架构蓝图--软件架构 "4+1" 视图模型

Philippe Kruchten
2005 年 1 月 01 日发布

引言

我们已经看到在许多文章和书籍中，作者欲使用单张视图来捕捉所有的系统架构要点。通过仔细地观察这些图例中的方框和箭头，不难发现作者努力地在单一视图中表达超过其表达限度的蓝图。方框是代表运行的程序吗？或者是代表源代码的程序块吗？或是物理计算机吗？或仅仅是逻辑功能的分组吗？箭头是表示编译时的依赖关系吗？或者是控制流吗？或是数据流吗？通常它代表了许多事物。是否架构只需要单个的架构样式？有时软件架构的缺陷源于过早地划分软件或过分的强调软件开发的单个方面：数据工程、运行效率、开发策略和团队组织等。有时架构并不能解决所有"客户"（或者说"风险承担人"，USC 的命名）所关注的问题。许多作者都提及了这个问题：Garlan & Shaw 1、CMU 的 Abowd & Allen、SEI 的 Clements。作为补充，我们建议使用多个并发的视图来组织软件架构的描述，每个视图仅用来描述一个特定的所关注的方面的集合。

架构模型

软件架构用来处理软件高层次结构的设计和实施。它以精心选择的形式将若干结构元素进行装配，从而满足系统主要功能和性能需求，并满足其他非功能性需求，如可靠性、可伸缩性、可移植性和可用性。Perry 和 Wolfe 使用一个精确的公式来表达，该公式由 Boehm 做了进一步修改：

软件架构 = {元素，形式，关系/约束}

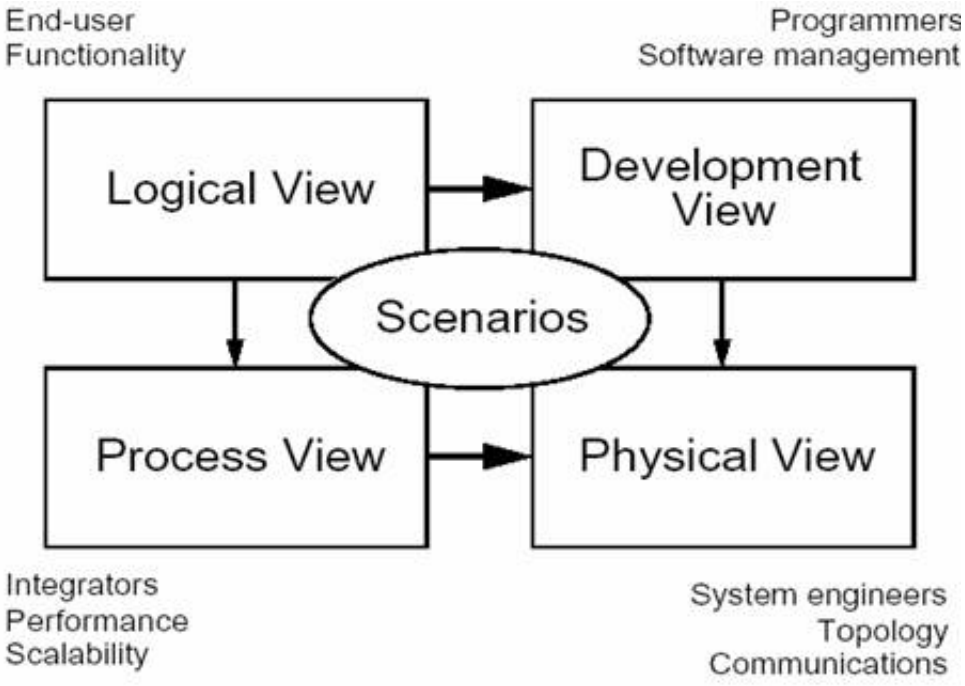
软件架构涉及到抽象、分解和组合、风格和美学。我们用由多个视图或视角组成的模型来描述它。为了最终处理大型的、富有挑战性的架构，该模型包含五个主要的视图（请对照图 1）：

- 逻辑视图（Logical View），设计的对象模型（使用面向对象的设计

方法时）。
- 过程视图（Process View），捕捉设计的并发和同步特征。
- 物理视图（Physical View），描述了软件到硬件的映射，反映了分布式特性。
- 开发视图（Development View），描述了在开发环境中软件的静态组织结构。

架构的描述，即所做的各种决定，可以围绕着这四个视图来组织，然后由一些用例 （use cases）或场景(scenarios)来说明，从而形成了第五个视图。正如将看到的，实际上软件架构部分从这些场景演进而来，将在下文中讨论。

图 1 － "4＋1"视图模型



我们在每个视图上均独立地应用 Perry & Wolf 的公式，即定义一个所使用的元素集合（组件、容器、连接符），捕获工作形式和模式，并且捕获关系及约束，将架构与某些需求连接起来。每种视图使用自身所特有的表示法－蓝图（blueprint）来描述，并且架构师可以对每种视图选用特定的架构风格（architectural style），从而允许系统中多种风格并存。

我们将轮流的观察这五种视图，展现各个视图的目标：即视图的所关注的

问题，相应的架构蓝图的标记方式，描述和管理蓝图的工具。并以非常简单的形式从 PABX 的设计中，从我们在Alcatel 商业系统（Alcatel Business System）上所做的工作中，以及从航空运输控制系统（Air Traffic Control system）中引出一些例子—旨在描述一下视图的特定及其标记的方式，而不是定义这些系统的架构。

"4+1"视图模型具有相当的"普遍性"，因此可以使用其他的标注方法和工具，也可以采用其他的设计方法，特别是对于逻辑和过程的分解。但文中指出的这些方法都已经成功的在实践中运用过。

