统层次结构是：

4 — 系统的系统

3 — 系统

2 — 子系统组

1 — 子系统

0 — 类

UML 中定义了类和子系统；在 UML 中，较大的聚集是子系统（包含子系统）。我将它们命名成不同名称，以使讨论变得简单一些。对于那些知道军用标准术语如 2167 或 498 的人（这些标准使子系统成为 CSC，使类成为 CSU）来说，聚集子系统组的大小与 CSCI 类似。据我回想，在经历了 2167 天有关什么 Ada 构造应该映射到什么级别的讨论后，最后，Ada 软件包通常映射到 CSU。我不建议系统一定要严格遵循此层次结构，各个级别之间可以混合，但是层次结构可用来推断出每个用例的大小对工时的影响。

每个级别都有用例（尽管可能不是用于单个类），但不是一堆详细得不可思议的用例，而是用于该级别的每个组件（即，子系统、子系统组等）的用例[3]。 我在上文中断言每个级别的每个组件应该有 10 个用例。如果平均用例描述有 10 页，这意味着规范文档的长度为 100 页（加上一个类似大小或更小数字的用于非功能需求的页数）。该数字为：

---

[1] 在 UML1.3 中，将场景描述为："**场景**：列举出行为的一系列特定操作。场景可用于列举出交互或用例实例的执行。"。
此处使用它的第二个含义，即列举出用例实例的执行。

[2] 注意：此场景号用于反映用例的复杂性，不建议开发者必须为每个用例生产和记录 30 个场景，而建议用 30 个场景捕获一个用例的大部分重要行为，即使有更多路径可以完成该用例。

[3] 一些复审者提出了对四个级别的用例前景的忧虑，但注意该四级用例只针对系统中的系统，该系统
通常将非常大。在这种情况下，我看到四级用例不会感到惊讶，尤其当工作是由一级承包商（对于系统的系统）、二级承包商（对于系统）和三级承包商（对于子系统）完成时。

Stevens98 喜欢的数字和 Royce98 建议的数字接近。但是为什么是 10 个用例呢？为获得该数字，我从底层开始，根据我认为对于每个子系统的类数量来说合理的大小、类大小和操作大小等进行推理。将这些数据收集到一起，与下表中的其他假设一起作为参考。

操作大小	70 sloc
每个类的操作数	12
每个子系统的类的数量	8
每个子系统组的子系统数量	8
每个系统的子系统组数量	8
每个系统中系统的系统数量	8
外部用例的数量（每个系统、子系统等）	10
每个用例的场景数量	30
每个用例描述的页数[4]	10

我没有很多经验数据 － 在文中零星散布了一些。Lorentz94 和 Henderson-Sellers96 有一些数据，我有一些来自澳洲项目（主要是航天军事领域的项目）的数据。在任何情况下，在此阶段或多或少将框架定位在正确的位置是很重要的。

层次结构中组件的大小

我要说我是用了数行代码了解到一些人不喜欢这种测量。这些代码行是 C++（或同等级别的语言）代码行，因此很容易回朔到功能点。

在容器中类的数量和所能表达的行为丰富程度之间有某种关系。我选择了 8 个类／子系统[5]、8 个子系统／子系统组和 8 个子系统组／子系统等。为什么是 8 个呢？

- 它在 7 加或者减去 2 之间。
- 因为在每个有 850 行 C++ 软件代码（每 70 sloc 有 12 个操作）的类中，提供一个大小约为 7000 sloc 的子系统 － 一大块功能／代码，它们可由小型团队（假设 3-7 人）在 4-9 个月里交付，它应该适应系统迭代的长度，其范围在 300,000 到 1,000,000 sloc（RUP99）。[6]

因此，多少个用例才能表达 8 个类（这些类是相互关联的，并且都在一个子系统中）的外部行为？不仅仅是用例的数量而且每个用例的场景的数量决定了丰富程度。目前，没有太多指导场景／用例扩展的方法 － Grady Booch 在 Booch98 中指出：“从用例到场景，有一个扩展因子。一个中等复杂的系统可能有几打能捕获其行为的用例，并且每个用例可以向外扩展为几打场景”，Bruce

[4] 在本书的后面，针对不同级别的系统，对这进行了细分。

[5] 我相信这类计数是有代表性的分析 － 在设计和实现过程中会有扩展和重新确定因子类的数量会增加三个或更多，而操作大小和类大小则相应地减小。

[6] 对于迭代时间较短的较小系统，可以将子系统规划得较小，或者总可能针对每个迭代规划部分交付 － 尽管这需要进行小心的控制并且可能需要交付“桩代码”。

Powel Douglass 在 Douglass99 中说“…。需要许多场景才能全面详细地描述一个用例 — 通常是一打到几打”。我选择了 30 个场景／用例 — 这在“几打”中是较少的，但是 Rechtin（在 Rechtin91 中）说工程师能处理 5-10 个交互的变量（为了说明此参数的用途，我将此参数解释为 5-10 个协作的类和 10-50 个交互（我已将此解释为场景）。这样解释后，多个用例就是这个可变范围的多个实例。

