利用 Rational Unified Process 开发大规模系统

Maria Ericsson

Rational Software 白皮书

TP 156

Rational
the software development company

目录

历史记录

本文是根据发表在 ROAD 1995 年 5-6 月期的由 Ivar Jacobson、 Karin Palmkvist 和 Susanne Dyrhage 合著的 "Systems of Interconnected Systems" 进行撰写的[1]。本文吸收了数个大规模系统开发项目的宝贵经验，并有意将它们与 Rational Unified Process 版本 5.1 [2] 和统一建模语言结合起来 [3]。

互连系统构成的系统

开发大规模系统时，其复杂程度将大大增加。它不但要求您能够理解一套更为复杂的工件，并且由于需要管理更多的一组资源，您为此还要负担额外的开销。本文描述了一个构架模式，它用于帮助管理新增的复杂性开销。该构架模式在 [4] 的其他地方讨论时被称为**互连系统构成的系统。**

在构建诸如命令和控制系统等大型复杂系统或高度集成的 IT 解决方案时，这种构架模式是很有用的。这些类型的“超系统”在大多数情况下分成几个不同的部分，每个部分作为单独的系统独立开发。超系统通过一组互连系统实现，而互连系统之间相互通信，履行超系统的职责。其中一个系统体现整体性能，我们称之为**上级系统**。其余系统代表整体的一个部分，我们称之为**从属系统**。上级系统与实现它的从属系统截然不同。不同类型系统之间的关系泾渭分明：从上级系统的角度来看，从属系统是子系统，请参见图 1。

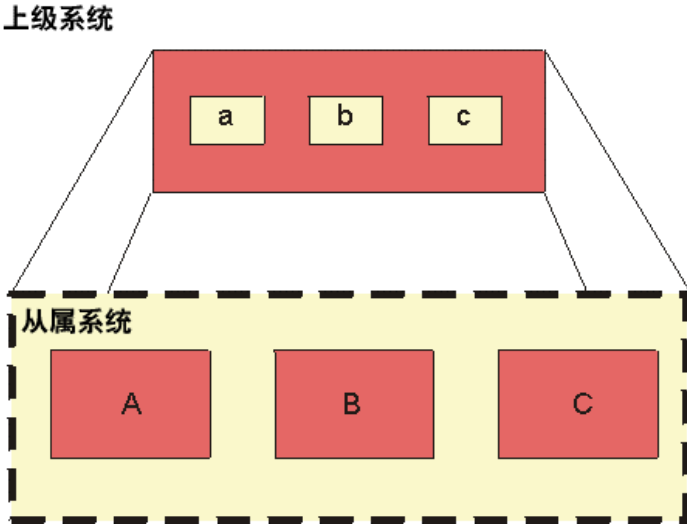

图 1. 上级系统的规约由一个互连系统构成的系统来实施，其中系统 A、B 和 C 分别是上级系统的子系统 a、b 和 c 的实施。

区分上级系统及其从属系统有几个好处：

- 从属系统在生命周期内的所有活动中都可以单独管理，其中包括销售和交付。
- 通过把从属系统插入到由互连系统构成的其它系统，就很容易使用从属系统来实施其他上级系统。

- 在开始构建系统的时候，并不总是知道它是不是一个由互连系统构成的系统。您可以从“简单的”系统视图开始，在生命周期的晚期再决定是否需要应用由互连系统构成的系统模式。
- 您不必开发新版本的上级系统，就可对从属系统进行内部变更。只有在主要功能变更时才要求开发上级系统的新版本。

每个从属系统都有一组相关的工件，工件之间有清晰的可追踪性关系。从从属系统的工件集到上级系统的相应工件集，它们之间也有可追踪性。每个从属系统可以作为独立的开发项目进行管理，它们都有自己的生命周期阶段：先启阶段、精化阶段、构建阶段、移交阶段。

如果正在构建的“超级系统”规模很大，可能需要进一步划分从属系统，从而将其作为一个由互连系统构成的系统。

软件开发生命周期

在 Rational Unified Process，开发生命周期从两个角度进行介绍和讨论：管理角度和开发角度，请参见图 2。

从管理角度来看，开发一个系统或者开发新一代的系统都要经历四个生命周期阶段。从开发角度看，您以迭代方式开发日臻完善的系统版本。迭代过程中执行的活动在 Rational Unified Process 分成一组核心工作流。每个核心工作流着重描述系统的某个方面，最终形成一个系统模型或文档集。

