The Unified Process

作者：Ivar Jacobson, Grady Booch, and James Rumbaugh, Rational Software
翻译：环球联动开发组（Global Empower Inc.）

统一软件开发过程

当前，软件的趋势是朝着更大更复杂的系统发展。这部分地是因为计算机的处理能力每年都在增大，导致用户对它的期望更多。同时，这种趋势也受到为交流各种信息（从纯文本到格式化文本到图像到图表再到多媒体）而不断扩大互联网的使用的影响。在产品版本的不断升级过程中，我们了解到产品是如何被改进的，因此我们对越来越复杂的软件的胃口也就越来越大。我们需要更符合我们的需要的软件，但是，这种需要反过来又使得软件越来越复杂。总之，我们需要更多。

我们希望软件运行得越来越快捷。推向市场的时间是另一个重要的推动因素。

然而，要达到这个目的是困难的。我们对强大、复杂软件的需要与软件开发的当前状况并不一致。今天，大多数人还在使用 25 年前使用的旧方法来开发软件。这就是症结所在。除非我们革新我们的方法，否则，我们无法达到开发当前所需的复杂软件的目标。

我们可以把这个软件问题归结为软件开发人员面临的将一个大型软件项目的众多线索综合在一起的困难。软件开发界需要一种受控的工作方式。它需要一个过程来集成软件开发的许多方面。它需要一种通用方法， 该方法能：

ω提供应如何对整个开发团队的开发活动进行组织的指导；

ω综合指导单个开发人员和开发团队；

ω规定开发成果是什么；

ω提供监控和衡量一个项目中的产品和活动的标准。

一个定义良好且管理良好的过程是区别成效卓著的项目和不成功项目之间的重要指标。“统一软件开发过程”正是我们在软件开发上面临的难题的解决之道。

“统一过程”概述

第一点也是最重要的一点是，这个“统一过程”是软件开发过程。软件开发过程是将用户的需求转化为一个软件系统的一系列活动的总称（见图一）。然而，“统一过程”不仅仅是一个过程。它是一个通用过程框架，可以应付种类广泛的软件系统、不同的应用领域、不同的组织类型、不同的性能水平和不同的项目规模。

“统一过程”是基于组件的，这意味着利用它开发的软件系统是由组件构成的，组件之间通过定义良好的接口相互联系。

在准备软件系统的所有蓝图的时候，“统一过程”使用的是“统一建模语言（Unified Modeling Language）”。事实上，UML 是“统一过程”的有机组成部分——它们是被同步开发的。

然而，真正使“统一过程”与众不同的方面可以用三个句话来表达：它是用例驱动的、以基本架构为中心的、迭代式和增量性的。正是这些特征使得“统一过程”卓尔不群。

“统一过程”是用例驱动的

开发软件系统的目的是要为该软件系统的用户服务。因此，要创建一个成功的软件系统，我们必须明

图 1. 软件开发过程

白其潜在用户需要什么。

“用户”这个术语所指并不仅仅局限于人类用户，还包括其他系统。在这种意义上，“用户”这个术语代表与利用“统一过程”开发出来的系统发生交互的某个人或者某件东西（例如在所要开发的系统之外的另一个系统）。交互的一个例子是使用自动取款机的一个人。他/她插入塑料磁卡，回答机器显示器上提出的问题，然后就得到了一笔现金。在响应用户的磁卡和回答时，系统完成一系列的动作，这个动作序列为用户提供了一个有意义的结果，也就是提取了现金。

这个交互就是一个“用例”。一个用例就是系统中向用户提供一个有价值的结果的某项功能。用例捕捉的是功能性需求。所有用例结合起来就构成了“用例模型”，该模型描述系统的全部功能。这个模型取代了系统的传统的功能规范说明。一个功能规范说明可以描述成对这个问题的回答：需要该系统做什么？而用例战略则可以通过在该问题中添加几个字来描述：需要该系统为**每个用户**做什么？这几个字有着重大意义。它们迫使我们从用户的利益角度出发进行考虑，而不仅仅是考虑系统应当具有哪些良好功能。

