CDRL: B008
29 January 1994

UNISYS

Software Architecture Seminar Report
Central Archive for Reusable Defense Software (CARDS)

Informal Technical Data

CARDS

Central Archive for Reusable Defense Software

STARS-VC-B002/001/00
29 January 1994

CDRL: B008
29 January 1994

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Software Architecture Seminar Report
Central Archive for Reusable Defense Software
(CARDS)

STARS-VC-B008/001/00
29 January 1994

Informal Technical Data

CONTRACT NO. F19628-93-C-0130
Line Item 0002AB

Prepared for:

Electronic Systems Center
Air Force Material Command, USAF
Hanscom AFB, MA 01731-2816

Prepared By:

Azimuth Incorporated
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston VA 22091

Distribution Statement "A"
per DoD Directive 5230.24
Approved for public release, distribution is unlimited

CDRL: B008
29 January 1994

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Software Architecture Seminar Report
Central Archive for Reusable Defense Software
(CARDS)

STARS-VC-B008/001/00

Informal Technical Data
29 January 1994

CONTRACT NO. F19628-93-C-0130
Line Item 0002AB

Prepared for:

Electronic Systems Center
Air Force Material Command, USAF
Hanscom AFB, MA 01731-2816

Prepared By:

Azimuth Incorporated
under contract to
Unisys Corporstion
12010 Sunrise Valley Drive
Reston VA 22091

CDRL: B008
29 January 1994

Data ID: STARS-VC-B008/001/00

Distribution Statement "A"
per DoD Directive 5230.24
Approved for public release, distribution is unlimited

Developed by: Azimuth, Incorporated under contract to
Unisys Corporation

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U. S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130 the STARS program is supported by the military services with the U. S. Air Force as the executive contracting agent.

This document retains Contractor indemnification to the Government regarding copyrights pursuant to the above referenced STARS contract. The Government disclaims all responsibility against liability, including costs and expenses for violation of property rights, or copyrights arising out of the creation or use of this document.

In addition, the Government, Unisys, and its subcontractors disclaim all warranties with regard to this document, including all implied warranties of merchantability and fitness, and in no event shall the Government, Unisys, or its subcontractor(s) be liable for any special, indirect, or consequential damages or any damages whatsoever resulting from the loss of use, data, or profits, whether in action of the contract, negligence, or other tortious action, arising in connection with the use or performance of his document.

CDRL: B008
29 January 1994

INFORMAL TECHNICAL DOCUMENT
Software Architecture Seminar Report
Central Archive for Reusable Defense Software
(CARDS)

Principal editors:

H. Jeff Facemire

Aleisa Petracca

Stephen Riesbeck

Approvals:

System Architect *Kurt Wallnau*

Program Manager *Lorraine Martin*

(signatures on File)

CDRL: B008
29 January 1994

ABSTRACT

In order to increase awareness, explore current research into software architectures as a means of implementing software reuse, and examine current practices and issues involving architectures, the Central Archive for Reusable Defense Software (CARDS) Program sponsored a Software Architecture Seminar and Workshop at West Virginia University's Concurrent Engineering Research Center (CERC) facility in Morgantown, West Virginia on November 16 and 17, 1993. The goals of the Seminar and Workshop were to understand the various meanings of software architecture, current research in the field of architecture, and current efforts in applying software architecture. This document provides highlights of the Seminar and Workshop.

This document contains an overview of the proceedings of the Architecture Seminar on Tuesday, November 16 and the Architecture Workshop on Wednesday, November 17. This includes issues discussed, questions and answers, working group discussions, and references. This document also contains presentation slides from the Seminar, the Seminar panel discussion, and the Workshop.

CDRL: B008
29 January 1994

PREFACE

Just as the CARDS Software Architecture Seminar and Workshop could not have been a success without the efforts of many individuals, this document also is based on the efforts of many contributing authors. Thanks to primary authors Kurt Wallnau, Paul Kogut, Charlie Snyder, and Kerri Haines, of Unisys Corporation, for their efforts, work, and research, and all CARDS Program members who contributed to the Seminar and Workshop.

The CARDS Program also thanks all participants, who were able to make the Seminar and Workshop enjoyable and enlightening.

Comments on this document are welcomed and encouraged.

TABLE OF CONTENTS

LIST OF TABLES

1 Introduction

In an effort to improve software quality and cost effectiveness, the Department of Defense (DoD) is actively endorsing software reuse, the process of implementing new systems by using existing software products and information. As noted in the DoD Software Reuse Initiative Vision and Strategy, DoD aims:

> *[t]o drive the DoD software community from its current "re-invent the software" cycle to a process-driven, domain-specific, architecture-centric, library-assisted way of constructing software.*

A key element of the Vision and Strategy, architecture-centric reuse involves defining reuse-oriented flexible architectures for DoD domains which are well supported by industry and the R&D community, then spurring investment in creation of generic software components and tooling which facilitates development of systems complying with approved architectures. The creation of generic components must be independent of development of fieldable production systems. One of the principal challenges of reuse is to develop processes and standards that can facilitate development of a convention that enables effective sharing of components.

In order to increase awareness, explore current research into software architectures as a means of implementing software reuse, and examine current practices and issues involving architectures, the Central Archive for Reusable Defense Software (CARDS) Program sponsored a Software Architecture Seminar and Workshop at West Virginia University's Concurrent Engineering Research Center (CERC) facility in Morgantown, West Virginia on November 16 and 17, 1993. The goals of the Seminar and Workshop were to understand the various meanings of software architecture, current research in the field of architecture, and current efforts in applying software architecture. This document provides highlights of the Seminar and Workshop.

1.1 The CARDS Program

The Central Archive for Reusable Defense Software (CARDS) Program is a concerted DoD effort to transition advances in the techniques and technologies of domain-specific software reuse into mainstream DoD software procurements. This technology transition effort combines a concrete demonstration project to illustrate the potential of domain-specific reuse -- in this case for the domain of Command Centers -- with a broad-scale attack on the cultural and contractual inhibitors to software reuse. The CARDS Program goals are to:

- Produce, document, and propagate techniques to enable domain-specific reuse throughout the DoD
- Develop and operate a domain-specific library system and necessary tools
- Develop a Franchise Plan which provides a blueprint for institutionalizing domain-specific, library-centered reuse throughout the DoD
- Implement the Franchise Plan with users and provide a tailored set of services to support reuse

1.2 Document Organization

This document is organized into five chapters and two appendices.

Chapter One, *Introduction*, provides a general introduction to the document.

Chapter Two, *Software Architecture Seminar*, gives a summary of the Seminar and Seminar Panel Discussion, along with highlights of issues discussed.

Chapter Three, *Software Architecture Workshop*, provides a summary of the Workshop and issues surrounding the Workshop presentations.

Chapter Four, *Architecture Seminar and Workshop Summary*, contains a summary of the two day event based upon evaluanon forms which were distributed to all participants.

Chapter Five, *Architecture Seminar and Workshop Presentation Slides*, contains over 500 presentation slides from the Seminar, Panel Discussion, and Workshop.

Appendix A is a list of all participants, along with contact information.

Appendix B is a bibliography of sources used for development of the Seminar, and suggested sources for additional information.

2 Software Architecture Seminar

This Chapter outlines the proceedings and key points of the CARDS Software Architecture Seminar conducted on November 16, 1993. The Seminar consisted of formal presentations followed by a panel discussion. Accompanying presentation slides and speaker notes for the Seminar and the Panel Discussion are located in Chapter Five; page numbers for the slides are noted in text.

2.1 Seminar Proceedings Summary

The Architecture Seminar was divided into five sessions:

- Session I *Why Architectures?*
- Session II *Senses of Architecture: Building the Category*
- Session III *Software Architecture and Reuse*
- Session IV *Architecture-Based Reuse Tools*
- Session V *CARDS Approach to Reuse and Software Architecture*

Session I (pages 35-58) of the Seminar focused on why architectures are needed, why architectures are becoming more evident, and definitions of architecture. A major topic of discussion in Session I was the various definitions of architecture and style; the notion of architecture often depends on the perspective of the individual or organization.

Session II (pages 59-140) built upon the definition of architecture discussion in Session I, drawing parallels to perspectives in manufacturing and engineering. Session II then contained overviews of software architecture from a scientific foundation, engineering application, and considerations in practice.

Session III (pages 141-216) examined architecture from a reuse standpoint, concentrating on architecture "defined," the science of architecture, trends in architecture for reuse, and architecture-based reuse systems.

Session IV (pages 217-274) involved a examination of specific architecture-based reuse tools.

Session V (pages 275-321) presented the CARDS approach to Domain Engineering activities as related to software architectures and reuse from scientific, engineering, and transition-to-practice views.

2.2 Seminar Proceedings Issues

Throughout the Seminar, many participants raised issues on Seminar topics which generated discussion. This section highlights some of these issues and includes some of the questions raised by participants.

2.2.1 Style

A significant topic of discussion during the Seminar was architectural style.

The question was raised: can we name styles of architecture (pages 87-98, 143-160)? It was offered that there are certain specializations and rules, but there are limited capabilities on how to apply these specializations and rules. There is a significant challenge in that there is no formal representation or formal basis to build systems. But, there are tools for use in the "real" world.

It seems we are still in a pre-paradigm stage regarding style. There may be a style emerging for real time systems but it is very immature; since it is difficult to get a good definition for architecture, it is difficult to get a clear style. There is still confusion on defining architectures, and what style actually is.

One participant's previous understanding of style was design patterns plus organizational structures plus the ensemble (system specific features), but now the notion of style implies globality. Another participant offered that a computational model (how the components communicate) is the style, and that the computational model is the prime distinguishing feature between architectural styles. Also, there are well known computational models.

With regard to the characteristics of an architecture, one participant stated that he'd like to apply a test to architecture and style: if one has an architecture to preserve behavioral attributes (such as security), where is this information captured? It was observed that some systems may have wonderful qualities but bad style. These questions must be considered: What elements of design have to be represented? Where does it stop? It was offered that architectural models should focus on an understanding of style and coherence.

Another participant noted that there is a larger issue still; everything has an architecture but architectures are viewed subjectively. However, there is objectivity regarding style: understandability.

The point was also made that with regard to emphasis on style, the emphasis must be on all elements. It was also pointed out that one should ensure that style captures operational principles; software designs often end up with style cluttering it up or getting in the way. Another observation was that functionality is the key; style alone is not enough.

2.2.2 Architectures Defined

With regard to the definition of architectures (pages 45-50, 85-98, 145-152), one participant noted that based on experience, architectures should be at a higher level of reuse. There needs to be a move away from expressing this as, for example, a compiler, so that architectures can move closer to DoD application areas and can be used as examples for better understanding by management level personnel. It was also noted that somewhere there should be data and process views for mature design areas, such as combat weapons systems. Another participant noted the importance of domain independence; we should think of things that will work in different systems.

In a discussion of work done by Don Batory regarding design methods and architectural style (pages 125-126), several participants made comments. Some felt that Batory's work is similar to others, but differs only in perspective. It is notable that Batory used a recursive way of putting modules together, with the only difference being data types. Another participant noted that Batory's method "feels" different, while another noted that Batory's work was somewhat domain dependent.

The point was made that Batory's work looks similar to other processes, but that he arrived at his results in a different manner. Batory didn't start with idioms; he performed a domain analysis and abstracted idioms. Through domain analysis and domain modeling, new idioms can be found and the form of architecture can be the same.

It was also questioned if language should be used to drive the system. A response was that form comes from the design method, and that language should be at the level of components and connectors. One participant felt that there was no difference, while another felt that the difference is only in perspective.

It is possible the difference between architecture and software/computer systems is that computer systems deal with codifying a wide range of business processes. When building a system to support these processes, there is a clash between pre-defined components and the process which you're trying to support. This calls for a close look at requirements.

In Session III, seven characteristics of software architecture were discussed (pages 147-150). One participant noted that it is easy to see the part in the whole, but how can one see the whole? Does seeing the part in the whole actually change the part? One reply was that if one can see the part, such as a subsystem, one doesn't necessarily need to see the whole, but can gain an understanding of the whole system.

2.2.3 CARDS Approach

In the discussion concerning the CARDS approach to reuse and architectures (pages 281-285, 301-308), one participant observed that Prieto-Diaz's idea of a faceted classification scheme usually results in 5 or 6 facets, while the CARDS approach involves more. CARDS chooses to show more relationships, and, having a model-based library, concentrate on representing a domain-specific model. Also, one participant noted that a knowledge based classification scheme can also involve a high cost to implement and maintain.

2.3 Seminar Panel Discussion Summary

The panel discussion included presentations from four participants, followed by a question and answer discussion. The four panel members were:

- Mr. T. F. "Skip" Saunders, Mitre Corporation
- Mr. Hans Polzer, Unisys Corporation

- Mr. Stan Levine, US Army Communications Electronics Command (CECOM)
- Capt Frederick Swartz, Training System Program Office, ASC/YTE

The panel discussion consisted of presentations from each panel member. Mr. Saunders presented views on architecture and reuse in terms of three points: goals, views, and trends (pages 324-371). Mr. Polzer's presentation (pages 372-383) concentrated on the economic factors surrounding architectures. Mr. Levine offered some case history examples and lessons learned on projects involving architectures (pages 384-423). Captain Swartz discussed the role of architectures or structural models in proposals (pages 424-435).

2.4 Seminar Panel Discussion Issues

2.4.1 Open Systems

One participant questioned the panel regarding open systems. The participant's customer had requested that architectures be re-defined to open systems, presenting difficulties in conflicting standards. The question was raised: are architectures and open systems the same?

With regard to open systems and architecture, issues such as compatibility and interoperation are often difficult; products are often built to different standards. However, these issues need to be considered from an architectural standpoint so that components will connect in a disciplined manner. This is starting to surface in the commercial sector. However, a problem in the Government arena is that the Government can not specify one single system; this could lead into contracting/legal difficulties. Therefore, the Government states the properties of a desired system, then leaves it up to the contractor to decide how to meet the requirements. The Government then evaluates the contractor's approach.

The solution also depends on one's definition of open system. A system doesn't necessarily have to follow a Government sanctioned standard. One approach is to follow an economic approach: what/how much financial resources are available and "is it for me" in relation to risk? Often open systems aren't really open; there are so many alternatives. "Open" sometimes means avoiding a large economic lock-in while still accomplishing what was wanted. Also, from the Government point of view, there may be times when a Government agency/customer can't afford an open system. It may be best to let the contractor decide.

2.4.2 Structural Modeling and Proposals

Several participants were interested in specifying certain architectures (referred to in this context as structural models) in Statements of Work (SOWs) and Requests For Proposal (RFPs) (pages 424-435).

At times, the Government may not want to limit the contractor by specifying a certain architecture; other times, the Government may be limited by policy assuring that bids are competitive. Also, architectures/structural models are still relatively new and not well defined.

Architectures/structural models can be in SOWs as long as a specific product is not specified. However, there need to be trained people who know the structural model and there must be no flaws in the structural model. Also, if the architecture/structural model is not specified, then no one may bid it.

In order to evaluate proposals, evaluatable criteria must be in the SOW/RFP. The criteria that are pushing the use of a certain architecture must be known. A track record that the architecture works will help. If there is no track record, one option is to let the contractor offer an architecture or structural model, remembering that the burden will still remain on the issuer/Government. It is important to know what attributes are desired.

2.4.3 The New Concept of Architecture

There was some debate as to whether architectures are a new concept, or have been used for some time. Often architectures are developed unplanned. While the development community seems to have been using architectures for a long time, current emphasis is on their formalization. Pieces of a system are better defined when this formalism is in place. It also appears that vendors are now able to dictate architectures used in their products.

3 Software Architecture Workshop

This Chapter outlines the proceedings and key points of the CARDS Software Architecture Workshop conducted on November 17, 1993. The Architecture Workshop began with presentations from leading Government and industry specialists on current efforts and research in software architecture. The participants then split into six working groups to continue discussion and examine issues in particular fields of interest. Accompanying presentation slides and speaker notes for the Workshop are located in Chapter Five of this document; page numbers for the slides are noted in text.

3.1 Workshop Presentation Summary

Fourteen individuals representing Government and industry gave short presentations on their current work in architectures. These diverse presentations offered an enlightening view into the latest views and practices regarding software architectures, their respective definitions, and role in application engineering. Workshop presentations were given by:

- Mr. Will Tracz, IBM FSD (pages 442-457)
- Mr. Mark Gerhardt, ESL, Inc. (pages 458-471)
- Ms. Deborah Gary, DISA (pages 472-479)
- Mr. Jim Baldo, Unisys (pages 480-489)
- Mr. Charles Plinta, ACCEL (pages 490-505)
- Capt Paul Valdez, USAF ESC/ENS (pages 506-513)
- Mr. Ulf Olsson, CelsiusTech Systems (pages 514-527)
- Mr. Jim Bonine, Design Metrics Technology (pages 528-533)
- Mr. Steve Roodbeen, NUWC (pages 534-543)
- Major Grant Wickman, CECOM (pages 544-551)
- Capt Kelly Spicer, USAF SWSC/SMX (pages 552-561)
- Mr. Stellan Karnebro, Defence Materiel Administration (pages 562-574)

3.2 Workshop Presentation Issues

Because of the diverse composition of the Workshop speakers, many issues surrounding software architectures and reuse were examined. The following is an overview of some of those issues, along with key points of discussion.

3.2.1 The Role of Software Architectures

People often feel that they're communicating requirements effectively, but may instead have different views. An architecture can serve as a common point of reference. Blueprints, schematics, and the like are all ways that people communicate in their elements.

Architecture is the software communication vehicle. From an architecture point of view, systems are treated as components.

How can architectures be used in maintenance and sustained engineering activities? Mission needs shift with time; as time goes by, things change. It is valuable to have a process for transition from one architecture to another as technology changes.

In using domain specific software architectures, meeting requirements and creating particular applications in a solution space may create tension. A solution is to draw the line between the problem space and the solution space: create a domain model, pick out constraints, then create specific applications.

Currently, components aren't always compatible. Fatal component combinations must be recognized. The more layers that are added to a software architecture, the less interaction there may be between components. In some cases, it may be best to extract high level elements and start from scratch, rather than try to extract low level components to build a system.

3.2.2 Investment Considerations

The more detailed standards are, the more difficult it may be to communicate to another platform. One solution is to publish a set of "building codes" with a broad scope that will allow for architected systems.

There must be investment into a software architecture before it can be used. Initial cost of software architecture development may be prohibitive. Also, some projects may be closing down due to budget constraints. The knowledge from these projects needs to be captured rather than lost. This approach involves capturing a design hierarchy, documentation, development history, and design decisions.

Some felt the use of architectures may not apply to all kinds of systems, such as real time embedded systems at this point in time.

Experiences and experiments in developing architectures need to be documented, even from fatal architectures.

3.2.3 Architectures Defined

A good architecture is stable with a cover of customizations, while a poor architecture is the reverse with props to make it stable. When customizations get too bulky, they outweigh the base and make the system unstable.

Architectures are frameworks, but are not necessarily a solution; architectures are a layered subset of the solution.

Every design problem has an objective logical architecture. A logical architecture is an architecture in purely mathematical form.

3.3 Workshop Working Group Summaries and Issues

The Workshop participants then separated into working groups to identify common problems involving architecture and reuse implementation, and to develop a common approach to solutions to these issues. The groups were organized as follows:

- Working Group 1: Evaluation and Measurement of Architectures
- Working Group 2: Software Architecture Technologies
- Working Group 3: Software Architecture and Reuse
- Working Group 4: Software Architecture and Standards
- Working Group 5: Software Architecture and Strategic (Product-line) Planning
- Working Group 6: System Architecture Technical Committee for Reuse Library Interoperability

3.3.1 Working Group One: Evaluation and Measurement of Architectures

Working Group One concentrated on two topic questions:

- *For procurement issues, how can many proposed architectures be evaluated?*
- *For design issues, what are the "architecture-level" qualities which can and should be measured?*

In order to compare one architecture against another, we must establish a common understanding of what we mean when we refer to an architecture. Properties we are looking for in an architecture should be specified. We should provide our definition of an architecture and give examples of how we represent it.

1. The offeror must describe the architecture in 10 pages or less using the following guidelines:

   - Describe the basic elements which make up the architecture.
   - Define the rules for how the elements interact with each other.
   - Describe how these basic elements make up the system design.

   Evaluation criteria:

   - Is the design based on the architecture?
   - Is the style for defining and representing the architecture consistent?
   - Are the functions separate from the interactions?
   - Are the rules for combining the elements consistent?

2. **Evaluate the offeror's architecture on how well it addresses non-functional requirements (e.g., interoperability, ability to tolerate change, cheap to build, use of COTS). The offeror must explain and/or demonstrate this through a prototype.**

   Evaluation criteria:

   - Can the architecture incorporate new functionality based on new technology?
   - How much COTS software is used and at what level?
   - The ability to address changes in requirements.
   - How the system interacts with other systems in the domain.
   - Does the architecture incorporate open system standards?
   - Can stress points be identified? How does the architecture compensate?

3. Evaluate the offeror's architecture with respect to how it is similar or different from examples provided in the RFP.

   Evaluation criteria:

   - How much does the offeror understand about the domain?
   - Did the offeror find innovative improvements to the architecture?

3.3.2 Working Group Two: Software Architecture Technologies

Working Group Two focused on the following topic questions:

- *What are the current and emerging technologies for software architecture?*
- *Where is the "low hanging fruit" (i.e., easily attained but useful technology)?*

Views about software architecture technology depend upon your goal and perspective. Current technologies for software architecture involve the following issues:

1. Application Composition
   - Composition formalisms
   - Common infrastructure

2. Techniques for Reusable Components
   - Multi-level
   - Includes context for use definition (operational, testing, development)

3. Legacy Systems/Software
   - Extraction of architecture and components
   - Reuse in existing form

Although technologies for software architectures still need to emerge, there currently is ›vident "low hanging fruit."

