Software Tech News

1998

The DoD Source for Software Technology Information.

In This Issue :

Read additional Software
Architecture articles at:

www.dacs.dtic.mil/
awareness/newsletters/
listing.shtml

DACS

COTS Software: Five Key Implications for the System Architect

by Kurt Wallnau, Ph.D. - Software Engineering Institute

Introduction

Some of the most significant changes that have confronted DoD software acquisition efforts in the past few years are the result of using Commercial Off-The-Shelf (COTS) software. However, these changes are not unique to the DoD—virtually all segments of US Government and industry have been forced to deal with the implications of COTS software. These changes are the inevitable and irreversible consequence of increasing industrial and social reliance on computing technology. And if this assertion is not convincing to the DoD program manager, there is a range of Government and DoD acquisition policies, guidelines, and directives that provide more than ample motivation for using COTS software.

The implications of COTS software on DoD software acquisition are many and varied, as suggested by the SEI monograph series on COTS software [1]. This short article is focused more narrowly on the topic of COTS software on software architecture. To side step the issue of what is meant by "architecture," this article examines how COTS software affects the strategies and tactics employed by the successful system architect or lead designer. Although this article focuses on the architect, DoD program managers and executives will find this information useful in understanding the issues faced by integration contractors, and in assessing how well integration contractors are responding to these issues.

Software Architecture

Software Architecture Representation: Architecture Description Languages and "Styles"

John Salasin, Ph.D. -
Defense Advanced Research Projects Agency (DARPA)

What is Architecture and What is it Good for?

Systems are represented by a continuum of notations ranging from highly informal descriptions of functional requirements to the (fully specified) executing system. "Architectures" occupy a space in this continuum. They are more explicit than requirements, since they describe components and component interactions. They may be less explicit than detailed designs, since they do not describe specifically how the building blocks function. However, the separation between architecture and design is fuzzy at best.

According to Garlan and Shaw, [6] [Software architecture] goes beyond the [design of] algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

One adds to this, the explicit representation of constraints on or boundaries of a system. Much

of system integration requires understanding constraints (e.g., with respect to data access, event sequencing and timing, resource utilization, allowable parameter values, allowable topologies, fault tolerance, real time, survivability, redundancy, and replication) of components that are composed in the context of a hardware and software architecture.

An architecture is said to represent a family of systems rather than a single instance. If

this is to be useful, we must have some way of representing what the boundaries of the family are; what is "inside" and what is "outside" the architecture. The constraints can provide explicit boundaries on implementation variability and dynamic modification.

One way of defining architecture (or Architecture Description Languages (ADL) or architectural styles) is by the functions they perform. Figure 1 describes them by analogy with language typing.

Architecture and Language Types

Provide checking and generation
Simplification through specialization

Data Types	Architectures (Styles)
• Abstracts data types (strong typing), e.g. ▪ X := list of apples ▪ Y := array of oranges	• Abstracts component interactions ▪ Pipe and Filter ▪ Transaction Processing
• Defines legal operations, e.g. ▪ Apples + Apples OK ▪ Apples + Oranges Ⓧ	• Defines legal connections/ interactions ▪ Pipe => Filter ▪ Pipe Ⓧ Transaction
• Generates code to implement logical operator specialization ▪ "+" for array, vector, boolean ▪ "sort" for integer, real, character	• Generates "glue" to implement component interaction/constraints ▪ Control relationships for Pipe/ Filter vs. Transaction Processing ▪ Triggers to control (dynamic) typology

Figure 1: Architecture and Language Type Comparison

New-Generation Architecture Description Languages are Useful for Four Reasons

ADLs:

1. Enable automatic analysis and early detection of errors. We can analyze architectures to prevent errors and to generate automated runtime checks.
2. Enable reuse and product line development. We can use architectures to formalize component interrelationships in families of related systems tailored to a specific domain.
3. Support incrementality. Architectural specifications allow us to do the minimum work required to accommodate change.
4. Support optimization (non-functional attributes). We can use architectures as the basis for optimizing component placement and partitioning with respect to "non-algorithmic" attributes such as performance, reliability, security, and safety.

Examples of Automatic Analysis

Architecture provides abstractions adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. The abstractions encompass multiple views,

varying in level of detail and properties represented (e.g., data or control flow views, timing, and resource use). They need to support both static and dynamic analyses.

Static analysis includes internal consistency checks, such as whether appropriate components are connected and their interfaces match. Certain concurrent and distributed aspects of an architecture can also be assessed statically, such as the potential for deadlocks and starvation, performance, reliability, security, and so on. Finally, architectures can be statically analyzed for adherence to design heuristics and style rules.

Examples of dynamic analysis are testing, debugging, assertion checking, and assessment of the performance, reliability, and schedulability of an executing architecture.

Specific languages provide different types of checks. Thus, for example:

- Wright [3] detects mismatches between parts that fail to agree on protocols of interaction. It identifies race conditions and potential deadlocks.
- Aesop [2] provides facilities for checking type consistency, cycles, resource conflicts, and scheduling feasibility.

- C2 [13] establishes adherence to style rules and design guidelines.
- Rapide [7, 9] simulates architectures in terms of Partially Ordered Sets of Events (POSETS) and animates their execution. It provides tools for viewing and filtering events generated by the simulation (or an executing system).
- MetaH [15] analyzes schedulability, reliability, fault handling, and, security errors.

Examples of Enabling Reuse and Product Line Development

A product line is a group of applications that share a common architecture (e.g., a standard set of applications for Missile Guidance, Navigation and Control (MGN&C)). Large-scale reuse is possible because the applications share a set of generic components and employ common interaction protocols.

An architectural style is a recurring pattern of system organization. It defines a standard vocabulary of components and connectors and rules for their use. The use of architectural styles can promote design reuse by clarifying the context of applicability of particular solutions. It can also promote code reuse by permitting

Software Architecture Representation
Continued from page 3

shared implementations of invariant aspects of an architectural style.

The use of composition/ generation based on architecture, run-time constraint checking, architecture generated testing, and architecture recovery (for legacy systems) can help assure that the implementation is a valid instantiation of architecture.

All ADLs unambiguously specify the important interfaces in a system and provide tool support for using/checking these interfaces. This is critical for enabling the reuse and easy reconfiguration of subsystems/ components.

Some specific language examples include:

- Wright [1] was used to model and analyze the Runtime Infrastructure (RTI) of the Department of Defense (DoD) High-Level Architecture for Simulations (HLA). The original specification for RTI was over 100 pages long. Wright was able to substantially condense the specification and reveal several inconsistencies and weaknesses in it. It is now being used as (one) tool to determine if implementations conform to the HLA product line.

- SADL [14] was applied to an operational power-control system, used by the Tokyo Electric Power Company. The system was implemented in 200,000 lines of Fortran 77 code. SADL was used to formalize the system's reference architecture and ensure its consistency with the implementation architecture.