然而，用例并不仅仅是定义一个系统的需求的一个工具。它们还驱动系统的设计、实现和测试。也就是说，它们驱动整个开发过程。基于用例模型，软件开发人员创建一系列的设计和实现模型来实现各种用例。开发人员审查每个后续模型，以确保它们符合用例模型。测试人员将测试软件系统的实现，以确保实现模型中的组件正确实现了用例。这样，用例不仅启动了开发过程，而且与开发过程结合在一起。“用例驱动”意指开发过程将遵循一个流程：它将按照一系列由用例驱动的工作流程来进行。首先是定义用例，然后是设计用例，最后,用例是测试人员构建测试案例的来源。

尽管确实是用例在驱动整个开发过程，但是我们并不能孤立地选择用例。它们必须与系统架构协同开发。也就是说，用例驱动系统架构而系统架构反过来又影响用例的选择。因此，随着生命期的继续，系统架构和用例都逐渐成熟。

“统一过程”是以基本架构为中心的

软件架构的作用在本质上与基本架构在建筑物结构中所起的作用是一样的。我们从不同的角度来观察建筑物：结构、服务、供热装置、水管装置、电力等。这样，在开始建设之前，建设人员就可以对建筑物有一个完整的把握。同样地，软件系统的基本架构也被描述成要创建的系统的各种不同视图。

软件基本架构这个概念体现了系统最为静态和动态的方面。基本架构根据企业的需求来设计，而这种需求则是由用户和其他利益关联人所感知，并反映在用例之中。然而，它还受其他许多因素的影响：软件运行的平台（例如计算机基本结构、操作系统、数据库管理系统和网络通信协议等）、可得到的可再用构件（比如图形用户界面框架）、配置方面的考虑、已有系统和非功能性需求（比如性能和可靠性）等。基本架构是一个关于整体设计的视图，在这个视图中，省略了一些细节，以使软件的更为重要的特征体现得更为明显。由于什么东西是重要的部分取决于主观判断，而这种判断又来自于经验，因此，基本架构的价值取决于被指派完成这一任务的人员。然而，过程有助于架构设计师集中精力于正确的目标，比如可理解性、顺应未来变化的灵活性和可再用性。

用例和基本架构之间的关系如何呢？每个产品都是功能和形式的有机统一。仅仅只有其中之一，都是不完整的。只有平衡把握这两个方面才能得到一个成功的产品。在这种情况下，功能应与用例相对应，而形式应当与基本架构相对应。用例和基本架构之间必定是相互影响的，这几乎就是一个“鸡和蛋”的问题。一方面，我们实现的用例必须与基本架构相适应。而另一方面，基本架构必须留有实现现在和未来需要的所有用例的空间。在实践中，基本架构和用例必须平行开发。

因此，架构设计师将软件系统构筑在一个形式当中。正是这个形式即基本架构必须被设计成让系统不

仅在初始开发期间,而且在未来的版本进化过程中能不断发展。要找到这样的一个形式，架构设计师必须对系统的关键功能也就是系统的关键用例有一个总体性把握。这些关键用例只占用例总数的 5-10%，但是它们却是最重要的，因为它们将构成整个系统的核心功能。下面是这个过程的简单描述：

ω架构设计师首先从不与特定的用例相关的部分（比如平台）着手来创建基本架构的大致轮廓。尽管基本架构的这部分是用例无关的，但是，在建立基本架构的轮廓之前，架构设计师必须对用例有一个总体把握。

ω其次，设计人员应当从已经确认的用例子集着手开始工作，这些用例是指那些代表待开发系统的关键功能的用例。每个选定的用例都应当被详细描述，并在子系统、类和组件层次上实现。

ω随着用例已经被定义并且逐渐成熟，基本架构就越来越成形了。而这种状况，反过来又导致更多用例的成熟。

这个过程会不断持续下去，直至基本架构被认定为稳定了为止。

统一开发过程是迭代式的和增量的

开发一个商业软件产品是一项可能持续几个月、一年甚至更长时间的工作。因此，将此种工作分解成若关更小的部分或若干小项目是切合实际的。

每个小项目是指能导致一个增量的一次迭代。迭代指的是工作流中的步骤，而增量指的是产品的成长。为了更加高效，迭代必须受到控制；也就是说，必须对它们进行选择并有计划地实现它们。这就是为什么它们是小项目的原因。

