兰州交通大学毕业设计

英文文献及翻译

学 院： 电子与信息工程学院

专 业： 软 件 工 程

姓 名： 张 翔

学 号： 201211019

指导教师： 金 静

二Ο一六年六月

# Abstract

This article presents a model for describing the architecture of software-intensive systems, based on the use of multiple, concurrent views. This use of multiple views allows to address separately the concerns of the various ‘stakeholders’ of the architecture: end-user, developers, systems engineers, project managers, etc., and to handle separately the functional and non-functional requirements. Each of the five views is described, together with a notation to capture it. The views are designed using an architecture entered, scenario driven, and iterative development process.

Keywords: software architecture, view, object-oriented design, software development process

# Introduction

We all have seen many books and articles where one diagram attempts to capture the gist of the architecture of a system. But looking carefully at the set of boxes and arrows shown on these diagrams, it becomes clear that their authors have struggled hard to represent more on one blueprint than it can actually express. Are the boxes representing running programs? Or chunks of source code? Or physical computers? Or merely logical groupings of functionality? Are the arrows representing compilation dependencies? Or control flows? Or data flows? Usually it is a bit of everything. Does an architecture need a single architectural style? Sometimes the architecture of the software suffers scars from a system design that went too far into prematurely partitioning the software, or from an over-emphasis on one aspect of software development: data engineering, or run-time efficiency, or development strategy and team organization. Often also the architecture does not address the concerns of all its “customers” (or “stakeholders” as they are called at USC). This problem has been noted by several authors: Garland & Shaw1, Abowd & Allen at CMU, Clements at the SEI. As a remedy, we propose to organize the description of a software architecture using several concurrent views, each one addressing one specific set of concerns.

# An Architectural Model

Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability. Perry and Wolfe put it very nicely in this formula, modified by Boehm:

**Software architecture = {Elements, Forms, Rationale/Constraints}**

Software architecture deals with abstraction, with decomposition and composition, with style and aesthetics. To describe a software architecture, we use a model composed of multiple views or perspectives. In order to eventually address large and challenging architectures, the model we propose is made up of five main views (cf. fig. 1):

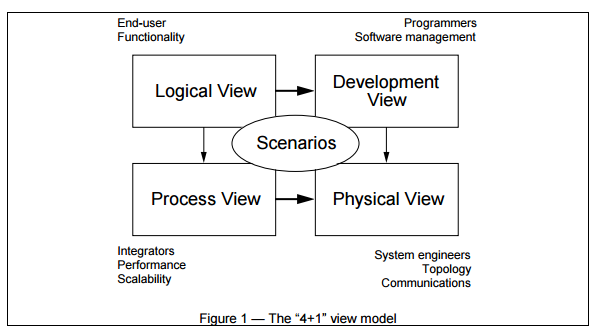
• The logical view, which is the object model of the design (when an object-oriented design method is used),

• the process view, which captures the concurrency and synchronization aspects of the design,

• the physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect,

• the development view, which describes the static organization of the software in its development environment.

The description of an architecture—the decisions made—can be organized around these four views, and then illustrated by a few selected use cases, or scenarios which become a fifth view. The architecture is in fact partially evolved from these scenarios as we will see later.



We apply Perry & Wolf’s equation independently on each view, i.e., for each view we define the set of elements to use (components, containers, and connectors) , we capture the forms and patterns that work, and we capture the rationale and constraints, connecting the architecture to some of the requirements. Each view is described by a blueprint using its own particular notation. For each view also, the architects can pick a certain architectural style, hence allowing the coexistence of multiple styles in one system.

We will now look in turn at each of the five views, giving for each its purpose: which concerns is addresses, a notation for the corresponding architectural blueprint, the tools we have used to describe and manage it. Small examples are drawn from the design of a PABX, derived from our work at Alcatel Business System and an Air Traffic Control system, but in very simplified form—the intent here is just to give a flavor of the views and their notation and not to define the architecture of those systems.

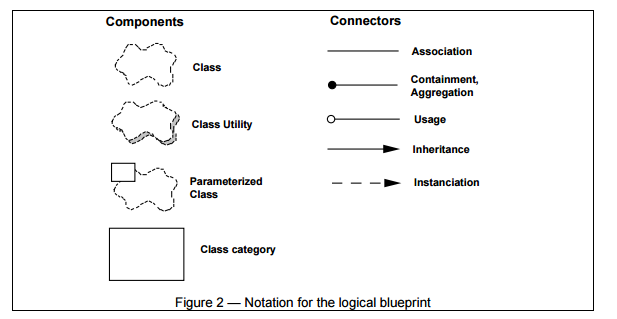
The “4+1” view model is rather “generic”: other notations and tools can be used, other design methods can be used, especially for the and the logical and process decompositions, but we have indicated the ones we have used with success.

# The Logical Architecture

The logical architecture primarily supports the functional requirements—what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes. They exploit the principles of abstraction, encapsulation, and inheritance. This decomposition is not only for the sake of functional analysis, but also serves to identify common mechanisms and design elements across the various parts of the system. We use the Rational/Booch approach for representing the logical architecture, by means of class diagrams and class templates. A class diagram shows a set of classes and their logical relationships: association, usage, composition, inheritance, and so forth. Sets of related classes can be grouped into class categories. Class templates focus on each individual class; they emphasize the main class operations, and identify key object characteristics. If it is important to define the internal behavior of an object, this is done with state transition diagrams, or state charts. Common mechanisms or services are defined in class utilities. Alternatively to an OO approach, an application that is very data-driven may use some other form of logical view, such as E-R diagrams.

## 2.1 Notation for the logical view

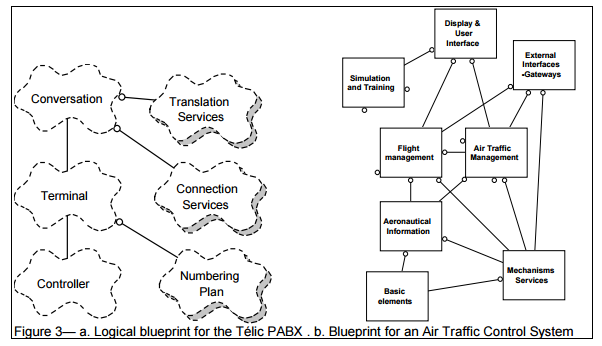
The notation for the logical view is derived from the Booch notation. It is considerably simplified to take into account only the items that are architecturally significant. In particular, the numerous adornments are not very useful at this level of design. We use Rational Rose® to support the logical architecture design.



## 2.2 Style for the logical view

The style we use for the logical view is an object-oriented style. The main guideline for the design of the logical view is to try to keep a single, coherent object model across the whole system, to avoid premature specialization of classes and mechanisms per site or per processor.

Examples of Logical blueprints Figure 3a shows the main classes involved in the Telic PABX architecture.



A PABX establishes commmunications between terminals. A terminal may be a telephone set, a trunk line (i.e., line to central-office), a tie line (i.e., private PABX to PABX line), a feature phone line, a data line, an ISDN line, etc. Different lines are supported by different line interface cards. The responsibility of a line controller object is to decode and inject all the signals on the line interface card, translating card-specific signals to and from a small, uniform set of events: start, stop, digit, etc. The controller also bears all the hard real-time constraints. This class has many subclasses to cater for different kinds of interfaces. The responsibility of the terminal object is to maintain the state of a terminal, and negotiate services on behalf of that line. For example, it uses the services of the numbering plan to interpret the dialing in the selection phase. The conversation represents a set of terminals engaged in a conversation. The conversation uses translation services (directory, logical to physical address mapping, routes), and connection services to establish a voice path between the terminals.

For a much bigger system, which contains a few dozen classes of architectural significance, figure 3b show the top level class diagram of an air traffic control system, containing 8 class categories (i.e., groups of classes).

