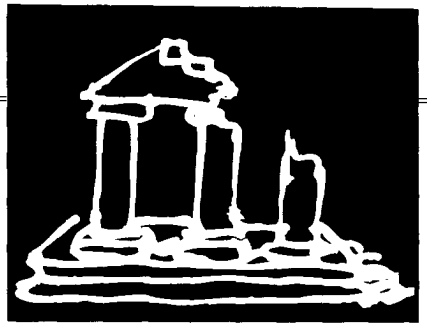

The 4+1 View Model of Architecture

PHILIPPE B. KRUCHTEN, *Rational Software*

◆*The 4+1 View Model organizes a description of a software architecture using five concurrent views, each of which addresses a specific set of concerns. Architects capture their design decisions in four views and use the fifth view to illustrate and validate them.*

We all have seen many books and articles in which a single diagram attempts to capture the gist of a system architecture. But when you look carefully at the diagram's boxes and arrows, it becomes clear that the authors are struggling to represent more in one diagram than is practical. Do the boxes represent running programs? Chunks of source code? Physical computers? Or merely logical groupings of functionality? Do the arrows represent compilation dependencies? Control flows? Dataflows? Usually the answer is that they represent a bit of everything.

Does an architecture need a single architectural style? Sometimes the software architecture suffers from system designers who go too far, prematurely partitioning the software or overemphasizing one aspect of development (like data engineering or runtime efficiency), development strategy, or team organization. Other software architectures fail to address the concerns of all "customers."

Several authors have noted the problem of architectural representation, including David Garlan and Mary Shaw,[1] Gregory Abowd and Robert Allen,[2] and Paul Clements.[3]

The 4 + 1 View Model was developed to remedy the problem. The 4 + 1 model describes software architecture using five concurrent *views*. As Figure 1 shows, each addresses a specific set of concerns of interest to different stakeholders in the system.

♦ The *logical* view describes the

0740-7459/94/$04.00 © 1994 IEEE

design's object model when an object-oriented design method is used. To design an application that is very data-driven, you can use an alternative approach to develop some other form of logical view, such as an entity-relationship diagram.

♦ The *process* view describes the design's concurrency and synchronization aspects.

♦ The *physical* view describes the mapping of the software onto the hardware and reflects its distributed aspect.

♦ The *development* view describes the software's static organization in its development environment.

Software designers can organize the description of their architectural decisions around these four views, and then illustrate them with a few selected use cases, or *scenarios*, which constitute a fifth view. The architecture is partially evolved from these scenarios.

At Rational, we apply Dewayne Perry and Alexander Wolf's formula[4]

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

independently on each view. For each view we define the set of elements to use (components, containers, and connectors), capture the forms and patterns that work, and capture the rationale and constraints, connecting the architecture to some of the requirements.

Each view is described by what we call a "blueprint" that uses its own particular notation. The architects can also pick a certain *architectural style* for each view, thus allowing the coexistence of multiple styles in one system.

The 4+1 View Model is rather generic: You can use notations and tools other than those we describe, as well as other design methods, especially for the logical and process decompositions.

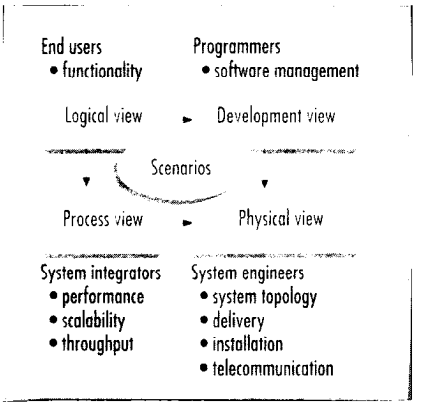

***Figure 1.** The 4+1 View Model is used to organize the description of the architecture of a software-intensive system.*

4+1 VIEW MODEL

Software architecture deals with abstraction, decomposition and composition, and style and aesthetics. It also deals with the design and implementation of software's high-level structure.

Designers build architectures using several architectural elements in well-chosen forms. These elements satisfy the major functionality and performance requirements of the system as well as other, nonfunctional requirements such as reliability, scalability, portability, and system availability.

Logical view. The logical view primarily supports the functional requirements — the services the system should provide to its end users. Designers decompose the system into