- Rapide [8] has been used in several large-scale projects thus far. A representative example is the X/ Open Distributed Transaction Processing (DTP) Industry Standard. The documentation for the standard is over 400 pages long. Its reference architecture and subsequent extensions have been successfully specified and simulated in Rapide.

Examples of Support to Incrementality

Incrementality means reuse, rather than redevelopment, in the face of change. It allows us to do the minimum work required to accommodate change. Architectural representations and analyses can support incrementality in four ways. They can provide:

1. Assurances that properties can be relied upon while the system evolves, with these properties expressed at an architectural, rather than code, level;

2. Automated code development / evolution, where architecture modification triggers code modification;

3. Automated support to test and analysis, where the architecture is a basis for specifying / deriving test and analysis plans (or plan modifications); and

4. Dynamic (run-time) modification by specifying and controlling change mechanisms.

Some examples:

In Integrated Modular Avionics (IMA) systems, global architectural constraints can ensure that no defects in partitions at lower levels of certification could interfere with the proper operation of more highly certified partitions. Enforcement of such laws has been coded in MetaH.

Rapide's Constraint Checker analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture. A Rapide model of

the architecture of a Chip Fabrication Line control system is now installed in TIBCO Software's demonstration facility. It uses an event hierarchy to quickly zero in on a low-level error.

Darwin [11] allows deletion and rebinding of components by interpreting Darwin scripts. C2 specifies a set of operations for insertion, removal, and rewiring of elements at runtime [10]. C2's ArchShell tool enables arbitrary interactive construction, execution, and runtime-modification of C2-style architectures by dynamically loading and linking new architectural elements.

MetaH[15] includes a feature called a mode, which allows the set of processes, or the connections between those processes, to be changed dynamically by the application during system operation.

Examples of Support to Optimization

Architecture is the way in which non-functional requirements such as performance, fault tolerance, and security/safety concerns are expressed and analyzed. If we were only interested in functional requirements, we could write a formal spec in Z, automatically convert it to Prolog and let it run (slowly).

Performance-related properties are important determinants of design. Carnegie Mellon University's Acme-based performance analyzer uses stochastic models to calculate latency, throughput, and bottlenecks for systems that use asynchronous message passing. Aesop analyzes resource conflict, and checks scheduling feasibility. MetaH optimizes the generated glue code/middleware for each application, significantly reducing the time and space requirements for communication, dynamic reconfiguration, etc. It supports analyses of schedulability, reliability, and security.

DARPA's Work in Architecture

Architecture (or architecture-centered systems) is key to DARPA's Evolutionary Design of Complex Software (EDCS) concept of evolution. The ability to define and analyze system designs and to specify and analyze changes at the architecture level are important notions for evolution. In addition, the ability to evolve systems through generation and composition technologies based on architecture makes evolution more affordable and increases the confidence associated with system change.

Research is going on to improve our ability to represent, evaluate, and analyze architectures, and to use these architectures to generate or compose systems. Projects are attempting to identify and quantify the benefits provided by various architectural representations. We are emphasizing architectural languages and analysis tools that describe systems in terms of component interactions and legal and illegal sequences of events, as contrasted with more traditional design tools that emphasize component topology and configurations. The EDCS program is adding notions of constraints [4], dynamic configurations, and standard representation [5].

Medvidovic provides an excellent summary of language features. [12]

Further information can be found at: http://www.darpa.mil/ito/research/edcs/index.html.

Software Architecture Representation

About the Author

Dr. John Salasin has conducted information processing research for his entire professional career - - on systems ranging in size from the encoder mechanism of a single cell in the Limulus (horseshoe crab) eye to the World Wide Military Command and Control (WWMCCS) system. His education includes a Ph.D. in Computer Science (1972), a M.S. in Neurophysiology (1969) from the University of Minnesota,, and a B.S. in Zoology from George Washington University (1964).

Author Contact Information

Dr. John Salasin
DARPA / ITO
3701 North Fairfax Drive
Arlington, VA

jsalasin@darpa.mil

References

[1] HLA: A Standards Effort as Architectural Style, Robert Allen, Second International Software Architecture Workshop (ISAW-2), October 1996.

[2] http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html

[3] , Robert Allen and David Garlan, A Case Study in Architectural Modeling: The AEGIS System, Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD-8), March 1996.

[4] Robert T. Monroe, Armani LRM Edition 0.1, April 1997, personal communication

[5] David Garlan, Robert Monroe, David Wile, ACME: An Architecture Description Interchange Language, http://www.cs.cmu.edu/afs/cs/project/able/www/acme-web/v3.0/white-paper-v3.0/white-paper.html

[6] D. Garlan and M. Shaw. An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing, 1993.

[7] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapid. IEEE Transactions on Software Engineering, 21(4): 336-355, Apr. 1995.

[8] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann. Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4):336-355, April 1995.

[9] D. Luckham and J. Vera. An Event-based Architecture Definition Language. IEEE Transactions on Software Engineering, 21(9):717-734, Sept. 1995.

[10] Nenad Medvidovic. "ADLs and Dynamic Architecture Changes", Proceedings of the Second International Software Architecture Workshop (ISAW-2), pages 24-27,San Francisco, CA, October 14-15, 1996.

[11] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pages 3-14, San Francisco, CA, October 1996.

[12] Nenad Medvidovic and David S. Rosenblum , "Domains of Concern in Software Architectures and Architecture Description Languages", Proceedings of the 1997 USENIX Conference on Domain-Specific Languages, October 15-17, Santa Barbara, California

[13] P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-Based Runtime Software Evolution. Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 19-25, 1998.

[14] "Introduction to SADL 1.0", SRI Computer Science Laboratory Technical Report SRI-CSL-97-01, March 1997.

[15] Steve Vestal, "Mode Changes in a Real-Time Architecture Description Language," International Workshop on Configurable Distributed Systems, Pittsburgh PA, March 1994.

DACS

Research Directions in Software Architecture
Major Mark J. Gerken — Air Force Operational Test and Evaluation Center

Introduction

Over the last several years, there has been an increased emphasis on techniques for specifying and analyzing software architecture. During this time, software architecture research has generally fallen into one of the following four areas:

1. Architecture representation;

2. Transforming and communicating architectures;

3. Architecture-based analysis; and

4. Architecture-based generation.

One of the basic premises of architecture-based research is that systems can be specified, designed, analyzed, built, tested, and evolved *through architecture.* Thus researchers are seeking to make architecture explicit and, to some degree, formal, and are seeking to provide manipulation and analysis tools supporting architecture-based development and evolution.

