A Four-in-One View of Architecture

Abstract

In "The 4+1 View Model of Architecture" Philippe Kruchten states the limitations of capturing the gist of an entire system in a single diagram [Kruchten95]. Any non-trivial system requires significant and diverse descriptions to frame concerns of all relevant stakeholders. To capture and convey these details, our architecture development process follows well-accepted methods, including architectural "views." This paper describes how we evolved four commonly-used views into a comprehensive diagram to effectively communicate the essence of a very complicated system in one "Big Picture."

Keywords: architecture frameworks, architecture viewpoints

Developing complex architecture involves a continual and delicate trade-off between technical alternatives of system style, organization, operation, constraints, cost, performance, and other factors. Throughout this process the architect and architecture team must solicit inputs and address concerns of subject matter experts, organizations, and customers. A primary challenge is representing the nascent system in an unambiguous, easily-digestible way so all parties share a common understanding of the proposed solution. To this end, our profession has adopted architectural "views" to represent key architecture-related aspects of a system.

The concept of using views to document systems architecture was introduced by Zachman in 1987 [Zachman87], and evolved as a proven world-wide standard adopted by the IEEE [IEEE00] and International Standards Organization [ISO09]. (For an excellent history of software architecture, see the guest editor's introduction to an IEEE Software issue on the topic [Kruchten 2006].) These standards codify relationships between a system, affected stakeholders, system architecture and description, and rules mapping these items to each other. For purposes of this paper, we present each view as a diagram capturing stakeholders' concerns about a system.

When describing architecture, no predefined limit to the numbers of views exists, nor does a predefined set of views that must be used. For example, Kruchten's model uses four views: logical (system object model), process (design concurrency and synchronization), physical (mapping of software onto hardware), and development (software static organization). These views capture the essence of most concerns, and nearly every system description we see includes, at minimum, views identical with, or similar to, these four.

For each view, the architect might use a variety of design methods and discipline-specific diagramming notations to capture stakeholder concerns. Methods and notations guide us

on what to include and exclude, and when to divide a complex diagram into simpler ones to clarify key points. The architect continually struggles with how to best represent the system and effectively communicate its essence to a diverse audience. In most cases, Kruchten observes that architects tend to represent more in one diagram than is practical, resulting in confusing diagrams that fail to provide an unambiguous reference for long-term projects.

In our case, our architectural description evolved while working on a large, complex system comprised of over 400,000 lines of (mostly) C code deployed for a mission critical control system. We developed the architecture using well-established methods, which resulted in numerous operational scenarios, a context diagram, and four architectural views. However, in time we migrated to a unified view of our system described below.

Our "four-in-one" diagram faithfully served developers, management, and customer as a single unified reference over six-years of development and deployment. Since then, we used this approach on many other projects. While we concede limitations of including too much information in one all-encompassing view, others may find our experience useful for their teams and customers because of the benefits offered by a common reference.

Developing an Architecture

Although this paper assumes the reader is familiar with architecture development, we describe our experience in part to normalize terminology and to open our story to a wider audience. The core of our process centers on classic software and systems engineering principles as taught and practiced by our organization and others. The entire development cycle on a large project often takes several years, typically beginning with a thorough review of customer requirements and design options as we discuss potential approaches to meeting requirements.

Establishing Scope

When defining requirements and establishing preliminary design, architects must clarify what lies within the system solution and what doesn't. The *Context Diagram* captures this information with a box around the included solution space, excluding all agents and systems outside the project scope. We place any point of interaction with the outside world on the box's boundary; these interaction points eventually become Interface Control Documents (ICD) detailing the system's interface with a corresponding external system.

Within the contextual boundary, we develop an initial Concept of Operations (our *ConOps*) for how we and the customer want the system to act in various situations. For each situation we create a Use Case scenario analogous to the "+1" in Kruchten's famous paper. We find that the ConOps helps define system functions implied by customer requirements, and bridges the system organization with its operation. For reference, our

ConOps had 10 baseline scenarios with 37 specialized (e.g., exception processing) scenarios.