逻辑结构

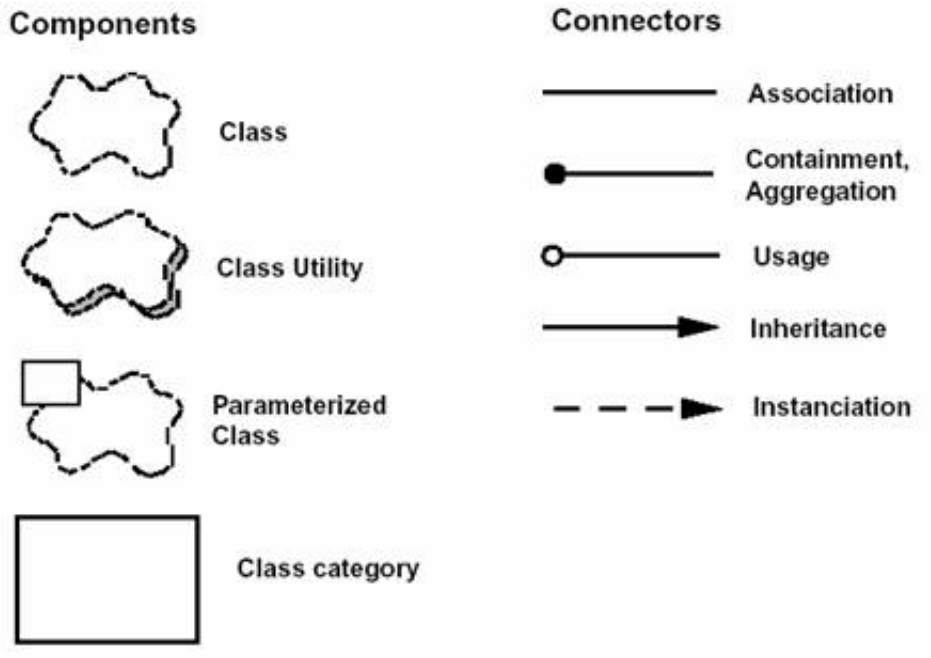
面向对象的分解

逻辑架构主要支持功能性需求——即在为用户提供服务方面系统所应该提供的功能。系统分解为一系列的关键抽象，（大多数）来自于问题域，表现为对象或对象类的形式。它们采用抽象、封装和继承的原理。分解并不仅仅是为了功能分析，而且用来识别遍布系统各个部分的通用机制和设计元素。我们使用 Rational/Booch 方法来表示逻辑架构，借助于类图和类模板的手段 4。类图用来显示一个类的集合和它们的逻辑关系：关联、使用、组合、继承等等。相似的类可以划分成类集合。类模板关注于单个类，它们强调主要的类操作，并且识别关键的对象特征。如果需要定义对象的内部行为，则使用状态转换图或状态图来完成。公共机制或服务可以在类功能 （class utilities）中定义。对于数据驱动程度高的应用程序，可以使用其他形式的逻辑视图，例如 E-R 图，来代替面向对象的方法（OO approach）。

逻辑视图的表示法

逻辑视图的标记方法来自 Booch 标记法4。当仅考虑具有架构意义的条目时，这种表示法相当简单。特别是在这种设计级别上，大量的修饰作用不大。我们使用 Rational Rose? 来支持逻辑架构的设计。

图 2 – 逻辑蓝图的表示法



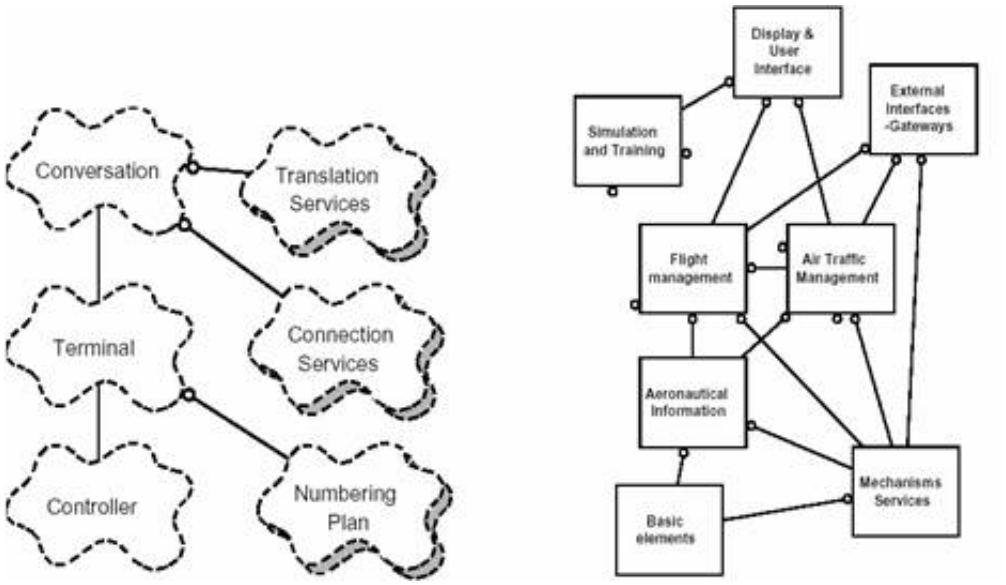
逻辑视图的风格

逻辑视图的风格采用面向对象的风格，其主要的设计准则是试图在整个系统中保持单一的、一致的对象模型，避免就每个场合或过程产生草率的类和机制的技术说明。

逻辑结构蓝图的样例

图 3 显示了 Télic PABX 架构中主要的类。

图 3 – a. Télic PABX 的逻辑蓝图 b.空中交通系统的蓝图



PABX 建立终端间的通信连接。终端可以是电话设备、中继线（例如，连接到中央办公室）、连接线（PABX 专线到 PABX 线)、电话专线、数据线、ISDN 等等。不同的线路由不同的接口卡提供支持。线路 controller 对象的职责是在接口卡上对所有的信号进行解码和注入，在特定于接口卡的信号与一致性的小型事件集合之间进行相互转换：开始、停止、数字化等。controller 对象同时承载所有的实时约束。该类派生出许多子类以满足不同的接口类型。terminal 对象的责任是维持终端的状态，代表线路协调各项服务。例如，它使用 numbering plan 服务来解释拨号。conversation 代表了会话中的一系列终端 。conversation 使用了 Translation Service(目录、逻辑物理映射、路由),以及建立终端之间语音路径的Connection Service 。

对于一个包含了大量的具有架构重要意义的类的、更大的系统来说，图 3 b 描述了空中交通管理系统的顶层类图，包含 8 个类的种类（例如，类的分组）。

进程架构

过程分解

进程架构考虑一些非功能性的需求，如性能和可用性。它解决并发性、分布性、系统完整性、容错性的问题，以及逻辑视图的主要抽象如何与进程结构相配合在一起－即在哪个控制线程上，对象的操作被实际执行。

进程架构可以在几种层次的抽象上进行描述，每个层次针对不同的问题。在最高的层次上，进程架构可以视为一组独立执行的通信程序（叫作"processes"）的逻辑网络，它们分布在整个一组硬件资源上，这些资源通过 LAN 或者 WAN 连接起来。多个逻辑网络可能同时并存，共享相同的物理资源。例如，独立的逻辑网络可能用于支持离线系统与在线系统的分离，或者支持软件的模拟版本和测试版本的共存。

进程是构成可执行单元任务的分组。进程代表了可以进行策略控制过程架构的层次（即：开始、恢复、重新配置及关闭）。另外，进程可以就处理负载的分布式增强或可用性的提高而不断地被重复。