The use of architecture specifications in the development of software intensive systems is depicted in Figure 1. As seen in the figure, architecture specifications typically identify three elements:

1. Components (the loci of computation);

2. Connectors (data conduits or other relations between components); and

3. Constraints, which may address both structural aspects (styles) and behavioral aspects.

Architecture Description Languages (ADLs) generally provide support for specifying these three elements, although the level of support varies between ADLs. For example, Wright [2] emphasizes component interaction protocols while UniCon [10] emphasizes architectural styles. As can be inferred from this discussion, there is no widely accepted definition of the term "software architecture." Architecture " ... is generally taken to be a view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission."[5] Rather than attempt to provide a formal definition of software architecture, this article reviews the major research areas listed above and identifies a few usage considerations.

Architecture Representation

Software architecture involves descriptions of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns. ADLs provide language support for expressing these elements and lend themselves to providing a scientific and engineering basis for design, analysis, and composition. Some language and tool development efforts, such as the Domain Specific Software Architecture program, have as an additional goal support for domain specific language in architectural specification.


```
System simple-client-server = {
    Component client = {  Port send-request;
                          Properties {
                              request-rate : float=17.0;
                              source-code : external-file = "client.c" }}
    Component server = { Port receive-request;
                          Properties {
                              idempotence : boolean = true;
                              max-clients : integer = 1;
                              source-code : external-file = "server.c" }}
    Connector rpc = { Role caller;
                      Role callee;
                      Properties { synchronous : boolean = true;
                          max-roles : integer = 2;
                          protocol : Wright = "..." }}
    Attachments {  client.send-request to rpc.caller;
                   server.receive-request to rpc.callee;
```
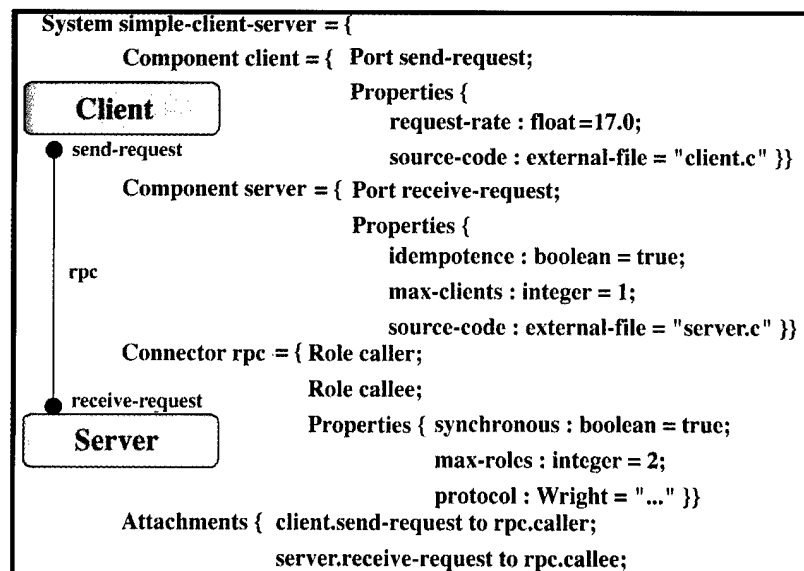
Figure 1: Architecture Specification in ACME

Research Directions in Software Architecture

Continued from page 7

Some progress has been made with respect to representation and generation:

- UniCon: components are loci of computation and state, and connectors are loci of relations between components; generates C/C++ code.[10]

- Wright: defines component interaction using process algebras.[2]

- Aesop: defines a system for developing style-specific architectural development environments.[12]

- Jakarta: defines a software generator environment that uses constraint propagation to refine and integrate reusable software artifacts.[4]

- Planware: Architecture is defined as a diagram of formal specifications.[11]

Transforming and Communicating Architecture

Although certain needs are shared by all ADLs, such as support for multiple views and tool support for control and data flow analysis, ADLs have been developed to meet different needs; thus expressive and analytical capability varies between them. Rather than developing multiple architectural specifications for a given system (or family of systems), a group of researchers is developing an architecture interchange language called ACME whose goal is to facilitate the exchange of architectural information.[1] These researchers are designing ACME along with translators into and out of ACME so that, for example, architectures described in UniCon can be translated through ACME into another ADL such as Aesop. This interchange language provides a way for different ADLs and ADL tool suites to work together. An interesting by-product of this research is that it is leading to a better understanding of what ADLs should be capable of representing and reasoning about.

Figure 2 depicts architecture interchange using ACME. The heavy arrows in the figure represent language to language transformations. Another type of transformation is one that takes place within the same language. These transformations seek to recast architectural elements into alternate representations. In conjunction with his work on the SADL language, Moriconi has developed several architectural transformations that can be used, for example, to enhance system performance.[9]
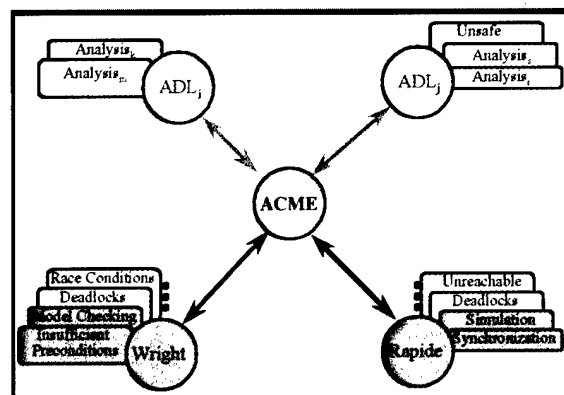
Architecture-Based Analysis

Architecture specifications are more than "boxes and arrows" diagrams. They are formal entities subject to analysis; they can be investigated to determine properties of the system(s) they specify. Investigating system properties at the architectural level has at least two advantages:

- Unnecessary implementation details are abstracted away, allowing a developer to concentrate on architectural rather than implementation issues.

- Early analysis. It is not necessary to have an implementation constructed before investigating family/system properties. For example, both Wright and Rapide were used to model the Department of Defense's High Level Architecture (HLA) simulation framework. Analysis of the Wright specification revealed HLA problems associated with distributed start-up, paused on join, and in-transit messages after a resign.[3] Similarly, analysis of the Rapide specification revealed that the run time interface could lose the event order and that there were orphaned attributes after a player resigned from the simulation.

Figure 2: Architecture Interchange

The depth and type of investigation will vary between ADLs, but generally speaking, investigations include:

- Static analysis. Ambiguities, incompleteness (e.g., missing connectors), wrong directionality, and, depending on the ADL, syntactic and semantic data type compatibility can be investigated.[13]

- Model checking, including insufficient preconditions, faulty control models, and latent deadlocks.[2]

- Simulation-based testing, including event order and causality anomalies.[7]

As a further example, the Model Integrated Computing (MIC) framework developed at Vanderbilt University has been used to investigate production flow at Saturn [6] . This investigation (through architectural modeling) led to a 10% increase in the throughput of Saturn's Spring Hill, Tennessee plant.

