SYSTEMS ARCHITECTING AND SOFTWARE INTEGRATION

E Rechtin

Rapporteur: Alcides Calsavara

SYSTEMS ARCHITECTING AND SOFTWARE INTEGRATION

Eberhardt Rechtin
University of Southern California
University Park
Los Angeles
CA 90089-2565
USA

Abstract

It is increasingly being recognized that unless complex, real-time systems are well-structured -- architected -- they are unlikely to be satisfactory, regardless of the subsequent excellence of systems engineering and integration. This talk addresses the art of systems architecting -- its heuristic (qualitative) foundations, its role in system software development, and its educational methodology. Among the system issues raised are the synchronization of the disparate waterfall and spiral processes, the early establishment of acceptance criteria, and the risks of reusable modules.

Part One: Foundations

Introduction

This seminar is being held during a remarkable time in the history of software. In a major paradigm shift, software for real time systems is transitioning from being a "glue" that holds together previously specified hardware to being the centerpiece of system design and operation. Increasingly, software determines what the user perceives the system to be -- its behavior and structure -- the friendlier, the better.

In a deceptively simple example of what the user perceives, the driver of an automobile believes that moving a wheel steers the car, moving a lever changes gears, pushing on one pedal accelerates and another decelerates the car and so on. Instead (Figure 1), these pseudo-mechanical devices send electrical signals to microprocessors which modify and convert them into engine management, antilock braking, grade-sensitive propulsion and speed-sensitive maneuvering and suspension -- modulated by sophisticated vehicular instrumentation and completely hidden from the driver. And this is only the beginning of a new era of automotive redesign. Electric cars, with their compelling need for computer-controlled energy management, can do no less.

A more dramatic paradigm shift is the one in personal computer design from a hardware-constrained "keep the *CPU* busy" to a software-enabled "keep the *user* busy"; i.e., from maximizing computer efficiency to increasing system effectiveness. It is an eminently practical shift; hardware costs are so low compared to software costs, that adding hardware can be more cost effective than constraining software for the same system gain.

On a global scale, it is now readily apparent that two major systems, global air transportation and global communications, have radically changed the social, political and economic conditions of the world. Two others, global positioning and cellular telephony, can only accelerate the change.. The results are more efficient financial transactions, tighter inventory control, and rapid product distribution anywhere any time around the world. All these systems are critically dependent on software to make their information-intensive operations effective and profitable. Literally, success or failure rides on that software.

As a consequence, software engineers are increasingly drawn into systems design. Central to their understanding of such systems is their architectures and how software architectures

fit into and control them. Not surprisingly, specialized, domain-specific, software architectures are now a prime topic for both software and system engineers. (Figure 2) Parenthetically, common interest in architectures may well bring these usually separate parties together. At least, Bill Gates of Microsoft seems to be betting on it.

What software architects now see as they look outward into systems is a much more chaotic, diversified and domain-specific world than the mathematically-elegant, abstract one of von Neumann and Turing. In this new world, problems are far less structured, optimization is meaningless if not impossible, analysis can produce paralysis, and sociopolitical issues really count. Why? (Figure 3) Because there are so many stakeholders, so many definitions of "success," so many variables, and so many tradeoffs -- with far too little data for meaningful analyses and crisp decisions. In the system world, the only practical hope is for reasonable satisfaction based on feasible constructs.

The first step is to bring some order to the situation -- to "structure" it, to bring practical "form" to the desired "functions," and to deliver a certifiable result to the user-client.

The process of architecting

And so, welcome to the process of architecting, conceived thousands of years ago in response to just such situations. It necessarily is an art as well as a science.

First, the science.

The science of architecting, like any science, is based on established experimental and mathematical principles and proofs. It is based on measureable "facts," defined as reproducible data. Literally, if something isn't measureable, it isn't science. Its problems must be carefully structured. Its boundary conditions must be specified with precision. It is the foundation of systems engineering. It is well taught, well documented, and highly certifiable. "Optimization" is an almost universal byword. When its conditions are met, it is extraordinarily powerful. The difficulty is that in the early stages of new system development, facts and figures are at best imprecise. Many requirements -- like customer preferences and social acceptance -- are not measureable. Something has to be done to bring more certainty and order before and during the use of scientific and rational problem solving methods.

The art of architecting, in contrast, deals with the challenges of the data-sparse, the unprecedented, the ill-structured, the unbounded and the complex. The systems of most interest have never been built before -- a first manned flight to the moon, an in situ exploration of the planets, a new style of computing, an information superhighway, a stealth aircraft. Rank-ordering requirements, crucial for design, can be an exercise in frustration. The systems themselves are unbounded; i.e., they both contain, and are part of, other systems each of which has its own "externals." Their mix of diverse elements -- technical, social, financial and political -- is too complex to treat mathematically with much if any confidence.

The foundations of systems architecting

Systems architecting is based on a small number of synergistic core concepts. Roughly in their order of appearance, they are: a systems approach, a purpose orientation, cofied experience, ultraquality, modeling, and certification. Omitting any of them sharply undercuts the total. By general consensus -- and for good reasons -- judgments of worth, detailed subsystem expertise, technology development and project management are usually placed outside its scope.[1] Each of these concepts will be discussed briefly.

A systems approach

One of the simplest and most useful definitions of a system is that it is a collection of different elements which together produce results unachievable by the elements alone.[2] The key words here are collection, different, together and unachievable. The results, system-level functions, are achievable only by the system as a whole; for example transportation is the system-level function of an automobile and life is that of the human body.

The primary purpose of systems is to produce these functions; that is, to add value to that of the elements, themselves, by making connections between them. Hence, the principal focus of the systems approach, and the principal design leverage of its practitioners, is at the interfaces. One conservative strategy of architecting, posited in 1964 by a classical architect, is that the best architectures are those with the fewest interface mis-fits.[3]

But herein lies a dilemma. Generally speaking, the more system functions that are desired, the more elements (and still more interrelationships) are needed. But the more there are, the greater the risk of instability[4] if not chaos. This phenomenon is all too often encountered in evolving computer communication networks, electric power grids, nonlinear feedback control systems -- and software. Even though the individual elements or modules may be stable, the system may not. Once again, the system behaves differently than the sum of the individual behaviors of the elements.

For these reasons, systems to be successful must be viewed as a whole, the specialty of system architects, analysts and engineers. As to be expected, these practitioners are experts in system functions and interfaces and how they fit together. In any case, for them to try to be expert in all the facets of all the elements is impractical and counterproductive.

But architects would be remiss in their responsibilities for the system as a whole if they knew little or nothing of the "insides" of the elements. On that count, they are justifiably wary of software hidden routines, commercial off the shelf (COTS) items and reused software. Obviously,the depth of understanding must be selective. An appropriate criterion is how critical the individual subsystem, discipline, or detail is to the system as a whole. (Figure 4)

Predictably, as systems become smarter and smarter, systems architects and engineers will have be expert in critical software techniques, languages, routines and architectures. If so, we shouldn't be surprised if before long the best systems architect-engineers come from the software field instead of electronics and control systems which have supplied them to date.

Purpose orientation

Next in importance as a core concept is that architecting is inherently purpose-driven. That is, the client's purposes come first. A first objective of client and architect is to discover and then maintain those purposes throughout system conception and build. They will, or should, drive system design, analysis and implementation. Experience demonstrates that the client's initial statement of purpose -- that is, the needed system functions -- may well be internally contradictory, mixed up with preconceived "solutions," based on erroneous assumptions, and poorly derived from more fundamental purposes. The client seldom can rank-order requirements, even if each is eminently reasonable, without some idea of what the system might be like. There is nothing more dismaying than excellent engineering, construction, and management that has been wasted on a system without a valid purpose. Whatever time and however many iterations it takes to establish the purpose, it is worth it. As Bob Spinrad of Xerox put it in 1988 in the context of software, *"All the serious*

mistakes are made in the first day.[2] Worse yet, you may have to live with them for decades."

The first step in architecting is therefore establishing, jointly with the client, a good match of acceptable system functions with a feasible system form (a systems architecture). As a classical architect might put it, architecting is making "form fit function." The process is unavoidably top-down and, of course, purpose-driven. Its tools will be discussed shortly. But first, to better understand systems architecting, a contrasting way of building complex systems -- client-less.

From the foregoing rationale, for architecting to work properly, there must be a real client/sponsor willing and able to make value judgments of what is acceptable and satisfactory, to decide what is "good" or "bad." (The client, more than anyone else, knows that there can never be an "optimum.") But very successful complex systems have been and are built without such a client.

Consumer products (PCs, TVs, automobiles), are built with, at best, a surrogate client (marketing and sales). During the Cold War, some defense systems were built "bottom up" driven by the assumed combat value of new technologies (hypersonic flight, stealth, fly-by-wire) against high-tech opposition. The closest thing to a client was an uneasy combination of the Secretary of Defense, the Unified and Specified Commands, The President and the Congress. By default, the technologists made most of the decisions. For lack of a better term, this client-less approach might be called "technical innovation," because technology is the driver.

