# Experiences Applying a Practical Architectural Method[*]

David E. Emery[1], Richard F. Hilliard II[2] and Timothy B. Rice[2]

[1] Hughes Aircraft of Canada, CAATS Development Facility, #200, 13571 Commerce Parkway Richmond, BC V6V 2J3 Canada
[2] The MITRE Corporation, Mail Stop B155, Bedford, MA 01730 USA

**Abstract.** Software architecture has come to be recognized as a discipline distinct from software design. Over the past five years, we have been developing and testing a practical software architecture method at the MITRE Software Center. The method begins with an initial statement of system goals, the purchaser's vision for the system, and *needs*, an abstraction of the system's requirements. Multiple *views* of the system are then developed, to address specific architectural concerns. Each view is defined in terms of *components*, *connections* and *constraints* and validated against the needs. This paper briefly introduces the method and describes our experiences with its "alpha" and "beta" applications to two U.S. Army management information systems.

## 1 Introduction

Software architecture is becoming recognized as a discipline distinct from software design [1]. There are several reasons for the emergence of software architecture, including: the realization that software design does not always address critical system needs; the emergence of "product lines" where the generic basis of the product line must be separated from the specifics of any one instance/delivered system [2]; and research into software engineering as a discipline which has lead many people to compare software with other established disciplines [3]. It has been the authors' experience that even before the recognition of the notion of "software architecture" in the research community, "good" systems often exhibit an architectural consistency (and conversely "bad" systems often lack this consistency.)

The Ada community has led in the construction of well-architected systems and the consideration of software architecture for construction large systems (see for example, CCPDS-R [4], [5], CAATS [6], and Tyndall RCS [7]).

This paper provides an overview of a software architectural method developed at the MITRE Corporation, and provides experiences from the first two applications of the method. Our method has these characteristics:

[*] In *Proceedings Ada Europe '96 Conference on Reliable Software Technology, Montreux, Switzerland, June 4–10, 1996*, *Lecture Notes in Computer Science*, volume 1088, Springer-Verlag. Awarded "Best Paper of Conference".

- It includes the ‘acquirer’ and end users, as well as the system developer, as participants in architectural activities.
- It can be applied to develop an architecture for a new system, or to capture the architecture of an existing system.
- It does not prescribe any particular design framework, computational model or fixed set of views; thus it accommodates varying needs and interests for a wide range of classes of systems.
- It supports traceability of architectural products back to both formal requirements and less formal goals and visions.
- The method is rooted in a line of architectural thought that emphasizes “context” as opposed to “form.”

The remainder of the paper consists of a short history of the method to introduce our notion of “architecture” and define some client goals for an architecture. We briefly summarize the method, which has been detailed elsewhere [8], [9], [10]. Following that we discuss our experiences on the first two major applications of the method, and provide observations from this experience. We conclude with a set of lessons learned on architecture and discuss on-going and future work.

## 2 “What is a software architecture, and why do I want one?”

Our ideas about architecture have been formulated over the past five years. At the inception of ARPA’s Domain-Specific Software Architecture program, one of us [11] examined the limitations of model-based approaches for “large” domains like Command and Control. This led us away from domain-specific approaches to considering the problems of general-purpose architecting techniques.

Many of these ideas have crystallized into the current method in support of our work for the U.S. Army Program Executive Office for Standard Army Management Information Systems (PEO STAMIS), and the Sustaining Base Information Services (SBIS) program in particular. MITRE was tasked to define a notion of “software architecture” for information systems and to apply this notion to several Army projects. Specifically, this tasking came about when we were asked, “What is a software architecture, and why do I want one?”

### 2.1 Our Definition of Architecture

Although it seems that everyone is using the term “architecture” these days, there aren’t many helpful definitions of the term. Our definition has these aspects:

- An *architecture* is the highest-level conception of a system in its environment.
- That conception includes the structure and behavior of the *whole* system in that environment, to show how the system will meet its requirements.

