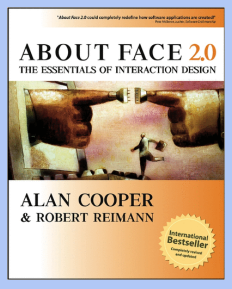
这一期审稿非常仓促，肯定有不少问题，抱歉！

——《非程序员》

第 46 期（2005 年 2 月）



脸

X 非程序员
-Programmer
软 件 以 用 为 本

联系：think@umlchina.com

http://www.umlchina.com/

Borland 购入外脑来驱动 SDO

[2005/1/19]

在 2002 年购买了 TogetherSoft 、Starbase 和 BoldSoft 之后，两年后 Borland 开始了它的第一次收购：TeraQuest。Borland 高级副总裁 Chris Barbin 说，从许多公司中选择了 TeraQuest，主要是考虑到该公司的 CMM 经验。

Borland 打算在应用生命周期管理工具、过程、服务中集成 TeraQuest'的专长，例如变更管理、项目计划和需求收集.

这次收购也意味着 Borland 拥有了 CMM 的两位作者：Bill Curtis 和 Charlie Webber。Curtis 已经被任命为 Borland 的首席过程官，负责帮助 Borland 的软件和服务朝 CMM 模型靠拢。

Borland 的 CEO Dale Fuller 认为，CMM 和 Borland 工具的结合能使客户组织从软件开发项目中更多的获益。

自从去年宣布 SDO 策略以来，Borland 一直在围绕它的 ALM 工具推行基于过程的开发。SDO 试图让业务用户对软件开发周期有更好的理解，从预算、移交到管理。Borland 相信这次收购将帮助软件公司在预算内准时移交符合客户需要的项目。

（摘自 cbronline，不得转载用于商业用途）

微软冷眼旁观 UML2.0

[2005/2/17]

当诸如 IBM 和 Borland 等软件开发商跟随 OMG UML2.0 （为模型驱动架构 MDA 制定的标准）前进时，应用开发领域一个重要选手却在一边冷眼旁观。

他就是微软。

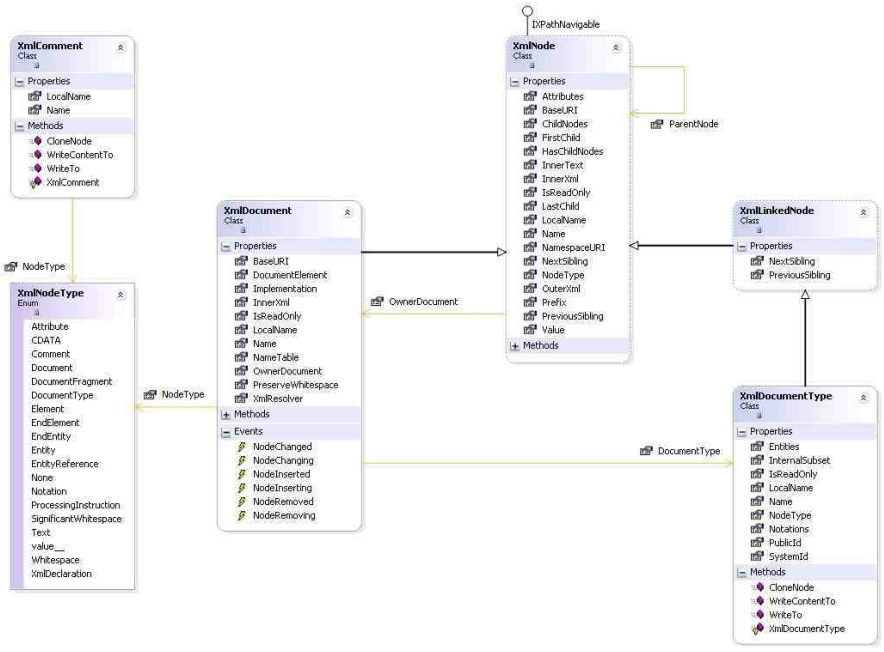
有 IBM“杰出工程师”头衔的 Alan Brown 认为，微软正在忙着通过微软的应用设计技术“试图创建具有自己特色的建模标准”。

微软官员 Keith Short 是 Visual Studio 组的架构师，他给出了回应。他说，微软实际上并不计划支持 UML2.0，但却尊重合作伙伴支持 UML2.0 的决定。

“我们对模型驱动开发的见解稍有不同。我们试图要做的是观察开发人员实际在做什么，并寻找在什么地方我们能够通过模型驱动的开发为他们增加价值。”

此前，微软认为 UML2.0 对开发人员的用处有限。

在即将到来的 Visual Studio 2005 Team System 平台中，微软为架构师提供的建模方法包括四部分：

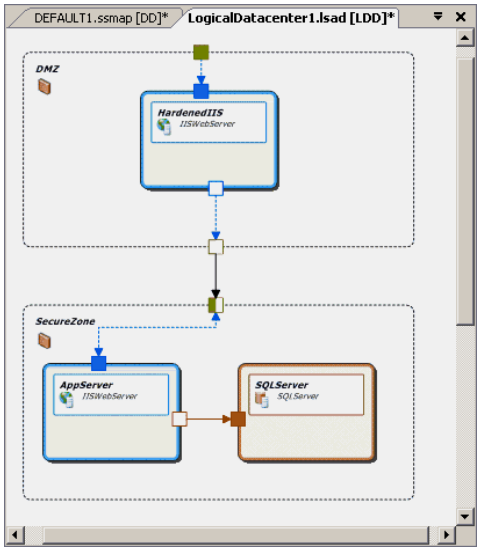


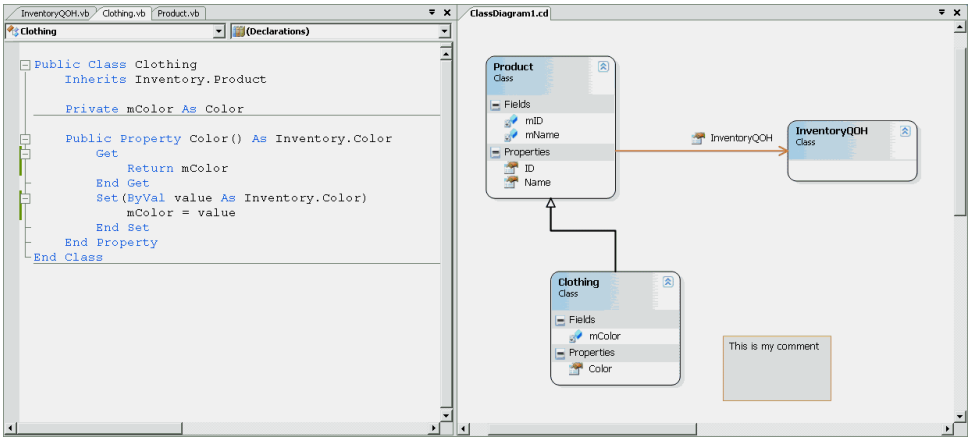
*应用设计师建模语言（Application Designer modeling language），定义可以提供或使用服务的可重用程序。

*Class Designer，可视化代码

*System Definition Designer，为部署进行服务的打包操作。

*Logical Datacenter Designer，提供数据中心的虚拟视图。





该工具根据微软的 DSL 技术构建。Short 认为，UML2.0 可以看作 DSL 中的一种。但 UML 并不足够精确，从而难以映射到诸如 ASP.NET 等底层框架上。

Short 介绍，如果愿意的话，开发人员可以使用微软的建模技术开发 UML2.0 图。这似乎是相当程度上的妥协。但很显然，如果业界领先的开发商如微软可以直接支持大多数人所认可的业界标准的话，将会更好。

（自 infoworld，袁峰 摘译，不得转载用于商业用途）

Compuware 和 SteelTrace 结盟

[2005/2/16]

Compuware 公司和 SteelTrace 公司今天宣布他们已经合并了各自在模型驱动开发和需求定义/管理领域的解决方案。Compuware OptimalJ 和 SteelTrace Catalyze 现在已经集成，支持在应用开发生命的全周期利用需求管理的信息。

集成方案在支持业务和技术需求的同时，通过保证应用对业务需求的完全符合来提高生产效率，提高质量。

Compuware 的副总裁 Bob Barker 认为，“超过 80%的 IT 项目失败，原因通常是定义失败的需求，以及不能形成项目业务方和技术方人员关于需求的有效交流。我们和 SteelTrace 的结盟有效地解决了这个问题，所有项目参与者都可以参与到需求过程中，并在一个模型驱动架构和开发的平台上实现了需求信息交互的自动化。”

Catalyze 的 SteelTrace 提供了项目生命周期中的需求驱动的开发方法，这意味着业务分析人员可以轻松地捕获需求的细节内容。另外，通过联合使用 Catalyze 和 Compuware OptimalJ，详细需求可以转换为 UML 用例的形式，后者正是模型驱动（model-driven）、基于模式开发（pattern-based development）的起点。

这种方法是全面的、基于 role 的，它使得业务单元和 IT 单元之间的协作不会给开发过程带来不应该的开销，同时提高了效率，并得到更高的软件质量。

SteelTrace 的 CEO Mark Melville 指出，“在业务分析人员和项目经理、最终用户对需求的定义之间存在一个 gap，在需求如何得到、解释、以及如何被开发人员实现上同样也存在 gap。当今企业的 IT 团队需要更有效地得到和定义需求，并高效地将它们传播到整个开发团队中。Compuware 和 SteelTrace 的结盟提供了目前唯一对项目开发中业务和技术人员给予同样支持的解决方案。”

（自 itweb，袁峰 摘译，不得转载用于商业用途）

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

"*About Face 2.0* could completely redefine how software applications are created!"
— Pete McBreen, author, Software Craftsmanship

ABOUT FACE 2.0

THE ESSENTIALS OF INTERACTION DESIGN

ALAN COOPER

& ROBERT REIMANN

当木匠看到锤子时，他不想和锤子讨论钉子的问题。他会直接用锤子钉钉子。在车里，如果司机想改变方向，他转动方向盘。司机喜欢通过合适的设备从车子和外部环境直接获得反馈：挡风玻璃外面的视野；仪表板的读数；疾驰而过的风声；轮胎压在道路上的声音；对侧向重力的感觉以及路面传来的的振动。木匠也希望有类似的反馈：钉子下沉的感觉，铁互相击打的声音以及举起锤子的感觉。

International Bestseller
Completely revised and updated

UMLChina 辅助教材。中译本即将由电子工业出版社出版，詹剑峰 译

审稿：国内首个交互设计网站 De Dream'

会写代码就会写需求

Jim Heumann 著，李胜利 译

讨论

本文来自 Rational Edge。通过使用已经用来书写代码的相同原则和概念，开发人员能够有效地服务于需求工程。本文评论了这些原则并解释了如何使用它们生成好的需求。

许多软件开发团队真的没有需求工程师，开发人员找出、书写和管理所有需求。这在资源高效方面看起来很有用。在系统停机的时间内，在认真的编码工作开始前的一段时间内，开发人员可以收集和书写需求。然而，这样做的缺点是程序员经常在书写好的需求的技术和工具上没有受过训练。这样做的结果，他们常常发生争执，工作低效，并且有时生成不合规格的需求规范。

书写好的代码，开发人员必须知道许多事情：像控制结构和调用约定那样的基本概念；至少一种编程语言，包含它的语法和结构；操作系统基础；和怎样使用像编译器、调试器和集成开发环境（IDE）那样的技术。好消息是他们能够利用这些技术的优势来书写好的需求。通过使用已经用来书写代码的相同原则和概念，开发人员能够有效地服务于需求工程。

让我们看一下一些开发人员能够使用的编程概念。

遵循一种结构

所有的编程语言都有一种结构。结构提到一个程序的不同部分是如何定义的和他们如何相互发生关系的。Java 程序使用类形成结构，COBOL 程序有不同的“DIVISION”，C 语言有一个主程序和多个子程序。

正如程序有一个特定结构一样，做需求也是如此。设想你将你写的所有 C 程序代码放入主程序中，它将变得难以理解和不可维护。同样地，如果你的需求规范一个巨大的没有特定顺序的列表，你将不能使用它。一组需求总有结构，无论你是否认识到。对需求中结构的最适宜处理方式是以不同的类型来组织它们，它们经常对应不同的级别。

为了理解不同类型间的区别，让我们看一下保险索赔处理应用中的四个需求例子：

1. 我们必须能够减少索赔的单据。
2. 系统必须能够自动检查索赔表单中合格的条款。
3. 系统将根据他/她的社会安全号码决定一个索赔是否已经是一个已注册用户。
4. 系统能支持达到 100 项索赔的同时处理。

你的直觉可能告诉你关于每一条需求有一定的差别。第一条是非常高级别的；它表达业务需要，甚至没有提及一个系统。第二种表达系统应该做什么，但仍然是在较高级别的；它仍然太广泛而不能直接转化为代码。第三种是较低级别的需求；它真正提供足够详细的关于软件必须做什么，使你能够写代码。另外第四种需求，尽管非常详细，不告诉你系统必须做什么，相反它规定系统必须如何快。这些需求是非常典型的，在你与用户和其他涉众谈话时你将得到的。或许你能明白为什么将它们都放到一个大的、不分类的列表将导致混乱。

通过将它们分不同目录或不同类型放置，你能得到更加可用的需求，例如：

- 业务需要
- 特性
- 功能性软件需求
- 非功能性软件需求

这些是在 IBM Rational 统一过程或 RUP 中建议的类型。决不是只有这些可能的类型，但是它们代表一种有用的方式。在你的项目早期，你应该决定使用什么类型。然后，随着你从涉众那里收集信息，决定他们描述的是什么类型的需求，并写出需求。

注意你能够以两种格式中的一种详细说明功能性软件需求：声明式和用例。上面的第三种需求是以声明形式声明的；它相当粗粒度并使用“应”语句。另一种形式是用例。它也详细说明了系统应该做什么，在一个足够低的级别上能够根据它写代码。然而，它提供更详细关于用户和系统将如何交互实现部分价值的上下文关系。（关于用例的更详细内容见下面。）在你开始为一个项目收集需求前，你应当决定你想用哪一种类型的功能性需求，然后坚持下去。

使用确保质量的实践

你知道写出好的和坏的代码都是可能的。有很多编写坏代码的方式。一种就是使用像 `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff`, 和 `do_args_method` 那样的非常抽象的函数和变量名。这些名称不提供关于方法和过程做什么的任何信息，逼迫读者去挖掘代码以发现这些信息。其它不好的实践是使用像 i, j 和 k 那样的单字母变量名。你不能使用简单的文本编辑器容易地查找它们，并且函数不清晰。

