# Introduction to System Architecture

A Manager's Guide to
Building better
Information Systems

*** DRAFT ***

Dr. Jeffrey S. Poulin

# Preface

Everywhere you look today, you see people talking about "architecture." Technical articles and presentations inevitably include the "architecture" of a particular system or the "architecture" of a solution to a problem. You hear about "logical architectures," "physical architectures," and "data architectures." Technical professionals have titles such as "software architect" and "enterprise architect." You even see interviews on television with "architects" of ideas and initiatives that do not even have any relationship to computers or high technology.

What does all this mean?

With the many uses of the term "architecture," no wonder that so little agreement exists on what it takes to "have an architecture" or to "be an architect." However, in today's highly complex environment, a project cannot succeed without a leader capable of setting and maintaining a technical vision for everyone on the project. This vision becomes the project's "system architecture" and the leader is the System Architect.

In this book, I introduce the topic of "system architecture" to a general reader, to include executives, managers, and developers. I intend to present system architecture concepts in a straightforward and concise way. In short, I will answer your questions by making sense of the issues and hype surrounding "architectures."

I start the book with an explanation of the typical roles and responsibilities of the technical leaders on a project. I then provide a common understanding of system architecture by describing what system architecture involves and what it does not. Finally, I give a step-by-step tutorial on how to develop and document a system architecture.

As a result of reading this book, you will develop the understanding of system architecture that you require to make you better prepared to organize, manage, and control your most critical projects.

# Acknowledgements

TBD…….

I would like to thank Bill Council for his encouragement though out this effort. A book is a tremendous undertaking (as he will attest, having just finished co-editing Component Based Software Engineering (Addison-Wesley, 2001) with George Heineman. George has provided an inspiration….TBD

Oleg Fedorof taught software and system architecture at IBM, Loral Federal Systems, and then Lockheed Martin for many years. I gained many of my formative ideas on this topic from him. Oleg retired in 2001, but his legacy obviously lives on.

I would also like to thank Marco Temaner of Popkin (www.popkin.com) for his insights and lively discussions on this topic. His presentation at Lockheed Martin on "Architecture-Based System Modeling" provided inspiration for some of the diagrams and text in this book.

# Table of Contents


Preface ..... i
Acknowledgements ..... ii
Table of Contents ..... iii
Introduction ..... vii
1 The Benefits of "System Architecture" ..... 1
Why have a System Architecture? ..... 1
Define the system and how it works ..... 1
Why document the System Architecture? ..... 1
A Typical Project Organization ..... 3
The role of the System Architect ..... 3
The role of the System Engineer ..... 5
The role of the Software Architect ..... 5
Qualities of a good System Architect ..... 6
Concerns of the System Architect ..... 6
Understand, Partition, Scope, and Develop the technical solution ..... 7
What happens if you don't have a System Architecture ..... 8
Tying It All Together ..... 9
2 The Architecture of Systems ..... 11
What is System Architecture? ..... 11
What is the Relationship to Software Architecture? ..... 11
Software provides one part of the total system solution ..... 12
The influence of software on systems ..... 12
*Views* and Sub-Architectures ..... 13
Each specialty has its own perspective ..... 13
Guides to Documenting a System Architecture ..... 13
Available resources ..... 14
Tailoring the guides to meet your needs ..... 15
3 Developing a System Architecture: Step-by-Step ..... 16
Getting Started ..... 16
The first technical activity on a project ..... 16
System Architecture involves many diverse skills ..... 16
System Architecture ends where High Level Design begins ..... 17
Step 1: The System Architecture Document (SAD) ..... 17

Contents ..... 17
Introduction ..... 19
Step 2: Analyze Key Requirements and Constraints ..... 19
Architectural Drivers ..... 20
Techniques for requirements analysis ..... 21
Tool Support ..... 22
Requirements Verification ..... 23
A Final Caution on Requirements ..... 24
Step 3: Define the Scope of the Problem ..... 24
The Context Diagram ..... 24
Interfaces with other systems ..... 25
Interfaces: another high-risk item ..... 25
Step 4: Select Your Views ..... 26
Guidelines for selecting views ..... 26
Why different situations require different views ..... 27
Views evolve over the life of the project ..... 27
Step 5: Develop the Logical View ..... 28
List the logical components ..... 28
Allocate Requirements to Components ..... 29
Review and analyze alternatives ..... 30
Select a solution ..... 30
Conduct informal reviews ..... 30
Step 6: Develop the Data View ..... 30
Identify major data sources and stores ..... 30
Examine flows of information ..... 31
Establish constraints on data ..... 31
Review and analyze alternatives ..... 32
Select a solution ..... 32
Conduct informal reviews ..... 32
Step 7: Develop the Operational View ..... 32
Show how components communicate ..... 33
Review and analyze alternatives ..... 34
Select a solution ..... 34
Conduct informal reviews ..... 34
Step 8: Develop the Physical View ..... 34
Allocate the logical components to physical devices ..... 35

Distribute Devices and Services ..... 36
Set Direction for the implementation ..... 37
Review and analyze alternatives ..... 37
Consider performance of devices ..... 38
Select a solution ..... 38
Conduct informal reviews ..... 39
Step 9: Develop a Concept of Operations ..... 39
Define the vision for how the system will operate ..... 39
Validate the vision with the users ..... 40
Step 10: Address Key Issues ..... 41
Timing constraints ..... 42
Performance allocations to subsystems ..... 43
Interfaces ..... 44
System States ..... 44
Security ..... 45
Technical Performance Measures (TPMs) ..... 46
Establish the migration plan ..... 46
Consider alternatives, document rationale ..... 47
Step 11: Add Supporting Information ..... 47
Tools ..... 47
Processes ..... 48
Step 12: Formally Review the Solution ..... 51
Technical Solution Review (TSR) ..... 52
Communication! ..... 52
The Architecture Board ..... 52
4 Examples from Production Systems ..... 54
5 Conclusion ..... 57
The System Architecture “Elevator Pitch” ..... 57
6 Further Reading: Software Architecture in a Nutshell ..... 59
What is Software Architecture? ..... 59
The software approach to the solution ..... 59
One part of the greater system solution ..... 59
Definition of Software Architecture ..... 60
The “Three Cs” of Software Architecture ..... 60
Components: The software sub-systems ..... 60
Connectors: How the sub-systems communicate ..... 61

Constraints: The rules software must play by ..... 61
Styles ..... 61
Layers: The most common style ..... 61
The Object-Oriented Style ..... 63
The Blackboard Style ..... 64
The Many *Views* of Software Architecture ..... 65
Each specialty has its own perspective ..... 66
Summary: the 3Cs, Styles and Views ..... 67
Additional Resources ..... 68
Glossary ..... 69
Annotated Bibliography ..... 71
Index ..... 75

# Introduction

In the world of computing and especially of information systems, "architecture" has become synonymous with the highest level of organization in a system. "Architects" have the responsibility for understanding, defining, validating, and implementing that system. With the many technical options available to create a solution, a fine line exists between building a system that works and one that will ultimately fail. The architect must balance the options (along with cost and schedule) to lead the technical team to a common and achievable solution.

The output of these steps is a roadmap to the solution for a project. Today's complex projects need both this clear roadmap and a strong leader to run to successful completion. This book explains how to make that roadmap and how a System Architect uses it to implement a complex information system.

The extremely large information systems developed by industry today demand a well-documented path that translates the vision for the solution into clear direction for the implementation team. The System Architect starts by developing a high level organization of a technical solution that meets the needs of the Customer, within the constraints of cost, schedule, and technology established by the contract. The architect must then effectively communicate this solution to all members of the development team and to the Customer.

To do this, architects define the solution using tools such as context diagrams and models that depict how the architect intends to build the system. The architect documents the rationale used to make major decisions and outlines the concept of how the system will operate. The architect also defines the key processes that the team will follow to effectively manage and coordinate their development activities.

This planning and designing activities result in a *System Architecture Document* (SAD) that the team uses as a common framework for design and implementation. This book walks the reader through the key steps in developing a SAD for a major project, starting with key requirements, putting a boundary around the problem to limit the scope of the solution,

making initial decisions about major functions and how to allocate them to development teams and physical hardware upon which they must execute. Taking a disciplined approach to analyzing these issues and making the best possible decision within established constraints can eliminate future problems and avoid many of the development nightmares suffered by so many large-scale systems today.

It makes sense, therefore, that we begin by discussing the benefits of System Architecture and why System Architecture has attracted so much attention.

# 1 The Benefits of "System Architecture"

## Why have a System Architecture?

### Define the system and how it works

*Every system has an architecture*

Every existing system has some kind of high-level organization, so we can claim that every system, in fact, has an architecture. However, in many cases a person new to the system would have a hard time discerning the major system components and how they interact. This is because either during development, or simply over time, the design goals for the system eroded in order to address new requirements or to fix implementation errors.

*Documentation may not exist*

Furthermore, older systems often lack suitable documentation on their major functions and organization. The first step in understanding a system that you must either repair or interact with usually involves obtaining this high-level view of what the system does and how it does it. Unfortunately, most of us have had difficulty obtaining this information on a system when we need it.

*Documentation may not match reality*

If the documentation exists for older systems, it likely is out-of-date. We all know that the last task on the development team's priority list is to update the manuals and documentation on what they have implemented. Many reasons exist from diverging from the original designs, such as having to accommodate discoveries made during implementation. However, other people must continue to support or interact with the system long after the implementation team disbands.

*New systems should start by defining their System Architecture*

Especially for large-scale projects and systems that will have a long life, a project can avoid these traditional maintenance and integration problems by establishing, documenting, and enforcing the rules by which the system should evolve and behave.

### Why document the System Architecture?

*To communicate!*

Clearly, the most important role of a System Architecture and of the System Architect is to communicate the high level organization of the system to a wide variety of audiences. In fact, from the time that the vision for the system first starts to take form, the architect will begin a long series of

presentations in which he or she repeatedly conveys how the system works and why. These presentations keep all the audiences involved and informed.

*To have one source for key information*

Each audience will, of course, have its own unique concerns and will require varying levels of detail, depending on their technical interests. For example, end-users may focus on the how the system will affect their operations at end-state, whereas developers may want to know the technical standards and constraints that they must follow. Despite the various concerns, each audience should get their information from a common source so that the purpose, approach, and rationale for the system solution remain consistent.

*To show the solution*

While the System Architecture serves as the central authority from which subsequent design decisions flow, the technical solution has many aspects. The architecture clearly defines the boundary of the system and therefore the scope of solution. It outlines the key functions that the solution will provide. It also defines the major parts of the system (everything from hardware to software) and how those parts interact. In short, the system solution begins with the architecture.

*To show key issues*

Every system has a series of constraints within which it must operate. In some cases, one or two constraints will serve as the key drivers for the systems solution. For example,

- Mobile devices must balance power output with battery life
- Miniature devices must dissipate large quantities of heat in proportion to their size
- The problem may require large amounts of processing power versus large flows of data.
- The cost of a wide-area network, computing hardware, recurring maintenance, or having the system fail during operation may completely overwhelm other cost factors.

The architecture document must address each constraint by explaining how the solution satisfies the constraint and by providing the rationale for the chosen solution.

*The primary source for information*

Most importantly, the System Architect document serves as one of the primary source for the vision and solution between team members and with the Customer.

# A Typical Project Organization

We have all seen project organization charts and probably debated the merits of the different ways to organize the people that work on a project. Of course functional organizations work best in certain situations and cultures, just as matrix organizations work best in other situations. The following describes a common way to organize the *technical team* on projects on which I have worked.

*This book addresses the technical leadership of a project*

These projects all have had a technical hierarchy that ran independently of the management hierarchy. The technical chain had responsibility for the solution (within cost and schedule), whereas the management chain retained responsibility for actually managing the cost and schedule, as well as all human resource responsibilities.

*Separate the technical chain of command from management*

Although the titles and division of responsibilities may not exactly match those used in your company, the organization described below will serve as not only a possible organization for future projects but as a common reference for the activities described in this book.

*The following roles serve as proven examples*

## The role of the System Architect

Every project must have a technical leader. This leader serves as the one person that must ultimately make the call on the myriad of complex and difficult technical issues that come up during the life of a project. When diverse technical teams have a conflict over the best way to solve a problem, the System Architect must weigh the alternatives and help the teams arrive at the best possible solution for the project.

*The System Architect is the technical leader of the program*

In many ways, the System Architect has a job similar to the Program Manager, who is the highest-level manager on the project. In fact, the System Architect often works for the Project Manager (the Project Manager is the System Architect's manager). However, the Project Manager leads business issues such as contracting, Customer negotiations, and business planning, whereas the System Architect focuses building a solution for the Customer.

*The System Architect may work for the Program Manager of the project*

Figure 1 shows the relationship between the technical leaders on a typical project. The System Architect "owns" all aspects of the system solution, as we have only begun to describe so far in this book. On some projects, the lead technical person has also had the title of "Chief Engineer." For the

*The key technical leads report to the System*

most part, the title has little impact on the day-to-day responsibilities of the person in that position.

*Architect*

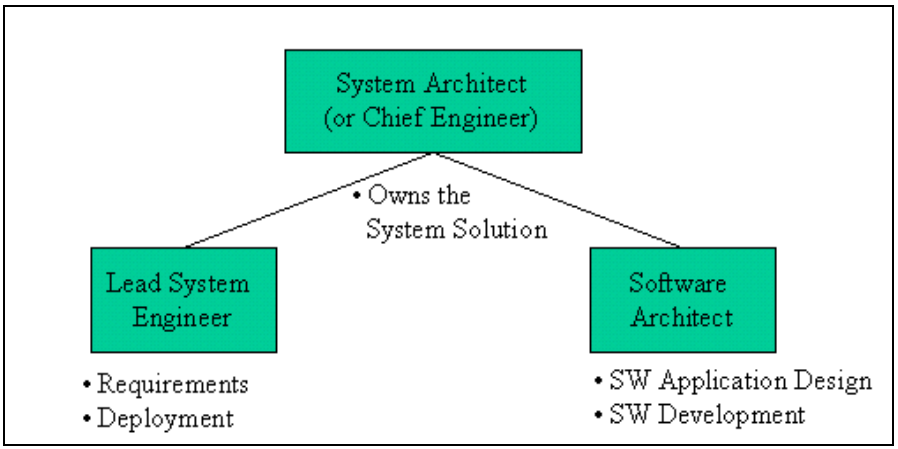



**Figure 1 - The technical leadership for a typical project**

*Engineers focus on requirements and deployment*

However, in general, *engineers* have a role that connotes a closer relationship to requirements elicitation and verification than to developing the system solution. They also have responsibility for fully understanding how the system will behave after it has been delivered to the Customer and ensuring that the system successfully makes it out into the end-user's hands.

*Architects focus on system design and development*

On the other hand, *architects* have responsibility for designing the solution and overseeing its development. Their title connotes the role of a *builder*. Architects can build entire systems (the System Architect) or have responsibility for a subset of the entire solution. **Error! Reference source not found.** shows a *Software Architect* reporting to the System Architect. In this example project, the software solution requires a specialist to focus on the design and implementation of the software. This is a common situation in today's software-intensive systems.

*The two overlap significantly*

This does not imply that the two persons never do the other's job. In fact, I have found that, independent of the title, quite often the same *person* serves as both the System Architect and the Chief Engineer. This actually works quite well as long as the project doesn't get too large because of the natural flow of responsibilities over the life of the project. A well-rounded technical lead can easily perform both functions. Now we'll see talk about

a few specific responsibilities of these technical leads.

### The role of the System Engineer

*System Engineers handle requirements…*

The System Engineer (SE) has his most critical time at the beginning of the project because the Lead SE has primary responsibility for eliciting and verifying requirements. As we all know, issues surrounding requirements have caused far more projects to fail than issues surrounding the systems solution. The old days of *waterfall lifecycles* have thankfully gone, but this makes it even more critical to have strong SE leadership. Even though this role diminishes as the project progresses through design and development, modern projects must routinely deal with evolving requirements and project scope.

*… the Customer…*

Understanding requirements requires a very close relationship with the Customer. This becomes especially important over time, when evolving requirements or new interpretations of requirements become evident. The SE must delicately balance the Customer needs with contractual obligations (such as available budget). Because the Customer relationship sets the tone for the entire project, the SE must have exceptional personnel and communications skills.

*… and system deployment*

Finally, the Lead SE has responsibility for getting the solution into the hands of the Customer. This really involves several unique skills. The first involves determining the Bill of Materials of equipment that will need to support the system in the field. Next, the SE must size the equipment so that it scales to support the anticipated number of users. Finally, the SE must actually bring the hardware and software to the field and oversee its installation and initial use by end-users.