1. Object-Oriented Technology
   - Development
   - Re-engineering
2. Formalisms For Composition
   - Type Expressions (Batory)
   - Architecture Description Languages
3. Interconnection Techniques
   - LIF, MIF, POLYLITH
   - UNAS
   - Wrappers/mediators
   - Standards: CORBA, OSI, etc.
4. Parameterized Programming
5. Consensus Definition of Architecture
6. Inductive Analysis of Current Exemplars
7. VHDL (Bailor)
8. Ontological Structuring

3.3.3 Working Group Three: Software Architecture and Reuse

The topic questions for Working Group Three were:

- *What does it mean for an architecture to be "reusable?"*
- *What is needed for product-line architectures to sustain a commercial component provider industry?*

Working Group Three presented an example of a layered architecture for discussion. Layering helps in understanding design. However, abstractions may be violated in implementation, and layering may be incomplete. Advantages for reuse include a partitioning strategy, and an abstraction mechanism. A disadvantage for тeuse is a need for optimization.

With regard to reusable architectures in domains, the architecture should be reusable and should also support the reuse of components. Do these conflict? Is there an issue surrounding the variability of components versus the variability of the architecture? One strategy is to utilize generative techniques and a generic architecture, which may require trade-offs. It is also noteworthy that a small domain is more vulnerable to external architecture constraints, and that a large domain involves a large number of resources.

There are also numerous issues for consideration.

Different domains, organizations, and/or audiences may have different architecture languages, views, representations, and levels of abstraction (ravioli). How can these be made reusable?

If context is linked to architecture, what about "domain-independent" idioms? Does a class/inheritance based taxonomy help capture this?

Tension between architectural "quality" (from first principles) versus fit to existing systems.

Are there "complete" architectural style taxonomies, e.g., OO procedural, pattern-directed inference, list processing?

An architecture must include at least components, connections, constraints, plus context and dynamic aspects.

Are generic architectures applicable for every domain? Are they high level designs with "plug and play" variability at lower levels?

What is meant by reuse in architecture? Reusable architectures? Component reuse in architectures? What is the difference between usability and reusability?

Architectural representations as assets: Freely accessible versus export controlled? Are they attractive? Are they from fielded systems?

Facets/keywords for describing architectures: Are they agreed to (de facto)? Where are they documented (standards)? Can they be retrofitted to existing assets?

Is a layered architecture descriptive enough to describe everything needed to develop a system? For reusability?

Are architectures from Domain Analysis results integratible with existing components? Are architectures from existing systems/components limited to existing capabilities?

3.3.4 Working Group Four: Software Architecture and Standards

Working Group Four examined two topic questions:

- *What is the relationship between architecture and open systems?*

- *What are the areas of architecture standardization, e.g., "building codes?"*

There is definitely a relationship between software architecture and open systems. While a "good" architecture is cheap and modifiable, a "good" architecture also exploits open systems for the lifetime of the product. However, an open system should not dictate the architecture. In this context, there are restrictive standards; this applies to a wide range and to certain system attributes. Also, there need to be enabling standards which deal with market opportunity, especially in areas such as component suppliers and cost effective system solutions.

The topic of standardization and architecture often involves architecture and multiple "building codes." There are often degrees of constraining architecture, and regional variation in the "codes." There needs to be standardization at various layers of software architecture. The purpose of standardization has multiple elements, such as:

- Portability
- Interoperability
- Product Family
- Component Supplier Market
- Conformance
- Bureaucracy Preservation

Approaches to standardization include:

- Proprietary, Publicly Known
- Negotiation
- Forum

Areas for standardization can include:

- Interfaces - syntax connections
- Data Consistency - semantic connections
- Usage Consistency

3.3.5 Working Group Five: Software Architecture and Strategic (Product-line) Planning

The topic questions for Working Group Five were:

- *Where in the DoD should architectures be specified? Maintained? Implemented? What are the pros/cons of various approaches?*
- *How can DoD architectures, if specified, be used prescriptively in procuring systems?*

Group Five noted that there must be some assumptions made:

- Offerers may provide an architecture.
- It is important that the Government own the Domain Model (source of evaluation criteria).

The following issues were raised.

How do we convey what we mean by architecture? This can be done through white papers and examples.

What questions can be asked about architecture which can discriminate alternative proposals? There is reasonable certainty that answers to this question will be different.

How can you get common representations?

How is it possible to get an apples to apples comparison against criteria? Approaches include:

- develop evaluation characteristics
- likely to be non-functional
- scenarios make these concrete and evaluatable

3.3.6 Working Group Six: System Architecture Technical Committee for Reuse Library Interoperability

Working Group Six, a subgroup of the Reuse Library Interoperability Group (RIG), concentrated on issues surrounding reuse library interoperability. A topic of discussion was:

- *What are some techniques for analyzing and comparing architectures (of reuse libraries) for interoperability?*

The discussion was difficult because of vocabulary problems, but a suggestion was offered; there should be at least the possibility of a domain analysis for interoperability. The Group also discussed a Technical Reference Model (TRM) for interoperability. This can be divided into three elements:

- User Services (focus on the end user/the driver)
- Support Services (common for interoperating applications)
- Framework Services (common for all interoperating applications)

1. Using end user services maps to support services which maps to the framework in order to interoperate.
2. Missing user services indicate missing support or framework services.
3. Adding support or framework services implies new user services.

Projecting the TRM through the architecture shows the implications of the architecture style. Also, this will work for designs and implementations, providing greater detail.

4 Architecture Seminar and Workshop Summary

Approximately eighty people attended the Seminar and Workshop on November 16 and 17, 1993. Twenty-nine participants were from Government or DoD organizations, twenty-four represented industry, twelve were from academia, and fifteen were from CARDS or other organizations. Key points from the Seminar and Workshop include:

1. There were multiple, valid perspectives regarding architectures.
   - Computer Science (idioms, computational models, etc.)
   - Design (standards, methods, education, etc.)
   - Engineering (prediction, measurement, non-functionals, etc.)
   - Systems (high-level designs for applications)
2. There is a relationship between architecture and software reuse.
   - High-level designs accompanied by context information
   - Trends toward intersection of object-orientation and event systems
3. There is significant interest in the subject of software architectures.
4. While the Seminar focused on technology, there are equally strong connections to economics.

Participant responses and results from evaluation forms are in the following sections.

4.1 Evaluation Form Summary

4.1.1 Overview

As Seminar and Workshop participants registered, they were provided with evaluation and feedback forms as part of their registration packets. Twenty-nine of the participants responded, and the following results are based on those responses.

There was a consensus that the Seminar and Workshop were very successful and beneficial, and that there should be similar events in the future, either annually, every two years, or every six months. Many noted that there should be more time allotted, as a large amount of information was presented in a relatively short time. There was also a consensus that there should be smaller working groups which focus on particular areas of interest.

4.1.2 Detailed Comments

The participants suggested that particular individuals be invited to future Seminars/Workshops. That list includes Bruce Anderson or a real building architect and a movement training specialist (spatial analogies), Christopher Alexander, Gary Whitted (IMASS Program), Rob Sturtenant (McDonnell Douglas and CIT Program), select individuals from the software engineering community, architects from other fields (panel session), DISA, CFA, NRaD, MICOM, DSSA, service and DoD group leaders that are working on joint and multi-service common architectures, international representatives (Europe and Japan), Reuben Prieto-Diaz, Sholom Cohen, Mary Shaw, and John Foreman.

Several suggestions were made regarding the Workshop. It was suggested that there be more working groups and more time for discussion. Also, three groups in one room was impractical. It would be better to have smaller working groups; if they must be large, they should focus on diverse viewpoints with mechanisms for synthesizing input (e.g., future search conference). Some noted that there should have been more information geared to the participant who has limited or no previous knowledge of architectures. The next Workshop should attempt to produce, as a group, a viewer definition of software architectures and examples, including success stories.

Several comments were also made with respect to how software architectures were defined and presented. Comments indicated a good mix of CARDS and non-CARDS experts. One attendee noted, "I think the audience was opened tôo broadly too early. It would have been better to have an initial workshop to solidify the issues and CARDS viewpoints before having a workshop/forum like this one." It would have also been useful to have the CARDS Architecture Task Force (ATF) talk delivered earlier to provide some context. Also, the tool/representation survey was presented with virtually no context and was, therefore, ıelatively of little benefit.

It was suggested that there be more specific architectures presented. Following this, have participants provide constructive criticism, and break into a domain working group and develop architectures. Then, present the results to the main group. There could have been more discussion of the qualities of an architecture and distinctions between design and architectures. Another suggestion was to have more examples and hands-on interaction. Participants want information and examples which they can apply. One recommendation was to use a lecture room that is more accommodating for this type of event.

Regarding supporting materials, significant papers or books might be made available, either for free or purchase. Workshop presentation slides should be provided beforehand, and handouts should also be provided from the panelists and invited guest speakers. A speaker/attendee list should be available, as well as more information provided electronically. A bibliography with list of references, citations, and resources should also be distributed. Demonstrations of the tools should be included (if for nothing else, to interrupt the flow of the "talking heads").

4.1.3 Evaluation Form Results

Ninety-six percent of responding participants acknowledged that they would be able to apply knowledge gained from the Seminar and Workshop on the job and three percent were unsure. Sixty-eight percent said they had some previous knowledge of software architectures, twenty-nine percent had limited knowledge, and four percent indicated no previous knowledge of software architectures. One hundred percent of responding participants said that their knowledge of software architectures was enhanced or increased in some way. One hundred percent also desired to have future seminars. Four percent preferred to have them quarterly, twenty four percent preferred to have them semi-annually, sixty eight percent preferred to have them annually, and seven percent preferred to have them every other year.

Additional evaluation form results are summarized below.

	% Adequate	% Inadequate	% About Right
Session 1	70	—	30
Session 2	73	8	19
Session 3	63	15	22
Session 4	58	27	15
Session 5	54	29	17
Session 6	65	---	35

Table 1: Time Given for Each Session

	% Too Specific	% Too General	% Adequate
Session 1	4	---	96
Session 2	4	9	87
Session 3	4	12	84
Session 4	5	22	72
Session 5	---	13	87
Session 6	5	5	91

Table 2: The Material Covered

	% Too Much	% About Right	% Not Enough
Session 1	---	100	---
Session 2	9	83	9
Session 3	8	63	29
Session 4	---	67	33
Session 5	---	73	62
Session 6	---	95	5

Table 3: Contents of Concepts

	% Poor	% Fair	% Good	% Excellent	N/A
Refreshments	8	12	58	23	---
Facilities	8	15	50	26	---
Visual Aids	---	20	63	15	---
Lunch	19	19	50	8	4
Handouts	---	7	43	50	---
Examples	---	27	46	15	12

Table 4: Supporting Services

	% Poor	% Fair	% Good	% Excellent	N/A
Knowledge	---	8	48	44	---
Responses	---	5	15	24	---
Selection	---	9	57	35	---

Table 5: Panel Discussion

5 Architecture Seminar and Workshop Presentation Slides

This Chapter contains presentation slides from the Seminar, the Seminar panel discussion, and the Workshop. The slides are divided into three sections, prefaced by introductory slides.

Software Architecture Seminar slides (pages 21-321) are from the five Seminar sessions:

- Session I *Why Architectures* (pages 35-58)
- Session II *Senses of Architecture: Building the Category* (pages 59-140)
- Session III *Software Architecture and Reuse* (pages 141-216)
- Session IV *Architecture-Based Reuse Tools* (pages 217-274)
- Session V *CARDS Approach to Reuse and Software Architecture* (pages 275-321)

Slides from the Seminar Panel Discussion (pages 322-435) were used by the four panel members:

- Mr. T.F. "Skip" Saunders, Mitre Corporation (pages 324-371)
- Mr. Hans Polzer, Unisys Corporation (pages 372-383)
- Mr. Stan Levine, US Army CECOM (pages 384-423)
- Capt Frederick Swartz, USAF ASC/YTE (pages 424-435)

Software Architecture Workshop slides (pages 436-574) are from Workshop presentations given by:

- Mr. Will Tracz, IBM FSD (pages 442-457)
- Mr. Mark Gerhardt, ESL, Inc. (pages 458-471)
- Ms. Deborah Gary, DISA (pages 472-479)
- Mr. Jim Baldo, Unisys (pages 480-489)
- Mr. Charles Plinta, ACCEL (pages 490-505)
- Capt Paul Valdez, USAF ESC/ENS (pages 506-513)
- Mr. Ulf Olsson, CelsiusTech Systems (pages 514-527)
- Mr. Jim Bonine, Design Metrics Technology (pages 528-533)
- Mr. Steve Roodbeen, NUWC (pages 534-543)
- Major Grant Wickman, CECOM (pages 544-551)
- Capt Kelly Spicer, USAF SWSC/SMX (pages 552-561)
- Mr. Stellan Karnebro, Defence Materiel Administration (pages 562-574)

The slides from the Panel Discussion and the Workshop were optically scanned and imported into this document. Page numbers are at the bottom right corner.

Central Archive for Reusable Defense Software (CARDS)

Software Architecture Seminar

16 November 1993

Kurt C. Wallnau and Paul A. Kogut, Unisys Corporation

with contributions from:

James Estep, Unisys Corporation
John Gaddis, DSD Laboratories
Kerri Haines, Unisys Corporation
Scott Hissam, Unisys Corporation
Terry Huber, DSD Laboratories
Quiang Lin, Galaxy Global Corporation
Mark Lipshutz, Unisys Corporation
Roslyn Nilson, Unisys Corporation
Charles Snyder, Unisys Corporation
Nancy Solderitsch, Unisys Corporation
Roger Whitehead, DSD Laboratories

Acknowledgments

We want to thank the following contributors, without whose help this seminar would not have been possible:

Tom Bock, Shelly Jones and George Jackelen, Electronic Warfare Associates, for their heroic efforts.

Charlie Snyder, Unisys, for his organizational skills.

Jim Estep, Unisys, for his cool-headed optimism and ability to make things happen.

Welcome to CERC

CARDS would like to thank the Concurrent Engineering Research Center (CERC) for donating the use of their facilities to host this seminar.

CERC was established in 1988 by the DoD's Advanced Research Projects Agency (ARPA) in response to a national need to improve the product development capabilities of the U.S. defense-industrial base. As the centerpiece of the (D)ARPA Initiative in Concurrent Engineering (DICE), CERC's mission is to design, develop, and promote concurrent engineering technologies.

CERC has recently expanded the application of its technology to the healthcare informatics domain. Funded by the National Library of Medicine, CERC is developing a pilot healthcare information system that will integrate the latest developments in multimedia, networking, and user interfaces to provide shared access to multimedia patient records, and to enable remote consultation among participating state medical facilities.

Miscellaneous

MESSAGES:

Messages for participants of the forum can be left at the CERC switchboard: (304) 293-7226

All messages will be posted outside the door to this room

PARKING:

Ignore the "parking decal required" signs - the WVU parking authority has been notified not to ticket cars parked at the CERC facility

ASSISTANCE:

For help or assistance at any time, contact the seminar support staff (red ribbons)

LUNCH:

Will be served on the fourth floor

There will be a box available for depositing the $10.00 to cover food and beverage costs

Seminar Schedule 16 November

8:00 AM	Seminar Logistics - Charlie Snyder
8:10 AM	CERC Welcome - Dr. Ramana Reddy
8:20 AM	CARDS Welcome - Bob Lencewicz
8:30 AM	Why Architectures? - Charlie Snyder/Kurt Wallnau
9:15 AM	Break
9:25 AM	Senses of Architecture - Paul Kogut/Kurt Wallnau
10:35 AM	Break
10:45 AM	Software Architectures and Reuse - Wallnau/Kogut
12:00 AM	Lunch - 4th Floor Antechamber

Seminar Schedule 16 November - continued

1:00 PM	Case Studies of Reuse Systems - Kogut
2:15 PM	Break
2:25 PM	CARDS use of Architectures - Nancy Solderitch
3:05 PM	Break
3:15 PM	Panel Session - Architectures in Practice - T. Saunders, Mitre - H. Polzer, Unisys - S. Levine, CECOM - F. Swartz, Air Force ASC/YTE
5:00 PM	Summary and Closing Remarks
5:30 PM	CERC Demonstrations and Tour

Architecture Forum Workshop - 17 November

Purpose:

- **Explore the current practice of software architectures and software reuse on actual projects**
- **Explore current research into architecture as a means of implementing reuse**

Overview:

- **Morning:**
  - **Short presentations by practitioners and researchers on their current work with architectures**
- **Afternoon:**
  - **Working session to identify common problems in reuse implementation and develop a common approach to solutions**

Workshop Schedule 17 November

8:00 AM **Transitioning from research to practice - T. Saunders, Mitre**

8:30 AM **Architecture as the framework for realizing the benefits of reuse - W. Tracz, IBM**

8:45 AM **Abstraction and layering within software architectures - M. Gerhardt, ESL**

9:00 AM **Overview of DISA Software Reuse Domain Analysis - D. Gary, DISA**

9:15 AM **Software Architecture, Reuse, and Maintenance - Jim Baldo, Unisys**

9:30 AM **Break**

9:45 AM **The Object-Connection-Update Architecture - Charles Plinta, ACCEL**

Workshop Schedule 17 November - Continued

10:00 AM	PRISM software architecture - P. Valdez, ESC/ENS
10:15 AM	NSA Unified INFOSEC Architecture (UIA) - B. Koehler, DIRNSA
10:30 AM	9LV Mk3 shipboard C2 architecture - U. Olsson, CelsiusTech Systems
10:45 AM	Architectures and the real world, based on the Army C2 common software program experience - S. Levine, Army
11:00 AM	Break
11:15 AM	Architectures in the CIS field - applying Christopher Alexander's work - J. Bonine, Design Metrics Technology
11:30 AM	OO-based architecture use at NUWC - S. Roodbeen, NUWC
11:45 AM	Capturing domain knowledge at NTF - T. Gill, NFT/ENS

Workshop Schedule 17 November - Continued

12:00 PM	STARS demo project architecture - G. Wickman, CECOM
12:15 PM	The STARS Air Force Demo Project - K. Spicer, SWSC/SMX
12:30 PM	Lunch - 4th Floor Antechamber
1:30 PM	Working Groups
4:30 PM	Working Group Report
5:00 PM	Wrap-up

Proposed Working Groups and Topics - 17 November

- WG 1: Evaluation and Measurement of Architectures
  - procurement issues: how can many proposed architectures be evaluated?
  - design issues: what are the "architecture-level" qualities which can and should be measured?
- WG 2: Software Architecture Technologies
  - what are the current and emerging technologies for software architecture?
  - where is the "low hanging fruit" (i.e., easily attained but useful technology)?
- WG 3: Software Architecture and Reuse
  - what does it mean for an architecture to be "reusable?"
  - what is needed for product-line architectures to sustain a commercial component provider industry?
- WG 4: Software Architecture and Standards
  - what is the relationship between architecture and open systems?
  - what are areas of architecture standardization, e.g., "building codes?"
- WG 5: Software Architecture and Strategic (Product-Line) Planning
  - where in the DoD should architectures be specified? maintained? implemented? What are the pros/cons of various approaches?
  - how can DoD architectures, if specified, be used prescriptively in procuring systems

Forum Evaluation Form

Please take a few minutes at the end of the forum to complete the evaluation form provided in your handouts.

We need your comments to improve our seminars and ensure that their contents are relevant and timely to the software reuse community.

Any comments, suggestions, or criticisms are solicited, either attach them to the evaluation form or contact either:

Charlie Snyder, Forum Coordinator, (304) 363-1731, snyder@cards.com

or

Kurt Wallnau, CARDS System Architect, (304) 363-1731, wallnau@cards.com

Dr. Reddy

Dr. Ramana Reddy is a Professor of Computer Science and the Director of the Concurrent Engineering Research Center (CERC) at the West Virginia University. At CERC, Dr. Reddy leads the development into enabling technologies for concurrent engineering. He has achieved significant research results in multimedia communications, constraint management, uncertainty reduction, and knowledge-based systems.

"Don't let it get away!" 81

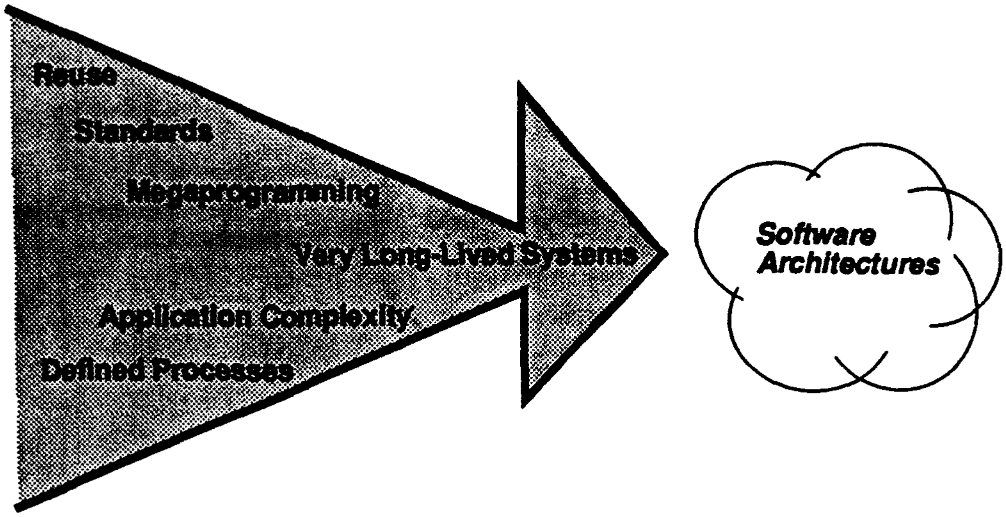
Central Archive for Reusable Defense Software (CARDS)

Session I
Why Architectures?

16 November 1993

This page intentionally left blank.

A Natural Continuation of Current Work



A Natural Continuation of Current Work

Software Architecture is a topic of considerable interest to practitioners and researchers in the academic, government, and commercial software areas.

Why? Why now?

What is the relevance to an organization trying to improve its software development capability?