In some sense, the two approaches are time reversed. (Figure 5) One combined approach is to use them in series, architecting first to establish guidelines followed by incremental, technology-driven upgrades. MS-DOS, copying machines, television sets and launch vehicles are typical examples.

But, when all is said and done, unless a system has a valid, valued purpose, it will not last. Architecting begins with that truth by putting purpose first.

Codified experience

Even more than engineering, architecting is based on practical experience. Large scale, complex systems pose special problems. Space systems, communication networks, weapons, computers and other complex systems take years, even decades, to design, build, deploy and operate. Product-line architectures may last just as long, even if their technologies change. There are very few systems architects who have had an opportunity to work on more than one or two major programs. The lessons learned in the first program may be unique to the first program or now obsolete, technically, socially or politically.

The problem is acute today because a whole generation of architect-engineers with 25 to 50 years experience in major complex systems are leaving the work force. At the same time, a wide array of new systems needs to be architected -- and by software-literate people young enough to commit 20-25 years to their accomplishment.

It is not a new problem. It has been around as long as the pyramids and been tackled by builders of cathedrals, warships, electric power distribution and telephone systems. Their solution, still valid, is to codify, or formalize, practical experience so that it can be transferred between programs and generations of architects. Most of the formalisms are quantative; that is, expressible in numbers. In the building trades this codification results in building codes. In defense system acquisition it generates military specifications, a

primary mechanism for retaining corporate memory. In engineering, it results in algorithms, tradeoff charts, equations and handbooks.

Less well known but possibly broader in its application across the disciplines of engineering, business and social disciplines, are insightful "lessons learned," or heuristics. Because they are less well known, this paper will concentrate on them. The ones shown here are in italics.

Heuristics are an attempt to codify experience by condensing, in a few words, enough common experience to offer a useful guidelines for system building. The word "heuristic" is traceable to a Greek word meaning to find or to guide. "Heuristics" is also used in computer science to describe certain problem solving techniques.[5] It is one of four architecting techniques -- the normative, rational, participative and heuristic -- and most distinguishes it from systems engineering. [2,6,7] It is the most inductive, a-rational and experiental of the set.

A few examples: (Figure 6)

* Two descriptive heuristics, *Murphy's Law: If it can fail it will* and a conflicting one, *If it ain't broke, don't fix it* provide a telling contrast between the automotive industry strategies of Japan and other countries in the 1980's. The first heuristic, calls for zero defects, the second reinforces the status quo on any organization wary of change.
* *Simplify, simplify, simplify* is one of the best prescriptive heuristics for dealing with Murphy's Law.
* *All the serious mistakes are made in the first day*, mentioned earlier, also has associated prescriptive heuristics. Two of them are: *Don't assume the original statement of the problem is the right one* and *Build in and maintain options as long as possible. You will need them.*
* Some heuristics provide still more direct guidance, such as *In partitioning, choose the elements so that they are as independent as possible, that is, elements with low external complexity and high internal complexity* also expressible as *In partitioning a system into subsystems, choose a configuration with minimal, but effective, communications between the subsystems.*

The first of these heuristics came from the aerospace industry, the second from the automotive industry. *Simplify* came from commercial aircraft, *Serious mistakes* from copier products, and *Partitioning* from aerospace and its software sector. Yet they are evidently broadly applicable as shown by their intuitive acceptance by practitioners in other fields.[8] Their acceptance almost invariably is pragmatic - they seem to work. No mathematical or statistical proof seems necessary and in any case, none exists. Applicability is by example, case by case, context by context. As such, heuristics, like any technique, have their characteristic limits.[9]

About six years ago we at the University of Southern California became seriously interested in heuristics as tools for systems architects (and others). About 100 appeared in a textbook[2] followed by hundreds more in subsequent student research reports[8] on systems they worked on in industry. The sheer number was a surprise, certainly, but it has posed a serious problem for further research -- how best to access and use them. An early conclusion of such research is that a remarkable number of heuristics, discovered in one field, apply in many others with only a change in wording (for example, into business administration). The same heuristic may apply throughout a product life cycle. Because they are experience-based, they tend to be more useful at avoiding disaster than at generating marvels; but, used as check lists, some of them evidently can inspire or refine

new ideas and products.

Clearly, systems architects do not live on deductive analytics alone. They also need inductive guidelines and lessons learned by their predecessors to structure their ill-structured environment, purposes and products.

Ultraquality, a quality too high to be measured

Quality is a measure of excellence. Ultra means beyond. Ultraquality is excellence beyond the ability to measure it. In this instance it means a failure rate so low it is impractical to measure with any degree of statistical confidence. The ultraquality challenge reaches its peak at certification when the architect must assure a client that the level of reliability is high enough that a first-of-a-kind, unprecedented system can be accepted and used -- even though the hardware and software are untestable to that level!

Ultraquality is a non-negotiable requirement for many complex systems -- safe nuclear power plants, manned space flight, single flights to the outer planets, multi-hundred passenger airliners, carcinogen-free food products, secure transmissions systems, and electronic financial transactions.

The challenge is greatly increased by the nature of human intolerance of failure. Demonstrably, the level of tolerance is progressively lower the greater the size and complexity of the system. Home appliances should not fail more than 1% of the time. A still smaller failure rate is demanded for an automobile, even less for an airliner, and still less of manned space flight. Thus as complexity and piece part count increases, the tolerated failure rate per part drops to a point where it can't be certified without unobtainable numbers of parts over unavailable time. Only microprocessor chips have demonstrated a reliability per operation that decreases with the number of operations performed.

It is no wonder that unthinking integration of available parts or modules into larger systems can be a prescription for trouble.

Techniques have been developed to address this challenge. Available modules must be recertified for the larger systems. Zero defects, simplification, carefully-introduced redundancy (or its complex management can cause failure), rigorous elimination of single point failures, and inspiring the work force are a few of the measures required.

Systems architects really have no choice; they must be obsessive about quality, especially that of their own design. Least forgiveable are the "planned accidents" which cause users, from nuclear plant operators to airline pilots, to make fatal mistakes. Quality is not something to be left to the builder, the parts supplier, the quality assurance staff or the test crew, vital as their contributions are to the whole. It must be designed in and maintained. It is the architect's responsibility to see that it is.

Modeling and design management

The concept of modeling is as old as architecture itself and as modern as virtual reality. Its techniques need no further elaboration here. More important are its nature and purposes as seen in systems architecting.

Modeling serves systems architecting as a communication medium among participants, particularly between the the client and architect, but also among engineers, builders, users and the public. It and its close cousin, simulation, help manage the design by maintaining continuity of purpose and intent throughout the build cycle.

Like architecting, modeling is an art as well as a science. Like art, it is an abstraction of reality. Like art, it deals with perceptions, representations and visualizations. It adds new perspectives to uncertainty and tensions. It gives form to intuitive judgments about what is important or not, what to include in the model or not, and what is "good enough" for the purpose intended. The resultant system-level models become the principal deliverables of the architect; from them are derived engineering and manufacturing specifications and acceptance criteria.

But it must be understood that *models are not reality,* only abstractions of it. Because models cannot be complete without being identical to the system, itself, initial assumptions and exclusion decisions can be critical. *The most dangerous assumptions are the unstated ones*. Experienced architects, in any case, quickly learn that reality has ways of adding unexpected relationships, issues, constraints and phenomena to any model. Reality can demolish what look to architects-engineers like logical assumptions, especially if they involve human behavior or social acceptance. On the technical side, the difference between a system's model and its reality can become a major issue in the event of system failure. There is nothing more unnerving to a review board, confronted with seemingly clear indications of a specific system failure, than a firm comment from a model-oriented architect "It can't fail that way!" Which is right, the model or the data? It may be a surprise to some, but the model may be right and the cause of the failure must be sought elsewhere. A notable example was the explosion of a Titan space launch vehicle, erroneously assumed to be a failure of a joint between two solid rocket segments because a "similar" explosion had occurred in a Shuttle launch. (The real cause turned out to be debonding of the propellant from its case.)

It is often presumed that modeling ends with the completion of conceptual design. After all, it is often the case, reinforced by competitive procurement regulations, that the original architect leaves, or is assigned to another design effort, and the engineers and builders take over. But modeling continues, nonetheless. During every phase -- engineering, development, manufacturing, testing, operations and final assessment, diverse models are built and used by specialists in their work. Unwatched or unmanaged, these specialized models can negatively impact on the system as a whole, letting it drift from its original purposes in the interest of sub-optimization, often complicating an already complex design. Or, truly beneficial possibilities identified through the model can be discarded locally that might have been welcomed at a broader systems level.