# The Process Architecture

The process architecture takes into account some non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, of system’s integrity, of fault-tolerance, and how the main abstractions from the logical view fit within the process architecture—on which thread of control is an operation for an object actually executed.

The process architecture can be described at several levels of abstraction, each level addressing different concerns. At the highest level, the process architecture can be viewed as a set of independently executing logical networks of communicating programs (called “processes”), distributed across a set of hardware resources connected by a LAN or a WAN. Multiple logical networks may exist simultaneously, sharing the same physical resources. For example, independent logical networks may be used to support separation of the on-line operational system from the off-line system, as well as supporting the coexistence of simulation or test versions of the software.

A process is a grouping of tasks that form an executable unit. Processes represent the level at which the process architecture can be tactically controlled (i.e., started, recovered, reconfigured, and shut down). In addition, processes can be replicated for increased distribution of the processing load, or for improved availability.

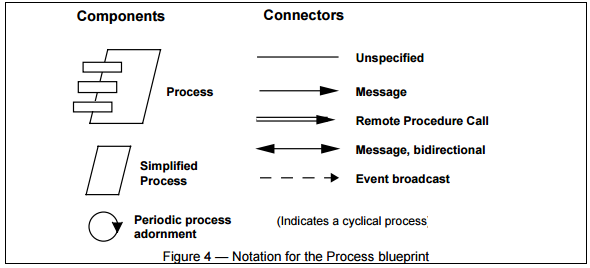
The software is partitioned into a set of independent tasks. A task is a separate thread of control, that can be scheduled individually on one processing node.

We can distinguish then: major tasks, that are the architectural elements that can be uniquely addressed and minor tasks, that are additional tasks introduced locally for implementation reasons (cyclical activities, buffering, time-outs, etc.). They can be implemented as Ada tasks for example, or light-weight threads. Major tasks communicate via a set of well-defined inter-task communication mechanisms: synchronous and asynchronous message-based communication services, remote procedure calls, event broadcast, etc. Minor tasks may communicate by rendezvous or shared memory. Major tasks shall not make assumptions about their collocation in the same process or processing node.

Flow of messages, process loads can be estimated based on the process blueprint. It is also possible to implement a “hollow” process architecture with dummy loads for the processes, and measure its performance on the target system, as described by Filarey et al. in their Eurocontrol experiment.

## 3.1 Notation for the Process view

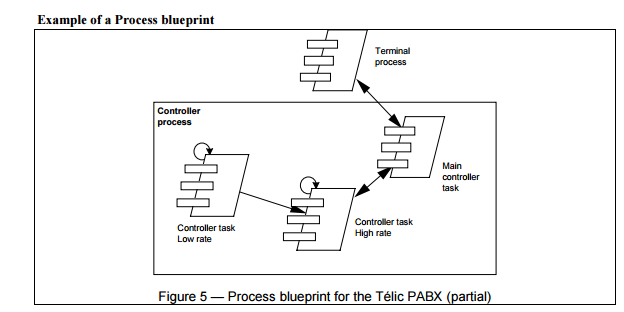
The notation we use for the process view is expanded from the notation originally proposed by Booch for Ada tasking. Again the notation used focuses on the elements that are architecturally significant. (Fig. 4)



We have used the Universal Network Architecture Services (UNAS) product from TRW to architect and implement the set of processes and tasks (and their redundancies) into networks of processes. UNAS contains a tool—the Software Architects Lifecycle Environment (SALE)—which supports such a notation. SALE allows for the graphical depiction of the process architecture, including specifications of the possible inter-task communication paths, from which the corresponding Ada or C++ source code is automatically generated. The benefit of this approach to specifying and implementing the process architecture is that changes can be incorporated easily without much impact on the application software.

## 3.2 Style for the process view

Several styles would fit the process view. For example, picking from Garlan and Shaw’s taxonomy we can have: pipes and filters, or client/server, with variants of multiple client/single server and multiple clients/multiple servers. For more complex systems, one could use a style similar to the process groups approach of the ISIS system as described by K. Birman with another notation and toolset.



All terminals are handled by a single terminal process, which is driven by messages in its input queues. The controller objects are executed on one of three tasks that composes the controller process: a low cycle rate task scans all inactive terminals (200 ms), puts any terminal becoming active in the scan list of the high cycle rate task (10ms), which detects any significant change of state, and passes them to the main controller task which interprets the changes and communicates them by message to the corresponding terminal. Here message passing within the controller process is done via shared memory.

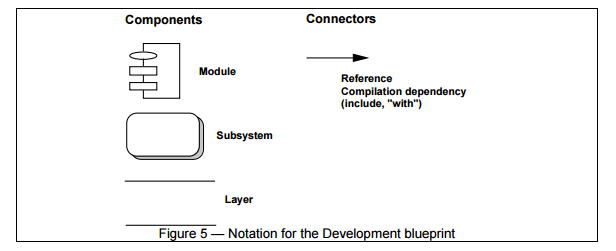
# The Development Architecture

The development architecture focuses on the actual software module organization on the software development environment. The software is packaged in small chunks—program libraries, or subsystems— that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

The development architecture of the system is represented by module and subsystem diagrams, showing the ‘export’ and ‘import’ relationships. The complete development architecture can only be described when all the elements of the software have been identified. It is, however, possible to list the rules that govern the development architecture: partitioning, grouping, visibility.

For the most part, the development architecture takes into account internal requirements related to the ease of development, software management, reuse or commonality, and to the constraints imposed by the toolset, or the programming language. The development view serves as the basis for requirement allocation, for allocation of work to teams (or even for team organization), for cost evaluation and planning, for monitoring the progress of the project, for reasoning about software reuse, portability and security. It is the basis for establishing a line-of-product.

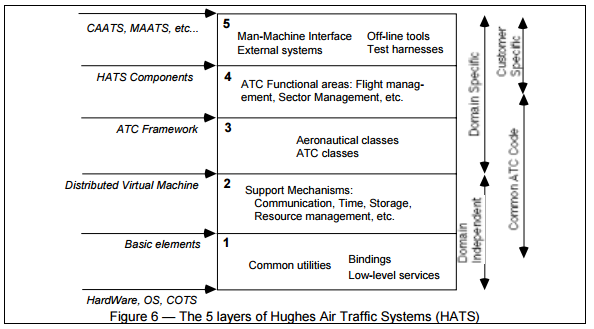
Again, a variation of the Booch notation, limiting it to the items that are architecturally significant.



The Apex Development Environment from Rational supports the definition and the implementation of the development architecture, the layering strategy described above, and the enforcement of the design rules. Rational Rose can draw the development blueprints at the module and subsystem level, in forward engineering and by reverse engineering from the development source code, for Ada and C++.

## 4.1 Style for the Development View

We recommend adopting a layered style for the development view, defining some 4 to 6 layers of subsystems. Each layer has a well-defined responsibility. The design rule is that a subsystem in a certain can only depend on subsystem that are in the same layer or in layers below, in order to minimize the development of very complex networks of dependencies between modules and allow simple release strategies layer by layer.



Example of Development architecture

Figure 6 represents the development organization in five layers of a line-of-product of Air Traffic Control systems developed by Hughes Aircraft of Canada. This is the development architecture corresponding to the logical architecture shown in fig. 3b.

Layers 1 and 2 constitute a domain-independent distributed infrastructure that is common across the line of products and shields it from variations in hardware platform, operating system, or off-the-shelf products such as database management system. To this infrastructure, layer 3 adds an ATC framework to form a domain-specific software architecture. Using this framework a palette of functionality is build in layer 4. Layer 5 is very customer- and product-dependent, and contains most of the user-interface and interfaces with the external systems. Some subsystems are spread across of the 5 layers, containing each from 10 to 50 modules, and can be represented on additional blueprints.