当然，也有很多方式，你能书写不好的需求。也许最坏的过错是含糊。如果两人用不同的方式解释一条需求，这个需求就是含糊的。例如这里有一个来自实际需求规范的需求：

该应用在大量（*numerous*）用户同时登录时必须极端（*extremely*）稳定。必须不能牺牲速度。

单词 *extremely* 和 *numerous* 对广义的解释是开放的，所以这个需求是含糊的。实际上，为了实现明确性，你应当从三个高度规范的需求真正表达它：

1. 系统失败的平均时间必须不能多于每星期一次。
2. 系统将支持 1,000 个并发用户同时查询数据库，同时不能损坏或丢失数据。
3. 系统平均响应时间在达到 1,000 个并发用户时将不少于 1 秒。

质量需求有很多更详细的属性；参见 IEEE 详细信息指南。

细心编写注释

写得好的程序包含增加代码信息解释它做什么或为什么这样书写的注释。好的注释不解释代码如何做，这些从代码本身看来很明显，相反地，它提供帮助用户，维护者和评审员去理解代码做什么和保证编码质量的知识。同样，需求也有属性，使需求更易于理解和使用的信息。当你得出需求时，你也应当发现属性信息。例如，

一个重要的属性是来源：需求来自何处？如果你需要回去得到更多的信息，对你的需求来源保持跟踪将节省大量的时间。另一个属性是用户优先级。如果一个用户给你 50 条需求，它也应当让你知道每一条需求与它相关的需求如何重要。然后在项目周期的后期，当时间越来越少，你意识到不能满足每一条需求时，你至少知道哪一些是最重要的。

正如没有规则明确地告诉你在代码里写什么注释一样，这里也没有通用的“正确”属性列表。来源和优先级几乎总是有用的，但是你应当定义适合你项目的其他属性。在你收集需求时，尽量预先估计出当你开始设计系统和编码时团队可能需要什么信息。

熟悉语言

很明显，开发人员必须熟悉他们用来编码的语言，无论它是 Java, COBOL, C++, C, Visual Basic, Fortran 或很多其它语言的一种。写出好的代码，你必须理解语言的细微差别。尽管在每一种语言中基本的编程概念是相同的，他们采用不同的方法完成特定的操作。例如，Java 循环结构使用“for”，Fortran 是“DO”。在 C 语言中你通过给出它的名字和参数来调用子程序；在 Fortran 中，使用一个 Call 语句。

很好地书写需求你也必须熟悉语言。大多数需求使用自然语言（法语，英语等等）。自然语言非常强大，但也非常复杂；有时在写作上没有受过训练的开发人员在写作上难于沟通复杂的想法。这里我们没有篇幅来教写作课程，但是一些指南能帮助我们。

首先，为声明的需求使用完整的句子（例如，这些内容使用”应”语句或相同的结构表达）；检查每一句中的主语和动词。

然后，使用简单的句子。只包含一个转达简单意思的独立子句的语句易于理解并且易于检验和测试。如果你的需求对简单的句子来说似乎太复杂，尽量将它分解成你能更易于定义的较小的需求。组合句和复杂句可能介绍依赖（分支）；换句话说，它们可能描述依赖于某种动作的变量。结果经常得到不必要复杂的需求，使测试变得困难。

简单句：系统应能显示轿车绕场一圈持续时间。

组合句：系统应能显示轿车绕场一圈持续时间，并且时间的格式应是 hh:mm:ss。（这是两个需求：一个是指明系统应当做什么的功能需求，另一个是指明时间格式的用户界面需求）

复杂句：系统应能在完成一圈后的 5 秒内显示轿车绕场一圈的持续时间。（这也是两个需求；一个功能需求，一个性能需求。）

书写足够的基于组合的或复杂的例子，你将不得不将这两个需求分开。为什么不做得简单些并且这样开始呢？这里有一种转换复杂句子为简单句子的方法：

系统应能显示轿车绕场一圈的持续时间。

持续时间显示的格式应是 hh:mm:ss。

持续时间显示应在一圈结束 5 秒内实现。

注意，除了更易于测试以外，这些需求易于阅读。

这里有一些书写优秀需求的技巧：使用一致的文档格式。你已经有编写代码的格式或模板。也用来书写需求。不变的是关键元素；每一个规范文档应当使用相同的头部，字体，缩排风格等等。模板能帮助我们。它们作为列表清单非常有效，开发人员书写需求不必从草稿开始或重新发明看起来很好的书写规格的工具。如果你想要例子模板，RUP 有很多。

跟随指南

大多数开发团队使用如下的编码指南：

- 在单独的文件中放置模块说明和执行说明（C++）。
- 在代码块中缩进代码（Java）。
- 在每一个 WORKING STORAGE SECTION 的开始放置高活动性的数据元素(COBOL)。

你也应当使用编写需求的指南。例如，如果你决定使用用例说明软件需求，那么你的指南应当告诉你如何书写事件流。用例的事件流解释系统和用户（执行者）之间大量的交互工作。指南应当描述在主流程（success scenario）和分支流程（exception scenarios）中进行什么和如何构建这些流程。它们也建议两种流程的长度和内部的单个步骤。如果你决定使用传统的，声明式的需求，指南则应解释如何编写它们。幸运的是，在 RUP 和其他有关的资源中，这些指南中的很多都是现成的，所以你不需亲自编写。

理解操作环境

开发好的代码，你必须了解你的系统将要运行的机器平台和如何使用它的操作系统。如果是 Windows 系统，你必须了解 MFC 和.Net。如果是 Linux 系统，你必须了解 UNIX 系统调用。

要擅长编写需求，你必须不仅了解操作系统，还要了解操作人员。你必须也了解用户界面和用户。Java 开发人员考虑类路径（classpaths）；需求编写人员考虑让人们使类（或工场）在正确的路径上。

得到需求是一向以人为本的工作。你不能虚构需求；你从其他人那里收集需求。这也许对内向的开发人员来说是一个挑战，但如果他们正确地应用他们的现有技能，他们能够成功。

常常，用户不知道他们想要什么；或知道，但不知道如何去描述它。开发人员有能帮助他们的能力：他们经常不得不破解来自编译器的神秘的和模糊的错误信息。例如，Java 开发人员写一个小程序（applet）可能遇到这个消息：

```
load: com.mindprod.mypackage.MyApplet.class can't be instantiated.

java.lang.InstantiationException: com/mindprod/mypackage/MyApplet
```

这是什么意思呢？如果开发人员不能确定，他将通过察看代码、编译器文档查找原因，甚至通过类似Google®的搜索引擎。最后，他将发现：他的代码缺少applet的缺省构造代码。

如果你正在收集一个天气预报系统的需求，并且用户告诉你系统应该能“显示在超过 200 平方英里的区域的大气层内的不同高度的风速和风向，使用带有小尾的标准箭头”，你应当挖掘深层次内容。你也许请求看一下来自相同系统的报告，查询气象学书籍或请求其他用户描述更精确的需求。你应当一直调查直到拥有关于期望功能的足够详细的信息。然后你应当重新陈述需求以使它们清晰和明确，提供足够详细的信息去支持设计。

启发需求的另一个技巧是尽量不要去询问诱导性的问题。尽管你可能拥有关于用户想要的内容的很多想法，如果你将这些倾盘倒出，你可能不能得到用户真正想要的真正画面。相反，问一些类似于“你愿意看到如何分开显示数据？”这样能自由回答的问题，避免类似于“你希望看到合在一起的气压和温度的图表吗？”的问题。

遵循确定的原则

设计和书写优秀程序的其他原则是信息隐藏，耦合和内聚。对于书写需求每一个都有它的相对物。

信息隐藏

这涉及到一个原则，即了解只言片语代码的，或者甚至是了解代码的用户/调用者不能访问数据的内部细节。所有对数据的访问和修改应当通过函数调用。通过这种方式，你能够不影响调用程序而改变内部数据结构。

这对需求来说也是一个好原则，特别是如果你用用例表达它们。正如上面我们注意到的那样，用例有事件流。写得不好的用例经常有充满数据定义的事件流。考虑这个“管理购买请求”用例的基本事件流：

基本事件流：

1. 系统显示所有未决的购买请求。
2. 每个未决的请求将包含关于请求的下列信息（限制为字符（char））
   - Approval ID (仅供内部使用)
   - PO #
   - Reference ID
   - Distributor Account Abbreviated Name
   - Dealer Account Name (first 10)
   - Dealer Account Number
   - Reason Codes
   - Amount Requested
   - Request Date
   - Assigned to (内部信息)
   - Comments Indicator

3. 授权管理员能做下述事情之一：1）批准 2）拒绝 3）取消 4）指定。他/她选择 1）批准。

4. ...等等，直到所有步骤完成。

显示的 15 行中，7 行主要告诉什么数据与一个挂起的请求一起。这是重要的信息，但是它使用例中发生的事情变得模糊。一个更好的方案是隐藏数据到别处。这些步骤则看起来像这样：

基本事件流：

1. 系统显示所有挂起的购买请求。

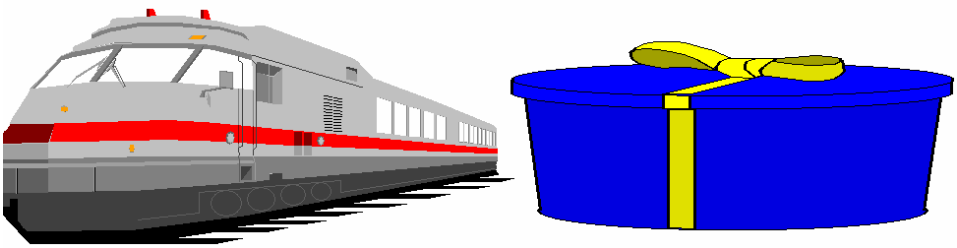
2. 授权管理员能做下列事情之一： 1）批准 2）拒绝 3）取消 4）指定。他/她选择 1）批准。

3. ...等等，直到所有步骤完成。

*挂起的购买请求*是有下划线和倾斜的以表明数据在别处定义（经常在特定的用例需求部分或可能在全局中定义）。这使得代表真正的功能需求的事件流更易于阅读和理解。

耦合与内聚

对于编码人员，耦合指的是一个程序中的单个模块应当尽可能独立的原则。一个模块中的处理应当不依赖于其他模块内的内部工作的知识。内聚指的是在一个给定的模块内，所有代码应当为一个单一的目标工作的原则。这些原则使得一个程序变得既易于理解也易于维护。



这些原则也适用于需求，特别是对于用例。用例应当独立（例如少或没有耦合）。每一个用例应当指定一个大的功能块并且显示系统如何为执行者提供价值。执行者的焦点是重要的；你能指定系统为执行者做哪些事而不担心用例按序排列的顺序。

一个用例中的所有功能应当和执行者目标的完成有关（即，高内聚）。在一个典型的 ATM 系统中，一个用例应当为“取钱”，另一个为“转帐”。每一个用例集中于单一目标。如果你想合并这些功能到一个单一用例中，它就会有低内聚（和不合适的依赖）。

然而，当心许多用例初学者走过了头并生成了太多的低层用例。我曾经看到一个拥有名为类似于“修改数据”的 150 个用例的银行债务收款系统。这个项目有一个 10 人团队并且计划持续大约一年。然而，因为这些用例太碎，这个组织有许多前进的障碍。它们描述不指明用户价值的低层功能；它们难于理解也难于使用。每一个用例非常内聚，但是也因此有高程度的耦合。提升到类似于“收集数据”的更明确的活动层次，能产生适当的耦合和内聚程度。

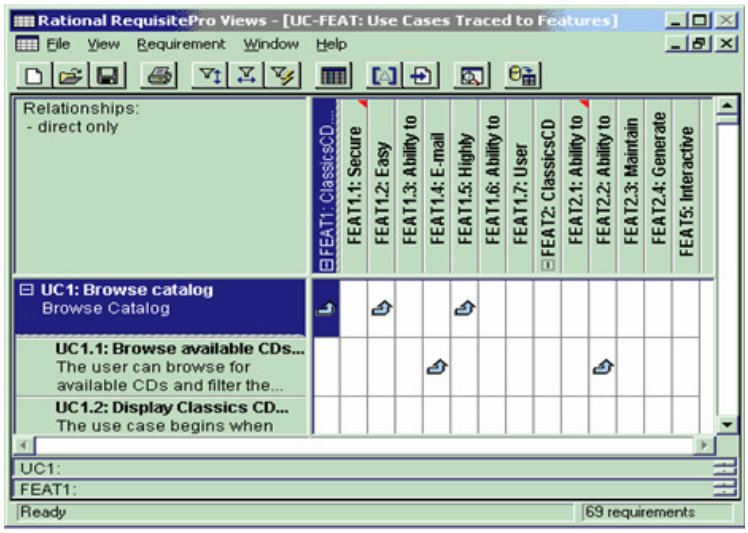
使用自动化工具

开发人员使用工具完成他们的工作。手工编译代码是不可能的，所以他们一直使用编译器。与组织争相提高效率一样，集成开发环境（IDE）正变得更全面和更受欢迎。用于代码生成和反向工程的 UML 建模工具也广泛使用。

书写和管理需求的自动化工具也是非常有用的。达到自动化的第一步是使用文字处理器。使用微软 Word 软件记录需求，相对于把需求保持在白板或餐巾纸上，已经是迈出了重要的一步，。文字处理器提供拼写检查，格式化和作为需求“容器”功能的文档模板。你能够将它们保存到一个地方并将它们送去以供评审。然而，文字文档不能帮你排序并过滤大的需求列表或建立追踪。

类似于 Microsoft Excel 的电子制表软件更利于高级的排序和过滤，但以丢失上下文关系为代价。你也能把工作薄用于追踪，但是操作仍是手工。

基于数据库的自制工具的优势和电子制表软件有一些相同；它们经常在过滤和可跟踪上非常优秀。然而，因为工具的功能经常特定于一个指定的项目结构，它难于适应新的变化。它也也许缺乏完整和最新的文档。



专门设计用于需求管理的工具（RM 工具）通常比 Word 或 Excel 有些复杂，但是不像一个编译器或 IDE 那样复杂。它们也提供重大的优势：

- 几乎所有的RM工具允许你倒入现有的需求文档到工具中。如果工具和你的文档是优秀的，工具将能自动找出文档中的实际需求。IBM Rational RequisitePro® 提供从文档中的需求到存储于工具（或后台数据库）中的需求的动态链接，所以，需求总是“活”的。