开发人员根据两个因素来选择在一次迭代中要实现什么。首先，迭代与一组用例相关，这些用例共同扩展了到目前为止所开发的产品的可用性。其次，迭代涉及最为重要的风险。后续迭代是建立在先前的迭代完成后的开发成果之上的。它是一个小项目，因此，从用例开始，它还是必须经过下列开发工作：分析、设计、实现和测试，这样，就以可执行代码的形式在迭代中实现了用例。当然，一项增量并不一定就是添加性的。特别是在生命期的早期阶段，开发人员可能会用一个更为详尽或者复杂的设计来取代那种较为简单的设计。在后期，增量通常都是添加性的。

在每次迭代中，开发人员识认并详细定义相关用例，利用已选定的基本架构作为指导来建立一个设计，以组件形式来实现该设计，并验证这些组件满足了用例。如果一次迭代达到了它的目标（通常如此），那么开发过程就进入下一次迭代的开发了。当一次迭代没有满足它的目标时，开发人员必须重新审查先前的决定，试行一个新方法。

为了在开发过程中实现经济效益最大化，项目组必将试图选择为达到项目目标所需要的迭代。它应当以逻辑顺序排列相关迭代。一个成功的项目所经历的过程通常都只与开发人员当初所计划的有细微的偏差。当然，考虑到出现不可预见的问题需要额外的迭代或者改变迭代的顺序的影响，开发过程可能需要更多的时间和精力。使不可预见的问题减小到最低限度，也是风险控制的一个目标之一。

一个受控制的迭代过程的好处有很多：

ω受控制的迭代降低了在一个增量上的开支风险。如果开发人员需要重复该迭代，整个开发团队损失的只是一个开发有误的迭代的花费而已，而不是整个产品的价值。

ω受控制的迭代降低了产品无法按照既定进度表进入市场的风险。通过在开发早期就确定风险，人们就会在开发早期花时间来解决它们，而在这时，人们通常都不会像他们在开发后期那样匆匆忙忙。在“传统”方法中，困难的问题往往是在系统测试阶段才发现的，而解决这些问题所需求的时间通常又比开发进度中剩下的时间要多，因此，产品的延迟发布几乎是必然的。

ω受控制的迭代加快了整个开发工作的进度，这是因为开发人员很清楚焦点所在，而不是在一个漫长而不断变化的进度安排下工作，因此,他们的工作会更有效率。

ω受控制的迭代承认了通常都被忽略的一个事实：用户的需要和相应的需求并不能在一开始就作出完全的界定。它们通常都是在后续迭代中不断被细化的。此种作业模式使得适应需求的变化更为容易。

这些概念即用例驱动的、以基本架构中心的、迭代式和增量性的开发是同等重要的。基本架构提供了指导迭代中的工作的结构，而用例则确定了开发目标并推动每次迭代工作。缺乏这三个概念中的任何一个，都将严重降低“统一过程”的价值。这就好像一个三脚凳一样，一旦缺了任何一条腿，凳子都会翻倒。