How does architecture relate to the other software development improvement concepts of

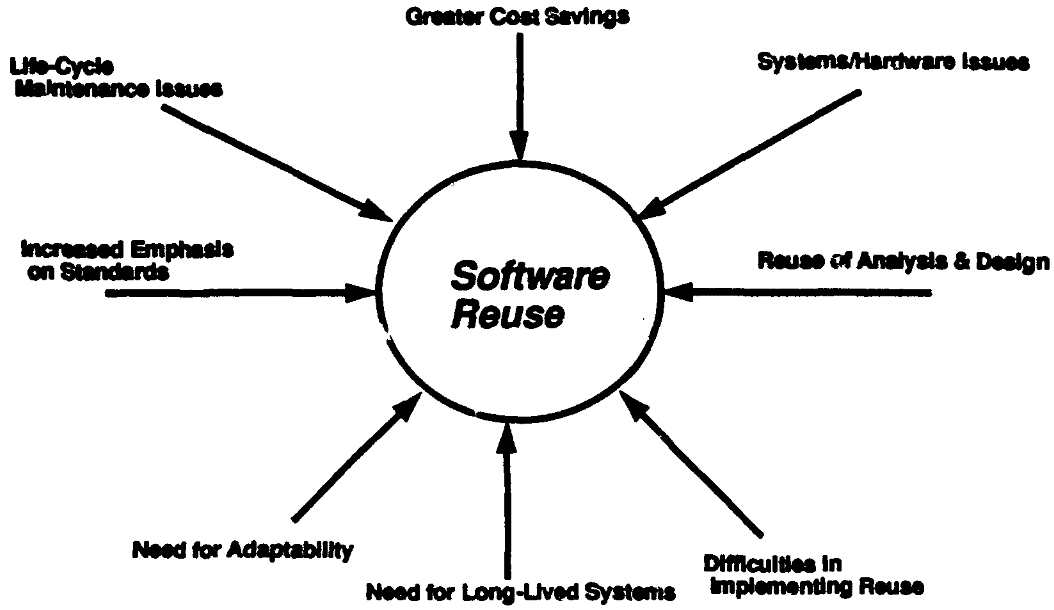
- Software Process Improvement - SEI CMM
- Total Quality Management
- Metrics and Statistical Process Control
- STARS Megaprogramming
- Domain Analysis and Domain Engineering
- Library based Reuse

Object-Oriented Analysis and Design

Many of the research topics and implementation efforts seem inevitably to lead to the study of software architectures. This seems to stem from the continual human endeavor of always trying to generalize and conceptualize from a specific instance to a more general case.

We believe that the current interest in software architectures represents the natural evolution of the historical focus on changing software development from a craft to an engineering discipline.

Why do we need Software Architectures?



Software Reuse

Greater Cost Savings

Systems/Hardware Issues

Reuse of Analysis & Design

Difficulties in Implementing Reuse

Need for Long-Lived Systems

Need for Adaptability

Increased Emphasis on Standards

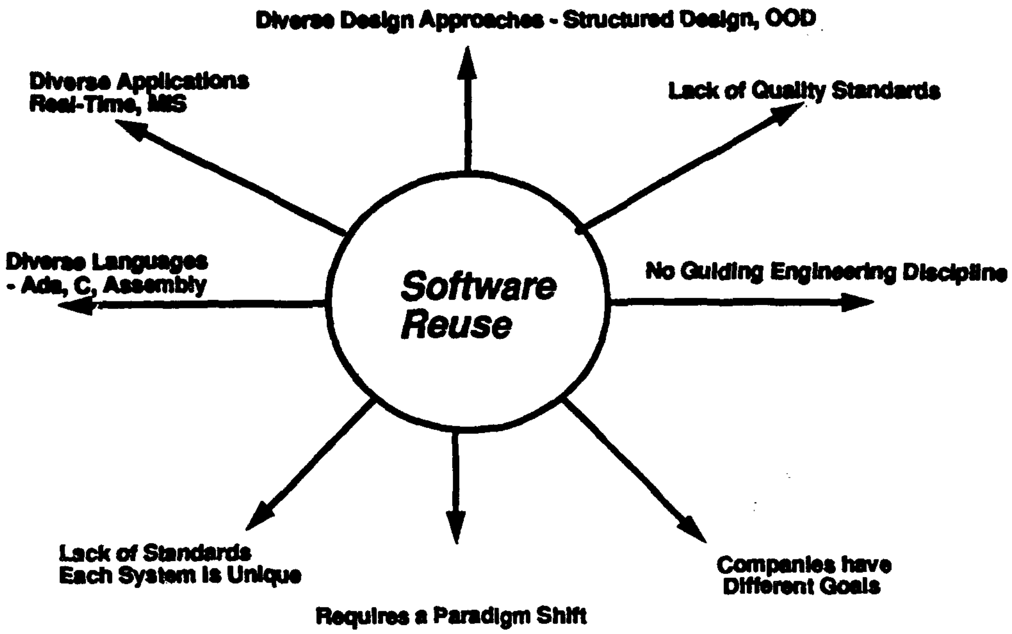
Life-Cycle Maintenance Issues

Why do we need Software Architectures?

There are many forces at work leading research and implementation efforts into considering architectures as an area of major payoff in software development improvement. Some major ones, and their implications are listed below:

- Reuse of Analysis & Design - The higher level at which the artifacts are reused, the greater the payoff.
- Systems/Hardware Issues - System performance and hardware capabilities often determine the software design
- Difficulties in Implementing Reuse - Reuse of other than minor code modules is very difficult because reuse is typically considered after system design decisions have been made.
- Need for long-lived systems - Systems must be enhanced as new technologies appear.
- Need for adaptability - Longer lived systems have to change to meet situations not envisioned when they were developed
- Increased emphasis on standards -Systems now must conform to various interface standards and often development standards that require interface to a variety of existing COTS software.
- Life-cycle Maintenance Issues -Software systems that use COTS and open standards are easier to maintain than custom developed software.
- Greater Cost Savings - Reuse and developing software using large-scale existing components promises to significantly reduce development cost. Those savings have been historically difficult to achieve.

Why not before Now?



Diverse Design Approaches - Structured Design, OOD

Diverse Applications
Real-Time, MIS

Lack of Quality Standards

Diverse Languages
- Ada, C, Assembly

Software Reuse

No Guiding Engineering Discipline

Lack of Standards
Each System is Unique

Companies have
Different Goals

Requires a Paradigm Shift

Why not before Now?

Architecture has just recently become a focus of study by the reuse community. While a major reason for this just occurring is the increased emphasis on recognizing patterns in domain engineering and other reuse activities, there are other forces serving to inhibit architecture engineering:

- Diverse Design Approaches - The myriad of design methodologies inhibits a recognition of common structure. And how can you reuse C++ classes in a structured design developed system?
- Diverse Applications - Practitioners consider each application domain as unique and unable to share with other outside domains. Thus the real-time practitioners and the MIS community continue to evolve in separate ways.
- Diverse Languages - While code incompatibilities are obvious, many times the choice of language dictates the design in subtle ways. This is most obvious with C++ and other OO languages, but Assembly Language also scopes the design choices available
- Lack of Standards - Standards define the boundaries and limits on the design. Without standards, there are no limits—every new system is a complete new challenge.
- Lack of Quality Standards - How can the choice be made between several designs and approaches without some standard defining the quality of the product. Software development is just beginning to have such a standard.
- No Guiding Engineering Discipline- Software engineering lacks the theoretical base of other engineering disciplines, it is currently more a craft.
- Companies have different goals - Consider a full fixed-price contract (FFP) vs. a cost plus fixed fee contract(CPFF). There is no incentive for the contractor to control software costs on the CPFF contract. Government auditors often disallow costs savings measures on the FFP contract. In all cases the benefits from controlling costs to the contractor are somewhat nebulous—the contractor wants to win new business as the major goal.
- Requires a Paradigm Shift - Just as with the concept of Software Process Improvement, reuse requires a major change in organization for a company. Software now must be understood, made an item of capital investment, and must be managed. But many managers come from hardware or business areas and have no understanding of or interest in software development.

The Goal of the Seminar

To use architectural concepts, we must understand:

- the various meanings of software architecture
- the current research in the field of architecture
- current efforts in applying software architecture

These and other concepts will be explored during the remainder of this seminar.

The Goal of the Seminar

The remaining sessions will explore software architectures and the usefulness of the concept for implementing software reuse.

Some Cautionary Thoughts Before Proceeding

Exercise: Define the concept "Game"

No matter what you try, you will define a conceptual category which:

— includes something which should be excluded
— excludes something which should be included

Intensional definitions do not work well with abstract conceptual categories

Some Cautionary Thoughts Before Proceeding

This example illustrates an old trick philosophy professors play on students—setting up definitions only to knock them down again. As it turns out, there are sound reasons why this trick "works" where fairly abstract concepts are concerned, as revealed by researchers in cognitive psychology.

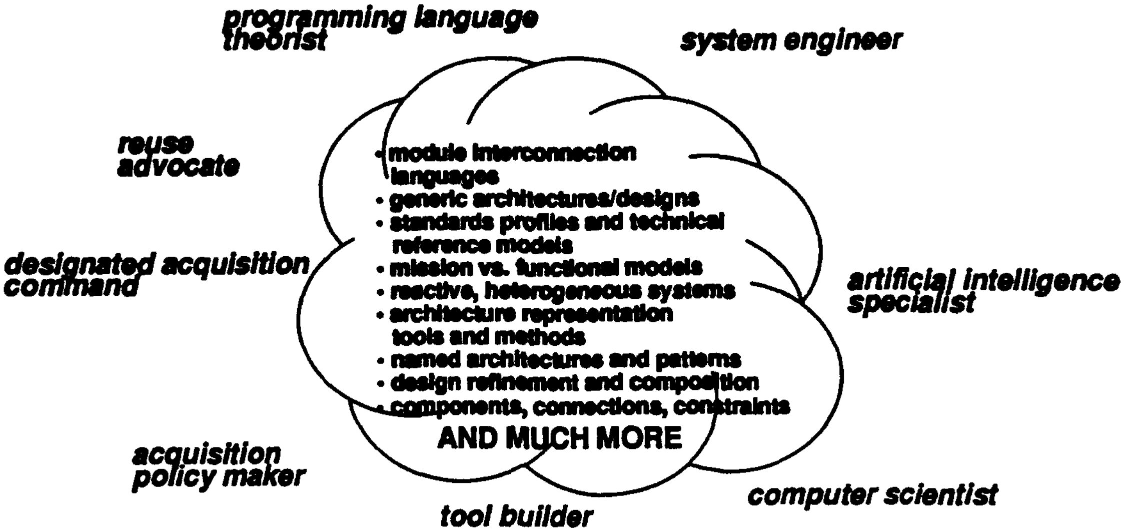
The bottom line is that understanding what forms a cognitive category is no mean feat.

References

George Lakoff, Women, Fire and Dangerous Things: What Categories Reveal About The Mind, University of Chicago Press, 1991. ISBN 0-226-46803-8.

Architecture as a Conceptual Category

Categories are formed from experience



programming language theorist

system engineer

reuse advocate

designated acquisition command

artificial intelligence specialist

acquisition policy maker

tool builder

computer scientist

- module interconnection languages
- generic architectures/designs
- standards profiles and technical reference models
- mission vs. functional models
- reactive, heterogeneous systems
- architecture representation tools and methods
- named architectures and patterns
- design refinement and composition
- components, connections, constraints

AND MUCH MORE

Architecture as a Conceptual Category

Given that we form conceptual categories based upon our own experiences (we can assume this proposition for the purposes of the seminar, even though this theory is by no means universally held as "truth revealed"), it should not be surprising that a number of different perspectives on the topic of software architectures, and domain-specific software architectures and reuse, have emerged.

Quite apart from the natural tendency in the research community to reward "innovative" and "unique" approaches (which tends to generate approaches which have commonality well-concealed beneath layers of obscure terminology), there is also a natural tendency to stress what is important in a category based upon personal experiences and personal needs.

The chart illustrates a number of different perspectives which might lead to a number of different interpretations about what constitutes the most central concept in the architecture category. Naturally no attempt has been made to enumerate all roles or all definitions/concepts for the architecture category, nor is it implied that one perspective is only narrowly interested in one concept (that is what is implied by "most central member").

A Smattering of Software Architecture Definitions

organizational structure of a system or component - IEEE Std 610.12

specification for the assembly of software components - Braun

elements, form (weighte [illegible] rties and relationships), and rationale - P [illegible] Wolf

high level description of a generic type of software system:
--functional roles of major components
--interrelationships stated in an application oriented language
--precise semantics for automated reasoning
--libraries of prototype components with executable specifications
--program synthesis capability to produce optimized code
--a constraint system for reasoning about consistency
--design records that link requirements to design decisions
Lowry

components and connectors that have structural patterns - Garlan/Shaw

the packaging of functions and objects, their interfaces, and control to implement applications in a domain - Peterson

fundamental structural attributes of a software system:
--partitioning into components
--flow of data and control
--critical timing and throughput
--layers/standards/protocols
--allocation of software to hardware
Saunders

--framework for logical and physical partitioning
--semantic model of communication and cooperation
--layering capability to add algorithmic functionality "on top of" the framework
Commons/Gerhardt

A Smattering of Software Architecture Definitions

In times of crisis, however, we can find comfort in definitions. There are a number of definitions of software architecture found in the literature (this list is not meant to be complete). The definitions usually reflect the perspective of the author (e.g. Lowry has an AI perspective). Note that in some cases a single author will have several different "senses" of the term. Perry and Wolf, for example:

> *"We use the term 'architecture' to invoke notions of abstraction, of standards, of formal training (of software architects), and of style."*

References:

- IEEE Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology, Dec. 1990
- Garlan, Shaw - "An Introduction to Software Architecture" to appear in Advances in Software Eng. and Knowledge Eng., vol.1 1993
- Perry, Wolf - "Foundations for the Study of Software Architecture" ACM SIGSOFT SEN, Oct. 1992
- Braun - "DSSAs: Approaches to Specifying and Using Architectures" STARS 92, Dec. 1992
- Peterson - "Coming to Terms with Software Reuse Terminology: a Model-Based Approach" ACM SIGSOFT SEN April 1991
- Saunders, Horowitz, Mleziva - "A New Process for Acquiring Software Architecture" MITRE TR
- Commons, Gerhardt - "A Model for Analyzing Megaprogramming, Reuse, and Domain Specific Software Architectures" TRI-Ada, Sept. 1993
- Lowry - "Software Engineering in the Twenty-First Century" AI Magazine, Fall 1992

. CARDS Context

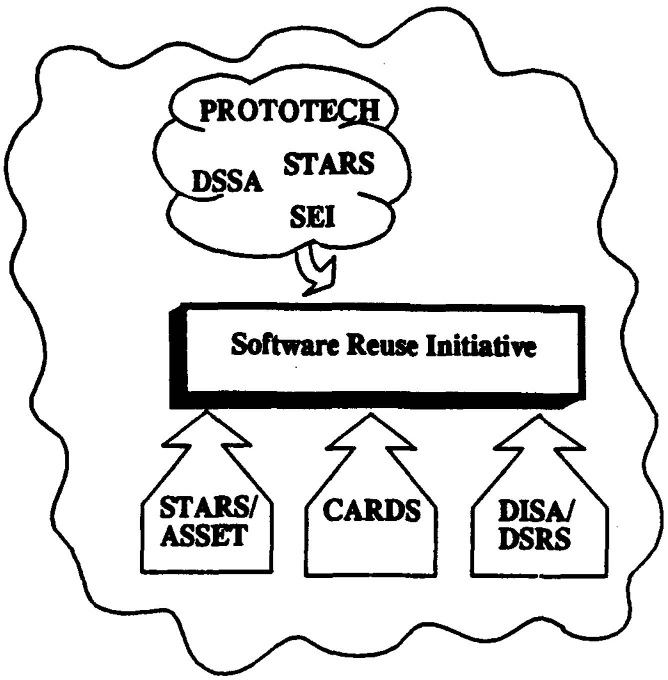
Fundamental Principles of DoD Reuse Vision and Strategy

Domain Specific Reuse

Process Driven Reuse

Architecture-Centric Investment

Interconnected Reuse Libraries



CARDS Context

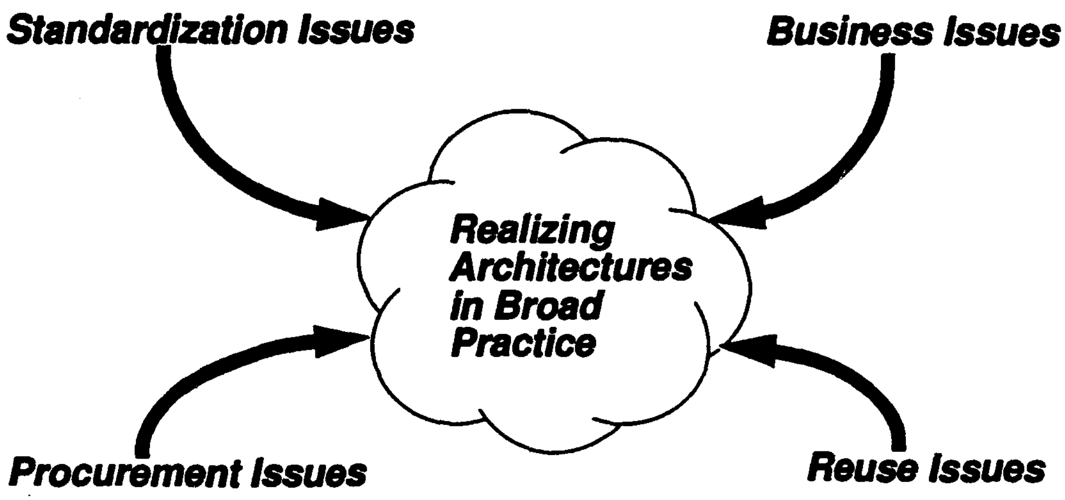
The CARDS program is one member of a larger DoD Software Reuse Initiative. The other member programs include the DISA/CIM software reuse program, and the STARS/ASSET program. These three programs provide cooperative, complementary coverage of the field of software reuse to help transition the techniques and technologies of reuse into practice.

Each of the programs are guided by the DoD Software Reuse Vision and Strategy. The four fundamental principles of the Vision and Strategy are listed on the left of the slide.

The CARDS program is interested in evaluating and transitioning reuse technologies which bring together the concepts of software architecture, domain-specific reuse and reuse libraries. As will be seen in a later presentation (Session V), CARDS is pursuing an advanced technology approach to fuse these concepts: our library technology is based on knowledge-representation formalisms which help us represent software architectures and provide automated reuse assistance based on architecture models and a library of software components.

This is one reason why CARDS is so actively interested in the state of research and the state of practice in the field of software architecture.

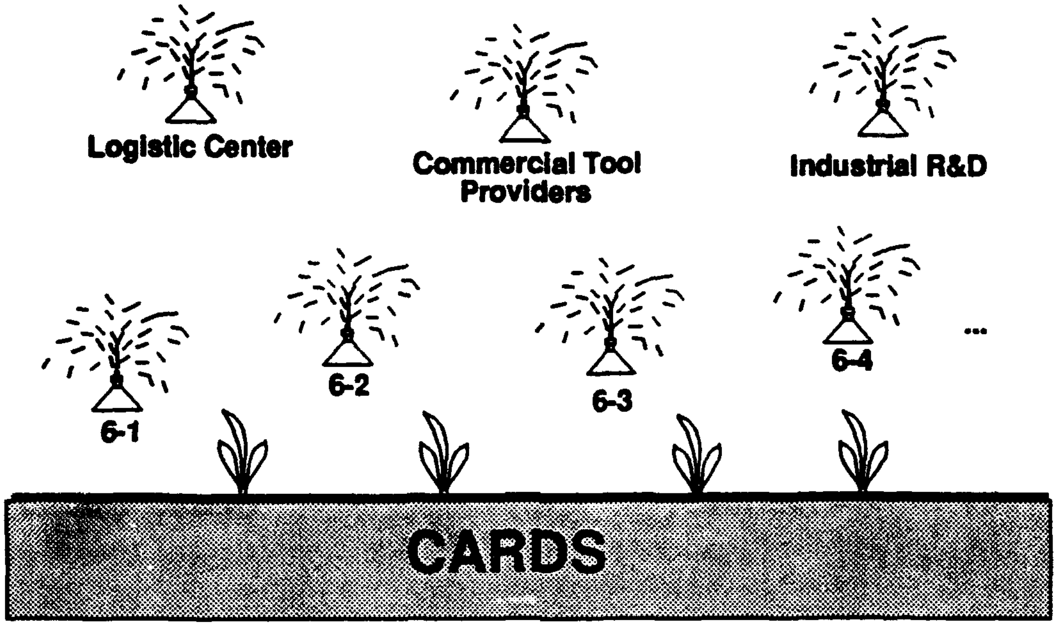
The Topic Transcends Technology



The Topic Transcends Technology

The convergence of business, policy, and technology must be a consideration, as well as differentiating architecture technology from reuse technology. CARDS, to be successful, needs to have a sufficiently broad technical foundation to express the trends of architecture and domain-specific architecture methods and technologies in order to help guide the formulation of business and acquisition models.