- A consequence of this definition is that every system has an architecture—whether or not it is written down.
- Of course, we are most interested in *articulated* architectures—those which are documented or otherwise modeled in some form which can be communicated to others.

## 2.2 Role of Architecture

Our definition has a number of differences from those of others which become evident as we discuss the role of architecture in system development.

An architecture is not just a design; in fact, an architecture might result in one or more designs. It is a framework by which multiple software developers produce consistent and integrated components over the life cycle, such that: each component is part of the integral system; and, the user is unable to distinguish between components developed by different developers. As a framework it identifies tools, methods, and facilities needed to develop an individual system that conforms with the architecture.

But an architecture has other "audiences" than just developers—it serves as basis for analysis and decision-making throughout the life cycle. In addition to designers, an architecture must be usable by end users, acquirers, the system's owner and operator, etc. It therefore should be able to support technical, cost and programmatic decisions.

A "good" architecture shows how to build a system to meet its users requirements and the often intangible "needs" of these other audiences. It does this by being a decision-making tool. By formalizing the ad hoc and implicit decision making of systems engineering, it clearly identifies decisions such as technology choices, allocations of function or performance, and the selection of APIs and other interfaces.

Once identified, decisions fall into three categories, obligations, commitments, open issues. Following Burns and Lister [12] (who were actually writing about design), our method distinguishes *commitments*, *obligations* and *open issues*.

> a progression of increasingly specific commitments ... properties of the system design which [detailed] designers are not at liberty to change. Those aspects of a design to which no commitment is made ... are the subject of obligations that lower levels of a design must address. The process of refining obligations into commitments is often subject to constraints imposed primarily by the execution environment.

The architecture resolves some of these decisions by making commitments. Commitments are captured as architectural rules imposed upon the designer. E.g., on SBIS the architecture resolved that there would be a single enterprise data model. In other cases, the decision is obligated to the designer to make; e.g., each application will define its data view in terms of the SBIS data model. When it is unclear which of these options may be taken, the decision remains an open issue for the architecture, subject to further investigation.

## 3 An Overview of the Method

In this section we walk through the steps of a candidate process for architecture development, the principles governing it, and how architectures are represented.

1. Understand program
   - Needs Analysis
   - Program cost and schedule
   - Products: needs, goals, vision statements
2. Select views
3. Analyze each view
   - Products: a "blueprint" of each view with associated decisions, rules and open issues
4. Integrate views (for consistency)
5. Trace views to needs (for completeness)
6. Iterate activities 3, 4 and 5
   - Until no more unresolved questions
   - If there are numerous unresolved issues, revisit step 2
7. Validate
   - Against formal requirements
   - Conformance of design and implementation (as built) to architecture (as specified)

**Table 1.** An Architectural Process

As in many branches of engineering, our method begins with understanding the problem and its constraints including the requirements, program cost and schedule demands; i.e., the concerns of the various stakeholders. Our understanding of these is distilled into three products, developed with the client, and ideally "owned" by the client, which capture both the specific requirements for the system, and also the general direction and intent of the client.

### 3.1 Goals and Vision: The Client's Responsibility

It is important that the client be an active participant in the architectural process. To this end, we ask the client to prepare two architectural products, a "goals statement" and a "vision statement." The goals reflect the client's evaluation criteria—how the client will evaluate whether the system is a success. These goals include meeting the formal needs, as well as preferences and desires which do not occur as requirements. The vision indicates the client's long-term direction for the system. This includes both the current system under development, and also provides a view of its evolution and eventual replacement.

### 3.2 Needs: An abstraction of requirements

For the kinds of systems we typically deal with, there are one or more requirements specification documents, often running into hundreds or even thousands of pages. Even when well-written, these formal requirements documents have proven to be too difficult to use effectively to derive and analyze system architectures. We analyze these requirements into a set of *needs*, which capture the architecturally-relevant requirements, without unnecessary detail.