因此，10 个用例，每个用例有 30 个场景，总共 300 个场景（然后这些场景会产生大约 300 个测试用例）足以覆盖 8 个类的重要行为。还有其他什么能表明这是一个合理的数字？如果应用 Pareto 的 80-20 规则，那么 20% 的类将表达 80% 的功能；类似地，在每个类中，80% 的功能将由 20% 的操作表达。保守地说，我们需要用 20% 的类等来获取 75% 的功能，并且通过该点构造一个 Pareto 分布（图 1）。

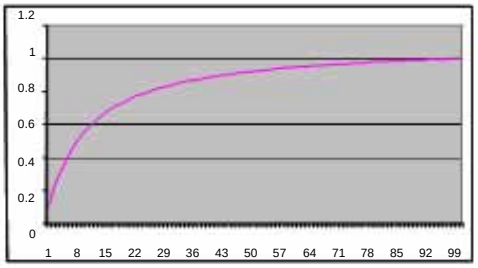


图 1：一个类似 Pareto 的分布

如果要对全部行为进行 80% 的覆盖并且在类、操作和场景的数量上应用 Pareto 规则，那么我们需要为每种对象进行 93%（0.93^3 是 0.8）的行为覆盖 — 每种对象需要 50%，即 4 个类和 5 个操作 (= (12 - 2 个构造器／析构器)/2)。为了表现有 4 个类（每个类有 5 个操作）的执行模式，对所构造的节点树的不同遍历数量可能达上千次。假设有以下层次结构，顶层有 10 个操作（接口操作），并且构造一个有三层的树后，我用每个节点至多 3 个链接构造了一个执行模式。这提供了近 1000 条路经和场景。因此 500 个场景应该有 93% 的覆盖。用 300 个场景，（使用同一假设）我们应该得到大约 73% 的覆盖。检查可以如何修改树，以删除冗余的行为规范，建议使用较小的数量就足够了，这取决于所选的算法。

另外一个达到此目的方法是询问个人 7000 sloc 的 C++ 需要多少测试用例（从场景派生而来）。这些测试是单元测试级别以外的测试，从 Jones91 和 Boeing 777 项目（Pehrson96）那里可以得到一些证据，表明这些数字是安全的（至少在代表实践时）。这些来源建议在 250-280[7] 之间是正确的。在完全不同的级别，加拿大自动化空中交通系统（CAATS）项目使用 200 个系统测试（专用通信）。

用例大小

一个用例应该为多大？足够大使其可以表达得足够详细，这样才可能实现所需的行为 — 这将依赖它的复杂性以及是内部用例还是外部用例，而这些又与系统类型有关。以下我们讨论应该描述多少个系统内部操作的问题。显然，根据对系统外部行为的描述构建一个系统需要使输出与输入相关。例如，现在如果行为是对历史敏感的和复杂的，那么如果没有一些系统内部及系统采取的操作的概念性模型，则很难描述行为。注意：尽管没有必要描述如何从内部构造系统 — 任何满足非功能需求和符合模型行为的设计都可行。

[7] 我在从 Rational 复审者那里收到的反馈中，感觉这对于非关键系统来说有点多余了，
这些系统的每个用例的场景少于 30 个。拥有关于此用例数量范围以及测试用例数量与使用中发现的缺陷数量之间的关系的更多数据，将非常重要。

UML1.3 中提供的定义是：“**用例[类]**：一系列系统（或者其他实体）可以执行、与系统参与者交互的的操作（包括变体）的规范。对于复杂的行为，有理由使用该定义
来包含内部操作（除非将它推迟到实现），这一实现离最终用户还有一定距离。在用例中还应该增加业务规则以限制参与者的行为，例如在 ATM 系统中，银行规定无论账户的余额是多少，每笔交易的金额都不得低于 $500。

根据这样的解释，事件用例流的描述可以有 2-20[8] 页。显然，行为简单且算法上简单的系统不需要很长的描述。也许我们可以说：可以用 2-10 页（平均 5 页）来描述简单业务系统的特征；对于更复杂（业务和科学领域）的系统，则用 6-15 页（平均 9 页）；对于复杂的命令和控制系统，用 8-20 页（平均 12 页）进行描述，这些数字反映了对于相同大小的系统，工时与系统类型的非线性关系），尽管我没有数据来支持这一说法。更具表达力的描述形式，例如状态机或者活动图，可能占用更少的空间。我们仍着重用文本进行阐述，这样我暂时可以忽略其他因素 － 现在实在没有什么数据。

根据大小区分系统的开发应该对从这些启发性问题中派生出来的每个用例所花的时间（以小时为单位）应用增效因子（我建议添加一个 COCOMO 风格的开销驱动器，根据观察，对于系统分类（简单业务系统、更复杂业务系统、命令和控制系统等），这是平均大小／建议的平均大小。）。

用例大小的另一方面是场景计数；例如，一个只有 5 页的用例可能有一个能允许许多路径通过的复杂结构。另外，需要估计场景的数量，并将该数与 30（我最初构想的每个用例的场景数量）的比值用作开销驱动器。