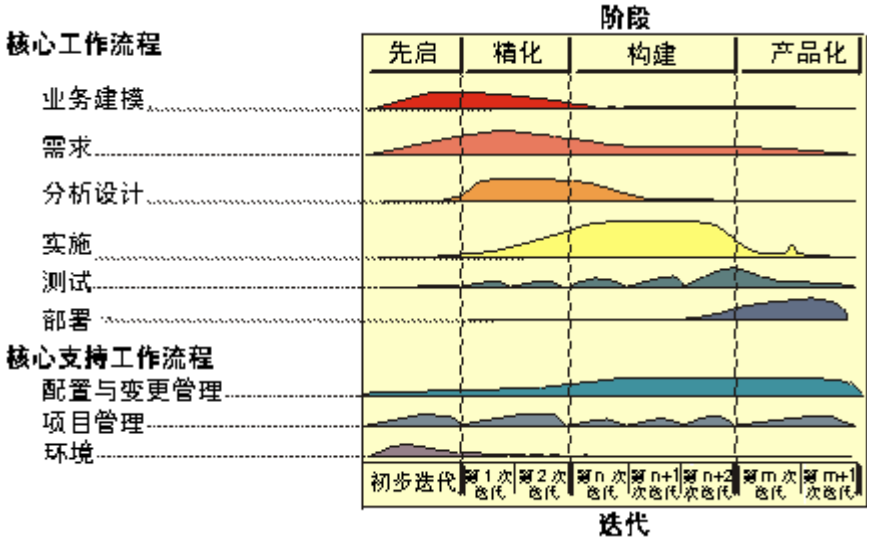

图 2. 迭代模型。

将其推广到互连系统构成的系统，上级系统及其每一个从属系统都要经历自己的生命周期，并且它们经常被当作单独的项目。

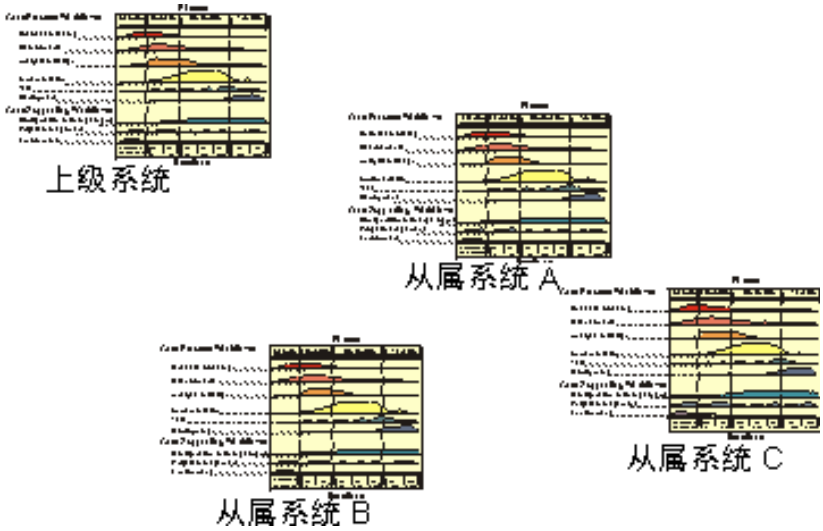

图3. 每个系统，包括上级系统和从属系统，都要经历自己的生命周期。

当然，生命周期确实存在依赖关系。如何正确管理依赖关系是开发由互连系统构成的系统所面临的难题之一。依赖关系有以下类型：

- 生命周期与时间相关。上级系统的生命周期首先开始。一旦上级系统经历过至少一次迭代而且从属系统的接口相对稳定，从属系统的生命周期即可开始。实际上，在上级系统至少经历一次迭代之前，您甚至可能不知道哪一个是从属系统。
- 从属系统的接口一旦稳定，上级系统的生命周期即可进入维护阶段。这意味着不进行主动开发，除非出现问题才要求 修改从属系统的接口。
- 从属系统的接口由开发上级系统的人掌握。有关接口的更多信息，请参见[3]和[5]。
- 实现从属系统接口的类由开发从属系统的人所有。

系统开发的工作流和工件

有一个自然前提，即上级系统和从属系统可以使用相同的工件集和通过相同的工作流来进行开发，这在开发非合成系统时尤为典型。在我们继续说明如何执行这一前提前，需要先引入工件和工作流。在 Rational Unified Process 中，我们引入了五个核心工作流，请参见图 4。它们分别是：

- 业务工程 - 目的是评估即将使用该系统的组织，更好地理解通过系统解决的需求和问题。最终结果是得到一个用例模型和业务对象模型。该工作流是可选的。如果即将部署系统的组织结构过于简单，则工作流可能不会增值。
- 需求 - 目的在于获取和评估需求，重点放在可用性。其结果是生成一个用例模型，其中主角代表与系统通信的外部单元，用例代表事务序列，为主角提供可测量的结果值。

- 分析设计 - 目的在于调查预期实施环境及其对系统构建的影响。其结果是生成一个对象模型（设计模型），包括用例实现。这些用例实现说明了对象如何进行通信以执行用例流。该工作流可能包括类和子系统的接口定义，指定它们在所提供的操作上的责任。这个对象模型还在实施语言、分布等方面进行改造，使之适应实施环境。将分析结果作为一个单独的模型考虑有时很有用，这时该模型称为分析模型。

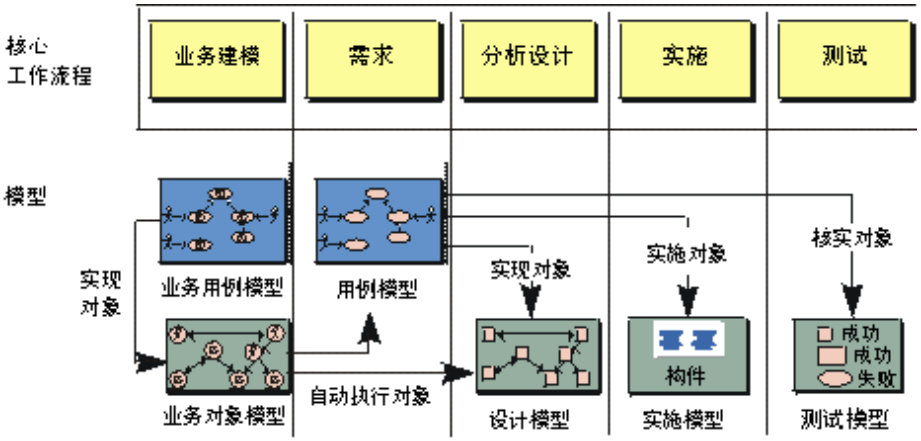

图 4. 每个核心工作流都与一个特定的模型集相关联。

- 实施 - 目的是在规定的实施环境内实施系统。其结果是生成源代码、可执行代码和文件。
- 测试 - 目的在于保证系统与预期系统相吻合，并且在实施中没有出现错误。它生成一个合格的、准备交付的系统。

互连系统构成的系统的开发

我们必须完成的工作是确定如何能够将一个系统的职责分布在几个系统之间，每一个系统负责一个明确定义的责任子集。这就是说，主要目标是定义这些从属系统间的接口。实现上述主要目标之后，其余工作就可以根据“分而治之”的原则，由每个从属系统独立完成。因此，除了在实施以后对系统进行测试之外，这就是我们要为系统做的全部工作。

分解标准

那么如何决定是否将系统分解为由互连系统构成的系统？下面是应值得注意的两个特征：

- 如果系统具有相当的规模和复杂性，则可以把问题分成许多小问题，这样更容易理解。
- 是否正在处理物理上独立的系统？在处理遗留系统或者遗留旧构架时常常会遇到这种情况。
- 分解有助于定义系统各部分之间的自然接口和窄接口。
- 您可以决定使用一些重要的 COTS（市售）产品来实施系统的一个部分。分解将有助于明确您打算如何使用 COTS 产品。

- 分区可以最大限度地发挥分布式开发组织的潜力，并且清楚划分在地理上分散的几个团队之间的工作。

要考虑以下风险包括：

- 过度分割会导致只见树木不见森林的问题。
- 使用物理上分离的系统或者物理上分离的团队，有扼杀任何形式的复用的危险，最后得到的是一个僵化的系统。

组织

要降低上述风险，关键是要委派一组人监督整个开发工作。这个小组通常称作构架设计师团队，他们应该重点关注以下问题：

- 定义一个整体构架，并且从属系统遵守该构架。
- 适当关注从属系统之间的复用和经验共享。
- 对生产什么工件、从属系统工件与上级系统工件之间有什么关系有清晰的理解。
- 定义一个有效的变更管理策略，并得到所有团队的遵守。

构架设计师团队可以（但不总是）控制上级系统的开发。有关组织方面的更充分讨论，请参见[6]。

上级系统的生命周期

首先，可以选择执行业务工程，以便更好地理解系统的环境。在以下情况中，它可以创造价值：

- 开发人员需要更好地理解组织，
- 组织本身的业务运作方式存在不同类型，术语和流程需要统一，或者
- 软件工程与业务重建工程一同完成。

另请参见[6]。

该工作将产生一个业务用例模型和一个业务对象模型。另一种方法是，选择执行有限的业务工程，只关注业务领域的关键概念，并在业务模型对象中将其记录下来。这种方法通常称为领域建模。