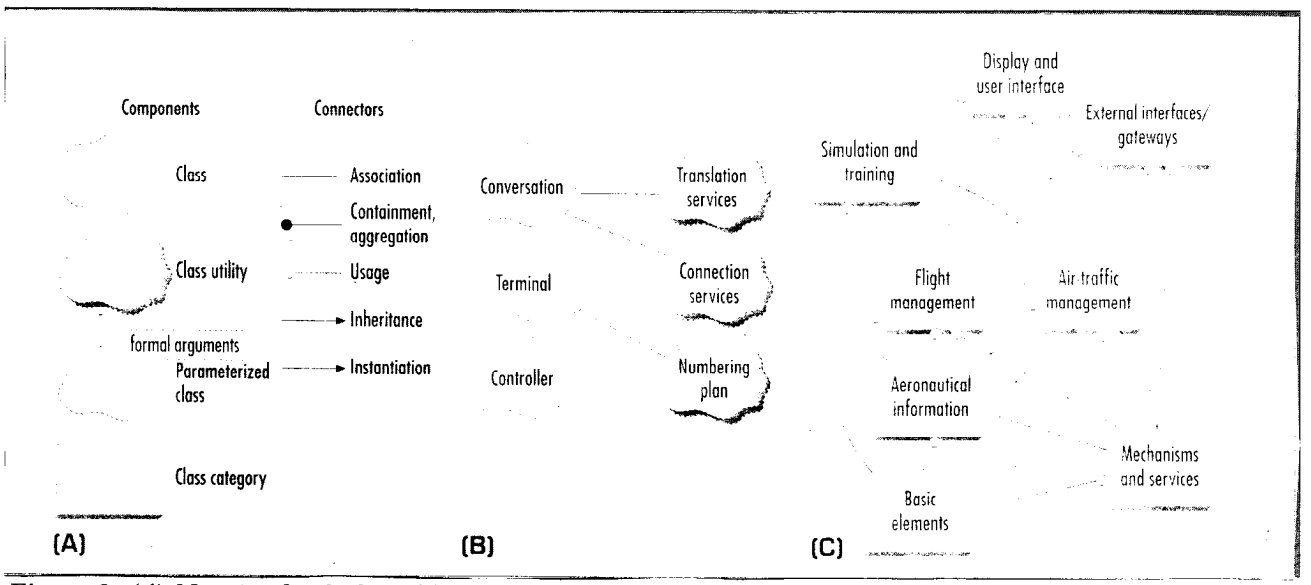

***Figure 2.** (A) Notation for the logical blueprint; (B) logical blueprint for the Télic PBX; (C) blueprint for an air-traffic control system.*

a set of key abstractions, taken mainly from the problem domain. These abstractions are *objects* or *object classes* that exploit the principles of abstraction, encapsulation, and inheritance. In addition to aiding functional analysis, decomposition identifies mechanisms and design elements that are common across the system.

We use the Rational/Booch approach[5] to represent the logical view through class diagrams and templates. A *class diagram* shows a set of classes and their logical relationships: association, usage, composition, inheritance, and so on. Designers can group sets of related classes into *class categories*. *Class templates* focus on each individual class; they emphasize the main class operations and identify key object characteristics. If an object's internal behavior must be defined, we use state-transition diagrams or state charts. *Class utilities* define common mechanisms or services.

Notation. We derived the logical-view notation in Figure 2a from the Booch notation, which we simplified considerably to account for only those items that are architecturally significant. The numerous adornments are not very useful at this level of design. We use Rational Rose to support the logical-view design.

Style. For the logical view, we use an object-oriented style. The main design guideline we follow is to keep a single, coherent object model across the entire system, avoiding the premature specialization of classes and mechanisms for each site or processor.

Examples. Figure 2b shows the main classes involved in a sample PBX architecture we developed at Alcatel. A PBX establishes communication among terminals. A terminal might be a telephone, a trunk line (a line to the central office), a tie line (a private PBX-to-PBX line), or a feature phone line.

Different lines are supported by different line-interface cards. The Controller object decodes and injects all the signals on the line-interface card, translating card-specific signals to and from a small, uniform set of events, such as a "start," "stop," or "digit." The controller also bears all the hard real-time constraints. This class has many subclasses that cater to different interfaces.

The Terminal object maintains the state of a terminal and negotiates services on behalf of that line. For example, it uses the services of the Numbering Plan object to interpret dialing.

The Conversation object represents a set of terminals engaged in a conversation. It uses the Translation Services object (for accessing a directory, mapping a logical address to a physical one, and routing) and the Connection Services object to establish a voice path among the terminals.

Larger systems contain dozens of architecturally significant classes, such as the top-level class diagram of an air-traffic control system[6] in Figure 2c. The system, developed by Hughes Aircraft of Canada, contains eight class categories.