Needs are developed using classical requirements analysis techniques [13], [14]:

1. Abstract system requirements to system needs
2. Analyze needs for architectural relevance
3. Convert to architecture need
4. Review and verify coverage
5. Document and gain buy-in from program

The step which classical techniques do not provide much guidance for is determining the architectural relevance of a requirement. This requires experienced architects.

### 3.3 View Selection

With the needs, goals and vision in place, we begin to architect the system. We document an architecture as a model, expressed as a number of *views*. The purpose of multiple views is to manage complexity and separate concerns—a single model would be too complex to capture all aspects of a system's architecture. Instead, each view focuses on one or more key concerns of the system. A key principle of views is completeness: for each view, that view is intended to cover the whole system from the selected vantage point. Another principle is that views are not mutually exclusive; rather, a cohesive understanding of the system is gained through integration of the interrelated views (see Traceability, below).

Unlike other techniques which pre-select a set of viewpoints [2], we do not. The selection of appropriate views is a critical early activity of the architect, driven by the critical concerns of the program.

### 3.4 Components, Connections and Constraints

Each view is modeled individually using three primitives: components, connections and constraints.

*Components* represent the major structural elements of a view. E.g., functions in a functional view, processors and data stores in a physical view.

*Connections* represent the major relations between components. These include "run-time" relationships like control or data flow as well as other dependencies.

*Constraints* represent laws the system must observe; constraints apply to components and connections. Constraints include performance and non-functional requirements, style and protocol rules, and laws of nature which constrain resources.

The modeling of views is defined by the "C3 meta model," a grammar which states the permissible syntactic and semantic relationships between these three kinds of modeling primitives [15].

### 3.5 Traceability

There are several levels of traceability defined by our method.

Once a view is syntactically checked, using the meta model, it may be integrated with others (step 4). View integration is a necessary step in a method like ours because of the nature of views; integration is a meaningful semantic act, not a syntactic projection or transformation which could be automated. The technique we use for view integration is based on Ross' "model tie process" from Structured Analysis (SADT) [16]. Using the tie process, we systematically cross-reference related entities from distinct views, thereby addressing what Shaw calls the "multiple views problem" [17].

Once the views have been integrated, the (resulting) architectural definition is reconciled against the original needs analysis (step 5). In this step we ascertain that the architecture *covers* the stated needs, goal and vision. It is not necessary that each statement of goals, vision, and needs be individually satisfied by the architecture definition. However, it is necessary that all questions or issues raised in goals, vision or needs are either identified and resolved by the architecture, or that the architecture definition explicitly delegates their resolution to the design.

Later in the development, there is a further level of traceability, that of establishing conformance of the design and/or implementation with the architecture (step 7).

## 4 The SBIS Architecture

### 4.1 The SBIS Program

*Mission.* Like many other enterprises, the Army needs to downsize its operations over the next decade, while creating the capability to respond with greater flexibility to changing mission requirements.

The objective of the Sustaining Base Information Services (SBIS) program is to acquire and implement a Government-owned and Government-operated (GO/GO) Open Systems Environment (OSE) infrastructure and to transition all active Army component automated information systems to that OSE by 2002. Such an infrastructure is intended to provide a common user interface and services to help eliminate proprietary, "stovepiped" Army management information system applications. It is intended to provide greater flexibility for access to information regardless of operational environments (ranging from Army installations to forward-deployed elements from those installations), database management systems, networks or physical location of the information, in accordance with the Army Enterprise Strategy.

The SBIS OSE infrastructure is based on current and emerging OSE Standards. Adherence to these standards as they continue to evolve ensures that the Army's information services remain independent of technology and vendor, while supporting a selection of vendor products. The design allows an ongoing infusion of new technology and emphasizes procured rather than developed items.

## 4.2 SBIS Needs, Goals and Vision

The SBIS goals and vision statements were formulated by the client working in concert with the SBIS system engineer and the architecture team. The SBIS goals statement is shown in table 2. The SBIS vision statement is shown in table 3.