一旦对业务模型集采取了措施，您需要开始获取整个系统的需求。我们对由互连系统构成的系统的需求建模要求与其他系统一样。用例模型是一种非常自然的表示结果的方法，请参见[7]。考虑该上级用例模型最直截了当的方式就是假定它完全获取了系统的行为需求。然而情况可能很少如此。既然我们需要用其他系统来实施该系统，整体系统很可能相当复杂。因此，在这个级别上钻牛角尖就不是一个好主意。因而，上级用例模型通常给出一个完整但经过简化的系统功能需求图。在该级别上没必要过于详细，因为详细建模将在每个从属系统中进行。而且，许多需求在分成几个子系统的上级用例中并不一定看得见，这也是事实。此类需求可以说是子系统的“局部”需求。

分析设计的目的是为了获得一个强壮的系统构架，当然这对由互连系统构成的系统是极其重要的。上级系统的开发人员必须获得一个强壮的从属系统结构，但他们根本不必为内部结构劳心费力。因此我们将使用子系统的概念对系统划分进行建模，将系统分成许多更小的部分。为了得到正确的子系统集，并得到如何在这些子系统之间分配上级系统的职责的初步构思，我们开发了一个分析模型。在执行高级用例时，分析类应该表现系统内的事务所承担的角色。因此，分析模型与高级用例模型类似，给出了完整对象结构的一个简化视图。

将功能相关的分析类进行组合，分到不同的子系统中。因而，我们得到的子系统结构从它仅仅基于功能标准的意义上看是理想的，例如，我们并未考虑任何需求分布。遗留系统的存在通常是一个影响巨大的因素。遗留系统可能执行一些或者许多在分析模型中定义的职责。这些系统的存在还可能引起对分析中发现的职责再进行划分，以便能最大限度地复用现有的能力。
设计结果可能是一个子系统结构，它与分析过程中基于功能标准所定义的结构差别很大。因此我们最后得到了一个设计子系统结构，每一个子系统将通过一个从属系统进行实施（请参见图 5）。为了能够继续每个此类系统各自的开发工作，我们为每个子系统定义了接口。实际上，定义接口是在上级系统级别执行的最重要活动，因为接口提供了开发从属系统的规则。不需要定义设计类，唯一要做的事情就是定义设计子系统的接口。

实施无需作为上级系统生命周期的部分来进行，而只执行一些原型设计工作，研究系统特定方面的技术。

最后的工作流是**测试**，这里指的是组装不同从属系统时的集成测试，并还测试每个上级用例是否根据其规约通过互连系统协作获得执行。

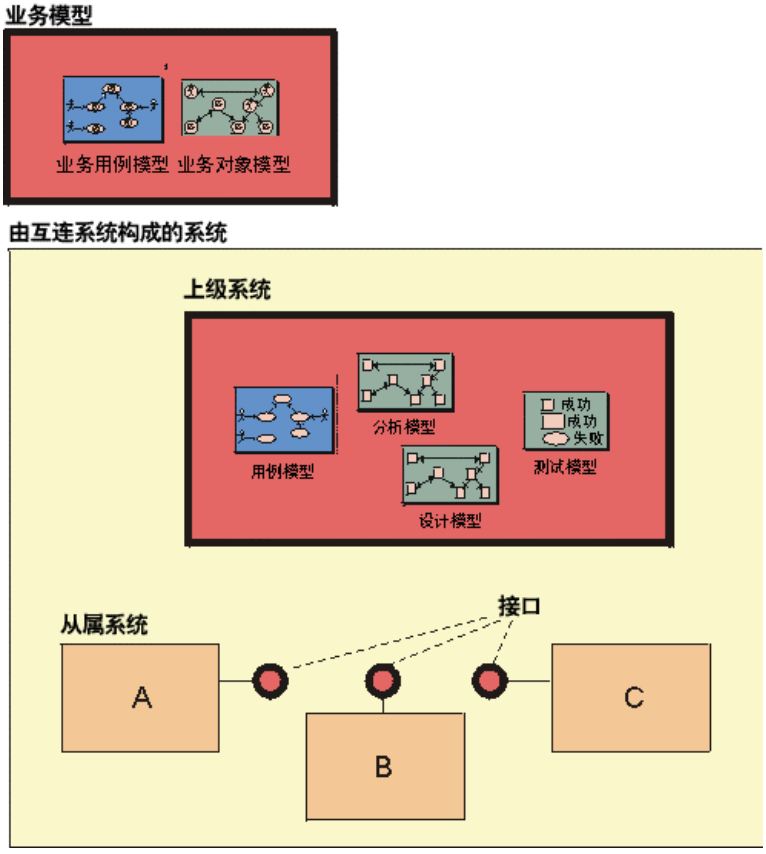

图 5. 上级系统通过一个模型集来描述，其中高级设计模型定义的子系统将通过从属系统实施。从属系统的接口归上级系统所有。

为说明如何使用上级系统，下面给出了几个迭代计划样本；一个是上级系统生命周期先启阶段的迭代计划，另一个是精化阶段的迭代计划。我们使用活动图来描述迭代计划。这些图中的动作状态对应 Rational Unified Process 定义的工作流明细。