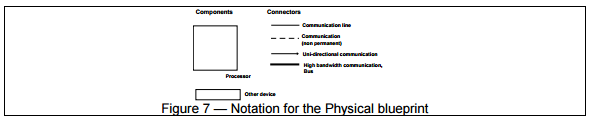
The Physical Architecture

Mapping the software to the hardware

The physical architecture takes into account primarily the non-functional requirements of the system such as availability, reliability (fault-tolerance), performance (throughput), and scalability. The software executes on a network of computers, or processing nodes (or just nodes for short). The various elements identified— networks, processes, tasks, and objects—need to be mapped onto the various nodes. We expect that several different physical configurations will be used: some for development and testing, others for the deployment of the system for various sites or for different customers. The mapping of the software to the nodes therefore needs to be highly flexible and have a minimal impact on the source code itself.

Notation for the Physical

Blueprint Physical blueprints can become very messy in large systems, so they take several forms, with or without the mapping from the process view.



UNAS from TRW provide us here with data-driven means of mapping the process architecture onto the physical architecture allowing a large class of changes in the mapping without source code modifications.

## 4.2 Example of Physical blueprint

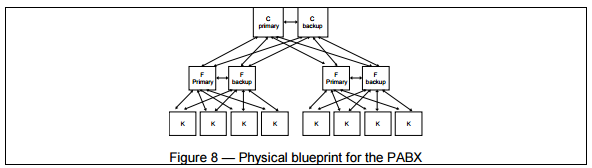
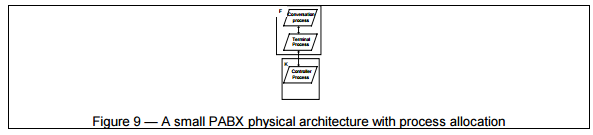
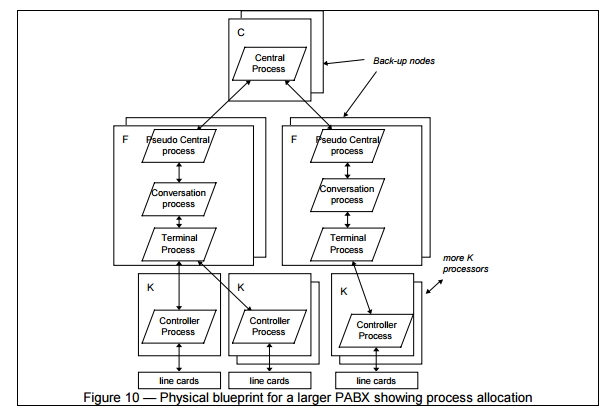


Figure 8 shows one possible hardware configuration for a large PABX, whereas figures 9 and 10 show mappings of the process architecture on two different physical architectures, corresponding to a small and a large PABX. C, F and K are three types of computers of different capacity, supporting three different executables.





Putting it all together The elements in the four views are shown to work together seamlessly by the use of a small set of important scenarios —instances of more general use cases—for which we describe the corresponding scripts (sequences of interactions between objects, and between processes) as described by Rubin and Goldberg6. The scenarios are in some sense an abstraction of the most important requirements. Their design is expressed using object scenario diagrams and object interaction diagrams.

This view is redundant with the other ones (hence the “+1”), but it serves two main purposes:

• as a driver to discover the architectural elements during the architecture design as we will describe later

• as a validation and illustration role after this architecture design is complete, both on paper and as the starting point for the tests of an architectural prototype.

Notation for the Scenarios

The notation is very similar to the Logical view for the components (cf. fig. 2), but uses the connectors of the Process view for interactions between objects (cf. fig. 4). Note that object instances are denoted with solid lines. As for the logical blueprint, we capture and manage object scenario diagrams using Rational Rose.

Example of a Scenario

Fig. 11 shows a fragment of a scenario for the small PABX. The corresponding script reads:

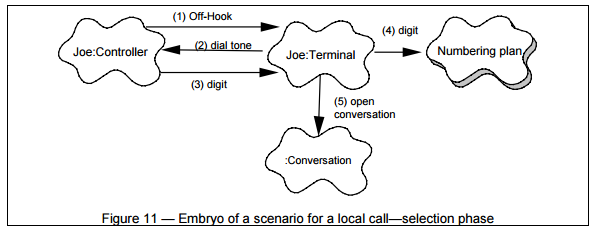
1. The controller of Joe’s phone detects and validate the transition from on-hook to off-hook and sends a message to wake up the corresponding terminal object.

2. The terminal allocates some resources, and tells the controller to emit some dial-tone.

3. The controller receives digits and transmits them to the terminal.

4. The terminal uses the numbering plan to analyze the digit flow.

5. When a valid sequence of digits has been entered, the terminal opens a conversation.



Correspondence Between the Views

The various views are not fully orthogonal or independent. Elements of one view are connected to elements in other views, following certain design rules and heuristics.

From the logical to the process view

We identify several important characteristics of the classes of the logical architecture:

• Autonomy: are the objects active, passive, protected?

-an active object takes the initiative of invoking other objects’ operations or its own operations, and has full control over the invocation of its own operations by other objects

-a passive object never invokes spontaneously any operations and has no control over the invocation of its own operations by other objects

- a protected object never invokes spontaneously any operations but performs some arbitration on the invocation of its operations.

• Persistence: are the objects transient , permanent? Do they the failure of a process or processor? • Subordination: are the existence or persistence of an object depending on another object? • Distribution: are the state or the operations of an object accessible from many nodes in the physical architecture, from several processes in the process architecture?

In the logical view of the architecture we consider each object as active, and potentially “concurrent,” i.e., behaving “in parallel” with other objects, and we pay no more attention to the exact degree of concurrency we need to achieve this effect. Hence the logical architecture takes into account only the functional aspect of the requirements.

However when we come to defining the process architecture, implementing each object with its own thread of control (e.g., its own Unix process or Ada task) is not quite practical in the current state of technology, because of the huge overhead this imposes. Moreover, if objects are concurrent, there must be some form of arbitration for invoking their operations.

On another hand, multiple threads of control are needed for several reasons:

• To react rapidly to certain classes of external stimuli, including time-related events

• To take advantage of multiple CPUs in a node, or multiple nodes in a distributed system

• To increase the CPU utilization, by allocating the CPU to other activities while some thread of control is suspended waiting for some other activity to complete (e.g., access to some external device, or access to some other active object)

• To prioritize activities (and potentially improve responsiveness)

• To support system scalability (with additional processes sharing the load)

• To separate concerns between different areas of the software

• To achieve a higher system availability (with backup processes)

We use concurrently two strategies to determine the ‘right’ amount of concurrency and define the set of processes that are needed. Keeping in mind the set of potential physical target architectures, we can proceed either:

• Inside-out:

Starting from the logical architecture: define agent tasks which multiplex a single thread of control 11 across multiple active objects of a class; objects whose persistency or life is subordinate to an active object are also executed on that same agent; several classes that need to be executed in mutual exclusion, or that require only small amount of processing share a single agent. This clustering proceeds until we have reduced the processes to a reasonably small number that still allows distribution and use of the physical resources.

• Outside-in:

Starting with the physical architecture: identify external stimuli (requests) to the system, define client processes to handle the stimuli and servers processes that only provide services and do not initiate them; use the data integrity and serialization constraints of the problem to define the right set of servers, and allocate objects to the client and servers agents; identify which objects must be distributed.

The result is a mapping of classes (and their objects) onto a set of tasks and processes of the process architecture. Typically, there is an agent task for an active class, with some variations: several agents for a given class to increase throughput, or several classes mapped onto a single agent because their operations are infrequently invoked or to guarantee sequential execution.

Note that this is not a linear, deterministic process leading to an optimal process architecture; its requires a few iterations to get an acceptable compromise. There are numerous other ways to proceed, as shown by Birman et al.5 or Witt et al.7 for example. The precise method used to construct the mapping is outside of the scope of this article, but we can illustrate it on a small example.