Selecting Architectural Views

In our organization, three specific views are prescribed by our System Architect Certification program and taught to all aspiring architects as follows:

1. **Logical View**: Allocates requirements and functions to software and systems components
2. **Operational View**: Shows the system's dynamic behavior in terms of data and control flow
3. **Physical View:** Maps software components to possibly distributed physical devices, including processors, storage and routers, and interconnecting networks.

In addition, because of the data-intensive nature of the large software systems with which we work, we almost always include:

4. **Data View:** Describes the key data stores and relationships present in the system.

Depending on circumstances and customer, we also use a variety of views such as:

- **Security View:** Defines protections and access controls
- **Tool View:** Shows tool selection and applications
- **Standards View:** Defines protocols and versions of technical standards and COTS products.

Although our organization allows us to tailor view selection, this method of documenting architectures is a mandatory checklist item during technical reviews conducted at key milestones throughout development. We note that the number of views we use, even on large (> $500 million) programs, is usually less than five. For this program, we selected the three prescribed views plus a data view.

Iterate and Converge

Developing a system solution requires iterating over many possible solutions, evaluating advantages and disadvantages, and documenting decisions with rationale. For ease of access, we recorded our progress and objective evidence in a shared workspace on the project website, including meeting minutes and solutions to problems.

One key decision was regarding fundamental architectural style [Garlan96]. Because our system has near real-time performance requirements and is physically distributed over almost 100 locations nationwide, we determined that the object-oriented architectural style was most appropriate. Another key decision allocated high-performance functions to software versus hardware. Each trade-off and decision rationale found its way into the on-line project development folder.

To reduce risk and cost, we depend on our experience from prior related programs. Successful reuse not only with architecture but also the code level depends on domain experience [Poulin97]. In our case, we had fielded three similar systems for at least two other major customers. These systems established the architectural patterns for our system and constrained the solution space during each iteration.

Having established a fundamental architectural approach defined by object-oriented patterns, and having chosen four views to document our system solution, we began work on our system solution and architectural description.

Specifying a Notation

After investigating several specialized architecture development tools, we determined we'd use widely-available office tools to capture our architecture. This was, in part, driven by limited availability of specialized tools and trained engineers across all development organizations, including our suppliers. However, this approach generated significant benefits – engineers focused on engineering, not drawings, and everyone had access to our work products. We merely needed to establish standards and conventions for notation so we all worked from common representations of common concepts.

Standard Colors and Patterns

We started by adopting standard colors and patterns for each logical subsystem and class of external system. As Figure 1 shows, colors convey the subsystem of each component, whereas background patterns assist in color-impaired situations (such as when working solely with black and white hard copies).

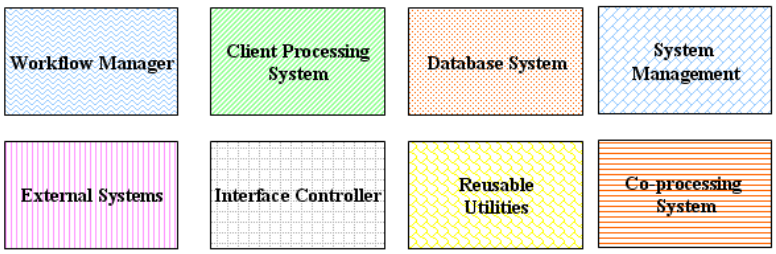

Figure 1- Notation quickly conveys the logical archicture. *Combining colors and patterns reflect functions of a component even when printed in black and white.*

Figure 2 shows additional notations useful for clarity (projects will develop notation that best applies for their situation). We clarified the meaning of connecting lines and arrows simply by labeling the lines as necessary. Once established, success is simply a matter of enforcing consistent terminology and graphic representation across diverse personnel and sub-systems.