Architecture-Based Generation

This line of research seeks to make better use of architecture specifications in software generation. As shown in Figure 3, rather than have a target architecture implicitly defined by the generator, researchers are developing generators that take as input an architecture specification
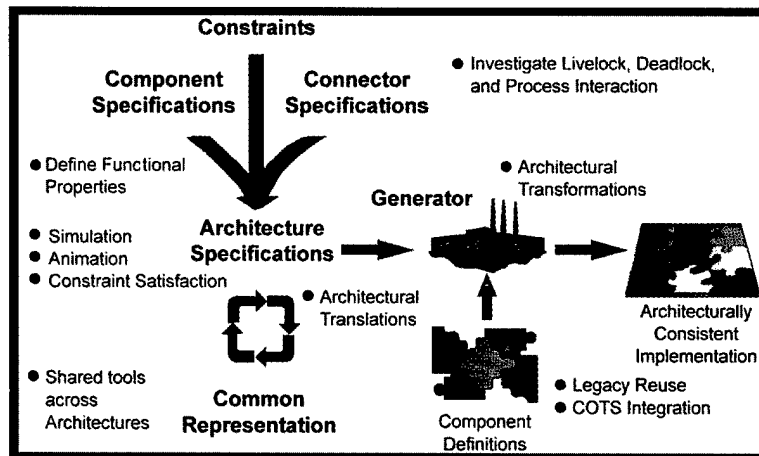

Figure 3: Architecture-Based Development and Evolution

of the target system. A closely related research topic is the specification of product line architectures.

The basic idea of product line development is to specify the architecture of a family of systems. Several efforts are underway in this area, including the Jakarta work at the University of Texas [4] and the Planware system under development at the Kestrel Institute.[11] The Planware system defines an architecture for a family of scheduling applications using formal specifications; implementations satisfying developer-selected constraints are generated from these specifications. Researchers at Kestrel have proven that the systems generated using Planware will find feasible schedules provided such schedules exist an important property of this family.

Usage Considerations

Several changes to current

system development practices may occur:

- Training. Developers will need training to understand and use ADL technology and architectural concepts/styles effectively.

- Change/emphasis in life cycle phases. Architectural design and analysis may precede code development; an ADL specification should provide a good basis for programming activities [10].

- Documentation. Because the structure of a software system can be explicitly represented in an ADL specification, separate documentation describing software structure may not be necessary.

- Expanding scope of architecture. ADLs are not limited to describing the software architecture; application to system architecture (to include hardware, software, and people) is also a significant opportunity.

Research Directions in Software Architecture

About the Author

Major Gerken has been a computer engineer with the Air Force for over 11 years. He was initially assigned to support large scale acquisition efforts, including the C-17 and AMRAAM systems. He later transferred to the Air Force Institute of Technology where he researched formal approaches for representing and reasoning about software architectures. After receiving his Ph.D., Major Gerken was assigned to the Air Force Research Laboratory's Rome Research Site where he directed research into formal methods and software architecture. He is currently assigned to the Air Force Operational Test and Evaluation Center where he supports test and evaluation of software intensive systems.

Further Reading

For a comparison of several ADLs, see [8] and for an in-depth treatment of architecture-based development, see [13].

Author Contact Information

Major Mark J. Gerken
Air Force Operational Test and Evaluation Center
HQ AFOTEC/TSS
8500 Gibson Blvd, SE
Kirtland AFB, NM 87117-5558

DSN: 246-7827
Telephone: (505) 846-7827
Fax: (505) 846-5145
gerkenm@afotec.af.mil

References

[1] The ACME Architectural Description Language. http://www.cs.cmu.edu/~acme/

[2] Allen, Robert and Garlan, David. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology,* July 1997.

[3] Allen, Robert; Garlan, David and Ivers, James. Formal Modeling and Analysis of the HLA Component Integration Standard. *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-96),* November, 1998.

[4] Batory, Don et al Jakarta: A Tool Suite for Constructing Software Generators. http://www.cs.utexas.edu/users/schwartz/JOverview.htm

[5] Software Engineering Institute (SEI): EDCS Architecture and Generation Cluster http://www.sei.cmu.edu/community/edcs/CLUSTERS/ARCH/

[6] Long, Earl; Misra, Amit and Sztipanovits, Janos. Increasing Productivity at Saturn. *IEEE Computer,* August 1998.

[7] Luckham, David C. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Technical Report CSL-TR-96-705, Stanford University, 1996.

[8] Medvidovic, Neno. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, University of California, Irvine, 1996.

[9] Moriconi, Mark; Qian, Xiaolei and Riemenschneider, R. Correct Architecture Refinement. *IEEE Transactions on Software Engineering,* 21(4):356-372, April 1995.

[10] Shaw, M. et al Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering,* April 1995.

[11] Smith, Douglas. Planware-Domain Specific Synthesis of High-Performance Schedulers. Technical Report AFRL-AF-RS-TR-1998-200, Kestrel Institute, 1998.

[12] Carnegie Mellon University. Aesop Software Architecture Design Environment. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html

[13] Shaw, Mary and Garlan, David. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall Publishing, 1996.

COTS Software
Continued from page 1

Five Key Implications of COTS Software

The following discussion is organized in terms of the implications of COTS software on the design activity, and the way in which a lead designer must accommodate these implications. In the interest of brevity only five implications are described. There are certainly other implications, but the five discussed are particularly revealing.

Before beginning, it is essential to be clear about the type of system that is being designed. COTS-*based* systems comprise a spectrum, ranging from COTS-*solution* systems at one extreme, to COTS-*integrated* systems at the other extreme. COTS-solution systems are pre-integrated systems that are customized and deployed for use; examples include enterprise resource management packages and payroll packages. COTS-integrated systems are assembled from (frequently many) COTS components provided by different vendors. Both extremes present unique challenges; this article is concerned with COTS-*integrated systems*.

Implication #1: Accept The Influence Of COTS Software On System Design

There are two common mistakes made by designers unfamiliar with the implications of using

COTS software. The first mistake is to design a system without reference to COTS software on the assumption that products are merely an implementation detail to be filled in after the major design decisions have been made. The second mistake is the complement of the first, that is, to blithely allow one or more COTS products to dictate the design of a system. The consequence of the first mistake is that opportunities to use COTS products will be missed, while the consequence of the second mistake is vendor lock. The successful architect will have a more balanced approach.

It is important to understand that COTS products often have an unavoidable impact on system design. Consider, for example, an architectural trade-off analysis conducted by the SEI for the Federal Aviation Administration (FAA) [2]. This study analyzed the architectural implications of using two different commercial technologies for interprocess communication, CORBA and POSIX.21*. The result of the study showed that each technology imposed its own unique constraints on the system design, resulting in different system structures and quality attributes (e.g., modifiability and performance).