结果是我们推断一个基于规范且有 100 页的用例对于任何给定级别的外部规范来说足够了，补充规范除外。范围是 20-200 页（这些限制是模糊的）。注意：***最底层***的系统（由子系统组组成）的总数是 3-15 页／ksloc（简单
业务系统）－ 12-30 页／ksloc（复杂命令和控制系统）。这似乎可以解释 Royce98 表 14-9（该表中，对工件的页的计数较少）和对真实项目的观察结果（尤其在防御系统中，它产生大量的页面）之间的明显矛盾。
该页产生自不需要用 页面来落实的规范级别（Royce 是正确的，像 Vision Statement 那样重要的事情应该处于该表中指出的那个等级），对一个大型且复杂的系统来说是 200 页。

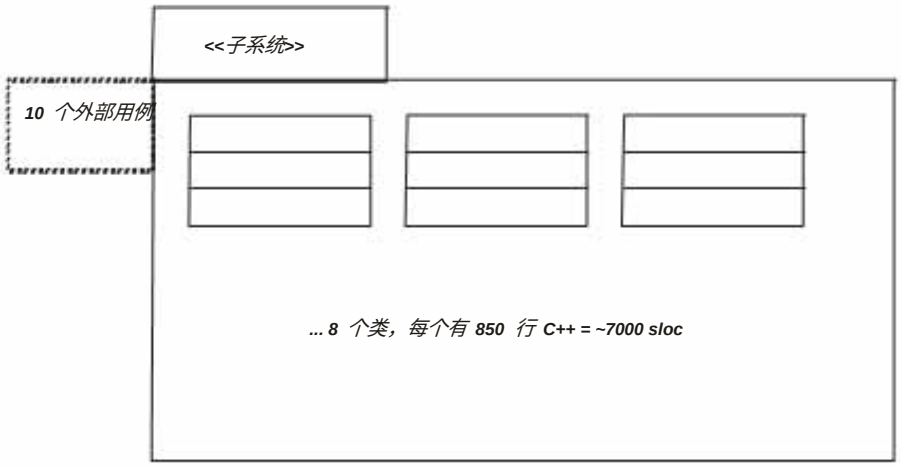
子系统层次结构

子系统层次结构看起来是什么样的？以下是我用过的简单“标准”格式。注意：这些是用于实现系统的概念性格式。实际的系统边界在这些格式集的外部，每个外部用例的总和是系统外部用例的总和；因此，一个真实的系统具有 10 个以上的外部用例，但是我们在后面将看到，上限没有限制。注意：这里不建议所有开发描述都必须使用 4 个级别的用例。较小的系统（<50000 sloc）可能只使用一个或者两个用例。

[8] 注意：这不是强制的上限；用例描述的长度将遵循某些种类的统计分布，这样发生极端情况的可能性会较低。

第 1 级

在第 1 级，我们使用零个以上数量子系统中的类来实现用例：



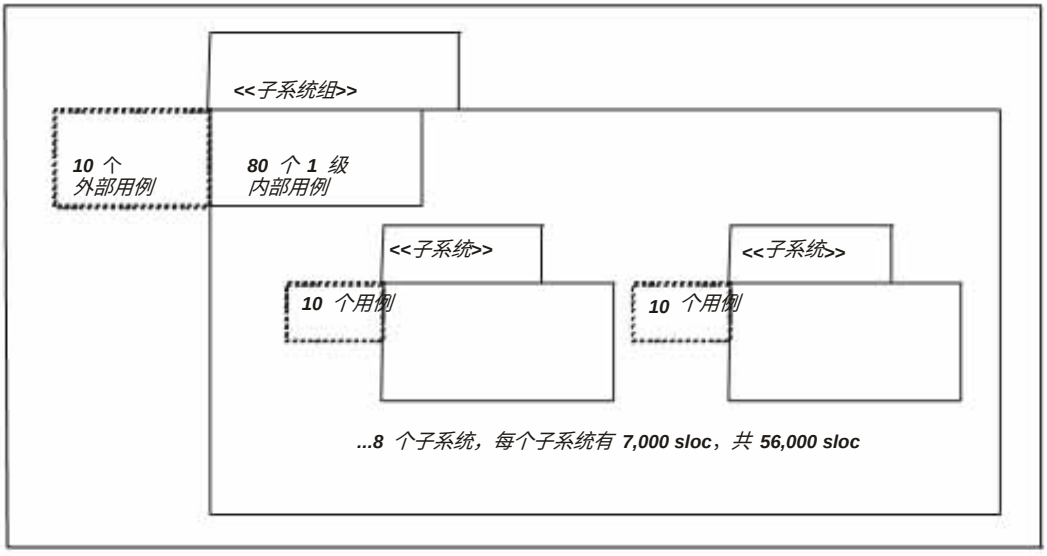
估计该级别系统大小的范围（使用 7 加减 2 的办法）：

- 2 到 9 个类（没有形成子系统） — 1700 slocs - 8000 slocs，或者
- 1 个子系统，其中有 5 个类，总计 4000 多 sloc
- 9 个子系统，每个子系统有 7 个类，总计 53,550 sloc，