以上我们已经介绍了三个主要概念，现在让我们来看看整个过程、它的生命期、成果、工作流程、阶段和迭代。

“统一过程”的生命期

“统一过程”将重复一系列生命期，这些生命期构成了一个系统的寿命。每个生命期都以向客户推出一个产品版本而结束。

每个周期包括四个阶段：开始阶段、确立阶段、构建阶段和移交阶段。正如上文已经讨论的，每个阶段可以进一步划分为多次迭代。见图 2。

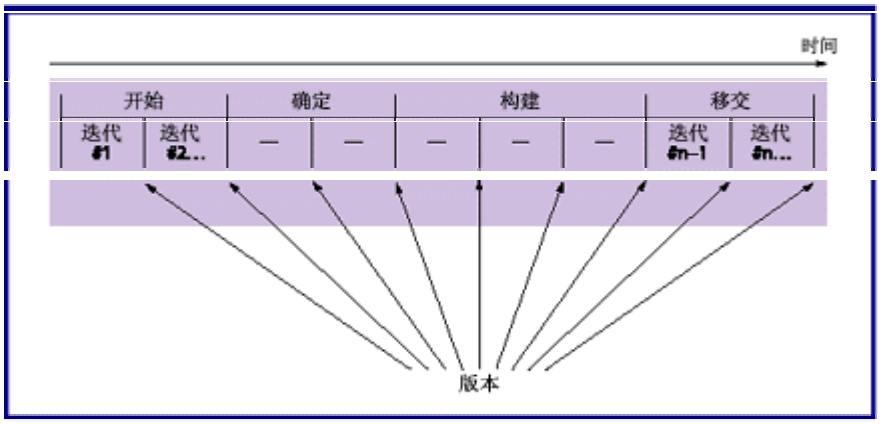

图 2. 一个开发周期及其阶段和迭代

产品

每个生命期都产生系统的一个新版本，而每个版本是一个能立即推向市场的产品。它包括体现在组件中的可被编译和执行的源代码，还有说明手册及其他可交付的相关产品。然而，最终产品也必须符合相关需要，此种需要并不仅仅是指用户的需要，而且指所有利益关联人的需要，也就是指那些将利用该产品来进行工作的所有人。软件产品应当不仅仅只是可执行的机器代码。

最终产品包括需求、用例、功能性需求和测试案例。它包括基本架构和可视模型(由统一建模语言构摸的产物)。事实上，它包括我们正在这里讨论的所有元素，因为它正是所有利益关联人（包括客户、用户、分析人员、设计人员、实现人员、测试人员和管理人员）要详细定义、设计、实现、测试和使用一个系统所需要的东西。此外，正是这些东西，才使得利益关联人能使用并逐代修改系统。

即使在用户看来，可执行组件是最重要的产物，但是，仅仅有这些产物是不够的。这是因为环境会不断发生变化。操作系统、数据库系统和作为系统基础的机器在不断发展。随着任务被更好地理解，需求本身也会发生变化。事实上，需求不断发生变化是软件开发领域里的诸常量之一。这样，开发人员就必须开始一个新的开发生命期，管理人员则必须为之提供资金。为了有效地实施下一个生命期，开发人员需要该软件产品的所有表现形式（图 3）。