Process view. The process view takes into account some nonfunctional

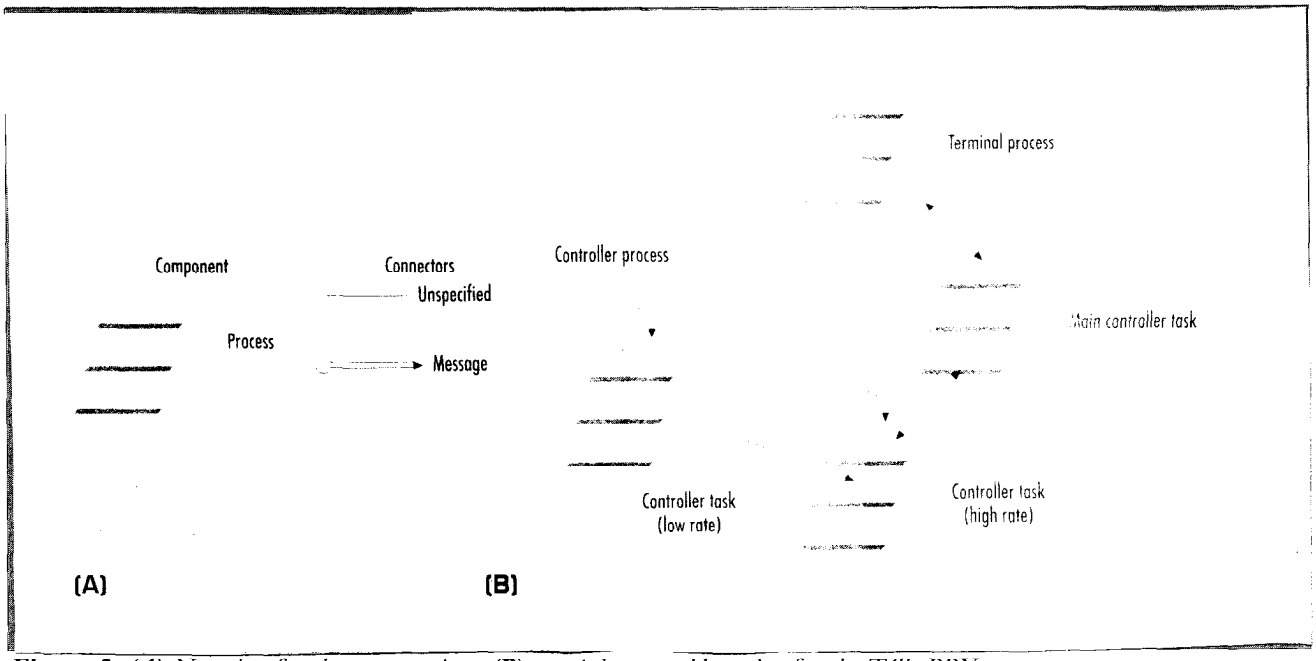

Figure 3. *(A) Notation for the process view; (B) partial process blueprint for the Télic PBX.*

requirements, such as performance and system availability. It addresses concurrency and distribution, system integrity, and fault-tolerance. The process view also specifies which thread of control executes each operation of each class identified in the logical view.

Designers describe the process view at several levels of abstraction, each one addressing a different concern. At the highest level, the process view can be seen as a set of independently executing logical networks of communicating programs ("processes") that are distributed across a set of hardware resources, which in turn are connected by a bus or local area network or wide area network. Multiple logical networks may exist simultaneously, sharing the same physical resources. For example, you can use independent logical networks to separate on- and off-line operational systems and to represent the coexistence of simulation or test versions of the software.

A *process* is a group of tasks that form an executable unit. Processes represent the level at which the process view can be tactically controlled (started, recovered, reconfigured, shut down, and so on). In addition, processes can be replicated to distribute processing load or improve system availability.

Partitioning. To develop the process view, designers partition the software into a set of independent *tasks:* separate threads of control that can be individually scheduled on separate processing nodes.

We separate tasks into two groups:

♦ *Major tasks* are the architectural elements that can be uniquely addressed (designated from another task). They communicate through a set of well-defined intertask-communication mechanisms: synchronous and asynchronous message-based communication services, remote procedure calls, event broadcasts, and so on. Major tasks should not make assumptions about their collocation in the same process or processing node.

♦ *Minor tasks* are additional tasks introduced locally for implementation reasons such as cyclical activities, buffering, and time-outs. They can be implemented as Ada tasks or lightweight threads, for example, and communicate by rendezvous or shared memory.

We use the process blueprint to estimate message flow and process loads. It is also possible to implement a "hollow" process view with dummy process loads and measure its performance on a target system.[7]

TO DEVELOP THE PROCESS VIEW, THE DESIGNER PARTITIONS THE SOFTWARE INTO SEPARATE TASKS.

Notation. Our process-view notation is expanded from Booch's original notation for Ada tasking and focuses on architecturally significant elements, as Figure 3a shows.

We have used TRW's Universal Network Architecture Services to build and implement the processes and tasks (and their redundancies) into networks of processes. UNAS contains a tool — the Software Architects Lifecycle Environment — that supports our notation. SALE lets us depict the process view graphically, including specifications of the possible intertask-communication paths. It can then automatically generate the corresponding Ada or C++ source code. Because it supports automatic code generation, SALE makes it easier to change the process view.

Style. Several styles would fit the process view. For example, picking from Garlan and Shaw's taxonomy,[1] you can use pipes and filters or client/server, with variants of multiple-client/single-server and multiple-clients/multiple-servers. For more complex systems, you can use a style similar to the ISIS system's process groups, as described by Kenneth Birman using another notation and toolset.[8]

Example. Figure 3b shows a partial process view for the PBX introduced in Figure 2b. All terminals are handled by a single terminal process that is driven by messages in its input queues. The Controller objects are executed on one of three tasks that comprise the controller process: a low cycle-rate task, which scans all inactive terminals (200 ms) and puts any terminal becoming active in the scan list of the high cycle-rate task (10 ms), which detects any significant changes of state and passes them to the main controller task, which interprets the changes and communicates them by message to the corresponding terminal. Message passing within the controller process is done through shared memory.

