Using UML Activity Diagrams for the Process View

by **Ben Lieberman**
Senior Software Architect
Blueprint Technologies

In the article "UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior," published last month in The Rational Edge, *I discussed ways that system architects and designers can use UML Activity Diagrams to detail use cases, capturing basic, alternate, and exceptional flows of execution. In this month's article, the focus shifts to a particular view in the 4+1 Architecture Views, defined by the Rational Unified Process.™ We will examine how to use Activity Diagrams as "roadmaps" for the Process View, to capture processing flows as a series of steps. We will also discuss several techniques for creating these diagrams and ensuring their effectiveness.*

Highly complex software systems are inherently difficult to construct and maintain. These challenges are often further complicated by a lack of comprehensive documentation explaining the responsibilities and dependencies for each system module. Then, to add insult to injury is the current industry drive to create reusable, modular, distributed systems (e.g., EJB, CORBA, COM+). These forces, in combination with an increasing need for more and more complex processing, can lead to fragile, non-maintainable systems if the architect does not find an effective way to describe and model critical system dependencies.

Embedded and real-time systems, for example, are subject to these pressures. Often they have tight time and resource constraints, which places a high premium on the efficient use of system resources. For real-time applications, process timing is often critical (e.g., the system must acquire and display a target in under .05 seconds) which may require parallel processing or other high-performance mechanisms. Likewise, embedded systems have steep performance requirements and must make maximum use of available system memory and processor power since these systems often run on dedicated hardware. Finally, the need to meet

these performance requirements creates increased demand for distributed systems that are deployed to 1+N processors. These deployments require that the architect be able to rapidly determine the exact functionality that is distributed to each processor and the control mechanisms that must be implemented for synchronization of parallel or asynchronous processing. Developers of such systems would therefore benefit from an accurate and complete description of system processing.

The 4+1 Architecture Views[1] present a set of models useful for describing software systems. Of these, the *Process View* is often used for modeling the processes and threads executing in a software system. Descriptions of the Process View have often focused only on the static structure of interacting processes and lightweight threads using UML stereotyped class diagrams.[2] This article presents the counterpart Dynamic View to the static class diagram representation. As will be shown, UML Activity Diagrams are used to capture the "roadmap" of the processing flow as a series of steps. The astute reader will note that this technique is very similar to flow-chart modeling familiar to experienced software developers. The primary difference here is the level of abstraction used to focus attention on critical processing details while hiding much of the underlying code complexity. These diagrams quickly capture and summarize the system dependencies at a processing level, leading to a better architectural definition of the system.

Note, however, that this technique violates some of the defined semantics for Activity Diagrams in the UML standard: most notably the requirement for all activity transitions to flow from entry to exit. In the example diagrams this requirement has been relaxed to better display the parallel nature of threads and processes.

Overview of the Process View

Processes are usually necessary to run executable computer programs. These processes are created by the operating system and usually run in a dedicated memory space with priority usage of the processor(s). In addition, "threads" are often required by many applications, particularly for batch or asynchronous processing. As we have already noted, the 4+1 views of architecture describe the overall system, focusing on the Logical, Implementation, Deployment, and Process Views. The last of these views, the Process View, is particularly well suited for describing the processes and threads that comprise the system.

Processes are heavyweight system run-time entities. A process is expensive to establish and requires time, memory, and CPU resources. Processes can communicate using various mechanisms provided by the host operating system; this is usually referred to as interprocess communication (IPC). Often this communication requires synchronization so that one process will not corrupt the state of another, particularly when using shared resources.

In many modern operating systems, processes are permitted to spawn lightweight executing "threads" that operate within the memory space of the running process. These threads can either operate independently or

communicate with one another. Threads often need to coordinate the synchronization of data or processing between other threads and the host process. The primary value of threads is that they reduce system resource costs, since a thread shares the resources assigned to the parent process. A real-time system often needs to use threads to perform parallel processing that is coordinated by another thread or the host process itself.

The Process View is defined as the static structure of threads and processes (i.e., ownership, control, coordination), and the dynamic aspects of their interactions (i.e., synchronization, communication, timing). The static nature of the system processing is often shown using stereotyped UML diagrams (Figure 4). Although the dynamic aspect is not often discussed and few examples exist, Activity Diagrams are particularly well suited to the description of process and thread interactions. Figure 1 shows the symbols and extensions used for UML Activity Diagrams as well as three stereotyped "Activities" useful for describing dynamic processing. We'll review these below.