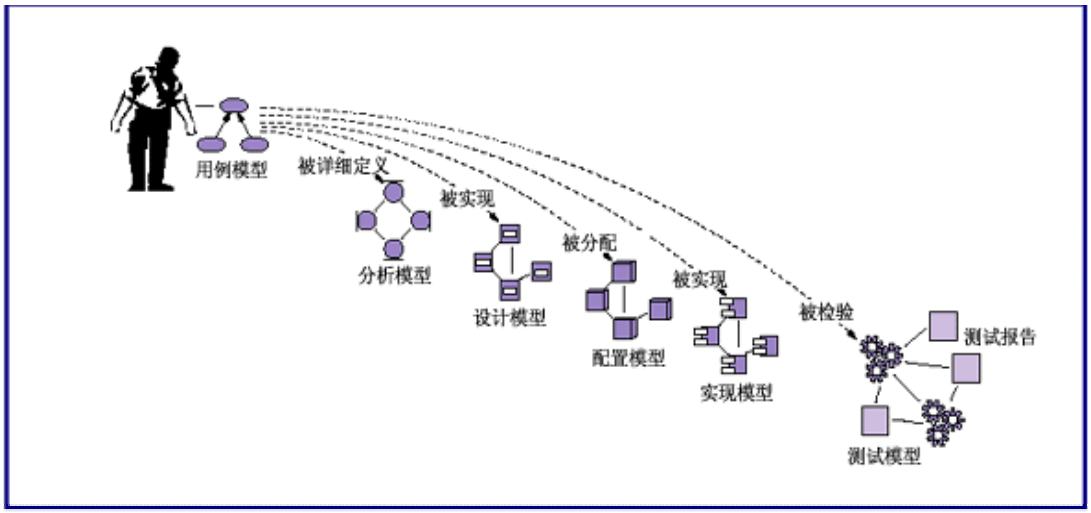

图 3. 用例模型与其它模型的依赖关系

ω用例模型，这个模型表明了所有用例和这些用例与用户之间的关系

ω分析模型，该模型有两个目的：更加详尽地细化用例，以及将系统的行为初步分配给一组提供这些行为的对象。

ω设计模型，该模型（a）以子系统、类和接口的形式定义系统的静态结构，（b）以子系统、类和接口之间的协作来定义用例的实现。

ω实现模型，该模型包括组件（代表源代码）和类向组件的映射。

ω配置模型，该模型定义计算机的物理节点和组件向这些节点的映射。

ω测试模型，该模型描述那些用来验证用例的测试实例。

ω还包括基本架构的表示。

系统还应当有一个域模型或业务模型来描述系统的业务上下文。

所有这些模型都是彼此相关的。它们一起表达了整个系统。在一个模型中的元素可向前或向后追溯到其它模型中。例如，（用例模型中的）一个用例可以被追溯到（设计模型中的）一项用例实现，再追溯到（测试模型中的）一个测试案例。可追踪性有利于理解和处理变化。

一个生命期中的诸阶段

每个生命期都持续一段时间。这段时间反过来又可以分成四个阶段（见图 4）。通过一系列的模型，利益相关人可以直观地看到这些阶段的进展状况。在每个阶段中，经理人员或开发人员还可能进一步对工作进行细分：分成多次迭代并确保增量。每个阶段都结束于一个里程碑。我们以一组获得的成果来定义每个里程碑。亦即特定的模型或者文档已经达到了预定的状态。

里程碑用于许多目的。最重要的是，在工作进入下一个阶段之前，管理人员必须作出某些关键决策。里程碑还能帮助管理人员以及开发人员在开发工作经过这四个关键点时监控工作进度。最后，通过对每个阶段上花费的时间和精力进行追踪，我们还能获得一组数据。这些数据在评估其他项目需要多少时间和人员、计划项目期间内需要的项目人员和根据这个计划来控制工作进度上是非常有用的。

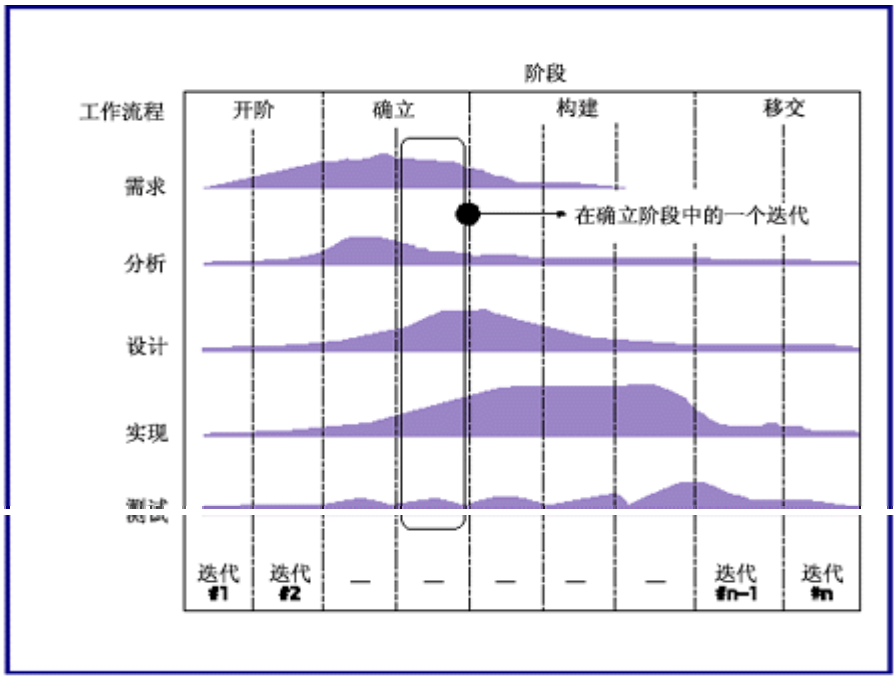

图 4. 五个工作流 — 需求、分析、设计、实现和测试以及四个阶段

图 4 左边一栏列出了工作流程——需求、分析、设计、实现和测试。曲线大致（不能认为此种曲线就完全代表了实际情况）表示了每个阶段中工作流程被执行的内容。注意，每个阶段通常都被分解为迭代或称小项目。如图 4 中的确立阶段所示，一次典型的迭代要经历全部五个工作流程。