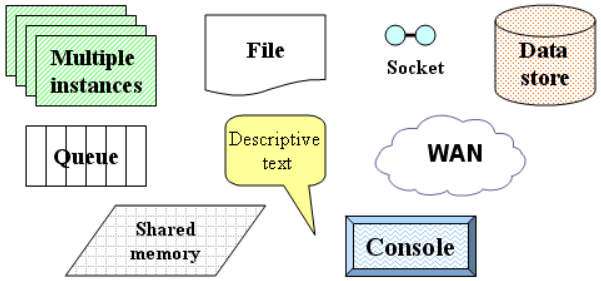

Figure 2- Special symbols for specific features. *Consistent use of standard colors and patterns with intuitive symbols convey key concepts as well as ownership.*

Our Four Views

Whatever views you select for your project, developing views is highly iterative as you develop, refine, and update your overall architecture. The following paragraphs describe the presentation of our four views using our chosen notation. Although there is no prescribed order to developing views, we tend to develop the logical view first. This consists of tentatively grouping the functions defined by our user requirements into candidate subsystems (both procedural and data). We are then able to sketch the operational view to test candidate interactions between the candidate logical subsystems. Finally, we conclude by distributing possible allocations of functions and subsystems to physical devices.

Logical View

This view depicts a system's major logical components and functions allocated to them. As Figure 3 shows, the example Logical View depicts logical components and key connectors [Garlan96] – in this case arrows labeled with primary data flows between components. Unless stated, lines (connectors) in our views represent the flow of data between objects and control over that data. Note that we often label connectors for clarity.

The Logical View is important for program managers and developers because it determines development responsibility and organizational dependencies. These, in turn, become the basis for baselining schedules, budgets, and ultimately earned value monitoring.

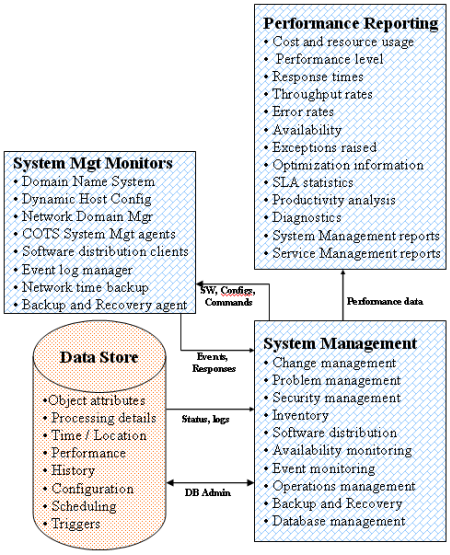

Figure 3- Example Logical View. *A portion of a Logical View using the established notation; blue "brick" components eventually implement System Management functions; the orange "dot" component implements the persistent data store.*

Operational View

As shown in Figure 4, the Operational View of the example system segment conveys not only the logical components but the behavior and interactions between components. Again, the proposed logical subsystems for each component are quickly identifiable by color and pattern. Yellow numerical callouts in this example describe the processing sequence through one of the ten primary scenarios in the ConOps and reference descriptive text in the accompanying System Architecture Document (SAD).

In addition to behavior, Figure 4 also shows boundaries for this system segment. The scope of the segment is defined using standard context diagram notation, shown here with a grey wire frame around objects contained within the segment boundary. Interfaces to other system segments and external systems are depicted by dark grey boxes on the segment boundary (i.e., sitting on the grey wire frame).

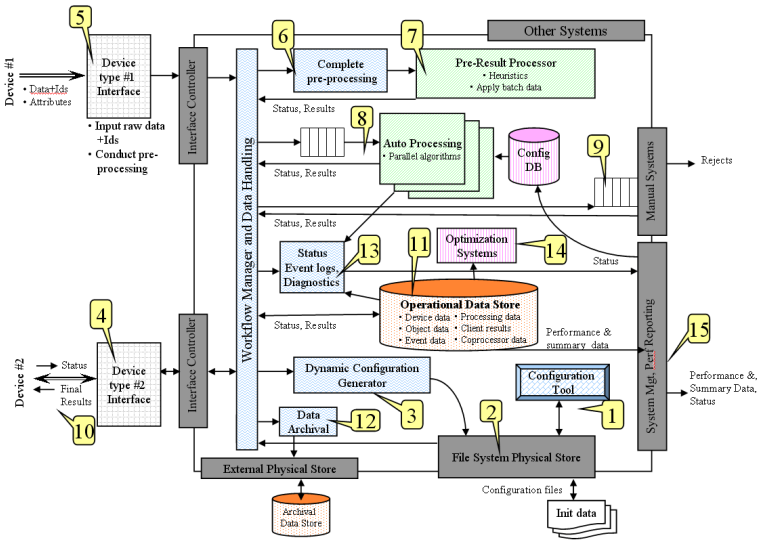

Figure 4- Example Operational View. *A portion of an Operational View using established notation; yellow call-outs reference a textual description of processing steps and data flows at that location, in temporal sequence. Note that subsystem and functional allocations are labeled, or indicated by color and pattern, and the scope of this segment is defined in grey using standard Context Diagram notation.*