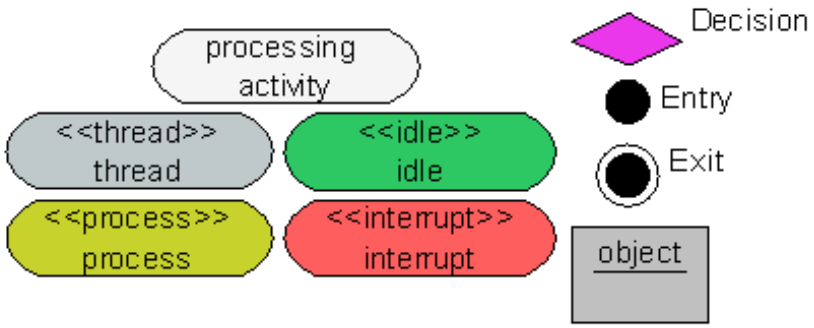

Figure 1: Activity Diagram Symbols and Extensions

Review of Activity Diagram Symbols and Extensions

Processing Activity. *Activity Diagram symbols encapsulate activities performed by the system. These activities may have associated entry conditions, exit conditions, and actions. The simplest and most useful form, however, is the one shown in Figure 2; it represents the functionality to be performed.* Processing activities the most numerous Activity Diagram components, represent the actions taken by the system. As Figure 1 shows, the symbol for these activities is a rounded-off rectangular shape with a neutral color. Please note that the color is intended to add meaning and understandability to the diagram. Addition of too much color detracts from the simplicity of the approach; not enough color leads to bland, uninteresting diagrams. See the figures in this article for guidance in selecting diagram element colors. Also, the Appendix to this article contains a script that can be used to automatically add color to Activity Diagrams in a Rose model.

In addition to the standard symbols provided by the UML, this article introduces several extensions that permit additional information to be conveyed for special system activities.

<<interrupt>> Stereotype. Batch processing must allow for interruptions of the processing flow for input of information and control of the system. When this stereotype is applied to the standard activity element, it indicates that the activity will generate a system interrupt to the current processing. These elements are colored red to draw particular attention to this type of critical system interaction. Interrupts can occur on both processes and threads if allowed by the deployment operating system.

<<idle>> Stereotype. All batch processing activities, user interface activities, and asynchronous communications require the ability of the process or thread to go into an idle state. While in this state, the process/thread requires limited processor time and can permit other system activities to proceed. The <<idle>> stereotype is applied to activities that will enter into this "wait" state.

<<thread>> Stereotype. Many processes, particularly those involved in complicated batch processing, will spawn lightweight processes that run within the parent processing memory space. These are referred to as process threads. The <<thread>> stereotype is used to indicate that an activity is related to the creation, suspension, resumption, or destruction of a system thread.

<<process>> Stereotype. All computer software is composed of processes that are loaded into and run on a hardware CPU. The <<process>> stereotype is used to indicate an activity that will result in the creation, suspension, resumption, or destruction of a system process.

Creating Process View Activity Diagrams

Determining the correct level of abstraction in modeling is always a challenge. This is especially true when creating models for the Process View. Since this system description information is critical to multiple team members, including testers, developers, managers, and system administrators, there is a high premium placed on the clarity of these models. Countering this force is the high level of complexity inherent in parallel, asynchronous processing. How should we capture this information at a level that is most useful to the largest audience? Perhaps more important, who should be responsible for the creation and maintenance of these models?

Process level information is a design activity and as such is under the control of the architect and component engineer. Thus, the creation of Process View Activity Diagrams most naturally falls to these roles. However, since the consumers of these models include the test team and deployment engineers, it is often beneficial for a member of one of these other teams to act in the role of analyst. An individual who is not directly involved in the system design but is technically knowledgeable (such as a technical tester) can capture the model information from the design team at a level of abstraction more appropriate to a wide audience. The level of detail required by the model and the expected stakeholders will heavily influence the decision as to which team member should perform the modeling.

There are two cases that require system modeling: design of new systems and reengineering of existing systems. The techniques for each case are sufficiently different to warrant separate discussion.

Modeling a New System

For new systems, system process definition (i.e., the Process View) occurs during the Analysis and Design workflow. Following the Rational Unified Process, the Architect and the Designer perform this activity. The usual progression in the creation of these models is to find the core processes (as defined by the static model), then follow with a design session to outline the processing flows.