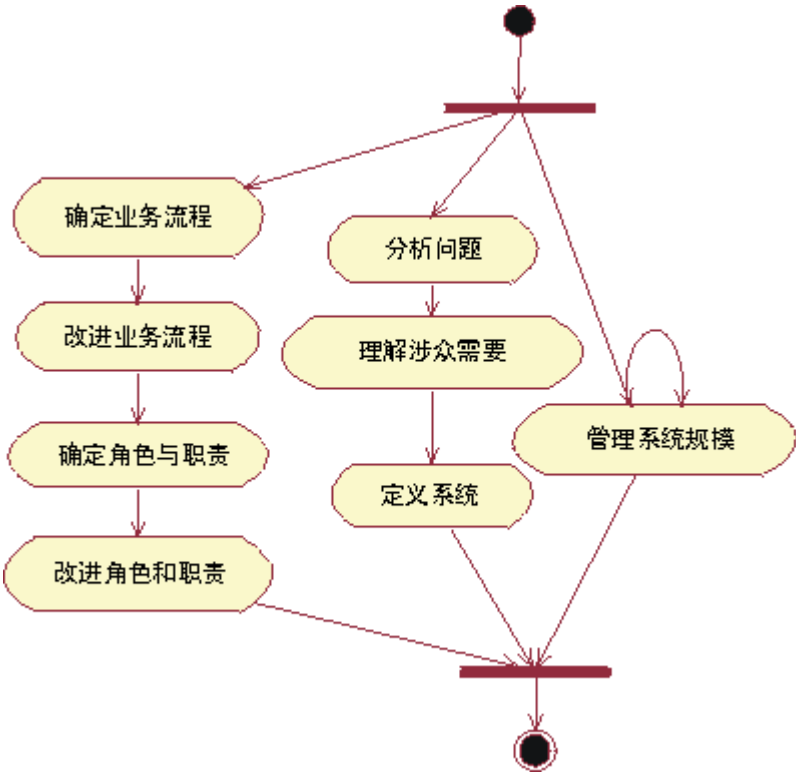

图 6. 描述上级系统先后阶段迭代计划示例的一个活动图。

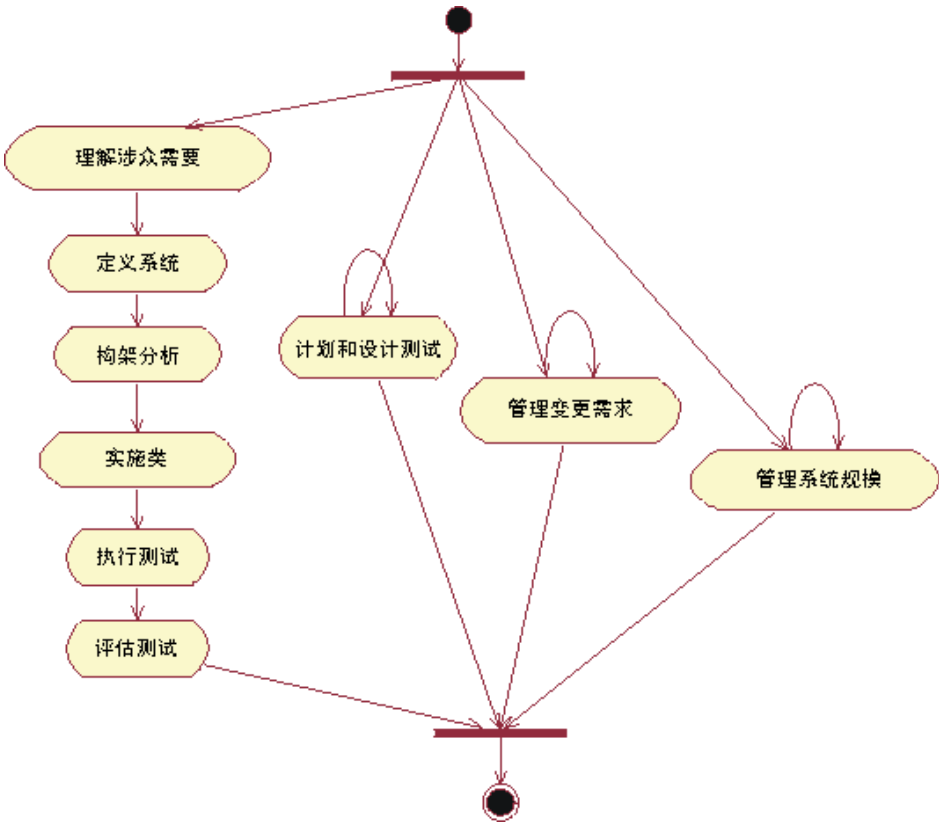

图7. 描述上级系统精化阶段迭代计划的一个活动图。由于您为了研究系统的有关技术方面执行了有限的原型实施，这里出现了动作状态“实施类”。

从属系统的生命周期

每个从属系统按正常方式进行开发，把其作为主角与其他系统通信的黑箱来考虑。如前所述，您可以为每个子系统执行正常的活动集并开发通常使用的模型集。如果上级级别的模型在所有明细中都有定义，那么在不同级别的模型之间都能得到完整递归，但如前所述，实际上情况很少是这样的。