Development view. The development view focuses on the organization of the actual software modules in the software-development environment. The software is packaged in small chunks — program libraries or *subsystems* — that can be developed by one or more 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 view takes into account internal requirements related to ease of development, software management, reuse or commonality, and constraints imposed by the toolset or the programming language. The development view supports the allocation of requirements and work to teams, and supports cost evaluation, planning, monitoring of project progress, and reasoning about software reuse, portability, and security. It is the basis for establishing a line of product.

The development view is represented by module and subsystem diagrams that show the system's export and import relationships. You can describe the complete development view only

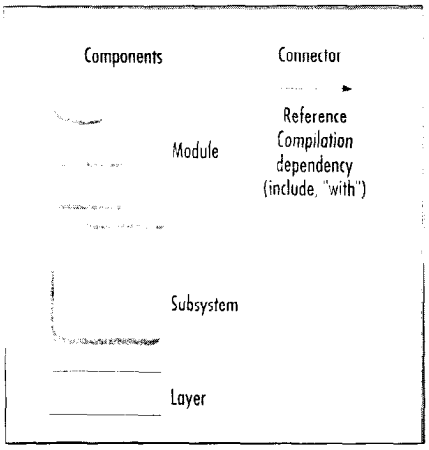

Figure 4. *Notation for a development blueprint.*

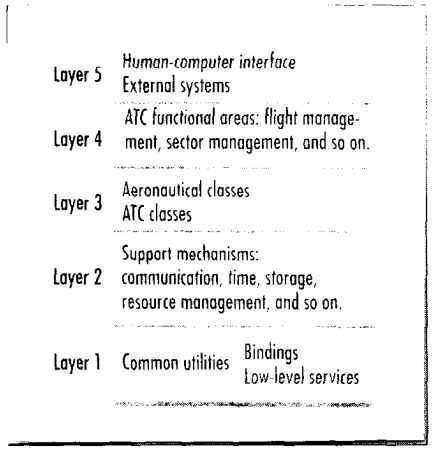

Figure 5. *The five layers of Hughes' Air Traffic System.*

after you have identified all the software elements. However, you can list the rules that govern the development view — partitioning, grouping, and visibility — before you know every element.

Notation. As Figure 4 shows, we again use a variation of the Booch notation, limited to architecturally significant items. Rational's Apex development environment supports the definition and implementation of the development view, the layering strategy described above, and design-rule enforcement. Rational Rose can draw the development blueprints for Ada and C++ at the module and subsystem level, in forward engineering, and by reverse engineering from the development source code.

Style. We recommend you define four to six layers of subsystems in the development view. One design rule we follow here is that a subsystem can only depend on subsystems in the same or lower layers. This minimizes the development of very complex networks of dependencies between modules in favor of a simpler, layer-by-layer release strategy.

Examples. As Figure 5 shows, the Hughes Air Traffic System has five development layers.[6] Layers 1 and 2 — utilities and support mechanisms — constitute a domain-independent, distributed infrastructure that is common across the line of products. These layers shield the application from variations in hardware platforms, operating systems, or off-the-shelf products such as database-management systems. To this infrastructure, layer 3 adds an air-traffic control framework to form a *domain-specific* software architecture. Layer 4 adds a palette of functionality, and layer 5 contains most of the user interface and the interfaces to external systems. This top layer is customer- and product-dependent. Spread across the five layers are some 72 subsystems, each containing from 10 to 50 modules. We represent these subsystems on additional, more detailed blueprints.