使用类实例所表达的可实现用例。该范围中有 2-76 个用例。这些是模糊限制，至少上限是 — 此方法构造一个系统的可能性（以此大小），决不要在一些更高级别的格式中 表达需要的行为，应该在此限制上减少到零。一个大的用例计数可能会表现出一些问题。

级别 2

在此级别，有一个由 8 个子系统组成的子系统组。我认为该子系统组等价于与防卫术语中的一个计算机系统配置项（CSCI）。在此级别，用例由子系统协作实现：

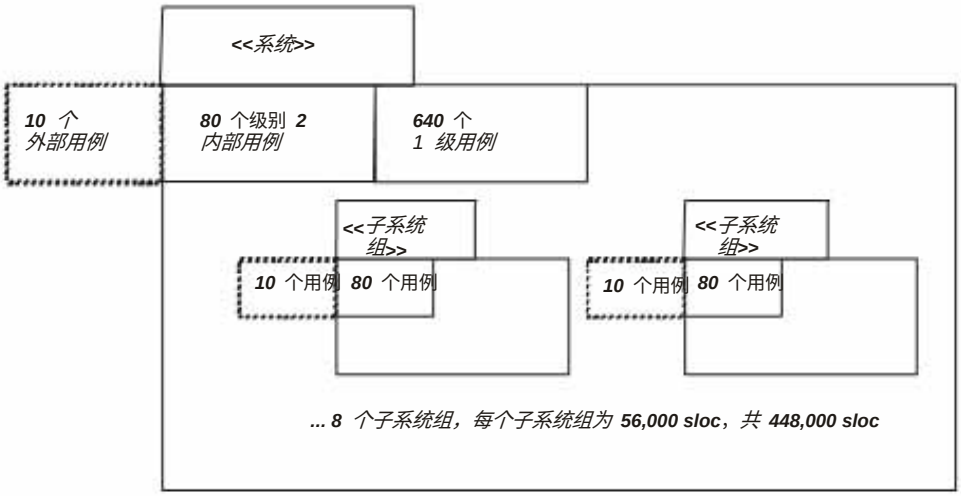


估计该级别系统大小的范围（使用 7 加减 2 表示法）： □1 个子系统组，其中有 5 个子系统，每个子系统有 5 个类，共 22,000 sloc - □9 个子系统组，每个子系统组有 7 个子系统，每个子系统有 7 个类，共 370,000 sloc

该范围中有 4-66 个外部用例。这些也是模糊限制。

级别 3

在此级别，有一个子系统组组成的系统。在级别 3，用例由子系统组协作实现：



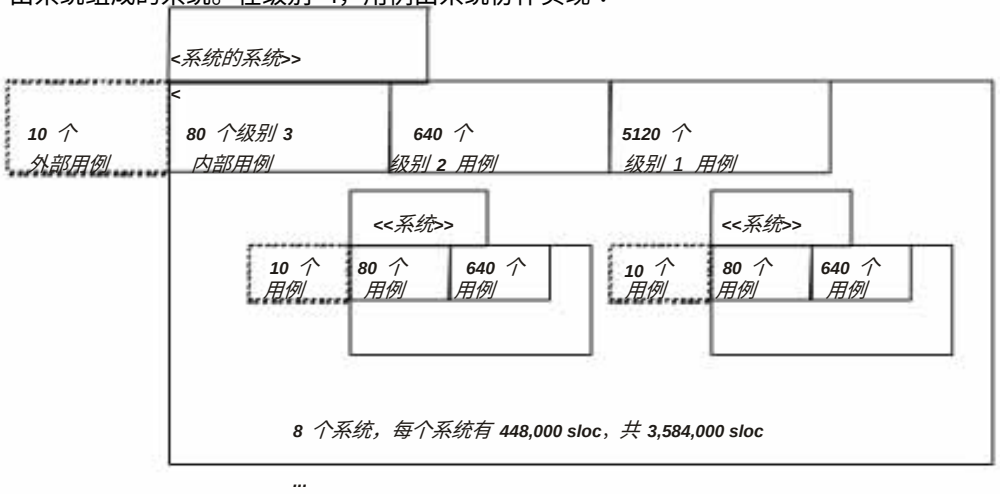
估计该级别系统大小的范围（使用 7 加减 2 表示法）：

□1 个有 5 个 子系统组的系统，每个子系统组有 5 个子系统，每个子系统有 5 个类，共 110,000 sloc - □9 个系统，每个系统有 7 个子系统组，每个子系统有 7 个类，共 2,600,000 sloc。

该范围中有 3-58 个外部用例。这些也是模糊限制。

级别 4

在此级别，有一个由系统组成的系统。在级别 4，用例由系统协作实现：



估计该级别系统大小的范围（使用 7 加减 2 表示法）：

- 1 个系统的系统，每个有 5 个系统，每个系统有 5 个子系统组，每个子系统组有 5 个子系统，每个子系统有 5 个类，共 540,000 sloc -
- 9 个系统的系统，每个有 7 个系统，每个系统有 7 个子系统组，每个子系统组有 7 个子系统，每个子系统有 7 个类，共 18,000,000 sloc。