- RM 工具允许你很容易地生成需求类型并赋予它们属性。这允许用于排序和过滤，给用户一个灵活的查询机制非常容易发现感兴趣的需求。这些工具也允许你根据属性值排序需求。例如，如果有 “用户优先级”和“风险”属性，你可以生成下列查询：“显示所有高优先级和高风险特性。”这应当能帮你决定在早期的迭代中实现那些特性以保证你不遗漏重要的功能，并能减轻项目早期的风险。

- 优秀的 RM 工具提供需求间的可跟踪性；一个真正优秀的工具提供到其他工件和事物的可跟踪性，例如设计和测试。可跟踪性是一个帮你验证系统的重要能力。

- 一个需求管理的最好实践是跟踪每一条需求的历史。RM 工具帮助完成这些。它不仅告诉我们需求的起源，也告诉我们决策为什么制定和由谁制定的。

- RM 工具也能在基线方面帮助我们：在你能与未来快照比较的特定时间点上取一个“快照”。基线提供在其上工作的一个静态集合。 它也提供你能参考的项目周期中的分支点，你应当想拷贝一份以用于新的开发工作。

所以，像编译器和 IDE，RM 工具帮助开发人员做那些不容易手工做的事情（或可能一点不可能的），并帮助他们实现更大的效益。

管理变更

优秀的开发组管理代码变更。开发人员根据设计和规格说明编写代码；他们不会根据自己的判断增加特殊的内容。另外，代码处在源代码控制下；当他们更改代码时，开发人员指明为什么这样做，他们也建立代码基线，集成代码，并且测试用于发布。

需求也需要变更控制。更改是不可避免的；为此制定计划是重要的。在项目开始的时候，需求经常是（和适当地）在一种不断变动的状态中。但是在一些要点上，在更多代码编写前，在沙堆上划出一条线并建立一个需求基线是重要的。那之后，需求变更必须经过批准，典型情况下由一个变更控制委员会（CCB）批准。然而，一些组织仅指定一两个人来定期复审变更请求。

那些没有需求变更控制流程的团队不得不在所有岗位上收到变更请求，并且经常难于说“不”。如果你想避免不得不经常重写代码以与需求变更保持同步，以一个 CCB 开始。评审变更的流程可以帮助确认你们做的这些变更将提供业务价值，并且每一个人理解他们的影响。没有业务价值的变更只是消耗资源，带来的报酬很少。同样，没有业务价值的变更也会对现有的需求、设计、代码产生重大的影响，会带来不必要的工作量。

评估变更的潜在价值和可行性的另外一种方式是通过可跟踪性。它允许你跟踪需求的合理性并理解所有相关的事情。通过跟踪软件需求到较高层次的业务，或者用户需求，你能确认它有价值。如果你不能以这种方式跟踪需求，软件需求可能没有业务合理性。另外，通过跟踪从高端到底端的需求，一直到设计、编码和测试，你能容易地看出需求变更的影响。一个可跟踪的矩阵或更好的 RM 工具将显示所有相关事物，并且提供必要的知识以决定变更请求是否值得去做。

计划编制

大多数成功的软件开发项目都有一个指导项目的计划，指明谁做什么，事情将怎样做和里程碑是什么。架构师通常生成一个提供系统架构的全面概览的文档。它也能够使架构师和其他项目团队成员关于架构上的重大决定和指导开发人员实现系统方面保持联系。

与这些计划文档一样，需求管理（RM）计划能对项目提供巨大的好处。对编写需求的开发人员，该计划描述必要的需求，包括需求类型和各自的属性。它指明开发人员必须收集的信息和控制需求变更的机制。

正如我们以前看到的那样，需求类型可能包含业务需要、特性和功能性和非功能性的软件需求。你可能也有用户需求和市场需求。计划鼓励你考虑和指明你需要的需求类型，它反过来帮助确保编写的需求的一致性和可读性。

也正如我们以前注意到的那样，属性提供帮助你更有效理解和使用需求规范的追加信息。

RM 计划也描述你将要使用的文档。RUP 推荐三类：愿景文档、用例文档和不能在用例里描述的追加规范文档。

RM 计划也描述了变更管理过程，以使项目里的每一个人都理解它。

如果你已经在一个没有 RM 计划的项目中工作，你可以自己编写一份。它不一定很长：一页或两页也许就能包含你需要改进一致性和高质量需求的所有的信息。

优秀的开发人员能够编写良好的需求

在开发人员不得不得出和记录需求时，通常用来生成代码的原则和实践能够很好地为他们服务。如果你是一个认为在编写高效需求上没有背景和教育的程序员，我希望本文已经说服你了。简单应用你每天使用的知识和原则做这项新的任务，你会成功的。

补注

[1] IEEE Recommended Practice for Software Requirements Specifications, Software Engineering Standards Committee of the IEEE Computer Society. Approved 25 June 1998

[2] In RUP, look under Artifacts -> Requirements Artifact Set -> Use-Case Model -> Use-Case Modeling Guidelines. Also see IEEE Recommended Practice for Software Requirements Specifications, Software Engineering Standards Committee of the IEEE Computer Society. Approved 25 June 1998

[3] Unified Modeling Language, or UML, includes the concepts of extend and includes. You can learn about these advanced topics in IBM Rational Unified Process. Another good reference is Use Case Modeling by Kurt Bittner and Ian Spence (Addison-Wesley 2003).

本文最初发表于 IBM developerWorks & RationalEdge，经 IBM developerWorks 允许刊登。

The Unified Modeling Language Reference Manual, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

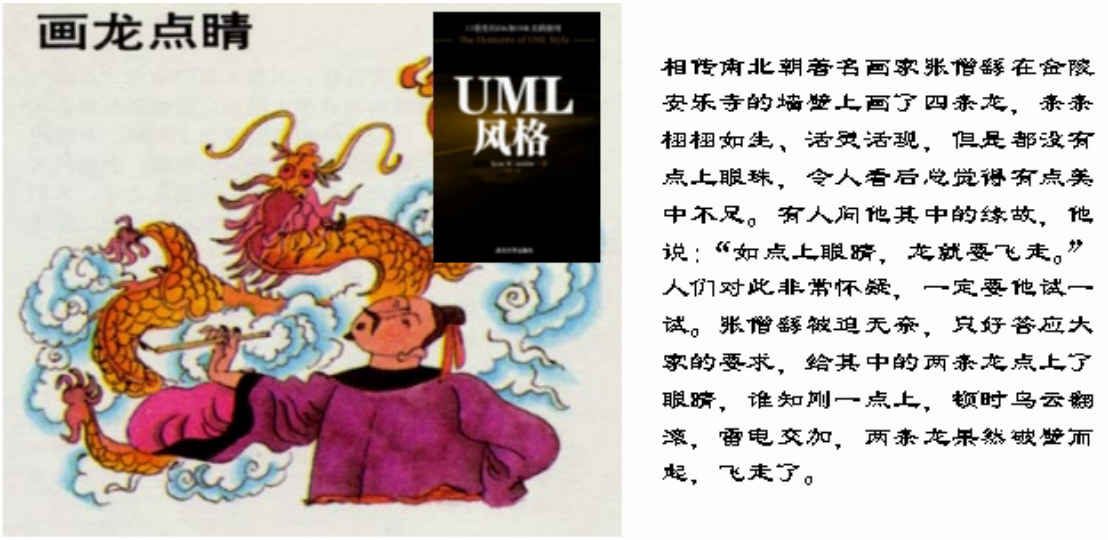
Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版



它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

它不讲过程，它假设读者已经了解某种开发过程。

它只是在读者已经了解方法、过程和工具的基础上，提醒读者在绘制 UML 图时需要注意的一些细节。

在这本类似掌上宝小册子中，Ambler 提出了 200 多条准则，帮助读者在画龙的同时，点上龙的眼睛。

精确用例

David Gelperin 著， James 译

讨论

概述

本文讲述了精确用例，精确用例进一步细化了执行者选项的说明，明确了路径的条件。精确用例使用一种结构化英语语法的精确活动描述语言/说明符，设计这种说明符的目的是让应用专家透彻地理解。为了同时支持手工和自动测试设计，需要提供足够的信息和结构，与面向对象概念有些不同的是，它还另外提供了详细的例子，开发过程，模型说明，和测试范围。本文主要适用于对 UML 有基本理解的读者。

1 介绍

用例可以用图[16]和文字[6]来描述。图主要用来说明执行者与用例之间以及用例与用例之间的关系，文字主要用来说明执行者/系统的交互，包括最初的概要性说明，以及细节上的精确描述。图和概要性描述在最初抽象时很有用，而进一步的细节分析和函数说明需要用准确的文字来描述。

用例和他们当前明细化的表达方式为手工测试设计提供了有用的信息，当前的用例表达方式不能满足包含结果检查的自动测试设计，本文讲述了一种满足测试基于文字的准确表达方式的用例：精确用例。这种方式明显增加了很多细节部分，从而降低了误解带来的风险，并且提供了系统开发以及手工测试的坚实基础。

虽然Constantine[4，5]也针对同样主题发表了一些方法，但是它更注重于测试而不是使用为中心的设计。精确用例来源于Constantine基本用例的定义，因此可能由此有效地支持用户界面设计.

另外，应用精确用例会为以下提供坚实的基础：基于使用的阅读，建立操作框架[23]，产生用户文档，提高测试精度以及开发工作量估算[24]。

本文的其余部分是如下组织的：第2和3部分讲述了测试需要的信息以及到达那些需要的方法。第4部分讲述了精确用例的架构。第5部分提供一个真实的例子。第6部分解释了例子的各个方面。第7部分详细说明了用来精确描述动作路径的结构化英语结构。第8部分讨论了路径描述中可选择的格式。第9部分提供了一些建模的指南。第10部分提供用例相关的过程。最后，第11部分提供了覆盖测试的标准。第12部分提供相关的工作。

2 测试需要

有效的测试必须包括使用场景。用例是使用信息和使用测试思路的重要来源。要设计以基于使用的测试，下面这些与简单使用场景有关的信息是必需的（但是不充分）：

- 场景中用户动作和系统动作的顺序
- 场景之前的不变量，以及前置条件
- 确定相关场景中活动对象的初始化状态
- 确定场景触发器事件
- 确定场景的备选输入
- 确定处理结果的备选布尔值决定场景流向

如果用例说明是非形式或者半形式的[3，6，16，19，23]，那么高质量基于使用的测试设计将遗留部分手工处理的过程并且上面的一些信息会丢失，或不清楚、不正确。和自动校验用例一样，形式和准确是自动测试设计的前提。

Cockburn[7]总结说人们不喜欢写形式的用例。他坚决认为，虽然形式或者非形式的用例都不能自动产生系统设计，用户界面设计，或者特征表单。形式用例似乎（做起来）费力，没有什么益处。但是，形式的用例能支持自动设计测试（手工或者自动的）脚本。另外，在精确上的尝试可以发现非形式描述中的潜在问题。有时，由于产品风险的本身和其涉及的范围必须保证是形式的。

注意，如果代码已写完，软件的开发必须到达到形式且精确。因此问题不是是否应该形式和精确，而是什么时候。因为模糊经常是失败的种子，在开发的初期引入形式和精确可以减少被开发者遗漏或者测试的信息引起的风险。

3 满足测试需要

精确用例位于被称为“精确使用模型”的更大规模描述的核心位置。除了用例以外，这个模型还包括：

- 应用程序名字 & 概要
- 人员描述 - 包括人员图和执行者简介
- 应用域描述 - 包括基本类描述，状态，转换规则，类关系，域不变量，功能限制
- 定义派生属性和条件 - 可选的
- 条件之间的依赖关系 - 可选的
- 非功能需求 - 可选的
- 工作分解结构 - 可选的
- 动作约束 - 可选的[12]

另外的信息提供了精确用例的相关上下文，包括与应用程序的对象描述一样的有关执行者的背景描述，它们的关系，状态变化，约束。这些额外的信息提供了用例的条件和动作的语义。

“精确使用模型”和Larman[21]描述的说明是一致的。在[9]中可以找到一个简单的精确使用模型--图书管理系统。

推动精确用例的目的就是让不熟悉面向对象概念的应用专家最大化地理解用例，并且提供足够的信息和结构来支持手工和自动的测试设计。

4 精确用例（PUC）的架构

精确用例的表达方法受 Cockburn 写的关于用例的书[6]的影响很大，但是有几部分与他的指导方针不同。它还吸取了 DeMarco 的结构化英语[8]。

精确用例包含下列核心信息：

1. 用例名称和可选的摘要/上下文图

2. 风险因素

发生的频率：

[频率范围 每次时间间隔和事件]

失败的影响，可能 & 最坏的后果

[高，中，低]

3. 用例条件

不变量和前置条件

4. 动作的交互路径

成功路径

调用路径

备选路径

单点路径

异常处理程序（EH）

替代备选（RA）

单点选项（SO）

定点范围内路径

非扩展性备选

多点选项(MO)

功能出口(UE)

应用出口(AE)

扩展

功能选项(UO)

应用选项(AO)

用例包含一条成功路径，0到多条调用或备选路径。

其他的信息包括版本号，建模者，来源，有效期，触发事件（例如 详细说明次数），撤销用例，或者后面可能包含在这个用例中的继承用例。另外还有[4]和[6]中的内容可能需要包含。

成功路径描述包含如下核心内容：(1)名称，(2)成功的条件 例如 不变量，前置后置条件， (3)执行者， 以及 (4)交互步骤。 每个使用场景(例如：通过用例的完全路径)从成功路径的第一步开始必须满足这个用例不变量和前置条件。成功场景，包括那些执行备选路径，成功地从路径的最后一步退出，从而满足所有的成功条件。

调用路径通过动作“does< 调用路径名 >”来调用。它的核心组成信息包括：(1)名称，(2)路径的条件， (3)执行者， 以及 (4)交互步骤。调用路径，可以看成共同的封装路径段，可以简化成功路径的描述。