Physical view. The physical view takes into account the system's non-

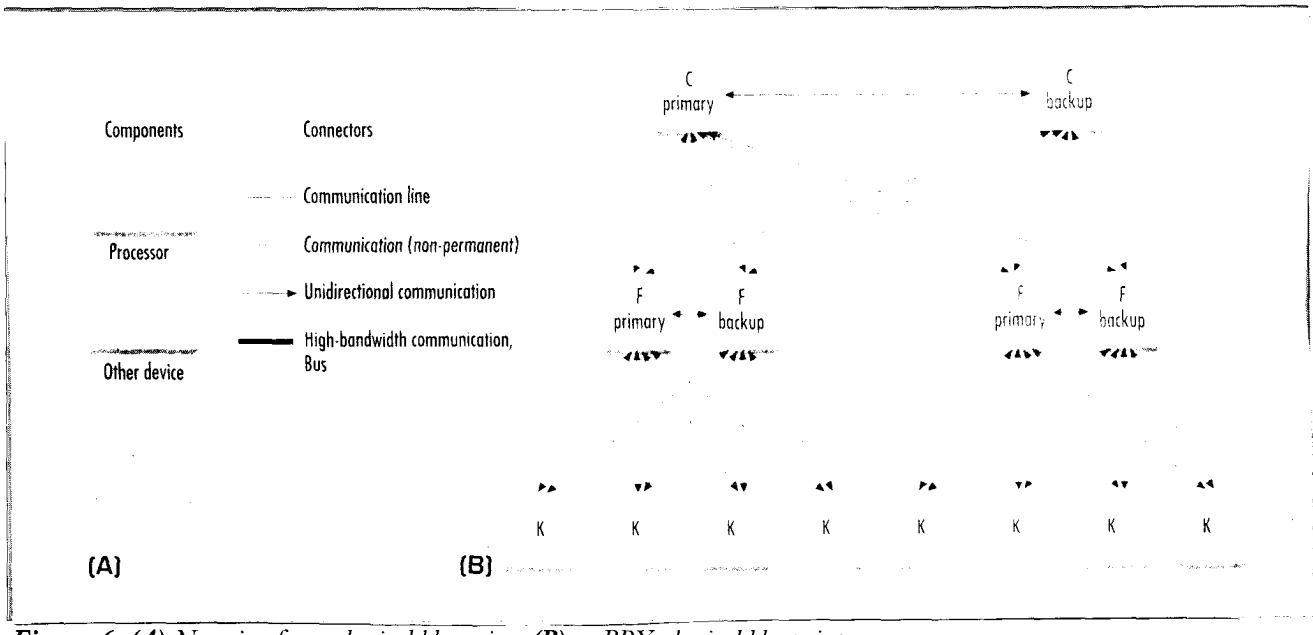

Figure 6. ***(A)*** *Notation for a physical blueprint;* ***(B)*** *a PBX physical blueprint.*

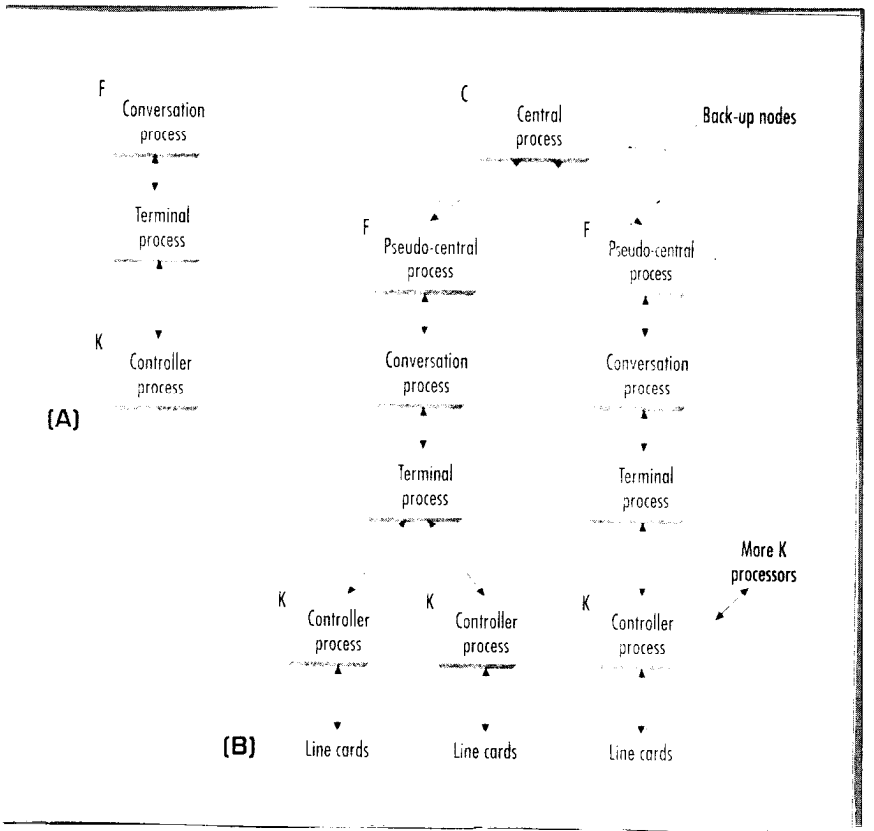

Figure 7. *(A) A small PBX physical view with process allocation; (B) a physical blueprint for a larger PBX; C, F, and K are three types of computers that have different capacities and support three different executables.*

functional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability. The software executes on a network of computers (the processing nodes). The various elements identified in the logical, process, and development views — networks, processes, tasks, and objects — must be mapped onto the various nodes. Several different physical configurations will be used — some for development and testing, others for system deployment at various sites or for different customers. The mapping of the software to the nodes must therefore be highly flexible and have a minimal impact on the source code itself.

Notation. Because physical blueprints can become very messy in large systems, we organize them in several forms, with or without the mapping from the process view, as Figures 6 and 7 show.

UNAS provides us with a data-driven means of mapping the process view onto the physical view. This lets us make many changes to the mapping without modifying the source code.

Figure 6b shows a possible hardware configuration for a large PBX; Figures 7a and 7b show mappings of the process view on two different physical views, a small and a large PBX.