Data View

This view contains information about data processing and data stores. We modeled the data processing portion of the Data View using the same method as the rest of the system's procedural components (Figure 3 and Figure 4). As shown in Figure 5, information in the data store portion of the Data View is primarily an entity relationship diagram showing data entities and their detailed data elements, built in and generated by our data modeling tool. At a minimum, understanding data entities within a data store is critical to most system stakeholders.

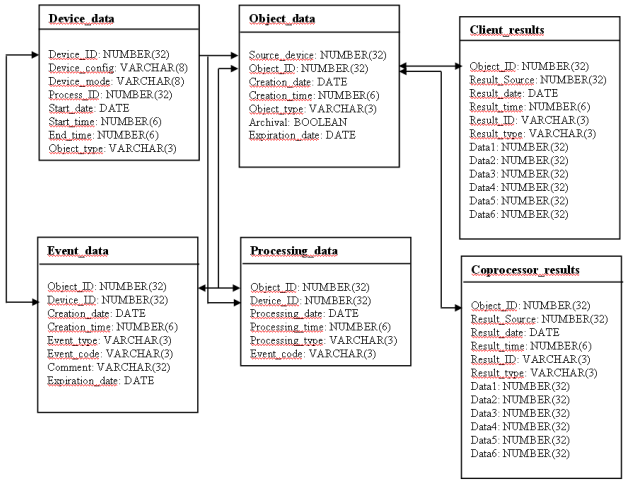

Figure 5- Example Data View. *A portion of a Data View generated from our database modeling tool depicts major data stores; these map to physical tables in a relational database. We modeled procedural components of the Data View (business logic) using the same methods and notations used for the other procedural components of the system.*

Physical View

This view maps software components and remaining functions to the hardware that will either run the software or implement the function. To ensure completeness, we cross-reference each customer and derived requirement in our COTS requirements database to functions, then to software components, and then to the hardware device that implements the requirement. This database maintains configuration control over all requirements, functions, and key artifacts, and by establishing links between elements in the database, we have an auditable mapping between these elements that we can use to prove completeness.

We've found that a good design principle is to delay decisions on physical implementation as long as possible. This reduces the risk of prematurely locking onto a particular technology solution and high costs of changing the selected technology should it fail to meet expectations. We must consider the interdependencies and constraints imposed by technology as we iterate across our candidate architectures [Garlan96]. These interdependencies are clearly evident as the descriptions and depictions of each individual view include elements from other views.

A typical Physical View diagram depicts a network with software components drawn onto devices attached to the network. This is precisely the method we used to develop our baseline physical architecture. To convey this information in our Physical View diagram, we used heavy lined boxes to group software components onto devices, and then labeled each heavy lined box with the device type and its attributes. As shown in Figure 6, our notation quickly conveys what components run on the platform and that this particular platform hosts software components belonging to four different logical subsystems.

Notice that some connectors in the diagram are replaced with TCP/IP socket symbols. Although not specified in the diagram, we implement remaining connectors (some labeled with the primary data flow) as C function calls or, in the case of specific components such as those in our common (reusable) utility libraries, via Dynamic Linked Libraries (DLL).