备选路径有八大类。所有的备选路径的核心组成信息包括：(1)名称，(2)嵌入点的位置或范围，（3）备选条件，（4）执行者，以及 (5)交互步骤。备选路径种类包括：

1. **异常处理** 处理妨碍系统路径正常完成的非正常情况。检查异常条件，如果成立，调用相应的处理。这个处理可以补偿不正常的条件，或者执行该情况下的其他步骤。

2. **替代备选** 是由一个或多个相互独立的备选路径组成的集合中的元素，必须被遵循的备选（举例说明：补偿性路径）路径之一，例如 支付备选路径。

3. **单点选项** 受到一组警戒条件（例如不变量以及前置条件）的限制。如果满足了警戒条件，这个选项将被包括到后面的流程中。举例，结账时处理优惠券。

4. **多点选项** 单一的选择，它可以中断执行者在处理多点时的处理流程。例如 结账退回一件商品。

5. **功能出口** 那些在执行者的处理流程中，可以结束交互的备选路径。它们是软件平台提供的，可以让人在任何时候选择结束交互。

6. **应用程序出口** 由应用程序提供的备选路径，它可以中断交互。可以永久或者暂时退出。永久性退出伴随着显式中止，来通知系统它要中止应用程序，也伴随着隐式中止，例如放弃正在购物的购物车或者顾客消失。暂时性中止，例如暂停销售，伴随着暂时中止交互。意味着在指定时期内，状态信息被保存起来等待执行者回来。如果执行者不回来，保存的信息将会被删除，并变成永久性中止。

7. **功能选项** 系统支持的扩展用例。由于是扩展用例，它包含的核心信息与用例的一样。

8. **应用程序选项** 来自相同或者不同应用程序的扩展用例，由于是扩展用例，它包含的核心信息与用例的一样。

可以在[11]中找到上述八大类更详细的描述。

5 一个精确用例的例子

下面的例子来自[22]中的叙述，其他的可以在[10]中找到。

用例名：在预定航班上得到[新的]座位

风险因素：发生频率：0 ~ 2 次/每次预定

失败影响：可能的情况 — 低，开放的座位实际上已被占用。

最坏的情况 — 中等，开放的座位和大型飞机上的昂贵座位混在一起，会惹恼重要的旅客。

用例条件 ：

不变量：无

前置条件：对预定系统来说，状态是活动的

对旅客来说，系统访问状态是已登录

交互 ：

旅客	基于 web 的航班预定系统
1. 请求安排座位	2. 请求一个预定的位置
3. 提供（较正过的）可供选择的预定位置	4. 查询预定
重复 执行 步骤 3，4 直到预定被落实或者所有的预定策略都被执行过了	

	5. 提供座位选择，除非 a. 预定不能落实 或者 b. 座位已预先安排了或者 c. 没有座位是空的 或者 d. 没有座位是可分配的
6. 选择可以被选择的座位	7. 安排被选择的座位 unless 没有座位可以被选择
	8. If（对预定，座位以前安排过） return 前面的座位给清单 后置条件--对航班来说，前面分配的座位是有效的 endif
	9. 确认安排 成功退出

成功路径条件

不变量 ：

对于预定系统，状态是活动的

对于旅客，系统访问状态是已登录

对于旅客 & 航班，预定是存在的并且可以查询

前置条件：

对于航班，某些座位是可以分配的

旅客想得到或者改换座位安排

后置条件：

对于航班，可以选择的座位已被选择

对于预定，被选择的座位已被分配

备选路径

异常处理（EH）：

EH 1 - 5a(预定找不到)

EH1 不变量：

对于预定系统，状态是活动的

对于旅客，系统访问状态是已登录

对于旅客 & 航班，预定是存在的并且可以查询

旅客	基于web的航班预定系统
	1. 提供关于预定的帮助 失败退出

EH 2 - 5b(座位已经安排)

EH2 不变量：

对于预定系统，状态是活动的

对于旅客，系统访问状态是已登录

对于旅客 & 航班，预定是存在的并且可以查询

对于预定，座位以前安排过

旅客	基于web的航班预定系统
	1.提供　改变座位安排的功能
2.可以： a.改换座位安排 b.不改变	3. If（旅客要改变座位安排） **继续**; else 提供帮助，并且**成功退出**

EH3 - 5c or 5d（没有座位是空闲的 或者可以安排的）

EH3 不变量：

对于预定系统，状态是活动的

对于旅客，系统访问状态是已登录

对于旅客 & 航班，预定是存在的并且可以查询

对于航班,没有座位可以安排

旅客	基于web的航班预定系统

	1. If（签入）
	把旅客放到 等待队列
	后置条件
	-- 对于旅客，名字在等待队列中
	else
	当可能安排的时候提供建议
	endif
	成功退出

EH4 - 7a（没有选择可以选择的座位）

EH4 不变量：

对于预定系统，状态是活动的

对于旅客，系统访问状态是已登录

对于旅客 & 航班，预定是存在的并且可以查询

对于航班,有座位可以安排

对于预定，没有选择可以选择的座位

旅客	基于web的航班预定系统
	1.if（预定以前已安排过） **成功退出** Endif

	2.if（是航班工作日），
	建议到签入处得到座位安排
	Else
	建议以后再试
	Endif
	失败退出

图 1 基于 web 的航班预定系统中已预定座位的功能例子

6 对例子的注解

直接应用异常处理的条件，需要确定异常处理程序有可见的**来源点**。个别异常处理条件在来源点前面标记了关键词unless . 例如成功路径中步骤7，在系统分配座位之前，可以选择的必须被选择。异常处理程序必须处理没有座位被选择的情况。来源点可以是动作中的任意路径，上面的例子中，成功路径中的步骤5和7包含来源点，而异常处理程序没有。

精确用例鼓励显式地鉴别正确的输入选择。当只有简单的选项时6，在异常处理程序第二步可能会直接提供。否则，选项会在输入选择的相关术语表中说明。成功路径的第3步引用的备选就可以在这样的术语表中找到。

7 精确用例中的结构化英语

原先，开发结构化英语是为了描述个别分析实体的路径[8]，精确用例中使用的版本用来描述多个实体之间的交互。

7.1 动作路径

出现在精确用例中的路径包括如下类型：

- **成功路径** -- 每一个精确用例都肯定有一个，每一个场景在它的第一步开始。
- **调用路径** -- 引用动作顺序来说明共同的或者简单路径段。
- **单点路径** -- 引用异常处理程序，代替备选，或者单一选项。
- **异常处理宏** -- 与程序设计中的宏和面向对象分析中的类相似。宏聚合了预处理不同的异常处理程序。它利用一些共同的步骤和处理程序指定的信息产生新的异常处理程序。宏使多个处理程序的共同结构清晰化。
- **非扩展路径** - 引用多点选项，功能出口，或者应用程序出口，即，非扩展备选。
- **扩展路径** --引用功能或者应用程序选项，即，扩展备选。

7.2 动作

<动作>[，unless<异常 条件[N%]>]

动作结构 ：<动词><直接对象[修饰成分]>< 直接对象>.参见文中的例子。

does〈调用路径名〉

[using〈参数1〉,〈参数2〉,…]

[, unless〈异常条件[N%]〉]

selects/wants

a. 〈有效的输入可选项[N%]〉

b. 〈有效的输入可选项[N%]〉

…

本文中说明了动作名字，信号的输入，输出，状态变化。

产生发出输入信号的有：

请求，供给，输入，选择，和需要。

产生输出信号的有:

建议，批准，确认，显示，打印，提议，请求，供给。

产生状态变化的信号有：

接受，存放，放置，记录，返回，存储和更新。

表示符 N% 是选项被选择或者条件成立的（整数性）概率。

所有选择概率的之和一定为100%，所有异常条件的概率一定小于50%。

例如 ：

- 让旅客处于等待状态
- 同意停止请求，unless 过程没有增加 1%
- does “增加一本书”using “编写高效的地用例”，
- unless 书已经被提供了

• wants 靠近通道的座位

• selects

a. 信用或借记卡 38%

b. 足够的现金或者等价物 62%

7.3 处理中间失败

我们尝试，失败，纠正或者选择不同的（方法），然后成功。中间的失败是成功目标导向行为中正常的组成部分。有三种基本的方法可以处理这种失败：（1）报告，（2）校正，并且（3）用不同的方法尝试。

〈动作〉，unless〈异常条件[N%]〉

异常处理程序可以有效对报告和校正行为建模。Unless子句中异常条件是布尔值，判定为假时，动作或调用路径将被执行。如果异常处理的判断条件成立，则相应的处理程序被调用。异常处理程序可以尝试校正条件，可能尝试成功或失败。如果没有尝试校正或者校正失败，那么后面跟着终止的格式[15]，处理程序将会纪录/报告条件。可能做其他事情，但是终究要从包含调用unless动作语句的路径中失败退出，也就是说，路径抛出异常。如果校正成功，(那么后面跟着重新开始的格式[15],)控制返回初始到达的并且异常条件是FALSE的调用点。这样允许判断条件继续。

does 备选方案

〈方案1 路径〉

[using 〈参数1〉，〈参数2〉，…]

〈方案2 路径〉

[using 〈参数1〉，〈参数2〉，…]

[〈失败预处理路径〉

[using 〈参数1〉，…]]

FAILURE EXIT

End

不同的结构（也就是，不同的备选方案）需要模拟那些预计会发生的启发性选项的用法。这种结构包含不同方案调用路径的有序序列，这些路径直到一个方案成功或者最后的失败出口子句被执行才会被执行。每一种方案路径，除了失败预处理，必须包含一个成功出口和一个或多个失败出口。失败预处理路径不能包含成功出口。

7.4 选择

If（<选择条件[N%]>）

<动作> or does < …>

[Else

<动作> or does <… >]

[Endif]

例子:

• if ((客户提供信用卡或者贷记卡)

and (支付总额被批准)33%)

接受信用卡支付

Endif

• if (客户拒绝所有的信用卡支付)

does “现金支付”,

unless “现金或者等价物不足”

Endif

• if (登记时间并且没有可分配的座位 2%),

把旅客放置到等待状态

Else

告知座位安排时间

Endif

避免在if语句中直接嵌套if语句。嵌套的if语句应该用调用路径代替。

7.5 循环

Until or While<条件> or For each <项目名>,

执行者重复 <处理范围 >

或者 执行者重复 <调用路径名 >

步骤范围必须是直接的上面或下面的重复语句。

循环必须同时包含执行者和系统动作。

3. 提供（校正过的）预定选择定位	4. 查询预定
Until（预定找到 或者 所有的预定查找方案都已经尝试过了），**执行者重复** 3 到 4	

图2 使用循环结构的例子

7.6 共同动作中包含的内容

动作或动作路径是可以用 Include <共同动作名> 包括在用例中的共同的动作或路径。

包含的部分可以作为公用的：调用路径，意外处理程序，单点选项（例如 登录），功能出口，或者应用程序出口。

7.7 场景流

成功路径中的每个场景的用法从输入开始，到退出结束。退出可能是成功或者失败（退出）。

继续意味着调用路径的返回。如果触发继续路径的所有条件集合中的条件没有被计算值，计算没有被计算的条件，否则这行下一步。

迭代仅出现在循环路径正在被执行的步骤中，这表示流程在重复语句将会继续并且依据重复条件的值正常处理。

7.8 步骤中不明确的顺序

对于下面的动作：

执行者提供了

a. 名称

b. 地址

c. 电话号码

提供的输入中没有说明顺序。

对于这一步的动作：“申请检验并提供项目”，动作的顺序没有说明。

如果顺序是重要的，写多个步骤。

对于下面的动作：

系统通过累加请求，除非

a. 路径是完全的

b. 必要条件不满足

c. 预计有冲突

d. 路径负载不可接受

没有说明计算异常条件值的顺序，因此，如果多个条件同时成立，那么任何相应的异常处理程序都可能被调用。

8 路径中可以选择的格式

路径中的活动可以格式化成单列，两列，三列的表，在单列表中，所有执行者和系统的动作表现为单一的流。因此，为了避免混淆，每个动作描述必须明确地命名它对应的执行者或者系统。

上面的例子展示了两列的格式。描述一个或多个执行者之间的对话并且完全自动化的系统可以用这种格式。系统动作放在第二列，所有执行者的动作放在第一列。所有的执行者在第一列的表中命名。如果有多个执行者，

第一列中每个动作必须明确说明它（对应）的执行者。

三列格式用来描述三者之间的对话，也就是，执行者和技术与人混合体的交互。例如，在机场或者超市，我们发现售票机和出纳员就像是相同的技术。从执行者的角度来看人和技术是在系统内的。和两列格式一样，如果有多个执行者或多个或两个人，它们应该在标头命名并明确说明动作（的执行者）。这种格式不适于多个执行者并且完全自动化的系统。

9 精确用例建模指南

a. 用例可能需要访问特定的资源（例如，文档文件或应用程序）才能成功，执行者可能启动资源（例如，打开文件或者标记应用程序）。在这种情形下，如果资源不可访问，在用例的开始定义一个单点选项来标记的已经启动的资源，比如执行者登录。

用这种方法去除两个用例间的依赖，引用了相同应用对象的多个用例除外。例如，一个用例创建了一个名字为bob的文档并且下一个试图创建相同名字的文档，用例应该给出与第一个已经发生的用例的不同反应。

b. 有时，在“<动作>，unless ” 和 “if （<选择条件>)”之间选择时可能会发生混淆。指导规则是：

（1）当 动作总是被执行，除非一些意外条件成立时。用“<动作>，unless ”。异常条件是预计的异常情况之一。有时它居然是成立的，比如，打印机经常不可用。

（2）当 动作有时被执行，有时不执行，用 “if （<选择条件>)” 比如，用if因为某些时候添加一些路径，其他的时候减少一些路径。

c. 模拟信息流，不是设计也不是浏览细节。使用输入动作，如录入，提供，或者请求而不是在表格中填入，点击按钮，或者连接到某个指定的网页。请看[4]基本用例的讨论。其他建议可以在[11]中找到。

10 精确用例集的开发过程

在交互启发的初期实际应用精确用例非常繁琐，它们的开发可以描述为交互路径的描述（相对于说明）设计。因此，通过不太形式的风格的用例[26]获得某些需求之后才开始精确用例的开发。最初的模式作为非形式的用户经历可以用在极限编程开发或者[6]或[21]中描述的任意风格中。精确用例开发与非形式抽象伴随出现。

分析人员或者测试人员可以“离线”开发精确用例，但是他们需要涉众的支持来答复不可避免的问题并检查结果，在精确用例开发过程中暴露问题，如果设计之前没有找出问题，精确用例对质量的作用就体现出来了。

下面一系列精确用例开发过程顺序来源于[6]。

1. 详细说明系统的范围和边界 - 用文字和相关图表

2. 头脑风暴列出首要的和支持的涉众[21]，加上他们的特征（可选的）

3. 把支持同一组织功能的首要涉众分组。

4. 头脑风暴列出每一个涉众目标

5. 头脑风暴穷举与那些目标相关联的概念类[21]