Tradeoffs involving COTS software are not limited to just the system architecture. For example, the CORBA design might have better modifiability than the POSIX.21 design due to its object-oriented nature, but in exchange might have worse performance. Making this tradeoff might involve requirements. Can performance requirements be relaxed to obtain benefits in modifiability?
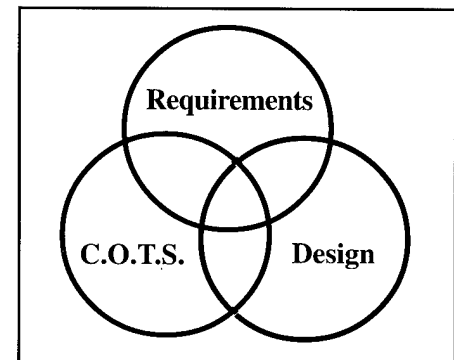
Figure 1: Tradeoff Regions

Figure 1 illustrates tradeoff regions as intersections among COTS software, design, and requirements. The architect needs to be actively engaged in each of these regions. The CORBA-induced performance versus modifiability tradeoff lies at the intersection of COTS product, design and require-ments. Other tradeoff decisions might involve only COTS products and the design. It is also possible for COTS products to influence system requirements independent of a design.

---

* CORBA: Common Object Request Broker Architecture. POSIX.21: Portable Operating System Interface, real-time communication annex. For simplicity we finesse the fact that CORBA and POSIX.21 are specifications rather than products. It suffices to say that there are commercial implementations of both specifications.

COTS Software
Continued from page 11

It might appear that it is difficult to manage all of these simultaneous tradeoffs and it is. Fortunately, there are tools that an architect can use to come to terms with these tradeoffs. The Architecture Tradeoff Analysis Methodology (ATAM)[3] used in the FAA study cited above is one tool. Another tool more closely linked to COTS software is *formative evaluation*. In brief, formative evaluation exposes how products ought to be used in a system. The architect will find formative evaluation useful early in the design process when there is more latitude for adapting design and requirements to product capabilities (and liabilities). A more detailed discussion of formative (and normative) evaluation can be found in an on-line tutorial on COTS software evaluation[4].

Implication #2: Plan for Instability

An ironclad rule of the software marketplace is that *things change*, and often change *very quickly*. New products and technologies emerge at a rapid pace (two thousand software products per month, according to CIO magazine [5]), and new versions of existing products add and modify capabilities in response to market pressures. The implication is clear: a design that has been influenced by commercial software may become infeasible in response to changes to that software. Of course, this takes a negative view of marketplace change when, in fact, it is precisely this market dynamism that produces steady improvements in product capabilities. So, in the same way that marketplace changes can render some design options infeasible, new design options may become feasible and perhaps desirable.

System designers have always had to accommodate the exigencies of change in the design activity. Designs evolve as more is learned about the problem at hand; and of course, changing requirements is the norm, not the exception. However, the software marketplace adds a new dimension of instability that becomes noticeably pronounced as the number of products used in a system increases. This instability is exaggerated where new and rapidly evolving technologies are employed, as is the case with Web technologies and distributed object technologies such as CORBA and Java™. It is an unfortunate "Catch 22" that these unstable technologies are usually the ones whose use is perceived (by customers, designers and end users) as being highly desirable, since it is precisely this interest that leads to the technology instability in the first place.

What can the architect do in the face of marketplace instability? One technique is to keep product-sensitive design options open for as long as possible. For large projects where the design activity may span many months this kind of "late binding" strategy may be appropriate. For example, three design options could be pursued: a safe option that is known to work with today's products; an anticipatory option that is expected to work with new capabilities that have been announced but not yet shipped by vendors; and a "blue sky" option that is more aggressively futuristic. For projects with a tighter timeframe for the design activity, an alternative to late binding is early binding, sometimes referred to as "anchor first." In this strategy early design commitments are made on key products that are presumed to be stable. For example, integration infrastructures such as message-oriented middleware products, or tool suites from relational database vendors, are often used as design anchors.

Selecting products as design anchors has the appeal of simplifying the design process, but on the other hand increases the risk of a system becoming too dependent on a particular software vendor. What if the anchor (or its vendor) turns out to be less stable than anticipated? This issue is taken up in the implications of *vendor lock*.

Implication #3: Sustain Core Technology and Product Competency

The first two implications combine to establish a third implication: deep product expertise is a critical design asset. Simply put, good design decisions about software products can not be made in the absence of sufficient (and often deep) knowledge about those products. As the number of products used in the system increases, so also increases the need for spanning expertise knowledge of how ensembles of products (or technologies) can be integrated. Unfortunately, given the dynamism of the commercial software marketplace, product expertise is a *wasting* asset: the useful half-life of expertise in some key technology areas is surprisingly short. For example, until recently "thin clients" via Java™ Applets was the "hot" technology. Today, experts in Web ensembles are much more skeptical about the universality of thin clients (see [6] for a good discussion of the pros and cons of thin clients and some practical alternatives).

Unfortunately, individuals with deep, spanning expertise across the range of products typically used in COTS-integrated systems for example, Web, database, transaction, distributed object, system management, and security technologies are exceedingly rare. Keeping current with any one

product in any of these product areas is difficult enough; and tracking an entire category of products (Web products, for example) can be a full time job. It is an expensive proposition to develop *and sustain* this level of technology competency.

How can the architect obtain the kind of product and technology expertise at the time it is needed? One choice frequently adopted is to hire consultants. In some cases this is the most economical approach, although there are two risks. First, the hiring organization often lacks enough expertise to assess the competency of the consultant and the consequence of accepting advice from charlatans is predictable. Second, consultants often have ulterior motives, especially if they are hired from a software product vendor, or if their expertise is limited in range rather than spanning if the consultant only knows how to wield a hammer, everything looks like a nail.

The architect does have another option in the use of model problems (a kind of formative evaluation technique see [4]) to develop "just in time" technology competency. Model problems are small-scale prototypes that are focused narrowly on specific critical design issues. Software products can be used to develop one or more model solutions, each representing design alternatives that have been

proven feasible (or infeasible). The trick in this case is to have sufficient technology competency to *recognize* a critical design issue relating to the use of COTS software (integration issues are a good place to start)—but this is not too much to expect from a COTS-savvy architect.

Implication #4: Understand Vendor Lock and Vendor-Neutral Options

The software market is driven by *differentiation*, not standardization, and it is often innovative (i.e., non-standard) features that cinch software sales. The temptation to take full advantage of unique product features is understandable, especially in high-end, expensive products. Using vendor-specific features can provide enhanced system capabilities, but on the other hand makes the sustainability of the system dependent upon a single supplier. There is a complementary temptation to insulate systems from specific products, usually as a hedge against market dynamism. For example, if a vendor goes out of business a new product can be inserted in place of the old, and clients will not be affected. Insulating products provides stability, but an abstract interface that can be mapped to competing products forces the system to rely on the common subset of features found in products.