该范围中有 2-51 个外部用例。这些也是模糊限制。我认为更大的聚集是可能的，但是我不想考虑它们！

每个用例的工时

我们可以通过在每个级别上为这些名义上的大小估计工时来深入研究每个用例所需的工时。使用 Estimate Professional™ 工具[9]（基于 COCOMO 2[10] 和 Putnam 的 SLIM[11] 模型），将语言设置为 C++（其他 开销驱动设置为名义）并且在每个名义大小点（假设10 个外部用例）为每个系统类型示例计算工时，给出了表 1 中的结果。

表 1：每个各种类型样本用例的工时

大小（slocs）	工时／简单业务系统用例	工时／科学系统用例	工时／复杂的命令和控制系统用例
7000（L1）	55（范围是 40-75）	120（范围是 90-160）	260（范围是 190-350）
56000（L2）	820（范围是 710-950）	1700（范围是 1500-2000）	3300（范围是 2900-3900）
448000（L3）	12000	21000	38000
3584000（L4）	148000	252000	432000

[9] Software Productivity Center Inc（http://www.spc.ca/）提供 Estimate Professional 工具。

[10] 请参阅 Boehm81 和 http://sunset.usc.edu/COCOMOII/cocomo.html。

[11] 请参阅 Putnam92。

表 1 中显示的级别 1（L1）和级别 2（L2）的范围考虑了一个单独用例的复杂性 － 这是用具有 COCOMO 代码复杂性矩阵的类似标准进行估计的。我认为从 L2 开始，系统类型特征会包含复杂性的变化，因此一个更高级别的复杂命令和控制系统的用例会混有较低级别的复杂性。以对数-对数标度绘制这些数据生成图 2。

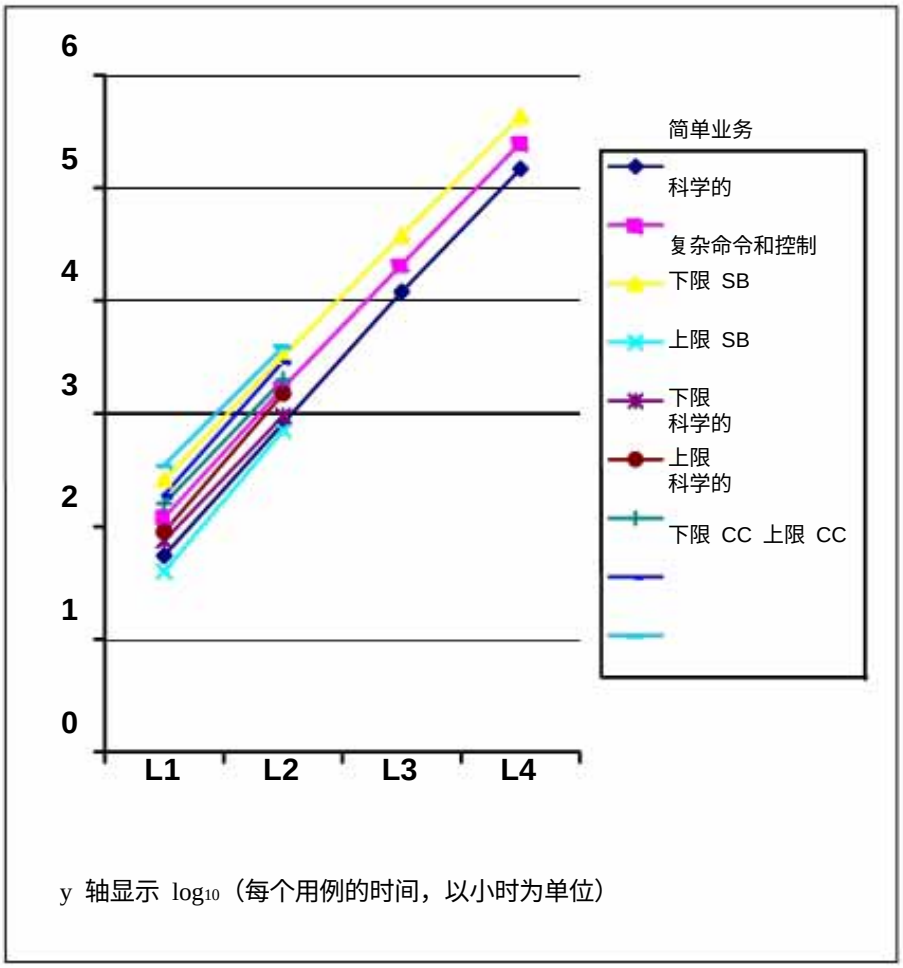


图 2：用例大小与工时

从中可以看出，旧的对象数量 150-350 小时／用例（$10^{2.17}$-$10^{2.54}$）很符合 L1，即，这些是可以通过类的协作来实现的用例 － 所以仍然会对此数字作一些调整。然而，在分析过程中表现所有项目的特征是不够的 － 就如一个同事在一份交流电子邮件中说的：“它太平直了”。