As shown in the Figure 2, the system under design is intended to use standalone threads listening on a port for a message. The primary process spawns two threads: a BroadcastThread and a monitor thread called a ThreadWatcher. The ThreadWatcher is responsible for a keep-alive function; it periodically checks to see if the Broadcast thread is still functional. The interval for the checks is defined by the guard condition on the transition from the idle state. Once a message is detected, the process simply passes the message to the BroadcastThread, which attempts to send the message to other registered listeners (shown as a network object using TCP/IP protocol). If there is a failure to broadcast, then a log (shown here as a file object, but any kind of design mechanism can be defined) is written. Notice the use of guard conditions to indicate the transition coordination, and that each processing path leads back to the idle state. For this example, each thread will continue indefinitely until an external interrupt is generated to kill the process and associated threads.

Also of note is the use of Swimlanes to indicate the threads and processes. While it may seem redundant to use Swimlanes named for the process and threads when there is a specific activity to represent them, remember that the intent of the process/thread activity is to show when a process/thread is created, controlled, or destroyed. The Swimlanes are intended to show the parallel processing for each zone of control.

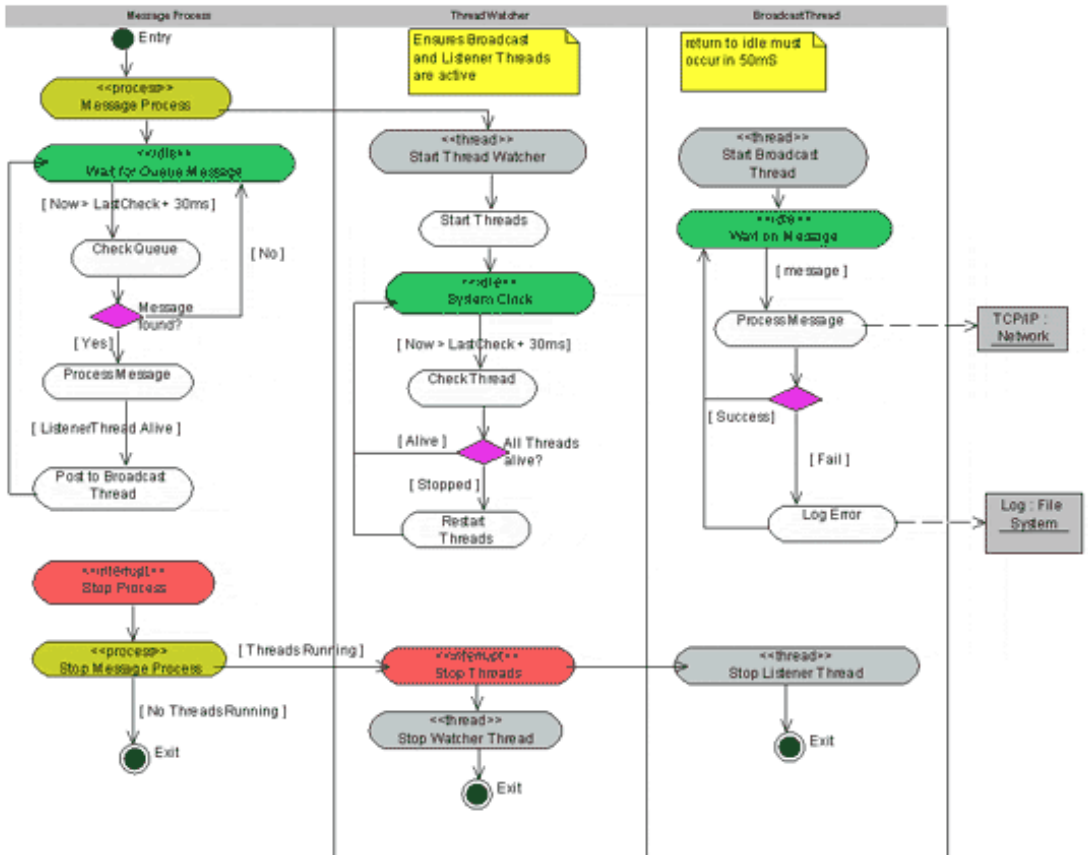

Figure 2: Using Swimlanes to Describe Processes and Threads
Click here for full size image

Modeling an Existing System

One of the most effective techniques for eliciting a model for an existing system is to conduct a personal interview with the subject matter experts (SMEs).[3] This works best in small group settings (three people or fewer), with a facilitator who creates the model directly into a modeling tool. This allows for the rapid capture and direct presentation of the model to the SMEs for feedback. In my experience, the best mechanism for this is to attach a projector to the facilitator's computer and project the model onto a large screen that is easily visible to all participants. Alternatively, the group can gather around a large monitor.