Scenarios. We use a small subset of important scenarios — instances of use cases — to show that the elements of the four views work together seamlessly. For each scenario, we describe the corresponding *scripts* (sequences of interactions between objects and between processes) as described by Ken Rubin and Adele Goldberg.[9] The scenarios are in some sense an abstraction of the most important requirements. Their design is expressed using object-scenario and object-interaction diagrams.[5]

This view is redundant with the other ones (hence the "+1"), but it plays two critical roles:

- it acts as a driver to help designers discover architectural elements during the architecture design, and
- it validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

Notation. The scenario notation is very similar to that used for the logical view, except that it uses the connectors from the process view to indicate object interactions. As for the logical view, we manage object-scenario diagrams using Rational Rose. Figure 8 shows a fragment of a scenario for the small PBX. The corresponding script reads:

1. The controller of Joe's phone detects and validates the transition from on-hook to off-hook and sends a message to wake the corresponding terminal object.
2. The terminal allocates some resources and tells the controller to emit a 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 AMONG VIEWS

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

From logical view to process view. We identify several important characteristics of the logical view classes: autonomy, persistence, subordination and distribution.

Autonomy identifies whether objects are active, passive, or protected. An active object invokes other objects' operations or its own operations, and has full control over other objects invoking its operations. A *passive* object never spontaneously invokes any operations, and has no control over other objects invoking its operations. A *protected* object never invokes spontaneously any operations but arbitrates

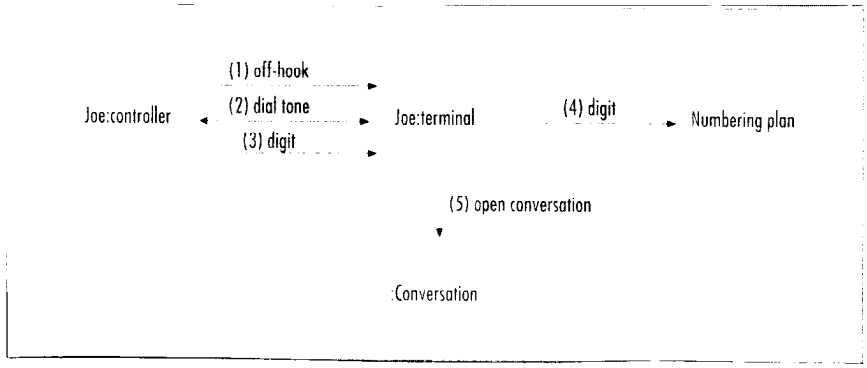

Figure 8. *A scenario example from a local-call selection phase.*

the invocation of its own operations.

Persistence identifies whether objects are transient or permanent. Do they survive the failure of a process or processor? *Subordination* determines if the existence or persistence of an object depends upon another object. *Distribution* determines if the object's state or operations are accessible from many nodes in the physical view and from several processes in the process view.

In the logical view of the architecture, we could consider each object as active and potentially concurrent; that is, behaving in parallel with other objects and paying no more attention to the exact degree of concurrency than it needs to achieve this effect. Hence the logical view takes into account only the requirements' functional aspects.

However, when we define the process view, it is not practical to implement each object with its own thread of control (such as its own Unix process or Ada task) because of the huge overhead this imposes. Moreover, if objects are concurrent, there must be some form of arbitration for invoking their operations.

On the other hand, multiple threads of control are needed to

- react rapidly to certain classes of external stimuli, including time-related events;
- take advantage of multiple CPUs in a node or multiple nodes in a distributed system;
- increase CPU utilization by allocating CPUs to other activities when a thread of control is suspended during another activity (such as access to some external device or access to some other active object);
- prioritize activities (and thus potentially improve responsiveness);
- support system scalability (by having additional processes sharing the load);
- separate concerns between different areas of the software; and
- achieve a higher system availability (with backup processes).

Determining concurrency. We use two strategies simultaneously to determine the "right" amount of concurrency and define the set of necessary processes. Keeping in mind the set of potential physical target views, we can proceed either from the inside out or the outside in.

- *Inside out.* Starting from the logical view, we define agent tasks that multiplex a single thread of control across multiple active objects of a given class. We execute subordinate objects on the same agent as their parent. Classes that must be executed in mutual exclusion, or that require a minimal amount of processing share a single agent. This clustering proceeds until we have reduced the processes to a small number that still allows distribution and use of the physical resources.
- *Outside in.* Starting with the physical view, we identify external stimuli (requests) to the system, and then define client processes to handle the stimuli and server processes that provide (rather than initiate) services. We use the problem's data integrity and serialization constraints to define the right set of servers and allocate objects to the client and servers agents. We then 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 view. Typically, there is an *agent task* for an active class, with some variations, such as several agents for a given class to increase throughput or several classes mapped onto a single agent either to assure sequential execution or because the class operations are infrequently invoked.

Finally, this is not a linear, deterministic process leading to an optimal process view; it requires a few iterations to reach an acceptable compromise. There are numerous other ways to proceed.[8,10]

Example. The exact method used to construct the mapping is complex. However, a brief example from a hypothetical air-traffic control system can illustrate it. Figure 9 shows how a small set of classes from the system can be mapped onto processes.

The *flight class* is mapped onto a set of *flight agents* that must quickly process many flights and spread the load across multiple CPUs while contending with large numbers of external stimuli. The persistence and distribution aspects of the flight processing are deferred to a *flight server*, which is duplicated to assure system availability. *Flight profile* or *flight clearance* is 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 display and external interfaces.