### The role of the Software Architect

*Software Architects develop the software solution*

Most large computer systems today consist primarily of software. When this is the case, a specialist in software design and implementation will lead the software development team. In keeping with the title of *architect* as one who *builds* a solution, we call the technical lead of the software team the *Software Architect* (or perhaps *Lead Software Engineer*). In short, the Software Architect works closely with the Lead SE to understand the functional requirements of the system and then to translate those requirements into code that can implement those functions. The Software Architect also has to work closely with the Lead SE to determine the type of

hardware upon which the software will run, the capabilities of that hardware, and to take measures to ensure that the software developed performs to the user's satisfaction once those platforms make it to the field.

### Qualities of a good System Architect

*The Architect must be a great leader…*

While we have talked a lot about what a System Architect does, we have yet to really expand upon the type of person that typically assumes the role of System Architect. It should come as no surprise that anyone that serves as a key leader on a large project should have, among all qualities, excellent leadership skills. Although it helps tremendously to be generally liked by the technical team, it is far more important that the System Architect command respect and the trust of his team. Without this, I suggest that even exceptional abilities in other areas will not make up for a lack of leadership ability.

*… And communicator...*

In addition, the System Architect must have an excellent ability to communicate many things to an audience. Whether presenting to Customers, auditors, reviewers, peers, or the technical team, the System Architect must clearly and concisely convey his intent. It amazes me how often people hear what they want to hear or interpret the same information in so many ways. This only leads to confusion or worse problems later on.

*… And have excellent technical ability.*

Finally the technical lead must have excellent technical ability and experience in many specialties. The System Architect's responsibilities span every discipline on the team, from management processes to very minute details of coding, networks, databases, and other technical domains. Although the Architect may not actually do any development, he will be called upon to investigate or solve problems in every one of these areas at some point in the project.

## Concerns of the System Architect

*System Architeture involves many issues*

Now that we have established a potential technical organization for our project and we have assigned the right personnel to their respective leadership roles, what kinds of issues will the System Architect have to worry about?

### Understand, Partition, Scope, and Develop the technical solution

*Assess the problem*

No work can begin until the System Architect develops an initial understanding of what the system has to do. While the Architect develops that understanding, he starts to also shape potential solutions to the problem. Experience plays an important role, as the Architect typically does a mental gymnastic, simultaneously trying out all the ways that he knows of for solving the problem, assessing the advantages and disadvantages of each, and mentally ranking them in terms of their "goodness" to solve this problem.

*Partition the system*

This mental exercise often involves partitioning the problem into smaller segments or areas of specialty. The smaller segments might eventually become subsystems (for example, a major software component) and the areas of specialty might get delegated to a specialist (for example, the network design, the database design, security considerations).

*Scope the solution*

The Architect draws a boundary around the work to clearly mark what belongs in the system and what does not. This not only tells everyone what has to get done but also helps identify the other systems with which his system must interact.

*Develop the high-level solution*

The solution, however, will almost certainly involve a mind-boggling number of technical issues. These include:

- The hardware environment, such as the size, number, and performance of machines.
- The network, to include the bandwidth, domain naming and time services, system management
- The software approach, to include the use of languages, design methods, processes, interfaces, and various standards.
- Performance of the software on the selected hardware, along with the required response times and amount of resources it can use.
- Security of the system, both physical and logical, to include the use of access control mechanisms such as Identification and Authentication, Virtual Private Networks (VPN), and database roles.

- Human factors concerns for graphical interfaces and physical devices
- Integration and test strategies, to include use of configuration management, problem tracking, diagnostic utilities, and test facilities.
- Migration of the Customer from the "as-is" environment to the "to-be" environment in a painless and logical way.
- The entire project lifecycle from planning to conclusion, to include dynamics in technology and economics.

*Deliver to cost and schedule*

The Architect must balance these issues (along with many others) to create a feasible solution to the letter of the requirements and the spirit of the user's need. The solution must be feasible within an acceptable margin of risk not only technical, but so that the Architect can have a high confidence of completing the job within cost and schedule.

### What happens if you don't have a System Architecture

*Lack of common vision leads to chaos*

I find it relatively easy to describe the benefits of System Architecture to people who have experience on projects that lack a well-documented Architecture, a strong Architect, or both. Without these, no wonder so many information technology projects end up with a rat's nest of interfaces and interdependencies such as those shown in Figure 2.

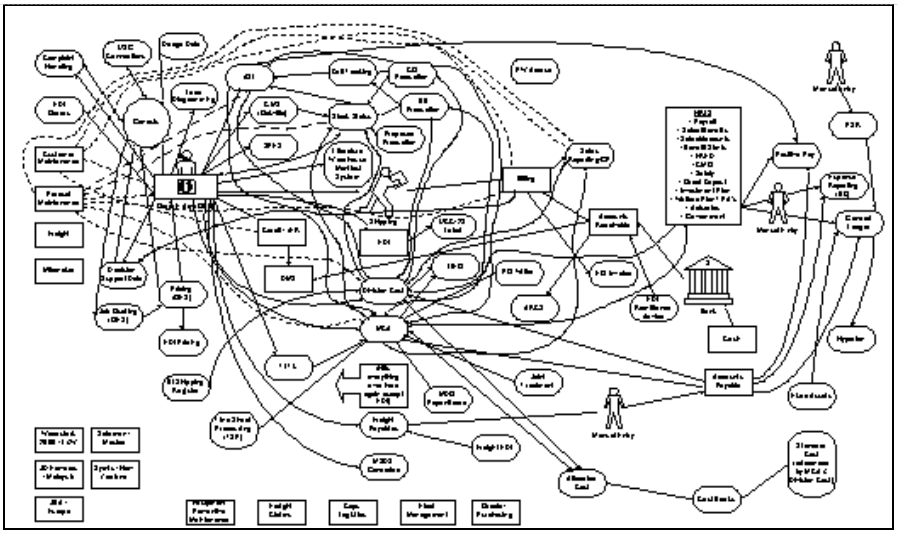


**Figure 2- What happens without a System Architecture**

*People understand complex issues differently…*

Spaghetti diagrams such as these can come about very easily. In part, they happen because people have different ways of learning, and tend to frame their understanding of issues based on their experience. Since we all have different experiences, even under the best of circumstances we will naturally understand complex issues differently.

*…or hear what they want to hear*

In the worst case, I have also seen very tough meetings that sought to resolve a contentious issue results in continued confusion. Why? Because everyone in the meeting will interpret the "agreement" in their most favorable light. Everyone leaves the meeting thinking what he or she wants to believe, so in reality everyone leaves with a different understanding of what will get done.

*Success depends on keeping the team on the same track*

The System Architect aligns the technical personnel of a project so that they all head towards the desired destination. It takes a clear initial direction and a continuous steady hand on the wheel to keep the technical team headed towards a common goal.

## Tying It All Together

*System Architecture defines your*

To tie it all together, System Architecture defines the high level organization of your system. Although every system has an "architecture," no modern computer system can get along without formally recognizing the

architecture and documenting how the system fits together.

*system*

The person that has responsibility for developing, documenting, and communicating the architecture is the System Architect. He will likely work with many specialists, such as Systems Engineers, Software and Hardware Architects, and others. However, the System Architect unquestionable leads the technical activities of the project.

*System Architects lead the technical activities of a project*

To do this, the System Architect develops a vision and plans based on his understanding of the problem and the merits of the various possible solutions. The Architect then relies upon continuous communications and leadership to keep the technical team on track towards the envisioned solution. Without that guide, the team will surely diverge and engage in costly, uncoordinated efforts.

*Good plans and leadership help prevent chaos*

Next, I will review some basic principles of Software Architecture. A lot of work has taken place over the years involving software and this work has heavily influenced the approach that I will present to Systems Architecture. Like software, Systems Architecture involves many complex and nebulous issues. Nonetheless, Systems tend to have common characteristics that, even though vary in importance depending on the situation, can best be solved through a systematic approach that takes issue in its turn.

*An understanding of System Architecture starts with Software*

# 2 The Architecture of Systems

Now we jump into the important aspects of System Architecture. We start with some quick definitions of System Architecture not so much as to have a common start point, but to show how the definitions flow from the field of software. Next, I will walk through the typical steps in developing a System Architecture. This includes documenting the architecture using views and models, much like we discussed with software.

*Next, we'll see how to develop and document system architecture*

## What is System Architecture?

While I typically avoid citing definitions, in this case it interesting to start with just a couple. The International Standards Organization (ISO) gives a general definition of architecture as:

> "A set of rules to define the structure of a system and the interrelationships between its parts."
>
> (ISO / IEC 10746-2, ITU-T X.902)

More importantly, the IEEE, which is arguably the most influential source of standards for such things, has adopted the following definition in its official dictionary of computer terms:

> "The structure of components, their relationships, and the principles and guidelines governing their design and evolution over time."
>
> (IEEE 610.12)

## What is the Relationship to Software Architecture?

Software Architecture has been around a lot longer as a topic of study than System Architecture. In part, this stems from the difficulties and mysteries surrounding software development. In contrast, the tangible nature of classic system issues of hardware and networks seemed easy. However, predominant thinking changed once people realized the extreme complexity of combining all these aspects into a complete, working solution.

*The study of software pre-dates the study of systems*

### Software provides one part of the total system solution

The System Architect will choose to deliver many of the other system functions in hardware, networks, or in other ways. Of course, the software has to interact with many of the other parts of the system. For example, if the System Architect tells the Software Architect that the software must execute with a response time of 1.5 seconds, the Software Architect will have to know such things as the transaction rates of the system and the processing power of the hardware upon which his software will run.

*Software interacts with the rest of the system*

### The influence of software on systems

Much of the study and research work in Software Architecture comes from Mary Shaw and David Garlan [Shaw96]. Their work is perhaps the most influential classic styles of software systems tend to follow.

The reason I gave the definition of system architecture, above, is clear when you read the following definition of *software* architecture:

> "The structure of the components of a program/system, their interrelationships and principles, and guidelines governing their design and evolution over time [SEI94]".

*System Architecture received many influences from Software Architecture*

Obviously, the IEEE definition of System Architecture has been heavily influenced by that of Software. It should not surprise us, then, that many of the concepts of Software Architecture have also flowed into the field of System Architecture. As we will see, these concepts mostly involve the ideas of views.

*System Architecture expanded the principles of software architecture*

## *Views* and Sub-Architectures

*Everyone has a special interest*

Software Architects often describe their architectures using several *views*, or perspectives of the system. Just as architects of houses prepare wire diagrams for the electrician, floor plans for the framers, etc., architects of software prepare special diagrams for each software specialty. These views represent the special interest, viewpoint, or perspective of the intended audience.

### Each specialty has its own perspective

*A module view show software components*

One of the first views that a Software Architect might prepare is the *module* view. The module view simply shows how the architect has decided to partition the software system into software subsystems. Each subsystem is a module in this view. The first iteration of preparing a module view might identify, for example, the layers in the layered architectural style.

*A logical view shows how components interact*

Next, the Software Architect might start to show how the modules interact. This *logical* view identifies the data or control flows in addition to the major software components. It portrays the high level behavior of the system and typically represents the cornerstone of the software architecture.

*A physical view fits the software to the system*

The most likely next step involves assigning the software to the physical devices upon which it will run. The Software Architect may have input into the physical devices, but normally the System Architect tells him what to expect for a physical environment. The interest of the Software Architect end at this boundary.

## Guides to Documenting a System Architecture

*Guides to architecture include journals, other projects, and formal*

An Architect does not just sit down and start documenting a system architecture. In addition to the obvious technical qualifications, he needs to understand at least the basics of how to document an architecture. He can get that in many ways, such as by reviewing academic work on the topic. He could also analyzing existing architecture documentation from other projects at his company to get examples of the standards expected on

similar projects. Finally he could investigate the applicable formal standards that might exist from reputable organizations, such as the ISO, the American National Standards Institute (ANSI), and the IEEE.

*standards*

### Available resources

As it turns out, in 2001 the ANSI Board of Standards Review (BSR) approved a new American National Standard on how to document a System Architecture for software-intensive systems [IEEE00]. This document formalizes some very general guidance and ideas that have been around for a long time, most notably in [Zachman87] and [Shaw96].

*The IEEE gives some general guidance*

Figure 3 summarizes the highlights of the IEEE guidance. In short, the IEEE recommends organizing an architecture document by views to represent the viewpoints of each stakeholder in the system. This aligns with what we have seen with software architectures. But how do we document a view?

*Organize the architecture into views*

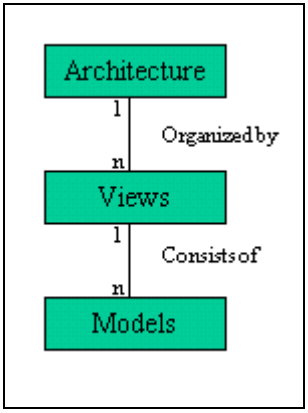



**Figure 3- The relationship between architectures, views, and models**

The architect may represent each view with one or more models, or diagrams. As we will see, architects and software designers tend to use a standard set of diagrams to depict each view. The choice of model or models remains up to the Architect. The IEEE guidance does not mandate any specific views or models to use; only that this makes an excellent method of presenting System Architectures.

*Represent each view with a model*

### Tailoring the guides to meet your needs

*Represent each view with a model*

The architect may represent each view with one or more models, or diagrams. As we will see, architects and software designers tend to use a standard set of diagrams to depict each view. The choice of model or models nonetheless remains up to the Architect. The IEEE guidance does not mandate any specific views or models to use; it only suggests this as a proven method of presenting System Architectures.

*Modify the guide to meet your needs*

As with any guidance, you should consider any recommendations and apply them as you see fit. Some, of course, you will use without change, others with considerable modification or not at all. This book is based on the same principles as those forwarded by the IEEE. However, like any guidance, this book can only suggest a roadmap to organizing the technical activities on a project and how to systematically develop and document your System Architecture.

# 3 Developing a System Architecture: Step-by-Step

We have come to the most important part of this book: walking through the tasks that an Architect completes on the road to creating and documenting the System Architecture for a large project.

*The project begins*

## Getting Started

At this point, we have assigned the leadership for the project and defined the responsibilities of the key technical and managerial leads. Everyone sits pretty much in the same boat, with little information on what has to get done nor why. Perhaps the team has just won a contract and now has just started to get organized. The technical team forms around the System Architect and other leaders, such as the lead SE and the Software Architect, begin to read about and digest what they will have to do.

*The management and technical organizations are put in place*

### The first technical activity on a project

No doubt that the System Architect has a large and daunting task ahead of him. While the management team assesses the budgets and required delivery dates, the technical activity starts in earnest. The budget and schedule dictate hard constraints on the technical solution, because failure to live within these constraints will doom the project to financial disaster.

*Cost and schedule constrain the Architect*

When it comes to cost and schedule, therefore, the Architect has as much responsibility as management. He must validate the realism and feasibility of the cost and schedule, adjusting them if necessary. If the cost and schedule cannot change, than the technical solution must fit these constraints. He cannot allow the technical team to develop a solution that falls outside of these concrete boundaries.

*Everyone on the technical team must know and live by these constraints*

### System Architecture involves many diverse skills

While a System Architect has to have a strong technical background in many areas, it is more than just hard to find a person with in-depth knowledge of all the technical areas involved on a large project. It is certainly impossible. As a result, the Architect must fill in the gaps in his personal knowledge and experience base with experts drawn from wherever

*No one person can know it all*

he can get them.

Fortunately, most large organizations have large numbers of personnel with many different talents. Some organizations formalize the arrangement by forming *task forces* or *multidisciplinary teams*. Whether by formal arrangement or through an informal network of contacts, the Architect will make use of all these resources. It never fails to amaze me how many different people contribute to the development of an architecture, and even more so once we start to uncover the details of design and implementation.

*Solicit input from the experts*

### System Architecture ends where High Level Design begins

The System Architect should plan to curtail his guidance to the team leads at the point where high-level design would normally begin. This can be a hard point to identify and will vary depending on the situation.

*Architecture is not design*

I try to make the most detailed depiction of each subsystem the point where I want the subsystem owner (Software Architect, Network Engineer, etc.) to begin their work. This provides a natural way to handle the transition from architecture to design and makes for a smooth transition in any presentation of the system. Whether in person or on paper, the final, most detailed diagrams presented by the System Architect are the first ones presented by the applicable subsystem owner.