CARDS Cross Section of Ideas

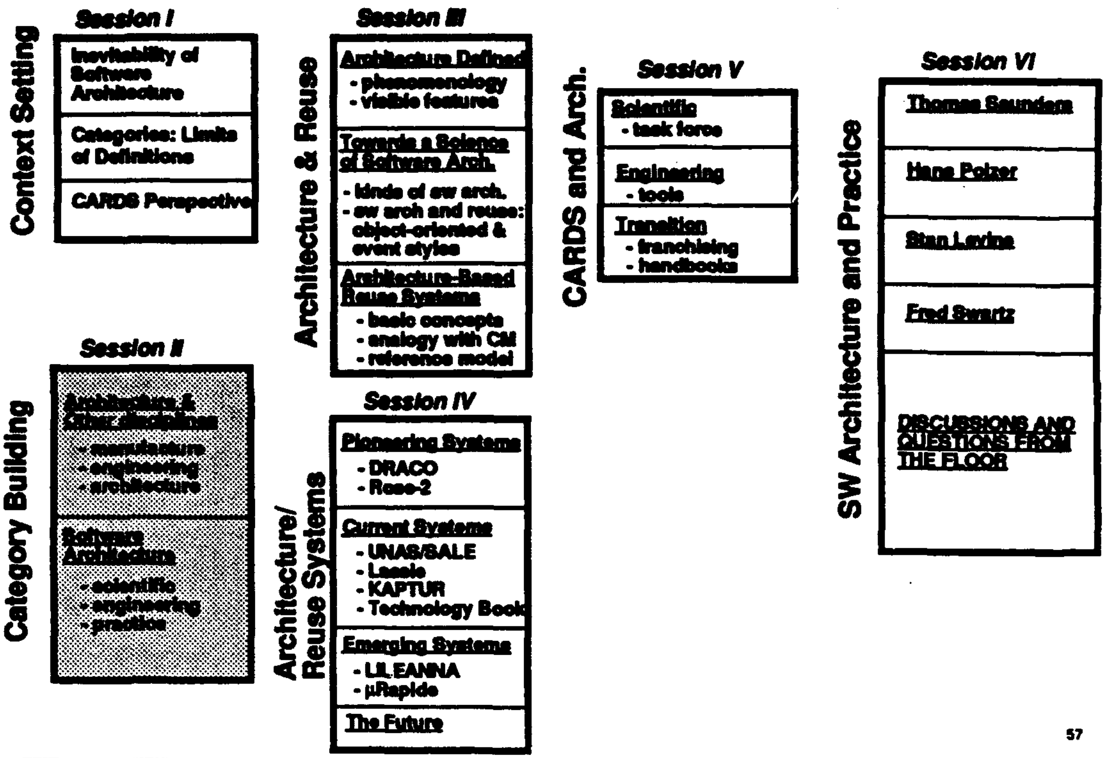


CARDS Cross Section of Ideas

Good ideas on the topic of software architecture are not emerging only from research programs. In a sense, the image of a technology pipeline is inaccurate—a better image might be a series of technology sprinklers:

- basic research: theory, concepts, taxonomies of architectures.
- applied research: experimental, proof-of-concept technologies
- advanced technology demonstrations: demonstrations of scale-ability
- ongoing development programs: transition issues
- logistics and support programs: retro-fitting, reverse engineering, integration and test

The motivation for this seminar, and especially the follow-up workshop, is to help the CARDS program to cut-across these boundaries, to identify a broad cross-section of ideas on software architectures. In turn, CARDS hopes to use this knowledge to help accelerate the transition of good ideas into practice, as well as provide feedback to research and development efforts into the perspectives of practicing engineers.



Context Setting

Session I

Inevitability of Software Architecture

Categories: Limits of Definitions

CARDS Perspective

Category Building

Session II

Architecture & Other disciplines
- manufacture
- engineering
- architecture

Software Architecture
- scientific
- engineering
- practice

Architecture & Reuse

Session III

Architecture Defined
- phenomenology
- visible features

Towards a Science of Software Arch.
- kinds of sw arch.
- sw arch and reuse: object-oriented & event styles

Architecture-Based Reuse Systems
- basic concepts
- analogy with CM
- reference model

Architecture/ Reuse Systems

Session IV

Pioneering Systems
- DRACO
- Rose-2

Current Systems
- UNAS/SALE
- Lassie
- KAPTUR
- Technology Book

Emerging Systems
- LILEANNA
- µRapide

The Future

CARDS and Arch.

Session V

Scientific
- task force

Engineering
- tools

Transition
- franchising
- handbooks

SW Architecture and Practice

Session VI

Thomas Saunders

Hans Polzer

Stan Levine

Fred Swartz

DISCUSSIONS AND QUESTIONS FROM THE FLOOR

Anatomy of this Presentation

This chart depicts a more detailed anatomy of the seminar, with the size of each session block in rough scale to the time allotted.

The top-level structure of the seminar are:

- Session I: Context setting
- Session II: Building a conceptual category for software architecture
- Session III: Synthesizing session II into a working model of software architecture, and extending our focus into kinds of software architectures and architecture-based reuse systems
- Session IV: A survey of architecture-based reuse systems
- Session V: A short overview of what CARDS is currently doing, relative to software architecture
- Session VI: A panel discussion on the practical concerns of adopting software architectures in the DoD—which, hopefully, will reveal interesting issues and ignite interesting discussions.

Tomorrow, of course, is a workshop where we can continue the discussions, and continue to exchange ideas.

Central Archive for Reusable Defense Software (CARDS)

Session II
Senses of Architecture: Building the Category

16 November 1993

This page intentionally left blank.

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

☞ **Manufacture Perspective** →
- production discipline and automation
- interchangeable parts and assemblies
- process control

Engineering Perspective →
- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

Architecture Perspective →
- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

Scientific Foundation →
- identification, classification, description
- abstraction and analysis

Engineering Application →
- abstract and concrete models
- engineering and production techniques

Considerations in Practice →
- strategic/business considerations
- policy issues
- economic issues

Roadmap for this Session

Our approach is to take two tacks on the subject.

First we examine architecture from a broader perspective, stepping outside of the "computer science" discipline. The objective of taking a multi-disciplinary perspective to start with (limited as it is) is to establish some reasonable analogies as a basis for further elaborating the characteristics of the emerging discipline of software architecture and engineering. We look at three perspectives:

1) Manufacture — how do architectures relate to the production discipline.

2) Engineering — how do architectures relate to engineering, i.e., problem-solving disciplines

3) Design — how do architectures relate to the design, i.e., creativity, disciplines

We take these perspectives since so much of the discussions about software engineering are biased by points of view related to these perspectives ("how to put the engineering in software engineering," "how to support reuse of designs," "we need more engineering and less creativity," "component factories," etc.)

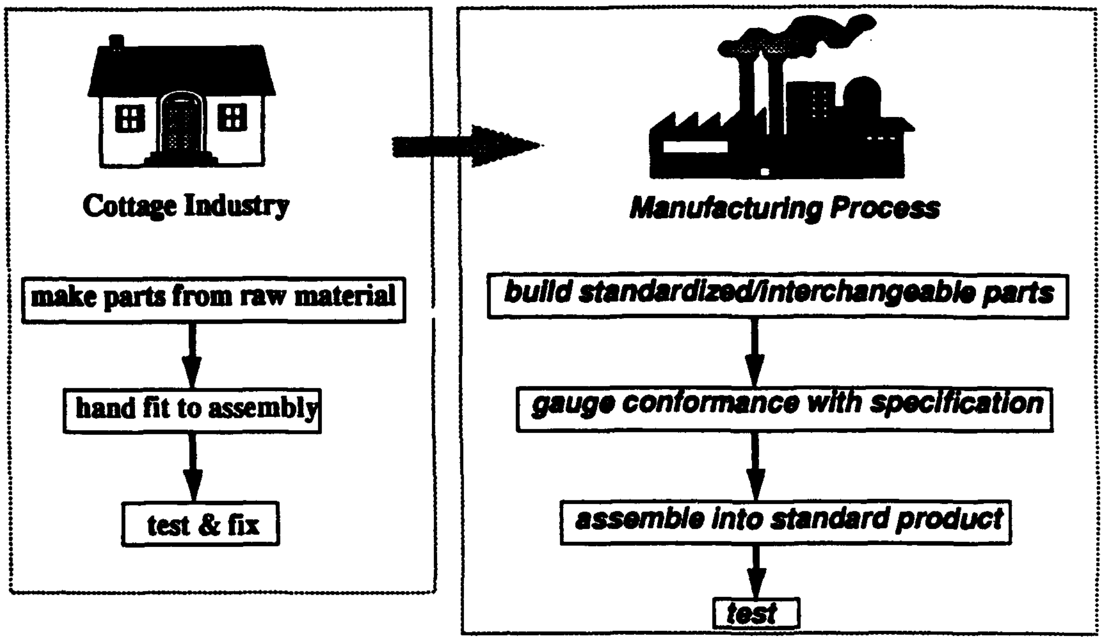
After establishing our analogy basis, we provide a high-level overview of some of the current approaches directly relevant to software architectures. Again, we take three perspectives:

1) Scientific — the study of software architectures in their own right

2) Engineering — the development of product models and production models based on architecture

3) Transition to practice — the organizational, economic and policy considerations

The Industrial Revolution

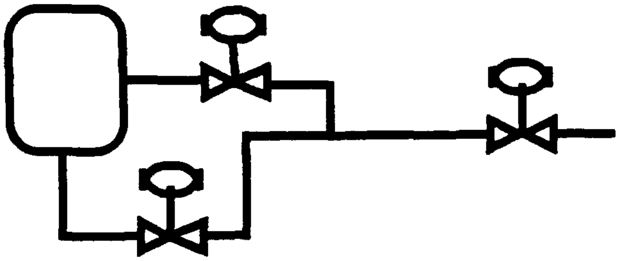

Cottage Industry
make parts from raw material
hand fit to assembly
test & fix
Manufacturing Process
build standardized/interchangeable parts
gauge conformance with specification
assemble into standard product
test

Industrial Revolution

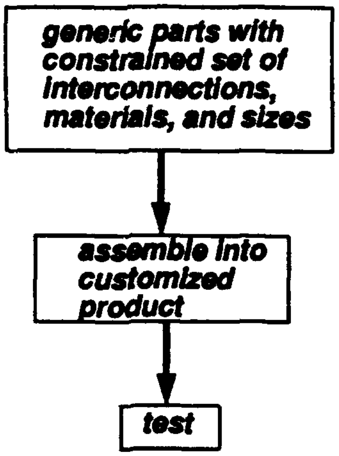
Before the industrial revolution, the production of goods and services was done in cottage industries where labor was cheap and materials were expensive. (notice that in software labor is expensive and materials —computer resources—are now cheap. In a cottage industry each part was made from raw materials (software analogy -- source code), and hand fit to an assembly (unique software design). Then testing was done and parts would be further adjusted (integration).

In the late 1700's the US Government looked for a better way to manufacture rifles. The idea was to build standard interchangeable parts which could be assembled into a rifle (a domain specific architecture). The key facilitating idea (~1820) was that a measurement procedure and tool/gauge was used to determine conformance to a specification within a certain tolerance (qualification process). It took 24 years for this "armory practice" to be adopted for commercial products (technology transition)

Build to Order



Plumbing Systems



generic parts with constrained set of interconnections, materials, and sizes

assemble into customized product

test

Build to Order

Ideas from manufacturing processes were then later adapted to "build to order" products like plumbing systems (which are more like software systems). In "build to order" there are generic parts such as valves and pipe segments which have limited ways of interconnecting, are made of only certain kinds of materials, and are available in only a certain set of standard sizes.

The "build to order" perspective most closely resembles component-based programming—i.e., programming with higher-level abstractions/building blocks. The ARPA/ProtoTech project provides one view of this kind of programming model, as reflected in some of the focus ProtoTech has on module interconnection languages (MIL) and formalisms (MIF). The idea of MIFs is to provide some standard interconnection mechanisms as a way for components to be assembled. Note that the separation of coordination from form is not a universally-held prerequisite for component-based programming.

What is most interesting in this discussion is that "build to order" need not require an architecture—there are some who believe that for specific application areas in software, e.g., information management systems, that build-to-order based on large component chunks may be more appropriate than a refine-able design solution.

The idea of separating the interconnection and coordination mechanisms from the component (or the "form") is an idea which will recur later.

References

Cox, "Planning the Software Industrial Revolution", IEEE Software Nov. 1990

Purtilo, J., Software Bus Organization: Reference Model and Comparison of Two Existing Systems, ARPA Module Interconnection Formalism Working Group Technical Note Series, TN No. 8, November 1991.

Nierstrasz, O., Component Oriented Software Development, Communications of the ACM, Vol. 35, No. 9 Sept. 1992.

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

Manufacture Perspective →
- production discipline and automation
- interchangeable parts and assemblies
- process control

☞ Engineering Perspective →
- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

Architecture Perspective →
- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

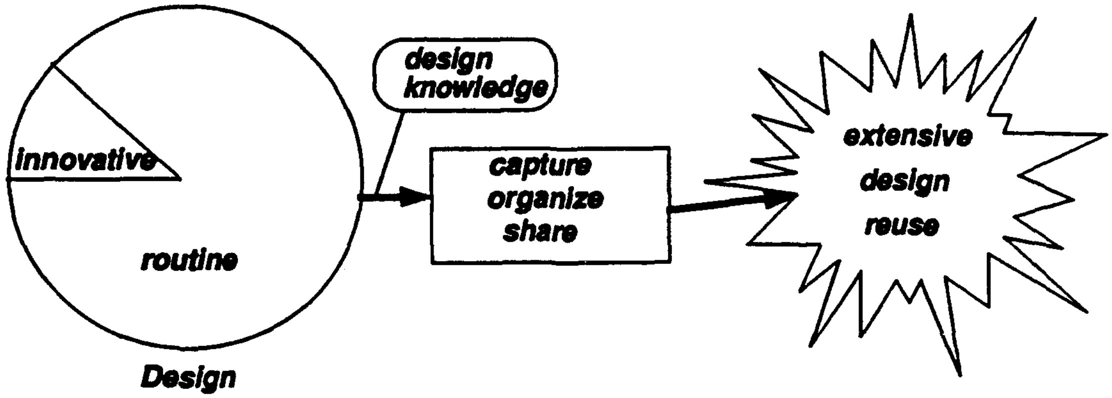
Scientific Foundation →
- identification, classification, description
- abstraction and analysis

Engineering Application →
- abstract and concrete models
- engineering and production techniques

Considerations in Practice →
- strategic/business considerations
- policy issues
- economic issues

This page intentionally left blank.

Engineering Design



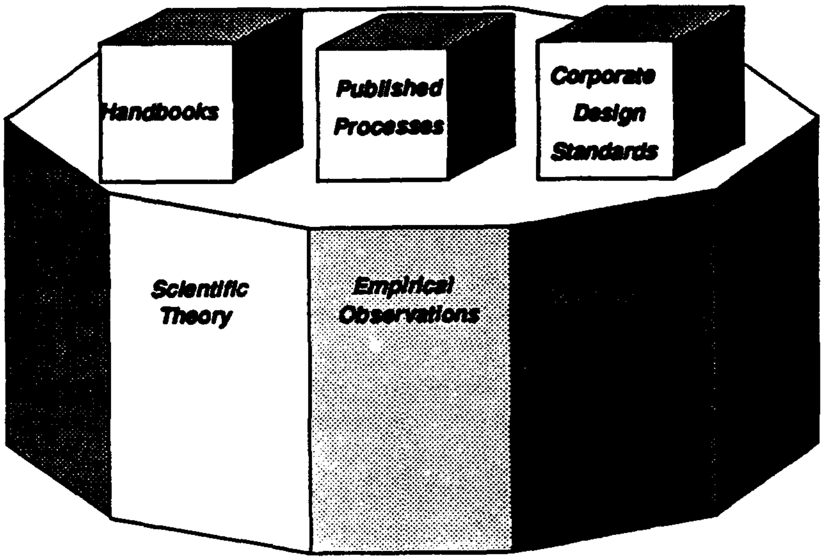
Engineering Design

In chemical engineering, as well as in other mature engineering disciplines, most design is routine rather than innovative (eventually even innovative designs become routine). Routine design involves solving familiar problems. The knowledge for routine design is captured, organized and shared within the engineering community. This leads to extensive design reuse.

References:

Shaw, M. "Prospects for an Engineering Discipline of Software" IEEE Software November 1990

Design Reuse in Chemical Engineering



Design Reuse in Chemical Engineering

The three main facilitators of design reuse in chemical engineering are handbooks, published processes (architectures), and corporate design standards. These are all based on empirical observations, scientific theory, and economics. We will look at these three facilitators in more detail.

Handbooks

Chemical Engineering		Software Engineering
One main handbook for the entire field	↔	Fragmented set of handbooks
Comprehensive coverage of unit operations	↔	Incomplete coverage of components/algorithms
Patterns of unit operations	↔	Few patterns
numerous heuristics	↔	some heuristics
over 100 authors	↔	One or a few authors
emphasis on economics	↔	processing/memory
common language - math and chemistry	↔	proliferation of languages and design notations - Ada, C, C++, Booch...

Handbooks

The one main chemical engineering handbook has more breadth and depth than existing software engineering handbooks (because the field is more mature). Unit operations (e.g. a heat exchanger, a distillation column) are the basic components in chemical engineering. A category of unit operations (e.g. heat exchangers) forms a horizontal domain (analogous to search algorithms or DBMSs in software). Most, but not all, software engineering handbooks deal with small grained components/algorithms (vs. large grained components like DBMSs) that are at a lower level of abstraction than unit operations.

Chemical engineering handbooks give patterns of how to put unit operations together in a process (see next slide). This is an important distinction. Software engineering is just beginning to capture and organize a wide range of information about patterns. Patterns in software may be more difficult to capture and organize.

It is interesting to note that the amount of expertise needed for a comprehensive chemical engineering handbook makes a large number of authors necessary. Also, the chem. eng. handbook emphasizes economics, whereas many software eng. handbooks only address processing/memory resources.

References:

Perry, Chilton "Chemical Engineers' Handbook" 5th ed. 1973

Knuth, "The Art of Computer Programming" vols. I-III 1973

Booch, "Software Components with Ada" 1987

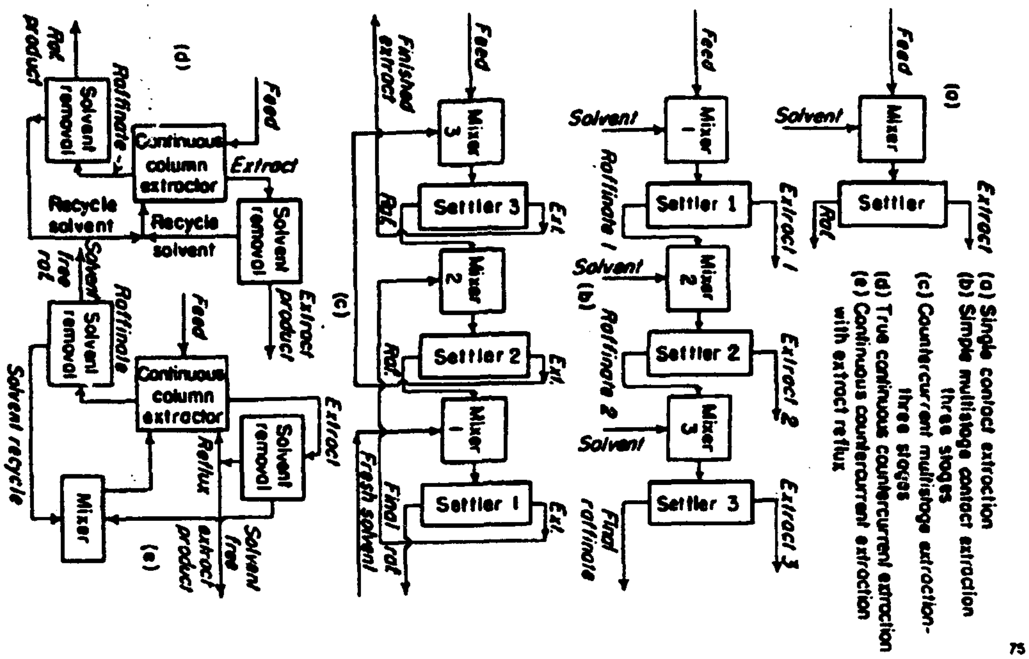
Sedgewick, "Algorithms in C" 1990 and "Algorithms in C++" 1992

Dumas "Designing User Interfaces for Software" 1988

Datapro "Reports on..." updated periodically

Barr, Feigenbaum, Cohen "The Handbook of Artificial Intelligence" vols. I-IV 1981 -1989

Patterns - Example: Liquid Extraction Systems



Patterns - Example: Liquid Extraction Systems

This slide shows a good example of what is meant by patterns of unit operations (components) that are contained in the chem. eng. handbook (right side of slide is actually the top). A discussion of heuristics and design trade-offs related to these patterns is also found in the handbook.

Reference: Perry, Chilton, "Chemical Engineers' Handbook", 5th ed. 1973

Published Processes

- Generic industrial processes (architectures) are published in:
  - handbooks
  - journals
  - patents
- Processes include:
  - constraints on choice/placement of unit operations
  - material flows
  - control: temperature, pressure, timing...
- Design steps:
  - refine generic process based on:
    - production rates
    - product and raw material specifications
  - do detailed design of unit operations
  - evaluate plant design by simulation/calculate return on investment

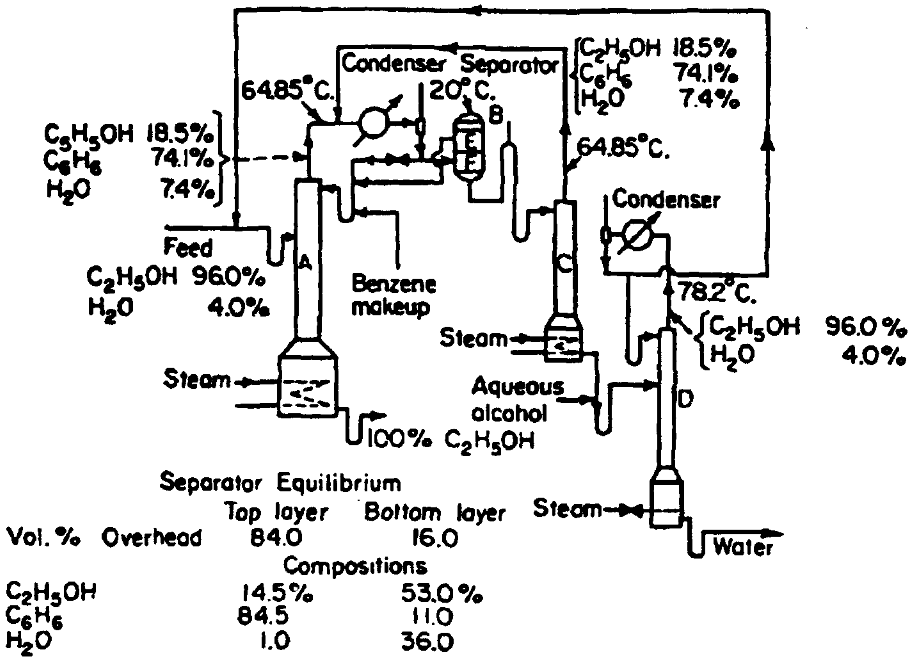
Published Processes

In chem. eng., industrial processes for producing chemical products are published more frequently and in more detail than in software engineering (note "industrial" -- many published system designs in software eng. are research prototypes). There is a widely known published catalog of processes that covers the entire spectrum of chemical process industries (I know of no equivalent for software eng. -- there are books that look at generic designs of one specific application area - e.g. compilers). Patenting a detailed chem. eng. process is common practice.