对从属系统，您需要执行**需求**工作流。上级系统的接口和用例将成为你理解从属系统边界及其主角的主要输入。

在执行从属系统的**分析设计**时，上级系统定义的接口与高级用例一起将成为“边界条件”。

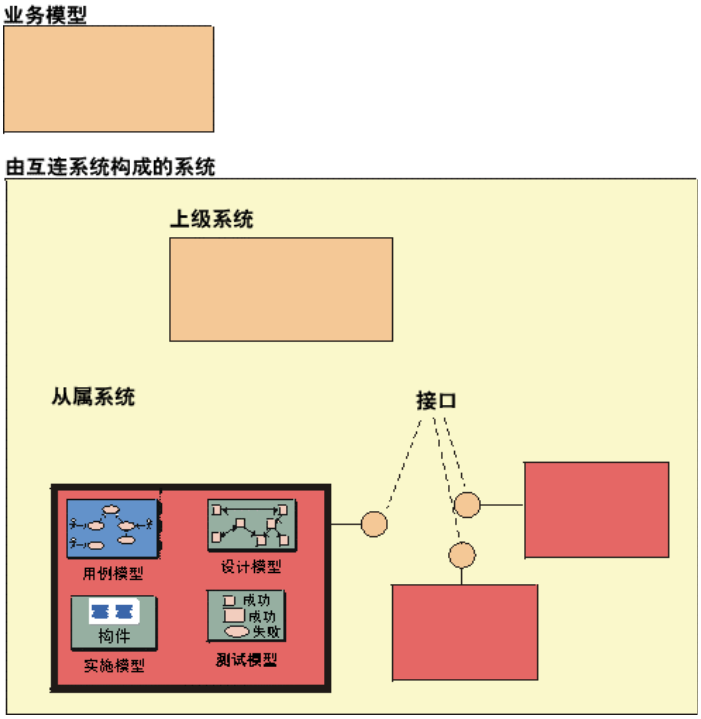

图 8. 从属系统由它们各自的模型集来描述。

为说明如何使用从属系统，下面给出了两个从属系统生命周期的迭代计划示例。

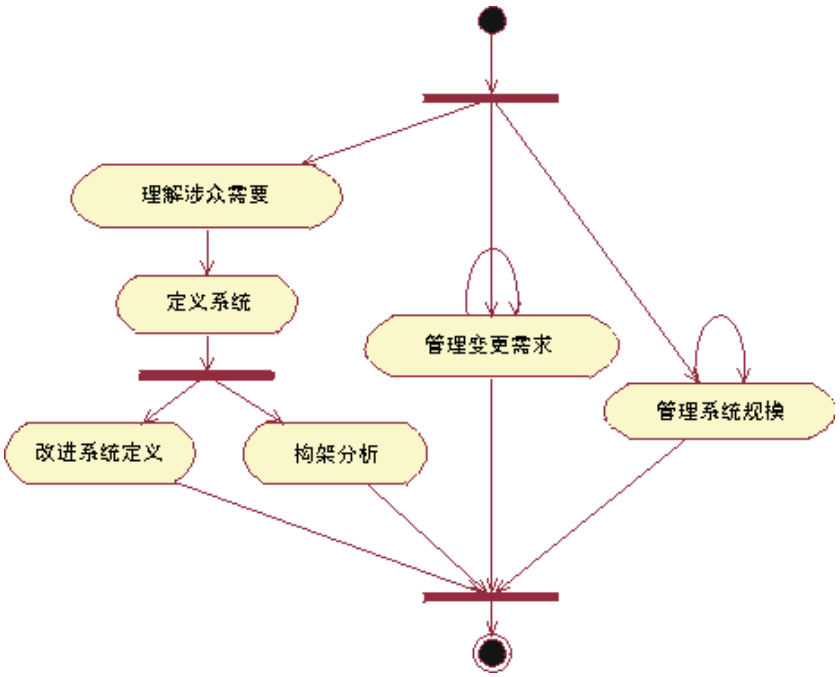

图 9. 从属系统先启阶段迭代计划示例。这是一个不完整的迭代，因为不产生任何可执行代码。

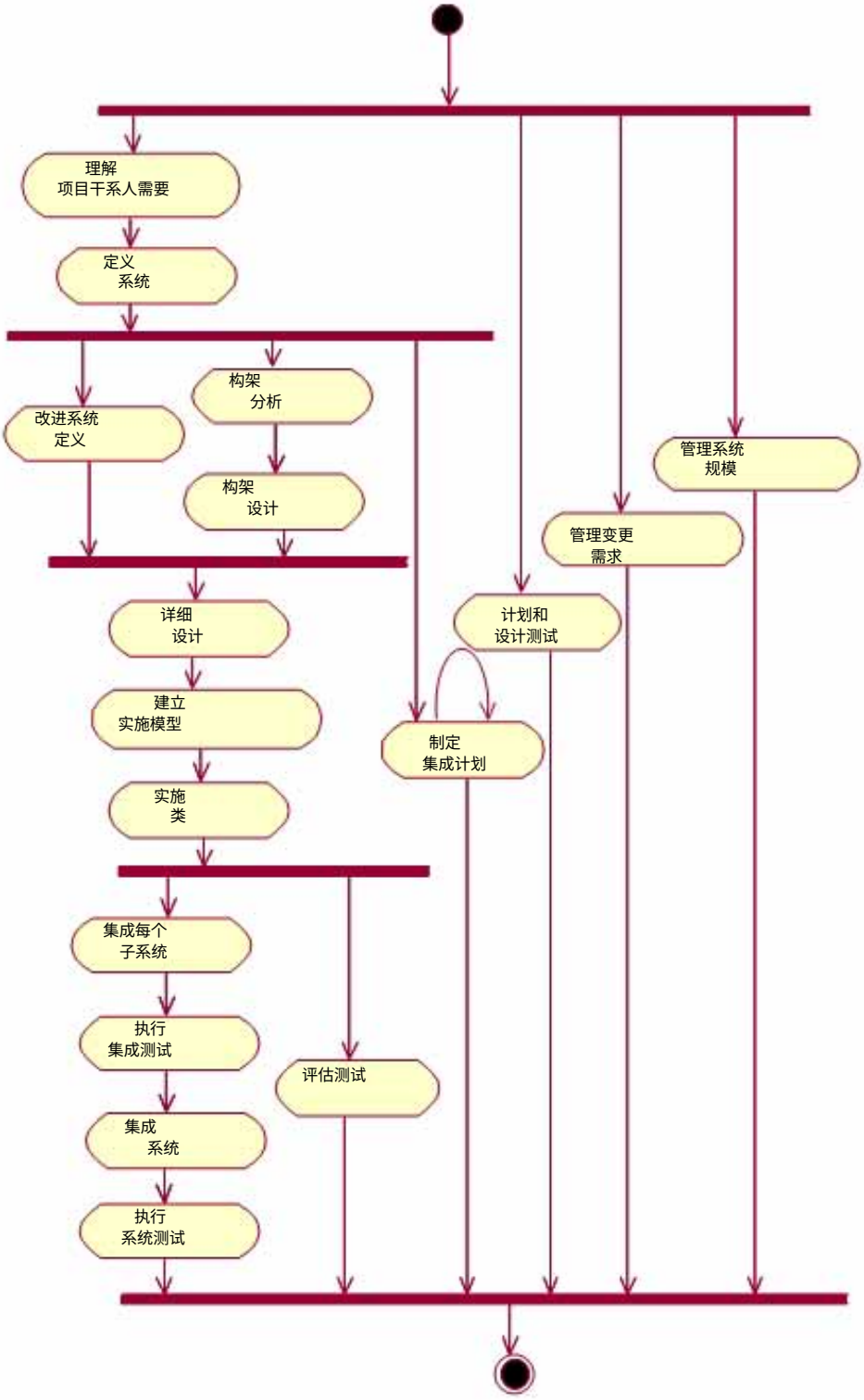

图 10. 从属系统精化阶段迭代计划示例。精化阶段的重点在于完成构架和已改进系统的定义上。

在互连系统构成的系统中的用例

您应该为每个系统，包括上级系统和从属系统，在互连系统构成的系统中建立一个用例模型。它们按以下方式相关（另请参见图 11）：

- 级系统的高级用例分解到子系统上（不一定，但通常是）。每个“分块”将成为从属系统模型中的一个用例，请参见图 11。
- 从一个从属系统的角度来看，其他从属系统是它用例模型的主角，请参见图 12。

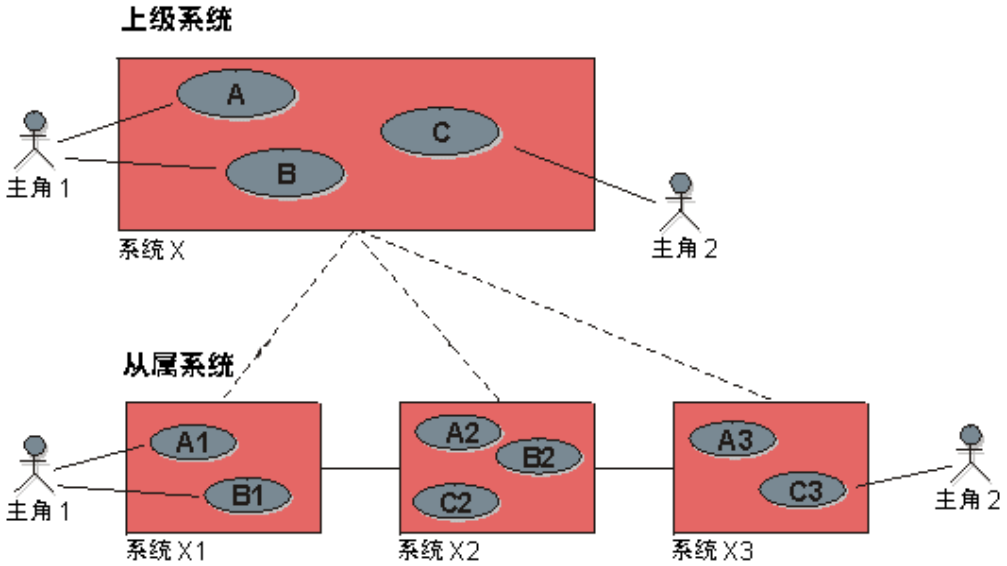

图 11. 上级系统的高级用例与从属系统的详细用例之间的关系。

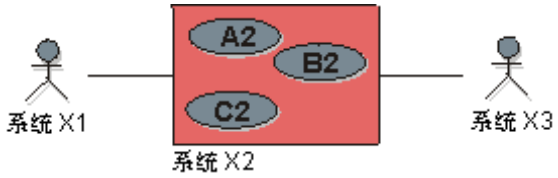

图 12. 在从属系统 X2 的用例模型中，从属系统 X1 和 X3 都被视作主角。

关于描述上级系统的用例有一些特殊注意事项。由于从某种意义上说，您将重新描述每一个从属系统的所有需求，因此深究这些用例毫无意义。在正常情况下，只要写下高级用例事件流的分步大纲就足够了，不必展开详述。