软件被划分为一系列单独的任务。任务是独立的控制线程，可以在处理节点上单独地被调度。

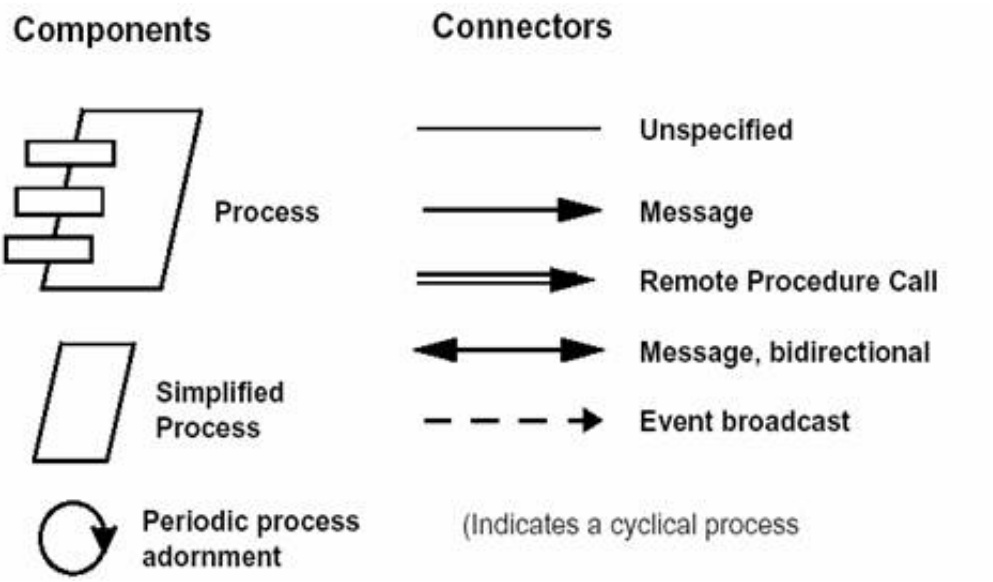
接着，我们可以区别主要任务、次要任务。主要任务是可以唯一处理的架构元素；次要任务是由于实施原因而引入的局部附加任务（周期性活动、缓冲、暂停等等）。它们可以作为 Ada Task 或轻量线程来实施。 主要任务的通讯途径是良好定义的交互任务通信机制：基于消息的同步或异步通信服务、远程过程调用、事件广播等。次要任务则以会见或共享内存来通信。在同一过程或处理节点上，主要任务不应对它们的分配做出任何假定。

消息流、过程负载可以基于过程蓝图来进行评估，同样可以使用哑负载来实现"中空"的进程架构，并测量在目标系统上的性能。正如 Filarey et al. 在他的 Eurocontrol 实验中描述的那样。

进程视图的表示法

我们所使用的进程视图的表示方法是从Booch最初为 Ada 任务推荐的表示方法扩展而来。同样，用来所使用的表示法关注在架构上具有重要意义的元素。(图 4)

图 4 – 过程蓝图表示法



我们曾使用来自 TRW 的 Universal Network Architechure

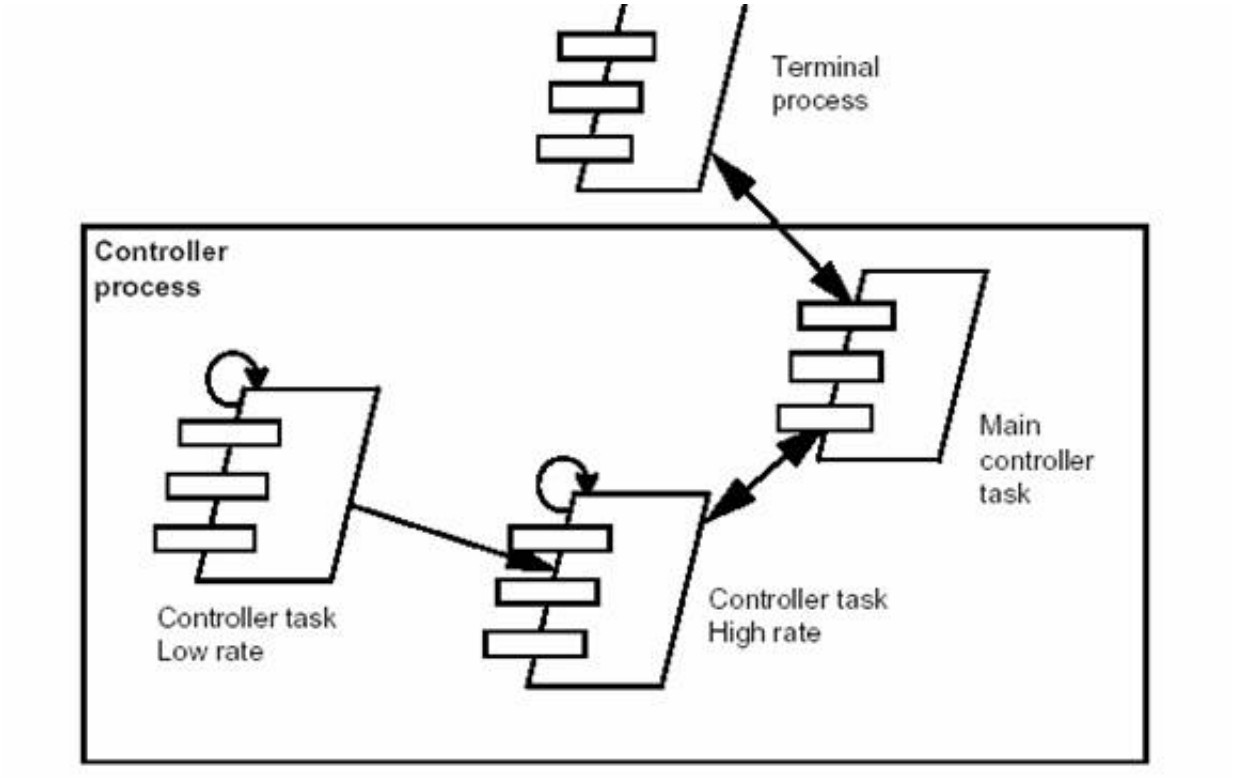
Services（UNAS0） 产品来构建并实施过程和任务集合（包扩它们的冗余），使它们融入过程的网络中。UNAS 包含 Software Architect Lifecycle Environment(SALE)工具，它支持上述表示方法。SALE 允许以图形的形式来描述进程架构，包括对可能的交互任务通信路径的规格说明，正是从这些路径中自动生成对应的 Ada 或 C++ 源代码。使用该方法来指定和实施进程架构的优点是易于进行修改而不会对应用软件造成太多的影响。