Notice the analogy of what a published chem. eng. processes includes to what is included in a software architecture (e.g. constraints on choice/placement of components, data flow, and control information (control is a major subfield of chem. eng.). See next slide for an example of a published process. Also notice the analogy of refining a generic design/architecture based on detailed requirements. This emphasizes the engineering mindset of composing solutions from past experience. Notice the lack of emphasis on calculating the return on investment for a software engineering design. Evaluating the composed system before it is built is also part of the engineering mindset.

Reference: Shreve, Brink "Chemical Process Industries", 4th ed. 1977

Published Process - Example: Alcohol Distillation



Separator Equilibrium

	Top layer	Bottom layer
Vol. % Overhead	84.0	16.0
	Compositions	
C_2H_5OH	14.5%	53.0%
C_6H_6	84.5	11.0
H_2O	1.0	36.0

Published Process - Example: Alcohol Distillation

Notice the choice and interconnections (architecture) of unit operations (component types), the material flow (data flow), and the temperatures (control information). Notice that each unit operation is treated as a black box (except the separator) so there is flexibility in choosing the size and exact internal design for the actual equipment (implementation components).

References: Perry, Chilton "Chemical Engineers' Handbook" 5th ed. 1973

Corporate Design Standards

- Management commitment to design reuse
- Captures and organizes experience/knowledge of corporate engineers
- Design standards include:
  - specific design equations
  - heuristics for:
    - design criteria for equipment "Avoid thin wall tubes"
    - parameter estimation
  - example calculations

Corporate Design Standards

These chem. eng. corporate design standards go beyond handbooks in helping to design unit operations (horizontal domains). These standards are used along with published processes (architectures) which are often supplemented by proprietary details. Can you imagine a set of corporate standard software components used in all systems across all application domains?

How Does This Apply to Software Architecture?

- How is community knowledge represented and shared?
- What are the architectures (product models)?
- What are the design processes?
- How does management demonstrate commitment to design reuse?

This page intentionally left blank.

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

Manufacture Perspective →
- production discipline and automation
- interchangeable parts and assemblies
- process control

Engineering Perspective →
- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

☞ **Architecture Perspective** →
- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

Scientific Foundation →
- identification, classification, description
- abstraction and analysis

Engineering Application →
- abstract and concrete models
- engineering and production techniques

Considerations in Practice →
- strategic/business considerations
- policy issues
- economic issues

This page intentionally left blank.

Obvious Analogies

Classical Architecture	Software Architecture
Blueprints, etc.: • plan, elevation, perspective • drawings/models, architect plans, shop plans	**Design Representations:** • multiple views • *models for differentiated roles* (customer, system engineer, software engineer)
Architecture styles: • Romanesque, • Gothic • Victorian	**Architecture styles:** • Distributed • Client/Server • Layered
Constraints: • circulation patterns • acoustics • air flow • lighting...	**Constraints:** • timing and schedules • reliability and fault tolerance • performance and throughput • data management and distribution

Obvious Analogies

Much has been made of the analogies between software architecture and classical (or "building") architecture. Some obvious analogies have been made between design notations used by software architects and building architects; *other analogies have been drawn between architecture idioms and recurring patterns* of software designs.

However, these analogies are of limited utility. For example, any discipline requiring problem solving where the information space relevant to the successful solution exceeds human short-term memory will involve specialized notations. This is also the case where multiple parties are involved in problem solving and production, in which case numerous specialized notations may be used.

Less obvious analogies can be drawn between the classical architecture and computer systems which are more revealing. For example, after centuries of practice, a few key families of constraints have emerged in the design of buildings, e.g., acoustics, circulation flow. These are areas of potential "misfits" between a design problem and its solution (in this case, a building). Similarly, in computer systems a number of families of constraints have likewise emerged—fault tolerance, security and human-machine interface ergonomics, for example, which can result in misfits between a system and its requirements.

The real benefits of understanding classical architecture as a precursor to studying software architecture is the relationship between classical architecture and a theory of design.

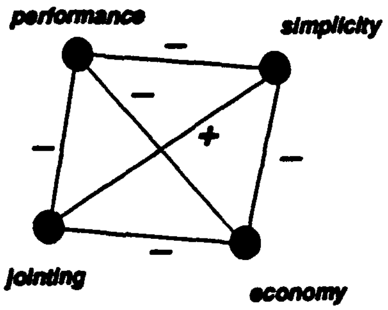
References:

Christopher Alexander, Notes on the Synthesis of Form, Harvard University Press, 1964. ISBN 0-674-62750-4

Dewayne Perry, Wolf, L., "Foundations for the Study of Software Architecture," Software Engineering Notes, Vol.17 No. 4, Oct. 1992.

Zachman, J., "A Framework for Information Systems Architecture," IBM Systems Journal, Vol 26, No. 3, 1987.

Classical Architecture Perspective on Design



A typical design problem[1]: requirements have to be met, and there are interactions among the requirements.

- **Quotes from[1] on the nature of design problems:**

[Information about requirements and their interactions] is hard to handle; it is widespread, diffuse, unorganized...the quantity of information itself is now beyond the reach of single designers [and] various specialists who retail it are narrow and unfamiliar with the form-maker's peculiar problems.

The average designer scans whatever information he happens on, consults a consultant...when faced by extra-special difficulties, and introduces randomly selected information into forms otherwise dreamt up in the artist's studio of his mind.

At the same time that the problems increase in quantity, complexity and difficulty, they also change faster than before. New materials are developed all the time, social patterns alter quickly, the culture itself is changing faster than it has ever changed before.

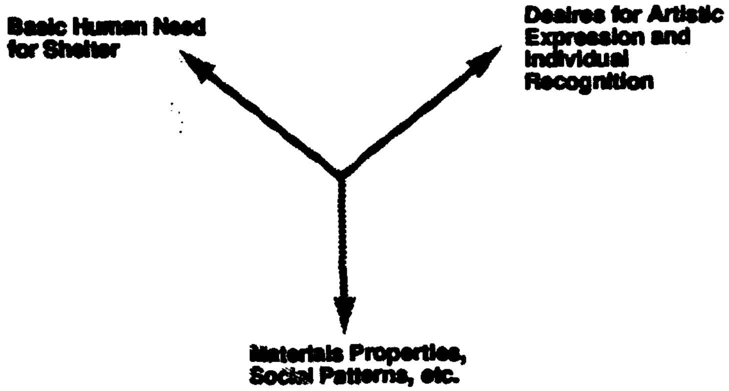
[1] Christopher Alexander, Notes on the Synthesis on Form, ISBN 0-674-62750-4

Classical Architecture Perspective on Design

Reading an overview of the design problem which Christopher Alexander is addressing is like reading an introduction to software/systems design textbook. Yet these are problems which classical architecture has been grappling with for centuries.

Reference: Alexander, C., Notes on the Synthesis of Form, pp. 2-4

Why Do Architects Introspect on the Design Process?


Basic Human Need for Shelter
Desires for Artistic Expression and Individual Recognition
Materials Properties, Social Patterns, etc.

Perhaps classical architecture represents the purest example of a discipline for controlling the creative design process

Architecture is considered an artistic discipline in addition to being an engineering discipline

What constraints are imposed on the urge for spurious creation?

Why Do Architects Introspect on the Design Process?

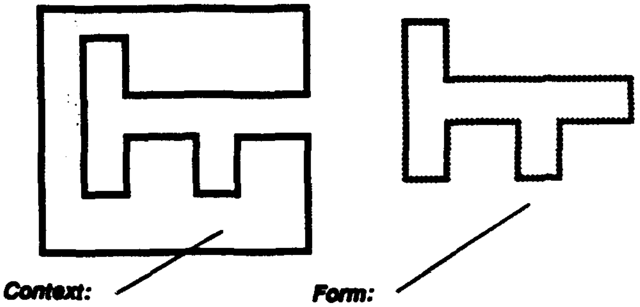
Public introspection is an important part of any mature professional discipline: it is what makes it possible for a community of practitioners to evolve the state of practice within a discipline.

The discipline of classical (or "building") architecture has a vast body of literature which deals with the nature of design. While other disciplines attend to the study of the design process, it is usually within the context of design methods—procedures and notations for representing and transforming the work products of problem solving. Classical architecture addresses these "syntactic" aspects of design, too. But the discipline also has a rich history of design theory bordering on mysticism, and certainly well into the realm of meta-physics.

This is probably true because the element of aesthetics plays a more overt role in classical architecture than in engineering. That is, while one may attain a Zen-like appreciation for the austere workings of a DC motor, such devises are not typically afforded appreciation as "works of art." This is certainly not the case in classical architecture, where a tension exists between the need to engineer a solution to the basic human need for shelter, while simultaneously satisfying additional cravings for artistic creation and individual distinction and recognition which accompanies classical architecture.

Architects study design because their problems are complex and ill-formed, their solutions must satisfy real needs and because there is a tendency for designers to engage in false creativity, non-essential creation and egotistical design—all of which interfere with achieving useful solutions.

The Nature of Design: The Context/Form Ensemble



Context:
- *that part of world we do not control*
- *represents the "problem"*

Form:
- *that part of world we can control*
- *represents solution to "problem"*

- A design problem consists of a two-part ensemble: a problem (context), and a solution (form).
- Form and context are inseparable and complementary.
- Design is an effort to achieve "good fit" between form and context. Fitness is a relation of mutual compatibility.
- It is impractical (perhaps impossible) to completely describe context—if it were possible there would be no design problem.
- What makes a design problem a problem is that we are attempting to create forms for contexts we do not completely understand or specify.

How does this apply to software architecture?

- *Reuse of architecture implies reuse of design*
- *Reuse frequently implies some adaptation*
- *"Form" of design depends upon complex "context" interactions*
- *Adaptation of the form (the architecture) makes sense only "in context"*
- *Seen in DSSA/ADAGE, ROSE-2,... as design records, design rationale...*

The Nature of Design: The Context/Form Ensemble

One of the most interesting, and useful, ideas which emerge from Alexander is the idea of a design ensemble, something which consists of both a context and a form as inseparable, complimentary aspects of the design problem.

An example of the complementary nature of context/form is the environment and a biological organism; natural selection is the mechanism by which we achieve a degree of "fit" between a form and its context (this fit is called "well adaptedness").

In software and systems, we typically refer to the "context" as the "requirements," and the "form" as the "design." The term "context" seems better, since it conveys the sense of ensemble better than does the term "requirements," though for most intents both terms are equivalent.

The idea of a complete context/form ensemble may seem obvious enough, but it is crucial in understanding reuse and software architectures. It will show up later in a variety of forms:

- component "qualification" is a measure of "fit" between a component and its context
- design reuse based on architecture refinement requires the encoding of context information to guide the refinement process

We should note that while the idea of a complete context/form ensemble makes a great deal of sense, in practice software and system "designs" are advertised as being reusable despite the fact that the context which produced the form is not included, has been discarded, or has never been formally documented to begin with.

References

Alexander, C., Notes on the Synthesis of Form, pp. 16-45.

Patterns: A System for Achieving Form/Context Fit

***Patterns*: context→conflicting forces→configuration**

the problem to be solved

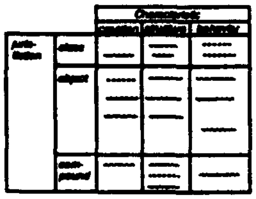
those characteristics of the problem which are known, and known to be in conflict

an arrangement of parts, or "form" which resolves the conflict

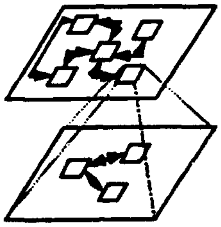
- Patterns are identified through observation
- Patterns are documented and held to public critique
- There are relatively few patte:ns

How does this apply to Software Architecture?

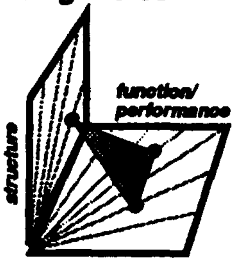
Gamma: Handbook of OO micro architectures



Shaw: Heterogeneous architecture idioms



Lane: Domain-specific design rules



Patterns: A System for Achieving Form/Context Fit

It should not be surprising that "patterns" should be an important concept in architecture, as this is an elementary prerequisite for learning, codifying knowledge and, ultimately, reuse. In the architectural sense, at least from Alexander's point of view, a pattern is a configuration of forms which bring conflicting forces into equilibrium. This notion of pattern crops up repeatedly in the study of software architecture:

Gamma, et. al., have identified recurring patterns in object-oriented systems, which he refers to as "micro" architectures. These are design abstractions, not code, which are used during object-oriented design, to fulfill specific needs within specific contexts.

Lane also searched for what amount to "patterns," or, what he referred to as design rules within a design space [illegible] idea was to uncover "design rules" which express structural solutions (i.e., implementation decisions) to interactions [illegible]tional/performance dimensions (e.g., response time vs. IPC means). These design rules are, in effect, patterns.

Finally, Shaw and Garlan have uncovered design idioms which have become widely used. While these idioms may be related to style (and may be style), when the idioms are composed they begin to look more like patterns.

What is significant in all of this is the search for and documentation of building block abstractions, or design elements, that work in practice.

Gamma, E., Helm, R., Johnson, R., Vlissides, R., Design Patterns: "Abstraction and Reuse of Object Oriented Design"—unpublished paper. Contact Erich Gamma at Taligent, Inc., 10725 N. De Anza Blvd., Cupertino, CA 95014-2000

Coad, P., "Object-Oriented Patterns," Communications of the ACM, Vol. 35, No. 9, September 1992.

Alexander, C., The Timeless Way of Building, Oxford University Press, ISBN 0-19-502402-8

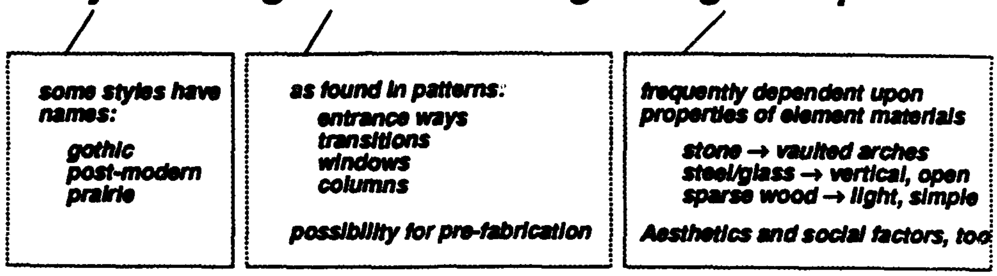
David Garlan, Shaw, M., "An Introduction to Software Architecture," to appear in Advances in Software Engineering and Knowledge Engineering, Vol I., 1993. World Scientific Publishing Company.

Lane, T. G., Studying Software Architectures through Design Spaces and Rules, CMU/SEI-90-TR-18, CMU, Pittsburgh, PA.

Architectural Style

Style refers to a quality of a solution which brings all of the design elements in an ensemble into a coherent whole.

Style = Design Elements + Organizing Principles



some styles have names:

gothic
post-modern
prairie

as found in patterns:

entrance ways
transitions
windows
columns

possibility for pre-fabrication

frequently dependent upon properties of element materials

stone → vaulted arches
steel/glass → vertical, open
sparse wood → light, simple

Aesthetics and social factors, too

How does this apply to software architecture?

- ***Can software architecture be expressed with a small, standard set of design elements? Are the design elements peculiar to a style?***
- ***Can a software architecture have a ubiquitous style?***

Architectural Style

There is a higher-level organizing principle than patterns and pattern languages called *architectural style* (although avid followers of Alexander might claim that pattern languages embody this organizing principle, and that the only "style" that matters is "patterns that live").

Some of the styles we refer to are known even to novices to architecture: the *Gothic* style, the *Post-Modern* style, the *American Prairie* style, etc. What constitutes a style is a combination of design elements and the manner in which the elements are related to each other. Some of the factors in selecting organizing principles are effected by the materials present in the design elements. For example, the use of stone or masonry leads to a very different organizational approach to relating, say, an entrance way to a large room, then will be the case if steel or wood are used.

What makes a style a style, of course, is that it represents a coherence among the design elements—this is what is meant by organizing principles. That is, we would not expect to see roman columns in front of an American Prairie home which uses reflective glass windows in steel frames.

This "definition" of style leads to a different applicability of style to software architectures than usually considered. That is, style in software architectures would relate more to the set of design elements used, and the manner in which those elements are related—not related in part, but related in the entire ensemble. That is, software architectural style—to be style—must describe system-wide organizational principles. Examples will be found in structural modeling and Genesis.

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

Manufacture Perspective →
- production discipline and automation
- interchangeable parts and assemblies
- process control

Engineering Perspective →
- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

Architecture Perspective →
- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

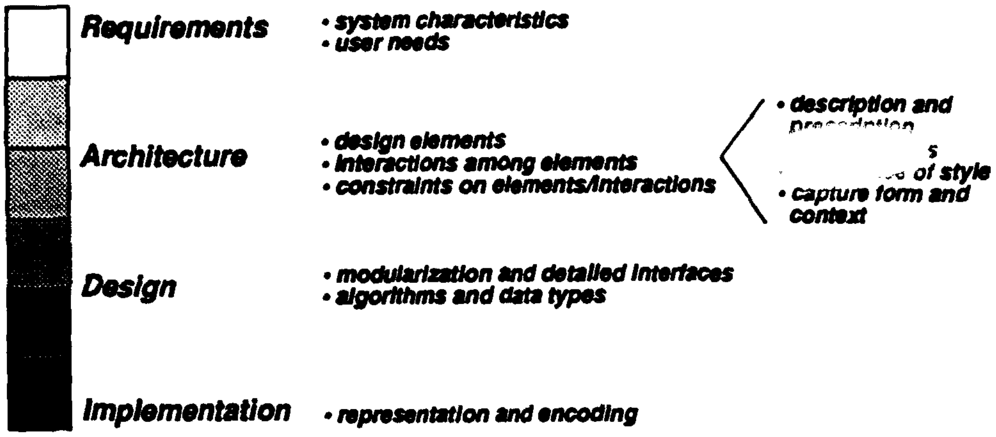
☞ **Scientific Foundation** →
- identification, classification, description
- abstraction and analysis

Engineering Application →
- abstract and concrete models
- engineering and production techniques

Considerations in Practice →
- strategic/business considerations
- policy issues
- economic issues

This page intentionally left blank.

Perry and Wolf: Context of Architecture



Requirements
- system characteristics
- user needs

Architecture
- design elements
- interactions among elements
- constraints on elements/interactions

- description and prescription
- [illegible] s [illegible] of style
- capture form and context

Design
- modularization and detailed interfaces
- algorithms and data types

Implementation
- representation and encoding

Perry and Wolf: Context of Architecture

A good place to start in understanding software architecture is the Foundations paper by Dewayne Perry and A. Wolf. We start here because Perry and Wolf make the strongest case for building on the analogy of classical architecture in the study of software architecture, particularly as concerned with the notion of architectural style.

The chart illustrates a starting point in the discussions: that software architecture is both a discipline of design, and also a representation of design. Specifically, software architecture as illustrated is a kind of high-level design. The key points of the Perry/Wolf paper are:

- architecture is a discipline with standards, codified styles and education
- architecture captures important high-level concepts in a system which must be preserved, and which make global assertions about the system
- multiple views are needed to express an architecture
- strong analogies are made between the notions of "style" in software and classical architecture

References

Dewcyne E. Perry, Wolf, A., "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4, October 1992, pp. 40-52.