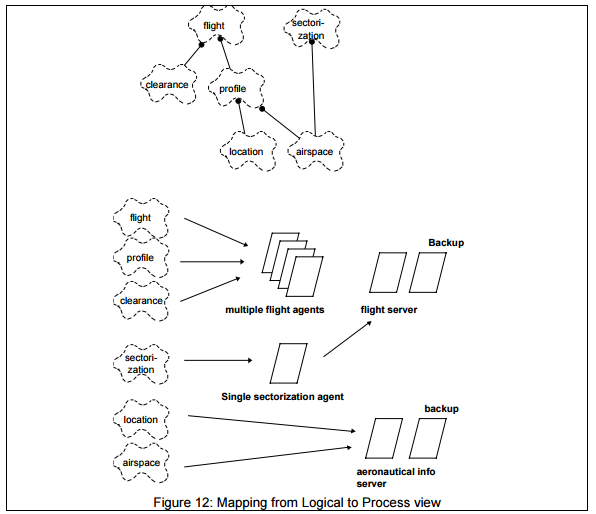
Fig. 12 shows how a small set of classes from some hypothetical air-traffic control system maybe mapped onto processes.

The flight class is mapped onto a set of flight agents: there are many flights to process, a high rate of external stimuli, response time is critical, the load must be spread across multiple CPUs. Moreover the persistency and distribution aspects of the flight processing are deferred to a flight server, which is duplicated for availability reasons.

A flight profile or a clearance are always subordinate to a flight, and although there are complex classes, they share the processes of the flight class. Flights are distributed to several other processes, notably for to display and external interfaces.

A sectorization class, which established a partitioning of airspace for the assignment of jurisdiction of controllers over flights, because of its integrity constraints, can be handled only by a single agent, but can share the server process with the flight: updates are infrequent.

Locations and airspace and other static aeronautical information are protected objects, shared among several classes, rarely updated; they are mapped on their own server, and distributed to other processes.



From logical to development

A class is usually implemented as a module, for example a type in the visible part of an Ada package. Large classes are decomposed into multiple packages. Collections of closely related classes—class categories— are grouped into subsystems. Additional constraints must be considered for the definition of subsystems, such as team organization, expected magnitude of code (typically 5K to 20K SLOC per subsystem), degree of expected reuse and commonality, and strict layering principles (visibility issues), release policy and configuration management. Therefore we usually end up with a view that does not have a one to one correspondence with the logical view.

The logical and development views are very close, but address very different concerns. We have found that the larger the project, the greater the distance between these views. Similarly for the process and physical views: the larger the project, the greater the distance between the views. For example, if we compare fig. 3b and fig. 6, there is no one to one mapping of the class categories to the layers. If we take the ‘External interfaces—Gateway’ category, its implementation is spread across several layers: communications protocols are in subsystems in or below layer 1, general gateway mechanisms are in subsystems in layer 2, and the actual specific gateways in layer subsystems.

From process to physical

Processes and process groups are mapped onto the available physical hardware, in various configurations for testing or deployment. Birman describes some very elaborate schemes for this mapping in the Isis project.

The scenarios relate mostly to the logical view, in terms of which classes are used, and to the process view when the interactions between objects involve more than one thread of control.

Tailoring the Model

Not all software architecture need the full “4+1” views. Views that are useless can be omitted from the architecture description, such as the physical view, if there is only one processor, and the process view if there is only process or program. For very small system, it is even possible that the logical view and the development view are so similar that they do not require separate descriptions. The scenarios are useful in all circumstances.

Iterative process

Witt et al. indicate 4 phases for the design or an architecture: sketching, organizing, specifying and optimizing, subdivided into some 12 steps7. They indicate that some backtracking may be needed. We think that this approach is too “linear” for an ambitious and rather unprecedented project. Too little is known at the end of the 4 phases to validate the architecture. We advocate a more iterative development, were the architecture is actually prototyped, tested, measured, analyzed, and then refined in subsequent iterations. Besides allowing to mitigate the risks associated with the architecture, such an approach has other side benefits for the project: team building, training, acquaintance with the architecture, acquisition of tools, runin of procedures and tools, etc. (We are speaking here of an evolutionary prototype, that slowly grows into becoming the system, and not of throw-away, exploratory prototypes.) This iterative approach also allows the requirements to be refined, matured, better understood.

A scenario-driven approach

The most critical functionality of the system is captured in the form of scenarios (or use cases). By critical we mean: functions that are the most important, the raison d’être of the system, or that have the highest frequency of use, or that present some significant technical risk that must be mitigated.

Start:

• A small number of the scenarios are chosen for an iteration based on risk and criticality. Scenarios may be synthesized to abstract a number of user requirements.

• A strawman architecture is put in place. The scenarios are then “scripted” in order to identify major abstractions (classes, mechanisms, processes, subsystems) as indicated by Rubin and Goldberg6 — decomposed in sequences of pairs (object, operation).

• The architectural elements discovered are laid out on the 4 blueprints: logical, process, development, and physical.

• This architecture is then implemented, tested, measured, and this analysis may detect some flaws or potential enhancement.

• Lessons learned are captured.

Loop:

The next iteration can then start by: • reassessing the risks,

• extending the palette of scenarios to consider

• selecting a few additional scenarios that will allow risk mitigation or greater architecture coverage

Then:

• Try to script those scenarios in the preliminary architecture

• discover additional architectural elements, or sometimes significant architectural changes that need to occur to accommodate these scenarios

• update the 4 main blueprints: logical, process, development, physical

• revise the existing scenarios based on the changes

• upgrade the implementation (the architectural prototype) to support the new extended set of scenario.

• Test. Measure under load, in real target environment if possible.

• All five blueprints are then reviewed to detect potential for simplification, reuse, commonality.

• Design guidelines and rationale are updated.

• Capture the lessons learned.

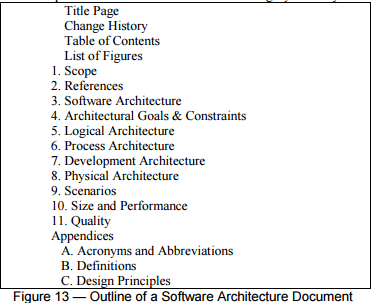
End loop

The initial architectural prototype evolves to become the real system. Hopefully after 2 or 3 iterations, the architecture itself become stable: no new major abstractions are found, no new subsystems or processes, no 14 new interfaces. The rest of the story is in the realm of software design, where, by the way, development may continue using very similar methods and process.

The duration of these iterations varies considerably: with the size of the project to put in place, with the number of people involved and their familiarity with the domain and with the method, and with the degree of “unprecedentedness” of the system w.r.t. this development organization. Hence the duration of an iteration may be 2-3 weeks for a small project (e.g., 10 KSLOC), or up to 6-9 months for a large command and control system (e.g., 700 KSLOC).

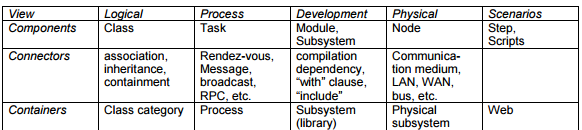
Documenting the architecture

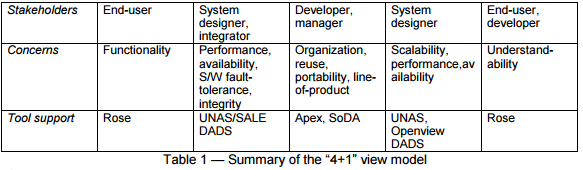
The documentation produced during the architectural design is captured in two documents: • A Software Architecture Document, whose organization follows closely the “4+1” views (cf. fig. 13 for a typical outline) • A Software Design Guidelines, which captures (among other things) the most important design decisions that must be respected to maintain the architectural integrity of the system.