The worst situation is having no model at all -- no guide, no cross-check, no predictor of performance and no diagnostic tool for system failures. Then, when trouble strikes, appropriate response is almost impossible. One notorious real-life example is the recent lack of a flight simulator during anomalous spacecraft operations. Another was the lack of a model to show the effects of scale on system design. Not all sytems can evolve to larger purposes or be used efficiently for smaller ones. Examples are single-stage-to-orbit launch vehicles and closed-architecture personal computers.

Certification and the architect's role in it

The realities at the end of the conceptual phase
are not the models, but the acceptance criteria. [2]

Certification is the architect's assurance to the client that the system as built is acceptable and ready for use. Architectural certification, in a sense, is the grade given to the builder after the final exam of systems qualification and acceptance testing. Based in large part on it, the client pays the builder and takes responsibility for its operation.

The architect, as the agent of, and co-conceptualist with, the client, should be in a unique position to certify the system. More than anyone else, in as unbiased a way as is possible for a participant, the architect knows the client's purposes, the systems functions, the design trades, the builder's methods and the decisions made along the way. Achieving this position of trust and respect by all the other stakeholders is not easy. It requires openness, tact, diplomacy, technical credentials, system-level objectivity and an acceptance by others of the architect's assigned role.

Clearly,the design of the "final exam" -- that is, the acceptance criteria --is crucial. They must be both complete and passable. The builder will build to pass that exam professionally but not in excess. The owner will be focused on system performance, which is not always the same thing as test results. There are bound to be tensions and disagreements, but they can be greatly reduced by agreement ahead of time on the acceptance criteria. Those criteria, though they may be modified by circumstances and agreement during the build, must be in place at the latest by the time of signing of the contract with the builder.

In short, certification does not and can not begin after system test. It must begin as part of system conceptualization and culminate with system acceptance by the client.

System certification, like other individual parts of architecting, can be undertaken in other ways. One alternate is certification by a third party that performs testing independent of the builder's testing -- an equivalent of a Good Housekeeping Seal of Approval for household goods, a Bureau of Standards test of conformance with standards, a Federal Aviation Administration safety certification, or an Operational Test and Evaluation report to the Department of Defense.

One notable difficulty has arisen in third party certifications. It can be traced back to the need for the certification process to begin early. The certifier must agree that the acceptance criteria are complete and passable, which often means engaging the certifier in judgments of system purpose. The certifier also will want either to observe (and comment upon) all the builder's tests or to perform parallel tests using separate test facilities The chances for basic disagreements on the design, build and test approach between certifier and the system team are always present, which can be disconcerting and costly to the client.

There can be serious consequences if the certification process is interrupted. Architects are often dismissed once a builder's contract is signed. The project is then turned over to engineers and managers who rethink and modify the concept to ease their own tasks or to rectify perceived defects. Continuity of purpose is easily broken, acceptance becomes a demonstration of "built to print" and conceptual integrity is reduced. The system will have been de facto re-architected without an architect.

Self-imposed limitations in scope

Performing the role of the architect as agent of the client, as amonitor of the build and as a certifier of the results has, over the years, resulted in the self-imposition of limits on what the architect should do.[2,10] Generally speaking, the limitations are aimed at reducing perceptions of conflict of interest, maintaining objectivity, respecting client prerogatives, focusing on the central purposes, and avoiding being overwhelmed by detail. The field may well be unique in not claiming universal applicability and primacy in all phases of engineering and management. In any case, the limitations are pragmatic and practical.

The limitations are: (Figure 7)

* Clients, not architects make value judgments (worth, acceptability, etc.). Architects,

not clients (preferably) make technical decisions (feasibility, stability, etc.). These limitations insure proper division of responsibility between client and architect. An heuristic is, *Don't try to tell the client what his life style should be. Respond with another question if the client asks, "What would you do in my case."*

* An architect should not attempt to be an expert on everything, only on those things which are critical to the system as a whole. *Leave the specialties to the specialist* (but gain and retain his confidence). This limitation is a practical one, overload on the architect, as well as an interpersonal one. The specialist must feel needed and trusted.

* Purpose, not technology, comes first. Otherwise, the result will be a solution looking for a problem. The greatest single risk in schedule and cost overruns, according to an Aerospace Corporation study (circa 1991), is immature technology. The client wants systems functions, not risk-prone technological showpieces. *Simplify, simplify, simplify* can serve as a criterion for incorporating a new technology; i.e., if incorporating it simplifies the system and its acquisition as a whole, OK. If it complicates the system as a whole to little gain, forget it. Most new technology comes with unexpected encumbrances (supporting developments, new constaints, effects on other parts of the system, etc.). In any case, be sure that the client knows the risks involved and is kept well informed of them.

* Unless specifically contracted for, basic architecting does not include project managment (contracting, resource management, progress reports, scheduling, cost control, audits, etc.).[10] This limitation is intended to reduce perceptions of conflict of interest, both technical and financial. An architect should not be in the position of deciding on, implementing, or profiting from a recommendation that the architect has made, particularly if it requires the expenditure of significant resources to achieve. As with any specialists, *Leave management to the managers*. There is also a practical reason for this limitation. Project management generally involves considerable staff. Architecting is best done in small teams (6-12 individuals). The architect can quickly spend more time administratively than in basic systems architecting.

Conclusion to Part One, Foundations

Systems architecting is an art and science founded on a handful of core concepts and self-imposed limitations in scope. Its purpose is to bring structure to ill-structured purposes, to match desired functions with feasible forms at least to the point where the powerful analytic techniques of engineering can be employed. It begins with a joint architect-client conceptual design and largely concludes with the architect's certification to the client of conformity with agreed criteria for client acceptance.

References for Part One

1 Rechtin, E., "The Foundation of Systems Architecting" for publication in the NCOSE/IEEE Journal Fall 1994

2 Rechtin, E., *Systems Architecting, Creating & Building Complex Systems*. Prentice-Hall, New York 1961

3 Alexander, Christopher. *Notes on the Synthesis of Form* Harvard University Press 1964

4 Klir, G.J. *Architecture of Problem Solving* Plenum, 21-22, 1985

5 Pearl, Judea *Heuristics* Addison-Wesley 1984

6 Lang, Jon *Creating Architectural Theory* Van Nostrand Reinhold, 1987

7 Rowe, P. G. *Design Theory* The MIT Press Cambridge, MA 1987

8 Rechtin, E. *A Collection of the Best Student Papers on Systems Architecting ,1988-1993* University of Southern California 1994 (unpublished but selectively available

for research) 150+ reports on complex systems written by industry-experienced, USC graduate students.

9 Rowe, Alan J. *The Meta Logic of Cognitively Based Heuristics*. University of Southern California, School of Business Administration, Los Angeles, CA 1988