A *sectorization class* establishes a partitioning of airspace to assign controller jurisdiction over flights. Because of its integrity constraints, this class must be handled by a single agent, but it can share the server process with the flight, as updates are infrequent. Locations, airspace, and other static aeronautical information are protected objects, shared among several classes. These are rarely updated and mapped on their own server and distributed to other processes.

From logical view to development view. A class is usually implemented as a module, and large classes are decomposed into multiple packages. Collections of closely related classes — class categories — are grouped into subsystems.

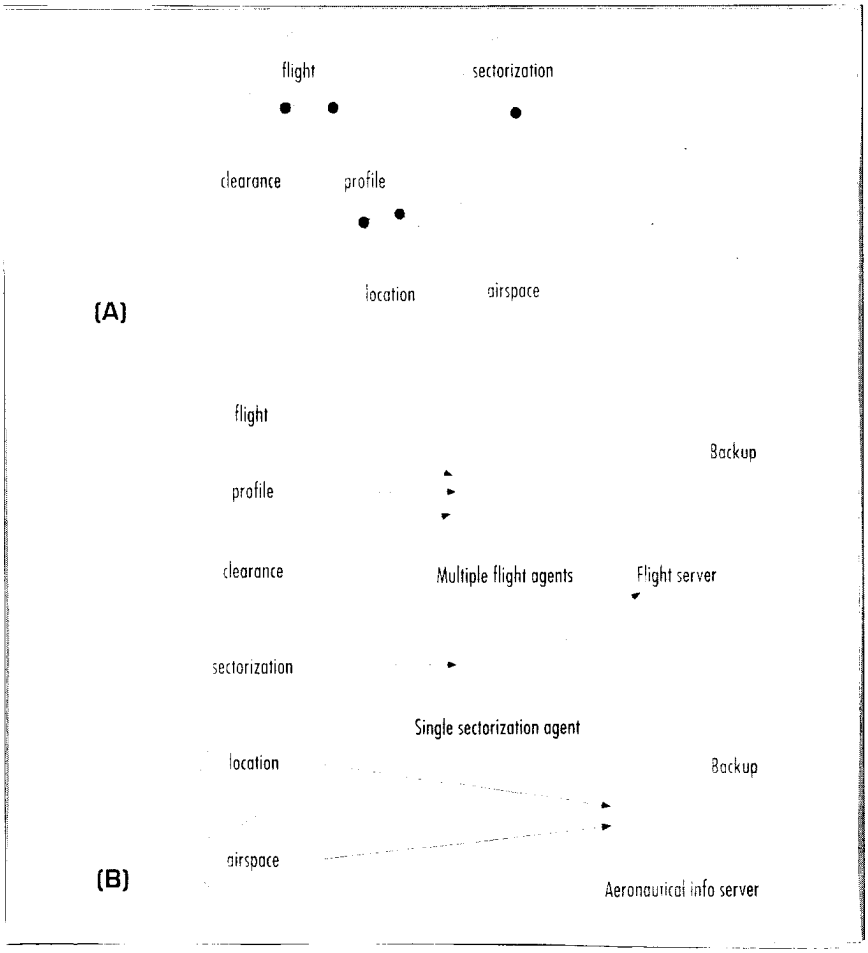

Figure 9. Mapping from the (A) logical to the (B) process view.

To define subsystems, we must consider additional constraints, such as team organization, expected magnitude of code (typically 5,000 to 20,000 lines of code per subsystem), degree of expected reuse and commonality, as well as strict layering principles (visibility issues), release policy, and configuration management. Thus, we usually end up with a view that does not have a one-to-one correspondence with the logical view.

General issues. 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. This also holds for the process and physical views. For example, comparing Figure 2c with Figure 5, there is no one-to-one mapping from the class categories to the layers. The External Interface/Gateway category 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 are in layer 5 subsystems.

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.[8]

In terms of which classes are used, scenarios relate mainly to the logical view, or to the process view when interactions between objects involve more than one thread of control.

ITERATIVE PROCESS

Bernard Witt and his colleagues describe four phases for architectural design — sketching, organizing, specifying, and optimizing — and subdivide them into 12 steps.[10] Although they do indicate that some backtracking may be needed, we think their approach is too linear for ambitious or unprecedented projects, because too little is known at the end of the phases to validate the architecture. We advocate a more iterative development, in which the architecture is actually prototyped, tested, measured, and analyzed, and then refined in subsequent iterations.

Our approach not only mitigates the risks associated with the architecture, it also helps build teams and improves training, architecture familiarity, tool acquisition, the initial run-in period for procedures and tools, and so on. (This holds for evolutionary, rather than throwaway prototypes.) An iterative approach also helps you refine and better understand the requirements.

Scenario-driven approach. Scenarios capture the system's critical functionality — functions that are the most important, are used most frequently, or present significant technical risk.

To begin, select a few scenarios on the basis of risk and criticality. You may synthesize a scenario by abstracting several user requirements. Then create a strawman architecture and script the scenarios, identifying major abstractions (such as classes, mechanisms, processes, subsystems)[9] and decomposing them into sequences of pairs (object, operation).