# Conclusion

This “4+1” view model has been used with success on several large projects with or without some local customization and adjustment in terminology4. It actually allowed the various stakeholders to find what they want to know about the software architecture. Systems engineers approach it from the Physical view, then the Process view. End-users, customers, data specialists from the Logical view. Project managers, software configuration staff see it from the Development view. Other sets of views have been proposed and discussed, within Rational and elsewhere, for instance by Meszaros (BNR), Hofmeister, Nord and Soni (Siemens), Emery and Hilliard (Mitre)8, but we have found that often these other views proposed could usually be folded into one of the 4 we described. For example a Cost & Schedule view folds into the Development view, a Data view into the Logical view, an Execution view into a combination of the Process and Physical view.





# 摘要

本文在描述软件密集型系统的体系结构的基础上，利用多个并发视图的模型，将重心放在单独处理架构的各种“利益相关者”：最终用户，开发人员，系统工程师，项目经理等，并分别处理功能性和非功能性需求。每五个视图的描述组合成一个完整的符号，这个观点描述了一个情景驱动，迭代开发而设计的架构

。

**关键词：软件体系结构；图；面向对象的设计；软件开发过程**

# **前言**

我们都已经看到了许多我们都已经看到了试图捕捉系统的体系结构的要点许多书籍和文章，但是在寻找的过程中我们发现他们的作者一直在努力寻找但是很难找到一个和代表实际架构的蓝图，是代表正在运行的程序？或源代码块？或物理计算机？或功能仅仅是逻辑分组？是代表编译依赖的箭头？或控制流动？或数据流？通常它是有点绝对。难道一个架构需要一个单一的架构风格？有时候，软件的架构陷入从系统设计的阶段过早的进入划分软件或过分强调软件开发的泥坑，如：数据工程、运行时效率、或团队组织的开发策略等问题。架构也并不能解决所有“客户”（或者称为“利益相关者”，因此被称为USC）的需求。这些问题在SEI和CMU上由Garland 、 Shaw1, Abowd 、 Allen 和Clements等人提出。作为补救，我们建议使用组织多个并发意见的软件架构的描述，关注与解决每一个特定问题的解决。

# **架构模型**

软件体系结构处理软件的高级结构的设计和实现，它是一种组装一定数量和一些精心选择的形式的架构元件，以满足系统的主要功能和性能的要求以及一些其他的非功能性要求，诸如可靠性、可扩展性、可移植性和可用性的结果。Perry和Wolfe提出并由伯姆修改一个很好的公式：

**软件架构={元素，形式，基本原理/约束}**

软件架构用分解和组合的美学风格解决了抽象的问题，我们用多个视图或视角组成的模型来描述软件架构，为了解决大的和具有挑战性的架构，我们提出的模型主要有五个（参见图1）：

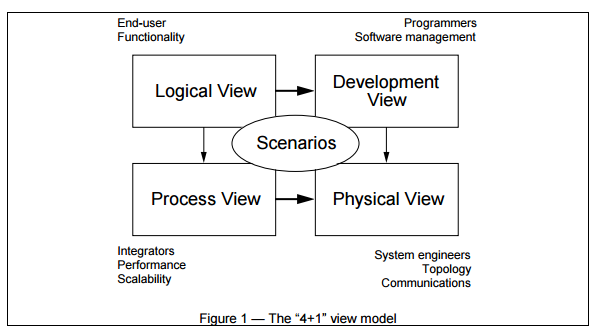
·逻辑视图，这是设计的对象模型（在面向对象的设计方法中使用），

·流程视图，它抓住了设计的并发和同步方面，

·物理视图，它描述了软件的映射（多个）到硬件和反映其分布方面，

·开发者视图：其中描述了在其开发环境的软件的静态组织。

一个体系的决定的定制可以围绕这四个视图来组织说明，然后由几个选定的用例，或成为第五个视图情景说明。该架构是在实际上从部分场景来演变为我们将在后面看到模型。



我们将Perry & Wolf的公式应用于每个视图，即每个视图我们定义元素组使用（组件，容器和连接器），我们捕捉基本原理和制约的形式和工作方式和该架构连接的一些限制。每个视图是通过使用它自己的特殊符号的描述蓝图。对于每个视图，架构师还可以选择一定的架构风格，因此允许多种风格并存于一个系统中。

现在，我们将逐个来看看每五个观点，并给出每个观点的宗旨：它涉及的地址、相应的架构蓝图的符号以及我们用来描述和管理它的工具。小例子是从PABX，我们从Alcatel的业务系统和空中交通管制系统工作所得的设计绘制，但是得到的只是简单的表现形式。在这里，我们的意图是对视图给吃一些建议，而不是定义这字儿系统的体系结构。

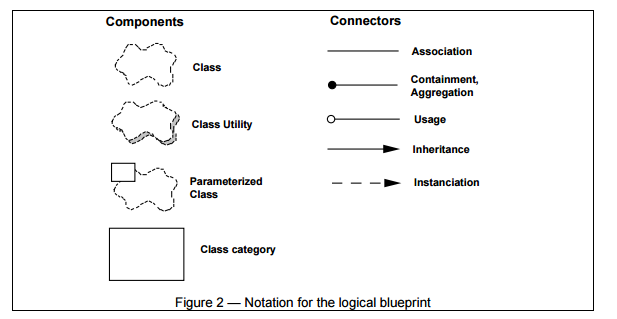
对于其他符号和工具来说，“4 +1”视图模型是相当“通用”，可以使用其他的设计方法可以使用，特别是对于该逻辑和过程分解，结果已经表明，我们已经能够成功的使用这些模型。

# 逻辑架构

逻辑架构主要支持系统应该在用户服务方面提供功能性需求，该系统从采取的问题域（大部分），以对象或对象类的形式分解为一系列的关键抽象模型。他们利用抽象，封装和继承的原则。用于在整个系统的各个部分确定共同的机制和设计元素。我们使用类图和类模板的手段和合理的方法表示逻辑架构，类图展示了一组类和它们的逻辑关系：关联、使用、组合和继承等等。类模板专注于每个类别，他们强调主类业务，并确定主要对象的特点。如果重要的是要确定的对象的内部行为，这是状态转移图，或状态图进行图。公共机制或服务类定义的实用程序。一个应用程序以面向对象的方式进行，这种数据驱动可使用逻辑视图来表示，如E-R图的一些表现形式。

## 2.1 符号的逻辑视图

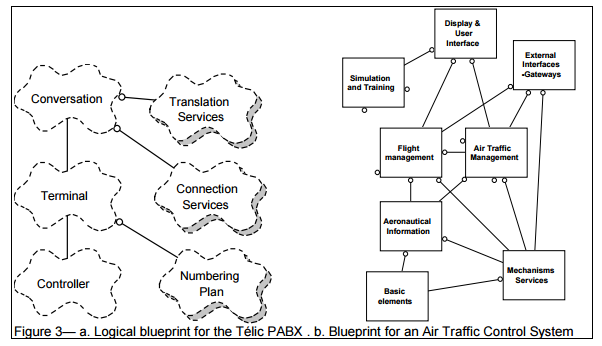
逻辑视图的符号是从Booch标记法来的。它大大简化为仅考虑那些架构显著的项目。特别是在众多的装饰不是在这个水平设计的时候是非常有用的。我们使用RationalRose中支持逻辑体系结构设计。



## 2.2 逻辑视图风格

我们使用的逻辑视图的样式是一种面向对象的样式，逻辑视图的设计的主要原则是尽量保持一个单一的、连贯的对象模型在整个系统中，避免每个网站或每个处理器类和机制过早专业化。