10 Canty, Donald "What Architects Do and How to Pay Them" *Architectural Forum* **119**, 92-95 September 1963

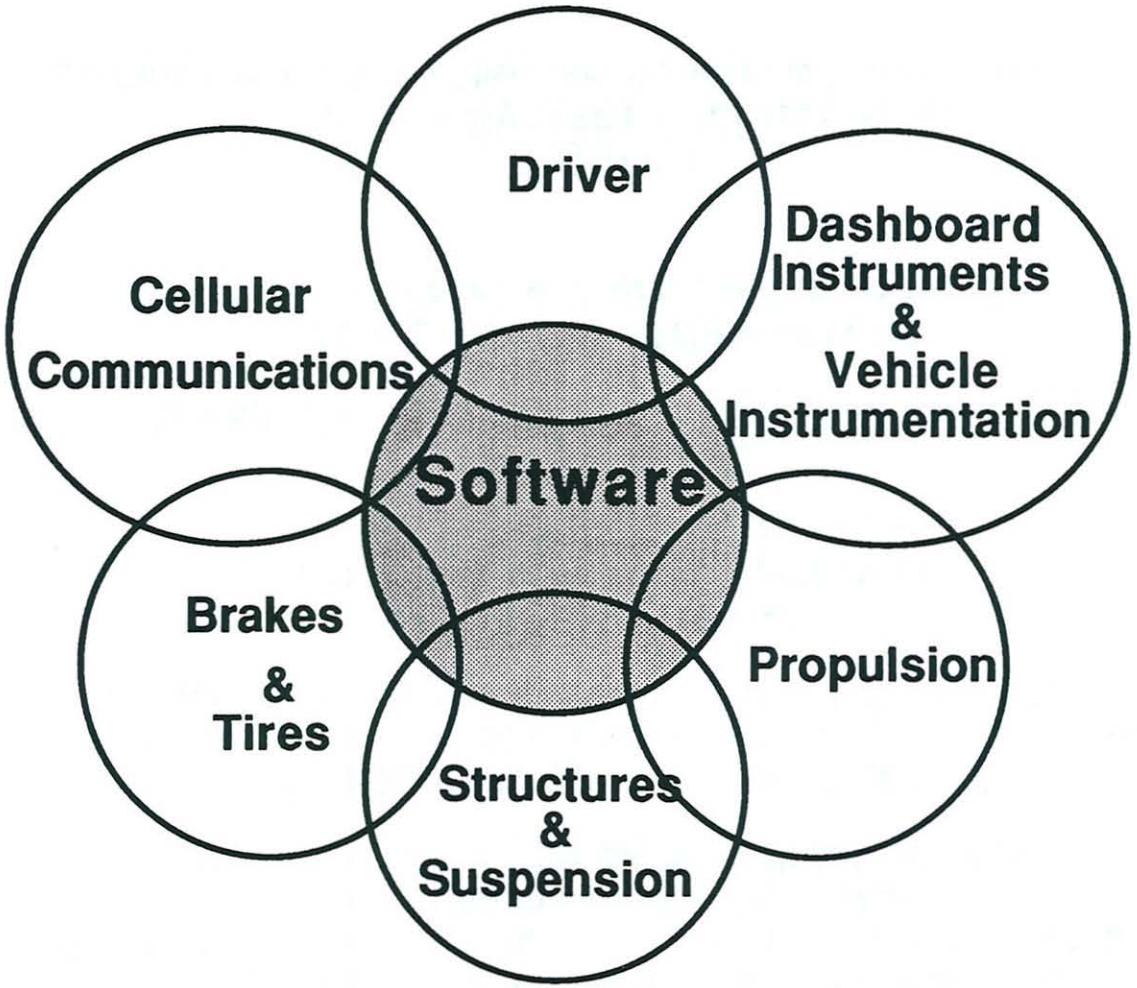


Figure 1
Wenn Diagram of an Automobile

U.N.-ER-1

PIPES AND FILTERS
Each component has a set of inputs and outputs
Output begins before input is consumed
Conceptually and operationally simple

DATA ABSTRACTION AND OBJECT-ORIENTED
Components invoke functions and procedures from each other.
Implementations hidden from clients
A natural match with problem decomposition

EVENT-BASED, IMPLICIT INVOCATION
Components broadcast events to which others respond.
Components register events of interest to themselves

LAYERED SYSTEMS
Hierarchical support to layers above
Example: Communications protocols

REPOSITORIES
Central data structure and independent operators

TABLE DRIVEN INTERPRETERS
A virtual machine is produced in software

Figure 2
Software System Architectures*

* From David Garlan and Mary Shaw, "An Introduction to Software Architecture" 1993

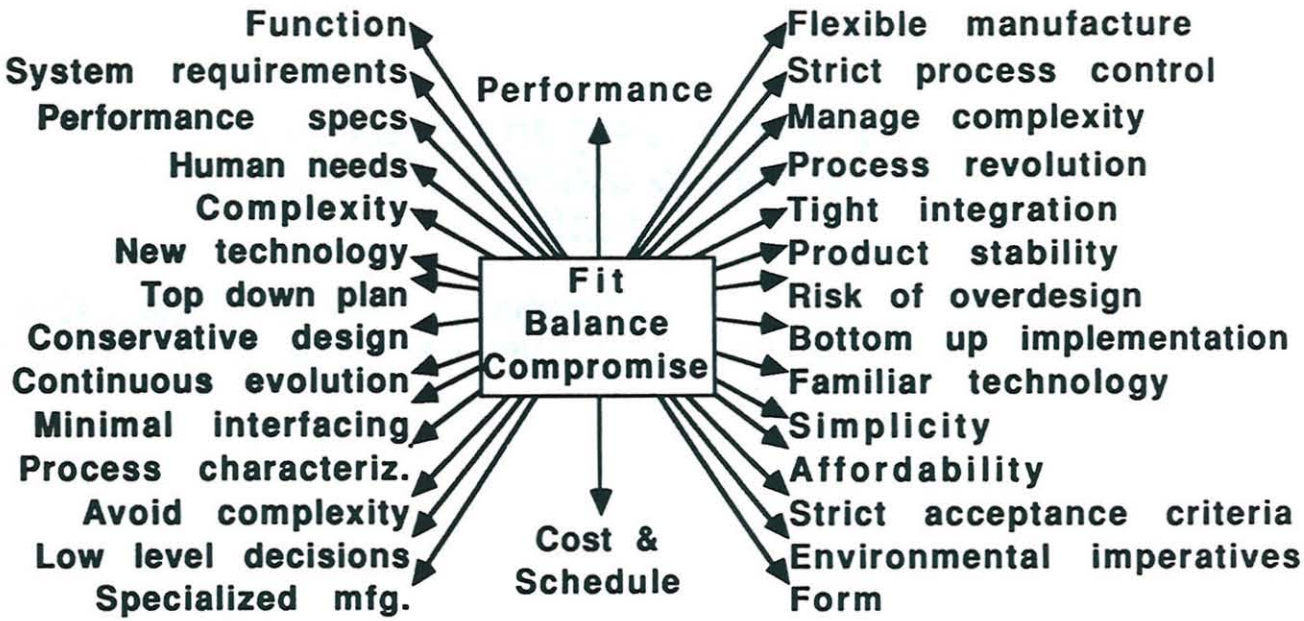
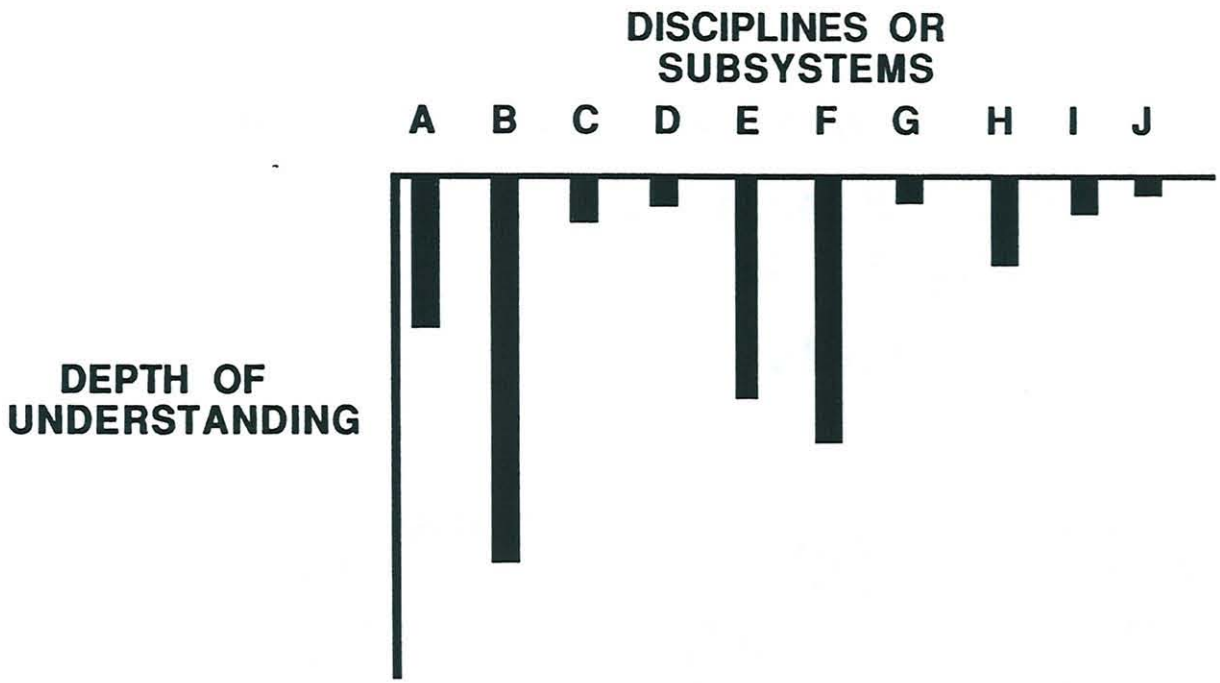


Figure 3

Tensions in Architecting
"A Compromise of the Extremes"

U.N.-ER-3



Criterion: The depth depends upon the criticality to the system as a whole and changes with time and circumstance.

Figure 4
The Architect's Depth of Understanding of Subsystem or Disciplinary Details

U.N.-ER-4

Comparison	Systems Architecting	Technical Innovation
Starting point	Purpose	Product
End point	Product	Purpose
Driver	Demand pull	Technology push
Typical customer	Single entity One at a time	Mass consumers All at once
Creativity	Top down	Bottom up
Technology	Whatever works	Source of product
Cost	Determined up front	Secondary
Profit	Importance depends on client purpose	Determined last by the market
Greatest risks	Immature technology Flawed purposes	No valid purpose Competition better

Figure 5
Architecting and Technical Innovation

Murphy's Law: If it can fail it will.
Hence:
Simplify, simplify, simplify.

All the serious mistakes are made in the first day.
Hence:
Don't assume the original statement of the problem is the right one.
Build in and maintain options as long as possible. You will need them.

In partitioning, choose the elements so that they are as independent as possible, that is, elements with low external complexity and high internal complexity.
Or:
In partitioning a system into subsystems, choose a configuration with minimal, but effective, communications between the subsystems.