*Use the transition to hand-off responsibility to subsystem owners*

## Step 1: The System Architecture Document (SAD)

It helps me to have a clear vision of the tangible output of the architecture effort. The most tangible output is almost certainly the System Architecture Document (SAD). It helps to have an example or template to follow as it can help keep the Architect focused on the major task while also reminding him of the smaller things that he must consider throughout the effort. As the Architect uncovers new evidence, it becomes part of the rationale captured in the SAD. When the Architect makes a decision, it becomes part of the solution presented in the SAD.

*The SAD will list the issues to conquer and record the decisions*

### Contents

Figure 4 shows an example table of contents for a SAD. I usually start with a SAD template in a word processing tool or presentation tool, such as

*Create an outline for the*

*Microsoft* Word® or Powerpoint®. If you do not have a template or example to follow, it is a simple matter to create an outline of the SAD from scratch.

*SAD*

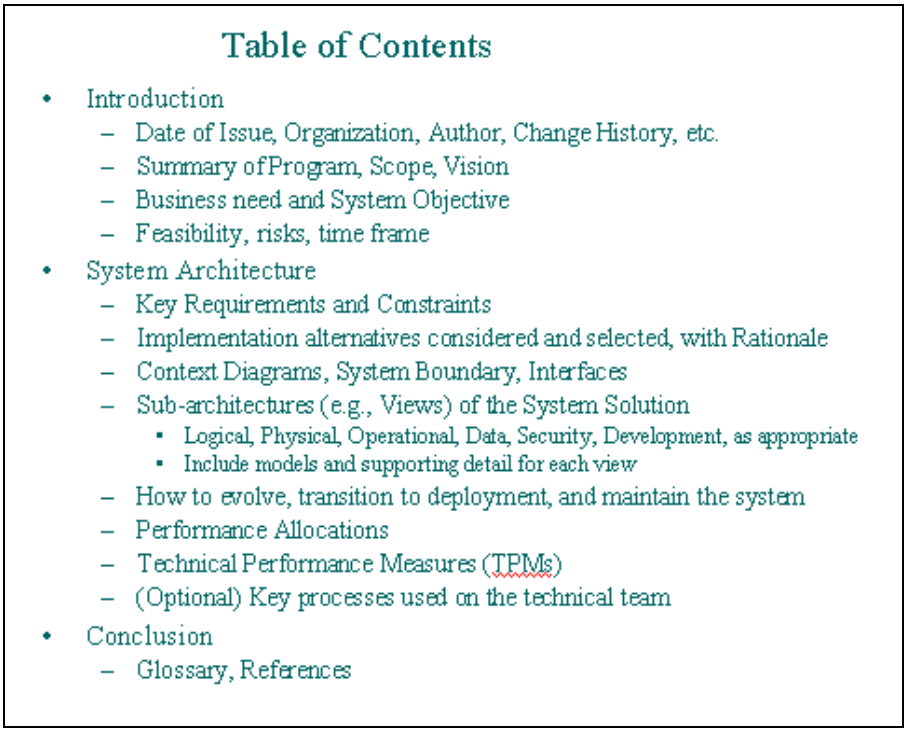



**Figure 4- Sample Contents of a SAD**

The use of Powerpoint for a document bears reiterating. Because an Architect spends so much time presenting the architecture, I have found it considerably more time-efficient to use Powerpoint along with the notes feature than to use a standard word processor. The use of Powerpoint has numerous advantages:

*Consider using Powerpoint with notes to document the SAD*

1. it prevents having to keep a document and a set of charts synchronized; everything exists in one place
2. it forces the use of graphics and simple lists to convey ideas concisely and clearly
3. it automatically creates a "summary" and "detailed" version of the architecture

4. it provides a versatile and non-intimidating way for someone to learn the architecture without necessarily having to take the Architect's time.

### Introduction

The Introduction to the SAD begins with the perfunctory administrative information, such as date, author, and change history. You will need this for version control. Since nearly every audience will need some sort of context, the Introduction should include a brief overview of the program, its intent, why the Customer needs the program, and the problem that the program seeks to solve. Most of this background can be assembled from information provided by the Customer. Finally, the Architect may want to provide some introductory information of his own. This might include a statement about the major challenges or greatest technical risks that the solution will have to overcome on the road to being successful.

*Provide the background information in the Introduction*

## Step 2: Analyze Key Requirements and Constraints

The best place to begin understanding what the Customer wants you to deliver comes from information provided by the Customer. These include the functional and other requirements that he spells out for you in a Request for Proposals (RFP) or the equivalent Invitation to Tender (ITT). Most large projects will include ample time to amplify these documents by meeting with the Customer. Such sessions may be informal but also may include workshops and bidder's conferences related to the contract. The Architect, as well as all members of the managerial and technical team, have to carefully uncover what the Customer wants, whether it has been captured as formal requirements or not.

*Understand what the Customer really wants*

This process aims to uncover functional requirements. In addition, the Architect should determine the business reasons and basis for the contract. Sometimes the Customer may formally provide this in a document such as a *Mission Needs Statement (MNS)*. Understanding the Customer's financial motivation often leads to better solutions and insight into where the Customer might trade-off functional requirements to obtain more overall value for the project.

*Understand the Customer's business need*

### Architectural Drivers

*Understand the key technical constraints, or "Architectural Drivers"*

In the course of reading documents and meeting with the Customer, the Architect will discover requirements that explicitly limit the technical solution in some way. These may specify or forbid a technical approach. Generally these come in the form a part of the system that costs significantly more than the rest of the system. They may simply result from some sacred cow in the eyes of the Customer. Example architectural drivers might include:

- *Performance* - the ability to perform a minimum of X million transactions per day
- *Availability* - 24 hours/day, 365 days per year
- *Network bandwidth* – cost or quantity
- *Power management*– especially with mobile devices such as PDAs and cell phones
- *Specific Hardware Platform* - required to use a specific hardware or operating system vendor
- *Client/Server* - Implementation of interprocess communications based on distributed queues
- *Database independence* -Implementation of applications must not depend on the COTS database system.
- *Table Driven* - Implementation of operating parameters separate from source code
- *Standards compliance* – such as TCP/IP, SQL, X-Windows/Motif, OSF/DCE, Internet RFCs, and well defined APIs, etc.
- *Multi-level security* – such as required by military applications

Each architectural driver will, by definition, have a profound influence on the solution. In other words, the optimal solution would almost certainly look much different in the absence of that driver.

*List and Prioritize Constraints*

It generally helps to list the constraints in their order of importance and record them in the Introduction to the SAD. In this way they serve as reminders when considering options for any possible solution.

*Handle hard to quantify*

The Customer may also list, or simply expect, the solution to meet some unstated criteria of success. These "I'll know it when I see it" requirements

often become the bane of the Architect's final days on the contract. They begin with "*Ease of use"* and extend to the entire family of "-ilities," such as *flexibility, extendibility, scalability.* In these cases the Architect can best bound his organization's commitment with measurable, testable tasks that show how the project meets these naturally hard to quantify issues.

*drivers*

### Techniques for requirements analysis

Plenty of excellent references exist on the best ways to elicit, record, and verify requirements, so it serves little purpose to further detail them here. I cannot emphasize, however, the importance of re-reading the Customer's documentation (e.g., RFP, ITT) and your organization's response (e.g., your proposal or contract). These documents outline your ultimate legal responsibilities to perform on the contract the architect must practically commit them to memory. It shocks me when I work on a project in difficulty to eventually learn how few of the people on the project actually read what their organization signed up to deliver.

*Re-read the RFP and re-read your proposal!*

The method that you use to understand what you have to do will depend on you and your organization's standards, experience, tools, as well as the nature of your project. Workshops that focus on specific issues and capture them in documents or diagrams work very effectively. These can be formal *Joint Application Development (JAD)"* sessions with trained moderates and tools to capture the discussion, or simply meetings with the customer. Remember that the days of specifying all requirements at the beginning of a project have long past. At best, you can expect to describe in general terms what the entire project and perhaps the first iteration of development will do. In reality, this process of exchange continues throughout the entire project.

*Choose an appropriate technique for information exchange*

A popular way to steer and capture requirements is through *Use Cases*. Use cases are a technique developed by Ivar Jacobson that has become very popular in the OO community as part of the Unified Modeling Language [Jacobson99]. As shown in Figure 5, the pictorial portion of the use case (an explanative text portion accompanies the diagram) shows people or things, called *actors* that interact with the system. Circles represent the actions that they perform.

*Use cases elicit requirements from scenarios*

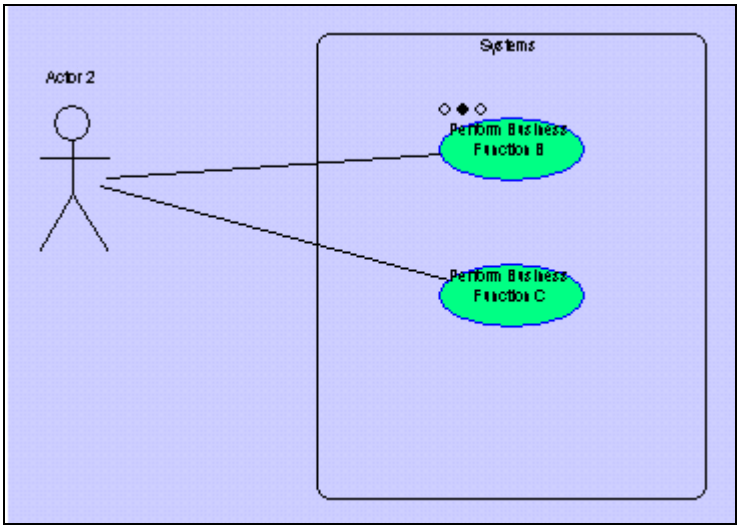



**Figure 5 - Example Use Case Diagram**

The Use Case diagram is only one of many types of techniques that you can use to elicit and document requirements. Figure 6 shows another UML diagram that you can use to who-does-what-and when. This *Activity Diagram* has been divided into three "swim lanes" to show the actions of three different people. The actors stay in their lanes, but the arrows to tasks performed by other actors can clearly be seen as they cross over into the other actor's lane. Time flows from the top of the diagram to the bottom, thereby also depicting the sequence of the actions. The SE or customer may use any technique that they prefer, to include diagrams and processes coupled with tool support.

*You should use any technique that you feel appropriate*

## Tool Support

Many tools exist to help with the administrative portion of this process. I prefer not to use specialized tools, because (despite the expense and training issues) I have found that users of the tools become more enamored in the tool than in the job of requirements. On the other hand, the tools can help coordinate the efforts of a large team and can help enforce consistency and constraints upon the architecture that become difficult to manage, especially on large projects. The tools also are most useful when beginning a new system from scratch, and rarely justify the effort should you find yourself

*Consider the use of tools on new projects*

maintaining a legacy system or integrating pre-built COTS or reusable components.

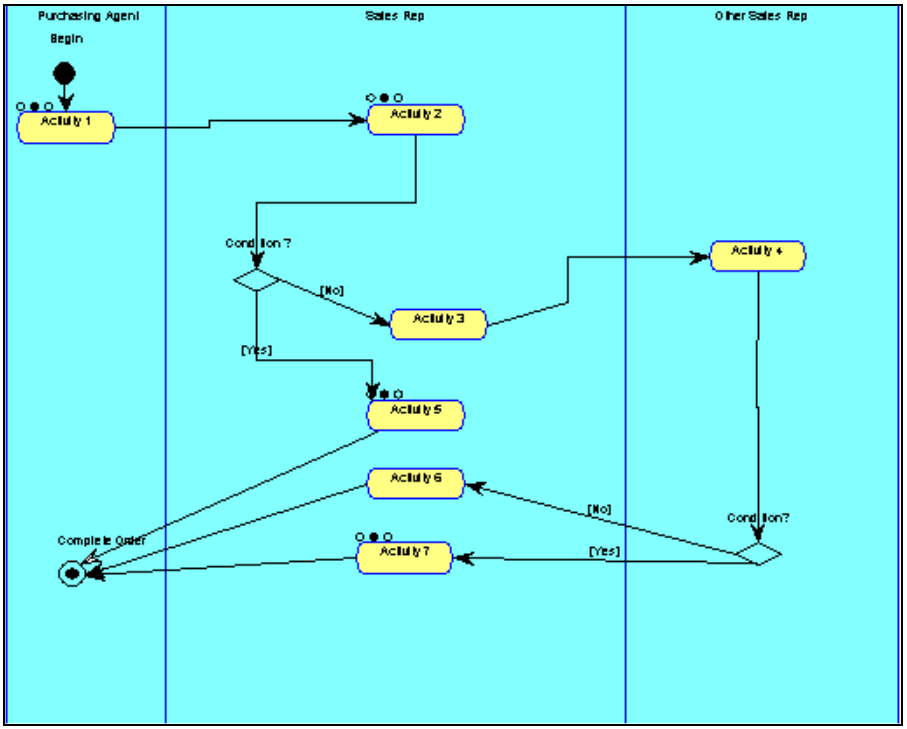



**Figure 6 - Example Activity Diagram**

## Requirements Verification

*Verify conclusions with the Customer*

Finally, you must verify the requirements with your Customers. The best method of requirements verification is to have your Customers close to you throughout the project. Openness and honestly usually prevail. Furthermore, by keeping your Customer informed you instill a strong sense of ownership and teamwork. Once the Customer develops buy-in to the solution and gains a sense of authorship, they will much more quickly accept the solution in the end.

### A Final Caution on Requirements

*This is a very dangerous time*

The requirements phase can be a lot of fun, with the SE team very excited about the opportunity to build something new and the Customer excited about getting to help specify something that they will soon get. This period can also get your project into trouble faster than any other, especially if the SE team does not understand the commitments they have to their organization.

*Don't allow your team to promise things that you or the Customer cannot afford*

Whether you have settled on payment by fixed-price or some version of time and materials, your SE team and the Customer rarely have blank checks. The Customer often has the feeling in these early meetings that they can ask for anything they want. The SE team wants to please the Customer but may find themselves in the uncomfortable position of having to tell them that their desires exceed the scope of the contract. In no case should the SE team lead the Customer to believe that they will receive something that the contract does not support or that they cannot afford. The best way to avoid this situation is to include contract representatives in these meetings, or at least as reviewers of agreements to act as safeguards against scope and cost creep.

## Step 3: Define the Scope of the Problem

*Set limits on where your system ends*

The next step in building the architecture is a fairly straightforward one. As the result of understanding what your system will do and with what other systems it will interact, you have developed the first iteration of what belongs inside your system and what does not.

### The Context Diagram

*What is in, what is out, and what is on the fence*

The Context Diagram is the standard tool for showing where your system begins and ends. As shown in Figure 7, the Context Diagram is a special kind of diagram that starts with a clearly defined box, which depicts the boundary of your system. Anything inside the boundary clearly belongs inside the system. Usually, the Context Diagrams contains the first iteration of the subsystems that the system will have and some depiction of how they will interact. On the outside of the boundary we find external systems with which our system must interoperate. The important of these systems lies in what we draw on the boundary of our system. These boxes represent the

formal interfaces that we must either develop or negotiate with the external systems.

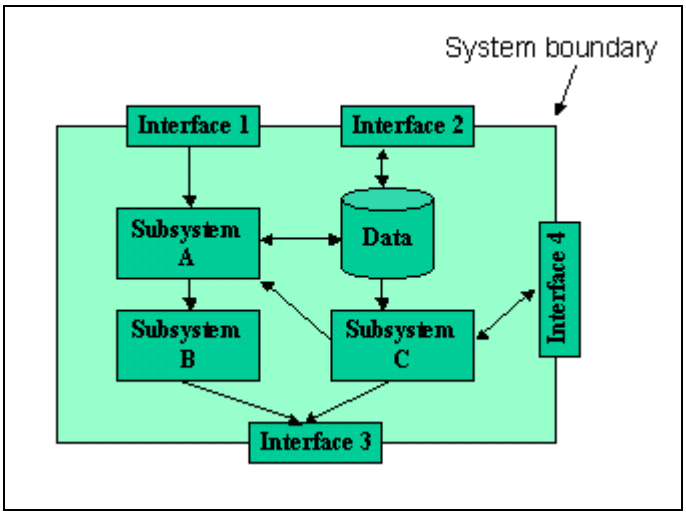



**Figure 7- The Context Diagram**

### Interfaces with other systems

*Formalize agreements with other systems*

By boxes on the boundary of the context diagram identify all the systems with which our system must interact. Each of these external systems will require some kind of interface that will require negotiations and agreements with whomever has responsibility for those systems. Your team must document each interface in an *Interface Control Document (ICD)* to detail the agreements, to include: information to exchange, frequency of exchange, protocols, ownership, read/write permissions, etc.