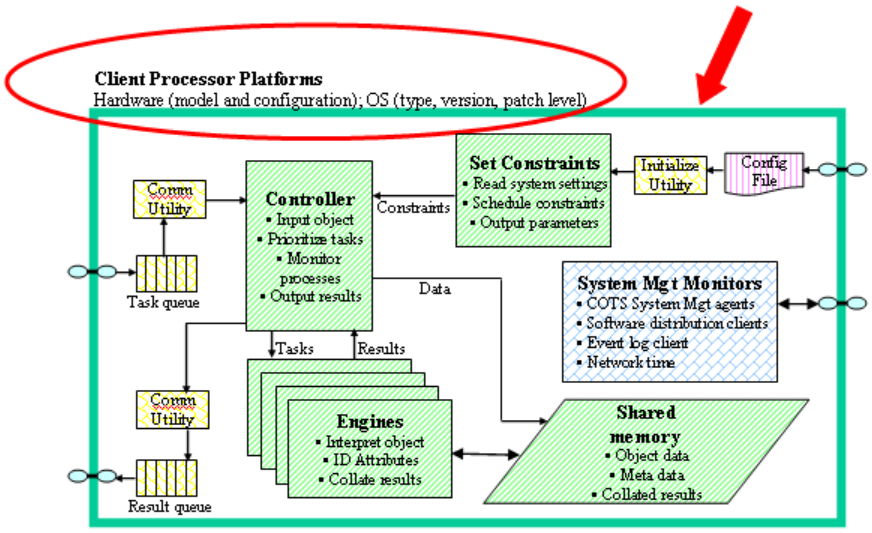

Figure 6- Example Physical View. *A portion of a Physical View showing the hardware specification (hightlighted in the red circle) for functional components grouped by the green box (red arrow). Note that we often denote not only "what something does" but "how it does it" – for example, implementing physical connections via TCP/IP sockets and using shared memory.*

After much iteration on many major system architecture alternatives and numerous design analysis meetings, informal reviews, and milestones, we began low-level design and system implementation.

Unifying Four Views

By the time we were knee-deep into our project, we observed that nearly every meeting resulted in a white-board sketch of our system, beginning with the overall system view and ultimately focusing on details representing the concern of the day. Frankly, we spent so much time re-drawing the same sketches that we decided to commit them to paper and, with that, the "Big Picture" came to life.

As shown in Figure 7, our notation actually made combining the four views into one diagram a simple matter, because using colors and patterns readily distinguished subsystems. This defined boundaries for component functional ownership and development responsibility. By adding data flows with explanatory labels, system behavior effectively emerged. Grouping logical components onto physical devices completed the message needed for implementation and clarified the system's distributed nature.

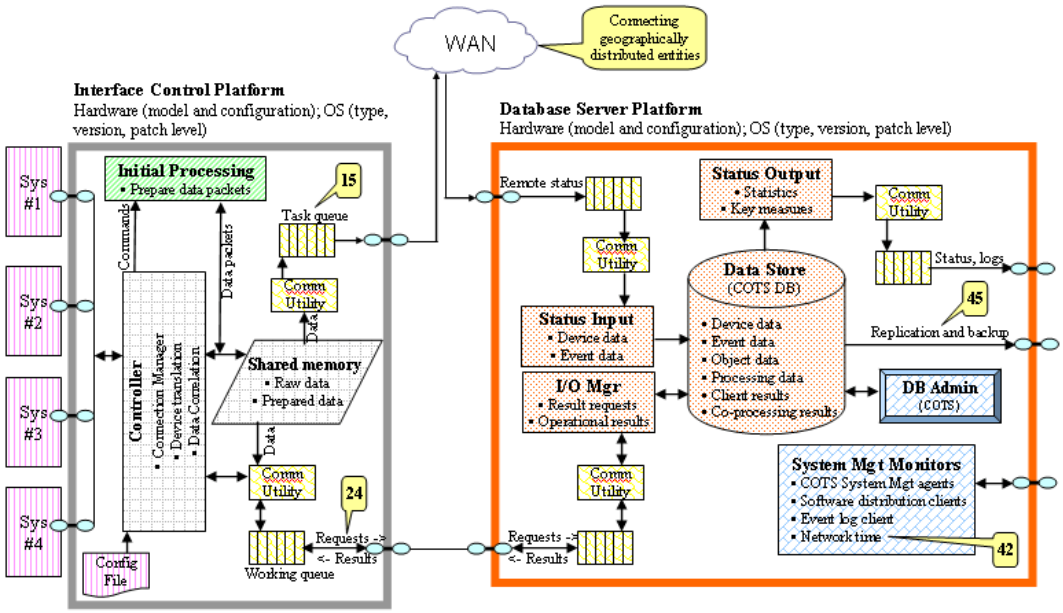

Figure 7- Combining Four Views into One. *A small portion of a "Big Picture" showing logical components with allocated function, operational flows with data and explanatory labels, physical allocation of software to subsystems using colors and patterns, physical allocation of components to hardware devices by groupings, and primary data stores with their contents.*