在该用例模型中，不应使用任何用例关系（泛化关系、扩展关系、包含关系）。一般情况下，它不创造价值，原因如下：

- 您不会详细描述高级用例，因此不必担心有关文字出现在几个地方。

- 无论如何，当您将高级用例分解到“从属系统”上时，都需要建立信息结构。将它与其他构造机制混合会产生疑惑。

但有一个重要例外，即如果您打算寻找由互连系统构成的系统中的可复用组件。建立上级用例模型以便查找一般用例模型，这是一种寻找可复用组件的好方法。有关该主题的详细信息，请参见[6]。

在互连系统构成的系统中的设计模型

由互连系统构成的系统中的每个系统，包括上级系统和从属系统，都应该有自己的设计模型。设计模型按以下方式相关：

- 上级系统设计模型中的子系统定义了从属系统的边界。
- 为上级系统子系统定义的操作是定义从属系统接口的输入。

对上级系统设计模型的描述不如描述从属设计模型详细。描述对象包括：

- 子系统，简要说明。
- 用例实现，利用子系统协作方式描述。记录这些高级用例实现的一般方法是绘制时序图。通过绘制这些图形，您可以定义在从属系统上的高级用例的“分块”，请参见图 13。
- 子系统的操作。
- 子系统的接口定义。

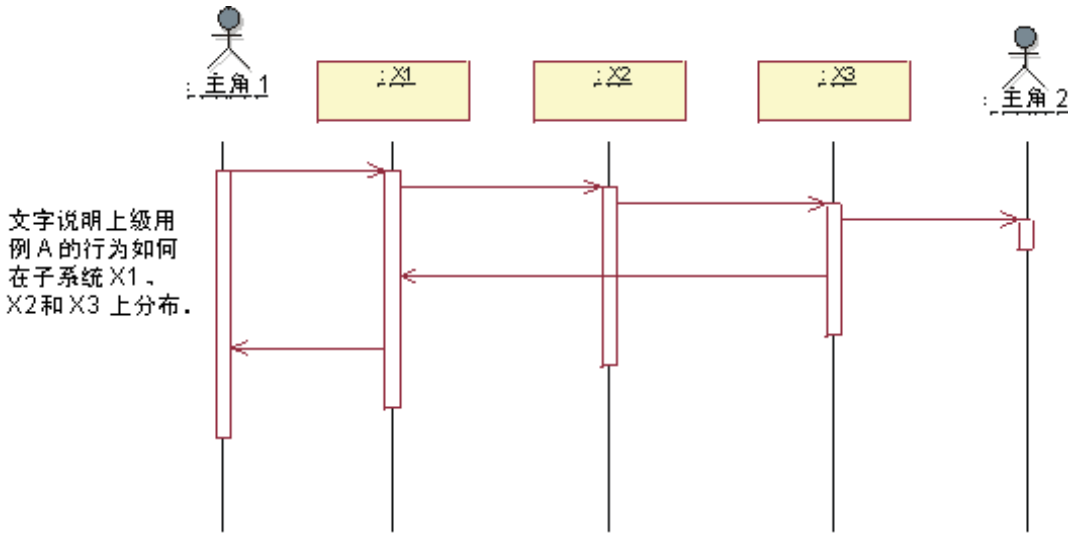

图 13. 实现上级用例 A 的时序图。

在互连系统构成的系统中的信息集

大多数组织在理解如何管理工件以及如何正确理解它们的依赖关系等方面都投入大量精力。我们在上一节特别讨论了上级用例模型和设计模型与从属用例模型和设计模型之间的依赖关系。另外还要考虑一些一般性的依赖关系问题。系统经过生命周期的某个关口后，将产生可组织成信息集[8]的工件，请参见图 14。这些信息集按“共同”演进的工件进行组织。

- 根据正在构建的应用程序类型，可以定制每个集合的确切内容，但集合保持不变。
- 您需要理解集合之间的依赖关系，这样才能有效地维护工件之间的可追踪性。

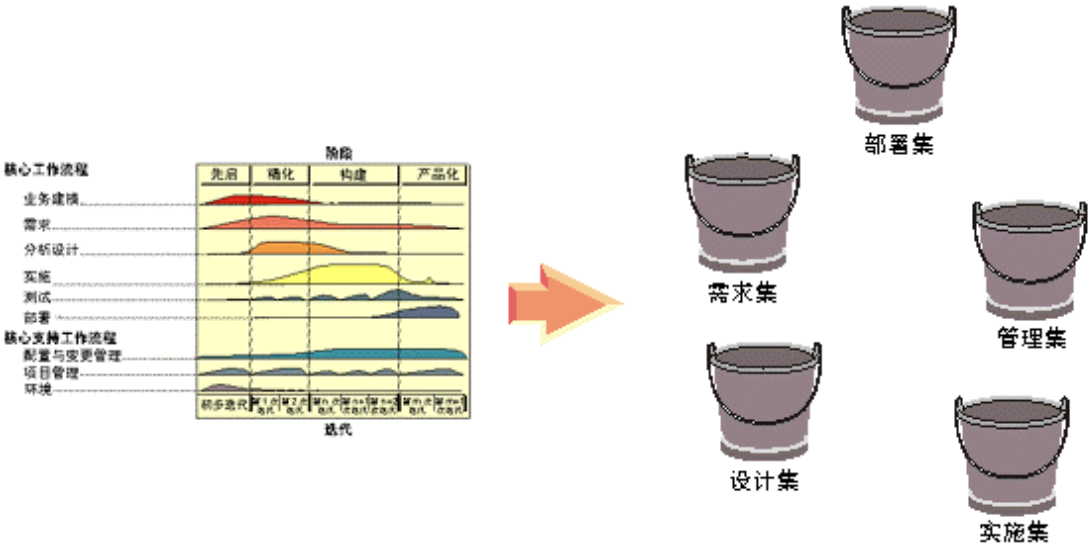

图 14. 系统的生命周期将产生信息集。

在互连系统构成的系统中，上级系统和每个从属系统将产生各自的信息集，请参见图 15。

- 从属信息集对相应的上级信息集有依赖关系。
- 由于应用程序类型不同，上级系统之间的相应信息集的内容类型也可能不同。
- 相关的从属信息集应该是独立的，只是它们服从于上级系统定义的相同子系统接口。