### Interfaces: another high-risk item

*Interfaces always cause problems*

Not surprisingly, interfaces become yet another one of your high-risk items. In part, this occurs because in most instances you have little control or influence over the external system with which you must interact. The operators of those systems may have little incentive to meet your schedule or otherwise support your project. Even after you have reached agreement, interfaces will change without warning, data will arrive with errors or in the wrong format or at the wrong time, and the interface will not work. Such is

the life of a System Architect.

## Step 4: Select Your Views

*Base your views on how you tend to describe your system*

Much has been written on how to select the optimal set of views for software architectures, but little exists for systems architectures. Even the IEEE sidesteps any specific recommendations in their standard [IEEE00]. On most of the projects on which I have worked, I based the views on the handful of diagrams that I routinely used to describe the system. Nothing works better than what comes naturally. Likewise, if you find yourself beginning every engineering discussion by drawing the same sketch on the nearest napkin, you should consider that sketch as the starting point for one of your views.

### Guidelines for selecting views

*A starting point might be the following four views*

Although the choice of views varies from project to project, I have successfully used the following views in a large number of projects. Of course, the name of a view might change to fit the norms of a particular organization. Nonetheless, the essential information conveyed by the view will remain fairly consistent.

1. Logical View
2. Data View
3. Operational View
4. Physical View

Sometimes we also refer to these views as sub-architectures, or simply "Logical Architecture," "Data Architecture," etc. For consistency, this book will refer to each as a *view*.

*The Logical View decomposes the problem*

The *Logical View* shows the major system components: hardware, software, interfaces, the network, and any other significant component. This view also shows the major functions of each component. In effect, the Logical View begins the process of decomposing the problem by assigning responsibility for major functions to subsystems.

*The Data View outlines the structure of*

The *Data View* is one view that few system architects use but that I have found essential on nearly every large computer system. The Data View outlines the future structure of any database along with the major

information flows. It shows data-intensive interfaces, the distribution of data, and addresses the rules that you will follow in the handling of data.

*any database*

The *Operational View* adds dynamics to the Logical View. This view shows the behavior of the system by depicting the data and control flows between components. In most cases, the Operational View will become the most used and therefore most important view of your system.

*The Operational View shows behavior*

The *Physical View* shows the physical components of the system and where they reside. The Physical View will depict all hardware, network components and interconnections, as well as the where all software and data will run or be stored.

*The Physical View maps the Logical View to devices*

### Why different situations require different views

These four views serve not only as a good starting point for most projects, but they will also serve as an outline for the remainder of this section. I have chosen these views not only because I have used them extensively, but also because I have found that they capture the essentials of System Architecture. Each major project will of course require special consideration in the choice of views.

*These four views capture the essentials of most architectures*

These special considerations will almost certainly result from the list of major risks and technical challenges that the Architect outlined in the Introduction to the SAD. For example, on a mission-critical system, the Architecture might require a view on availability. A banking system might require a view on security.

*Major challenges may require their own view*

Most importantly, choose only a small set of views. This allows the team to focus on just a couple of pictures of the system rather than allowing various perceptions to propagate. This enforces a common understanding of your system much more effectively than all the documentation you could possibly write.

*Choose what you need and no more*

### Views evolve over the life of the project

Choosing your views marks a milestone in the development of the System Architecture but it does not end the process. Validate your choices. Have you chosen too many? Can you eliminate any? For example, several guidelines on software architectures suggest having a *Standards View* to list the exact protocols and standards (i.e., SQL, TCP/IP) that the project will use. I have always found it sufficient to simply put that information into a

*Validate your choice of views*

memorandum and eliminate the need for that view.

*Continually refine your views…*

It may take a long time to develop your System Architecture. As you get closer to high-level design, your views will become more detailed and become consistent with each other. In other words, data in the Data View will be the same as that depicted in the flows of the Operational View. Developing the architecture might take 2-3 weeks on small projects (about 10k LOC) and 6-9 months on a large project (more than 700k LOC) [Kruchten95].

*…but maintain a stable Architecture*

While you will continually refine the architecture with more detail and small improvements, I believe that the architecture should become essentially stable after it has been developed and validated. If not, then you have certainly made decisions that will become cost ineffective to correct and therefore fatal to your project.

## Step 5: Develop the Logical View

*Break the problem into smaller parts*

The next step starts the process of breaking down the problem into parts and assigning each part to software, hardware, or other component. The Logical View is a depiction of *what* the system functionally does. It does not attempt to address how the system implements those functions.

### List the logical components

*List the major system functions*

Start with a list of the major things that the system must do. Next, group related functions into components. You might use a diagram such as the one shown in Figure 8. It usually becomes obvious which functions will be implemented in software, hardware, or by other means. However, this may change since this view does not prescribe implementation. Note that logical components that result in software will likely become classes or major executable programs in your system. The components that result in hardware will become computers, network routers, or similar devices.

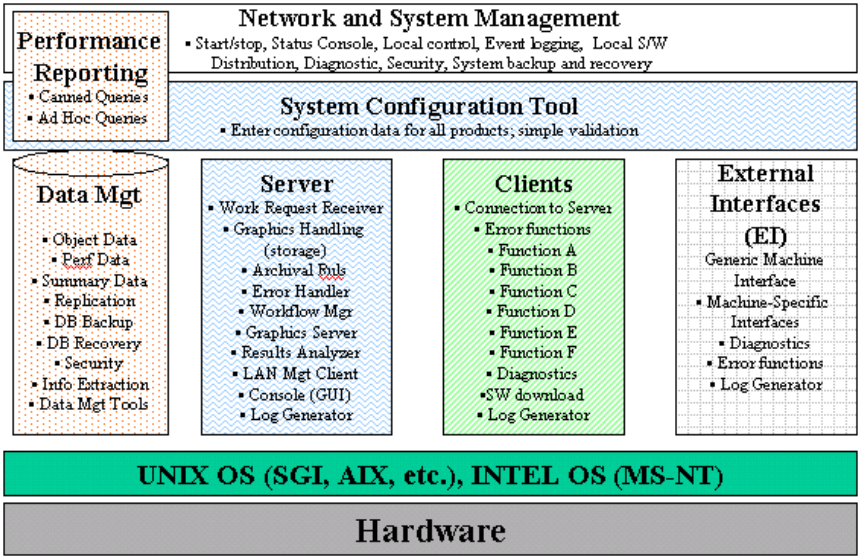



**Figure 8 - Example Logical View (Logical Architecture)**

### Allocate Requirements to Components

*Assign requirements to each component*

Once you have identified the components, you need to verify the Logical Architecture for completeness. In short, will the system do everything that it needs to do? To do this, create a cross reference of all the detailed functions or requirements that the system must perform and assign each to one of components that you have identified in the Logical View. A Requirements Management Tool may make this process easier, especially as you need to refine and change requirements over time. Smaller projects may find a spreadsheet sufficient.

*Check for completeness*

If you can successfully assign each system requirement in the spreadsheet to one of the components in the Logical View, then you have at least shown that the system can do all that it must. However, this process can become very complicated, especially when dealing with vague or hard to quantify requirements. As discussed earlier, hopefully the SE has been able to limit the vague requirements with testable criteria that you can assign to one or more components.

### Review and analyze alternatives

*Check your own work*

Have you considered any other alternatives? In any complicated system, many ways exist to approach a problem. This includes alternative ways to divide up the problem and thereby arrive at a Logical View. Can you think of a better way or simply other ways to approach the problem?

### Select a solution

*Document your rationale*

List any alternative approaches that you considered in the SAD. You should not elaborate on them in as much detail as the approach that you have chosen, but you should comment on the advantages and disadvantages of each. This effort further validates your chosen solution and documents the rationale for the solution.

### Conduct informal reviews

*Have others check your work*

At this stage you should feel sufficiently confident in the Logical View to circulate your diagrams and the draft SAD. Ask for reviewers beyond the persons who have already provided input. Ideally the reviewers will have experience in the subject area and will therefore be able to give you meaningful feedback on your initial breakdown of the problem.

## Step 6: Develop the Data View

*Data usually rules!*

In most systems, data rules. The flow or state of information dictates all the actions of the system. Therefore, it is important to address how your system will handle data from the architectural perspective. The Data View meets that need.

### Identify major data sources and stores

*Diagram data stores and flows*

Developing the Data View begins by identifying the major types of information in the system, where it comes from, and where it goes. The Data View shows the logical layout of all persistent data, whether in files or databases. The Data View also will identify the flows of data from each of its storage locations. Figure 9 shows an example depiction of a Data View for a Data Warehouse system. The cylindrical objects are the traditional icons for data stores- usually a relational database. The lines depict the

connectors between the applications and the data that the application uses.

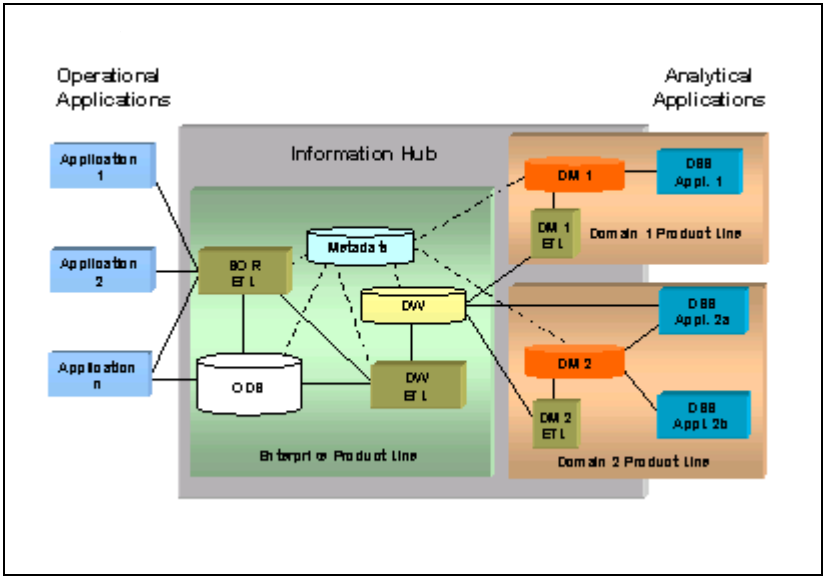



**Figure 9- Example Data View (Data Architecture)**

### Examine flows of information

*Analyze data flows for performance and control*

The next step in refining the Data View involves examining the information flows. This means looking at the types of data, frequency, and quantity of data in each of the flows. If you have a lot of data transferring very often, then you will have to increase your awareness of performance issues. Ensure that each external interface identified in the Data View also appears in the Logical View, and that plans to develop a formal interface agreement exist for each interface.

### Establish constraints on data

*Set policies for managing data*

The Data View includes much more than just identifying data. It also defines consistency constraints, replication methods, access controls, audit trails, recovery plans, backup policies, etc. For example, if the same information exists in more than one external system, which should we consider the authoritative source in the event of a conflict? Will the data require replication for either performance or availability? Will everyone

have access to the data all the time, or will the system restrict access to users based on their job function? How will the system recover from minor and major errors or disasters?

### Review and analyze alternatives

*Always review your work…*

As always, any complex system will have numerous possible solutions. This is especially true with the details of data recovery and availability. Ideally, the System Architecture will leverage options to optimize the overall solution. For example, mirroring disk drives on the physical storage devices may increase throughput at the same time that it significantly increases reliability. Of course this has a cost in terms of additional hardware.

### Select a solution

*…document your choice and rationale…*

The classic trade-off in databases is between *time* and *space*. Both involve cost, whether in taking longer execution time on the processors or in requiring more physical storage. Severely resource constrained systems may find it difficult to manage these tradeoffs (this includes systems that have to run fast but cannot afford large quantities of storage for any number reasons, such as space to put them, weight, or cost of the devices). In the end, the System Architect must work with the data experts on the technical team, weigh the options, choose and document the solution in the Data View of the Architecture.

### Conduct informal reviews

*…and let others validate the solution.*

Having completed the Data View, it is time to circulate the draft SAD (which now contains the Data View). Obtain comments from neutral and informed sources. Use the input to improve the solution and make the adjustments before making irreversible commitments to the technical solution.

## Step 7: Develop the Operational View

*Now give the system life*

I consider the Operational view as the most important view of the System Architecture. It shows not only all the components that we identified in the Logical View, but how they behave. For the first time we see the System Architect's concept of how the system will act and this additional

information seems to give life to the project.

### Show how components communicate

*Show data flows and resulting actions*

Whereas the previous views focused on *what* the system does, the Operational View shows *how* the system does it. The most common way to depict the Operational View is one of the many versions of a data flow diagram, such as the one shown in Figure 10. Boxes in this diagram represent the components from the Logical View (some hardware, some software, some machinery). Arrows depict data or control flows (unlike some diagramming conventions, this particular diagram does not differentiate between the two).

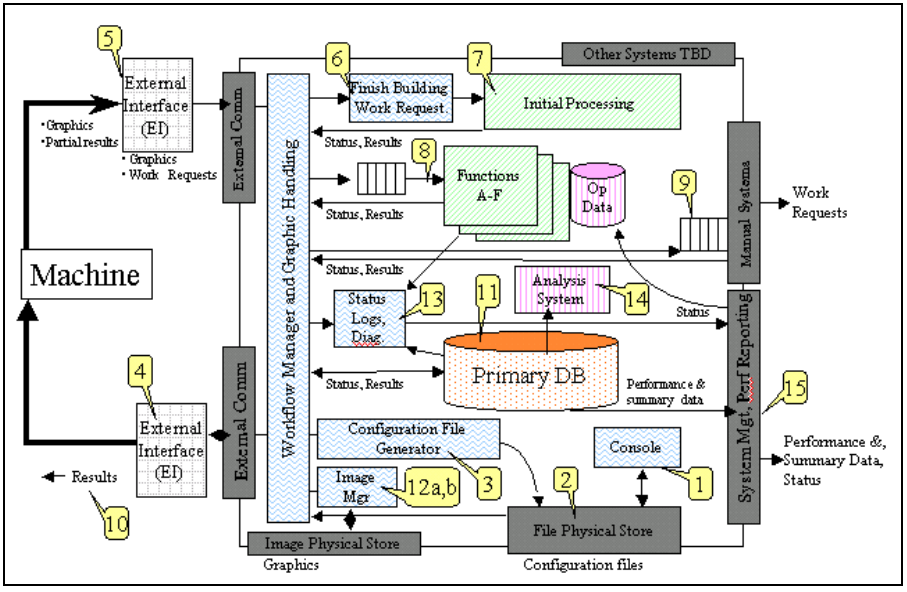



**Figure 10 - Example Operational View (Operational Architecture)**

*Build on the Logical View and Context Diagrams*

The Operational View also expands on the Context Diagram presented earlier in the SAD. As shown in the figure, this "data flow" diagram uses some of the same conventions used in the Context Diagram in that the major system interfaces appear as boxes on the boundary. These boxes should match the boxes shown in the context diagram. If they do not match, then the Architect must reconcile the inconsistency. Remember that in each of these iterations in the development of the System Architecture we

add detail to the solution and drive the views to consistency.

The yellow callouts in the figure walk the viewer through the actions in the diagram. Accompanying this diagram is a series of paragraphs that explain the data flow and actions that correspond to the numbers. This provides the first iteration on how the system works and the order in which the System Architect expects the actions to take place.

*Determine the initial sequence of events*

### Review and analyze alternatives

Once again the Architect enters the cycle of checking and validating his work. By this time, the Architect has developed a pretty good idea of what the system must do and how the system will do it. This involves considering all the alternatives and systematically reviewing the advantages and disadvantages of each.

*Review the options…*

### Select a solution

The selected approach will optimize the trade-offs between the available alternatives. Although the Architect has not committed to the solution, selecting the preferred approach in the Operational View is a major step because of the significant step it makes in setting the most likely technical direction for the program.

*…select the best…*

### Conduct informal reviews

This kind of step requires independent input and feedback. Corrections can still be made to the Architecture with little damage or cost. The Architect circulates the draft SAD (now containing the Operational View), explains the system as necessary, and adjusts the SAD accordingly. Once complete, he can start investigating how to actually realize the components in the Architecture.