逻辑蓝图示例图3a示出了所涉及的TELIC PBX架构的主要类。



在PABX建立终端之间交互。终端可以是一部电话，一个干线（即线路中心局），联络线（即私人PABX到PABX线），功能电话线，数据线，ISDN线路等。不同线由不同的线路接口卡支持。线路controller对象的职责是进行解码和注入线路接口卡上的所有信号，转换卡特定的信号，并从一个小的，统一组事件：启动，停止，数字等，该控制器还承担所有硬实时约束。这个类有许多子类以满足不同类型的接口。终端对象的责任是维持终端的状态，并代表该行洽谈业务。例如，它使用编号计划的服务，以解释在选择阶段的拨号。谈话表示一组从事的谈话终端。对话使用翻译服务（目录，逻辑到物理地址的映射，路由），以及连接服务来建立终端之间的语音通路。

一个更大的系统，其中包含的建筑意义几十类，图3b示出的空中交通管制系统的顶层类图，含有类（类即，组）。

# 流程架构

流程架构考虑一些非功能性需求，如性能和可用性，它解决并发和分配的问题。与系统的完整性、容错性，以及从流程体系结构的控制线程是否用于实际执行的对象的动作中的逻辑视图无关。

这个过程架构可以在几个抽象层次来描述，每个层次针对不同的担忧。在最高级别，则处理结构可以被看作是一组独立地执行通信程序的逻辑网络（称为“过程”），一组由局域网或广域网连接的硬件资源的分配。多个逻辑网络可以同时存在，共享相同物理资源。例如，独立的逻辑网络可以用于支持从离线系统到上线运作系统的分离，以及配套的软件的模拟或测试版本的共存。

一个进程是形成可执行单元任务的分组。流程代表在该进程架构可以战术控制的水平（即启动、恢复、重新配置和关闭）。除此之外，过程可以复制以增加处理负荷的分布，或用于改善系统可用性。

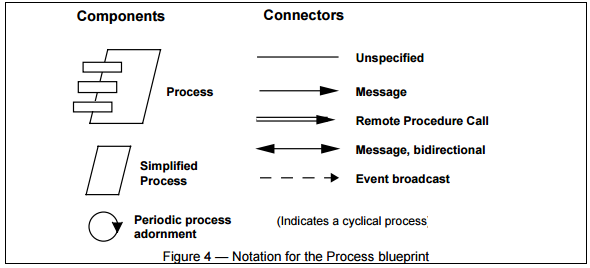
软件被划分为一组独立的任务。任务是控制一个独立的线程，可以单独一在个处理节点上进行调度。

主要任务，即是可以唯一地寻址的架构元素和次要任务，由于某些原因局部引入额外的任务（周期性活动，缓冲剂，超时，等等）。它们可被实现为关键任务，例如，轻量级线程。同步和异步基于消息的通信服务，远程过程调用，事件广播等次要任务可以通过会合或共享存储器通信：主要任务经由一组的定义良好的任务间通信机制进行通信。

消息流，过程负载可根据流程蓝图来估计。另外，也可以实现与用于进程假负载“空”的过程体系结构测量目标系统的性能，如由Filarey等人在他们的欧洲航空安全组织实验所述。

## 3.1 流程视图符号

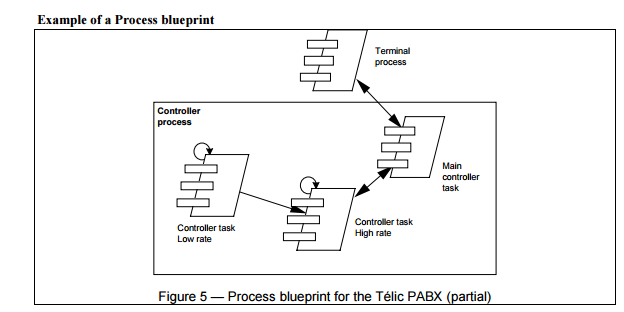
我们使用的过程视图的符号是从最初由Booch的VIA为任务提出的符号扩展。再次使用的符号侧重于在结构上比较显著的元素。（图4）



我们已经使用了通用网络架构服务（UNAS）的产品，由TRW架构师实施一整套流程和任务（和它们的冗余）到过程网络。 UNAS包含一个支持这样的符号工具软件的生命周期环境。要买允许处理架构的图形描述，包括可能的任务间的通信路径。这种方法来指定和执行的过程体系结构的好处是，变化可以容易而对应用软件太大的影响。

## 3.2 流程视图的风格

几种风格会适合流程视图。例如，摘自Garlan和Shaw的taxonomy文章我们可以有：管和过滤器，或者客户机/服务器，与多个客户机/单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，人们可以使用类似于由K. Birman与另一个符号和工具集所描述的ISIS系统的进程方法组的样式。



所有终端是由单个终端组成的过程，这是消息在其输入队列做的驱动处理。该控制器对象上组成控制器进程有三种任务执行：低周期率的任务会扫描所有的非活动终端（200毫秒），使任何终端在高循环率任务的扫描列表变得活跃（10毫秒），这检测状态的任何显著变化将它们传递给它解释的变化和由消息通信他们相应的终端的主控制装置的任务。在这里，通过控制器过程中的信息通过共享内存进行。

# 开发者架构

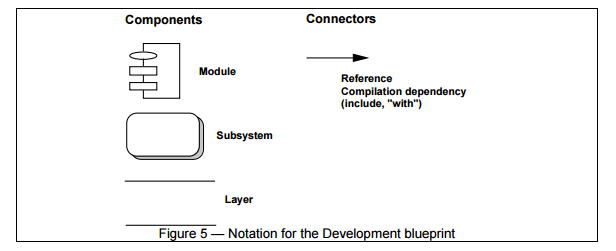
开发架构的重点是软件开发环境的实际软件模块组织。软件被封装在小块的程序库或子系统中使得少数的开发得到发展。每个层中提供明确的接口，子系统在接口之上。

系统的开发者架构是由模块和子系统图来表示，显示了“出口”和“进口”的关系。当软件的所有要素已经确定才能说是完整的开发架构。然而，也有可能会列出治理开发架构的规则：分区和分组的知名度。

在大多数情况下，开发架构考虑到易于开发和软件管理，再利用或通用的内部的要求，由工具集或编程语言施加约束。在此基础上，对工作团队（甚至是团队组织）分配、成本评估、计划、监测项目的进展和有关软件重用、可移植性、安全性的推理用于建立一个行的产品的基础。

符号的发展蓝图

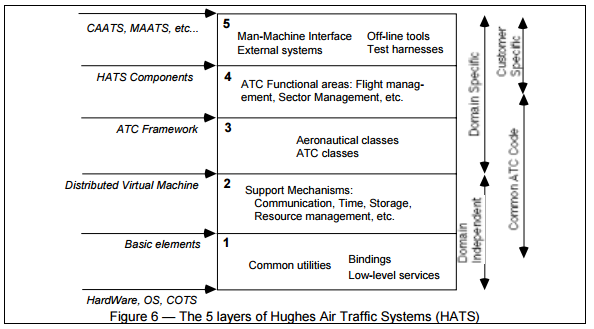
Booch标记法的再次变型中，它限制了那些结构上显著的项目。



从理性的Apex开发环境支持的定义和发展结构分析上述的分层策略的实施，以及设计规则的实施。 Rational Rose可以在模块和子系统级绘制蓝图，来为Ada和C ++推进工程和从开发源代码的逆向工程。

## 4.1 开发者视图的风格

我们建议采用的开发者分层的风格，定义一些4至6层的子系统。每一层都有一个明确定义的责任。设计规则是，某一子系统只能依靠在相同的层中或在下面的层中子系统，这样是为了最小化模块间的非常复杂的依赖关系，网络的发展允许简单释放策略层：