6. 说明每一个概念类的属性和选项值

7. 根据需要检查并修改目标和概念类

8. 根据需要说明定义和依赖

9. 根据需要说明非功能性需求

10. 对每一个目标，重复步骤 11 到 12

11. 选择目标并确定可选择成功路径

12. 针对每条成功路径

a. 列出所有的涉众

b. 说明风险因素

c. 列出步骤

d. 和说明成功路径后置条件一样，说明成功的不变量和前置条件

e. 说明用例的不变量和前置条件

f. 确定需要增加的备选路径并确定其步骤

g. 用调用路径封装重复步骤序列或者简化路径描述

h. 针对每一个步骤，头脑风暴列出其异常条件

i. 针对每一个异常条件，指定用异常处理宏来组织相似的模式的异常处理程序

j. 根据需要检查并修改不变量，前置，后置条件，并在需要的地方增加中间条件。

13. 根据需要与涉众和问题专家一起检查并修改精确用例集。

11 测试精确用例的覆盖标准

为了描述测试覆盖，我们需要一些定义。**单目标用例** 不包含应用选项 即 扩展。**多目标用例** 除了完成目标的步骤之外，另外包含一个或多个应用选项。**转换用例** 会引起对象状态转换。比如，归还一本书，**非转换用例** 则不会 比如，显示借来的副本。针对相同对象的转换用例的有序对可能有效或者无效。例如，用户纪录创建后，只能被更新。

对于精确用例集合，测试组合应该覆盖：

1. 包含一个或两个非转换的单目标用例的每一个（有效的）序列对

2. 引用相同对象的转换用例的每一个（有效或者无效）序列对

3. 每一个多目标用例

如果这样代价太高，那么就覆盖“高或中等风险”用例的每一个序列对，或者仅覆盖“高风险”用例的每一个序列对。两个用例中任意一个，覆盖每一个多目标用例。

在单目标用例内，测试组合应该覆盖成功路径，调用路径和备选路径的每一步骤。

用例通常包含多个场景 比如，完全执行路径，因为它包含重复循环和备选路径。一个场景可能有多种结果，因为它包含多种输入选择并且输入存在多种情况(例如，无效的)或者状态（例如，不可用的）。通常有多种不同方法来解释派生条件，比如，有多种方法针对无效的（输入）或者状态的。

对于用例，测试组合应该覆盖每一个：

1）重复循环 0，1，或者适当次数

2）选择，重复，备选路径，和派生条件为原因 +1 （种）路径

3）输入选择

另外，测试组合应该覆盖：

1）输出边界

2）有效和无效的输入边界值[18]。

如果一种或者多种标准导致了测试组合过于庞大（比如，标准太极端造成太昂贵），风险信息必须减弱或者删除标准。

如果有剩余的预算并且增加成本效能已经显现，可以考虑增加“每一对输入分离值”标准[28]。

12 相关工作

其他有些人提议增加描述使用的严密性。提议包括消息队列图和代数处理过程[2]，抽象状态机语言[14]，活动图[20]，和模块化Petri网[29]。可是这些可选内容，没有象精确用例那样提供应用选择，有效测试设计的基本要素。更重要的是，其他的方法没有从相同技术的非形式格式演化发展。有效的非形式格式更容易转换成正式形式。

13 感谢

本工作很多部分来源于 Alistair Cockburn 的书[6]，我也很感谢 Ian Alexander,James Bach，Sofia，Stanislov Passova和Natan Aronshtam，他们提出了很多帮助性的建议。

14 参考文献

[1]Extreme Programming info available at www.extremeprogramming.org/rules/userstories.html

[2]Andersson，Michael and Bergstrand，Johan **“Formalizing Use Cases with Message Sequence Charts”** MS Thesis，Lund Institute of Technology May 1995 Available at www.efd.lth.se/~d87man/EXJOBB/Formal_Approach_to_UC.html

[3]Armour，Frank and Miller，Granville **Advanced Use Case Modeling** Addison Wesley 2001

[4]Constantine, Larry and Lockwood, Lucy **"Structure and Style in Use Cases for User Interface Design"** Available at www.foruse.com/Resources.htm#style

[5]Constantine, Larry and Lockwood, Lucy **Software for Use** ACM Press Addison Wesley 1999

[6]Cockburn, Alistair **Writing Effective Use Cases** Addison- Wesley 2001

[7]Cockburn, Alistair **"Use Cases, Ten Years Later"** STQE, SQE, Vol. 4, No. 2, March/April 2002

[8]DeMarco, Tom **Structured Analysis and System Specification** Prentice-Hall 1978

[9]Gelperin, David **"Precise Usage Model for Library Management System"** Available at www.LiveSpecs.com

[10]Gelperin, David **"Precise Use Case Examples"** Available at www.LiveSpecs.com

[11]Gelperin, David **"Modeling Alternative Courses in Detailed Use Cases"** Available at www.LiveSpecs.com

[12]Gelperin, David **"Specifying Consequences with Action Contracts"** Available at www.LiveSpecs.com

[13]Gelperin, David **"Testing Complex Logic"** Available at www.StickyMinds.com

[14]Grieskamp, Wolfgang, Lepper, Markus, Schullte, Wolfram, and Tillmann, Nikolai **"Testable Use Cases in the AbstractState Machine Language"** in *Proceedings of Asia-Pacific Conference on Quality Software (APAQS'01)*, December 2001.

[15]Garcia, Alessandro F., Rubira, Cecilia M. F., Romanovsky, Alexander, Xu, Jie. **"A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software"**. Journal of Systems and Software, Elsevier, Vol. 59, Issue 2, November 2001, pp. 197-222.

[16]Ivar Jacobson et al *Object-Oriented Software Engineering* Addison-Wesley 1992

[17]Jungmayr, Stefan and Stumpe, Jens **"Another Motivationfor Usage Modeling: Generation of User Documentation"** Proceedings of CONQUEST '98, Nuernberg, Germany, September 28-29, 1998

[18]Kaner, Cem, Falk, Jack, Nguyen, Hung Quoc **Testing Computer Software** John Wiley 1999

[19]Kulak, Daryl and Guiney, Eamonn **Use Cases: Requirements in Context** ACM Press Addison-Wesley 2000

[20]Kosters, Georg, Six, Hans-Werner, and Winter, Mario **"Coupling Use Cases and Class Models as a Means forValidation and Verification of Requirements Specifications"** Requirements Engineering Journal, Vol. 6, No. 1 2001 pp 3-17

[21]Larman, Craig **Applying UML and Patterns** Prentice-Hall PTR 2002

[22]Rumbaugh, James **"Getting Started : Using Use Cases to Capture Requirements"** Journal of OO Programming, SIGS Publications, Vol. 7, No. 5, Sept 1994 pp 8-10, 12, & 23

[23]Runeson, Per and Regnell, Bjom **"Derivation of an integrated opertational profile and use case model"** 9th Symposium on SRE IEEE press Nov. '98 pp. 70-79

[24]Schneider, Geri and Winters, Jason P. **Applying Use Cases** Addison Wesley 1998

[25]Thelin, Thomas, Runeson, Per, and Regnell, Bjorn **"Usagebased reading - an experiment to guide reviewers with use cases"** Information and Software Tech. 43 (2001)pp. 925-938

[26]Wiegers, Karl E. **"Listening to the Customer's Voice"** Software Development, March 1997

[27]Warmer, Jos and Kleppe, Anneke. **The Object Constraint Language** Addison Wesley 1999

[28]Williams, Clay and Paradkar, Amit **"Efficient Regression Testing of Multi-Panel Systems"** IEEE International Symposium on Software Reliability Engineering, Nov. 1999

[29]Woo, Jin Lee, Sung, Doek Cha, and Yong, Rae Kwon **" Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering"** IEEE Transactions on Software Engineering, Vol. 24, No. 12, December 1998

软件与系统思想家温伯格精粹译丛

现代需求技术的基石

探索需求

设计前的质量

Donald C. Gause
Gerald M. Weinberg / 著
章柏幸 王媛媛 谢攀 / 译

Exploring Requirements: Quality Before Design

UMLChina 训练辅助教材

清华大学出版社

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

在嵌入式系统开发中应用敏捷方法

Doug Dahlby 著，rowvy 译

简介

纵观过去十年，软件开发中的迭代方法得到了广泛认可，很大程度上取代了较老的方法，如瀑布法或 V 模型的软件开发方法。特别是，近几年出现的敏捷方法取得普遍的拥护。敏捷方法确实提升了软件开发最佳实践，但在不同的软件开发体制下，敏捷方法提供不同的助益。嵌入式系统是软件开发体制的一种，在其中使用敏捷方法颇具挑战性，但可能对其使用敏捷方法的优点不像其他软件开发体制那么显著。本文讨论嵌入式系统软件开发的不同方面，如何影响了敏捷方法在嵌入式系统中的应用。虽然在嵌入式系统软件开发中使用敏捷方法，最终证明是改进，但为得到可能潜在的收益，也要求细致考虑。

背景

软件开发问题摘要

软件生命周期

创建、部署、支持软件的过程有几个相当清晰的阶段：系统说明、系统架构/设计、组件说明、设计、编码、单元测试、集成和维护。不同软件开发方法对各阶段的划分不一样。一些方法强调或忽略某些阶段。一些方法从头到尾执行这些阶段，同时考虑重叠采用这些阶段或循环迭代这些阶段。不论怎样，至少各阶段都会出现在任何软件开发方法中。

软件编码折衷

有很多评估软件“质量”的度量。许多和代码质量有关，也有些不相关。特别在性能方面，如周期和内存使用，一般和可读性、易测性、模块性、可维护性等代码清晰度无关。为了得到好的质量，机器码必须很好地和特定的芯片结构匹配。为了得到高清晰度，源代码必须容易被人类阅读者理解。理论上，编译器能翻译高清晰度的源代码为高性能的机器码，而实际要翻译得完全合理就太复杂了。理想地，编译器可根据自然语言或简单图形说明，高效产生和谐的机器码，但在可预见的未来，编译器都没有足够的人工智能达到这种理想。处理器能力的摩

尔法指数化增长则告诉我们，权衡性能/清晰度，应更倾向清晰度，因为机器持续提高他们阅读源代码的能力。可是，性能/清晰度的折衷观点依赖软件体制。

软件开发方法简史

混沌

软件开发的“混沌”模型一头扎进代码中，忽略需求、设计和增量测试。这种模型在计算机编程的早期使用，但仅用于非常小且简单的系统。所有随后的开发模型都通过分解来使大型系统开发变得可能，来探索混沌模型的改良。系统开发实践依赖于分解系统为独立大块，再逐步累加运行块来建造系统应用。因此，系统开发实践关注全系统的高层和系统小组件的精细。这和混沌法无法关注全系统的精细层形成对比。

功能和瀑布法

功能化编程（也称结构化编程）关注软件的预定行为。软件的总体行为被分解为内聚功能的程序。软件开发的瀑布模型规定一个单一的长系列过程覆盖生命周期所有的阶段，立即构建系统的所有部分并最后组装。图例说明见下：

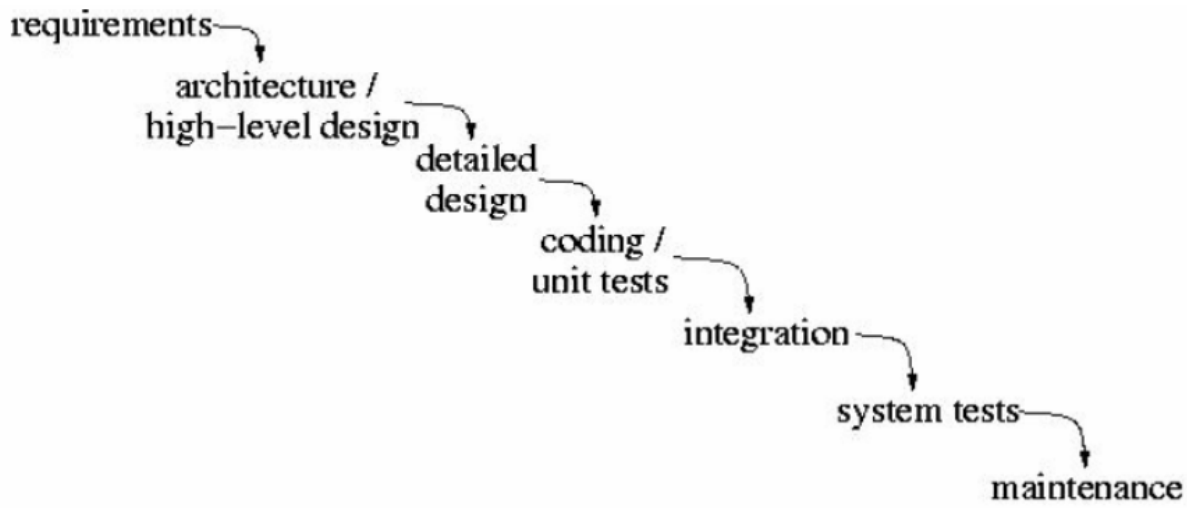


图 1 基本的瀑布模型

一些瀑布模型版本允许生命周期中相邻阶段的迭代，图例说明见下：

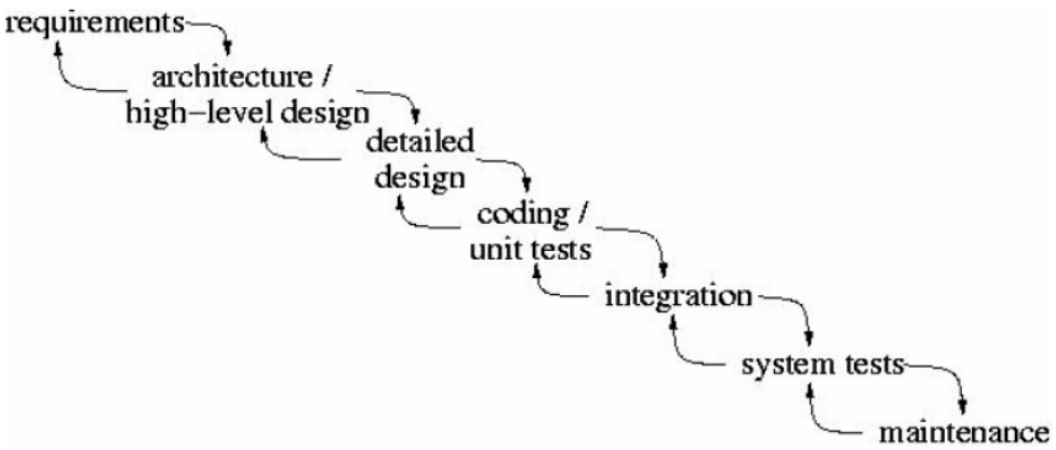


图 2 阶段重叠的瀑布模型

实际上，在大系统中肯定有在较晚阶段浮现的不能快速定位的问题，并反而要求重新构建架构或者重审需求。因此，无论是否有意，瀑布模型在实际中经常结束于大规模的迭代，那些迭代包括越来越多的生命周期阶段。预期结果的复杂性能轻易从下图看出。