- Design SBIS to evolve
  - Use Commercial off-the-shelf (COTS) products vs. "built to specification" items whenever possible
  - Assume that user needs will change over time
- Design SBIS for change
  - COTS products used in SBIS will change as the market evolves
  - Hardware will change, becoming more heterogeneous with time
  - Use of standards will allow SBIS to adapt to changes in hardware and COTS software

**Table 2.** SBIS Statement of Goals

*SBIS Needs.* A needs database was constructed early in the architectural effort and has been maintained throughout the project. It currently captures about 140 needs—in contrast to hundreds of formal requirements reflected in the Statement of Work, and "capstone" requirements document. In addition, each of the tens of SBIS applications to be supported has its own Functional Description document, typically containing numerous user requirements. Since the predominance of these requirements are functional specifics within the overall concept of Army management information systems, they do not have a direct impact on the architecture, and thus are bit reflected as needs.

Each need is recorded with a unique identifier, a statement of the need, cross-references to the formal requirements from which it is derived, and other information. We currently maintain the needs database as a spreadsheet.

1. SBIS will support day-to-day information processing needs of an Army installation and the units it supports, even if these units are deployed.
2. SBIS will support low-skilled end-users by providing integrated computer-based training, extensive help capabilities, and a consistent end-user interface.
3. SBIS will support a set of evolving and expanding applications, including legacy and new applications.
4. SBIS will support consistent development of applications by multiple organizations.
5. SBIS performance will increase over time, reflecting increases in application performance requirements.
6. Scaling and reliability will be tailorable based on needs of installations and their supported units.
7. SBIS reliability will increase over time.
8. SBIS will employ 'open concepts':
   - To take advantage of emerging technologies as they become stable,
   - To ensure overall affordability and 'value for the buck,'
   - To ensure that applications and data will be isolated from changes in basic computer hardware, network topology, user interface methods, and database implementations.

**Table 3.** SBIS Vision Statement

## 4.3 SBIS Views

For SBIS, we initially created four views. We have subsequently added a fifth view, to address security concerns. Each is described below.

*Application View.* As described by the SBIS vision, the system has strong needs for flexibility, portability, interoperability and forms of "openness." As SBIS evolves, new and legacy applications must coexist. The Application View is a "template" for applications new and old. For users, it defines a uniform "look and feel" for SBIS applications. For application designers, it establishes controlled interfaces and partitioning principles such that all applications may "plug and play" in SBIS, availing themselves of common services. For vendors, it establishes identifiable opportunities for new products and enhancements within a well-defined market. The Application View exhibits a 5-layered structure of components (from the top): Presentation – User Interface – Application Logic – Data Access – Data Storage. It is layered, in that each component only communicates with its direct neighbors through a single connection, manifest by a well-defined API. Each such pair of components stand in a client-server relationship. Applications are designed to be distributable across any of these layers, on a per-application basis, and governed by rules of the Distribution View (below).

*Data View.* This view is concerned with how SBIS data is structured, organized and related, and the manner in which applications may utilize this data.

The Data View addresses: Army-wide requirements on data, such as the mandated use of the DoD Data Model; that numerous legacy applications have implicit or explicit data models of their own; that there is a need for consistent *ad hoc* query access capabilities; and the demands for distribution and replication of data.

It is not a data model—an SBIS-wide enterprise data model is under development as a part of the design process. Rather, the major components of the Data View are the many SBIS data models (from enterprise, to application-specific, to legacy, to physical models). Connections between these models represent the types of transactions the system will need to implement. Constraints in the Data View reflect the logical relations between these models (that the SBIS-wide model must be a subset of the DoD model, that each application data model must be a projection of the DoD model, etc.)