进程视图的风格

许多风格可以适用于进程视图。例如采用 Garlan 和 Shaw 的分类法1,我们可以得到管道和过滤器（Pipes and filters）,或客户端/服务器，以及各种多个客户端/单个服务器和多个客户端/多个服务器的变体。对于更加复杂的系统,可以采用类似于 K.Birman 所描述的ISIS系统中进程组方法以及其它的标注方法和工具。

进程蓝图的例子

图 5 – Télic PABX 的过程蓝图（部分）



所有的终端由单个的 Termal process 处理，其中 Termal process 由输入队列中的消息进行驱动。Controller 对象在组成控制过程三个任务之中的一项任务上执行：Low cycle rate task 扫描所有的非活动终端(200 ms)，将 High cycle rate task(10 ms)扫描清单中的终端激活，其中 High cycle rate task 检测任何重要的状态变化，将它们传递给 Main controller task，由它来对状态的变更进行解释,并通过向对应的终端发送消息来通信。这里 Controller 过程中的通信通过共享内存来实现。

开发架构

子系统分解

开发架构关注软件开发环境下实际模块的组织。软件打包成小的程序块（程序库或子系统），它们可以由一位或几位开发人员来开发。子系统可以组织成分层结构，每个层为上一层提供良好定义的接口。

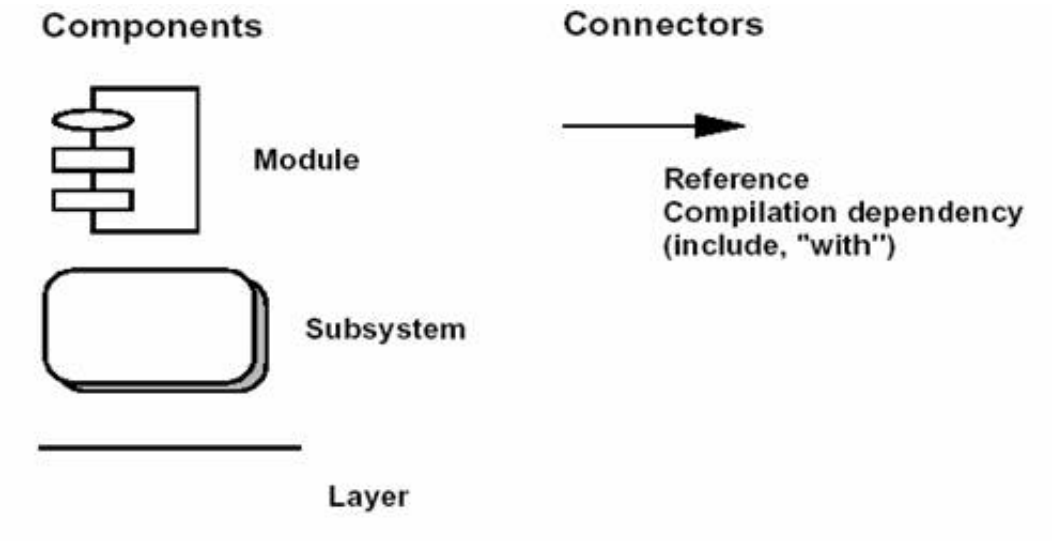
系统的开发架构用模块和子系统图来表达，显示了"输出"和"输入"关系。完整的开发架构只有当所有软件元素被识别后才能加以描述。但是，可以列出控制开发架构的规则：分块、分组和可见性。

大部分情况下，开发架构考虑的内部需求与以下几项因素有关：开发难度、软件管理、重用性和通用性及由工具集、编程语言所带来的限制。开发架构视图是各种活动的基础，如：需求分配、团队工作的分配（或团队机构）、成本评估和计划、项目进度的监控、软件重用性、移植性和安全性。它是建立产品线的基础。

开发蓝图的表示方法

同样，使用 Booch 方法的变形，仅考虑具有架构意义的项。

图 5 – 开发蓝图表示方法

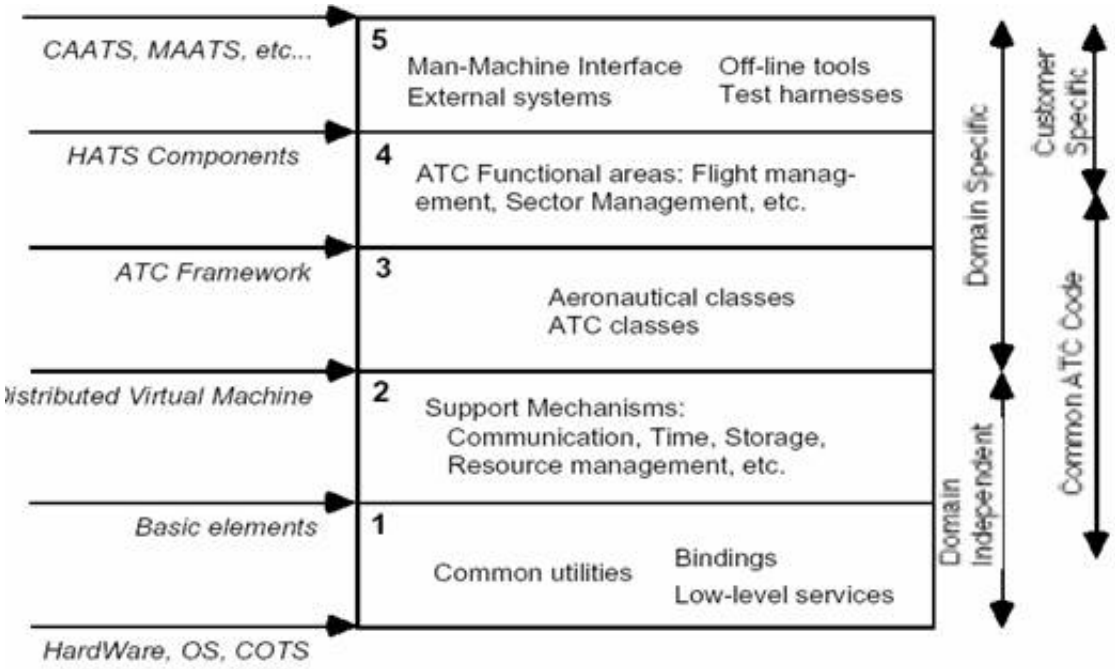


来自 Rational 的 Apex 开发环境支持开发架构的定义和实现、和前文描述的分层策略，以及设计规则的实施。Rational Rose 可以在模块和子系统层次上绘制开发蓝图，并支持开发源代码(Ada、C++）进程的正向和反向工程。

开发视图的风格

我们推荐使用分层（layered）的风格，定义 4 到 6 个子系统层。每层均具有良好定义的职责。设计规则是某层子系统依赖同一层或低一层的子系统，从而最大程度地减少了具有复杂模块依赖关系的网络的开发量，得到层次式的简单策略。