Perry and Wolf: Elements, Form, Rationale and Views

Architecture =

Elements

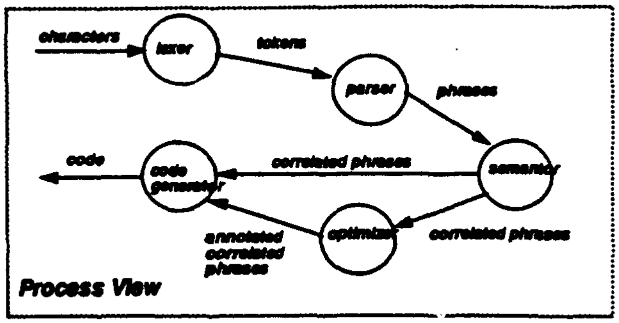
- processing elements
- data elements
- connecting elements

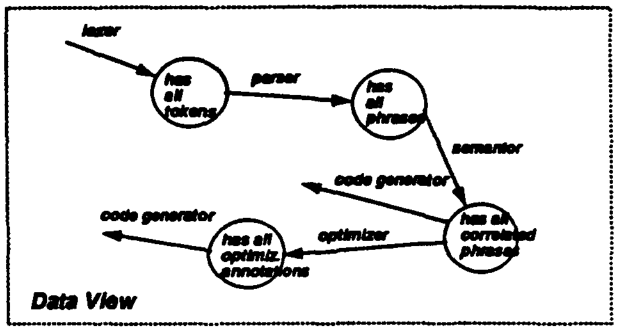
Form

- rules, weights which constrain placements of elements
- style + design

Rationale

- capture of rationale for selection of form
- links to requirements and to design
- functional/nonfunctional satisfaction





Perry and Wolf: Elements, Form, Rationale and Views

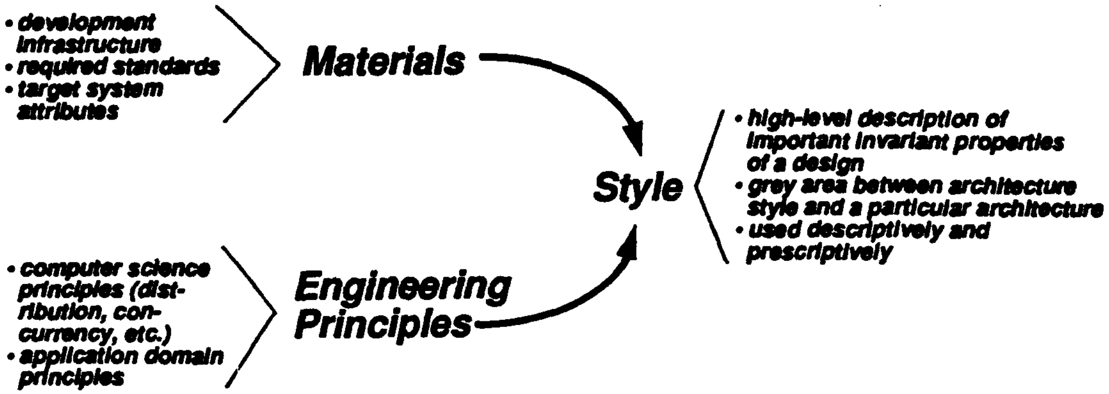
An architecture is comprised of elements, form and rationale.

Elements form the basis for various views: process, data and connectors. The figures illustrate two separate views for a canonical compiler: the connector view is implicit (s procedural/parameter connector view). Alternative process and data views emerge if alternative connector strategies are determined.

The notion of form parallels that of the discussion earlier in the classical architecture discipline. Form is concerned with constraints on the use and arrangement of various design elements. We should note that Perry and Wolf admit to some ambiguity between "style" and "design" decisions, indicating that there is some gray area between architecture style, architecture and design.

Note that rationale is also included. This relates strongly to the notion of architecture as a complete ensemble of context and form. In this case, additional rationale links are made between the form and its more detailed realizations in design.

Perry and Wolf: Constraints on and Nature of Style



- development infrastructure
- required standards
- target system attributes

Materials

- computer science principles (distribution, concurrency, etc.)
- application domain principles

Engineering Principles

Style

- high-level description of important invariant properties of a design
- grey area between architecture style and a particular architecture
- used descriptively and prescriptively

Perry and Wolf: Constraints on and Nature of Style

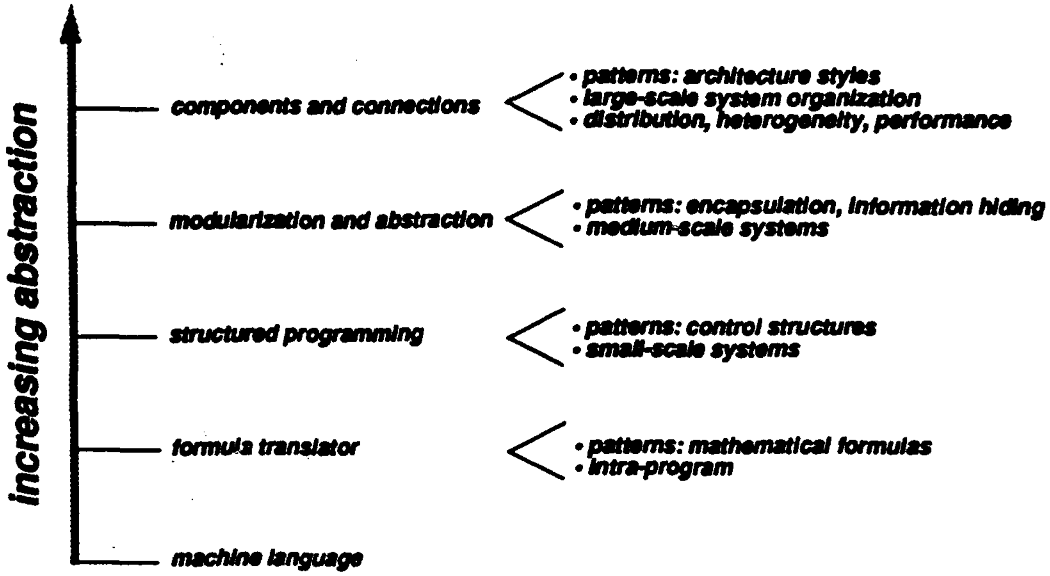
One of the most important points of the Perry/Wolf concept concerns the relationships between architecture style and materials and engineering disciplines.

In the context of software architecture, the following analogy can be made:

- style and materials: the selection of a style must take into account the kinds of components which may be reused or fabricated, the languages used to build and combine components, properties of the execution environment (network speed, processor speed, etc.).
- style and engineering principles: different computer science disciplines are involved in the use of different styles. A distributed and concurrent style will involve different principles than a simpler call/return style.

These considerations form part of the context for the form to be produced.

Shaw and Garlan: Context of Architecture



increasing abstraction

components and connections
- patterns: architecture styles
- large-scale system organization
- distribution, heterogeneity, performance

modularization and abstraction
- patterns: encapsulation, information hiding
- medium-scale systems

structured programming
- patterns: control structures
- small-scale systems

formula translator
- patterns: mathematical formulas
- intra-program

machine language

Shaw and Garlan: Context of Architecture

Shaw and Garlan are closer to the practice of architecture in their work than the Perry and Wolf paper. Although Shaw and Garlan shares the view of architecture as high-level design, they also consider the study of architectures to be a natural next-step in the evolution of computer science abstractions.

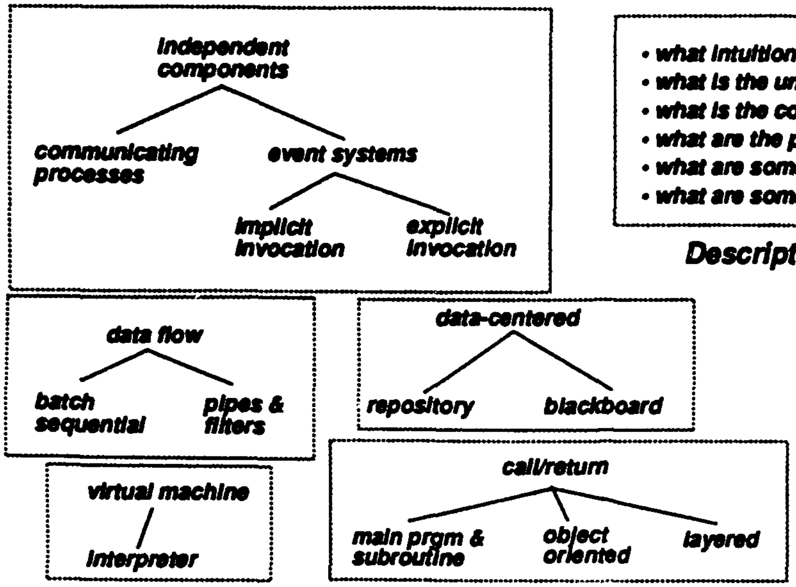
Again, using the metaphor of a pattern, we can see a certain historical trend towards the study of higher-level abstractions for larger-scale systems.

References

Shaw, M., Larger Scale Systems Require Higher Level Abstractions, 5th International Workshop on Software Specification and Design, May 1989.

Shaw and Garland: Taxonomy of Styles

Architecture Style =
{Component/Connector Vocabulary, Topology, Semantic Constraints}



independent components
communicating processes
event systems
implicit invocation
explicit invocation

data flow
batch sequential
pipes & filters

data-centered
repository
blackboard

virtual machine
interpreter

call/return
main prgm & subroutine
object oriented
layered

Taxonomic Framework

- what intuition does it capture?
- what is the underlying structural model?
- what is the computational model?
- what are the properties of the style?
- what are some common examples?
- what are some common specializations?

Descriptive Framework

Shaw and Garland: Taxonomy of Styles

Like Perry and Wolf, Shaw and Garlan define architecture in terms of constituent design elements and constraints on the elements. The exact definition is a bit different.

In this case, the elements are components and connectors, described in some idiom-specific manner. The particular idioms are represented as topologies of the component/connector vocabulary, along with constraints on how the topologies can be arranged.

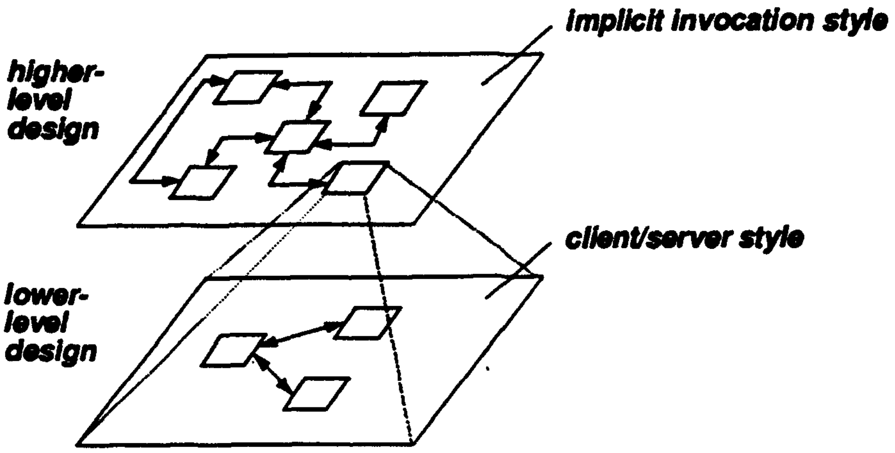
Shaw and Garlan have classified a number of idioms, and describe their general properties, etc. using a consistent descriptive framework. This taxonomy has emerged from case studies of actual systems. It is the foundation for courses taught at CMU on the topic of architecture and software design. it has also been widely published and distributed through technical literature and tutorials provided by Garlan and Shaw.

References

David Garlan, Shaw, M., "An Introduction to Software Architecture," to appear in Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

Shaw and Garland: Heterogeneous Styles

Systems need not be designed to only one style



higher-level design

implicit invocation style

client/server style

lower-level design

Shaw and Garland: Heterogeneous Styles

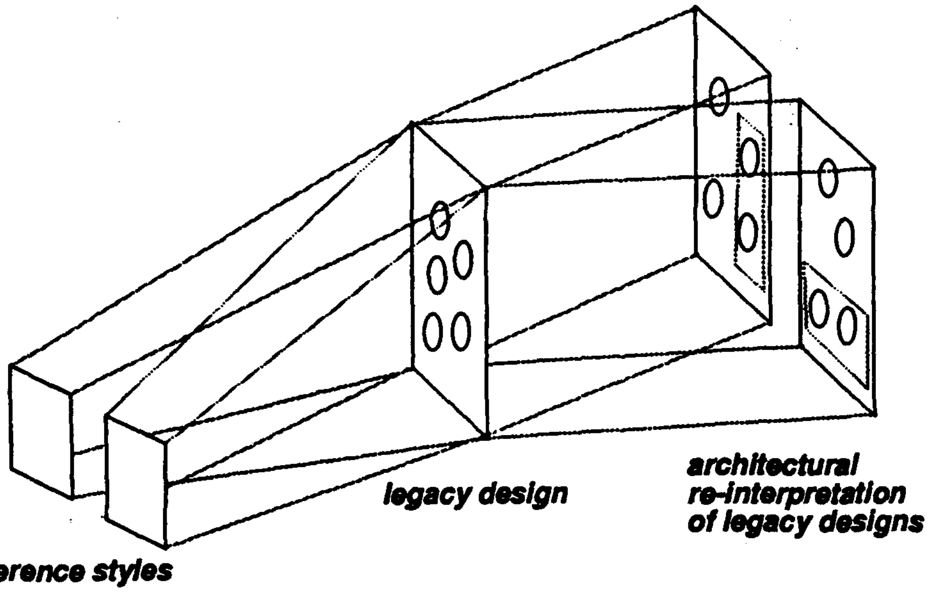
It is interesting to note that Garlan and Shaw have observed that systems do not usually consist of a single, consistent idiom that is used across an entire system. For example, they provide examples in case studies of systems which, at one level of abstraction present one idiom, while a single component within this idiom is realized through an entirely different idiom.

It is not clear whether this indicates the limits of the analogy made with traditional architecture—concerning the notion of style as a consistent, global property of a system. It may be that software systems are inherently "recursive" in design through many levels of abstraction, in which case "style" could be constrained to any one aspect or view of a system design.

References

David Garlan, Shaw, M., "An Introduction to Software Architecture," to appear in Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

Shaw and Garland: Styles as Reference Models



Shaw and Garland: Styles as Reference Models

Another interesting aspect of this work is the use of styles or idioms as a way of examining legacy designs. At least one case study is provided which illustrates how a system can be viewed from multiple idioms, and how each idiom reveals some characteristic about the system under observation.

The example illustrated is a natural language processing system viewed through the interpreter idiom and the blackboard idiom.

What is significant and worth noting is that this illustrates the usefulness of architectural abstractions in the analysis and understanding of properties of software designs.

References

David Garlan, Shaw, M., "An Introduction to Software Architecture," to appear in Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

Gamma, et.al.: Handbook of Object-Oriented Patterns

		Characterization		
		creational	structural	behavioral
Jurisdiction	Class	• Factory method	• Adapter • Bridge	• Template method
	Object	• Abstract Factory • Prototype • Solitaire	• Adapter • Bridge • Flyweight • Glue • Proxy	• Chain of responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy
	Compound		• Composite • Wrapper	• Interpreter • Iterator • Walker

— Intent
— Motivation
— Applicability
— Participants
— Collaborators
— Diagram
— Consequences
— Implementation
— Examples
— See Also...

Taxonomic Framework

Descriptive Framework

Gamma, et.al.: Handbook of Object-Oriented Patterns

Other researchers and practitioners have adopted a similar approach to Shaw and Garlan, but at a different scale. For example, this chart illustrates a fragment of a taxonomy of "micro" architectures found in object oriented systems. The term micro architecture is used by Gamma (one of the authors of the handbook) because the scale includes a configuration of objects and classes which would be combined with other micro-architectures to create an application. In contrast, the idioms of Shaw and Garlan "feel" larger grained.

Note that it is within the OO community that the largest direct use of concepts from Christopher Alexander are found. This might be because the OO community tends to be more avant guard, or it might be that the arguments made by Alexander—that the design elements of architecture must be closer to the physical world—have a natural setting in object-oriented design, which espouses a similar principle of abstraction forming.

With this we leave the science and philosophy of architecture behind, and examine some of the engineering factors—technology and process.

References

Gamma, E., Helm, R., Johnson, R., Vlissides, R., Design Patterns: "Abstraction and Reuse of Object Oriented Design"—unpublished paper. Contact Erich Gamma at Taligent, Inc., 10725 N. De Anza Blvd., Cupertino, CA 95014-2000

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

Manufacture Perspective →

- production discipline and automation
- interchangeable parts and assemblies
- process control

Engineering Perspective →

- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

Architecture Perspective →

- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

Scientific Foundation →

- identification, classification, description
- abstraction and analysis

☞ Engineering Application →

- abstract and concrete models
- engineering and production techniques

Considerations in Practice →

- strategic/business considerations
- policy issues
- economic issues

This page intentionally left blank.

Some Topics in Engineering Application

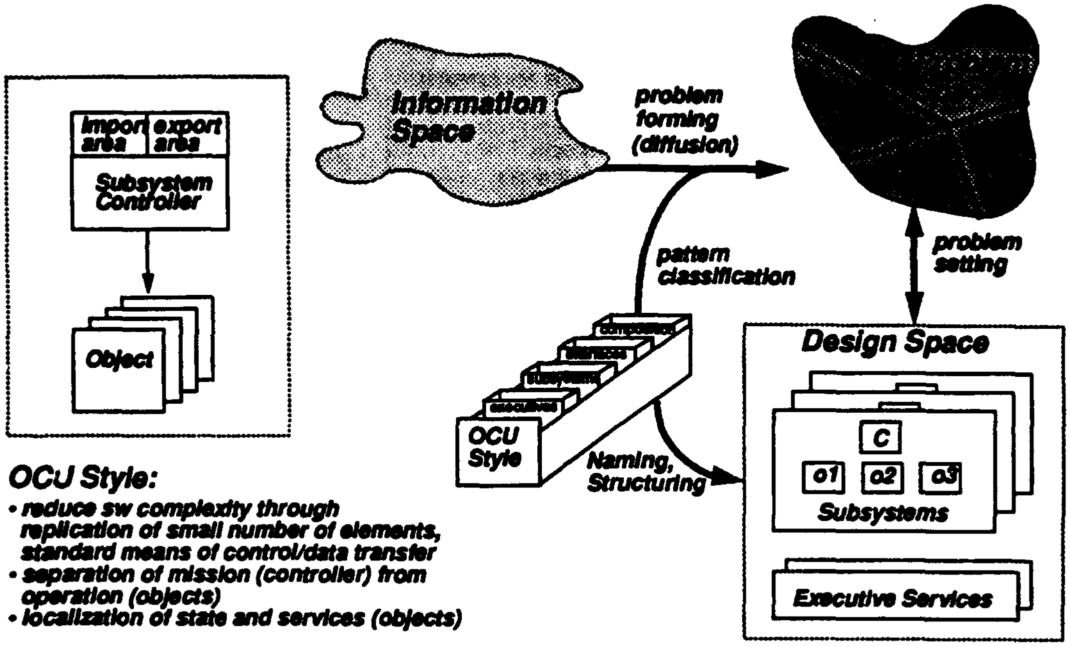
- Architectural Style and Formalized Design Elements
  - Style and Engineering Design
  - Style and Automation
- Design-Process Generated Design Elements
- Module Interconnection Formalisms
- Evaluation of Architectures

Some Topics in Engineering Application

There are quite a variety of topics. The following discussion touches on only a few important topics. Notably absent from the discussion are discussions on the relationships between architectural styles and design methods, impact of software architectures on life-cycle processes, relationship between structural versus behavioral descriptions in architecture, etc.

The topics which are addressed were selected: to amplify concepts introduced in the earlier discussions; to introduce some technology considerations which will be relevant in later discussions; and to provide ties wherever possible to ongoing software engineering efforts (both in theory and practice).

Architecture Style and the Engineering Design Process



OCU Style:

- ***reduce sw complexity through replication of small number of elements, standard means of control/data transfer***
- ***separation of mission (controller) from operation (objects)***
- ***localization of state and services (objects)***

Architecture Style and the Engineering Design Process

One illustration of the idea of consistent "style" in software architectures is provided by the OCU model: Object, Connect, Update. A thumbnail description of this "style" is provided. Essentially, the style is organized around the idea of subsystems, subsystem controllers and objects. It is an austere model which constitutes a style because it has a few primitive design elements, and rules for combining the elements.

The chart is meant to illustrate how an architectural style can be used within the context of an engineering process. First, by constraining the form of the solution so tightly, the style itself can serve as a tool for helping form the problem space during the problem forming process. That is, the style provides a kind of vocabulary for discussing the problem space. Similarly, once formed, the problem can be "set" in terms of the style as well.

Perhaps this is nothing more than the observation made by object-oriented designers in undertaking a kind of object-oriented analysis phase prior to design. On the other hand, the very restrictive style, if sufficient for the problem space, can be said to allow the software/system designer to focus creative energies where they are needed most, rather than on re-inventing structural or coordination models for each new problem.

The OCU style was used in practice as the basis for a flight simulator.

References

Putting the Engineering in Software Engineering, annotated briefing, Air Force Institute of Technology and the Software Engineering Institute, Carnegie Mellon University.

Lee, K, et. al., An OOD Paradigm for Flight Simulators, Technical Report CMU/SEI-88-TR-30, Software Engineering Institute.

Abowd, et. al., Structural Modeling: An Application Framework and Development Process for Flight Simulators, Technical Report CMU/SEI-93-TR-14, Software Engineering Institute, CMU, Pittsburgh, PA.

Architectural Style and CASE Tooling

Defined Design Elements and Constraints (Style) Permits Automation

CASE Tool	Style	Design Elements	Automated Services
UNAS/SALE	• Independent Objects	• Tasks • Message • Sockets • Connections • Processes • Process Groups	• Network Instrumentation • Test Scenario Injection • Graphical Design Tools
SARA	• Independent Objects	• Modules • Sockets • Interconnections • Nodes • Control arcs • Tokens • Processors • Datasets • Data arcs • Read/write • Delay/output	• Correctness Analysis • Performance Evaluation • Graphical Design Tools

Architectural Style and CASE Tooling

The previous chart illustrated the role that architecture style can play in the engineering process. It is also the case that defining an architecture style—identifying design elements and rules for combining these elements—provides opportunities for automation. Only two of many possible instances are illustrated here: UNAS/SALE, a commercial product marketed by TRW, and SARA, a well-known research system.

In each case, these systems are constructed on a foundation of a few primitive elements, and larger systems can be specified and executed. Other tools include the micro-Rapide language/system being developed as part of the ARPA/ProtoTech project, and various other tools for specifying properties of architectures.

Incidentally, although there are many design tools which provide primitives for describing characteristics of system designs, the term "architecture description lang‿age" tends to apply to only those notations that describe components and component interactions (further evidence of the appropriateness of the Shaw and Garlan perspective on software architectures).