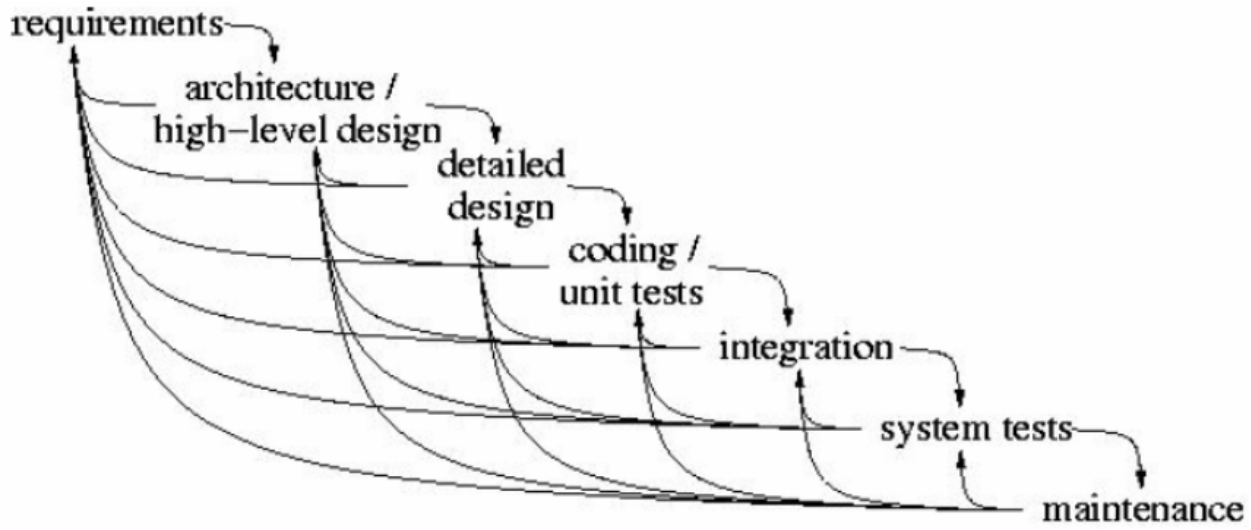


图 3 实际的瀑布模型

发现的这些范型也部分地反映在混沌模型的类似问题上，通常广泛在七、八十年代运用。

面向对象和迭代

面向对象编程关注于软件组件的角色。软件开发的迭代在生命周期阶段重复系列过程，再构造、组合系统小块。

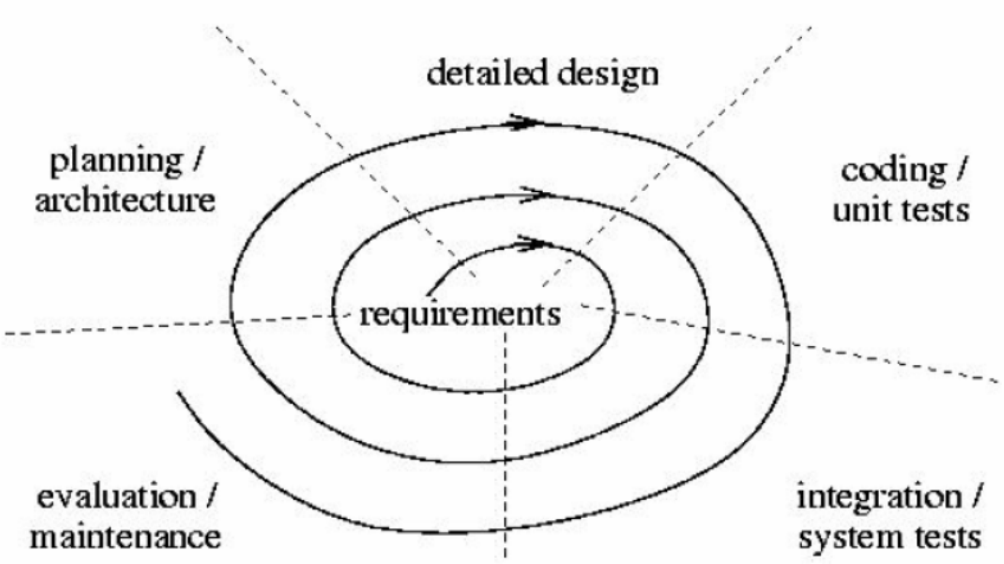


图 4 迭代模型

介绍的这些范型，部分是为了对应瀑布模型处理面临问题时的对策，在九十年代广泛使用。

面向方面和敏捷

面向方面编程结合了软件的功能视图（例如用例）和对象/组件视图（例如类图）。在软件生命周期的不同阶段，可使用功能性或基于对象的任一视图，按当时情况使用更适当的视图。一般而言，功能性视图最适合用于需求收集，因为用户关心系统做什么胜于关心怎么组织它。系统架构期间功能性和组件视图一样重要。详细设计、编码和单元测试更建议集中于组件视图，这样能用可管理的块构造软件。集成时，组件和功能视图都重要，最后的验收测试关注功能。维护期间，组件和功能视图同样重要，在考虑新组件的功能和原有组件的调整时，影响分析和回归测试适用于组件。面向方面编程不仅向基于组件的系统往回添加了一个结构化的观点，而且提供了一个概要的功能图。功能“方面”经常横切结构程序和组件。面向方面编程依赖“联结点”框架、在联结点上执行的不同方面的说明和实际使用方面的参考说明。通过结合联结点的建议行为，一个“方面编结器”能增强基础/预置功能。

敏捷软件开发过程是迭代过程的精化，它接受甚至鼓励不断修订需求。快速迭代使系统能响应变化，客户与开发人员之间，开发人员之间的紧密通讯，保证增长和修正系统能正确地说明变化。测试驱动开发保证只构造最小的必须软件，保证正确构造，保证始终在进行的改变的正确性。

严格说来，敏捷过程是迭代过程的一个子类型。可是，按传统理解，迭代过程只在生命周期的设计、编码、单元测试和集成部分进行，仍执行前期的需求和系统架构。相反，敏捷方法的每个迭代会重访软件开发生命周期的每个阶段，虽然有一些敏捷方法的子类型（如 RUP），却更强调在开发周期迭代中的特定“工作流”的比例。即，尽管早期的迭代结合了一些代码和集成，但它们集中体现规约和架构。尽管最终的迭代包括一些架构和设计，但它们集中体现影响分析、编码和回归测试。

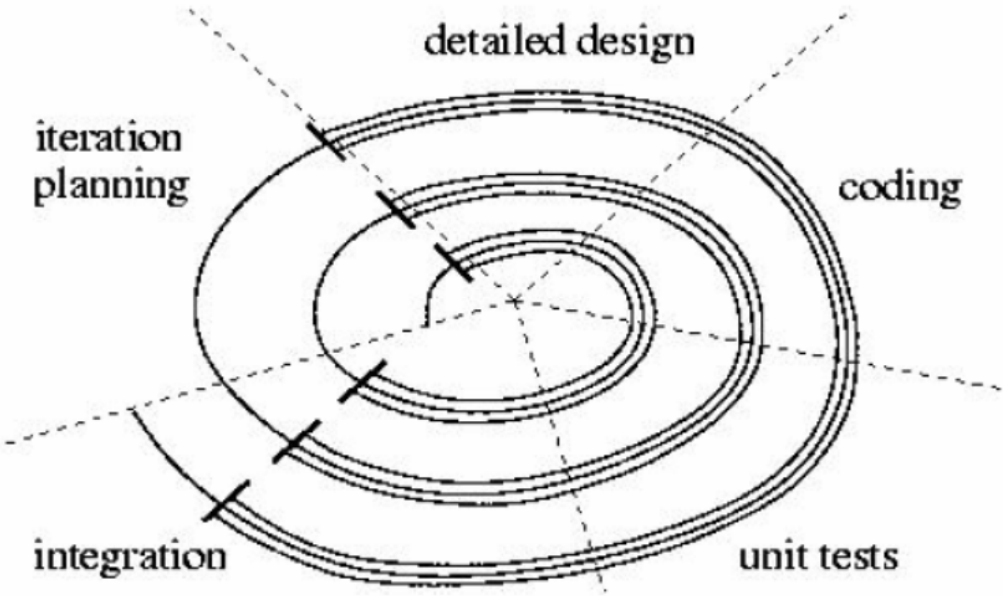


图 5 敏捷模型

传统迭代过程和敏捷过程的根本区别是在迭代计划阶段，敏捷过程首先根据已完成迭代得出的度量，来评估下个迭代阶段将要进行的总体工作，接着选择纳入下阶段的新特性和组件。因此，敏捷方法不仅像传统迭代方法一样增量创建系统，而且递增的决定系统应该建成什么样子。

同样，正如按并行流“泳道”所展示的那样，在设计、编码和测试阶段，迭代的工作要素假定是相互独立的，在这些阶段也按独立的步调执行。

以上范型在 90 年代晚期和 21 世纪早期引进，正被逐步广泛的实践。

4 什么是敏捷方法？

敏捷方法的原则在敏捷宣言网站http://www.agilemanifesto.org说明。简言之，敏捷方法

- 通过持续交流（客户/管理者/开发人员，管理者/开发人员，开发人员/开发人员）来管理变更。
- 保持客户需求（“故事”）和软件的紧密可溯性。
- 快速轮回地发布功能性增长软件版本。
- 不断选择性改进（“重构”）旧代码

在下面的章节中，将对这些关键原则做更详尽的描述。

敏捷方法的特征

敏捷方法的原则代表性地导致如下特殊实践：

1.使用规律的快速周期，以创建可执行的交付版本。

2.更多关注编码而不是计划或文档。

3.持续重构以改善代码。

4.在工程开发团队中不断广泛沟通。

5.和客户不断广泛沟通。

6.持续量度项目过程，推测最终形态，判断长期项目目标（项目完成日期和软件特征），并提出短期目标（下个迭代阶段的工作要素）

7.使用测试驱动开发来验证代码最初的正确性，强调回归测试保证代码保持正确。

敏捷方法实例

著名的敏捷方法包括极限编程、rational 统一过程（RUP 实际是一个“元过程”，它能配置来构造多种不同软件开发过程。尽管 RUP 经常以敏捷软件开发过程的配置出现。）、Scrum 和 Crystal。

• 极限编程（XP）－参见http://www.extremeprogramming.org。极限编程法集中于交互性、迭代性极强的小团队的工作代码交付。XP有四个被核心准则提倡的核心价值观。它们是沟通、简明、反馈和勇气（授权和确信）。核心准则是：

o 团队整体（客户、项目管理人员、工程师都齐心协力）

o 隐喻（共享概要系统图以提供公共环境和词汇）

o 计划的策略（用户列出“故事”（功能部件），选择在每个迭代阶段完成的故事，并为每个故事指定可接受的标准）

o 增量发布（在每个迭代周期交付可执行工具，根据客户提供的评价获得反馈）

o 简单设计（只创建立刻需要的东西）

o 成员结对编程

o 测试驱动开发（在编写故事代码前编写测试，最初测试验证失败，然后在实现故事后通过测试，单元测试和故事验收测试的自动回归）

o 重构（持续改善旧代码/架构）

o 持续集成（通过分段集成控制软件熵的增长）

o 代码的集体所有（任何人在任意时间能写/重写任何代码）

o 编码规范（公共规范的协议—承认重构是主要的方式而非只做形式上规范）

o 适当的节奏（每周工作 40 小时而非死亡之旅）

• RUP － 参见http://www-106.ibm.com/developerworks/rational/library/253.html。RUP将一个项目分解成多个开发周期，将每个开发周期分解为多个阶段：先启阶段、精化阶段、构建阶段、和移交（产品化）阶段。每个阶段由依次的开发迭代组成，每个迭代产生可用（理论上可执行）的工件。RUP在软件开发中确定了一系列“工作流”，或主题：业务建模、需求、分析、设计、实现、测试、部署、配置和变更管理、项目管理、环境管理。所有的工作流在每个阶段都会涉及，但每个工作流的重要性和成效在不同阶段有所不同。

• Scrum — 参见http://www.controlChaos.com/Scrumo.htm。Scrum管理过程将项目分成短期（30 天）迭代，或者“短跑”（sprints），每个短跑中在开发团队和团队管理间有个短（15 分钟）会（scrum）跟踪进展，记录前进途中现在和即将碰到的干扰，判断决定到下次会议为止应关注的工作。在短跑期间开发目标保持不变。每个短跑的目标都在短跑前期商定。从而反复调整项目目标。

• Crystal—参见http://alistair.cockburn.us/crystal/crystal.html。Crystal方法不仅考虑最佳理论，而且考虑切实可行，因此希望获得好的折衷并最终满足大批需求而取得成果。尤其，“标志和小道具”（文档、详述、会议等等）用于促进发展（并有可重用的“剩余价值”以促进未来的发展）。过程的形式由项目的大小和种类成比例决定。

关于敏捷方法的行业观点

极限编程

极限编程是最知名、最有趣、最极端的敏捷方法。它在行业中激起了很多议论和热情，迄今为止吸引了相当多的开发人员参与其事。很多信徒视它为被瀑布模型、严格顺应 ISO9000/9001 或坚持 CMM 污染了的开发实践的救世主。还有人在认为它是合理的方法和它是过度严格的开发实践的观点中摇摆。

保留看法

直到最近，人们对 XP 可用性的争论仍受轻蔑，但现在保留或不同意的显著声音指出了 XP 的潜在问题。尽管，行业整体仍然持续显示出对 XP 的更多兴趣，这也是敏捷方法益处得到认同的必经之路。

什么是嵌入式系统？

嵌入式系统可粗略的定义为“系统主要不是计算机，包含一个处理器”。但与其考虑定义，不如考虑多数嵌入式系统的共同方面更有用，至少一定程度上是。

• 嵌入式系统常常对价格、规模敏感。

许多嵌入式系统例如个人数字助理或蜂窝式便携无线电话是大量、低价位、低利润的。这要求尽可能使用最廉价的软件，即意味着简单处理器和小内存（RAM 和 NVRAM/flash）。这导致嵌入式系统软件牺牲可维护性，如可移植性、清晰性或模块化，来优化性能，如小的启动图像痕迹、小的 RAM 组件和小循环需求。前面的增量式软件开发成本和定期维护成本，可被大量销售摊销，也随元件不断减价带来的硬件