Figure 6
Examples of Heuristics

Clients, and not architects, make value judgments.
Architects, and (preferably) not clients, make technical decisions.

An architect should not attempt to be an expert on everything.
Leave the specialties to the specialist.

Architecting does not include technology development.
Purpose, not technology, comes first.

Basic architecting does not include project managment.
Leave resource management to the managers.

Figure 7
Limitations on Systems Architecting

SYSTEMS ARCHITECTING AND SOFTWARE INTEGRATION

Eberhardt Rechtin
University of Southern California
University Park
Los Angeles, CA 90089-2565
USA

Part Two: Integrating Software and Systems

Introduction

Part One began with an almost-fulfilled prediction that software would become the centerpiece of unprecedented, real-time systems. The reasons, technical and economic, applied from the scale and complexity of an automobile to that of global communications and transportation. The examples could just as well have been from microprocessors to information warfare or from coding to Ada.

The value, feasibility -- and problems -- of such systems are now self-evident. Less evident has been a recognition of an explicit need for an architecting process and experienced architects to carry it out. That situation is now changing.

In the United States, an appreciation of sound architectures and skilled architects is rapidly taking hold. Architects, architectures and architecting are terms increasingly seen in the technical literature, particularly in the information-intensive fields. High level advisory bodies to the Department of Defense are calling for architects of ballistic missile defense, C4I (command, control, communications, computers and intelligence), global surveillance, defense communications, internetted weapon systems and other "systems of systems." Formal Military Standards are in process defining the role, milestones and deliverables of system architecting. Many of the ideas and terms of those standards come straight out of the software community; e.g., object-oriented, spiral process model, and rapid prototyping. It should not be a surprise; most of the applicable systems are information-intensive. A peak in information intensity is likely reached in systems for fighting "the information war." In that war, victory is achieved by having control of one's own information while denying similar control to the opponent. The recent Gulf War with Iraq was a demonstration of the importance, if not the necessity, of winning such information wars. In my mind, it was comparably important to winning the air and sea wars. (Only the army wins the end game.)

Similar, if less dramatic, examples are the Information Superhighway, Internet, global cellular telephony , Health Care, manned space flight, and expeditionary forces into war-stricken countries. All need order out of chaos. All need structuring. All need the integration of the software and systems worlds.

Differences between the worlds of systems and software

Unfortunately, structuring and integrating the systems and software worlds will not be simple. The purpose-oriented systems world and the information-oriented software world have long been different. Systems architect-engineering and software programming, driven by different needs, are so dissimilar that the two communities easily misunderstand each other. (Figure 8)

Systems, for scheduling reasons, are built by a waterfall process. (Figure 9) Each phase has completion dates; by implication, the intent is to be error free at each milestone. Software, on the other hand, is built by a spiral process resembling the conceptual phase in

system building, in this case, designing a ship.[11] (Figure 10) In this particular spiral, design begins with a provisional selection of key parameters, progressively selecting others, cycling back to the originals to modify them, and spiraling to convergence of all characteristics in a completed, balanced design --and a beginning of a build phase. In contrast, software is virtually complete at the end of the conceptual phase; post IV&V phases largely serve to reduce the errors. Even if each were isolated developments which only had to come together at an overall system certification, integrating the two processes would still be difficult. In practice, they must interact frequently as subsystems are build and tested. The image is one of a software spiral ball rolling down the hardware waterfall with the software perceived as never "completed" on schedule.

To use a spiral model for hardware acquisition would be equivalent to repeated prototyping, cycling back repeatedly from system test to conceptual design. Such prototyping, a staple for software, is inherently too expensive for systems whose system functions can not be fully realized until all elements are present, accounted for, and connected -- a process which can take years for vehicle and communication systems -- only to have to go back and go down the waterfall again. NASA and the U.S. DoD have tried this approach with all the best intentions. They wound up with expensive and incomplete "fly-before-buy" machines and that contradiction in terms, "operational prototypes" which were neither operational nor prototypes.

Still more serious are the major differences in reliability.[12] Software does not wear out. Virtually all its faults are rooted in design; i.e., the design is never finished. Redundancy, a staple in hardware systems, may not work well in software systems. Indeed, it must be used with care even in hardware systems. Hard-bought experience shows that redundancy management can become so complex it can be a source of failure, itself. In most systems, failures are tracked back through hardware first, a questionable strategy for software intensive systems. But, disconcerting to systems managers, software diagnostic tests may be more difficult to design than (and take as long as) the original software. In point of fact, the certification test of choice is another software system that is intended to accomplishe the same functions -- independent verification and validation (IV&V). In a sense, most software is still being checked out ("debugged") in use, a process which is almost unimaginable for ultraquality systems and one-of-a-kind missions.

Architects of purpose-driven systems, designing for the best interests of the system as a whole, are understandably resistant to the constraints and unknown risks of mandated subsystems not produced to the same standards; e.g., commercial off-the-shelf (COTS) components, standard languages insensitive to domain-specific concerns, and components "qualified by flight," a euphemism for "it hasn't failed yet." The history of standard satellites in the space program is, with an exception or two, an unpleasant and expensive one.

Architects of information-driven software systems, on the other hand, generally create efficient software for moderate-risk applications. Weary of starting over each time, they see reuseable modules and reference architectures as eminently practical ways of shortening time-to-market and reducing costs. In order to remain competitive, they must depend on incorporating, even anticipating, new technology -- a high risk strategy for hardware acquisition and strongly inhibited by procurement regulations.

These differences are more than cultural mismatches; they are structural chasms.

Integrating systems and software

A number of forces are at work to bring systems and software together. The most

powerful motivation is likely to be the utility and profitability of smart systems; that is, demand from both customer and supplier.

Systems architects will have to change their present perception of software as "glue" to "software at the center." Software architectures will have to match overall system architectures if they are to contribute to system-unique functions. But, being at the center, software architecture will have to interface with all the other subsystem architectures, many of which will present structurally different, domain-specific software at their boundaries.

Software architectures must therefore have to become systems themselves. System properties like unboundness, interrelationships, instabilities, and external waterfall developments will have to be incorporated in them.

And by no means least, software architecting will have to be based on the foundations of systems architecting -- a systems approach, a purpose orientation, codified experience, ultraquality, modeling and certification.

Fortunately for the systems themselves, software engineers and architects are already at the forefront of the needed integration. Software architectures are discussed in terms a classical architect could understand. Software architects are recognized as such by their peers; few systems architects are. Software architects are much more intellectually engaged in systems thinking than systems architects are in software thinking. Software practitioners treat certification more seriously than their systems counterparts, perhaps because IV&V, being part perceptual and non-analytic, is so difficult -- and it comes so soon after design.

And, especially relevant to today's seminar, software architects seem to appreciate that their work is an art at least as much as a science, and that heuristics are a part of it.

But systems architects are not too far behind. And that suggests several tools that may speed up the integration process.

* Because both systems and software require structures to be effective, the structuring process of systems architecting should provide a bridge between them, or at least many lines of communication.
* Because similar if not identical heuristics appear to be applicable to both; e.g., *Simplify* and *Partition;* their general judgments and design principles should be consistent.
* Systems ideas and object-oriented ideas are structurally similar. They would seem to create a natural fit in system representations. Both fields have a scalability problem, which suggests some sort of mutually consistent hierarchical partitioning.
* Both have problems of instability due to complex interrelationships, recognizeable as feedback loops or "go to" instructions.
* In today's chaotic sociopolitical world it is becoming essential to have architectures which can respond to rapid change. Communications professionals call for reconfiguring. Software professionals call for open architectures. But both understand that the enabling technology will be software, architected to be open, reconfigurable, modular and certified.

Two more disciplines coming on the scene

Difficult as it may be to integrate the systems and software fields, the integration process will soon have to add two more fields -- human factors and political realities.

Human factors will have to be added because systems are becoming smarter; that is, they are more intimately involved with how humans <u>think</u>. "User friendly" is not an equipment specification, it is a description of an interaction between a machine and a human being.

While by any reasonable definition of "think," both can think, they think differently. Each is better in certain modes. Each use different tools. Their architectures are clearly different. Some of you may balk at these statements but they are already seen as design factors not to be ignored.

For example, the design of CAD and CAM (aids to design and manufacturing, respectively) make the following assumptions, among others, which are flatly contradicted by well-documented human factors research[13]:

* There is a single representation of key design features that is understandable to the entire design community. (No. Each represents the needs of only selected members.)
* A designer needs and wants access to all information relevant to a design. (No. The actual search is for a minimum of information. It is more important to determine what is needed and when it is to be used.)

Omitting human intelligence factors has predictable results. One such may be the failure of CAD and CAM to meet their expectations individually or together.

Another example is the video cassette recorder. Its design is so far removed from human behavior that it has become a metaphor for incomprehensibility even appearing in Presidential speeches!