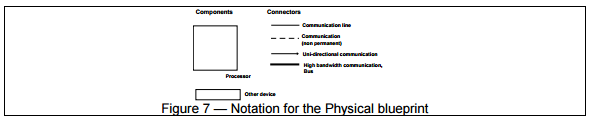
## 4.2 开发架构的例子

 图6表示了线的产品由Canada的休斯飞机公司开发的空中交通管制系统的五层开发组织。这是对应于图1所示的逻辑体系结构的发展架构。

图层1和2构成从硬件平台出发和领域无关的分布式基础设施中的产品和盾牌出发，操作系统，或关闭的现成的产品，如数据库管理系统变化的线普遍。此架构，层3增加了一个ATC框架以形成一个特定领域的软件体系结构。使用该框架的功能的调色板是建立在层45层是非常客户和产品相关的，并包含大部分用户接口和接口与外部系统。一些子系统横跨5层，含有每10至50个模块，可以在额外的蓝图来表示。

 物理体系结构考虑到主要系统的非功能性要求，诸如可用性，可靠性（容错），性能（吞吐量）和可扩展性。软件执行计算机，或处理节点的网络上（或只是短节点）。的各种元素identified-网络，过程，任务和对象 - 需要被映射到各个节点。我们预计，几个不同的物理配置将被使用：一些用于开发和测试，其他对系统的各种站点或针对不同客户的部署。因此，软件的到节点的映射需要高度灵活和对源代码本身的影响极小。

物理符号：物理蓝图可以成为在大型系统但是很凌乱，因此他们采取几种形式，利用或不利用从流程视图的映射。



从UNAS TRW提供给我们在这里与流程架构映射到物理架构，允许一大类在映射的变化无需修改源代码的数据驱动的方式。

物理蓝图的示例：

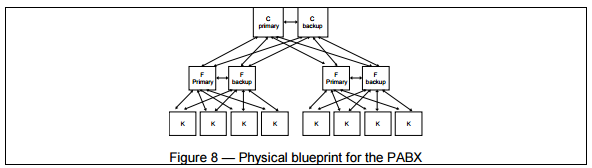
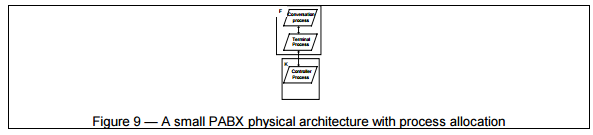
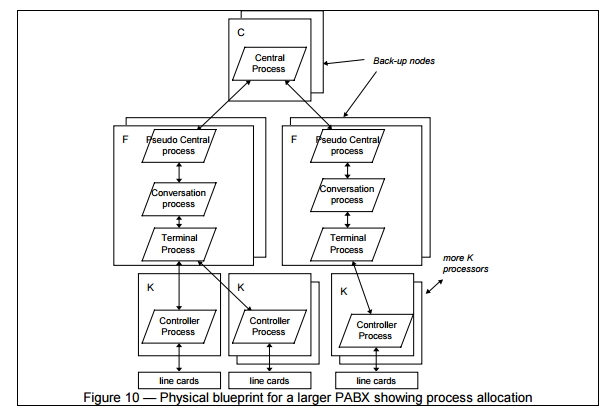


图8示出了用于大PABX的一个可能的硬件配置，而图9和处理体系结构的在两个不同的物理架构10示出了映射，对应于小和大的PABX。 C，F和K是三种不同容量的计算机，支持三种不同的可执行文件。





全部放在一起的四个视图的元素显示在一起无缝地通过使用一小部分的应用较为普遍的场景重要的-instances案件工作，为我们描述了相应的脚本对象之间的相互作用（序列之间如由Rubin和Goldberg6所描述的过程）。该方案是在某种意义上的最重要的要求的抽象。它们的设计使用对象场景图和对象交互图来表示。

这种观点是多余的，但是它有两个主要目的：

•作为驱动架构设计过程中发现的建筑元素，我们将在后面介绍

•本体系结构设计后验证和说明的作用完成后，无论在纸和作为起点用于建筑原型的测试。

符号是非常相似的组件的逻辑视图（参见图2），但使用对对象之间的交互的进程视图的连接器（参见图4）。请注意，对象实例被表示用实线。至于逻辑蓝图，我们获取和使用Rational Rose的管理对象的场景图。

 图11示出一个情形为小PABX的片段。相应的脚本如下：

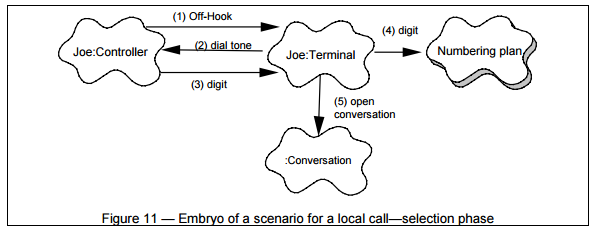
1. Joe的电话检测的控制器和验证从挂机到摘机的过渡，并发送一个消息以唤醒对应的终端的对象。

2.终端分配一些资源，并告诉控制器发出一些拨号音。

3.控制器接收数字并且将它们发送到终端。

4.终端使用编号方案来分析数字流。

5.当已输入的位数的有效序列，终端打开会话。



各种看法并不完全正交的或独立的。一种观点的元件连接到其他意见，遵循一定的设计规则和启发式的元素。

从逻辑到过程视图

 我们确定了逻辑体系结构中的类的几个重要特征：

•自治：是的对象主动的，被动的，受保护？

 -an活动对象需要调用其他对象的操作或自己的业务的积极性，并具有完全控制自己的业务则由其他对象的调用

 -a被动对象从未自发地调用任何操作，并拥有自己的业务则由其他对象调用没有控制

- 受保护对象从未自发地调用任何操作，但其操作的调用执行一些仲裁。

•持久性：是的对象暂时的，永久的吗？难道他们的过程或处理器的失败？ •从属：是根据另一个对象的物体的存在或持久？ •分布：是国家或对象的操作从多个节点在物理架构访问，从进程架构几个进程？

在体系结构的逻辑视图，我们认为每个对象为当前和潜在的“并发”，即行为“平行”与其他对象，我们付出没有更多的关注并发的准确度，我们需要达到这种效果。因此，逻辑体系结构只考虑要求的功能方面。

然而，当我们来定义过程架构，拥有自己的控制线程执行的每个对象（例如，它自己的Unix进程或Ada的任务）是不是在目前的技术状态相当实用，因为巨大的开销这会带来。此外，如果对象是并发的，必须有某种形式的仲裁的用于调用它们的操作。

在另一方面，还需要有几个原因多个控制线程：

 •要迅速反应以某些类的外部刺激的，包括时间相关的事件

•要利用中的一个节点多个CPU或多个节点的分布式系统

•为了提高CPU的利用率，通过在某些控制线程被挂起，等待其他活动来完成分配CPU的其他活动（例如，访问一些外部设备，或访问其他一些活动对象）

•优先活动（和潜在提高响应能力）

•为了支持系统的可伸缩性（额外进程共享负载）

•要分离的软件不同区域之间的关切

 •为了实现更高的系统可用性（与备份流程）

我们同时使用两种策略来决定并发的“正确”的数额和定义一组的是需要的过程。牢记该组潜在的物理目标架构，我们可以进行两种：

• 相反：

从逻辑架构开始：定义哪些复跨类的多个活动对象控制11单线程代理任务;对象，它们的持续性或生活是从属于活动对象也上同一代理执行;需要几类在互斥执行，或仅需要少量的处理共享单个代理。该集群进行，直到我们减少了流程，合理少数仍然允许物理资源的分配和使用。

•外而内：

与物理结构开始：确定外部刺激（请求）到系统中，定义客户端进程来处理的刺激和服务器流程，只提供，并且不启动它们;使用问题的数据的完整性和序列约束来定义权组服务器，和分配对象到客户端和服务器代理;确定哪些对象必须分布。