图 6 – Hughes 空中交通系统（HATS）的 5 个层



开发架构的例子

图 6 代表了加拿大的 Hughes Aircraft 开发的空中交通控制系统（Air Traffic Control system）产品线的 5 个分层开发组织结构。这是和图 3 b 描述的逻辑架构相对应的开发架构。

第一层 和第二层组成了独立于域的覆盖整个产品线的分布式基础设施，并保护其免受不同硬件平台、操作系统或市售产品（如数据库管理系统）的影响。第三层为该基础设施增加了 ATC 框架，形成一个特定领域的软件架构（domain-specific software architecture）。使用该框架,可以在第四层上构建一个功能选择板。层次 5 则非常依赖于客户和产品,包含了大多数用户接口和外部系统接口。72 个子系统分布于 5 个层次上，每层包含了 10 至 50 个模块，并可以在其他蓝图上表示。

物理架构

软件至硬件的映射

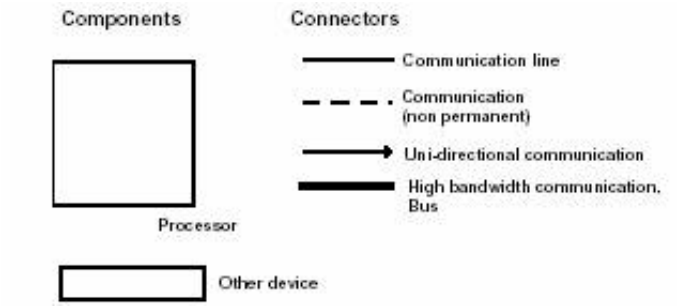
物理架构主要关注系统非功能性的需求，如可用性、可靠性（容错性），性能（吞吐量）和可伸缩性。软件在计算机网络或处理节点上运行，被识别的各种元素（网络、过程、任务和对象），需要被映射至不同的节点；我们希望使用不同的物理配置：一些用于开发和测试，另外一些则用于不

同地点和不同客户的部署。因此软件至节点的映射需要高度的灵活性及对源代码产生最小的影响。

物理蓝图的表示法

大型系统中的物理蓝图会变得非常混乱，所以它们可以采用多种形式，有或者没有来自进程视图的映射均可。

图 7 – 物理蓝图的表示法



TRW 的 UNAS 提供了数据驱动方法将过程架构映射至物理架构，该方法允许大量的映射 的变更而无需修改源代码。

物理蓝图的示例

图 8 – PABX 的物理蓝图

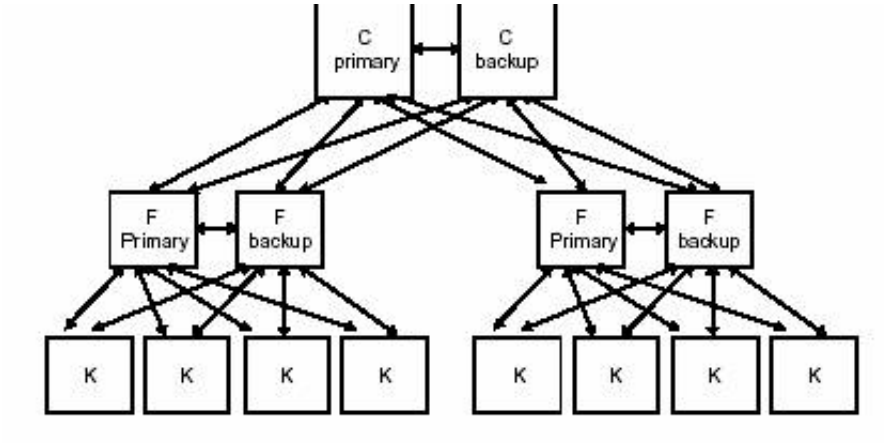


图 8 显示了大型 PABX 可能的硬件配置,而图 9 和图 10 显示了两种不同物理架构上的进程映射，分别对应一个小型和一个大型 PABX。C、F 和 K 是三种不同容量的计算机，支持三种不同的运行要求。

图 9 – 带有过程分配的小型 PABX 物理架构

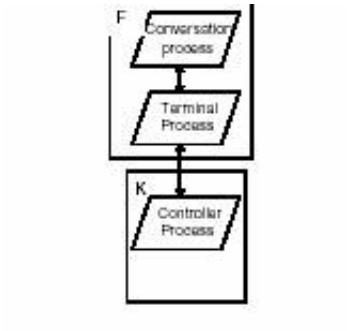
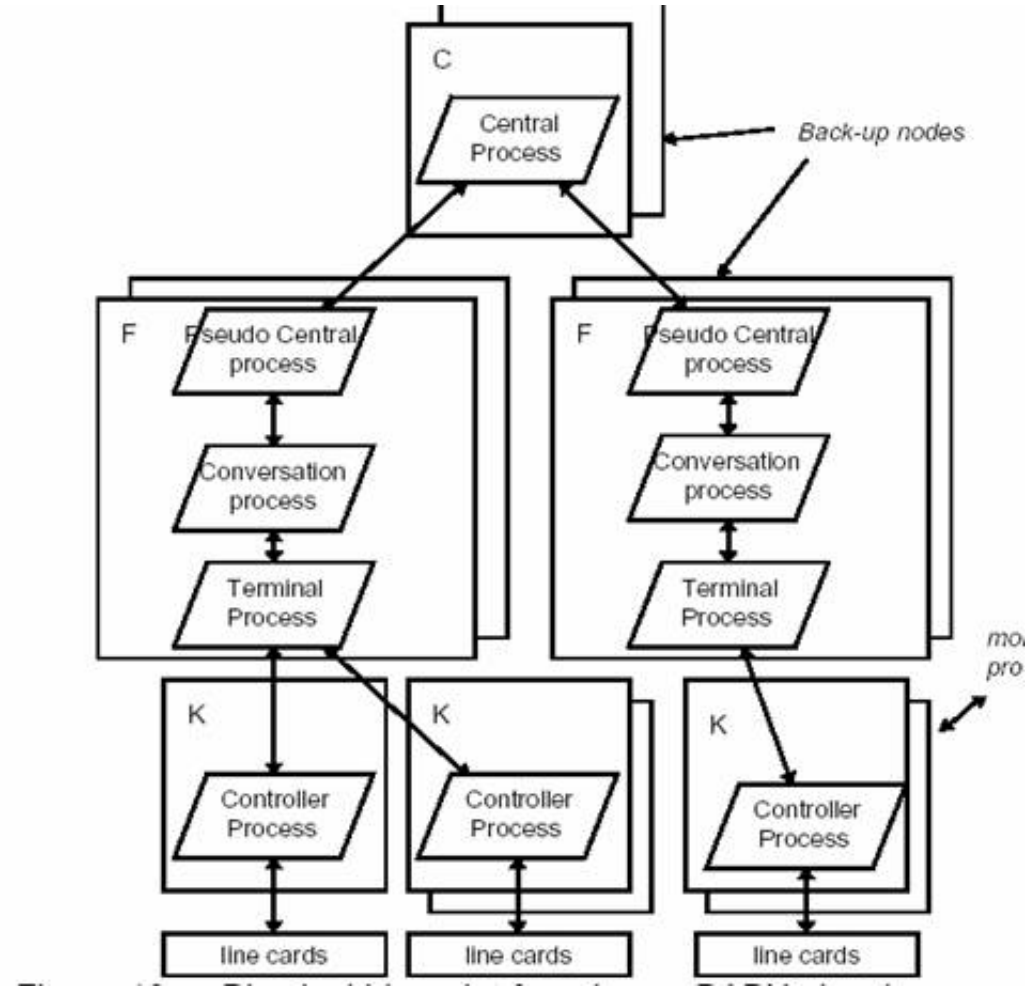