References

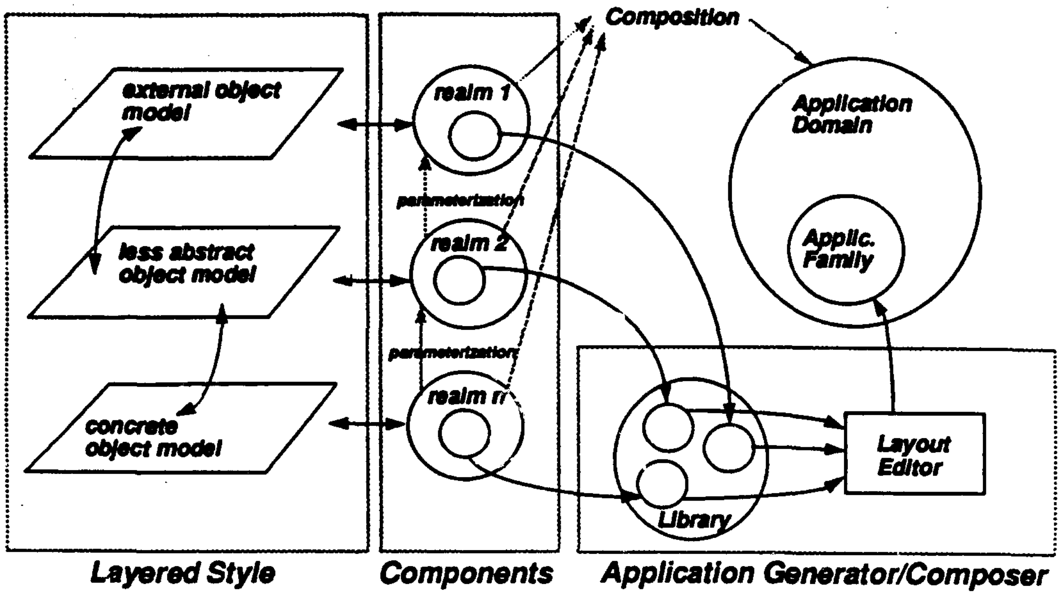
Walker Royce, Brown, D., "Architecting Distributed Realtime (sic) Ada Applications: The Software Architect's Lifecycle Environment," Ada IX, 1991. (Contact: Walker Royce, TRW Systems Integration Group. 213-764-3224)

Gerald Estrin, Fenchel, R., Razouk, R., Vernon, M., "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 293-311.

Harel, D., et.al., STATEMATE: A Working Environment for the Development of Complex, Reactive Systems, Technical Report, i-Logix Inc., Burlington, MA 01803.

David Luckham, Vera, J., "µRapide: An Executable Architecture Definition Language," April 7, 1993.

Batory: Design-Method → Architecture Style



Batory: Design-Method → Architecture Style

Architecture-level automation does not always appear to depend upon pre-definition of a small number of design elements. Batory has demonstrated application-specific generation/composition based upon software architectures in non-trivial application domains.

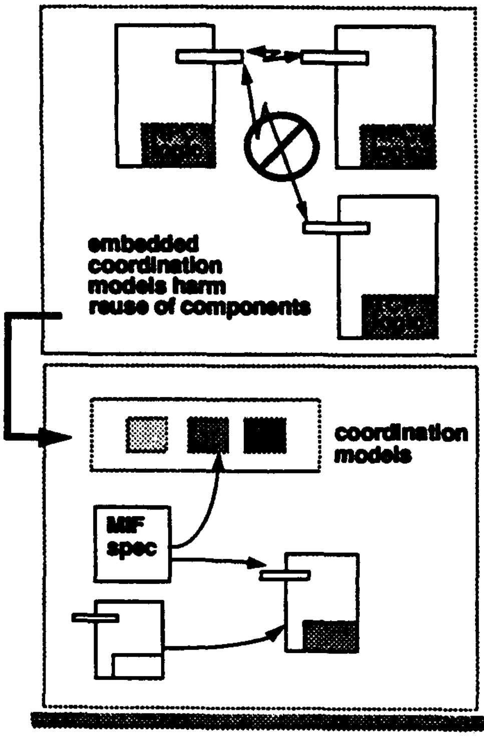
In this case, the architectural style is said to be layered, but there are no further design primitives for describing these layers beyond those reflected in the interfaces to components which result from a domain engineering/domain design process. That is, rather than defining primitive design elements for describing software architectural abstractions, Batory et. al. have defined a design process for producing components which have certain, constrained properties. It is these properties which allow automation and generation of applications from the design/architecture.

In this method, components are aggregates of classes and objects which implement what amounts to a "subsystem," with each component representing a specific layer in a layered architecture. The type model implemented by these higher-level (component) abstractions allows higher-levels of the design to be parameterized by lower levels.

References

Don Batory, O'Malley, S., The Design and Implementation of Hierarchical Software Systems with Reusable Components, Technical Report TR-91-22, University of Texas at Austin, Texas 78712-1188, January 1992 (revised).

Architecture and Module Interconnection Formalisms



embedded coordination models harm reuse of components

coordination models

One form of module interconnection formalism addresses the need to separate coordination from function

The need is especially strong in reusing components where systems will vary by distribution and heterogeneous platforms

Examples: Polylith, Linda

Another form of module interconnection formalism addresses higher-level semantics of component composition

Examples: LILEANNA, P++

Architecture and Module Interconnection Formalisms

In most cases an important consideration in software architectures is how concrete software components can "fit." A question concerning the relationships between software components and architectures arises where feature binding time is concerned. Especially where reuse is concerned, architecture reuse implies some flexibility in selecting application features. If components prematurely embed certain features the probability of reusing these components is decreased.

One frequently-encountered problem is that code, especially for distributed systems, embeds coordination logic which is arcane and makes the code non-reusable. Since the "connections" among components at an architecture level may imply coordination models, it would be nice to have the means of separating these coordination models from the underlying components—that is one purpose for MIFs.

A second purpose concerns the manipulation of software components as design elements in their own right. To some extent this is already possible with object-oriented languages (although Batory has noted some limitations along these lines.) MIFs which extend the encapsulation/abstraction of programming language modules to support a more flexible composition at design-time would be nice. Languages such as LILEANNA and P++ are designed with these kinds of issues in mind, and allow for combining modules, adding, removing and hiding capabilities of modules, parameterizing modules with other modules, and so on.

David Gelerator, Carriero, N., "Coordination Languages and their Significance," Communications of the ACM, Vol. 35 No. 2, 1992.

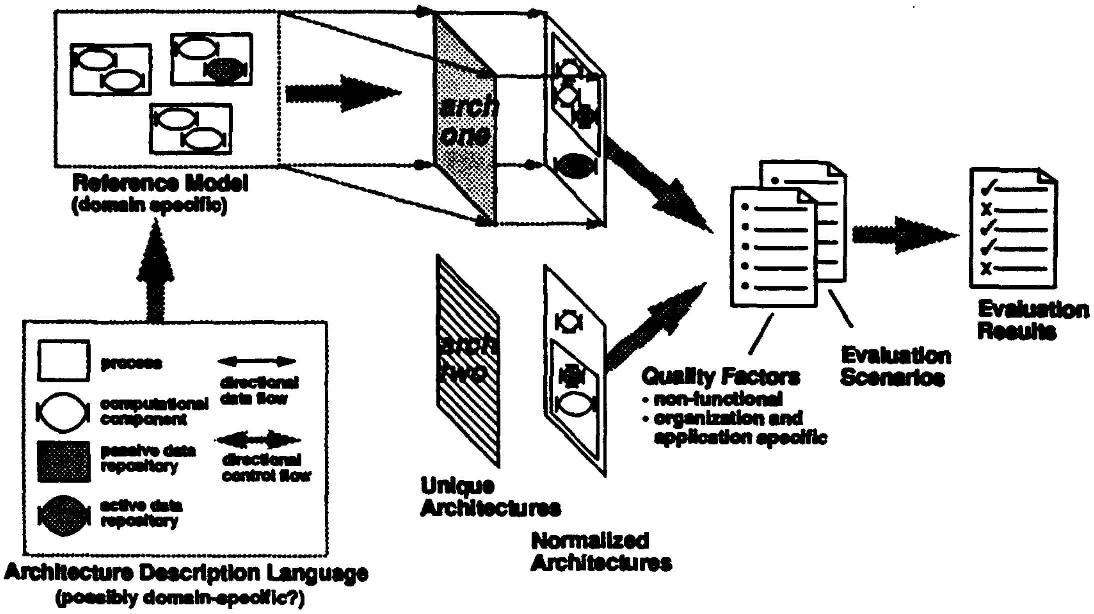
John Callahan, Purtillo, J., "A Packaging System for Heterogeneous Execution Environments," IEEE Transactions on Software Engineering, Vol. 17 No. 6, June 1991.

Vivek Singhai, Batory, D., P++: A Language for Software System Generators, Technical Report TR-93-16, Department of Computer Science, University of Texas at Austin, 1993.

Will Tracz, "Parameterized Programming in LILEANNA," unpublished, IBM Federal Systems Company.

OMG "The Common Object Request Broker: Architecture and Specification" 1992

SEI:SAAM—Software Architecture Analysis Method



SEI:SAAM—Software Architecture Analysis Method

Of practical concern is whether and how we can go about evaluating the qualities of software architectures. There are specific "metrics" available for assessing quality factors of source code—modularity, complexity, etc., and perhaps there are measures that could apply to behavioral characteristics of a system—data throughput, mean response time, mean-time to failure, etc. But, practically speaking, how does one evaluate the relative "goodness" of architectures?

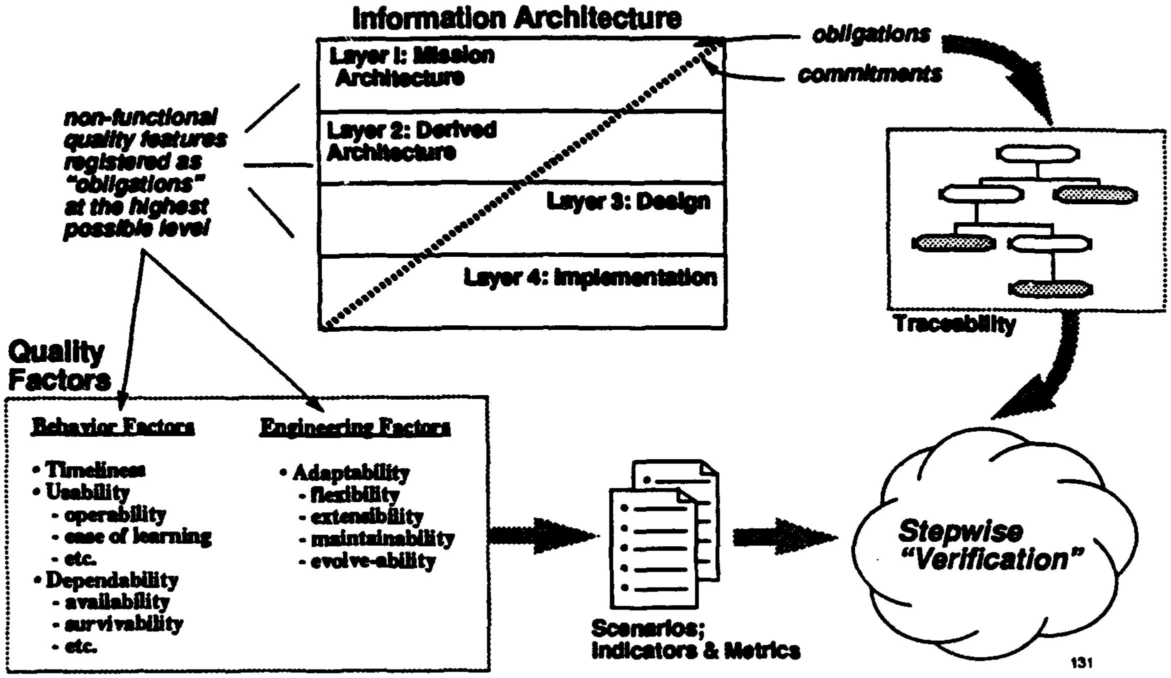
The Software Architecture Analysis Method has some features worthy of note. First, there is an inversion of the Garlan/Shaw concept of examining a design from the perspective of multiple styles. In SAAM multiple designs are examined from the perspective of a single reference model. The reference model is a canonical functional partitioning of application functions—it looks like a high-level *domain-specific design.*

The second interesting feature is the use of an architecture description language (ADL). In conjunction with the reference model, individual "unique" architectures can be "profiled," are in effect re-cast in terms of the reference model and the ADL. In this way disparate, unique designs are "normalized" to a common linguistic framework. Note that the ADL used is focused on structural aspects of the design; specific behavioral description is limited to the idea of "control flow" and "process." The design of the ADL may have been influenced by the application domain: the differentiation of "active" from "passive" repository" seems to indicate the influence of one or more representative architectures within the domain being studied.

The third interesting feature is that quality factors are selected, along with specific scenarios which exercise the quality factors. Note that the quality factors are focused on so-called non-functional system characteristics: in the paper these factors were focused on various dimensions of system adaptability. Kazman, et. al. deem the quality factors to be relevant to a specific organizational context, not necessarily to the application domain. Other non-functional quality factors may be of use in different contexts.

Kazman, R., Bass, L., Abowd, G., Webb, M., Analyzing Properties of User Interface Software, to be released as a Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA.

SEI: Information Architecture and Non-Functional Analysis



Information Architecture

Layer l: Mission Architecture

Layer 2: Derived Architecture

Layer 3: Design

Layer 4: Implementation

obligations

commitments

non-functional quality features registered as "obligations" at the highest possible level

Traceability

Quality Factors

Behavior Factors

- Timeliness
- Usability
  - operability
  - ease of learning
  - etc.
- Dependability
  - availability
  - survivability
  - etc.

Engineering Factors

- Adaptability
  - flexibility
  - extensibility
  - maintainability
  - evolve-ability

Scenarios; Indicators & Metrics

Stepwise "Verification"

SEI: Information Architecture and Non-Functional Analysis

A draft paper by Salasin of the SEI on analysis of non-functional characteristics of architectures for the Ballistic Missile Defense Organization (BMDO) Battle Management/Command Control Communications (BM/C3) System discusses process and representation issues of ensuring satisfaction of non-functional quality features. Instead of post-mortem evaluation of critical quality factors the approach described builds "satisfaction" into the architecture refinement process and architecture representation. Some notable points:

1) The "information architecture" reflects a complete design ensemble (context/form, here expressed as problem space/solution space). The "mission architecture," for example, models the operational requirements (the "shall") as well as the concepts of operation.

2) Non-functional qualities:

- are made explicit in the form of "indicators;"
- are tied to objects in the information architecture;
- are used to define scenarios for evaluation/verification purposes (similar to SAAM);
- have metrics associated for quantitative evaluation of indicators (i.e., did the commitment satisfy the obligation?)

3) The process for managing the non-functional requirements is step-wise, and can be integrated with existing design reviews.

References

John Salasin, Waugh, D., "An Approach to Analyzing Non-Functional Aspects During System Definition," Draft Technical Paper, in Proceedings of the ARPA/DSSA VII Workshop.

Roadmap for this Session

Architecture: Multi-Disciplinary Overview

Manufacture Perspective →
- production discipline and automation
- interchangeable parts and assemblies
- process control

Engineering Perspective →
- engineering discipline
- codified knowledge in engineering models
- predictable results through composition

Architecture Perspective →
- design discipline
- form and context: bounds on creativity
- design patterns and "style"

Software Architecture: Overview

Scientific Foundation →
- identification, classification, description
- abstraction and analysis

Engineering Application →
- abstract and concrete models
- engineering and production techniques

☞ **Considerations in Practice** →
- strategic/business considerations
- policy issues
- economic issues

This page intentionally left blank.

Practical Considerations

- **System v. software engineering and binding time of design decisions...**
- **Procuring architectures without over- or under-constraining the form (reference models, tools and representation standards)...**
- **How to allow technology progression and introduction of new, more optimal solutions (architecture life cycle)...**
- **Re-engineering and architectures—migration and interoperation of legacy systems...**
- **Ownership and rights...**
- **Domain engineering and domain management...**

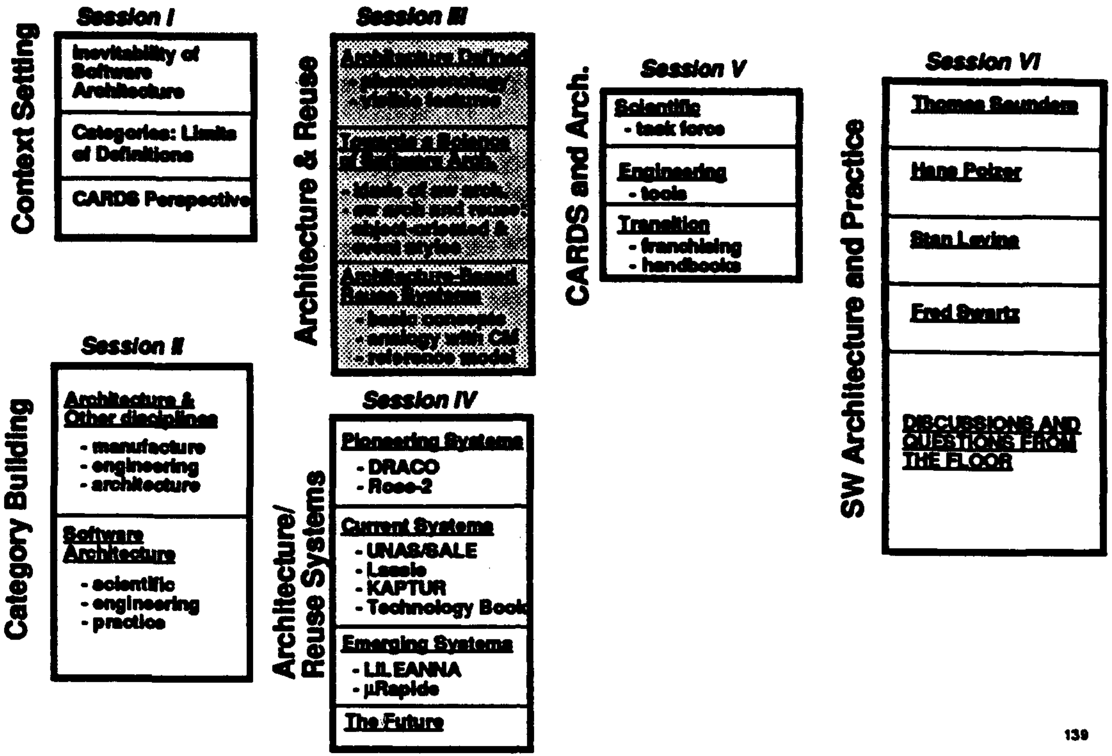
AND... Much Much More. The Workshop is intended to identify issues from the perspectives of engineering practitioners, program managers, policy makers and other stakeholders.

This page intentionally left blank.

Summary of "Senses of Architecture"

- There are a diversity of perspectives on what is "important" in the study of software architecture
- There are interesting and useful analogies in the areas of manufacturing, classical engineering and classical architecture
- The computer science and software engineering foundations are not mature
- There are a range of practical considerations for the adoption of software architecture in the DoD

This page intentionally left blank.



Context Setting

Session I

Inevitability of Software Architecture

Categories: Limits of Definitions

CARDS Perspective

Category Building

Session II

Architecture & Other disciplines
- manufacture
- engineering
- architecture

Software Architecture
- scientific
- engineering
- practice

Architecture & Reuse

Session III

Architecture Defined
- [illegible]
- [illegible] features

Towards a Science of Software Arch.
- kinds of [illegible] arch.
- [illegible]
object-oriented & [illegible]

Architecture-Based Reuse Systems
- basic concepts
- analogy with CM
- reference model

Architecture/ Reuse Systems

Session IV

Pioneering Systems
- DRACO
- Rose-2

Current Systems
- UNAS/SALE
- Lassie
- KAPTUR
- Technology Book

Emerging Systems
- LILEANNA
- µRapide

The Future

CARDS and Arch.

Session V

Scientific
- task force

Engineering
- tools

Transition
- franchising
- handbooks

SW Architecture and Practice

Session VI

Thomas Saunders

Hans Polzer

Stan Levine

Fred Swartz

DISCUSSIONS AND QUESTIONS FROM THE FLOOR

This page intentionally left blank.

Central Archive for Reusable Defense Software (CARDS)

Session III
Software Architecture and Reuse

16 November 1993

This page intentionally l·'ft blank.

Roadmap for this Session

☞ **Architecture "Defined"** →
- phenomenology
- externally visible qualities of architecture: a hypothesis

Towards a Science of Architecture →
- kinds of architectures

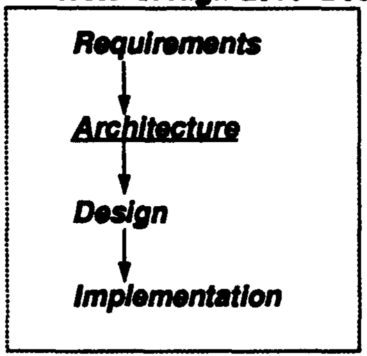
Trends in Architecture for Reuse →
- object-oriented architectures
- event-based architectures
- object-oriented/event hybrids

Architecture-Based Reuse Systems →
- overview of concepts
- analogy with configuration management

This page intentionally left blank.

Two Key Questions in the Search for Architecture

Architecture: High-Level Design



Requirements

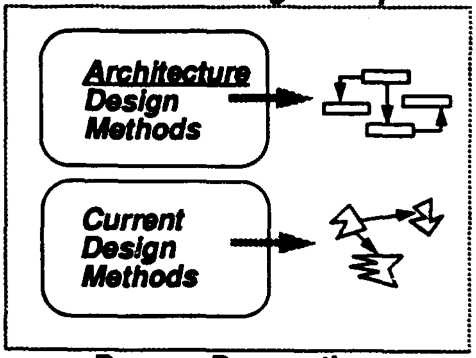
Architecture

Design

Implementation

Product Perspective

Architecture: Design Discipline



Architecture Design Methods

Current Design Methods

Process Perspective

If this is valid, then the question:

Does every design imply an architecture?

If this is valid, then the question:

How do "architected" designs differ from non-architected designs?

Two Key Questions in the Search for Architecture

Session two of the seminar covered many different perspectives on the topic architecture. We are in a position of hypothesizing about the structure of the conceptual category "architecture." This is not the same as providing an axiomatic definition. Instead, we will adopt a phenomenological approach: based on the concepts we have highlighted earlier, can we identify what characteristics we might observe of software architectures?