Next, organize the architectural elements into the four views, implement the architecture, test it, and measure it. This analysis helps you detect flaws or potential enhancements. Finally, capture lessons learned.

Begin the next iteration by reassessing the risks, extending the scenarios to consider, and selecting a few additional scenarios on the basis of risk or extending architecture coverage. Then try to script those scenarios in the preliminary architecture and discover additional architectural elements — or significant architectural changes — that must occur to accommodate these scenarios. Update the four views and revise the existing scenarios on the basis of these changes. Next, upgrade the implementation (the architectural

prototype) to support the new extended set of scenarios.

At this point, you should test the architecture by measuring under load (in the target environment, if possible) and review all five views to detect potential simplifications, commonalities, and opportunities for reuse. Then update the design guidelines and rationale and capture lessons learned. And then loop again.

Finally, the initial architectural prototype evolves to become the real system. After two or three iterations, the architecture itself should become stable, and you should find no new major abstractions, subsystems, processes, or interfaces. The rest is in the realm of software design — where you can continue development using very similar methods and process.

Timetable. The duration of these iterations varies considerably, depending on the size of the project, the number of people involved, and their expertise in the domain and the development method. It also varies relative to the development organization. Hence the iteration may last two to three weeks for a small project (10,000 lines of code), or from six to nine months for a large command-and-control system (700,000 lines of code or larger).

Tailoring the model. Not all software architectures need every view in the 4+1 View Model. Views that are useless can be omitted. For example, you could eliminate the physical view if there is only one processor or the process view if there is only one process or program. For very small systems, logical and development views are sometimes so similar that they can be described together. The scenarios are useful in all circumstances.

Documentation. The documentation produced during the architectural design is captured in two documents:

- a software architecture document, organized by the 4+1 views, and
- a software design guideline, which captures (among other things) important design decisions that must be respected to maintain the architectural integrity of the system.

We have used the 4+1 View Model on several large projects, customizing it and adjusting the terminology somewhat.5 We have found that the model actually allows the various stakeholders to find what they need in the software architecture. System engineers approach it first from the physical view, then the process view; end users, customers, and data specialists approach it from the logical view; and project managers and software-configuration staff members approach it from the development view.

Other sets of views have been proposed and discussed at our company and elsewhere, but we have found that proposed views can usually be folded into one of the four existing views. A cost and schedule view, for example, folds into the development view, a data view into the logical view, and an execution view into a combination of the process and physical view. ◆

ACKNOWLEDGMENTS

For their help in shaping or experimenting with the 4+1 View Model I thank my many colleagues at Rational, Hughes Aircraft of Canada, CelsiusTech AB, Alcatel, and elsewhere, and in particular, Chris Thompson, Alex Bell, Mike Devlin, Grady Booch, Walker Royce, Joe Marasco, Rich Reitman, Viktor Ohnjec, Ulf Olson, and Ed Schonberg.

REFERENCES

1. D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing Co., Singapore, 1993.
2. G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," *ACM Software Eng. Notes*, Dec. 1993, pp. 9-20.
3. Paul Clements, "From Domain Model to Architectures," A. Abd-Allah et al., eds., *Focused Workshop on Software Architecture*, 1994, pp. 404-420.
4. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Eng. Notes*, Oct. 1992, pp. 40-52.
5. G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd. ed., Benjamin-Cummings, Redwood City, Calif., 1993.
6. P. Kruchten and C. Thompson, "An Object-Oriented, Distributed Architecture for Large Scale Ada Systems," *Proc. TRI-Ada '94*, ACM Press, New York, 1994, pp. 262-271.
7. A. Filarey et al., "Software First: Applying Ada Megaprogramming Technology to Target Platform Selection Trades," *Proc. TRI-Ada '93*, ACM Press, New York, 1993.
8. K.P. Birman and R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE CS Press, Los Alamitos, Calif. 1994.
9. K. Rubin and A. Goldberg, "Object Behavior Analysis," *Comm. ACM*, Sept. 1992, pp. 48-62.
10. B. I. Witt, F. T. Baker, and E.W. Merritt, *Software Architecture and Design Principles, Models, and Methods*, Van Nostrand Reinholt, New York, 1994.

Philippe Kruchten is a senior technical consultant at Rational Software, where he is in charge of the Software Architecture Practice area. Kruchten has 20 years experience in software development. He has been associated with several large-scale software-intensive projects around the world, including the Alcatel 2505 and Alcatel 2600 private telephone exchanges in France, the Ship System 2000 command-and-control system in Sweden, and several other projects in avionics, defense, transportation, and compilation. Since August 1992, he has been the lead software architect for the Canadian Automated Air Traffic System, developed by Hughes Aircraft of Canada in Vancouver.

Kruchten received an M.Sc. in mechanical engineering from Ecole Centrale de Lyon, France, and a PhD in information technology from the French National Institute of Telecommunications, Paris. He is a member of the IEEE Computer Society and the ACM.

Address questions about this article to Kruchten at Rational Software Corp., 240-10711 Cambie Rd., Richmond BC V6X 3G5; pkruchten@rational.com