在开始阶段，一个好的理念被开发成对最终产品的设想，并且该产品应用的业务用例被提出来。最重要的是，这个阶段回答了如下问题：

ω这个系统将为其每个主要用户做些什么？

ω该系统的基本架构应是什么样子？

ω开发这个产品的计划是什么，费用是多少？

包括最关键的用例的一个简化的用例模型回答了第一个问题。在这个阶段，基本架构还是试验性的，通常它只是一个包括关键子系统的轮廓而已。在这个阶段，最重要的风险被确认，并按优先次序进行了排列；对确立阶段进行详细的计划；对整个系统进行粗略的评估。

在确立阶段，产品中的大部分用例被详细地定义下来，系统基本架构也被设计出来。系统的基本架构和系统本身之间的关系是极为重要的。简单地说，基本架构就像一个裹着皮肤但是肌肉很少的骨架——只有一些能让该骨架作一些基本运动的必要的肌肉。而系统则是由骨架、皮肤和肌肉组成的有机整体。

因此，基本架构被表示为系统中所有模型的视图，这些视图共同表达了整个系统。这意味着，存在用例模型、分析模型、设计模型、实现模型和配置模型的基本架构视图。实现模型的视图包括能证明基本架构是可执行的组件。在这个开发阶段，在确立阶段确定的最关键的用例被实现。这个阶段的成果是基本架构基线。

在确立阶段的末期，项目经理要计划活动并估算完成该项目需要的资源。在这里，关键的问题是，用例、基本架构和计划是否足够稳定，风险是否已经处在有效控制之下，从而能够提交合同中规定的整个开发工作？

在构建阶段，产品被建构——肌肉（即完工的软件）被添加到骨架（即基本架构）上。在这个阶段，基本架构基线已经成长为一个羽翼丰满的系统。最初对产品的设想已经演化成了准备交付给用户的一个产品。在这个开发阶段，项目所需求的大部分资源被使用。尽管系统的基本架构是稳定的，但是，由于开发人员可能发现结构化系统的更好的方式，因此，他们可能会建议架构设计师对基本架构进行一些小修小补。在这个阶段的最后，产品已经包括了管理人员和客户同意为这个产品开发的所有用例。然而，它可能不是完全没有缺陷的。在移交阶段，会发现并修正更多的缺陷。这个里程碑的问题是，该产品是否有效地满足了客户的需求，可以向某些客户提前交付该产品？

移交阶段是指产品发布测试版的阶段。在测试版中，由少数有经验的用户来使用该产品，并报告他们

发现的缺陷和不足。开发人员则更正这些缺陷和不足，并将有些改进建议融入到向更大的用户群发布的一般产品中。移交阶段涉及诸如制造、培训客户人员、提供热线帮助和修正产品发布后发现的缺陷等活动。维护队伍通常将这些缺陷划分为两类：那些对后续的正式版本的正常操作有一定的缺陷，和那些可以在下一个正式版本中更正的缺陷。

“统一过程”是基于组件的。它利用了新的可视建模标准 UML，并依赖于三个关键观点：用例、基本架构、迭代和增量开发。要使这些观点可用，需求一个多层面的过程，该过程应当考虑到生命期、阶段、工作流程、风险缓和、质量控制、项目管理和配置控制等。“统一过程”确立了一个集成了所有这些因素的框架。这个框架就像一把雨伞，在它下面，工具提供商和开发者可以构建工具来支持该过程的自动化、支持各个工作流程、构建所有不同的模型，并在整个生命期和所有的模型中集成这些工作。

Date Jun 2000