图10 – 显示了过程分配的大型PABX物理蓝图



场景

综合所有的视图

四种视图的元素通过数量比较少的一组重要场景（更常见的是用例）进行无缝协同工作，我们为场景描述相应的脚本（对象之间和过程之间的交互

序列）。正如 Rubin 和 Goldberg 所描述的那样6。

在某种意义上场景是最重要的需求抽象，它们的设计使用对象场景图和对象交互图来表示4。

该视图是其他视图的冗余（因此"＋1"），但它起到了两个作用：

- 作为一项驱动因素来发现架构设计过程中的架构元素，这一点将在下文中讨论。
- 作为架构设计结束后的一项验证和说明功能，既以视图的角度来说明又作为架构原型测试的出发点。

场景的表示法

场景表示法与组件逻辑视图非常相似（请对照图 2），但它使用过程视图的连接符来表示对象之间的交互（请对照图 4），注意对象实例使用实线来表达。至于逻辑蓝图，我们使用 Rational Rose 来捕获并管理对象场景。

场景的例子

图 11 显示了小型 PABX 的场景片段。相应的脚本是：

1. Joe的电话控制器检测和校验摘机状态的变换,并发送消息唤醒相应的终端对象。

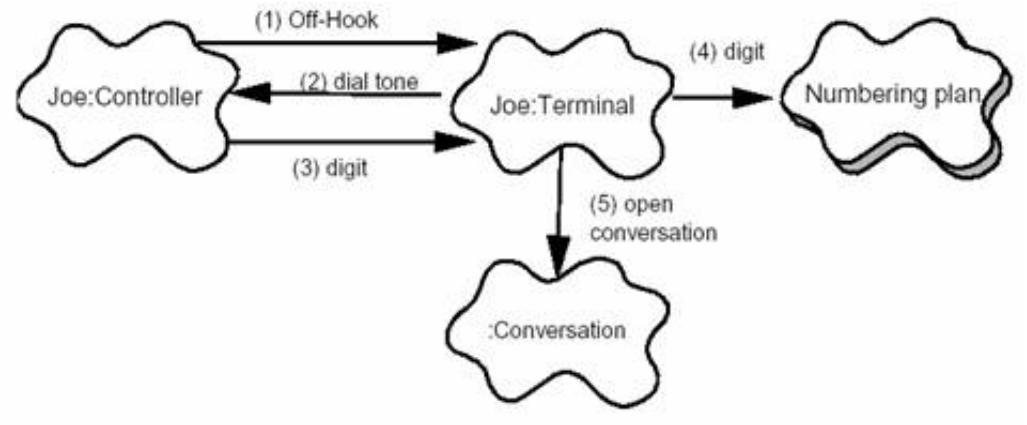
2. 终端分配一些资源，并要求控制器发出拨号音。

3. 控制器接受拨号并传递给终端。

4. 终端使用拨号方案来分析数字流。

5. 有效的数字序列被键入,终端开始会话。

图 11 – 本地呼叫的初期场景——阶段选择



视图之间的对应性

各种视图并不是完全是正交的或独立的。视图的元素根据某种设计规则和启发式方法与其他视图中的元素相关联。

从逻辑视图到过程视图

我们发现逻辑视架构有几项重要特性：

- 自主性：对象是主动的、被动的还是被保护的?
  - 主动对象享有调用其他对象或其自身操作的主动权，并且当其他对象对其进行调用时，具有对其自身操作的完全控制权。
  - 被动对象不能主动调用任何操作，对其他对象调用自身的操作没有控制。
  - 被保护对象不能主动调用任何操作。但对自身的操作有一定的控制功能。
- 持久化：对象是暂时的还是持久化的？它们是否会导致过程或处理器的终止?
- 依赖性：对象的存在或持久化是否依赖于另一个对象?
- 分布性：对象的状态或操作是否能被物理架构中的许多节点所访问?或是被进程架构中的几个进程所访问?

在逻辑视图中，我们认为每个对象均是主动的，具有潜在的"并发性"，即与其他对象具有"平行的"行为，我们并不考虑所要达到的确切并发程度。因此，逻辑结构所考虑的仅是需求的功能性方面。

然而，当我们定义进程架构时，由于巨大的开销，为每个对象实施各自的控制线程（例如，Unix 进程或 Ada 任务），在目前的技术状况下是不现实的。此外，如果对象是并发的，那么必须以某种抽象形式来调用它们的操作。

另一方面，由于以下几种原因需要多个控制线程。

- 为了快速响应某类外部触发，包括与时间相关的事件。
- 为了在一个节点中利用多个 CPU，或者在一个分布式系统中利用多个节点。
- 为了提高 CPU 的利用率，在某些控制线程被挂起，等待其他活动结束的时候（例如，访问外部对象其他活动对象时），为其他的活动分配 CPU。
- 为了划分活动的优先级（提高潜在的响应能力）。
- 为了支持系统的可伸缩性（借助于共享负载的其他过程）。
- 为了在软件的不同领域分离关注点。
- 为了提高系统的可用性（通过 Backup 过程）。

我们同时使用两种策略来决定并发的程度和定义所需的过程集合。考虑一系列潜在的物理目标架构。以下两种形式我们可以任选其一：

- 从内至外:

  由逻辑架构开始：定义代理任务，该任务将控制一个类的多个活动对象的单个线程进行多元化处理；同一代理任务还执行持久化处理那些依赖于一个主动对象的对象；需要相互进行操作的几个类或仅需要少量处理的类共享单个代理。这种聚合会一直进行，直到我们将过程减少到合理的较少数量，而仍允许分布性和对物理资源的使用。
- 由外至内:

  从物理结构开始：识别系统的外部触发；定义处理触发的客户过程和仅提供服务（而非初始化它们）的服务器进程；使用数据完整性和问题的串行化（serialization）约束来定义正确的服务器设置，并且为客户机与服务器代理分配对象；识别出必须分布哪些对象。

其结果是将类（和它们的对象）映射至一个任务集合和进程架构中的进程。通常，活动类具有代理任务，也存在一些变形：对于给定的类，使用多个代理以提高吞吐量，或者多个类映射至单个代理，因为它们的操作并不是频繁地被调用，或者是为了保证执行序列。

注意这并不是产生最佳过程架构的线性的、决定性的进程；它需要若干个迭代来得到可接受的折衷。还存在许多其他方法，例如 Birman 等人5 或 Witt 等人7提出的方法。 确切的实施映射的方法不在本文的讨论范围，但我们以一个小的例子来说明一下。

图 12 显示了一个小的类集合如何从假想的空中交通控制系统映射至进程。

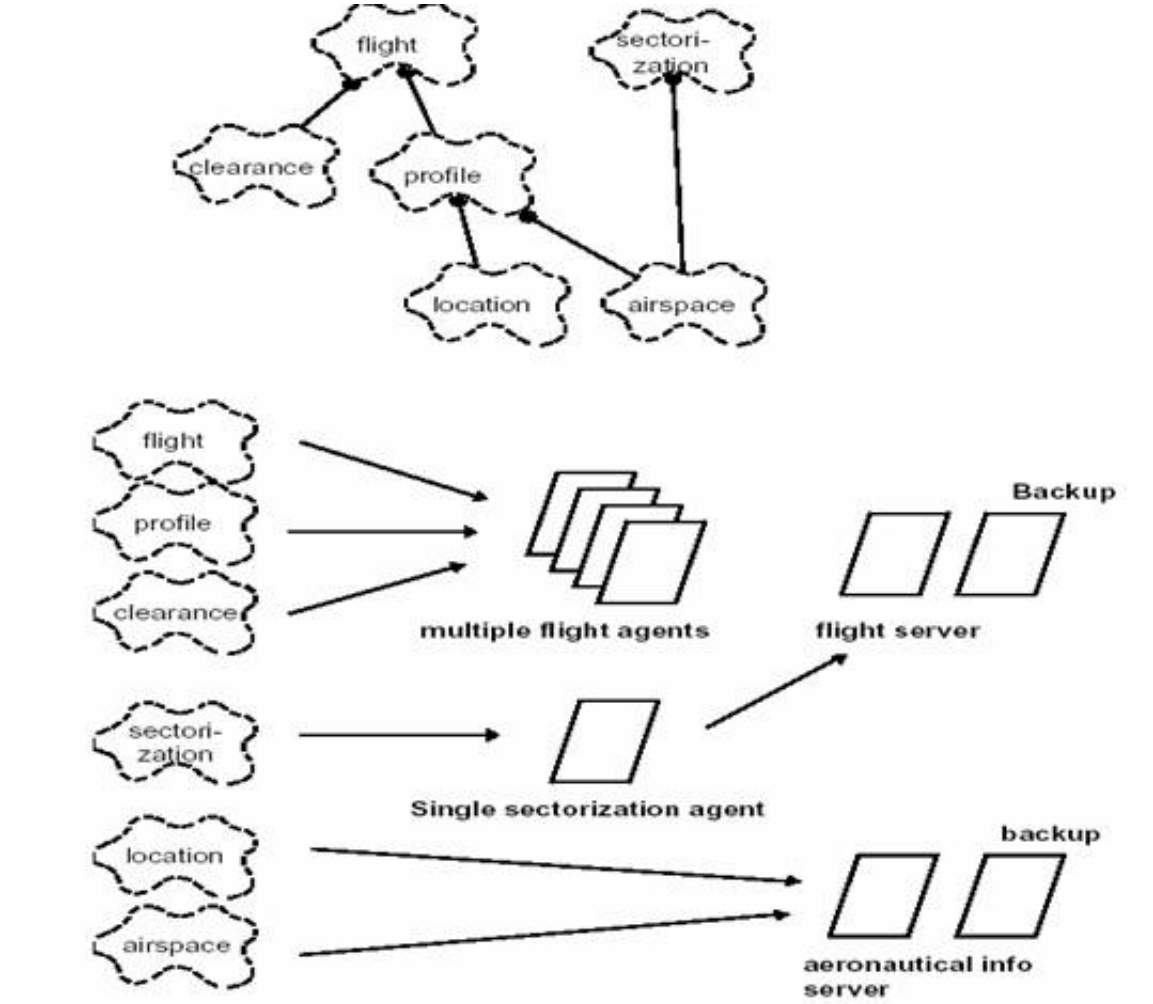
flight 类映射至一个 flight 代理集合：有许多航班等待处理，外部触发的频率很高，响应时间很关键，负载必须分布于多个 CPU。并且，航班处理的持久化和分布性方面都取决于 flight server，为了满足可用性，还是使用 flight server 的一台备份服务器。

航班的 profile 和 clearance 总是从属于某个航班,尽管它们是复杂的类,但它们共享 flight 类的进程。航班分布于若干其他进程，特别是对于显示和外部接口。

sectorization 类，为 controller 的权限分配建立了空域划分。由于完整性约束，仅能被一个代理处理，但可以与 flight 类共享服务器过程：更新得并不频繁。

location 和 arispace 及其他的静态航空信息是受到保护的对象，在几个类中共享，很少被更新；它们被映射至各自的服务器，分布于其他过程。

图 12 – 从逻辑视图到过程视图的映射



从逻辑视图到开发视图

类通常作为一个模块来实现，例如 Ada 包中可视部分的一个类型。密切相关的类（类的种类）的集合组合到子系统中。子系统的定义必须考虑额外的约束，如团队组织、期望的代码规模（通常每个子系统为 5 K 或 20 K SLOC）、可重用性和通用性的程度以及严格的分层依据（可视性问题），发布策略和配置管理。所以，通常最后得到的不是与逻辑视图逐一对应的视图。

逻辑视图和开发视图非常接近，但具有不同的关注点。我们发现项目规模越大，视图间的差距也越大。例如，如果比较图 3 b 和图 6，则会发现并不存在逐一对应的类的不同种类到层的映射。而如果我们考虑类的种类的"外部接口"－网关种类时，它的实现遍布若干层：通讯协议在第 1 层或以下的层，通用网关机制在第 2 层，而特定的网关在第 5 层子系统。