On a more positive note, the problems of making a cockpit match the thinking of a Mach 3 fighter pilot in combat have without question placed human factors at the top of the designer's list.

Matching the minds of human and machine is an unprecedented problem for architects. Matching the sinews and sensors is but a poor second in difficulty. But remarkable results have occurred in places like the Xerox Palo Alto Research Center. A DoD Advanced Projects Agency-funded effort, its most famous commercial offshoot is the Macintosh personal computer. The "Mac" is more than 10x easier to learn and far easier to recall from human memory than the comparable IBM PC.

Another closely related field of human factors research is the introduction of automation into the workplace. Attempts to integrate automation in manufacturing plants are demonstrably unsuccessful 50% to 75% of the time. Of the failures, the great majority are traceable to human resource and management practices that focus on what is to be done but not how; i.e., a failure to understand the impact on the human infrastructure.[14] As a consequence, warehouses are filled with discarded robotics literally worth billions. A sad commentary on the systems architecting design team, if any.

The second discipline that will need to be integrated is the political process, particularly as it affects system design. Engineers as a group think of the political process as an unavoidable evil working against all that is logical and reasonable. That perception is worse than wrong, it leads to mistakes, delays and costs in system acquisition.

The political process is as much of a system as is the human operator, the physical machinery, or the software. Like each of those, it behaves "strangely" if compared directly with other subsystems. And it has a direct effect on system design, sometimes detrimental, often constructive.

Ask the European partners in the new international space station. Ask the UK designers of the once-joint US-UK Ptarmigan military communication system. Or contrast the Apollo mission to the moon with the Shuttle as a cargo transporter. Political factors were not only

important, they were primary in the design.

Political factors become primary because political figures, representing the public interest, are asked to approve projects so costly that something else, comparable in cost, must be given up to pay for them. That economic fact calls for judgments of relative worth. And as seen earlier, worth is the "other half" of systems architecting, the half that properly belongs to the client, sponsor, or legislative authority. It should be expected that the political process will affect design.

It may be a surprise, but the political process has its rules, its imperatives and interrelationships, its professionals and amateurs, and its heuristics; e.g., *Cost counts* and *An organized constituency is essential).*[15] Its relationships to complex systems are not well understood in any direct mapping sense, but one thing is certain. It is better to understand them than to hold them in contempt. And it is still better to build them into the system architecture. Communication engineers are especially aware of the political process. A process that assumes that national boundaries are communication boundaries, it controls frequency allocation to the point of engineering irrationality.

Conclusion to Part Two

The coming era of smart complex systems will require the structuring (architecting) and integration of the systems and software worlds, worlds which are structurally dissimilar. System and software architectures, common heuristics and a common purpose in system effectiveness and profitability are forces which will help bring them together. Yet there are two more disciplines to be added, human factors and the political process, about which systems architects as yet understand much less.

Appendix

The Master of Science Program in Systems Architecture and Engineering (MS SAE) at the University of Southern California (USC)

Background

It became evident in the early 1980's that if major systems were well begun, they were highly likely to succeed. If not, they were rarely even satisfactory -- regardless of subsequent excellence of systems engineering and integration The clearest evidence came from programs in defense, space, computers, and transportation. The conclusion was refined to a more concise statement: unless complex systems were well *structured* or *architected,*satisfaction was unlikely.

The purpose of the USC MS SAE program, then, was to introduce systems architecting to graduate engineers in the aerospace, computer, electronics, industrial and software fields. The program began in September 1988 with an experimental, two-semester course. The challenge was whether the subject could be taught, if so, how, to whom, and to what useful purpose. The value would be judged by student enrollment and the interest shown be major Southern California firms in aerospace and electronics industries.

A graduate-level textbook was drafted based on post-WW II literature and experience in the aerospace and communications fields. It was revised after three years of teaching the course to working engineers and managers, and published as *Systems Architecting, Creating & Building Complex Systems* (Rechtin, E., Prentice-Hall 1991)[2].

A typical student, by design and interest, is a fully employed engineer or manager with seven years of systems experience beyond a bachelor's degree in engineering, fully sponsored by a major firm (e.g., Hughes Aircraft, Rockwell International, Caltech's Jet Propulsion Laboratory, Northrop, and The Aerospace Corporation). About 1/3 are managers, 1/2 are technical specialists, with the remainder being younger or foreign students. The students made major contributions to the textbook and continue to make them to the field through investigative reports and research studies. The most common student motivation has been to broaden their careers, but in a technical, rather than a business, direction; i.e., to obtain a technical equivalent of an MBA. About 200 students over a five year period have taken the first semester in systems architecting fundamentals. Of those, more than 40 have taken the second semester in research challenges and directed study.

Having demonstrated feasibility and value, the course became an elective in 1989 in the three most involved Departments: Aerospace Engineering, Industrial & Systems Engineering, and Electrical Engineering/Systems.

During the same time frame, 1988-1992, the major firms for business reasons decided to upgrade the educational level of their existing engineering staffs. Concurrently, the USC School of Engineering began contemplating a multidisciplinary, technical masters degree program that might help support, and be supported by, industry.

The result of these mutual interests was the creation, based on advice from the industry, of the USC MS SAE Program. The degree was first offered in September 1992. Candidates with less than three years experience in major systems are discouraged. Several dozen candidates are now studying for the degree; it takes students about 3 years to complete. Courses in five different Engineering School Departments and three USC Schools are offered in the curriculum. There is no plan as yet to offer systems architecting at the undergraduate level. A PhD offering is under consideration.

The most recent technical development in the program is an increased association of the MS SAE program with the USC Center for Software Engineering, also in the School of Engineering. The strong trend into smart systems is a natural motivation.

Course requirements[16]

The MS SAE is intended as a capstone degree to a university education. To be appropriate for experienced engineering students, the curriculum both complements and broadens each student's own background.

The curriculum consists of ten 3-unit courses of which three are required: the first semester of systems architecting, advanced engineering economics and one of three system design courses. Seven electives, tailored for each student, are chosen out of approved offerings in a dozen technical specializations. The curriculum recognizes that architects must understand a broad range of subjects but still be a specialist in at least one; i.e., architects in practice are building architects, aerospace systems architects, communication systems architects, software systems architects, and so on. To accommodate the needs of the employed students, many of the courses are taught after working hours, over an Instructional Television network, and via delayed tape to other locations in the United States and Canada.

All courses are taught and quality controlled in the established departments (AE, EE, ME, ISE, CSci, etc.). By design, the MS SAE is a cross-cutting program across existing departments, not a mini-department.

Research directions

The University of Southern California is one of the major research universities in the United States and believes that teaching and research in science and engineering are mutually supportive.

Initial research has been focused in two areas:

* Studies of major complex systems from an architectural perspective. Lessons learned are codified into heuristics useful for modifying or better understanding other programs -- as well as the studied programs themselves. Some 150 of the best studies are in a research library, available to new students and for graduate research.
* Studies of systems architecting as a process, particularly the generation and use of heuristics. Although the paradigm of inductive structuring leading to analytic analysis and design is understood, formal tools for doing so have not yet been developed. There are as yet no standardized representations for complex systems, nor formalized means of capturing the intuitive, non-analytic parts of the design. Surprisingly, useful heuristics are numbered in the many hundreds, over 700 to date, presenting a research problem in how to consolidate, organize, and access them as architecting aids.

Academic-quality research mandates peer review. To facilitate that review the MS SAE program has been working with the National Council on Systems Engineering (NCOSE) and the Institute of Electrical and Electronic Engineers (IEEE) which are in process of establishing a technical journal in the field of systems engineering. Systems architecting will be a part of its coverage.

Conclusions drawn from experiences with the program

* Systems architecting is a recognizable field, teachable, researchable and useful.
* Heuristics are more numerous and of broader application than expected. Appropriately constituted, they may be the foundation of non-analytic, inductive, General System Problem Solvers, complementing the analytically-oriented ones of George J. Klir.[4]
* Students and companies are interested in, and supportive of, the field and its educational component.
* The present educational trend is to increase the educational emphasis on real-time software systems, human factors and the affects of the political process on systems design.

References for Parts One and Two

1 Rechtin, E., "The Foundation of Systems Architecting" for publication in the NCOSE/IEEE Journal. Fall 1994

2 Rechtin, E., *Systems Architecting, Creating & Building Complex Systems.* Prentice-Hall, New York 1961

3 Alexander, Christopher. *Notes on the Synthesis of Form* Harvard University Press 1964

4 Klir, G.J. *Architecture of Problem Solving* Plenum, 21-22, 1985

5 Pearl, Judea *Heuristics* Addison-Wesley 1984

6 Lang, Jon *Creating Architectural Theory* Van Nostrand Reinhold, 1987

7 Rowe, P. G. *Design Theory* The MIT Press Cambridge, MA 1987

8 Rechtin, E. *A Collection of the Best Student Papers on Systems Architecting ,1988-1993* University of Southern California 1994 (unpublished but selectively available for research) 150+ reports on complex systems written by industry-experienced, USC graduate students.