估计工时

现在，一些真实系统不适合这些方便的（插）槽，因此，为了有助于推理出应该如何规划系统的特征，我们可以使用随该方法派生出来的模糊限制，并且绘成图，如图 3 所示。

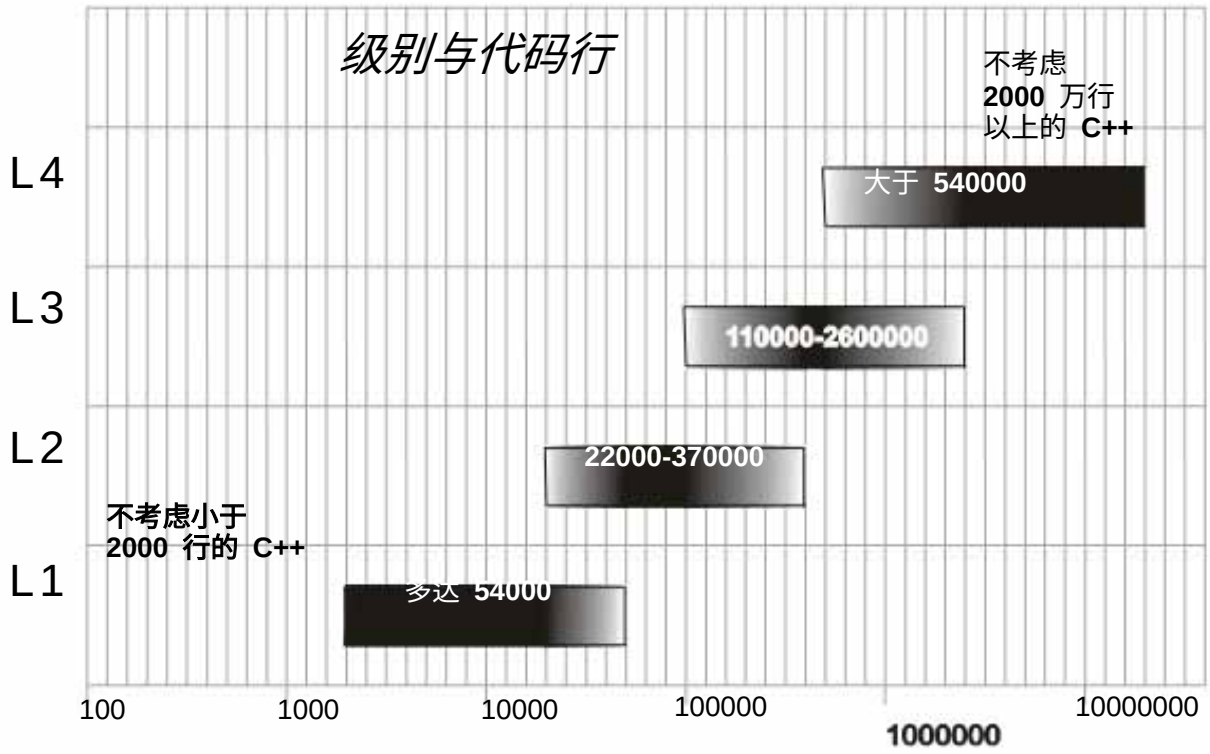


图 3：每个级别的大小区域

从图 3 中可以看出，达到 22000 sloc 的系统最可能在 1 级中，使用 2-30 个用例进行描述。在该大小，较高的用例计数可能表明该用例太详细。

在 22000 和 54000 sloc 之间，会混有级别 1 和级别 2 的用例，用例数为 4（全部是级别 2）到 76（全部是级别 1）之间的数字。正如图表尝试显示的，这些极限值的可能性很小。

在 54000 和 110000 sloc 之间，可能可以完全在级别 2 用 10-20 个用例描述构造良好的系统；L1/L2/L3 的混合（1-160 个用例，这些极限值的可能性*极*低）。

在 110000 和 370000 sloc 之间，很可能是级别 2 和级别 3 的混合，用例数为 3（全部是级别 3）和 66（全部是级别 2）之间的数字。

在 370000 和 540000 sloc 之间，如果完全在级别 3 进行描述，将使用 9-12 个用例；可能是 L2/L3/L4 的混合（1-100 个用例，这些极限值的可能性*极*低）。

在 540000 和 2600000 sloc 之间，可能是级别 3 和级别 4 的混合，用例数是 2（全部是级别 4）到 60（全部是级别 3）之间的数字。

在 2600000 sloc 以上，级别 4 用例数应该大于 ~8。

要多少用例才够用？

这当中有一些重要的观察流，它们支持一些主要规则。经常会问以下问题：“多少 用例才算太多？”该问题通常表示***需求捕获过程***中多少用例才算太多。 答案可能是：即使对于最大的系统，多于约 70 个可能表示详细程度太优先于设计。在 5-40 之间较为合适，但是如果不考虑级别，仅数字本身不能用来估计大小和工时。这是***初始***数字，适合于特定级别。如果一个大型的超级系统分解为系统，然后又分解为子系统等，则将会产生数百个用例。如果用例开发至到达类级别，那么最终的计数会成百甚至上千（例如，对于 140 个工作年的项目，约这 600 个用例，或者每个用例 15 个功能点）。然而，作为与设计无关的纯用例分解，不会发生这种情况。这些用例来自 Jacobson97 中描述的过程 － 其中系统等级的用例划分成跨子系统分布的行为，我们可以为这些行为编写较低级别的用例（把其他子系统当作参与者）。