*…and obtain independent feedback.*

## Step 8: Develop the Physical View

The Physical View makes the system real because it makes the system tangible. Abstract components from the Logical View become software subsystems or classes, physical computers, routers, machines, or procedures. Now we can see what the system will actually *look* like.

*The system becomes tangible*

Notice that in each step of developing the System Architecture, we have iterated on the system solution by adding detail and otherwise expanded on what the system will eventually become. The Physical View continues this process. Once complete, the Physical View often takes the place of the Logical View diagrams in day-to-day use. The Logical View remains in the SAD, but everything in the Logical View translates into some aspect of the Physical View and therefore the diagrams of the Physical View are generally more useful to the technical team.

*The Physical View expands the Logical View*

### Allocate the logical components to physical devices

Finally the System Architecture takes concrete form. The primary purpose of the Physical View is to map the components of the Logical View onto physical devices or things that we can otherwise implement, such as in software. Figure 11 shows a very simple Physical View of a simple System Architecture used by the large majority of Internet sites in the world. In this view we see the client personal computer (possibly located at someone's home) attached to the Internet by cable or telephone modem. The Internet site also connects to the Internet, but by using a router. The Physical View of this system shows two UNIX computers. One computer runs the application code with which the user interacts and the other computer stores the data involved in the transactions. The diagram also lists the software components that run on each computer.

*Abstract components become things such as software and computers*

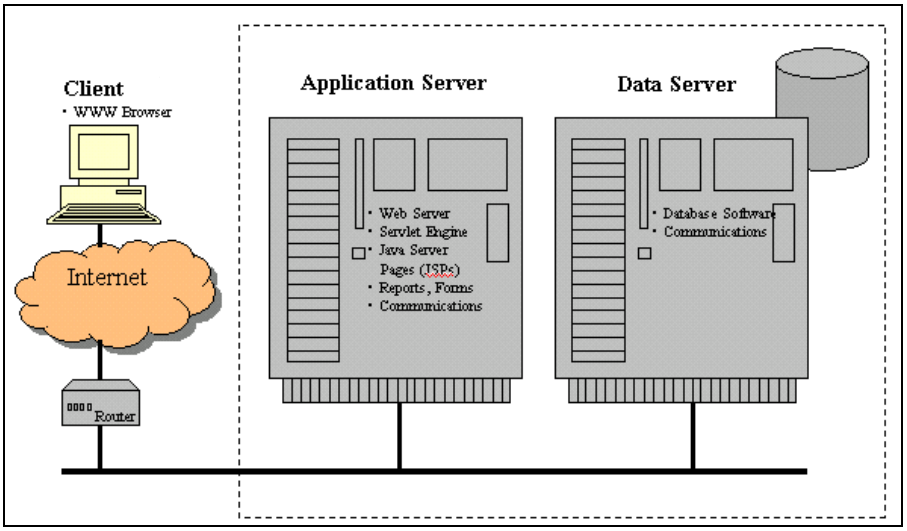



**Figure 11- Example Physical View (Physical Architecture)**

## Distribute Devices and Services

*Bandwidth is cheap*

Modern information systems offer many different options in terms of where processing actually takes place. Network bandwidth is so fast and the cost of communications is so low that the Architect can elect to physically place the computer processing power almost anywhere. Even wireless devices such as PDAs and cell phones have access to enough bandwidth (and it is increasingly rapidly anyway) to allow distribution of data for processing wherever the computers happen to be.

*Place the computers and people where they can function best*

Whereas Figure 11 shows a simple physical implementation, large systems have many more variables and options. The choice of implementation platform many range from personal computer to mainframe, depending on technology trends [Schuff01]. As shown in Figure 12, large projects may have parts of the development team in one place and parts in others, thereby requiring wide area connectivity. The development may take place in a location independent of the Customer site and require connectivity. The system solution may dictate a distributed architecture. Continue to refine the Physical View until you have determined the physical locations and connectivity option for all components in the system.

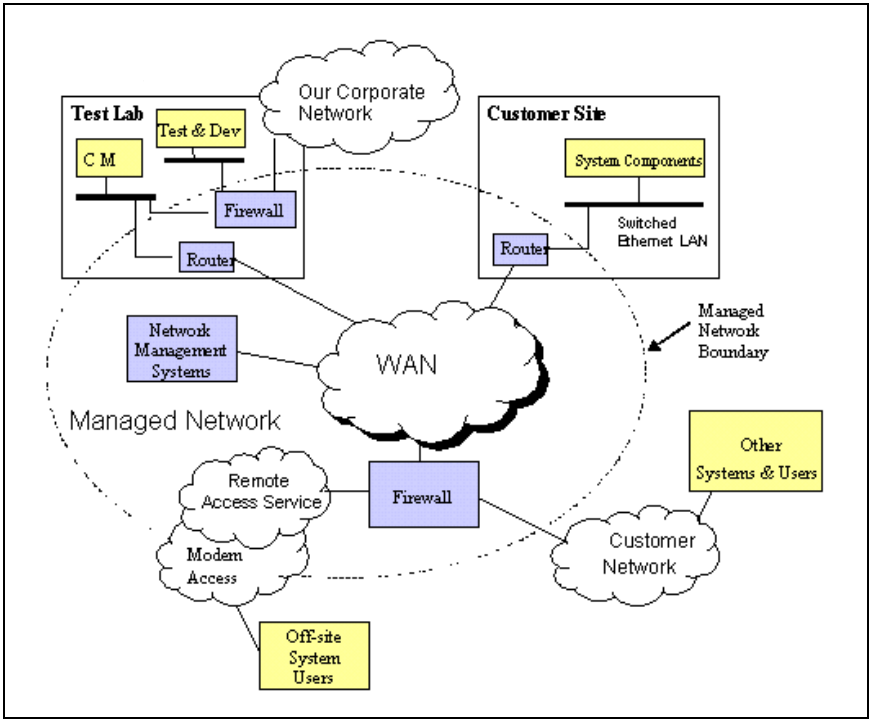



**Figure 12 - Distribute Services across the Network**

### Set Direction for the implementation

The Architect uses the Physical View to outline strategic direction for the implementation. The implementation of some components can easily be filled with commodity items such as personal computers. More serious processing requirements may require further study to determine the optimal platform (such as the choice of UNIX and then a preferred vendor).

*Determine how to implement each component*

### Review and analyze alternatives

Each component effectively must undergo a "make or buy" decision. The System Architect will have to make or subcontract for nearly all custom delivered function, whether implemented in hardware or software. On the other hand, the use of COTS products can usually provide a very cost effective means of implementing the system. Some COTS products have

*Decide whether to make, buy, or compete each component*

virtual monopolies within their product suite, which pre-determines the decision outcome. Other COTS products must fiercely compete for market share, in which case the Architect will usually commission a formal analysis of the alternatives (e.g., a "trade study") to assist in the final selection.

*Establish rules and constraints*

These final decisions must resolve any remaining fundamental architectural issues. For example, should the architecture adhere to fundamental design principles such as information abstraction, and if so, how? Do software components communicate solely by procedure calls or do they use messaging, queues, or third party middleware?

### Consider performance of devices

*Ensure devices can handle the load*

The selection of physical devices necessitates ensuring that the devices can handle the load that the system will place upon them. It would be irresponsible to send a personal computer to do a mainframe's job. The same goes for bandwidth requirements on the network whenever large amounts of data are involved. During development of the Physical View, the Architect should be concerned with the order of magnitude of what the device will need to do; for example, whether to assign the task to a PC or to a UNIX server.

*Gather data and size devices*

Sizing devices requires a significant effort. The Architect works with the Lead System Engineer to gather data on CPU capabilities, memory requirements, and capabilities of the available hardware options. They require similar data on the network bandwidth and latencies. These must be matched to the system timing constraints and user response times required by the Customer. The System Engineering team will then assist in the selection of the exact processor within the product family identified by the Architect and Lead SE.

### Select a solution

*Choose the best option…*

At this point the System Architect has completed the last of the views of the System Architecture, and therefore considered all the possible options for dividing up the project, managing data, detailing how components will interact, and assigning those components to physical devices.

### Conduct informal reviews

As before, the System Architect documents the decision in the SAD and solicits input on the work to date. The SAD now contains the fundamental System Architecture for the project and is ready for the next iteration of refinement.

*… and validate the choice.*

## Step 9: Develop a Concept of Operations

The Operational View depicts the behavior of the system in one or a few diagrams. However, the Operational View cannot convey all the behavior that will take place in the system [Kruchten95]. This requires a fair amount of explanatory text that covers the normal system operation as well as the operation of the system in unusual situations, such as the failure of one or more of the components. The Concept of Operations (ConOps) is, therefore, a companion to the SAD. The ConOps usually takes the form of a text document of 10-25 pages, more or less depending on the complexity of the project.

*The ConOps is a companion to the SAD*

### Define the vision for how the system will operate

The ConOps continues the iterative refinement of the System Architecture by adding detail to the views in the SAD. Specifically, the ConOps expands the Operational View to include the many variants of process flows beyond the nominal cases. The ConOps will specify component behavior as well as system behavior in each of these scenarios.

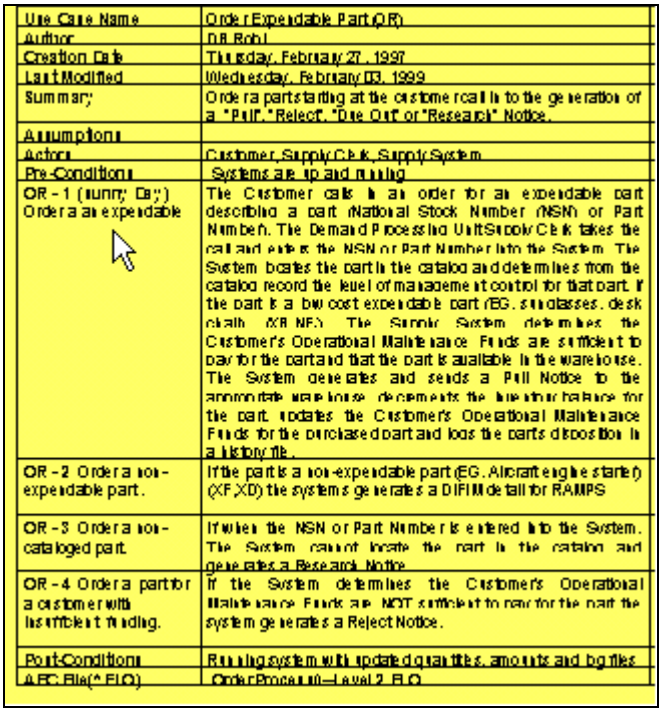
*The ConOps adds detail to the Operational View*

For each scenario, the ConOps explains as clearly and as concisely as possible what the Architect wants the system to do. The Architect should write the ConOps in the active voice because the passive can leave questions as to who has responsibility for executing an action. The ConOps should include scenarios for all the variants of actions in the system, to include special cases, partial and total system failure, and error handling.

*Describe special cases and errors*

As an option to writing a text document, the Architect may also elect to use *Use Cases*, as shown in Figure 13. Use Cases provide another systematic method to enumerate the possible system scenarios and, depending on the development methods used by the organization, may fit very well with the organization's requirements analysis process and tool support.

*The text of Use Cases can serve as the ConOps*



| Use Case Name | Order Expendable Part (OR) |
|---|---|
| Author | Bob |
| Creation Date | Thursday, February 27, 1997 |
| Last Modified | Wednesday, February 03, 1999 |
| Summary | Order a part starting at the customer call in to the generation of a "Pull", "Reject", "Due Out" or "Research" Notice. |
| Assumptions | |
| Actors | Customer, Supply Clerk, Supply System |
| Pre-Conditions | Systems are up and running |
| OR - 1 (sunny Day) Order a an expendable | The Customer calls in an order for an expendable part describing a part (National Stock Number (NSN) or Part Number). The Demand Processing Unit/Supply Clerk takes the call and enters the NSN or Part Number into the System. The System locates the part in the catalog and determines from the catalog record the level of management control for that part. If the part is a low cost expendable part (EG. subclasses, desk cloth (XB NF)) The Supply System determines the Customer's Operational Maintenance Funds are sufficient to pay for the part and that the part is available in the warehouse. The System generates and sends a Pull Notice to the appropriate warehouse, decrements the inventory balance for the part, updates the Customer's Operational Maintenance Funds for the purchased part and logs the part's disposition in a history file. |
| OR - 2 Order a non-expendable part. | If the part is a non-expendable part (EG. Aircraft engine starter) (XF XD) the systems generates a DIFI Mode tail for RAMPS |
| OR - 3 Order a non-cataloged part. | If when the NSN or Part Number is entered into the System. The System can not locate the part in the catalog and generates a Research Notice |
| OR - 4 Order a part for a customer with insufficient funding. | If the System determines the Customer's Operational Maintenance Funds are NOT sufficient to pay for the part the system generates a Reject Notice. |
| Post-Conditions | Running system with updated quantities, amounts and log files |
| ARC File(^ FLO) | Order Procedure—Level 2 FLO |

**Figure 13 - Explanatory text of a Use Case**

### Validate the vision with the users

As always, the Architect must validate his vision for the system with the technical team and with the Customer. Because of the additional detail provided in the text document or Use Cases, this will take more time than it took for the informal reviews of the draft SAD. It can also be a very tedious review, since in the attempt to make the text as clear and concise as possible, the reviewers will often find themselves arguing over the definition or connotation of individual words.

*The ConOps review is tedious*

Since reviewing the ConOps is tedious, we cannot expect a general audience to participate. Nonetheless, the Architect will have to present the document to this type of audience. The Customer will want to understand how the new system will change and benefit their current operations.

*A general audience needs to participate*

Putting a text document into pictures can be a challenge. In some situations

*Relate the*

*ConOps to the Customer's operations*

I have found that an effective method to help the Customer understand and become at ease with upcoming changes is to put the ConOps into a slide show as shown in Figure 14. For each scenario in the ConOps, take a digital picture of the Customer's operation affected by the scenario. When showing that slide, explain first how the operation takes place today and the shortcomings of the current operation. Then explain the proposed change and how that change will benefit the operation. This puts the ConOps firmly into a context that the Customer understands and helps them relate to the business benefits that they will receive.

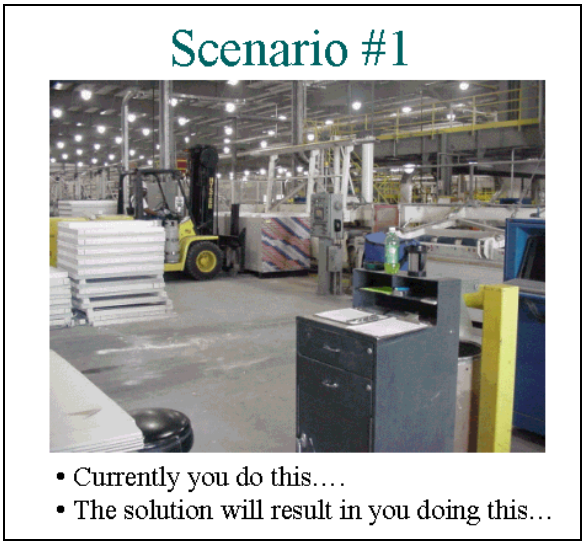



**Figure 14 - Presenting the Concept of Operations**

*Presenting the ConOps can be fun*

If the ConOps describes the actions of software, the visual portion of the presentation might include screen captures of the Customer's current applications and mock-ups of the proposed applications. The pictures might also come from the Use Cases diagrams captured during requirements analysis. Presenting the ConOps can be fun as well as an important means of communicating the proposed actions of the system.

## Step 10: Address Key Issues

*Resolve remaining*

The final steps in the development of the System Architecture continue to add detail to the previous work, now all contained and documented in the

SAD. As with the previous iterations, most of these issues require inputs from a wide variety of technical personnel and skill sets. The following sections highlight just some of the possible issues that will require attention.

*technical issues*

### Timing constraints

System Architecture requires an awareness of how all parts of the system operate together. Many systems have dependencies on the sequence of detailed events and the actions that must take place as a result. This goes beyond the behavioral flow depicted in the Operational View or ConOps, but rather gets into how long a mechanical mechanism might take to react to an impulse provided to it from a computer control device. Does the control system require a response within a certain time out period? Whether or not the response takes place, what actions should occur and by when?

*Create timelines for system events*

One of the most difficult systems to build is a real-time system. Real time systems must process data and control inputs within a specific time of their occurrence. Examples include air traffic control, airplane and missile guidance, and robotic control systems for factories. In each case the Architect must ensure that the combination of hardware and software will react to inputs according to a strictly defined schedule.