费用节省而降低。

许多其他嵌入式系统，虽然并不是价格敏感，希望尽可能使用廉价元件，却在外形因素或重量上受到实际约束。再次以可维护性为代价以支持性能最优化。

除牺牲可移植性、清晰性或模块化，嵌入式系统也可通过使用低级语言进行优化，例如使用汇编而非 C 语言，使用 C 语言而非从 UML 模型自动转换的代码。可是，这种手工调整仅代表性地用于大约“90/10”的小部分软件以突破系统主要性能瓶颈。

• 嵌入系统常是电源受限的

许多嵌入系统靠电池提供能源运行，不论是持续运行或紧急情况时运行。因此，很多嵌入系统推崇以复杂性和可维护性换取电源消耗性能。

• 嵌入系统通常是实时操作系统

大多嵌入式系统天生就是为了和数据流向和穿越系统进行实时互动建造的。实时操作约束（特别是周期使用）再次高于可维护性。系统通常都有两种约束，硬件实时约束要求定时处理某事件，软件实时约束设置了事件平均响应时间和容许的最大掉线值。实时操作系统采用先占先调度，来确保遇到的实时操作最终期限，但需要谨慎考虑“处理”到“执行环境（线程）”的分解、设置执行环境的相对优先权、在环境中管理控制/数据流。

•嵌入系统常使用定制硬件。

嵌入式系统常由现有外设构成的现场处理器组成。即使可能是标准组件、混合定制和符合对硬件和软件的高度内聚要求——一个嵌入系统软件的有效部分是操作系统和设备驱动软件。虽然低级软件可通过购买、许可或免费使用，嵌入系统操作系统的大部分是自身定制开发的，不论是正好匹配最近的硬件，或是在定制配制的现有软件上增加。一个嵌入系统的功能经常分布在多级处理器和（或）一层主从处理器上。需要仔细考虑处理器上处理任务的分布和扩展、方法和处理器间的通讯时间。

此外，许多嵌入系统使用特定的现场可编程门阵列或专用集成电路，因此要求低级软件和定制硬件交互。

• 嵌入系统的不可见性

嵌入系统本身常只包含几个代表性的“用户”（真实的用户或父系统的其他组件）交互界面。因此，许多系统被开发来满足在架构和高层设计期间产生的软件功能说明，而不是根据用户需求开发。

• 嵌入系统常是整体式功能的。

许多嵌入系统是基于单一基本的目的开发的。它们能被分解到各组件中去，而这些组件很可能做低的交叉内聚和交叉耦合。即：每个组件都提供一个独特的服务目的，组件间的互操作限于少数明确定义的点。尽管如此，只有系统的大多或全部组件都在运行，整体系统才提供操作。一个要求所有组件都工作才能提供有用功能的系统称为“整体式系统”。组件性能到系统性能的非线性跳跃和其他类型的软件相反，其他软件在完成 50%时就能提供 50%或更多的功能。例如，建造一个用于旅行或到其它星球并发回信息的航天探测器。虽然存在很多航天探测器组件的底层职责，例如定位、着陆、展开探测器、展开太阳电池板、和通讯，且每个底层职责都是整体功能不可缺少的组成部分。但如果任一重要组件失灵，航天探测器就无法使用，即使此时其他的所有组件都运作正常。另一个例子是蜂窝式无线电话，它所有的子部件，如用户界面、蜂窝基站选择、声码器和通讯协议，都是用于在用户和特定远程节点间交流双向声频信息，这个最终目的的重要特性。这些和软件体制相反，例如网络服务或者桌面工具，它们的底层职责更像在独立为集成系统功能贡献力量，而不是作为整体式系统的一个不可缺少的部分。

软件嵌入系统的软件组件被组合成一个整体式功能，组件自己却常是独特的。嵌入系统常组合执行单一处理的软件组件、底层设备驱动器输入/输入、通讯协议、导航和控制、以及用户界面。这些特定组件均要求开发人员有特定的专业技能。

• 嵌入系统常受限于开发工具

虽然一些软件开发方式有一大堆软件开发工具，嵌入系统软件开发却只有有限的工具，并经常只使用基本的编译器工具。这部分是因为嵌入系统常使用定制硬件，该硬件可能没有工具支持，并且因为嵌入系统常有实时和性能约束，使得难以固定在调试程序或传送控制下的执行环境，难以固定嵌入目标和基于主机的工具之间的数据，或捕获大范围跟踪运行的日志。

- 嵌入系统常严格要求稳健

嵌入系统常在苛刻的环境下使用，用于紧急情况或医疗目的。因此，要求可靠、有纠错处理能力、失灵的平均时间要求比其它软件类型的要求更加严格。这转化成严格的开发过程和测试要求。反过来，这增加了发布软件的所需费用。一些嵌入系统类型受规范需求支配，这意味着通过管理软件开发过程减少错误率，或至少规定嵌入系统产品必须完成什么文档。

- 嵌入系统经常长生命期使用

嵌入系统往往会使用多年。维护嵌入系统的工作期常远远长于源软件开发人员的更新速度。这凸现说明嵌入系统软件好文档的重要性，特别因为性能权衡而妥协源代码的自文档质量。

敏捷方法对于嵌入系统的适用性

本部分考虑上面列出的敏捷方法特性多好的适用于嵌入系统的软件开发。

1.敏捷方面：使用规律的快速周期创建可执行交付产品

推论：

为了在短周期（大约 1—2 周）内发布新特性，特性设置必须是极细粒度的。可是，因为嵌入系统是个倾向整体式功能的系统，所以将功能分解为小的并行块相当难。因此，需要在最开始做相当大量的工作，制定逐条的特性分解计划以适应快速周期。可延长周期使其能实现分解后正常特性的程序规模，而不要在分解上过多纠缠。

特别是，当模拟和/或操作系统结构开发与发生硬件升级时，可在项目开始时计划较长一些的周期。

此外，在周期中安装新特性的冲刺，会出现特性在初始实现时没有有效行为的现象，这是以期在之后的周期中重构或更有效实现。基本上，这是个有效途径，但也有潜在的问题。首先，可执行的发布可能根本无法工作，因为周期过多。第二，当产品几乎与周期不相称，需要进行性能优化时，如果再充分多给些时间去完成，当它在理解上不再陌生时，一些代码的质量能轻易提高。最后，产品可能结束于碰到一般的效率缺乏，但也胜于少数显著的缺乏周期的功能/特性。例如，假如调试代码可始终自由运行，那它可能结束在忙于产品的周期或内存使用。如果很好计划了代码调试，可轻易动态禁用它甚至编译出代码。可是，如果它是特别开发出来的，需要很多的重构工作来禁用它。这种“death by a thousand paper cuts”场景是

极难重调的。相反，预先的重要设计，能从开始就为需要实现的特性设置一些规约，编码实践内容能随处观测以辅助周期和/或内存效率。敏捷实践主张预先设计应最小化。这是个应付多变需求和意外问题的好习惯。可是，嵌入系统的顶层需求通常比一般软件领域更能有效的切割、净化、冻结。然而嵌入系统软件开发方法也需要大量变更，响应变更的方针应基于可能要发生的变更程度。

积极的敏捷方法提倡每个迭代为用户创建一个作为输出的发布版本。即使一个嵌入系统能发布的是小碎块可用功能，在每个迭代周期制作嵌入系统的用户发布版经常不可行，其大小、复杂性和必须的性能与可靠性要足够大以进行除系列单元测试外的有效系统测试。就系统测试可自动化方面来说，它能使开发过程大大收益。但持续的工作负载测试要求额外的测试硬件系统，可能昂贵。此外，大型嵌入系统要求许多迭代。向用户发布每个迭代会导致运作和维护的噩梦。因此，对嵌入系统，一个较好的策略是只对精选迭代过程申请费用，费用只用于有商业版本资格的大范围系统测试和运作维护。

这个方面在嵌入系统的使用：一般

2.敏捷方面：关注代码胜于关注计划或文档。

推论：

关注代码胜于关注设计和文档有许多好处。特别是，它允许采用一些早期的项目周期和内存使用度量。这些早期的量度，尽管只有最终版本的很少预示，也大大好于在封装考虑后量度。有了项目周期和内存使用的预计，可较早考虑方向调整，或至少考虑更准确的项目计划。如果及早发现项目周期的预计远远超过现有原型处理器的性能，也许会开发第二个原型，有更强大的处理器、更快的总线、或更好的内存系统。（虽然，像下面部分提供的“最佳实践”一样，初始的原型已具有最大可能的性能。）即使不可能引入原型的第二种风格，但至少能预先识别需要花费在优化上的时间。

尽早反馈的长处毫无疑问被嵌入系统特别重要的设计和文档减少了。嵌入系统的预先设计特别重要，因为有设计在后续流程中可变更多少的必有限制。在其它软件开发体制中，不良设计可能会导致产品的笨拙，但仍有一定功能性。相反，嵌入系统通常有实时的截止时间——如果它太笨拙，就不能再使用它。同样，因为嵌入系统对性能、可移植性的要求可折衷。这使得处理器、操作系统、编译器和架构的初始选择特别重要，并且要求可移植性/性能的恰当折衷设计，能说明产品生命期中系统特性可能出现的变化。处理器/任务的分工和任务的优先不能轻易改变，所以预先设计工作在处理器和任务的映射很重要。因此，在嵌入系统中设计工作特别需要注意，从系统级到代码组件级的所有设计。

嵌入系统也常特别要求具有一定文档。因为要求有最优化的代码，嵌入系统不能太多依赖于象其他体制内软件的“自说明代码”。此外，嵌入系统的被长期使用特性导致，很可能没有会一直支持到软件维护阶段的最初开发人员。最后，一个嵌入系统受文档的管理需求左右。

虽然敏捷方法考虑到一些编制软件外的其它工件，可是敏捷观点认为那些其它工件经常根本不必要（软件可自说明），或即使必要，也是直到软件的有关部分已经完成时再创建，它们源自于软件。编制软件是绝对目标的这个观点过于简单。虽然编制软件确实是软件开发项目最重要的工件，可仍有其他工件，它们，虽然次要，但经常不可忽略。

这些次要目标包括：

a. 软件正确工作的客观证据

这要求附加工件如测试/需求的可追溯性、测试记录、测试覆盖的记录、源代码质量度量。理想情况下，大部分这些文档能自动由敏捷过程中组合的工具生成。尽管有必要确定强调“软件编制”不会阻碍运用这些文档。

b.保持将来软件的正确编制

这要求附加的工件如，架构文档以帮助代码维护者理解系统的高层结构，设计文档以帮助代码维护者理解权衡、选择、及构造系统时使用的设计选择、源代码质量度量。这些是客户不要求的工件，因为不是所有的客户都有足够的经验知道和为未来的需求做计划。可不管客户是否明确地要求这类文档，开发人员作为专业要求，为需要用文档说明内在复杂性的系统书写文档。

c. 用户文档

虽然敏捷方法善于构建有良好适用性的软件，在软件上扩展软件的适用性。用户文档的额外工件可能包括用户安装使用手册、安装向导、变更日志/特性摘要、错误列表、和顾客服务。再次，这些会由用户明确要求，如果用户有一定软件使用经验。但很多时候，用户都不知道还需要哪些东西，直到他们碰上。在很大程度上，这些工件会在敏捷过程中生成、根据已编制成的软件书写，但这需要小心管理，并再一次违反敏捷论的“编制软件就是全部”。

*这个方面在嵌入系统的使用：*一般

设计模式精解

Design Patterns Explained

看不懂《设计模式》？
不用惭愧，别人也一样苦恼。
先把它供起来吧，看这本！

UMLChina 训练
辅助教材

Alan Shalloway & James R. Trott 著
熊节 译

清华大学出版社

软件工程技术丛书

设计系列

企业应用架构模式

Patterns of Enterprise Application Architecture

(美) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

UMLChina 训练辅助教材

UP 实作的一些常见问题（下）

think 著

改进的误区

1. 一次改进过多环节

如何引入 UML 是一个问题，改进的时候最容易陷入的误区就是一次性引入过多环节和元素。常见的情况是：技术主管在外面参加了培训或者请顾问到公司作内训，然后就开始“全盘实施”UP。

一个简单的 UML 符号后面往往隐含着大量的背景知识。用例技术的引进意味着开发团队要学会从涉众利益的角度来看系统的价值，而不是从内部构造的角度。对象技术的引进更是代表着构造软件的思考范型的改变。在团队成员没有具备这些背景知识的情况下，这些冒然引进的改进环节和元素必然不能为团队带来真正的改进价值，势必会引起团队对 UML、用例方法、对象方法的怀疑和反感。

一位软件技术主管到北京参加了笔者主讲的一次为期两天的训练班后，回去就开始依例实施，过了半年多，又请笔者到所在的企业里去，那是西南的一家钢铁集团。笔者看到当初笔者所教授的那些东西被应用在了该企业的 L3 系统（上接 L4 即 ERP 系统，下接 L2 即生产控制系统）的开发中。按照笔者所给的文档和模型模板，产生出许多的用例图、用例文档、类图、顺序图….，而这些文档模型本身是非常有问题的。笔者最后的建议是只改进已经写好的用例文档，放弃后续部分工件，用例之后，仍然沿用以前的开发方法（模块分解 + IPO 图）。从此以后，笔者在给团队作训练时，必定会强调分步改进和持续改进的重要。

很多团队是“一穷二白”的，也许他们也写有需求规格说明书，但上面说的不是需求。也许有设计文档或图形，但并不能对编码形成指导。对于从零开始的团队，第一个引进的概念是“愿景”，团队需要对“为什么要开发这个系统”达成共识。另一个需要建立的概念是“涉众”，什么人关心我在写的这些代码？这个时候即使还是没有正式的需求工程，没有改进分析设计，在开发过程中，把这两个概念记在心中，愿景和涉众的概念也能起到潜移默化的作用，我为什么要画这个图，为什么我要编写这个窗体？

图 1 愿景和涉众

接下来，先从需求过程开始改进。需求相当于找到目标，设计是用枪去打目标。如果需求不解决，目标找错了，用多好的枪去打都不会得到好结果。先引入用例技术改进需求，后面的分析设计还可以用老方法。等需求有了显著改进，再引入对象技术来改进分析设计。即使是对象技术，也可以只做分析不做设计，然后把职责压扁成“事务脚本＋数据入口”来实现……。

分步持续改进的另一方面就是挑选一个小项目来改进。关键是要真正体现出好处。最难的往往是开头，一旦尝到甜头，后面的路就好走多了。如果没有小项目，也可以挑选项目的一小部分来改进。