In the end, the big picture's physical size grew to legal-size paper and eventually a 4' x 3' wall chart – far too large to depict in a journal article. Despite the added real estate provided by the large paper, the format still could not accommodate all information. For example, we deleted system boundaries depicted in the context diagram once system scope was clearly understood throughout the team. To keep the diagram readable we eliminated other detailed information, such as diagnostic flow, error handling, timing, event logging, configuration options, and exception handling. Dozens of sequence diagrams and architectural constraints such as over 25 Technical Performance Measures (TPM) were better conveyed elsewhere, such as in the SAD.

This "Big Picture" proved its value throughout the project. One disadvantage of using multiple views to describe architectures is no matter how hard we try, portraying different perspectives in different ways can lead to different interpretations and consequently costly errors. Having one representation of our system eliminated this problem. Our combined diagram also greatly simplified group communication because coordinating amongst team members often depends on message traffic and the context for that traffic. Whenever a new requirement or issue cropped up, we'd all point to the spot of interest in the big picture. Instantly everyone understood the "where" and "what" of the situation.

Summary

A good architectural description must uniformly communicate a system's essence to customers and stakeholders – from the top to the bottom of every organization. While Kruchten explained the difficulty of effectively containing all necessary details in one diagram, even in the best circumstances multiple views can lead to divergent

interpretations. Our "Big Picture" experience validates thinking outside the traditional box. As a result of one unified system representation, we successfully framed concerns about a complex system to engineers, managers, and customers in a concise and efficient manner. This allowed our team to quickly achieve common understanding with less ambiguity and ultimately achieve more efficient development.

Acknowledgements

I am grateful for the invaluable technical expertise of Joe Zanovitch, Software Architect, and Christine Ackerman, Systems Engineering Assistant, co-owners and developers of the Big Picture. Furthermore, our system came to life due to efforts and abilities of our entire technical team, led by Systems Architecture Board members Bob Bewes, Sheila Curry, Linda Harowicz, Geoff Haylor, Linda Rockwood, Bob Strebel, and Dave Westgate.

About the Author

Jeffrey S. Poulin, Ph.D has over 18 years of experience as technical lead on very large systems development and integration projects at Lockheed Martin in Owego, NY. As a LM Certified System Architect, he serves as a subject matter expert on corporate architect working groups, chairs the Owego technical professional program, and teaches system architecture development as part of the corporate systems engineering course.

References

[Garlan96] Garlan, David and Mary Shaw, "Software Architecture: Perspectives on an Emerging Discipline," Prentice-Hall, Upper Saddle River, NJ, 1996.

[IEEE00] IEEE 1471:2000, Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Press, 2000.

[Kruchten95] Kruchten, Philippe B., "The 4+1 View Model of Architecture," *IEEE Software,* Vol. 12, No. 6, November 1995, pp. 42-50.

[Kruchten06] Kruchten, Philippe, Henk Obbink, and Judith Stafford, "The Past, Present, and Future of Software Architecture," *IEEE Software,* March/April 2006, pp. 22-29.

[ISO09] "ISO/IEC WD4 42010- Architecture description," ISO/IEC JTC 1/SC 7/WG 42 Secretariat, 26 January 2009.

[Poulin97] Poulin, Jeffrey S., Measuring Software Reuse: Principles, Practices, and Economic Models. Addison-Wesley (ISBN 0-201-63413-9), Reading, MA, 1997.

[Zachman87] Zachman, John A., "A Framework for Information Systems Architecture," *IBM Systems Journal*, Vol. 26, No. 3, 1987, pp. 276-292.