Begin the interview by asking where the processing begins, then capture the response as the entry point into the diagram. If the existing processes and threads are known, then they can be captured as Swimlanes at this point. Otherwise, they will be discovered as the information capture progresses. After you locate the entry point, proceed with the interview by asking what the steps are for the system processing. Decision points are of particular interest, as they represent areas of the system where branching will occur. Data important to the process flow should be captured in notes attached to the related activities. Interrupts affecting processing should be indicated where they occur. Figures 3, 4, and 5 show an example of how a distributed travel reservation system can be

diagrammed for the Deployment View, static Process View, and dynamic Process View, respectively.

The Deployment View diagram (Figure 3) presents a hypothetical, "pre-existing" travel booking system. For this example we are assuming a "Web services"-based architecture in which most of the system is accessed via a distributed mechanism (e.g., via CORBA or SOAP type interfaces). For the ticket fulfillment service we are proposing a design containing a core process and between two and five associated threads (one for the printer output and up to three for the printer queue; Figure 4). Within the Ticketing Service Activity diagram (Figure 5), we can see the interaction between each of the different threads during a request for processing. Here the main process is responsible for polling against an MQSeries message queue for printing requests. When a request is found, it is parsed and forwarded to the PrinterOutput thread. Based on the priority of the job, the thread will choose to interrupt a current job or queue the request to wait for a free printer. The PrintQueue thread(s) will check for the first available printer for processing the request. If an available printer is found, then the job is resubmitted to the PrinterOutput thread for printing. Not shown in this diagram is the handling of errors, but a technique similar to the one shown in Figure 2 can be used here to indicate the handling of exceptional conditions. Upon program termination (again shown here as an interrupt operation), each thread will perform additional activities to free system resources (e.g., Disconnect from Network Printer and Close Print Queue).