*Distribution View.* The Distribution View is concerned with the allocation of applications and data to actual computing elements (including processors, data bases, networks and end-user workstations) and geographic locales. In the case of SBIS, most distributed processing is accomplished via existing local area networks. However, at some SBIS sites the only available connectivity is via a dial-up phone line, substantially reducing the bandwidth and reliability of the connection. More importantly, the Army's approach to conducting operations has changed. Today, a unit being deployed expects to take "a piece of home" with it in the form of connections back to its supporting installation to provide payroll, finance, inventory, and other support. An installation may suddenly find that some of its elements have been moved from down the street to Kuwait, Somalia or Haiti. Thus, the Distribution View must handle both traditional LAN-based connections and *ad hoc* connections based on commercial broadband carriers, satellite communications, dial-up telephone lines, or even tactical radios. Rather than attempting to specify a single distribution paradigm to cover this range of situations and application and data distribution needs, the Distribution View specifies a set of rules for the designer to use in making decisions for each application and potential distribution case.

*Security View.* SBIS has strict requirements for handling medical and financial data, as well as interfaces to command and control systems. The Security View establishes rules for the protection of sensitive data and other resources. The components of the Security View comprise the subjects, objects and information domains of a security model [18]. Connections represent the forms of permissible access. Constraints include the security policies governing subjects' behaviors and access to protected data and resources.

*Development/Maintenance View.* It could reasonably be said that all of the views described above are "development" views, since a critical audience for each one are the developers. However, multiple organizations are involved in developing and maintaining the system which includes legacy and COTS elements. A unified development view is needed to integrate these disparate elements in a consistent fashion. This view describes the rules for documentation and the management of software artifacts in a multi-organization setting, through a set of building codes and construction rules.

### 4.4 Results and Accomplishments

As requested by the Government client, the original documentation of the SBIS architecture took the form of a set of annotated briefing charts. In this format, the architecture definition has been central to making the notion of a Government-owned/Government-operated infrastructure concrete throughout the program. Getting the client involved in the setting of goals and vision has established his direct involvement in the architectural process.

The architecture definition has been the basis for reconciling conflicting requirements between user proponents and for evaluating design alternatives such as component selection. Make vs. buy (e.g., COTS vs. developed Ada code) options are analyzed in the architectural context. The SBIS context (as expressed in the goals) also led away from an early communication-intensive design proposal. In the same way, the architecture is used to communicate a single integration framework for these disparate components: new code, COTS and legacy applications. The Data Access layer, for example, defines a single, common procedural interface to all data sources (specified with Ada/SQL) whether new, COTS or legacy. The approach is now being extended to an enterprise-wide “Interface Control Document Policy” for standard Army MIS applications to specify their interfaces in a common fashion.

At all stages of the architecture activity, and now in the development phase, we have used the architecture definition for life cycle costing studies, to identify “driving requirements” and to estimate maintenance and support costs.

## 5 The AMIS Architecture

### 5.1 The AMIS Program

MITRE was asked to support a government “tiger team” in a quick evaluation of another military information system (AMIS) and the state of the program over a 2–3 week period. The AMIS is intended to provide automated management information support to the U.S. military’s reserve components, much as SBIS is to provide to similar support to the active Army. We used our architectural method to develop short goals and vision statements for AMIS. The goals and vision statements were developed by team members with experience in the domain, and were subsequently validated with the project management. We then produced a list of key needs. From this, we developed a strawman architecture to understand the issues. The primary result of this 1-week effort was a list of architectural issues that AMIS needed to solve.

We used this list to discuss issues with the contractor, and produced some alternatives. Our method allowed us to identify and isolate some questionable requirements that (when expressed as needs in our method) drove parts of the system architecture and contributed substantially to the overall system cost. Using our goals and vision statements, we triggered a reassessment of some of these needs. By adjusting the security requirements, we showed that the system could save millions of dollars, through accomplishing the overall goals using alternative facilities (both automated and manual).

## 5.2 Architecture Validation

The tiger team led to a longer-term (3 month) investigation to validate their preliminary findings. For this effort, we systematically applied our method, albeit in a compressed time frame. Because of the strong similarities to SBIS, at each stage we used architectural products from that program as “first drafts” for AMIS.