从进程视图到物理视图

进程和进程组以不同的测试和部署配置映射至可用的物理硬件。Birman 在 ISIS 项目中描述了详细的映射模式5。

场景主要以所使用类的形式与逻辑视图相关联；而与进程视图的关联则是考虑了一个或多个控制线程的、对象间的交互形式。

模型的剪裁

并不是所有的软件架构都需要"4＋1"视图。无用的视图可以从架构描述中省略，比如： 只有一个处理器，则可以省略物理视图；而如果仅有一个进程或程序，则可以省略过程视图。 对于非常小型的系统，甚至可能逻辑视图与开发视图非常相似，而不需要分开的描述。场景对于所有的情况均适用。

迭代过程

Witt 等人为设计和架构指出了 4 个阶段：勾画草图、组织、具体化和优化，分成了 12 个 步骤7。他们还指出需要某种程度的反向工程。而我们认为对于大型的项目，该方法太"线性 化"了。在 4 个阶段的末尾，可用于验证架构的内容太少。我们提倡一种更具有迭代性质的方法，即架构先被原形化、测试、估量、分析，然后在一系列的迭代过程中被细化。该方法除了减少与架构相关的风险之外，对于项目而言还有其他优点：团队合作、培训，加深对架构的理解，深入程序和工具等等（此处提及的是演进的原形，逐渐发展成为系统，而不是一次性的试验性的原形）。这种迭代方法还能够使需求被细化、成熟化并能够被更好地理解。

场景驱动（scenario-driven）的方法

系统大多数关键的功能以场景（或 use cases）的形式被捕获。关键意味着：最重要的功能，系统存在的理由，或使用频率最高的功能，或体现了必须减轻的一些重要的技术风险。

开始阶段:

- 基于风险和重要性为某次迭代选择一些场景。场景可能被归纳为对若干用户需求的抽象。
- 形成"稻草人式的架构"。然后对场景进行"描述"，以识别主要的抽象（类、机制、过程、子系统），如 Rubin 与 Goldberg6 所指出的 —— 分解成为序列对（对象、操作）。
- 所发现的架构元素被分布到 4 个蓝图中：逻辑蓝图、进程蓝图、开发蓝图和物理蓝图。
- 然后实施、测试、度量该架构，这项分析可能检测到一些缺点或潜在的增强要求。
- 捕获经验教训。

循环阶段:

下一个迭代过程开始进行:

- 重新评估风险,
- 扩展考虑的场景选择板。
- 选择能减轻风险或提高结构覆盖的额外的少量场景,

然后:

- 试着在原先的架构中描述这些场景。
- 发现额外的架构元素，或有时还需要找出适应这些场景所需的重要架构变更。
- 更新4个主要视图：逻辑视图、进程视图、开发视图和物理视图。
- 根据变更修订现有的场景。
- 升级实现工具（架构原型）来支持新的、扩展了的场景集合。
- 测试。如果可能的话，在实际的目标环境和负载下进行测试。
- 然后评审这五个视图来检测简洁性、可重用性和通用性的潜在问题。
- 更新设计准则和基本原理。
- 捕获经验教训。

终止循环

为了实际的系统，初始的架构原型需要进行演进 。较好的情况是在经过 2 次或 3 次迭代之后，结构变得稳定：主要的抽象都已被找到。子系统和过程都已经完成，以及所有的接口都已经实现。接下来则是软件设计的范畴，这个阶段可能也会用到相似的方法和过程。

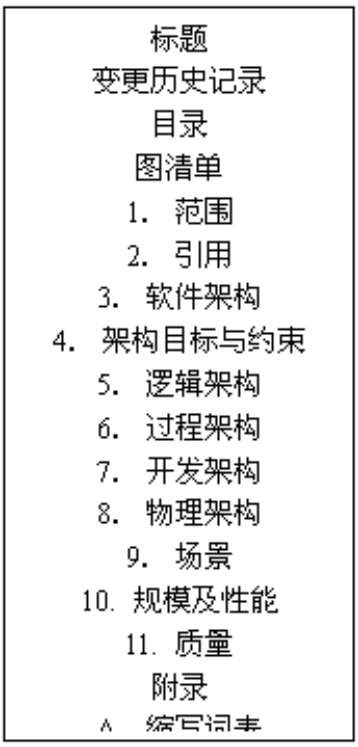
这些迭代过程的持续时间参差不齐，原因在于：所实施项目的规模，参与项目人员的数量、他们对本领域和方法的熟悉程度，以及该系统和开发组织的熟悉程度等等。因而较小的项目迭代过程可能持续 2-3 周（例如，10 K SLOC），而大型的项目可能为 6-9 个月（例如，700 K SLOC）。

架构的文档化

架构设计中产生的文档可以归结为两种：

- 软件架构文档，其结构遵循"4+1"视图（请对照图 13，一个典型的提纲）
- 软件设计准则，捕获了最重要的设计决策。这些决策必须被遵守，以保持系统架构的完整性。

图 13 – 软件架构文档提纲



结束语

无论是否经过一次本地定制的和技术上的调整，"4＋1"视图都能在许多大型项目中成功运用。事实上，它允许不同的"风险承担人"找出他们就软件架构所关心的问题。系统工程师首先接触物理视图，然后转向进程视图；最终用户、顾客、数据分析专家从逻辑视图入手；项目经理、软件配置人员则从开发视图来看待"4＋1"视图。 在 Rational 和其他地方，提出并讨论了其他系列视图，例如 Meszaros(BNR)、Hofmeister。Nord 和 Soni(Siemenms)、Emery 和 Hilliard（Mitre），但我们发现其他视图通常可以归入我们所描述的 4 个视图中的一个。例如 Cost&Schedule 视图可以归入开发视图，将一个数据视图归入一个逻辑视图，以及将一个执行视图归入进程视图和物理视图的组合。

表 1 － "4＋1"视图模型一览表

视图	逻辑视图	过程视图	开发视图	物理视图	场景
组件	类	任务	模块、子系统	节点	步骤、脚本
连接工具	关联、继承、约束	会面、消息、广播、RPC 等	编译依赖性、“with”语句、“include”	通信媒体、LAN、WAN、总线等	
容器	类的种类	过程	子系统（库）	物理子系统	Web
涉众	最终用户	系统设计人员、集成人员	开发人员、经理	系统设计人员	最终用户、开发人员
关注点	功能	性能、可用性、S/W 容错、整体性	组织、可重用性、可移植性、产品线	可伸缩性、性能、可用性	可理解性
工具支持	Rose	UNAS/SALE DADS	Apex、SoDA	UNAS、Openview DADS	Rose

致谢

"4+1" 视图的诞生要归功于在Rational、加拿大的 Hughes Aircraft、Alcatel 以及其他地方工作的同事。笔者特别感谢下面这些人的贡献： Ch. Thompson、A. Bell、M.Devlin、G. Booch、W. Royce、J. Marasco、R. Reitman、V. Ohnjec、E. Schonberg。