COTS Software
Continued from page 13

Although there is no universal answer to this tradeoff, it is important that the architect is aware of the conditions in which unanticipated and *de facto* vendor lock can arise. One simple technique is to ensure that every product used in a system has a viable competitor that is commercially available. If competitors exist, then a separate design decision about whether or not to insulate the design from the product (through abstract interfaces, for example) can be made. If there are no viable competitors, however, no amount of insulation can hide the reality of vendor lock. Note that "standards" are not completely sanative; in some cases vendors extend standards (SQL is a classic example), while in other cases too few products may implement a standard to prevent *de facto* lock-in.

Another technique for the designer to avoid vendor lock is to allow the use of product-specific features, but only for non-critical or discretionary parts of a system capabilities that can be sacrificed. This can be a good compromise, but it does introduce a very slippery slope.

Implication #5: Use Business and Software Analysis in Design Decisions

The previous implications are combined into one last implication of great significance: the skills of the system architect must encompass both technical competency *and* business competency. This can be seen in each of the above implications:

- Managing tradeoffs between commercial products and requirements requires deep knowledge of the products as well as the mission or business area being automated. Frequently, the architect will be required to negotiate directly with customers, explaining the business implications of using commercial software in addition to technical implications for example, impacts on cost to sustain the system, added operational efficiencies, and return on investment.

- With respect to system and design instability, the architect must make early decisions regarding the kinds of technologies to be used. Clearly, some technologies are less stable than others; a decision to use a newer but less stable technology in place of an established but more stable technology requires investment analysis. Does mission criticality justify the added cost and risk of using unstable but "feature-rich" technologies? Or would "good enough" technologies suffice?

- Concerning technology competency, the architect must understand the costs of acquiring product expertise versus the cost of making a poor decision. In some cases sufficient design risk will warrant a significant investment in acquiring competency perhaps a small-scale prototype will need to be developed. The point is that uncertainty is inevitable with COTS software, and the management of uncertainty invariably requires business analysis.

- The need for business acumen is most apparent in addressing issues of vendor lock. Lock-in might be the outcome of a strategic alliance between product suppliers and integrators. Such alliances might bring competitive advantages to integrators, and cost and capability benefits to acquirers. Of course, technical and business risks accrue as well. In the DoD these issues can arise on a per-acquisition basis, given the size and longevity of the systems concerned. In these cases, the architect must play a key role in mediating the technical and business tradeoffs.

Summary

The system architect of the future will possess a range of personal and technical skills especially adapted to the implications of extensive use of commercial

software components. Acquirers will need to understand these implications in order to be informed consumers. In the final analysis, building COTS-integrated systems requires a partnership of owner, integrator and product suppliers.

About the Author

Kurt C. Wallnau is a senior member of the technical staff at the Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA, where he co-leads the Commercial Off-The-Shelf (COTS) COTS-Based Systems project.

Mr. Wallnau's current interests include the role of product evaluation in the system design activity, and the transition from COTS-based to component-based systems. Mr. Wallnau has published several papers and tutorials on COTS product and technology evaluation as well as on various facets of component-based software engineering with distributed objects. Prior to the SEI, Mr. Wallnau was the Unisys System Architect of CARDS, a $7M/Year DoD program focused on the use of COTS software as a strategy for improving software reuse.

Mr. Wallnau graduated in 1985 summa cum laude from Villanova University with a B.S. in computer science.
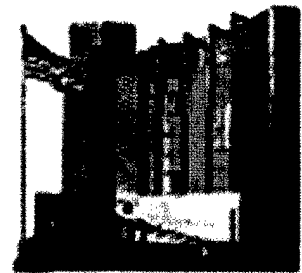
Author Contact Information

Kurt C. Wallnau
Software Engineering Institute
(412) 268-3265,
Fax: (412) 268-5758

kcw@sei.cmu.edu
http://www.sei.cmu.edu

The Software Engineering Institute

References

[1] See http://www.sei.cmu.edu/cbs/monographs.html for a list of available monographs.

[2] Meyers, C., Plakosh, D., Place, P., Klein, M., Kazman, R., "Assessment of CORBA and POSIX.21 Designs for FAA En Route Resectorization," Special Report CMU/SEI-98-SR-002, Software Engineering Institute, Pittsburgh, PA, April 1998. An online version is available at http://www.sei.cmu.edu/publications/documents/98.reports/98sr002/98sr002abstract.html.

[3] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipsom, H., Carriere, J., "The Architecture Tradeoff Analysis Method, Technical Report CMU/SEI-98-TR-008, Software Engineering Institute, Pittsburgh, PA. An online version is available at: http://www.sei.cmu.edu/publications/documents/98.reports/98tr008/98tr008abstract.html

[4] Wallnau, K., Carney, D., Morris, E., Oberndorf, P., Buhman, C., "A Tutorial on the Theory and Practice of COTS Software Evaluation," half day tutorial presented at the 20th International Conference on Software Engineering, Kyoto, Japan, 1998, and at the 1998 Software Engineering Symposium, Pittsburgh, PA. An online version is available at: http://www.sei.cmu.edu/cbs/cbs_slides/98symposium/eval_tut/index.htm

[5] J. Bresnahan, "Mission: Possible," CIO Magazine, October 15, 1996.

[6] Seacord, R., Hissam, S., "Browsers for Distributed Systems: Universal Paradigm or Siren's Song?" Technical Report CMU/SEI-98-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1998. An online version is available at: http://www.sei.cmu.edu/publications/documents/98.reports/98tr010/98tr010abstract.html

The Profession of Software Architecture

Laura and Marc Sewell - Worldwide Institute of Software Architects

Introduction

It is beyond debate that the software development industry is characterized by troubled and failed projects fraught with missed deadlines, scrapped code, muddy accountability, and escalating cost. There has been a tremendous effort to improve tools and methodologies, as well as project and risk management techniques. However, the underpinnings of the software development industry are flawed and projects will continue to fail until it is understood that we are attempting to build huge information technology structures without architects, plans, and logical construction processes.

Mention the word "architect" and even kids on Career Day will intuitively form a clear image in their minds. They envision someone at a drafting table or at a building site supervising blueprints in hand. But put the word "software" before "architect" and the clear images become enshrouded in dense fog. This is true not just for children, but also for the clients of software development projects, the users of the systems, and even the software professionals themselves who have difficulty defining the title, responsibilities, and the role.

The Yellow Pages do not have listings for "Software Architects." There are no degrees offered in Software Architecture, and yet there are increasingly large numbers of software professionals assuming the title despite the role remaining vague and variable.

Departments of Computer Science are educating software professionals, but they are producing engineers, researchers, and programmers the builders of software systems not architects.

Nonetheless, a spontaneous, inchoate trend toward software architecture continues. It is time to take this beyond a trend by formally establishing the profession of software architecture. The Worldwide Institute of Software Architects (WWISA) has been founded to accelerate this movement, as the American Institute of Architects did in the 1850's.