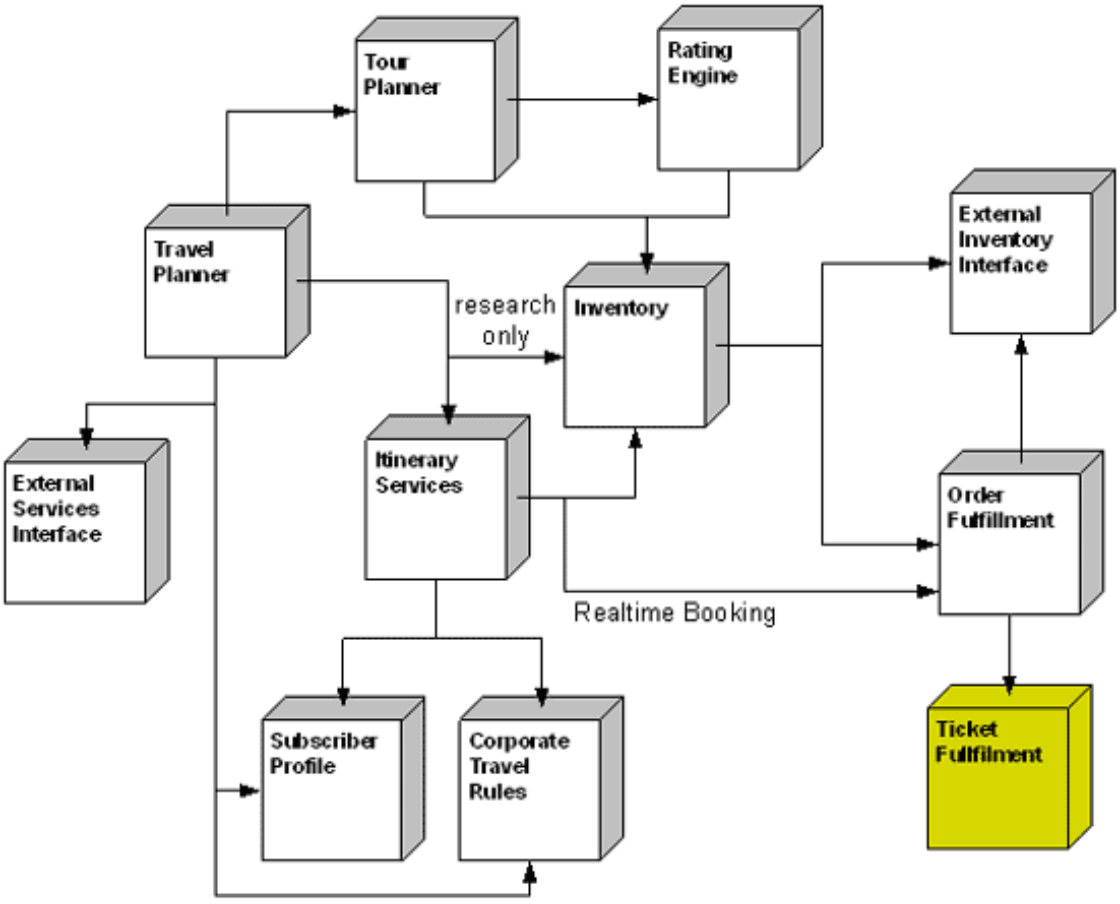

Figure 3: Travel Planning System Deployment View Diagram

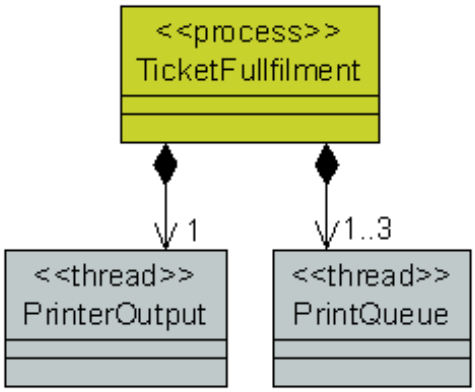

Figure 4: Travel Planning System, Static Process View, Process Class Diagram (Ticket Fulfillment Subsystem)