工时估计过程

那么我们如何进行估计呢？现在有一些前提：要根据用例进行估计，***必须先理解问题领、对提出的系统大小有概念、对体系结构有一些概念并处于进行估计的阶段。***

第一次粗略划分估计时，可以采用专家的意见；或稍微正式一些，使用 Wideband Delphi 技术（由 Rand 组织在 1948 年，关于其描述，请参阅 Boehm81）。这将允许估计者将系统置于图 3 中的某个大小区域中。这种放置会给出一个用例计数范围，并指示表达级别（L1、L1/L2 等）。然后估计者必须根据对该体系结构和该领域词汇表的当前了解以事件流表达方式确定：用例是否很适合一个级别（单独的级别或是几个级别的混合）。

根据这些考虑，也应该可以看出数据是否有问题；例如，如果 Delphi
的估计是 600,000 行代码（或相当的功能点），由于几乎没有体系结构方面工作，因此还不是很了解系统结构，图 3 指出用例计数应该在 2（全部是级别 4）和 14（全部是级别 3）之间。如果实际用例计数是 100，那么可能过早地分解用例或者 Delphi 的估计需要很多时间。

继续此例子：如果实际用例计数是 20，并且估计者确定这些都是 L3，而且 用例长度平均是 7 页，系统是复杂业务类型，那么每个用例的时间（图 2 中所示）是 20000 小时。考虑到明显较低的复杂度（基于用例长度），这个数字要乘以 7/9。因此，按此方法计算出的总工时是 20*20000*(7/9) = ~310000 个工作小时，或者 2050 个工作月。根据 Estimate Professional，对于一个复杂的业务系统来说，600,000 行 C++ 代码需要 1928 个工作月。因此，对于这个混合例子，两种方法得出的结果是一致的。

如果实际用例计数是 5，并且估计者确定这些用例中，1 个是 L4 用例，4 个是级别 3 用例，并且L4 用例有 12 页，L3 用例平均 10 页，那么工时就是 1*250,000*12/9+4*21000*(10/9) = ~2800 个工作月。这可能意味着需要重新访问 Delphi 的估计，尽管仍在非常高的级别对系统的主要部分进行了了解，错误偏差较大。

如果原 Delphi 估计是 100,000 行 C++，图 3 提示用例应该在 L2，且应该有约 18 个用例。如果实际上有 20 个用例，就如同第一个例子中的那样，在没有考试实际用例级别的情况下应用此方法会导致错误结果，如果 Delphi 的估计是错误的话。

因此，估计者必须检查用例真的是在建议的抽象级别（L2）中并且通过子系统协作可以实现它们，还要检查用例实际不全在 L3 － 尽管 Wideband Delphi 方法并不总是那么糟糕的（即，实际数量接近 600,000 时预测数为 100,000）。问题是，尽管在不构造一些对应用例级别的想象或者概念体系结构的情况下，无法有信心地使用这一估计方法。 对于在该领域中非常有经验的估计者来说，该模型可能是精神上的，它支持对级别的判断；对于经验较少的估计者和团队，做一些体系结构模型来看在特定级别实现用例的情况，是明智的做法。

混合表达用例的计数（级别 N 和级别 N+1 的混合）对于下限用例类型应该以 n=8 来计数（两个级别之间的小距离 ）。因此，在 50% L1 和 50% L2 估计的用例应该计数为
$8_{0.5}$ = 3 个 L1 用例来得到总的计数。在 L2 和 L3 之间的 30% 估计的用例应该计数为 $8_{0.3}$ L2 用例 = 2 个 L2 用例。在 L2 和 L3 之间的 90% 估计的用例应该计数为 $8_{0.9}$ = 7 个 L2 用例。

表大小调整

实际上还需要对个别时间／用例图做进一步的调整来计算总体大小 － 在那种***系统大小***的情况下，工时图适合每个级别。因此在 L1，在表 1，当构建一个有 7000 sloc 的系统时，每个用例 55 小时将是适用的。实际数字取决于总的系统大小，因此，例如，如果要构建的系统 有 40,000 sloc，并且用 57 个 1 级用例来描述它，那么对于一个简单的业务系统，工时不是 55*57 小时，而是 $(40/7)^{0.11}$* 55 = 66 小时／用例。这是基于 COCOMO 2 的工时与大小的关系。 根据 COCOMO 模型，工时 = A * (大小)$^{1.11}$：

- 大小是 ksloc
- A 中包含了开销驱动器因子
- 项目规模因子并不重要（假定指数是 1.1）

注意：可以使用 Estimate Professional 等工具来计算这些因子，以减少计算负担；为了完整性，才在此处显示它们。

因此，每 ksloc 或者每单位的工时 = A* (大小)$^{1.11}$/大小，其结果是 A* (大小）$^{0.11}$，且大小 S1 的工时/单位与大小 S2 的工时/单位的比是（S1/S2）$^{0.11}$。