WWISA
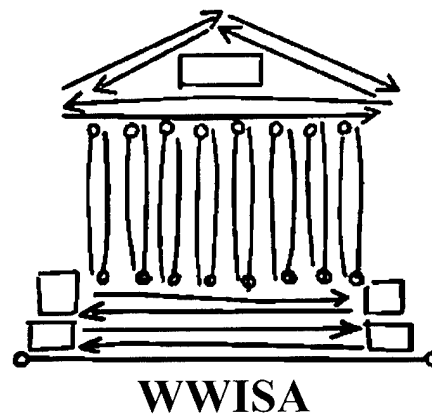
To build a foundation for this new profession, we need to look no further than our own human history. It is there that architecture is defined and its role understood.

Software Architects: Assuming an Ancient Role

"Architect" is a word with a great deal of history behind it; a history waiting to be assumed by the new profession of software architecture. Throughout the centuries, there have been architects who have arisen humbly from the trades and gone on to anonymously design and build cathedrals. There have also been great artists who have turned their attention to architecture at the behest of kings and popes. Since the Industrial Revolution, when the sheer variety, scope, and function of buildings multiplied dramatically, architects have been university-trained, licensed, and subject to professional standards.

Regardless of origins and the wide range of architectural styles through the centuries from Gothic to Post Modern the role of the architect has never varied. That role has always been to design structures to meet human needs and house their activities. This is a starkly minimalistic way to describe such masterstrokes as the chateaux at Chenonceau, Grand Central Station, and the Parthenon, but it is the definition of an architect and the only reason they exist.

The American Institute of Architects was established in 1857 to formally establish that

profession. It was understood, of course, that architects had successfully been educating themselves and designing structures for centuries, but the scope of the Industrial Revolution changed everything. Degree programs were established and standards and codes were introduced to meet the challenges of the new age. We are now at a similar point in the Information Revolution. Compared to building architecture, our history is rather abbreviated, but in another sense, the introduction of the professional software architecture is long overdue.

The Analogy

The analogy between building construction and software construction is not new. It has been used to illustrate points, especially to end-users, and to borrow terms like "architect," but it has never been fully developed. It is referred to as "simplistic" by software theorists, or dismissed as being just a tool for raising questions, but not supplying answers.

But the analogy is profoundly true and has the power to transform software construction out of its current crisis. It is simple and elegant with the elemental force to shift our current paradigm. It will empower not only software professionals, but clients and end-users, as well.

With the analogy, we can solidify the growing trend toward software architects, design, and plans as well as transform software processes, titles, roles, and accountabilities. With the analogy, we see that a software architect is as much an architect as Frank Lloyd Wright. An architect is an architect, whether a structure is erected from lumber, bricks, or computer code. Software architects design information technology structures to meet human needs and "house" our multifarious activities.

The Architectural Process

All architects, regardless of the "building materials," are client advocates, and it is there that the process begins. Before ground is broken, or a site even selected, the client hires an architect as a designer and guide. The architect first listens to the client and studies the needs, desires, problems, resources, and environmental issues all of which define the client's domain.

Based on these needs, preferences and constraints, the architect develops a vision of a structure and, in collaboration with the client, revises the plan until it is affirmed. The architect's role is to then guide the plan to reality, spanning the worlds of the client and the technical builders. The client can be wholly ignorant of technical aspects of construction (footings,

bearing walls, programming languages) but with the architect and blueprint, can validate and manage the logical, sequential building process.

The architect is the arbiter of design decisions and changes as the project continues, the design conscience, as it were. Just as in building construction, however, not all design decisions are made by the architect. The architect's plan outlines, for example, where the electrical outlets are located, but the electrician designs the actual circuitry and configuration of the circuit box. In turn, the electrician's helper would design the path of the wires strung through the walls and supports. Low and high level, construction and architectural level, design decisions are made in an analogous way in software construction.

The Software Architect: Bridging Clients and Builders

Despite the increasing numbers of self-conferred software architects and the emphasis on architectural design, many clients are, in a sense, going in the opposite direction. The risk of large project failure has led to a short-term approach in which small software applications are built, one at a time, and added to in a modular fashion.

The Profession of Software Architecture
Continued from page 17

This reduces risk and increases manageability but, through the analogy, we can see that it is akin to building a powder room on a vacant lot and worrying about the rest of the house later. It is fine to build in phases, but it is folly to begin to build without an overall plan of the entire structure. The lack of architects with understandable plans, and the resultant chaos, has thwarted client expectations to the point where this limited, shortsighted strategy has tremendous appeal. But the old adage applies "If you don't know where you are going, any road will take you there."

Compounding this problem is a chasm separating software professionals from the people they serve; the clients, users, indeed, the general public, who find themselves intimidated by scary lingo and acronyms. Software professionals, in turn, are frustrated by the ambiguities of their roles and responsibilities, as well as their inability to communicate effectively with clients and users.

Without effective communication and without understandable plans, clients are unable to validate and manage software construction and users are unable to communicate their needs. There is growing discontent, but few systemic solutions. The profession of software architecture provides a bridge over this gulf, as the profession of building architecture has since ancient times. Most clients and users, and certainly the general public, do not even know what "methodologies" are, but they can follow diagrams and drawings, as well as rely on the judgement of an architect who is accountable to them.

The architectural plan is consistent with needs and desires of the users and, at the same time, with the needs of the builders. Both sides are given a cognitive map of the design, as well as the logical process that leads to completion. Construction, even with a blueprint, is fraught with difficulty, but at least buildings get built and inhabited. Unlike software structures, total building failure is virtually unknown. With a profession of software architecture, the same happy fate awaits our information technology skyscrapers.

Architectural Education

Software architects are organizing to establish their profession, but the client or customer will be the true driving force of this movement by demanding architects with plans. Degree programs will follow which will not only train qualified architects, but will attract new students to information technology, where the numbers are now stagnant and insufficient to meet ever growing demand.

The field of Computer Science has an engineering and programming focus that fits students with that vocational profile and range of interests. Students with different interests, perhaps in business or liberal arts, express a desire to have a career in the information technology sector, but simply do not see themselves as systems engineers or programmers. So, they major in business or psychology, for example, and take a few computer-related courses thinking that will cover all bases. It does not, however, work out in "the real world" where they find their computer skills too superficial for practical use.

A degree in software architecture would be similar to a traditional architecture degree in its multi-disciplinary approach. Both forms of architects require grounding in the technical aspects of construction to know what can be engineered and built. They do not, however, have to master engineering and building techniques. The educational focus would be on information technology and design, as well as such disciplines as business, management, and art. The goal of a software architect's education is to provide a foundation in design and problem solving with information technology. These tools allow the architect to leverage the full range of information technology in the client's favor.

DoD Software Tech News, Vol. 2., No. 3 (January 1999)
1999 DACS Patron Survey

Your views and experience are important to us and so we ask that you take part by completing this questionnaire.

The first 100 people to return this survey will receive a computer shaped squeeze toy.