*Real-time systems depend on timely responses*

As shown in Figure 15, sequence diagrams are normally used to show these dependencies. The boxes in the diagrams can represent components such as software processes or physical devices. The arrows depict the messages between the components, with the sequence of the messages determined by the order of the arrows from top to bottom in the diagram.

*Sequence diagrams show these dependencies*

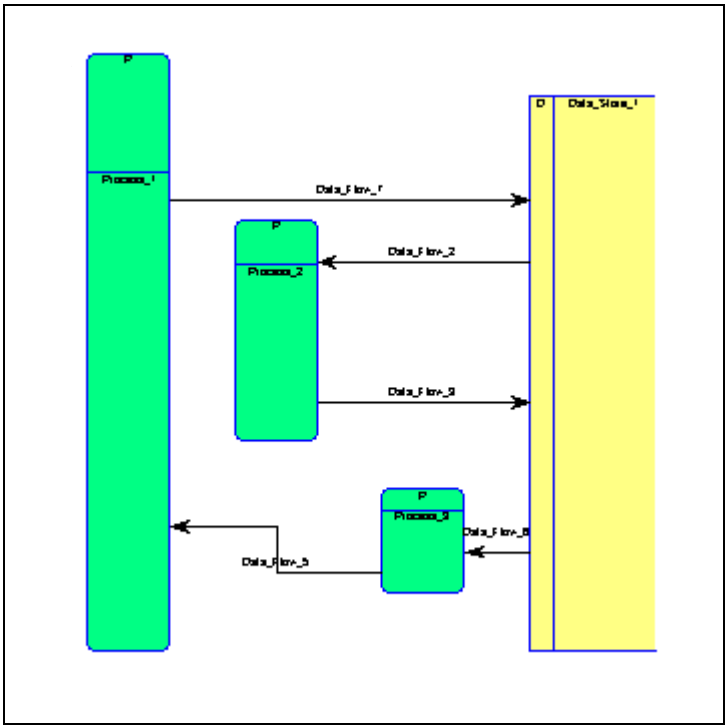


**Figure 15- Sequence diagram show the flow and order of messages**

### Performance allocations to subsystems

*Allocate resources to each subsystem*

Not only much each subsystem expect to execute in the order defined by the sequence diagram, they must also know how much time and resources they are allowed for their execution. The Architect must allocate the resources that he will make available to each subsystem, such as cost, memory, disk, bandwidth, latency and throughput. For response times, insert the timeline for the messages (for example, in seconds or milliseconds) adjacent to each arrow in the sequence diagram. The technical leads will undoubtedly want more resources than the Architect can afford to give. The Architect must therefore balance the requirements of all subsystems along with the total available system resources when determining these performance allocations.

### Interfaces

*Assign owners to each interface*

Both the Context Diagram and the data flow diagram in the Operational View depict the external interfaces required by the system. As previously noted, each interface will require a formal specification to record the types of data, frequency, timings, and all other issues related to the exchange of information between the two systems. One system engineer should own each interface. Depending on the complexity of the interfaces, the SE may own several interfaces. However, the SE must take responsibility for researching and resolving all the issues related to this link between the two systems and documents the agreements in the Interface Control Document (ICD).

### System States

*Determine allowable system states and transitions*

Many systems will be required to exhibit behavior that depends on prior inputs. All inputs, therefore, put the machine into a pre-determined *state* from which it will transition to other states based on a particular input. Perhaps the most common example of this is the transition between "off," "on," and "test modes" as shown in Figure 16.

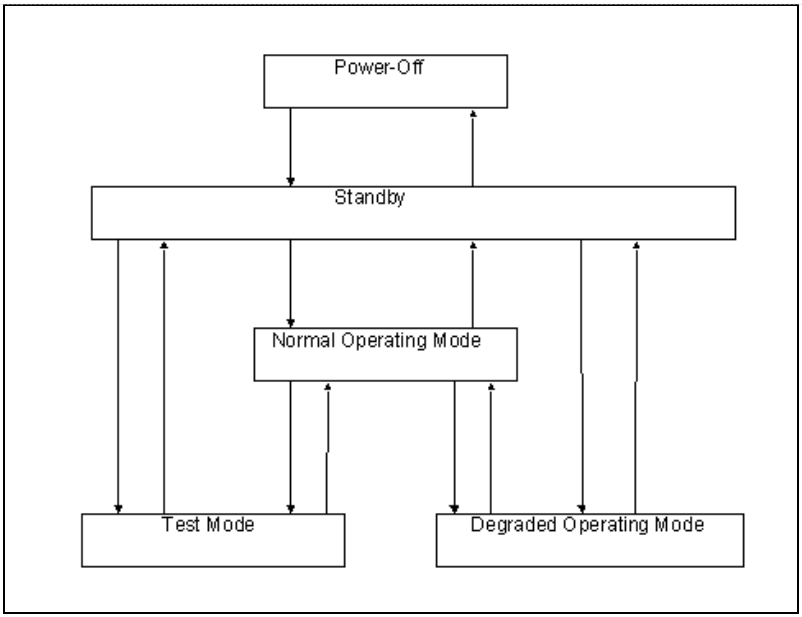



**Figure 16 - Allowable system states and transitions between states**

Considerations in each state might include the recording of various system information when in test mode, for example, writing hardware status (e.g., CPU utilization, RAM utilization, paging rates, etc.) and system events (e.g., writing to files, execution of software subsystems) to special diagnostic logs. When in degraded mode, the system might re-allocate resources to processes to ensure critical systems continue to function, while at the same time suspending ancillary functions such as audit logs, backups, and data archiving.

*Specify actions unique to each state*

## Security

System security can be a very difficult issue, especially when dealing with military (i.e., top secret) or highly sensitive (i.e., banking, medical) applications. The options and technology change rapidly and therefore the System Architect will depend on the expertise of a security expert or consultant in these situations.

*Control access to the system*

Most information systems and personalized applications, such as a shopping cart at a web site, require "routine" access controls such as those routinely provided by operating systems, applications, and databases. Operating

*I&A and roles meet most security needs*

systems and applications will prompt the user for an userid and password, a process called *Identification and Authorization.* The person providing both is assumed to be the correct user. The userids and passwords are stored by the system in a protected and/or encrypted area. In addition, the application or database may associate the user with a *role*, such as a clerk or supervisor, and grant those roles access to different functions and data.

*Special software and encryption add further safeguards*

The system may require specialized software for access and may encrypt data transmissions on the network in order to defeat malicious eavesdropping attempts. A large number of telecommuters today use *Virtual Private Networks (VPNs)* for remote access. VPNs use special software, encryption, and possibly hardware assists to guarantee only authorized users gain access to systems. All of these are common practices.

### Technical Performance Measures (TPMs)

*Decide how to monitor system progress*

All critical aspects of the system require regular monitoring. It is not sufficient to rely on subjective assessments of progress or status without eventually becoming a hostage to the project that is perpetually "90%" done. Critical items might include maximum response times, maximum processing times, network utilization, mean time between failure, and items directly related to contract payment milestones. Each of these items deserves regular attention by means of an objective *Technical Performance Measure (TPM).*

*Monitor TPMs and adjust resources accordingly*

Collect TPMs on a weekly or monthly basis, as required, and plot them on a graph that clearly compares the current state of the TPM as compared to where the TPM needs to be at the current time in the project. If the project appears in jeopardy of meeting a TPM, then the project will have to adjust priorities in time for corrective action to have its effect.

### Establish the migration plan

*Explain system deployment*

Many projects fail to plan for a smooth transition from the Customer's current systems to those that will replace them. In most cases, the two system will need to coexist for some period. If this is the case, the Architect needs to establish rules governing how the Customer will migrate to the new system.

*Data usually causes the*

Most migration issues result from data. While it is relatively easy to perform a one-time load of data from one system to another, maintaining

two copies of the data during a roll-out period is always problematic. The Architect must decide which system has "master" versus "slave" authority over the data, how information will flow between the two systems to ensure data consistency, as well as a myriad of other issues.

*problem*

### Consider alternatives, document rationale

This incremental refinement of the Architecture involves consideration of numerous technical options for each issue. In all cases, the SAD should contain a summary of the options considered and the rationale for the selected course of action. Any difficult issue will run into problems at some point of its implementation. At these times, someone will ask "why didn't we do it this other way?" The SAD should contain the answers to those types of questions.

*Record rationale to save work later*

## Step 11: Add Supporting Information

Large projects probably suffer as much turmoil from personnel issues such as communications and coordination of effort as they do from technical challenges. Hopefully, the System Architect has taken a structured and methodical approach to developing the architecture and validating it for feasibility and overall acceptance. Now his burden becomes to coordinate the efforts of the technical team as they further refine and implement the system. Well-defined and understood tools and processes, as well as strong communications and leadership, are the keys to this phase.

*Provide final coordinating instructions*

### Tools

Engineers of all kinds seem to love tools. This is no truer than in the fields of software and systems engineering. However, I have found that in most cases a large number of specialized tools require more effort in license fees, training, and use than the benefit that they deliver. I much prefer to have the technical team learn to use a small set of general tools very well.

*Consider tool support across the lifecycle*

When people love tools they also can become religious about the relative capabilities of particular tools. An additional benefit of using a small selection of tools is that the team can make the selection of tools and get to work sooner. Figure 17 shows one way of depicting the tools that the team will use and what they will use them for (note that unlike in a classic

*Choose a small set of tools*

waterfall lifecycle, each phase may happen many times, depending on the project).

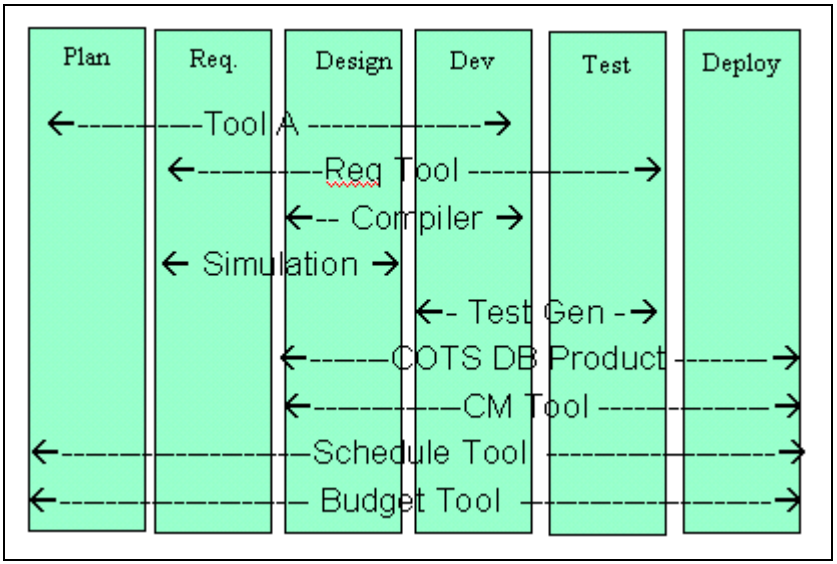



**Figure 17- Tools used across the lifecycle**

*Everyone uses requirements tools*

Requirements tools can range from simple spreadsheets to more robust databases that provide specialized reports, track changes, and provide many other functions. Everyone from the Lead SE to the test team typically uses requirements tools.

*Simulation can reduce risk*

Simulation can allow designers to test the feasibility of a solution before actually implementing it, as well as aid in system sizing. Simulation requires a significant amount of analysis and preparation before a reliable and faithful model of the system can be produced. Therefore, it is not cheap. However, it is almost always more cost effective to build simulations of a system when the system itself is very expensive, such as in hardware and integrated circuit design. This reduces the overall risk that the solution will work or will perform as expected under load.

## Processes

*How much process do you*

Coordinating and controlling a large number of professionals, each of whom has their own ideas about how to do each activity on a project, is a

little like herding cats. There are two extremes of thought on the level of control that a project should have, especially over software development. For ease of explanation, these extremes consist of those that believe in maximum control over a project and those that believe in almost no control.

*need?*

The most right-wing position would prescribe detailed processes and plans for every activity in the lifecycle. The thinking is that any project can succeed if it follows the formula, even if the staff consists of relatively low skilled personnel. The increase in control leads to greater predictability and lower overall risk, but it also inflicts significant overhead and stifles creativity.

*Does a detailed plan guarantee success?*

The most left-wing position would eschew any form of control beyond a broad vision statement. The ideal realization of this position would have developers miraculously hacking perfect code at an unbelievable rate of productivity. The decrease in control leads to significant flexibility and creativity, but also lacks the classic attributes of "good project management" evident in the more right-wing position.

*Does a team of mavericks beat disciplined development?*

Of course the right answer for any particular project lies somewhere in between the extremes. Excessive process constraints inhibit progress and cannot effectively handle even minor changes in personnel, technology, or situations [Boehm02]. On the other hand, the most agile of processes highly depends on the individuals on the team; such an approach must have high-quality people. On all of my projects, I have found existing process guidelines very flexible and have been able to tailor required processes to fit project needs. In some cases we have omitted them entirely. On a recent project, our mantra was "if it doesn't make sense, we won't do it." Having this "common sense" check served us very well on many occasions.

*Apply common sense to all controls*

With that said, it would be hard to imagine a project of any significant size without some basic controls. The most critical of these probably is Configuration Management (CM), which centers on keeping everyone on the team synchronized in their activities. This means harmoniously upgrading everything from development, test, and production hardware, to operating systems, COTS software, COTS patches, and application software (whether in development or in the field). Another key process involves the management of problem reports at every point in the lifecycle. The following sections discuss these in more detail.

*Every project needs some basic controls*

CM involves the coordination of effort of any large team over time. Each subsystem in the system will evolve at a different rate. Versions of COTS

*CM is all about*

products will change as vendors enhance the products and repair bugs. Computer and networking hardware becomes "old" within a year and nearly obsolete in three years.

*coordination*

The Architect needs to establish policies regarding how and when to upgrade hardware and COTS products. For example, he might require a systematic test of new operating system patches on one development machine before allowing the patch into the test environment and then subsequently into the field. The Architect must identify when to purchase hardware based on the deployment plan and the overall project schedule. Projects that run over three to five years will probably require a hardware refresh at some time during the project.

*Coordinate with clearly defined processes*

Figure 18 shows a process intended to coordinate the flow of software during its testing cycle. As the software undergoes progressively more comprehensive testing, from development, to software test, to integration and test with hardware and other system components, it becomes more stable. Passing each level of test marks a well-defined milestone and the software physically progresses to the next stage of testing. Everyone involved in the test process needs to understand and adhere to such a process.

*Ensure everyone understands and follows the process*

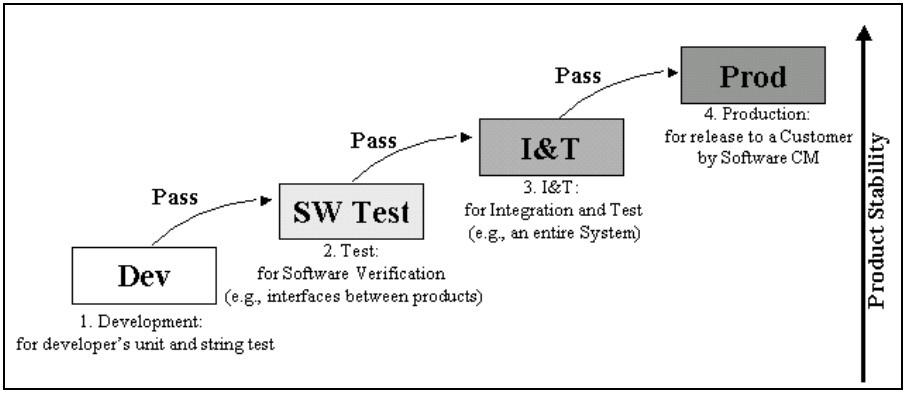



**Figure 18 - Configuration Management of software during test**

Another common process might be how the team should handle trouble reports. Throughout each stage of testing (to include the deployed software in the production environment), testers and users will report problems of various severities. Coordinating the analysis and repairs of problems will involve another process such as the one shown in Figure 19. The number

*Define processes where needed to guide the*

and types of processes that a project needs depends on many factors. The important thing is that the burden for coordinating the efforts of the technical team falls on the System Architect.