A Customer Focus Team (CFT)—representing the AMIS users—was formed. We worked with them to affirm the earlier strawman goals and vision statements. The CFT used the SBIS needs database to begin development of a full set of AMIS needs. Many could be used as-is or easily restated, others were deleted, a number were added. The needs were then prioritized by the CFT as input to the architecture.

In parallel with the needs analysis, we started with the extant SBIS views: Application, Data, Distribution, and Development/Maintenance. Subsequently we added a Security View (the recognition of the need for this vantage point was later fed back to SBIS).

Using these draft views and the AMIS needs as they became stable, we were able to quickly sketch a new architecture for AMIS. The Application View was a generalization of what was already being done; the Data View was quite consistent with the approach to data engineering already being taken on the program; the Distribution View reflected opportunities that simply were not present when the original effort was begun; and the Development/Maintenance View focused on program management level concerns for maximising use of COTS and reuse of existing Ada code. The Security view allowed us to capture and compare alternate approaches to system security. In particular, we produced several architectural alternatives, which showed that the right choice of security mechanisms could achieve substantial cost reductions.

## 5.3 Experiences

The major goal for a ‘tiger team’ is to identify critical issues. Our architectural framework gave us a set of tools for analyzing a program, by producing the initial architectural products. With these products, we did a strawman architecture, which identified where the architecture was affected by specific system needs. Thus rapidly identified a set of issues that had significant architectural impact. By adjusting the needs, we produced alternate architectures, which were then used in cost analysis.

The method worked very well to focus the team. Through the Needs production, we gained a reasonable familiarity with the system requirements. Our work on Goals and Vision allowed us to identify the truly important parts of the system, and were critical in identifying Needs that could be changed without affecting the overall system function. Our strawman software architectures (several versions, based on varying needs) was sufficiently detailed to allow economic analysts to estimate system costs. Thus the architectural approach produced alternative system designs and associated costs, using a consistent method.

## 6 Lessons Learned with Architectures

### 6.1 Our Architectural Method

We have now used our method both for architecture synthesis and analysis. In addition to the information systems described above, we are using it for a portable (laptop) command and control system at present. In the analysis mode, we have used it to identify weaknesses and risks in proposed designs. In the synthesis mode, it allows us to rapidly isolate key decisions and trade-offs.

The method and the resulting products seem accessible to key decision makers: needs, as we have defined them, are easily understandable by stakeholders—perhaps more naturally than formal requirements. The factoring of a complex system into one or more views also helps to communicate with key personnel.

We have found the principles governing views to be useful in getting started and bounding the architecture problem. By applying these principles and the vocabulary of components, connections and constraints uniformly, we have attained a level of rigor wherein the notation itself rules out possible solutions. Of course, this means not anyone can simply start "architecting"—individuals must be trained in the principles, and our method seems to require a high degree of expertise on the part of its principals.

The architecture descriptions we delivered for AMIS and SBIS were in the format of annotated briefings—as required by our clients. We believe we have reached the limits of usability for this format, at least for large systems. For our on-going SBIS work, the architecture specification has been converted to a textual form. We are considering HTML for other efforts.

### 6.2 Evolution of the Discipline of Software Architecture

*Principles of the Method.* As we have evolved our method, we have extracted the following principles we believe to be useful to any architectural approach and the basis of our success:

- Architecture documentation must be understandable by diverse audiences, including users and clients, not just developers
- There's a need for a general-purpose "blueprinting" technique
  - Applicable to single systems, product lines, domains
  - Independent of design method (e.g. object-oriented)
  - Independent of computational model
- Always separate concerns
  - Requirements $\neq$ Architecture $\neq$ Design

*Comparison to Other Work.* These principles have led us to a conception of architecture, much closer to its traditional (building) sense, than is currently employed in software. Most current work focuses on the internal structure of systems, making architecture a part of design, rather than a contextual activity, situating the system in its environment of users and other stakeholders.

We believe we have made two important technical contributions in our work. First, we capture constraints as first-class entities of an architecture, on par with components and connections. Second, we use these three primitives as a unifying notation across different views—whereas others have selected view-particular constructs (whether functions, objects, etc.).