除了 Delphi 估计，可以根据各种级别的用例计数粗略计算系统大小：如果在级别 1 有 N1 个用例，在级别 2 有 N2 个，在级别 3 有 N3 个，在级别 4 有 N4 个，那么总大小是 [(N1/10)*7 + (N2/10)*56 + (N3/10)*448 + (N4/10)*3584] ksloc。因此我们可以为表 1 中每个用例数字的工时计算工时增效因子，方法是按照在表 1 的第 1 列中显示的每个级别（以 ksloc 为单位）的大小来划分总大小。

- 因此，在级别 1　　$(0.1*N1 + 0.8*N2 + 6.4*N3 + 51.2*N4)^{0.11}$
- 在级别 2　　$(0.0125*N1 + 0.1*N2 + 0.8*N3 + 6.4*N4)^{0.11}$
- 在级别 3　　$(0.00156*N1 + 0.0125*N2 + 0.1*N3 + 0.8*N4)^{0.11}$
- 在级别 4　　$(0.00002*N1 + 0.00156*N2 + 0.0125* N3 + 0.1*N4)^{0.11}$

很明显，例如在级别 4，级别 1 用例的数量和级别 3 的数量或者级别 4 的数量比起来就显得微不足道。

总结

此处提供了基于用例的估计框架。为了使陈述更具体，为框架参数选择了一些值，它们并不都是错误，还处在争论阶段。通常，收集数据时应该根据实际情况测试这类猜想，并重新估计参数。框架考虑了不同类别系统的用例级别、大小和复杂性，并且框架不会经常进行细致的功能性分解。为了减轻计算负担，可能要为工具构造一个前端，比如 Estimate Professional，它根据用例的输入大小提供了一个备选方法。

关于获得本白皮书的意见和反馈，请联系 John Smith（jsmith@rational.com）。

参考资料

1. Armour96：Experiences Measuring Object Oriented System Size with Use Cases，F. Armour，B. Catherwood 等，Proc. ESCOM，Wilmslow，英国，1996 年
2. Boehm81：Software Engineering Economics，Barry W. Boehm，Prentice-Hall，1981 年
3. Booch98：The Unified Modeling Language User Guide，Grady Booch，James Rumbaugh，Ivar Jacobson，Addison Wesley，1998 年
4. Cockburn97：Structuring Use Cases with Goals，Alistair Cockburn，Journal of Object-Oriented Programming，1997 年 9-10 月和 1997 年 11-12 月
5. Douglass99：Doing Hard Time，Bruce Powel Douglass，Addison Wesley，1999 年
6. Fetcke97：Mapping the OO-Jacobson Approach into Function Point Analysis，T. Fetcke，A. Abran 等，Proc. TOOLS USA 97，Santa Barbara，加利福尼亚，1997 年
7. Graham95：Migrating to Object Technology，Ian Graham，Addison-Wesley，1995 年
8. Graham98：Requirements Engineering and Rapid Development，Ian Graham，Addison-Wesley，1998 年
9. Henderson-Sellers96：Object-Oriented Metrics，Brian Henderson-Sellers，Prentice Hall，1996 年
10. Hurlbut97：A Survey of Approaches For Describing and Formalizing Use Cases，Russell R. Hurlbut，Technical Report: XPT-TR-97-03，http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf
11. Jacobson97：Software Reuse - Architecture, Process and Organization for Business Success，Ivar Jacobson，Martin Griss，Patrik Jonsson，Addison-Wesley/ACM Press，1997 年
12. Jones91：Applied Software Measurement，Capers Jones，McGraw-Hill，1991 年
13. Karner93：Use Case Points - Resource Estimation for Objectory Projects，Gustav Karner，Objective Systems SF AB（copyright reserved by Rational Software），1993 年
14. Lorentz94：Object-Oriented Software Metrics，Mark Lorentz，Jeff Kidd，Prentice Hall，1994 年
15. Major98：A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object Oriented Software Projects，Melissa Major and John D. McGregor，Dept. of Computer Science Technical Report 98 002，Clemson University，1998 年
16. Minkiewicz96：Estimating Size for Object-Oriented Software，Arlene F. Minkiewicz，http://www.pricesystems.com/foresight/arlepops.htm，1996 年
17. Pehrson96：Software Development for the Boeing 777，Ron J. Pehrson，CrossTalk，1996 年 1 月
18. Putnam92：Measures for Excellence，Lawrence H. Putnam，Ware Myers，Yourdon Press，1992 年
19. Rechtin91：Systems Architecting, Creating & Building Complex Systems，E. Rechtin，Prentice-Hall，1991 年
20. Royce98：Software Project Management，Walker Royce，Addison Wesley，1998 年
21. RUP99：Rational Unified Process，Rational Software，1999 年
22. Stevens98：Systems Engineering - Coping with Complexity，R. Stevens，P. Brook 等，Prentice Hall，1998
23. Thomson94：Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects，N. Thomson, R. Johnson 等，Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics，OOPSLA ’94，1994 年

Rational
the software development company

两家总部：

Rational Software
18880 Homestead Road
Cupertino, CA 95014
电话：(408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
电话：(781) 676-2400

免费电话：(800) 728-1212
电子邮件：info@rational.com
Web：www.rational.com
全球网址：www.rational.com/worldwide