总之，匍匐前进，能用一点用一点，用一点是一点。

2. 不恰当的多余工件

UML 的元素和工作流之间并没有一一对应的关系，在每个工作流中，经常使用的工件可以不止一种。笔者把各工作流常用的元素列表如下：

工作流	常用元素	描述目的
业务建模	用例图	业务用例－－业务对外提供的价值
	活动图	业务用例实现的业务流程
	类图	现实业务中的人、事物、关系
	顺序图（协作图）	业务对象如何协作实现业务用例
需求	用例图	系统用例－－系统对外提供的价值
分析&设计	类图	系统内部的各个成分
	对象图	辅助说明类图
	顺序图（协作图）	类之间如何协作完成用例
	状态图	跟踪一个类对象在所有用例中的变化
	活动图	类操作的算法
	构件图	封装同源的类为构件
	部署图	构件在物理上的部署

图 2 工作流的常用元素

开发人员怀着学习的心理，常常会希望在项目中尽可能多地使用上各种工件，却忘了这些工件都是为了最终的软件服务的。

笔者在给一家企业做训练时，使用活动图、用例、类图、顺序图几种元素，和开发人员们从业务建模到设计剖析了他们当前在做的项目。训练结束后开发人员都填了反馈表，一片叫好声中，有一位开发人员写：讲的内容太少了，不值！上次××××公司来培训 UML 时，9 种图都细细讲了，老师你才讲了几种？

UML1.x 一共有 9 种元素，这些元素并不是每一个项目都需要使用的。最常用的是用例、类图、顺序图三种，其他元素则根据需要添加。如果有业务建模工作流，可能需要活动图来详细说明业务用例；如果对象本身的行为不断改变，很可能需要添加状态图来描述。

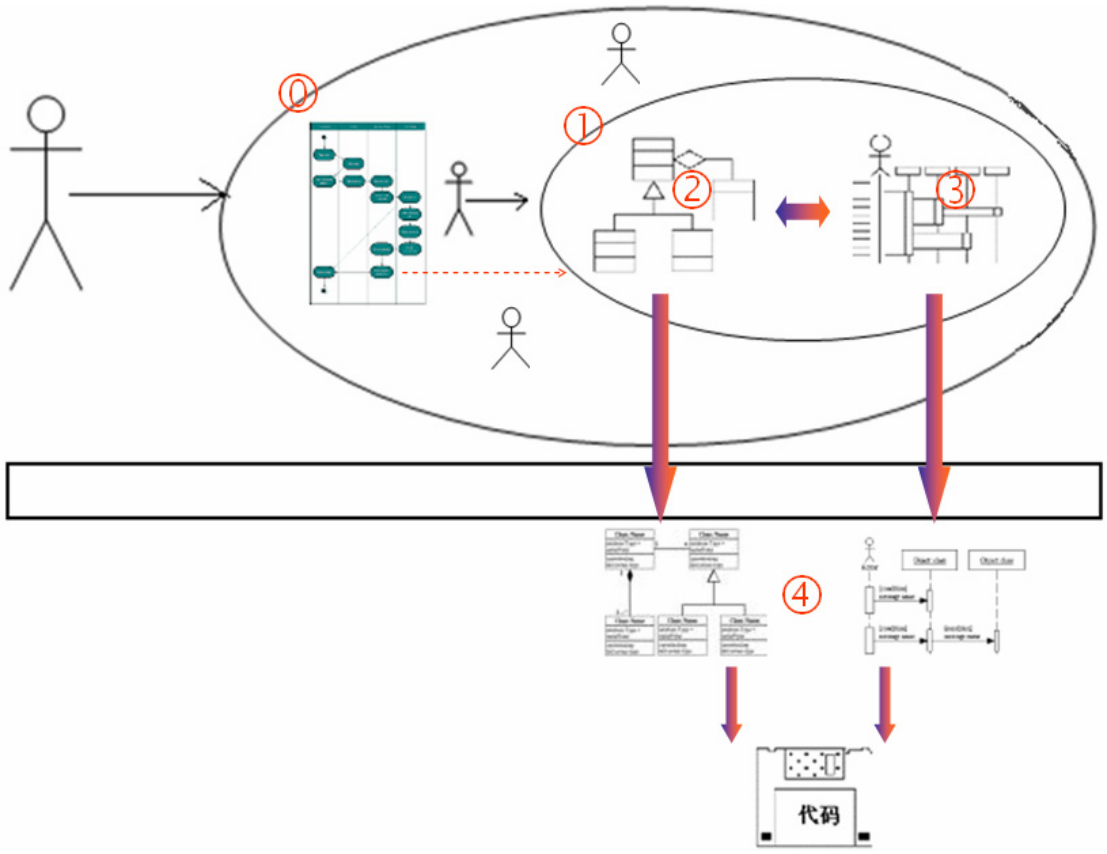


图 3 最常用的元素就这几种

不要担心没有使用或者使用了什么元素。需求工作流，难点在于和人打交道，重点不在于符号上，这个时候可能什么符号都用不上，那也没关系，如果需要画活动图、类图、用例图和涉众探讨，那就画吧。如果对一个类画不出它的状态图，也不用苦恼，因为很可能它在系统上下文中根本没有状态变化。构件图和部署图在很多工具里只是起到演示作用，对软件的最终形态并没有实质改变。

够用就好。如果添加一个工件并不能给开发起到增值作用，干脆就不要添加它。但由于人脑的局限，往往需要不同的比例尺和视图来剖析软件时，这时也要毫不犹豫地添加工件，甚至是很灵活地添加。

如果掌握了各个元素的内涵和背后的原理，用法还会更灵活。例如下图：

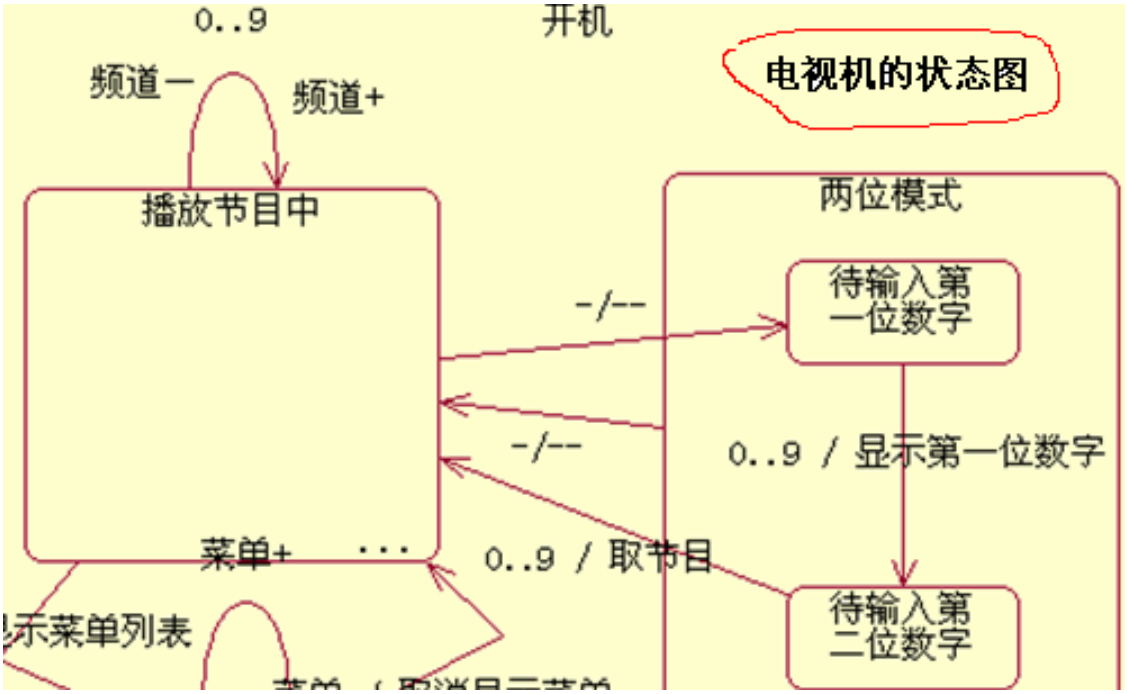


图 4 状态图也能用来表达需求

显然图 4 只表述了电视机系统的“外观”，并没有暗示电视机内部的“构造”，所以上图确实表达的是需求。

甚至也可以不用 UML 的元素。如使用 CRC 卡取代顺序图（协作图）作为职责分配的工具，用户故事取代用例作为一种需求的表现形式等。模型不一定等于图形，它可以表现成任何一种形式，用例文档一定要是“文本”文档吗？对象交互一定是交互“图”吗？关键是其中的内容！

笔者对“用例图方便和客户交流”的说法一向不甚乐观，但前几天有一个学员告诉我，他的客户喜欢看用例图，说“那个小人就是指我们，圈圈就是指要做的事，对吧？”（真是难得的客户）。然后学员问，“可不可以这样，干脆我不写用例文档了，把需求画成顺序图给客户看”——当然可以，Just do it!（如果不用考虑需求管理不方便的因素）

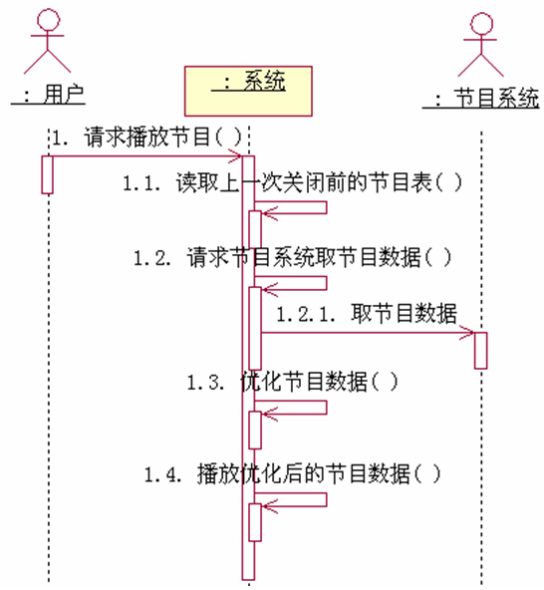


图 5 把需求画成顺序图

3. 瘫痪

笔者最近得到一份开发团队寄来的需求规格说明书，最后部分列出了几条该项目的风险，其中一条是“用户的需求在项目开始设计后发生变化的风险——需求一旦确定就不能有大的改动，否则对以后的开发和设计会造成很大影响，容易造成延期和成本增加。”

提升需求能力、设计能力固然是好，但有一个事实不得不承认，不管前面做得再怎么仔细，在真正可运行的软件被涉众验证之前，它们的正确性是不能得到最终认同的。认为需求和设计能做到尽善尽美再拿去实现的瀑布式想法既不现实，也不值得。在过度的评审再评审中，项目陷入了一种“瘫痪”状态。未知和变化总是一种“不可避免的坏事”，增量和迭代式开发就是为了应付这种未知——能做到“拥抱变化”更好，但并非如此容易。

UP 把开发的阶段从原来人们熟悉的“需求、分析、设计、实现”变成了“初始、细化、构造、移交”，各阶段又包含若干迭代，每一个迭代包含完整的工作流程。应用 UP 本身就意味着要接纳迭代开发的思想，不只是开发团队，而且也包括涉众。

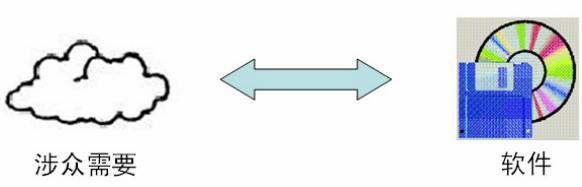


图 6 软件开发的两端

开发系统是为了满足涉众的某些需要和价值，只有运行起来而且发挥出价值的系统对他来说才是有意义的，其他的中间环节只是为了最后的实际工作的软件服务。极限编程（XP）和测试驱动开发（TDD）成为流行的词汇，决非偶然。不断地添加用户故事，编写测试用例，编码满足测试用例的做法加速了两端的反馈。

但这只是问题的一方面。另一方面，在很多时候，正式的需求分析设计过程往往也是一种“不可避免的坏事”。在和涉众交流相当方便，而且问题的规模能够通过口头或其他非正式手段所把握时，产生过多的中间环节可能是不必要的。如果涉众的种类很多而且利益各异，如果涉众不方便做“在场客户”，如果系统的规模相当庞大，如果参与的人数相当之多，如果是开发新产品……这些情况的出现将使得中间的“文档”或“模型”变得必要起来。正如 Philippe Kruchten 所说，“我写是为了更好地理解我所说的”。

足球比赛中，最终目的就是（1）不让对方进球；（2）进球。但演变了多年，主流阵形并不是 9-0-1 或 1-0-9，而是 4-4-2。为什么需要“中场”这个环节，而且至关重要？这个无奈的妥协，恐怕与人的体能限制有关。如果场地缩小成篮球场这么大，是不是还是这样的阵形？如果场地扩大到飞机场这么大呢？

不同的场地需要不同的战术；不管采用什么战术，传球、抢断、带球、射门这些技能是越强越好。软件开发也是一样，不同的情况需要采用不同的具体流程，但不管是采用什么流程，找到问题的技能（需求）和解决问题的技能（设计）越强越好。用例技术可以用于改进前者，对象技术可以用于改进后者。两种技术的原理是类似的：对象技术用“对象”封装了同源的数据和行为，用例技术用“价值”封装了同源的各种需求。

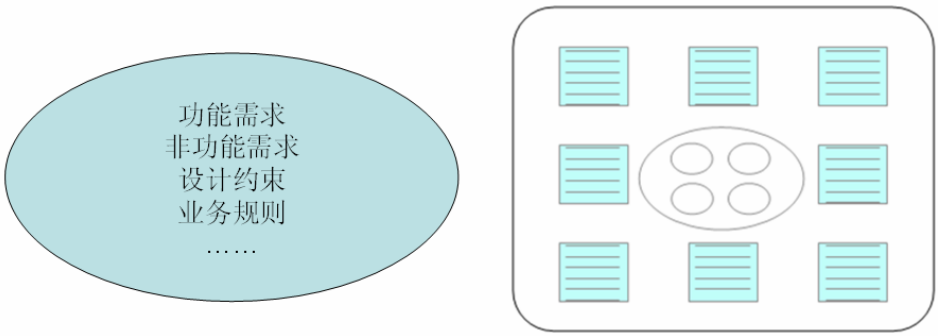


图 7 用例和对象技术的意义

有一句名言：顾客不是要买 1/4 英寸的钻头，而是要买 1/4 英寸的洞。我们也可以说，团队要过程、方法、工具，更需要在项目中不断复制成功。

（本文首发于《程序员》2005年2期）