Name _____

Position or Title _____

Agency or Organization _____

Address _____

State _____   Country _____

Zip _____   +4 _____

Phone _____   Fax _____

E-mail _____   URL _____

Org Type
❏ Air Force
❏ Army
❏ Navy
❏ Marines
❏ DISA
❏ DTIC
❏ Other DoD
❏ Coast Guard
❏ Other Federal
❏ Commercial
❏ Non-Profit
❏ Academia
❏ Media
❏ Foreign
Other _____

1. When did you first become aware of the products and services offered by the DACS?

   Month _____ Year _____

2. How did you first become aware of the DACS products and services? (Please check all that apply.)

   ❏ Software Tech News   ❏ WWW Search   ❏ DACS Website   ❏ DTIC

   ❏ Colleague   ❏ Magazine   ❏ Other IAC   ❏ E-mail Advertisement

   ❏ Conference (Please tell us which one) _____

   ❏ Other Website (Please tell us which one) _____

   ❏ Other _____

3. How often does your organization use the products and services offered by the DACS, including visiting the DACS Website? (Please check one)

   ❏ Daily   ❏ At least once a week   ❏ At least once a month   ❏ Less than monthly

4. How many people from your organization have access to the DACS products and services? (Please check one)

   ❏ Just me   ❏ 2-10   ❏ 11-25   ❏ 26-50   ❏ More than 50   ❏ Don't know

5. What other sources do you use for Software Technology products and services?

   _____

6. Who uses DACS products and services in your organization? (Please check all that apply)

   ❏ Managerial   ❏ Research and Development   ❏ Faculty   ❏ Students

   ❏ Other _____

7. How would you rate the content of the DACS material you have received? (Please check one)

   ❏ Too general   ❏ Just right   ❏ Too technical   ❏ Don't know

8. **Have any of the products or services from the DACS ever saved you time or money? If so, please describe below:**

   _____

   _____

   _____

9. The DACS offers many services as the DoD Software Information Clearinghouse.
   Please rate your satisfaction with a few of the products & services of the DACS using the following scale:

   5=excellent **4**=above average **3**=average **2**=below average **1**=poor **0**=Don't know or never used

Customized Electronic Technology Magazine (CETM) http://www.dacs.dtic.mil/cetm/cetm.shtml	5	4	3	2	1	0
DACS Courses & Seminars http://www.dacs.dtic.mil/training/courses.shtml	5	4	3	2	1	0
DACS Broadcast Service http://www.dacs.dtic.mil/forms/broadcastform.shtml	5	4	3	2	1	0
DACS Technical Reports http://www.dacs.dtic.mil/techs/tr.shtml	5	4	3	2	1	0
DACS Topic Areas (21 topical sections on homepage) http://www.dacs.dtic.mil/index.shtml	5	4	3	2	1	0
DACS Website (general opinion of all areas) http://www.dacs.dtic.mil/	5	4	3	2	1	0
Software Engineering Bibliographic Database (SEBD) http://www.dacs.dtic.mil/databases/sebd.shtml	5	4	3	2	1	0
Software Lifecycle Empirical Database (SLED) http://www.dacs.dtic.mil/databases/sled.html	5	4	3	2	1	0
Software Tech News (this newsletter) http://www.dacs.dtic.mil/awareness/newsletters/listing.shtml	5	4	3	2	1	0
Special Study or Technical Area Task http://www.dacs.dtic.mil/about/services/special.html	5	4	3	2	1	0
Technical Inquiry Service E-mail: webmaster@dacs.dtic.mil	5	4	3	2	1	0

10. **The DACS would like to help you.**

    Please check the box if you would like to be contacted to discuss the consulting services available from the DACS.
    ❏ Yes I'd like to more about your Special Studies (Technical Area Tasks)
    http://www.dacs.dtic.mil/about/services/special.html

    Please check the box if you would like to be the DoD DACS Broadcast Service.
    ❏ Yes I'd like to be registered for your Broadcast Service
    http://www.dacs.dtic.mil/forms/broadcastform.shtml

Fold here and tape. No staples please.

The first 100 people to return this survey will receive a computer shaped squeeze toy.
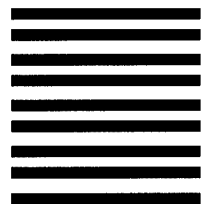
Just as in building construction, the scope and range of architectural services can be all encompassing or limited. The project may be a kitchen remodeling or a corporate headquarters; a total software reengineering of a behemoth corporation or a lowly website. In either case, though, the architect strives to do more than merely fulfill a catalogue of client requirements. It is hoped that the structure will attain what noted (building) architect Christopher Alexander calls "the quality without a name." That is, a structure that is ineffably greater than a sum of its parts, more than mere "function."

This is the ancient marriage of the aesthetic and the practical that lies at the heart of architecture. It is hoped that it will flower and thrive in software where artistry is as important as it is in buildings. It has been quite rare in the software industry to date but, understandably, how can great aesthetic design be achieved with a chancy, design-as-you-build technique? Besides, as we all can

imagine, it is simply impossible to be artistic in a crisis.

The Worldwide Institute of Software Architects (WISA), is a non-profit organization dedicated to the establishment of the profession. For more information on the profession of Software Architecture see the author contact information.

About the Authors

Marc Sewell is the President of the Worldwide Institute of Software Architects, which opened on September 1, 1998. He has been Chief Architect of IBM, VP of Information Technology for Morgan Stanley, and is currently an independent software architect.

Laura Sewell has written for the Atlanta Journal and Constitution and The Washington Post. She is the author of the WWISA website, and also works as a Disability and Rehabilitation Consultant in the insurance industry.

Author Contact Information

Laura and Marc Sewell

The Worldwide Institute of
Software Architects
North Cobb Parkway
Suite 109-211
Kennesaw, GA 30152
(404) 786-WISA (9472)

wwisa@wwisa.org

http://www.wwisa.org

Software Tech News on the World Wide Web
This newsletter in its entirety and past newsletters
with such topics as Risk Management, Rapid Application
Development, and Software Measurement are available
on the DACS Website at:
http://www.dacs.dtic.mil/awareness/newsletters/listing.shtml

Other Software Architecture Web Resources

DoD DACS Software Architecture Topic Area -
http://www.dacs.dtic.mil/

A Testbed for Analyzing Architecture Description Languages (ADL) -
http://source.asset.com/stars/lm-tds/Papers/arch/

Cetus Links - Architecture & Design - http://www.cetus-links.org/
top_architecture_design.html

Software Architecture Technology Guide -
http://www-ast.tds-gn.lmco.com/arch/guide.html

STARS Software Architecture Papers -
http://source.asset.com/stars/darpa/Papers/ArchPapers.html

Software Engineering Institute (SEI) Software Architecture Definitions -
http://www.sei.cmu.edu/architecture/definitions.html

DoD Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Return Service Requested