*team*

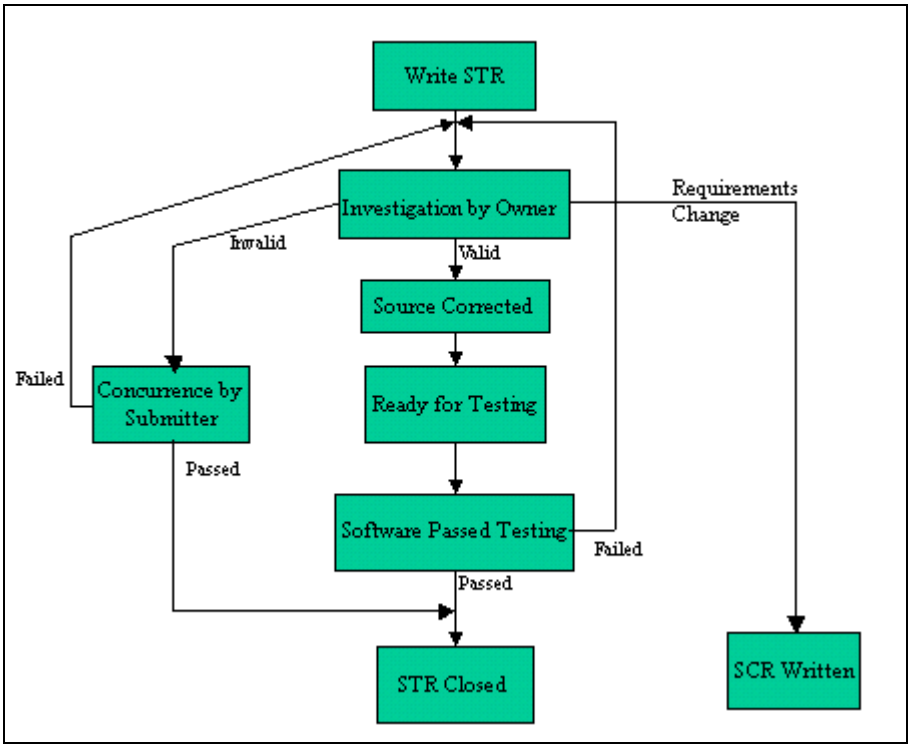



**Figure 19 - Process for Software Trouble Reports (STRs)**

The final touches on the SAD will include any features that the Architect feels will make the SAD easier to use and more accessible. A glossary of acronyms is helpful, as is a change history to make updates easier to consume. Add cross-references to sources of detailed plans, such as for CM or trouble reports, or simply to provide pointers to more information on a particular topic. The goal is to make the SAD as user-friendly as possible.

*Add information to increase usability*

## Step 12: Formally Review the Solution

Throughout the development of the System Architecture, I have suggested that the Architect repeatedly seek input and informal reviews. This serves

*Informal reviews have*

many purposes, most of them good. The benefits include getting the best possible expertise involved early in the project, when it can most influence the System Architecture and result in a better overall solution. The informal reviews also gain buy-in and support from potential reviewers of the system. Now that the time has come for a formal review, the political aspect of soliciting input usually pays off because the reviewers will understand the process, rationale, and conclusions for the System Architecture even before they arrive.

*set the stage*

### Technical Solution Review (TSR)

The formal review of a System Architecture is called a *Technical Solution Review (TSR).* A TSR may last from several hours to several days, depending on the complexity of the system. The Architect will typically do a stand-up presentation of the System Architecture at the TSR using the diagrams and pictures in the SAD.

### Communication!

The formal part of the TSR should involve a panel of formal reviewers, most of whom probably provided informal input to the SAD during every step of its development. The reviewers should receive the SAD at least a week in advance of the TSR (depending on their schedules) to allow adequate time for them to read it and formulate questions. If the Architect heeded the informal device given to him earlier, the TSR should go smoothly.

*Preparation greases the formal TSR*

So why all the emphasis on having a formal review? Simply because the audience for the TSR will also include every stakeholder possible. This includes management as well as technical personnel. Whether or not the TSR marks a formal project milestone (for example, one that involves payment from the Customer), presenting the solution in understandable terms to the people that must live with it and pay for it has immeasurable value towards building support and buy-in for the project.

*The TSR is key to building support for the solution*

### The Architecture Board

High-level design begins upon completion of the TSR. It is at this point that the subsystem leads, network designers, and hardware designers begin to refine their subsystems. They will all undoubtedly encounter problems and issues as soon as they begin. As long as the issue involves only one

*The Architecture Board resolves technical*

subsystem the System Architect can resolve the issue with the subsystem owner. However, most of these issues will involve more than one subsystem. To do this, the System Architect chairs an "Architecture Board" made up of all the subsystem leads.

*issues for the life of the project*

The Architecture Board typically meets on a weekly basis as a standing meeting, complete with a published agenda and meeting minutes to document decisions. I recommend a simple set of rules for the Architecture Board meetings: discuss *technical* issues of interest to *most* of the subsystems. If an issue only affects two subsystems then it can best handled in a meeting between those two subsystems. Likewise, if the Architecture Board lacks a meaningful agenda, then it should be canceled out of respect for everyone's time.

*It resolves technical issues that effect the system*

# 4 Examples from Production Systems

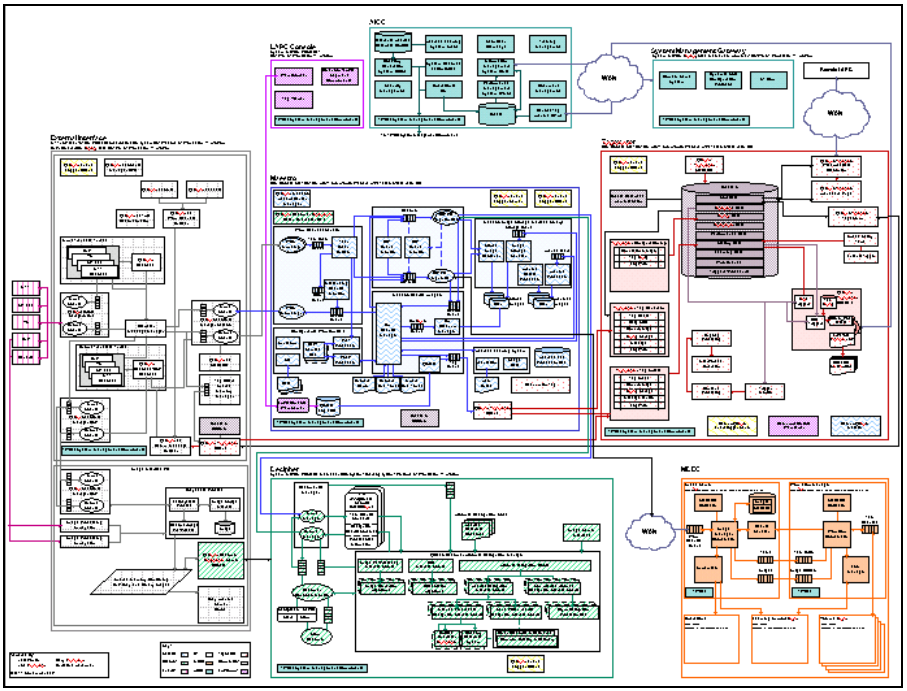


**Figure 20 - A Realistic Operational Architecture Diagram**

*I WILL PROBABLY NOT INCLUDE THIS CHAPTER!!!!*
*- you can't read the details on something this small*
*- it could get out of hand*

*- I have more, but…. They don't seem to fit the book. They go great in the presentation, however.*

*This figure might make a good cover for the book!*

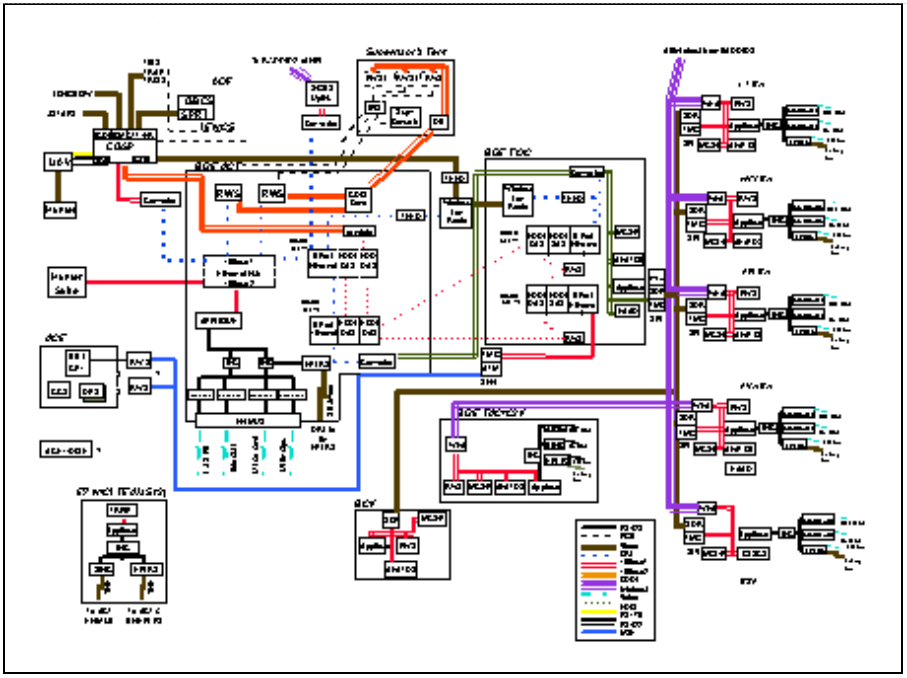


**Figure 21- A Realistic Physical Architecture Diagram**

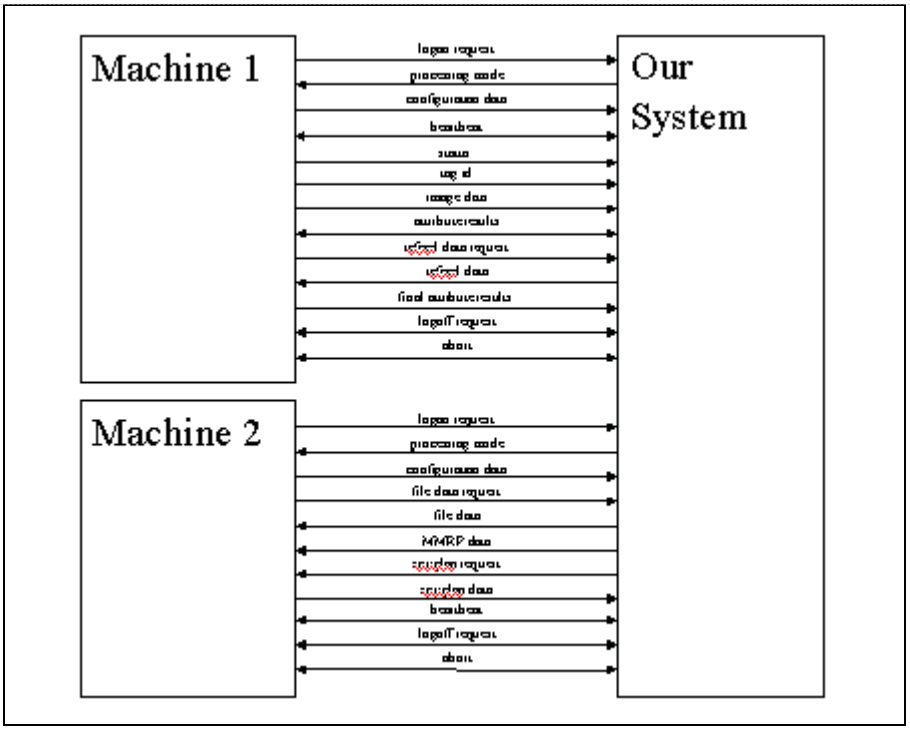



**Figure 22- A Realistic sequence diagram**

# 5 Conclusion

## The System Architecture "Elevator Pitch"

*System Architecture defines the system*

System Architecture is all about the highest-level organization of a system. The term appears frequently but in many contexts and at many levels of detail. This book attempts to outline the responsibilities of a "typical" System Architect on a large project and to show what goes into a "typical" System Architecture.

*The System Architect is the technical lead*

The System Architect is the technical lead of the project. He or she works closely with the project manager and with all the technical leads. Therefore, not only does the System Architect require a well-rounded and in-depth base of technical knowledge and experience, he or she must be able to demonstrate that base through communications and leadership.

*The SAD documents the System Architecture*

The System Architecture takes form in the System Architecture Document (SAD). The SAD serves as the common and authoritative source for high-level technical issues. Developing a System Architecture and documenting it in a SAD can take from several weeks to many months.

*The SAD may contain several views of the system*

The SAD contains a series of *views*, each of which contains a diagram or diagrams that depict a particular aspect of the system. Recommend views include:

- *Logical View*, to show the major components in the system
- *Data View*, to show how the system handles data
- *Operational View*, to show how the system behaves
- *Physical View*, to show how the major components will be implemented and where they will run

The Concept of Operations (ConOps) document serves as a companion to the SAD by detailing how the system behaves.

*The Architecture Board maintains the SAD*

Once complete, the core of the architecture should remain fairly stable, because changes at this level can be cost and schedule prohibitive. However, the architecture still requires continued attention from the technical leads of the project, which make up the project's "Architecture Board." The Architecture Board regularly meets to address technical issues

and steer the project to successful completion.

# 6 Further Reading: Software Architecture in a Nutshell

Software Architecture has been around a lot longer as a topic of study than System Architecture. In part, this stems from the difficulties and mysteries surrounding software development. In contrast, the tangible nature of classic system issues of hardware and networks seemed easy. However, predominant thinking changed once people realized the extreme complexity of combining all these aspects into a working solution.

*The study of software pre-dates the study of systems*

Much of the study and research work in Software Architecture comes from Mary Shaw and David Garlan [Shaw96]. Their book presents the classic styles of software systems tend to follow. It also presents the classic ways to *view* those styles by focusing on one item of interest. We'll start by reviewing some terms and definitions, and then jump right into what have become known as the "Three Cs" of Software Architecture.

*System Architecture expanded the principles of software architecture*

## What is Software Architecture?

Software Architecture depicts the highest level of organization of the *software* portion of a system. In that regard, it presents the "Big Picture" of how the software is organized and therefore, quite often the major functions seen by your audience.

*"Big Picture" of the software*

Like the System Architecture, this audience includes managers, maintainers, developers, and especially Customers. The Software Architect inevitably spends a significant portion of his time communicating the software architecture and its details to all of these audiences, just as the System Architect spends time communicating system issues.

*It targets a wide audience*

### The software approach to the solution

Note that the software only involves things written in code. It does not include the hardware on which that code runs, nor any other physical attributes networks or other devices. Think of the software as the functions that the system architect has elected to provide by writing code.

*Software delivers some of the system requirements*

### One part of the greater system solution

The System Architect will choose to deliver many of the other system

*Software*

functions in hardware, networks, or in other ways. Of course, the software has to interact with many of the other parts of the system. For example, if the System Architect tells the Software Architect that the software must execute with a response time of 1.5 seconds, the Software Architect will have to know such things as the transaction rates of the system and the processing power of the hardware upon which his software will run.

*interacts with the rest of the system*

### Definition of Software Architecture

Although I generally dislike citing definitions, I find it important to start with this one for Software Architecture because it forms the basis for the one for Systems Architecture that we will see in the next chapter. Software Architecture is:

*Architecture is not design*

> "The structure of the components of a program/system, their interrelationships and principles, and guidelines governing their design and evolution over time [SEI94]"

In addition to this definition, we widely accept that architecture is not design. In fact, design begins where architecture stops. This might seems a bit fuzzy, especially if you consider the highest-level organization of the system as the "system design." However, at this level we intend to describe the abstract software functions and guidance with which the Software Architect must work to deliver his portion of the system solution.

## The "Three Cs" of Software Architecture

Garlan and Mary Shaw expand on the definition of software architecture, explaining that software architecture consists of software components connected together by connectors, subject to a set of constraints. The following sections explain each of these "Three Cs" in turn.

*Components, Connectors, and Constraints*

### Components: The software sub-systems

Components are the objects in a software system. For those familiar with software, components typically become the major software subsystems in the software solution. This could become major classes, functions, procedures, or packages, depending on the implementation language. Software Architecture does not specify how to package the components. At

*Components are the software subsystems*

this level, we only concern ourselves with what we want the component to *do*.

### Connectors: How the sub-systems communicate

*Connectors tie components together*

Connectors are how components interact. Connectors are quite often simply messages between components (e.g., in an object-oriented language) or possibly function or procedure calls (e.g., in most third generation languages such as C). Connectors can also involve "middleware" solutions such as queues (e.g., message-oriented middleware) or object solutions (e.g., CORBA, COM). In any case, connectors allow the components to talk to each other.