在维护上级系统与从属系统工件之间可追踪性上投入的精力应该尽量要少。应确定维护系统内的可追踪性的优先顺序。

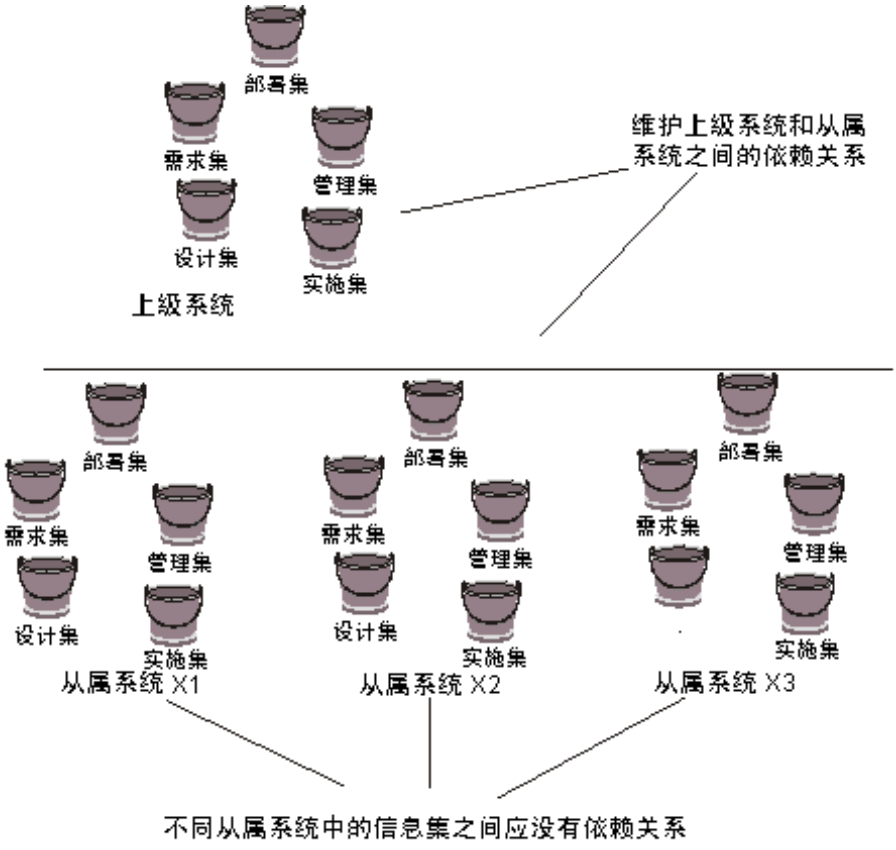

图 15. 在互连系统构成的系统中的每个系统将生成自己的信息集。

互连系统构成的系统之构架

由互连系统构成的系统中的每个系统，包括上级系统和从属系统，都应该定义自己的构架。

对于上级系统，构架文档应包括以下方面：

- 上级系统的关键用例或场景。
- 对互连系统构成的系统的分层。
- 如何处理从属系统之间的复用以及复用内容。
- 所有从属系统都能使用的通用关键机制及其实施。例如，所有从属系统都应该使用通用的通信、错误报告、容错管理机制，否则上级系统不会表现为同类系统。

对于从属系统，构架文档应该澄清以下问题：

- 从属系统在互连系统构成的系统内的角色。
- 从属系统的关键用例或场景。
- 从属系统如何使用为由互连系统构成的系统定义的分层结构。换句话说，就是定义从属系统如何履行在互连系统构成的系统的分层结构中为它定义的职责。

- 使用哪些通用的关键机制、如何使用用以及添加哪些应用程序专用的关键机制。
- 如何复用。确切地说，哪些子系统是两个或多个从属系统公有的，需要建立哪些机制来允许从属系统进行通信。

系统之间的关系

您已经看到，常用的系统开发活动也可应用于通过互连系统构成的系统实施的系统。这是非常有利的，因为它意味着你不必用一种与其他系统大相径庭的方式处理此类系统。您还可以很好地将上级系统与其以其他从属系统形式进行的实施分离。在互连系统构成的系统中的每个系统都有自己的生命周期。既然每个系统可能有不同的特点，您可以使用不同的开发流程来生成系统。依据 Rational Unified Process [2]，每个系统将有一个不同的开发案例。最后注意在互连系统构成的系统中所涉及系统之间的独立性：首先，查看一下从属系统。在上级系统的设计模型中，每个从属系统实施一个子系统。子系统依赖于彼此的接口，但相互之间不直接依赖，请参见图 12。因此，可以用新版本替换其中的一个子系统，只要新的子系统使用同样的接口，就不会影响其他子系统。您将在从属系统之间得到完全相同的关系。每个从属系统都将其周围环境看作一个接口集。这意味着您可以更换系统，只要新系统对其他系统扮演的角色相同就可以，例如新系统可以用相同的接口集来表示。系统相互引用各自的接口，这是由上级模型中子系统和接口之间的对应关系指定的。

在从属系统的用例模型中，与之交互的其他从属系统的接口表示为主角。可以说，从属系统把另一个系统的接口看作是由相关主角提供的，因此永远不必直接引用其他系统，请参见图 16。注意在图 12 中，有几处地方出现了接口 B，说明接口 B 是上级系统中的子系统和相应从属系统实际引用的同一个接口。

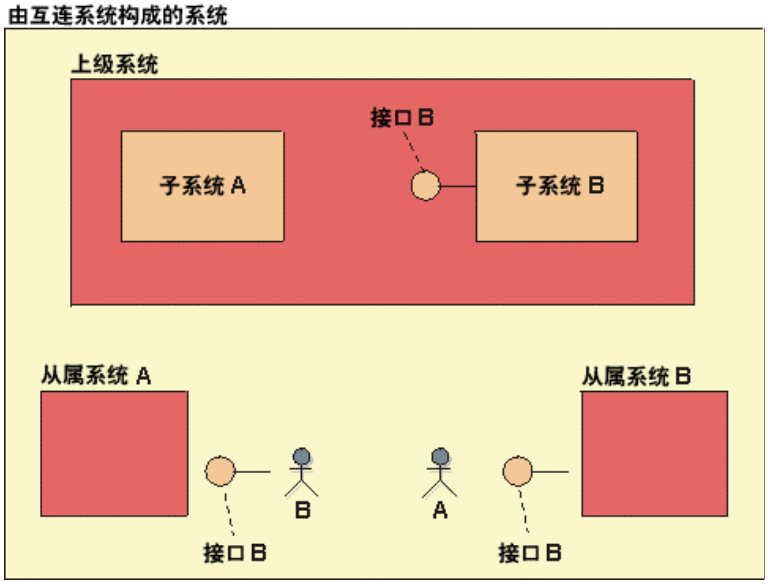

图 16. 上级系统的子系统仅仅通过接口相互依赖。因此实施从属系统就得到了同样的独立性。在上级系统的模型中，子系统 B 提供与其他子系统连接的接口 B。因而，相应的从属系统 B 也需向其他从属系统提供相同的接口 B。

那上级系统情形又如何呢？它与从属系统有什么关系？从以下方面讲，它独立于其实施系统：每个这类的系统仅仅是上级系统模型中所指定内容的一个实施，并不属于上级规约的一部分。出于实际原因，为了追踪需求，您需要定义不同级别上系统之间的可追踪性链接，而最“简洁”的方法就是只在接口之间定义此类链接，请参见图 11。事实上，甚至可以说从属系统只不过是提供了上级模型中所定义接口的实施。