9 Rowe, Alan J. *The Meta Logic of Cognitively Based Heuristics.* University of

Southern California, School of Business Administration, Los Angeles, CA 1988

10 Canty, Donald "What Architects Do and How to Pay Them" *Architectural Forum* **119**, 92-95 September 1963

11 Evans, J. Harvey "Basic Design Concepts" American Society of Naval Engineers Journal November 1959. The original paper on which figure 10 of this paper was based.

12 Rahimi, M., P.A. Hancock and A. Majchrzak "On Managing the Human Factors Engineering of Hybrid Production Systems" *IEEE Transactions on Engineering Management* Vol 35 No. 4 Nov 1988

13 King, Nelson and A. Majchrzak "Concurrent Engineering Tools: Have They Missed the Human Factors Boat?" Unpublished 1994 graduate research study, University of Southern California, Los Angeles, CA 90089

14 Majchrzak, Ann *The Human Side of Factory Automation* Jossey Bass San Francisco 1988

15 Forman, B. "The Political Process and its Effect on Systems Architecting Design" Unpublished notes for a course of the same name. 1993 University of Southern California, Los Angeles CA 90089

16 *Master of Science in Systems Architecture and Engineering* (brochure) Systems Architecture and Engineering Program. Tel. (213) 740-7295 FAX (213) 740-7290 University of Southern California, EEB 504, Los Angeles, CA 90089-2565.

ARCHITECTING	PROGRAMMING
interaction among parts	implementation of parts
structural focus	computational focus
declarative descriptions	operational descriptions
implictly static	generally dynamic
system-level performance	algorithmic performance
major issues outside modules	major issues inside modules

Figure 8
Comparing Architecting with Programming *

* Shaw, Mary, "Architectures for Software Systems," Lecture 7 in Sledge, Carol A.,*Software Design,* Continuing Education Series Carnegie Mellon University Software Engineering Institute April 1994.

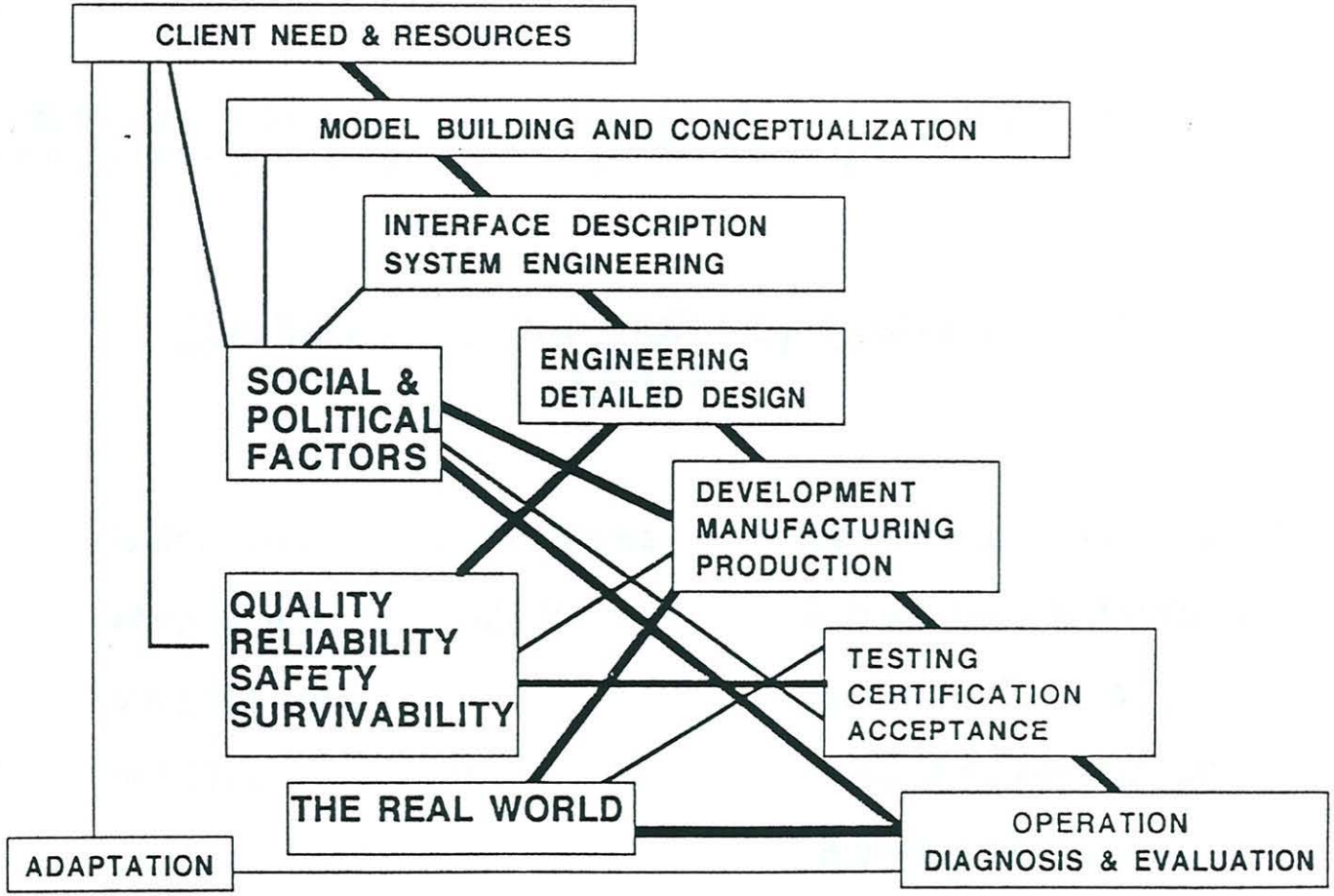


Figure 9
The Expanded Systems Waterfall

U.N.-ER-9

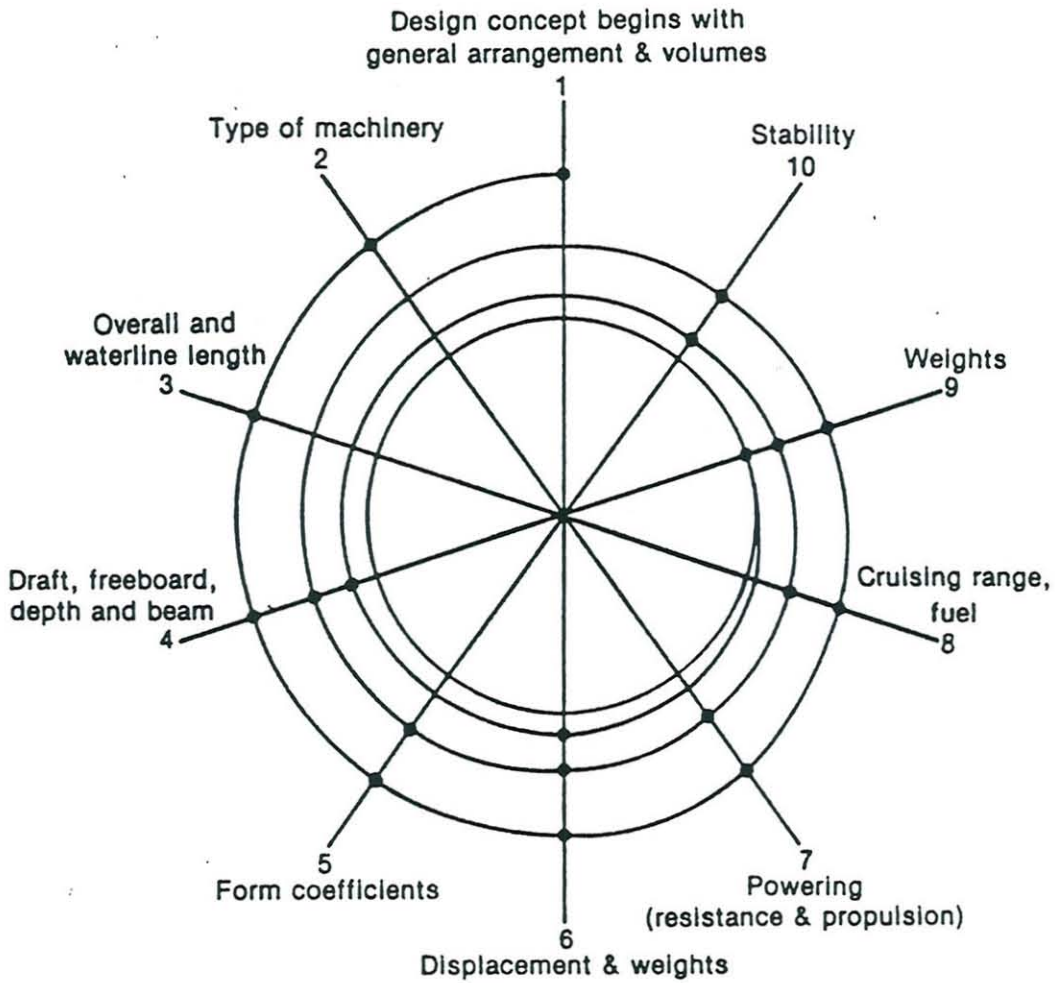


Figure 10 The Design Spiral*

*Courtesy Geoff Peters, Naval Surface Warfare Center, Carderock Division Code 20 Bethesda, MD 20084-5000