## 7 Conclusions

We have developed an architectural method that emphasizes placing a system into its context, and captures this context through the goals and vision statements. We abstract the requirements into a set of needs, which are then used to develop the subsequent architectural products, a set of blueprints, or views, of the system. Each view is expressed in terms of components, connections and constraints, governed by a meta-model. A key ingredient of our approach is traceability between views, which we believe we have the conceptual mechanisms to handle.

During the development of the method thus far, we have used only text processing and graphical presentation tools for architecture specification. This has been intentional—we did not want to force what we were doing into existing tools without knowing our real needs for automation. Now, we believe we have enough experience with the method to take the next step; that is the subject of future work.

*Acknowledgments.* The applications described here were under the sponsorship of the U.S. Army Information Systems Engineering Command (ISEC). Our development of the practical architecture method is sponsored by the MITRE Software Center Core Technology program. Other MITRE contributors to the definition and application of the method include: R. Baldwin, C. Byrnes, R. Harris, K. Heideman, J. Hustad, M. Kinnebrew, J. Knisley, J. Moore, T. Nixon, R. Ringdahl, S. Schwarm, and D. Waxman.

## References

1. D. Garlan, editor. *Proceedings of the First International Workshop on Architecture for Software Systems*, Seattle, WA, April 24–25 1995. Published as CMU–CS–TR–95–151.
2. Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 28(11):42–50, November 1995.
3. Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Sofware Engineering Notes*, 17(4), October 1992.

4. Walker E. Royce. Reliable, reusable Ada components for constructing large, distributed multi-task networks: Network Architecture Services (NAS). In *TRI-Ada Proceedings*, Pittsburgh, October 1989.
5. Walker E. Royce. TRW's Ada process model for incremental development of large software systems. In *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 26–30 1990.
6. Philippe B. Kruchten and C. J. Thompson. An object-oriented, distributed architecture for large scale Ada systems. In *Proceedings TRI–Ada '94*, Baltimore, MD, November 1994.
7. Dan DeJohn. The Tyndall Range Control System: bringing network computing to C2 systems. In Charles B. Engle, editor, *TRI–Ada '94 Proceedings*, pages 474–485. ACM, 1994.
8. David E. Emery and Richard F. Hilliard. "Architecture," methods and open issues. In D. Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995. Published as CMU–CS–TR–95–151.
9. Richard F. Hilliard, Timothy B. Rice, and Stephen C. Schwarm. The architectural metaphor as a foundation for systems engineering. In *Proceedings of Sixth Annual International Symposium of the International Council on Systems Engineering*, 1996.
10. Richard F. Hilliard and David E. Emery. Patterns : Design :: Blueprints : Architecture, 1996. Work in progress.
11. Richard F. Hilliard. Comments on kogut and clements. email.
12. A. Burns and M. Lister. A framework for building dependable systems. *The Computer Journal*, 34(2), 1991.
13. Michael S. Deutsch and Ronald R. Willis. *Software Quality Engineering*. Prentice-Hall, 1988.
14. Douglas T. Ross and Kenneth E. Schoman. Structured Analysis for requirements definition. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.
15. Richard F. Hilliard. Representing software systems architectures or, components, connections and (why not?) first-class constraints and views. In *Proceedings 2nd International Workshop on the Architecture of Software Systems*, 1996. to appear.
16. Douglas T. Ross. Removing the limitations of natural language (with the principles behind the RSA language). In Herbert Freeman and Philip M. Lewis II, editors, *Software engineering: proceedings of the Software Engineering Workshop held in Albany, Troy, and Schenectady, New York, from May 30–June 1, 1979.* Academic Press, New York, 1980.
17. Mary Shaw. Comparing architectural design styles. *IEEE Software*, 28(11):27–14, November 1995.
18. DoD. Trusted computer systems evaluation criteria. Technical Report DoD 5000.28–STD, Department of Defense, 1985.

This article was processed using the LaTeX macro package with LLNCS style