结果是类（和它们的对象）到一组任务和过程体系结构的过程的一个映射。典型地，有用于有源类的试剂的任务，有一些变化：几种药剂对于给定的类，以增加吞吐量，或者映射到单一试剂，因为它们的操作被频繁调用或以保证顺序执行几个类。

注意，这是不是线性的，确定性过程导致一个最佳的处理体系结构;它需要一些迭代来得到一个可接受的妥协。还有许多其他的方法来进行，如由伯曼等[5]或维特等[7]例如。用于构建映射的精确的方法是本文的范围之外，但我们可以说明它在一个小例子。

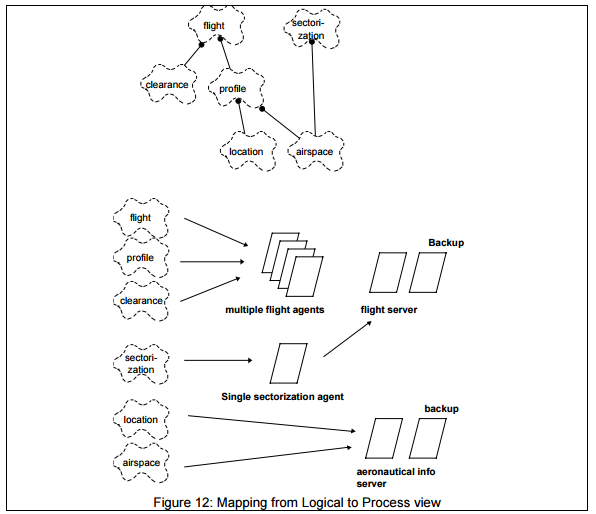
图 12显示了如何从一些假设的空中交通控制系统的一小部分的类可能映射到进程。

飞行类映射到一组飞行剂：有许多航班来处理，外界刺激率高，响应时间很关键，负载必须跨多个CPU进行传播。此外，该飞行处理的持久性和分布方面推迟到飞行服务器，该被复制的可用性的原因。

一个飞行剖面或间隙总是从属于某个航班，尽管有复杂的类，它们共享飞行类的过程。航班被分发到其他几个过程，特别是用于显示和外部接口。

一个扇区的类，它建立空域的分割为控制器辖航班的分配，因为它的完整性约束，可以通过一个单一的代理只处理，但可以用飞行共享服务器进程：更新频繁。

位置和空域等静态航空信息被保护的对象，几个类中共享，很少更新;它们被映射自己的服务器上，并分发到其它过程。



对应视图之间

各种看法并不完全正交的或独立的。一种观点的元件连接到其他意见，遵循一定的设计规则和启发式的元素。

从逻辑到过程视图

 我们确定了逻辑体系结构中的类的几个重要特征：

•自治：是的对象主动的，被动的，受保护？

 -活动对象需要调用其他对象的操作或自己的业务的积极性，并具有完全控制自己的业务则由其他对象的调用

 -被动对象从未自发地调用任何操作，并拥有自己的业务则由其他对象调用没有控制

- 受保护对象从未自发地调用任何操作，但其操作的调用执行一些仲裁。

•持久性：是的对象暂时的，永久的吗？难道他们的过程或处理器的失败？ •从属：是根据另一个对象的物体的存在或持久？ •分布：是国家或对象的操作从多个节点在物理架构访问，从进程架构几个进程？

在体系结构的逻辑视图，我们认为每个对象为当前和潜在的“并发”，即行为“平行”与其他对象，我们付出没有更多的关注并发的准确度，我们需要达到这种效果。因此，逻辑体系结构只考虑要求的功能方面。

然而，当我们来定义过程架构，拥有自己的控制线程执行的每个对象（例如，它自己的Unix进程或Ada的任务）是不是在目前的技术状态相当实用，因为巨大的开销这会带来。此外，如果对象是并发的，必须有某种形式的仲裁的用于调用它们的操作。

在另一方面，还需要有几个原因多个控制线程：

 •要迅速反应以某些类的外部刺激的，包括时间相关的事件

•要利用中的一个节点多个CPU或多个节点的分布式系统

•为了提高CPU的利用率，通过在某些控制线程被挂起，等待其他活动来完成分配CPU的其他活动（例如，访问一些外部设备，或访问其他一些活动对象）

•优先活动（和潜在提高响应能力）

•为了支持系统的可伸缩性（额外进程共享负载）

•要分离的软件不同区域之间的关切

 •为了实现更高的系统可用性（与备份流程）

我们同时使用两种策略来决定并发的“正确”的数额和定义一组的是需要的过程。牢记该组潜在的物理目标架构，我们可以进行两种：

• 反了：

从逻辑架构开始：定义哪些复跨类的多个活动对象控制11单线程代理任务;对象，它们的持续性或生活是从属于活动对象也上同一代理执行;需要几类在互斥执行，或仅需要少量的处理共享单个代理。该集群进行，直到我们减少了流程，合理少数仍然允许物理资源的分配和使用。

•外而内：

与物理结构开始：确定外部刺激（请求）到系统中，定义客户端进程来处理的刺激和服务器流程，只提供，并且不启动它们;使用问题的数据的完整性和序列约束来定义权组服务器，和分配对象到客户端和服务器代理;确定哪些对象必须分布。

结果是类（和它们的对象）到一组任务和过程体系结构的过程的一个映射。典型地，有用于有源类的试剂的任务，有一些变化：几种药剂对于给定的类，以增加吞吐量，或者映射到单一试剂，因为它们的操作被频繁调用或以保证顺序执行几个类。

注意，这是不是线性的，确定性过程导致一个最佳的处理体系结构;它需要一些迭代来得到一个可接受的妥协。还有许多其他的方法来进行，如由伯曼等[5]或维特等[7]例如。用于构建映射的精确的方法是本文的范围之外，但我们可以说明它在一个小例子。

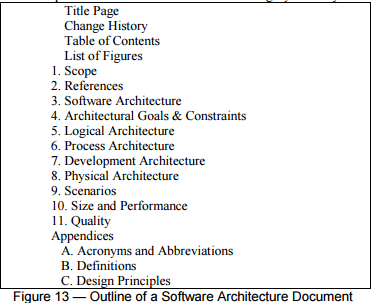
图. 12显示了如何从一些假设的空中交通控制系统的一小部分的类可能映射到进程。

飞行类映射到一组飞行剂：有许多航班来处理，外界刺激率高，响应时间很关键，负载必须跨多个CPU进行传播。此外，该飞行处理的持久性和分布方面推迟到飞行服务器，该被复制的可用性的原因。

一个飞行剖面或间隙总是从属于某个航班，尽管有复杂的类，它们共享飞行类的过程。航班被分发到其他几个过程，特别是用于显示和外部接口。

一个扇区的类，它建立空域的分割为控制器辖航班的分配，因为它的完整性约束，可以通过一个单一的代理只处理，但可以用飞行共享服务器进程：更新频繁。

位置和空域等静态航空信息被保护的对象，几个类中共享，很少更新;它们被映射自己的服务器上，并分发到其它过程。.



# 结论

这种“4 +1”视图模型已被用于与带或不带一些当地的定制和调整一些大型项目的成功。它实际上是允许各利益相关者找到他们想要了解的软件架构是什么。系统工程师从物理视图，则该进程视图接近它。最终用户，客户，数据专家从逻辑视图。项目经理，软件配置的工作人员看到它从发展观。已经提出的意见等集和讨论，在Rational和其他地方，例如通过梅萨罗斯（BNR），霍夫迈斯特，北站和瑞里（西门子），金刚砂和希利亚德（米特），但我们发现，提出往往是这些其他视图可以通常折叠成我们描述了4中的一个。例如，一个成本及日程视图折叠成发展观，数据视图进入逻辑视图，执行视图到过程和物理视图的组合。