但是，这仅仅对简单系统成立。接口除了说明在一个特定交互点所发生的事情之外，并未指定任何东西。一个从属系统可能有 100 个接口，每个接口有 10 个操作。在接口说明中，把某个接口的输入与另一个接口的一个或多个输出相关联是不切实际的。这就是为什么需要用例来解释从属系统的语义的原因。可以得出结论，当一个系统由互连系统构成的系统实施时，所涉及的每一个系统都独立于其他系统，但它们严重依赖彼此的接口。这就为您提供了一个很好的从属系统并行开发平台。

应用范围

互连系统构成的系统的构架和建模技术可以用于多种系统，如：

- 分布式系统
- 很大或者很复杂的系统
- 综合几个业务领域的系统
- 复用其他系统的系统
- 系统的分布式开发

情况也可以反过来：从一组现有的系统，通过组装系统来定义由互连系统构成的系统。实际上，在某些情况下，大型系统在演进的早期阶段就是以这种方式发展的。你意识到你有几个意识到你有几个创建一个“大型”系统，它比两个独立的系统能创造更多价值。

事实上，如果一个系统的不同部分本身可以看作系统，建议把它定义为由互连系统构成的系统。即使现在它还是单个的系统，由于分布式开发、复用或者客户只需购买它的几个部分等原因，以后也会证明把系统分割成几个独立的产品是有必要的，这里仅仅举几个示例。

作为结束，我们来认真分析两个可以使用由互连系统构成的系统构架的案例。在每一个示例中，我们将说明讨论中的系统既要作为单个的系统考虑，又要作为分离系统的集合考虑，以此说明它应该作为通过由互连系统构成的系统实施的上级系统来对待。

大规模系统

电话网络可能是世界上最大的由互连系统构成的系统。这是一个很好的示例，在电话网络中需要管理两个以上系统级别的复杂性。它还作为这类案例的示例：顶层上级系统由一个标准化实体掌握，不同的竞争公司开发一个或几个必须符合该标准的从属系统。这里，我们将讨论移动电话网络 GSM （全球移动电话系统），说明将大规模系统作为互连系统构成的系统来实施的优势。

大规模系统的功能通常包含几个业务领域。例如，GSM 标准包括了从呼叫用户到被呼叫用户的整个系统。换句话说，它既包括移动电话的行为，也包括网络节点的行为。由于系统的不同部分是单独购买的产品本身，甚至是由不同客户购买的，因而它们本身可当作系统。例如，开发完整 GSM 系统的公司向用户销售移动电话，向话务员销售网络节点。这是把 GSM 系统的不同部分当作不同从属系统的一个原因。另一个原因是，把 GSM 这样大型复杂的系统作为单个系统进行开发的时间太长；不同的部分必须由几个开发团队并行开发。

另一方面，由于 GSM 标准包括整个系统，因此有理由将系统作为一个整体即上级系统来考虑。这有助于开发人员理解问题领域，以及不同部分是怎样彼此相关的。

分布式系统

在分布在几个计算机系统的系统中，使用由互连系统构成的系统构架是非常合适的。顾名思义，分布式系统至少由两个部分组成。由于分布式系统中必须有明确定义的接口，因此这些系统也非常适合进行分布式开发，也就是说，几个独立的开发团队并行开发。分布式系统的从属系统本身甚至可以当作产品来销售。因而，将分布式系统视为独立系统的集合是很自然的。分布式系统的需求通常包括整个系统的功能，有时不预先定义不同部分之间的接口。而且，如果对开发者而言问题领域是陌生的，他们首先需要考虑整个系统的功能，而不管它将如何分布。这是把它当作单个系统看待的两个重要原因。

遗留系统的复用

大型系统常常复用遗留系统。遗留系统可作为从属系统来描述。为了理解遗留系统如何在上级系统的大环境中工作，需要为它“重建”一个用例模型，有时还有一个分析模型。这些重建的模型不必完整，但至少要包含对互连系统构成的系统的其余部分有直接影响的遗留功能，否则需要进行修改。

使用预制包

系统可以集成和定制两个或多个预制包。企业资源规划(ERP)系统就是一个典型示例。许多 ERP 系统是 MRP（材料资源规划）、库存管理、供应链管理等从属系统的组合。其他领域也有类似组合，如人力资源和工资单应用等。它们就像预制系统一样，为了得到一个完整的系统，必须令其专用化并与其他标准包互连。为理解包集合的作用，我们需要上级系统。这是在金融界许多客户所面临的情况。

总结

本文提出了一种由互连系统构成的系统构架模式。这个构架模式不但允许在一个模型内递归，它还把每个子系统本身作为一个系统考虑，并允许在每个系统所有的工件集之间递归。引入的构架用于那些由几个相互通信的系统实施的系统。所涉及的每个系统用它自己的模型集描述，与其他系统的模型分开。采用这种技术的优势是显而易见的：可以处理相当复杂的问题，并使用“分而治之”的原则来理解它们。然而，缺点就是您的开销增大，进度无法同步。我们还考察了几个示例，在这些示例中，组织发现上级系统很难使用迭代式生命周期，从而承担风险被推到上级系统生命周期末尾的危险。您还要注意执行合理而有效的复用策略，以避免开发出一个“僵化”的系统集。本文给出的示例说明了由互连系统构成的系统建模构架可用于许多应用领域。实际上，只要系统的不同部分本身可看作系统，就可以使用本文提出的构架。

参考资料

[1] - Jacobson, I.; Palmkvist, K.; and Dyrhage, S., *Systems of Interconnected Systems*, ROAD, 2(1), 1995. [2] - *Rational Unified Process* version 5.1.

[3] - Rumbaugh, J.; Booch, G.; Jacobson,I., *UML Reference Manual*, Addison Wesley Longman, 1999. [4] - Herbert A. Simon, *The Sciences of the Artificial*, MIT Press, 1981.

[5] - Jacobson, I.; Bylund, S.; Jonsson, P., *Using Contracts and Use Cases to Build Plugable Architectures*, Journal of Object-Oriented Programming, May/June, 1995.

[6] - Jacobson, J.; Griss, M.; Jonsson, P., *Software Reuse - Architecture, Process and Organization for Business Success*, Addison Wesley Longman, 1997.

[7] - Jacobson, I., *Use Cases in Large-Scale Systems*, ROAD, 1(6), 1995.

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

Rational、Rational 徽标和 Rational Unified Process 是 Rational Software Corporation 在美国和／或其他国家或地区的注册商标。Microsoft、Microsoft Windows、Microsoft Visual Studio、Microsoft Word、Microsoft Project、Visual C++ 和 Visual Basic 是 Microsoft Corporation 的商标或注册商标。其他所有名称均仅用于标识目的，它们是其相应公司的商标或注册商标。ALL RIGHTS RESERVED.

Copyright 2006 Rational Software Corporation.
如有更改，恕不另行通知。