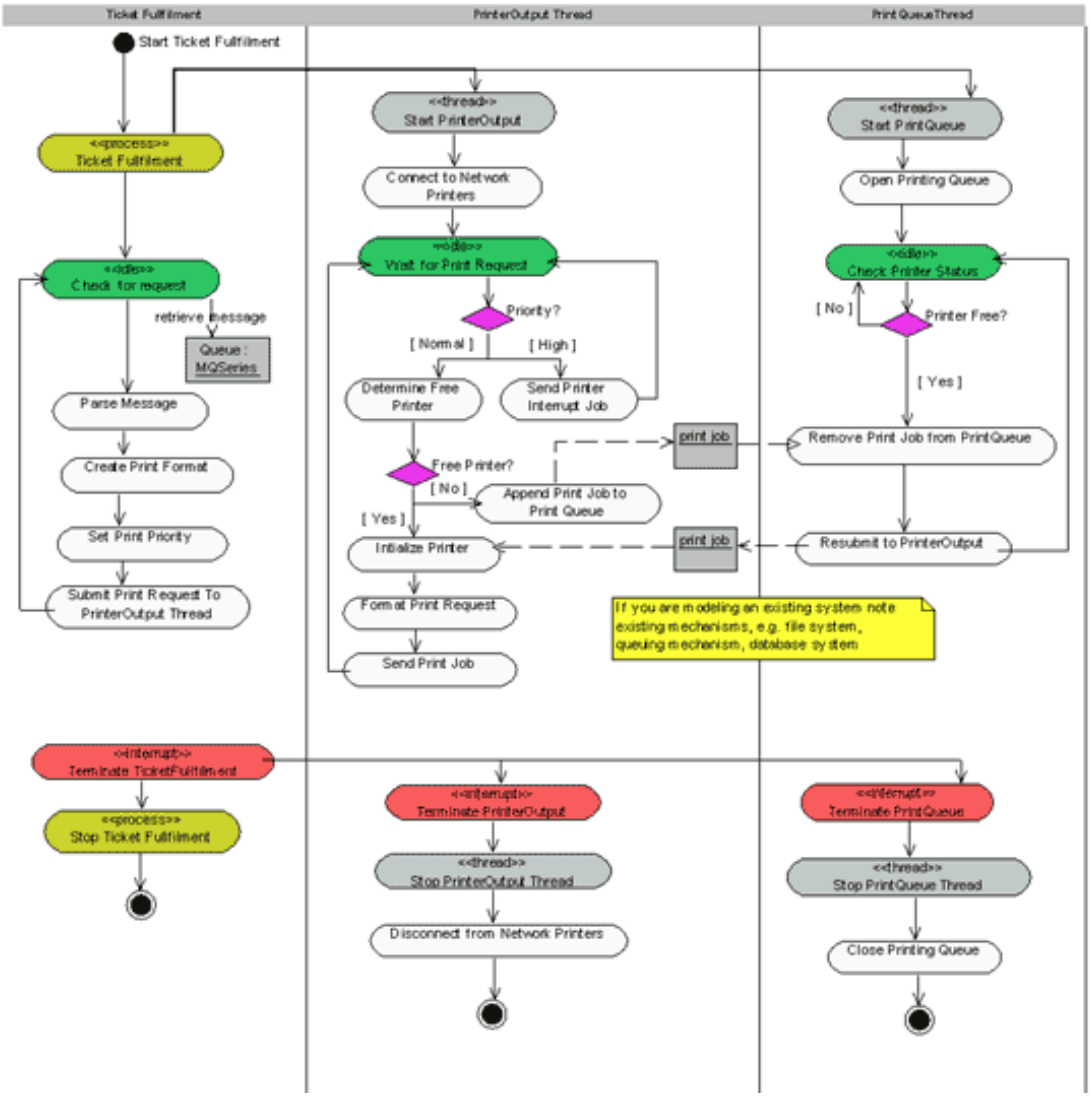

Figure 5: Travel Planning System, Model of Existing System Elements for the Dynamic Process View (Ticket Fulfillment Subsystem)
[Click here for full size image]

Technique Use and Abuse

As with any modeling method, there is a high potential for misuse that can limit or eliminate the effectiveness of the technique. When using Activity Diagrams for modeling system processing, it is important to capture the initial information at a relatively high level of abstraction. This top-down approach allows you to emphasize important details while hiding programmatic ones. If you get trapped in the low-level processing details (e.g., algorithmic flow-charting), then the diagram will become cluttered with programmatic activities that obscure the model's true purpose: to indicate key processing activities and dependencies. Low-level details that can seriously impact the clarity and usefulness of these models include the description of complex algorithms, specific steps in persistence operations, system-level connection and communication details, and other such specifics of operating system activities. For the interviewing process described above to be successful, the facilitator should be practiced at finding appropriate levels of abstraction and be on guard against programmatic details entering into the model.

A High-Value Technique

The Process View, as defined in the 4+1 Architecture Views, is intended to capture and communicate the system processes and threads. This article presents a technique for capturing the key processing steps for system processes/threads and shows the utility of such models during the software development process. *These models are not simple to create*, but they do represent complex system processing in a "clean," efficient way. By using Activity Diagrams to model internal system processing steps, the test team, architect, and system designer can achieve an identical understanding of the key processing steps and subsystem dependencies present in the software design. For complex systems such as embedded, real-time, or distributed systems, this kind of understanding amongst team members can be of immense value: It ensures that the system the team creates can be maintained and extended, and that it meets the needs of the customer.

Appendix - Rational Rose™ Activity Diagram "Colorizer" Script

This Rational Rose™ script (for version 2000 and greater) will automatically add diagram element fill color to the Activity Views on each Activity Diagram included in the use case model (e.g., under the Logical View root package).

References

G. Booch, J. Rumbaugh, et al., *The Unified Modeling Language, User Guide*. Reading, Massachusetts: Addison Wesley, 1999.

A. Cockburn, "Goals and Use Cases." *Journal of Object Oriented Design*, September 1997: 35-40.

P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, 12 (6), November 1995, IEEE, pp. 42-50.

D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach*. Boston: Addison Wesley, 2000.

J. Rumbaugh, I. Jacobson, et al., *The Unified Modeling Language, Reference Manual*. Reading, Massachusetts: Addison Wesley, 1999.

---

[1] See P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, 12 (6), November 1995, IEEE, pp. 42-50

[2] G. Booch, J. Rumbaugh, et al. *The Unified Modeling Language, User Guide*. Reading, Massachusetts, Addison-Wesley,1999 and J. Rumbaugh, I. Jacobson, et al, *The Unified Modeling Language, Reference Manual*. Reading, Massachusetts, Addison Wesley, 1999.

[3] D. Leffingwell and D, Widrig *Managing Software Requirements: A Unified Approach*. Boston: Addison-Wesley, 2000 and A. Cockburn, "Goals and Use Cases." *Journal of Object Oriented Design*, September 1997: 35-40.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!