### Constraints: The rules software must play by

*Constraints govern component interactions*

Constraints limit the interactions between components by placing requirements on the interactions. These rules can specify things like "components of one type can't talk directly with components of another particular type," or "components must only talk with other components when authorized to do so by a third component." These rules lead software architects to configure software in certain unique ways, called *styles*.

## Styles

*Software architectures follow certain styles*

The many different successful configurations of components, connectors, and constraints fall into a set of basic structural patterns, or *styles.* Some of these styles are pretty rare. On the other hand, most software systems fall into one of just a couple of styles, so the next sections discuss these common styles so that we recognize them when we see them.

### Layers: The most common style

*Most software systems use the layered style*

The *layered* style is by far the most common software architecture. It uses a hierarchical organization to separate low-level functions from high-level functions. Layers may contain many software objects, modules, or functions. Each layer in the architecture then builds on the services of the next lower layer. Connectors between the layers are quite often messages, procedure calls, or function calls, depending on the implementation language.

Pictorially, the layered architecture looks like a sandwhich. Normally the "highest level" functions appear at the top and the functions closest to the operating system appear at the bottom. The high-level functions are typically ones that the user sees or are accessible via a Graphical User Interface (GUI). **Error! Reference source not found.** gives an example of layered software architecture. Examples include the Open Systems Interconnect and the TCP/IP models used in networking [NEED CITATION].

*Diagrams of layered software looks like a sandwich*

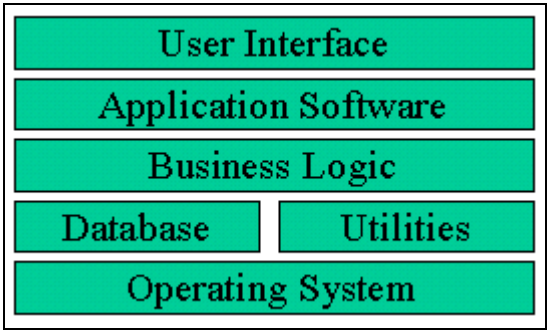



**Figure 23- Layered software style shown as a "sandwich"**

In addition to the "sandwich" view, layered architectures can also be drawn in a circular fashion, as shown in **Error! Reference source not found.**. When drawn as a "bulls-eye" the lower-level functions appear in the middle and the higher-level, user-accessible functions appear on the outside. An example of this is the IBM *Corporate Information Management (CIM)* system model developed in the 1980s [NEED CITATION].

*Layered software can also look like a bulls-eye*

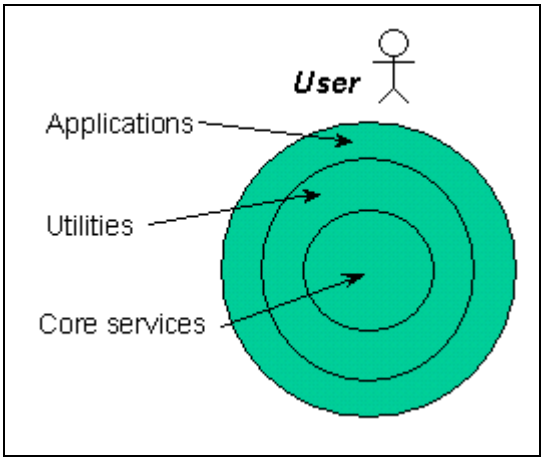



**Figure 24- Layered software style shown as a "bulls-eye"**

Layered styles have many advantages, including how it naturally allows increasing levels of abstraction at each level in the architecture. This enforces modern software principles by isolating the affect of change to a tightly controlled portion of a system. Furthermore, most software systems easily lend themselves to this kind of partitioning, which makes it easy to apply. Assuming that the system lends itself to a layered architecture (not all do), the only real technical disadvantage lies in the possibility of performance problems induced by having to many levels of abstraction.

*The layered style is popular for a reason*

The choice of diagramming technique depends on the preferences of the software architect. Whether he elects to portray the organization of his system as a sandwich or as a bulls-eye, the architecture has the same desirable properties of the layered style: an easily partitioned system with responsibilities isolated to each partition.

*Choose the diagram that best communicates the architecture*

### The Object-Oriented Style

The object-oriented (OO) style enjoyed a lot of attention throughout the 1990s as a revolutionary way of better modeling the real world for the purposes of designing a software system. An object can simply be anything "thing" in the world. The OO method maintains the proven principles of abstraction by encapsulating all the details of things in the world-

*OO intends to reflect the real world*

information about them and how they behave- inside the object.

*Objects in the world interact with other objects via messages*

Objects interact by sending messages to each other, much as people and systems do in the real world. Objects know about themselves (they exist in a certain *state*) and their surroundings. As depicted in **Error! Reference source not found.**, objects interact with other objects by sending messages to each other (the messages are the connectors). The receipt of messages *invokes* a method, or process, in the receiving object. The method may cause some action to occur and almost certainly will change the state of the object.

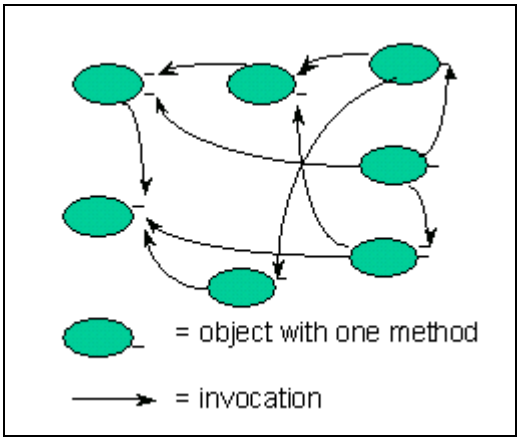



**Figure 25- Object-Oriented software style**

*OO provides a common, natural way to organize software*

The OO style appears quite commonly, not only because of its popularity but also because of how well it reflects dynamic systems. As with the layered style, the OO style provides a very effective way of partitioning a system so that changes to one part of the system only affect a tightly controlled part of the system (in this case, the object). Disadvantages include the fact that objects must know about each other and changes to an object interface affect all objects that use that interface, but procedural methods have similar, corresponding disadvantages.

### The Blackboard Style

*Blackboards are basically*

The last style that we'll discuss is the blackboard style. In this style, applications share a common data structure in memory or on disk, such as a

database. As shown in **Error! Reference source not found.**, these applications can read from and write to the shared data. The applications can operate independently; in fact, the connector in the blackboard style is the shared data, which applications use for communication and coordination.

*database systems*

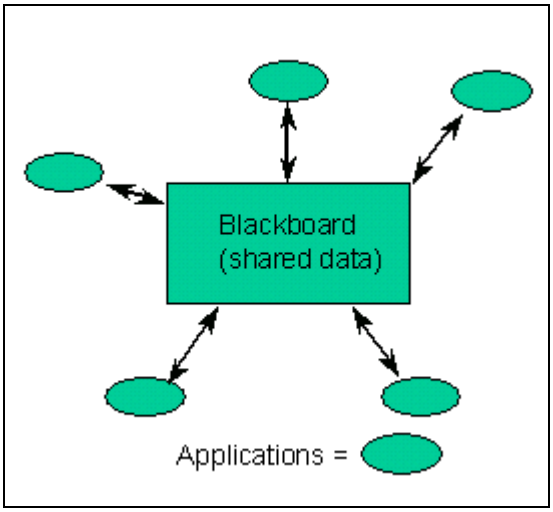



**Figure 26- Blackboard, or repository software style**

Control in the blackboard architecture depends on the state of the shared data, which make this style very good for data-centric applications, such as personnel or similar Management Information Systems (MIS). They allow minimal data redundancy, which saves space, while at the same time reducing the chances of having conflicting values for the same data. It has the disadvantages shared by any database system; it has a central point of failure and requires substantial coordination for concurrency control. However, commercial database vendors have proven solutions to these issues.

*Blackboards often appear in MIS*

## The Many *Views* of Software Architecture

Software Architects often describe their architectures using several *views*, or perspectives of the system. Just as architects of houses prepare wire diagrams for the electrician, floor plans for the framers, etc., architects of software prepare special diagrams for each software specialty. These views

*Everyone has a special interest*

represent the special interest, viewpoint, or perspective of the intended audience.

### Each specialty has its own perspective

*A module view show software components*

One of the first views that a Software Architect might prepare is the *module* view. The module view simply shows how the architect has decided to partition the software system into software subsystems. Each subsystem is a module in this view. The first iteration of preparing a module view might identify, for example, the layers in the layered architectural style.

*A logical view shows how components interact*

Next, the Software Architect might start to show how the modules interact. This *logical* view identifies the data or control flows in addition to the major software components. It portrays the high level behavior of the system and typically represents the cornerstone of the software architecture.

*A physical view fits the software to the system*

The most likely next step involves assigning the software to the physical devices upon which it will run. The Software Architect may have input into the physical devices, but normally the System Architect tells him what to expect for a physical environment. The interest of the Software Architect ends at this boundary.

*The technical view defines standards and protocols*

Several sources on software architecture (e.g., [C4ISR97, Putnam01] consider a *technical* view, which has the goal of defining the software standard that the software must adhere to. These standards might specify a specific version of a specific protocol, such as TCP/IP, or a specific version of a COTS product. This view addresses inconsistencies that can arise not only in getting a large project started but in coordinating software upgrades on multi-year projects.

*The data view shows major information*

One view that I recommend but otherwise rarely see identified is the *data* view. I recommend this view because most systems today have a critical dependency on data, how it flows, and how it is managed.

*Many views could exist to meet the needs of a specific audience*

It may take many views to adequately capture and express the many unique perspectives of a software system. Other example views include:

- *Business* view, to show processes, document flow, and how people interact or perform their tasks
- *Thread* view, to show flow of data or control flow in critical sequences of events in the system
- *Network* view, to show the wide-area or even local connectivity,

bandwidth, and other aspects of a distributed system.

It really does not matter which views are used, as long as the chosen views satisfy the needs of the Software Architect and convey the necessary information correctly and concisely to the intended audience.

## Summary: the 3Cs, Styles and Views

In summary, Software Architects start with the Three Cs of software architecture: the software *components* that make up the solution, the *connectors* that allow the components to communicate, and the *constraints* under with the components and connectors must operate.

*Start by identifying the 3Cs*

The constraints determine the *style* of the software architecture. The high level organization of software will typically fall into one of just a couple of styles, the most common of which is the layered style.

*The constraints define the style*

Because every audience has a unique interest the architect must portray his system in a way that makes sense to each of these audiences. This necessitates a series of *views* of the architecture. Each view depicts the characteristics of a specific part of the system in a way that makes the most sense to the user of that information.

*Communicate the architecture using views*

## Additional Resources

| | |
|---|---|
| ANSI | American National Standards Institute |

# Glossary

| | |
|---|---|
| ANSI | American National Standards Institute |
| API | Application Programming Interface |
| BSR | Board of Standards Review |
| CM | Configuration Management |
| CMM | Capability Maturity Model |
| COM | Common Object Model (Microsoft) |
| ConOps | Concept of Operations |
| CORBA | Common Object Reference Broker Architecture |
| COTS | Commercial Off-the-Shelf |
| CPU | Central Processing Unit |
| DCE | Distributed Computing Environment |
| GUI | Graphical User Interface |
| I&T | Integration and Test |
| ICD | Interface Control Document |
| IEEE | Institute of Electrical and Electronics Engineers |
| ITT | Invitation to Tender |
| ISO | International Standards Organization |
| JAD | Joint Application Development |
| LOC | Lines of Code |
| MIS | Management Information System |
| MNS | Mission Needs Statement |
| OO | Object Oriented |
| OSF | Open Systems Foundation |
| PDA | Personal Digital Assistant |
| RFC | Request for Comments |
| RFP | Request for Proposals |
| SAD | System Architecture Document |
| SCR | Software Change Request |
| SE | System Engineer |
| STR | Software Trouble Report |
| SW | Software |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TPM | Technical Performance Measure |

| | |
|---|---|
| TSR | Technical Solution Review |
| VPN | Virtual Private Network |

## Annotated Bibliography

[Boehm02] Boehm, Barry, "Get Ready for Agile Methods, with Care," *IEEE Computer*, Vol. 35, No. 1, January 2002, pp. 64-69.

Barry Boehm contributes periodically to *IEEE Computer* with an excellent series of concise, enlightening, and enjoyable articles on software development. Previous articles have ranged from metrics to program management. This particular article compares the advantages and disadvantages of having a strict software process (e.g., CMM) and the more agile, but *ad hoc*, "extreme programming (XP)."

[C4ISR97] C4ISR Integration Task Force, "C4ISR Integration Framework," *Version 2.0*, 18 December 1997.

C4ISR stands for Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance. This 239-page document represents a large multi-year effort by the U.S. Department of Defense to standardize on a way to document software architectures. Similar in recommendations to the other references listed below, the C4ISR document recommends three views of the software:

1. Operational view, which depicts the major components and information flows between them
2. Systems view, which shows the physical layout, to include hardware, network, etc.
3. Technical view, which lists the standards, rules, guidelines, and standards that the software must follow.

[IEEE00] Institute of Electrical and Electronics Engineers (IEEE), " IEEE Recommended Practice for Architectural Description," *IEEE Std 1471-2000*, IEEE Press, NY, 21 September 00.

Although targeted at software-intensive systems, this represents the only "official" reference for documenting a System Architecture. Furthermore, the ANSI Board of Standards Review (BSR) approved this as an American National Standard (ANSI/IEEE Std 1471-2000) in

August 2001. This reference contains excellent definitions and a discussion on describing Systems Architecture using views, where each view may have any number of models (e.g., ER diagram, data flow diagram, structure diagram). It was heavily influenced by the Zachman framework.

[Jacobson99] Jacobson, Ivar, Grady Booch, and James Rumbaugh, "The Unified Process," *IEEE Software*, Vol. 16, No. 3, May/June 1999, pp. 96-102.

This paper contains an extract from their book; a good overview of UML and the unified process. Good pictorial representation of how all the models they recommend fit together.

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

This highly cited article applies the Zachman framework to software architectures by suggesting the following four views: the logical view (object model or entity-relationship model), the process view (design concurrency and synchronization), the physical view (the mapping of software onto hardware and the distribution of processes), and the development view (static organization of the software in the software engineering environment. The paper ties these together with a fifth view to show the behavior of the system, or "scenarios."

[Putnam01] Putnam, Janis, "Architecting with RM-ODP", Prentice Hall, 2001.

Reference Model of Open Distributed Processing (RM-ODP) is an object-based architecture approach for developing distributed systems. RM-ODP provides a set of distributed processing concepts for architects of distributed systems, as well as a set of techniques to use in specifying the architecture. The RM-ODP guidelines are based on object modeling, to include a set of interrelated viewpoint specifications for describing open distributed systems. This book makes an excellent reference for on

the topic of system architecture, especially in the area of open distributed processing.

[Schuff01] Schuff, David and Robert St. Louis, "Centralization vs. Decentralization of Application Software," *Communications of the ACM (CACM)*, Vol. 44, No. 6, June 2001, pp. 88-94.

A concise overview of the cyclic nature of preferring for mainframes to PCs and back. This article contains a very good explanation of the history of physical System Architectures and how they depend on the cost trade-offs and technical capabilities of things such as communications, processing power, maintenance, etc.

[SEI94] Software Architecture Discussion Group, Software Engineering Institute (SEI), 1994.

Referenced here as the oldest working definition of a "software architecture.

[Shaw96] Shaw, Mary, and David Garlan. Software Architecture: Pespectives on an Emerging Discipline. *Prentice Hall*, Englewood Cliffs, NJ, 1996.

The most respected text on software architecture. The book reads very well and has numerous examples on the "three Cs" of software architecture and the many possible styles of software architecture.

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

The article that forms the basis for most of the work in software and systems architecture. It contains an excellent discussion of the various ways to view a system depending on the role of the observer. Through many examples, it gives a cogent argument for the need for multiple

views to fully describe a system.

# Index

This is a list of items that need to be indexed:


Context Diagram
- 

Interfaces

Organization

Requirements
- eliciting
- documenting

System Engineer

System Architect
- responsibilities
- qualities

System Architecture
- benefits
- definition
- documenting
- drivers
- tool support

Software Architecture
- definition
- styles
- views
- “3 Cs”

Shaw

Technical Performance Measures (TPMs)

Use Cases

Views
- logical view
- physical view
- operational view
- data view
- selecting

Zachman