DISCUSSION

Rapporteur: Alcides Calsavara

Lecture One

After the lecture, Professor Randell cited the quotation "A complex system that works evolves from a simple system that works", from the book Systemantics, by John Gall, remarking that Professor Rechtin did not say anything about such approaches as evolutionary acquisition, which seems like an alternative approach to the way he had described the several other alternatives to architecting. In response, Professor Rechtin said that the more extreme version of that is that "You build a clock by clock.". On the other hand, if you do not have an idea of what the end-architecture is, then we all know what happens in the building if you do that; you get things that in English we say "built like topsy" -- miscellaneous things are sort of added-on that may or may not get a satisfactory ending result. So, although you would like to start with simpler things, with software modules, and you would like to take software modules and connect them together, unless you have some idea as to what the end-point is likely to be, some broad checking, some broad system functions, when you start to assemble all this you can very easily run into trouble. Well, it turns out that human error rate for alert awake dedicated thoroughly committed people is one percent. Figure out how many software errors there are then in the individual miscellaneous pieces of software. And you may spend a long time with what is called "main laine", or something -- terms and euphemisms. And what you are really trying to do is to dig out all the errors which are natural in human beings. And when you put the machine together, the errors are in the machines, and you have the same problem.

So, starting with simple systems is fine if you treat the simple system as a model for something you will go. Modelling is an abstraction; modelling is a heuristic, model is not reality. The trickery in a model is what you have excluded from your model; you have assumed in this model certain things and all the other things are regarded as unimportant. The colour of an automobile is probably not really that critical at first. Well, that was easy, but some other things might not be. That you can use the same tires on new automobiles as on old automobiles, that is not true. That you can have a better autobahn safety record with anti-block braking turns out not to be true. The Germans, unfortunately, have shown that anti-lock brakes increase the probability of failure accidents. The reason is something that the rest of the world might not understand but there is no speed limit on the German autobahn and it turns out that the Germans like to go as fast as possible. And they figured out that with anti-lock braking they can probably drive closer to each other. So they come faster and closer and an accident may happen. So, the simple systems have yet to be careful about what you are saying. There is a famous one called "no silver bullet in the software" and in essence it says: build things in parts which work and build on that and see if it works and if it works build on that and see if it works and if it works ... OK, as long as you know you are going there and not there. The architecting is their part of this. And it is better to do that at first. Then you can say "What is the simplest thing I can do which will demonstrate some aspects of the system function and can make it worth it".

Dr Aho asked Professor Rechtin to give some examples of what he considers well architected software systems or to give clues of some of the better software architects of the world today. Professor Rechtin answered that you may regard that as a generation problem and try to figure out how to make people use more sophisticated systems. He realized at the age of 60 that he was to old to learn software but not to tell software people something about systems. He could name the architects in a large number of other kind of systems but not in software. There were architects in the Apollo Program to the Moon, and they are list-able. He was one of the architects, using what we used to call architecting in 1960, and it worked. There were no architects in the Space Shuttle. And, as you know, it ran into difficulties because the first objectives and the end results were

not tied together very well. There is a long list of very successful complex systems in the text books, which leads to an interesting dilemma. There are lots of causes for failure. It is very difficult to say that a given system fails for one particular cause. As a matter of fact, one weak heuristic that we figured out, says that if you can design a system really well. You can probably stand one failure and not have a complete disaster, two is unlikely. That came out of the testing of aircraft, originally.

If there is only a single cause of failure the pilot will recover and it will be al right. If there are two, the different consequences of failures interact and it becomes extraordinarily difficult for the human mind to untangle them in real-time. If it is three it is hopeless even after analysis. And the difficult part of failure analysis is not conceiving a failure that might have happened, but proving that all other possibilities could not. There is a kind of long entrance to success and failure in business that we do not know. There is one exception and that was the one that invented the IBM 3/60 operating system. There are others but he was the architect and he did form a very small team. It was the first time that architecting teams were probably closest to a surgical team in the way they are to be formed. We have not seen those ideas seriously contradicted yet.

Lecture Two

During the lecture, while Professor Rechtin was explaining a diagram named "The Expanded Systems Waterfall", several questions related to it were raised. Firstly, Dr Kramer asked for a more detailed explanation about the box called "Real World", so Professor Rechtin answered that the real word is the one we mostly forget but it is the one that counts. It comes in and tells you things that you did not plan; that box includes such things as the unknown unknowns. You know you do not know all the information but you also know that you do not know that you do not know certain things. It is what happens when you do a test and something shows up and you say "Where would that come from?". It came out from the real world, it came out of the interaction of people and things and circumstances. It came about because you cannot replicate data perfectly, it came about because the instrument you bought yesterday does not turn out to be the same one which you bought two years ago and you thought it would make the same measurement while you should know better that the real world would tell you that "Why do you expect things to be perfect?".

Then Dr Kramer asked why is the "Real World" box only in the three bottom layers, to which Professor Rechtin replied that that is where it shows up. You are beautifully abstracting, analyzing, you think you should know all the parameters, you think you have done all the different things you needed, you think you have collected all the data you are supposed to do, but, when you get to the bottom layers, guess what: you try to make the thing but it will not work. Your can find, for example, that you have designed something to work beautifully but not to be built. The Australians know that very well indeed. They have a huge auditorium in Sidney, and it was designed to work but nobody could figure out how to build it, unless they do another complete building first, pile this one on top of it and then remove the inside. Expensive and elegant: two whole buildings. They never designed it to be built. You can easily construct mechanical devices that will work but that you cannot make. You can design something which has two little pins coming out each side to connect to little holes, but the two little holes are in a rigid structure and the device with the pins is also a structure and you cannot put the pins in the holes! If you can only put them in the holes you have a marvellous device, but you cannot put the thing in. That is the "real world".

Then Dr Aho asked for more clarification of the box named "Client Needs & Resources". The software people call these things the non-functionals. All the rest is supposed to be functional, but those are the non-functionals. What they mean is that the specification which cannot be strictly functional, such as performance, they call it non-functional. Professor Rechtin remarked that non-functional sounds to him like "disfunctional". But

what these things really are, are the imperatives of the clients, that is what the clients are interested in. So, reliability is whatever you can get. If you ask them: "Do you have a reliability specification for your instrument?". They say: "No, we build the best instrument that we can at a reasonable cost and reliability is whatever turns out to be.". Well, a customer does not think in those terms. He wants to know whether it works or not, if it works under these conditions and for how long.

Later, during the lecture, Professor Randell questioned Professor Rechtin how much commonality (terminology, people, how people think and so on) there is actually between the aerospace system that Professor Rechtin was using as the main example in his lecture and other systems, such as the railway system. Then Professor Rechtin answered that one of the famous stories about aerospace and railways is that when we first began to get into satellites, a very famous person in communications and part of AT & T Systems said "We do not intend to be the American Association of Railroads in the era of airplanes.", by which he meant "We do not intend to be American Association of Ground-Wires at the time of satellite communications.". So the Bell Systems marched very smartly into the satellites with their own particular ideas, with their own carriers. They wanted to say "We are not the ground carrier, we are in communications.". And so, the tie between railroads and aeroplanes is that we are in the business of transportation. Then, of course, there are similarities. But if you insist that railroads consist of iron and rails while aeroplanes consist of wings and tires they do not look alike. Putting it in another way, if you go up in the level of abstraction, you will probably see most of the similarities, for in the abstract we can talk about transportation, in the abstract we can talk about architecting whether it is hardware or software or social systems or whatever. And you start to see the similarities in the abstract level. At the detailed level: no, much less so. But at the abstract level we have a good chance, and that is where most of the bridges can occur between systems and software: at the abstract level. If we try to get too detailed we will get into the real world of each, and that is for the specialist in each. But the similarities tend to be at the higher levels of the abstractions and I think that is one of the major bridges, and architecting is a very abstract thing. So, it is a good chance where we can come together, by talking about it and trying to understand what we mean by some of these more abstract terms. And, of course, we will not see them in exactly the same way. But if you ask how do we feel about it, we might be very close. For example, software really in many applications "is" the system. There is not much else; there is the hardware where you put it on, but the name of the game is "software". But you do not need to go very far until you get into real-time systems, and real-time software now is the system because you are involving interaction, you are involving human being, you are involving many other aspects, and real-time software is quite different from what we say "old batch processing".