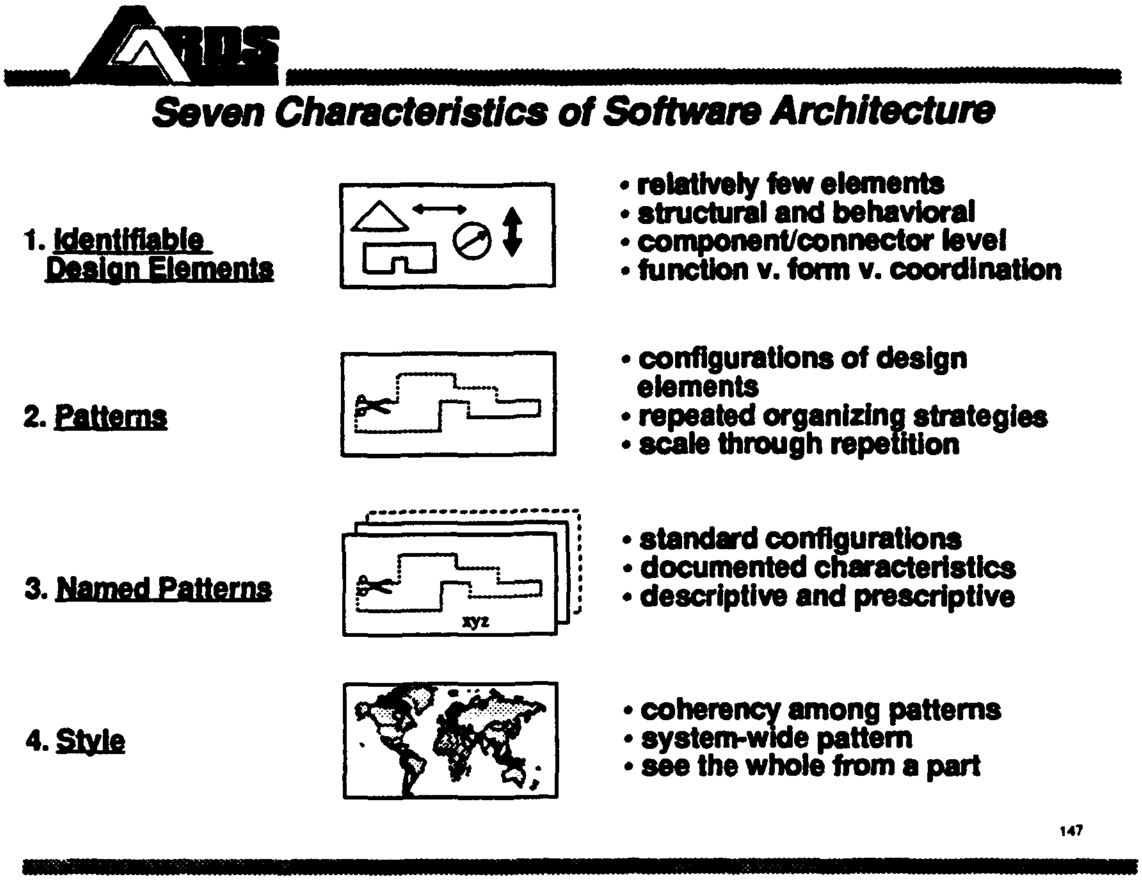
Before we do so, two premises need to be established, and two derivative questions proposed, to justify a phenomenological approach. Note that only one of the premises need to be true, although both could be true, for a phenomenological approach to be reasonable (although our notions of architecture phenomena might still be invalid).

1. If it is valid that software architecture is a high level design, then is it true that all designs have an architecture? We believe that not all designs are "architected" designs, in the same way that not all programs are structured programs.

2. If it is valid that architecture is a discipline of design, then is it true that the forms produced by the process will be different from the forms produced by a non-architectural design discipline? We believe that not all design processes are based on principles of architecture, and that, in general, current design processes do not produce architected designs.

If you accept the premises, the questions and our answers, then it is reasonable to ask whether, in theory, one could observe differences between architected and non-architected forms (i.e., designs). If there are no observable forms, then why study software architecture? If there are differences, what are they?

The following seven characteristics of software architecture need not be considered as a rigid statement. It is not clear that all elements need to be present (in the same way that a three-legged elephant is still an elephant). And, naturally, there may be characteristics which we have not included.



Seven Characteristics of Software Architecture

1. Identifiable Design Elements
   - relatively few elements
   - structural and behavioral
   - component/connector level
   - function v. form v. coordination

2. Patterns
   - configurations of design elements
   - repeated organizing strategies
   - scale through repetition

3. Named Patterns
   - standard configurations
   - documented characteristics
   - descriptive and prescriptive

4. Style
   - coherency among patterns
   - system-wide pattern
   - see the whole from a part

Seven Characteristics of Software Architecture

NOTE: We do not claim that all characteristics must be present, or that this represents a comprehensive set of characteristics. We believe all of these elements may be observed in architected designs.

1. Identifiable Design Elements. As we noted earlier, one characteristic of architectures is that they may be represented in terms of so-called architecture description languages (ADLs). There are various computer-aided software engineering (CASE) tools which claim to be "architecture" tools, and they have codified abstractions, rules for composing specifications from these abstractions, and environments for simulating/executing/evaluating these specifications. The SEI Object/Connect/Update (OCU) "style" also has *identifiable design elements: objects, controllers, import/export areas, etc.* Note that *architecture design elements should pertain to the structure and behavior of systems at the component/connector level of abstraction. It should be possible to separate application functionality from structure, and structure from coordination among structural elements.*

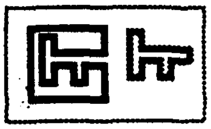
2. Patterns. Patterns may be reflected in the types of design elements and composition rules, and in specific configurations of design elements. However, patterns are not dependent upon specialized, architecture-level design elements—they can be reflected in the properties of implementation elements such as code components, modules. For example, type properties presented by component interfaces which are generated by a design method also represent architectural patterns.

3. Named Patterns. Patterns should have sufficiently regular and predictable form to be recognized and documented. The features of the pattern, its strengths and weaknesses, and the contexts for the use of the pattern, should be apparent in the pattern definition. The patterns should be descriptive, i.e., support understanding, and prescriptive, i.e., support reasoning.

4. Style. Style refers to a system-wide pattern, or the application of principles which bring about a state of coherency among the patterns used in a design. Styles should also be name-able, and permit description and prescription analogously to named patterns, but at a systems level.

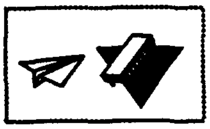
Seven Characteristics of Software Architecture (Cont.)

5. Complete Context/Form Ensemble



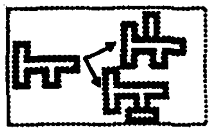
- problem and solution space
- alternatives and rationale
- reason about context from form

6. Tied to Physics



- general laws: mathematics
- application-specific physics
- material constraints: hardware

7. Adaptable Form



- form optimized for anticipated changes
- resilience to drift and erosion

Seven Characteristics of Software Architecture (Cont.)

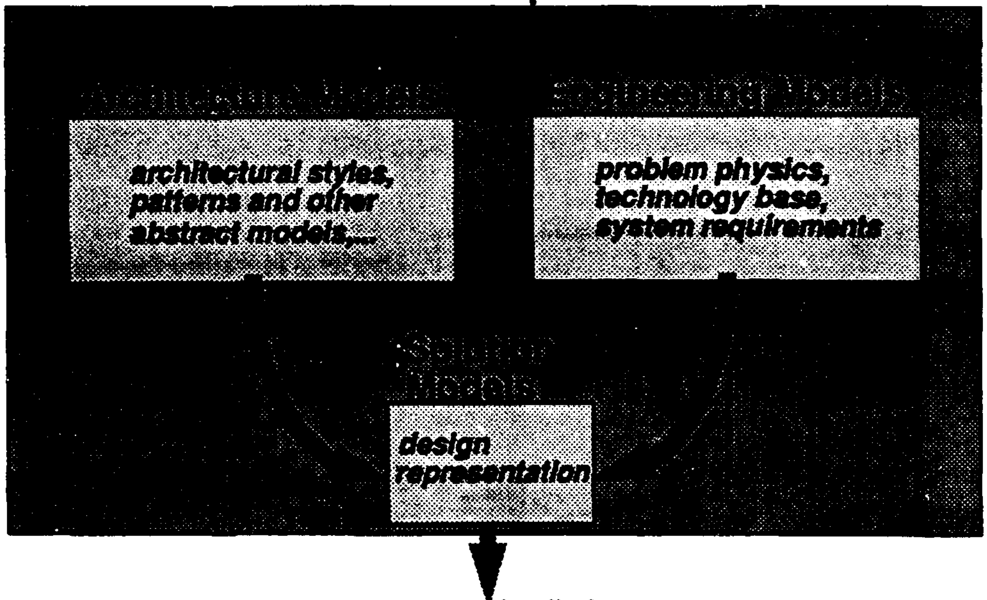
5. Complete Context/Form Ensemble. As noted earlier a design problem consists of a context and a form. The idea of linking the form to context appears repeated—in Perry and Wolf's definition, in Salasin's information architecture, and as will be seen where-ever design-level reuse is anticipated. We can think of the following two characteristics as revealing different aspects of a design ensemble.

6. Tied to Physics. In the engineering discipline the laws of nature define the boundaries of problems and solutions. There are equivalent laws of nature in the problems and solutions of software systems. As virtual machines, software depends upon the mathematics of computation—it is hoped that as the discipline of design and architecture mature, more formal, mathematical reasoning about designs will become commonplace (temporal logics, type logics, calculus of communicating systems, etc.). Designs need also be tied to the practice of engineering within an application area—designs for control systems may look different from designs for information management systems. Finally, there are materials physics—virtual machines are implemented on real machines which define physical constraints on software solutions. All of these factors represent part of the "context" for a design.

7. Adaptable Form. This may be the most important characteristic: it should be possible to reason about the adaptability of the design from its form. As already observed, the context for software is constantly changing, and changing at an increasingly fast pace. The missions for software are becoming more complex, and the capabilities of hardware are pushing (or are being hindered by) software capabilities.

A Note on Concept and Terminology

Software Architecture Discipline


Architecture Models
Engineering Models
architectural styles, patterns and other abstract models,...
problem physics, technology base, system requirements
Solution Models
design representation

more detailed solution models

A Note on Concept and Terminology

The myriad uses of the noun "architecture" is sometimes confusing—overuse may result in a degenerate vulgarization of important concepts. It should be possible to more cleanly differentiate the concepts of "the design" from "the architecture."

One possible partitioning strategy is illustrated on the chart. In it we establish the notion that architecture *is about producing designs. There are (at least) two disciplines involved: one involving the structuring of* software, the other involving the application of engineering "know how" in problem solving. The structuring of software involves computer science and software architecture, the engineering "know how" involves engineering problem-solving approaches, disciplines and domain/application expertise.

With this viewpoint the question "what is your architecture" is more clearly directed towards application-independent structuring and styling issues, while "what is your design" is more clearly directed towards the specified solution.

Roadmap for this Session

Architecture "Defined" →
- phenomenology
- externally visible qualities of architecture: a hypothesis

☞ Towards a Science of Architecture →
- kinds of architectures

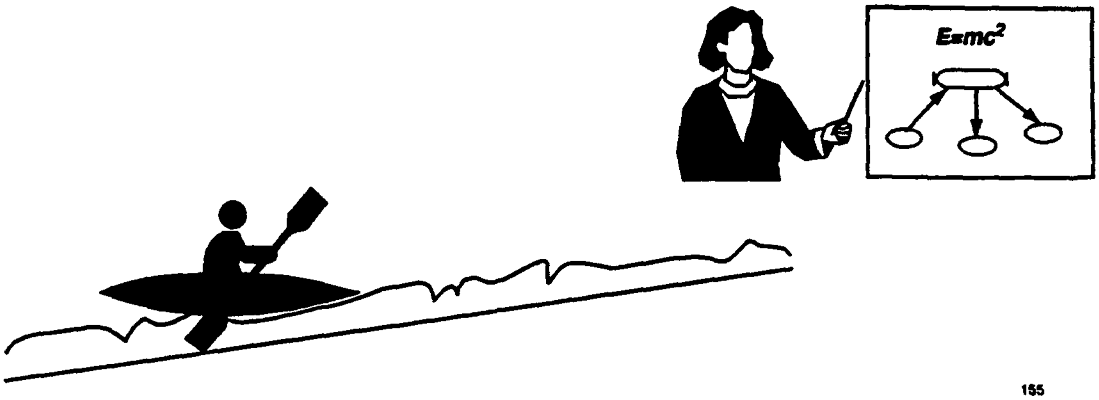
Trends in Architecture for Reuse →
- object-oriented architectures
- event-based architectures
- object-oriented/event hybrids

Architecture-Based Reuse Systems →
- overview of concepts
- analogy with configuration management

This page intentionally left blank.

Toward a Science of Software Architecture

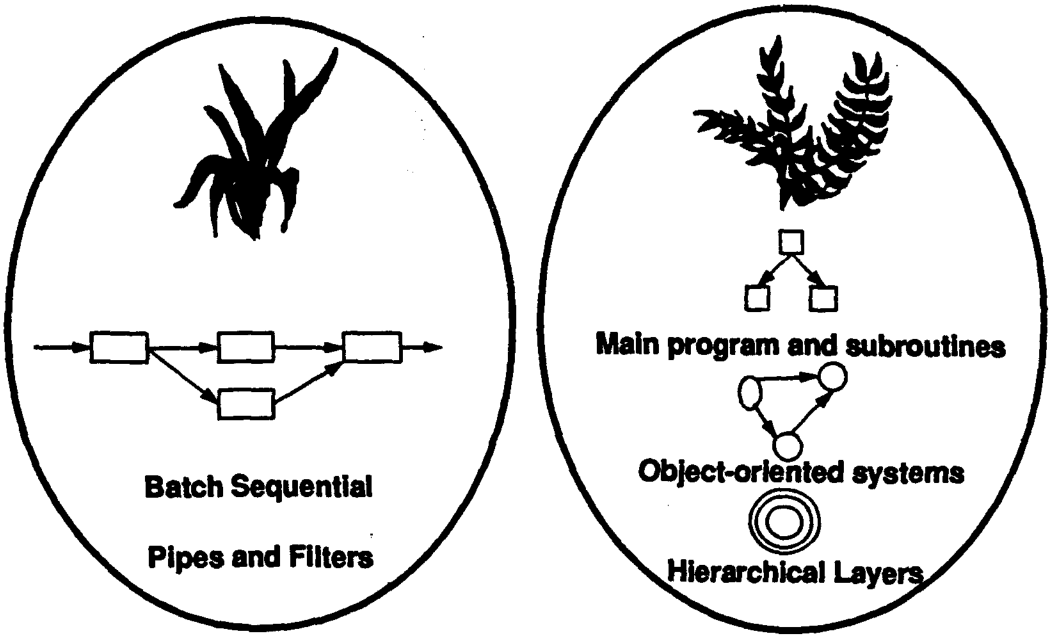
- What kinds of software architectures exist?
- What kinds of software architecture best support re-use?



Toward a Science of Software Architecture

These two questions are important for this seminar. The first part of this session will attempt to answer these questions. At this point it is appropriate to survey some of the architecture styles that were identified by Garlan and Shaw. The graphic shows that getting to a theory of software architecture is an upstream paddle.

What Kinds of Software Architectures Exist?



Batch Sequential

Pipes and Filters

Main program and subroutines

Object-oriented systems

Hierarchical Layers

Data Flow Systems

Call and Return Systems

What Kinds of Software Architectures Exist?

Academic researchers are currently studying and classifying architectures (similar to the way a biologist would study species of plants or animals). Hopefully this will lead to the identification of common styles (idioms) and system patterns. The long term goal is to develop guidelines for applying these styles and patterns in new/re-engineered systems. The main styles and patterns that have been identified so far are explained briefly below.

Data Flow style:

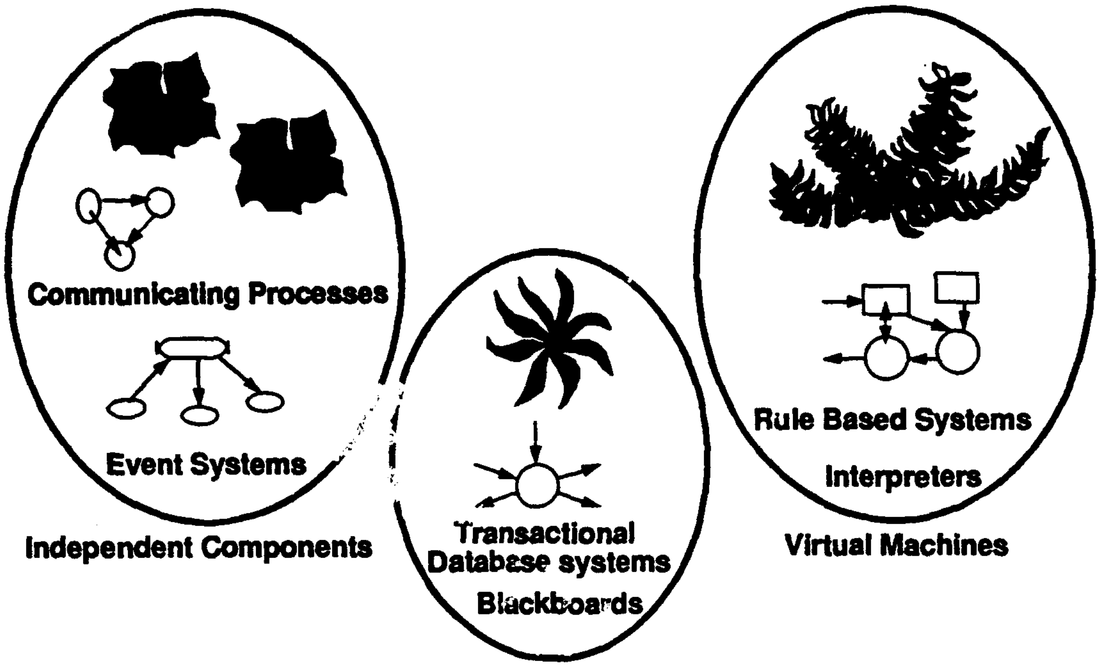
- Batch Sequential - each step runs to completion
- Pipes and Filters - linked stream transformers

Call and Return style:

- Main program and subroutines - traditional functional decomposition
- Hierarchical layers - well defined interfaces and information hiding (e.g. kernels, shells)
- Object-oriented systems - abstract data types with inheritance

Reference: Garlan, Shaw - "An Introduction to Software Architecture" to appear in Advances in Software Eng. and Knowledge Eng., vol.1 1993

What kinds of Software Architectures Exist?



Communicating Processes

Event Systems

Independent Components

Transactional Database systems

Blackboards

Data-centered systems

Rule Based Systems

Interpreters

Virtual Machines

What Kinds of Software Architectures Exist?

Independent Components style:

- Communicating processes - asynchronous message passing
- Event systems - implicit invocation

Virtual Machines style:

- Interpreters - input driven state machine
- Rule-based systems - rule based interpreter

Data-centered systems:

- Transactional Database Systems - central data repository/query driven
- Blackboards - central shared representation/opportunistic execution

Reference: Garlan, Shaw - "An Introduction to Software Architecture" to appear in Advances in Software Eng. and Knowledge Eng., vol.1 1993

Roadmap for this Session

Architecture "Defined" →
- phenomenology
- externally visible qualities of architecture: a hypothesis

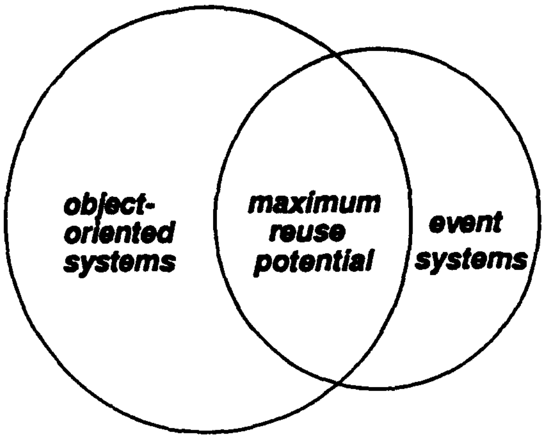
Towards a Science of Architecture →
- kinds of architectures

☞ **Trends in Architecture for Reuse** →
- object-oriented architectures
- event-based architectures
- object-oriented/event hybrids

Architecture-Based Reuse Systems →
- overview of concepts
- analogy with configuration management

This page intentionally left blank.

What Kinds of Architectures Best Support Reuse?



object-oriented systems

maximum reuse potential

event systems

Object-oriented systems

- how do they support reuse?
- trends

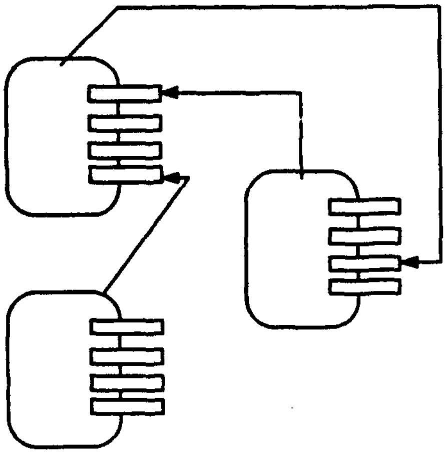
Event systems

- what are the key ideas?
- why do they support reuse?
- trends

What Kinds of Software Architecture Support Reuse?

An architecture that has a mixture of object-oriented and event systems characteristics is best suited for supporting reuse of design and code in our view. The following part of the presentation will discuss these architecture styles in more detail. There has been an explosion in object-oriented systems in the last decade and it is assumed that most of the audience is familiar with the basic concepts. Event systems are less well known so more background will be given.

Object-Oriented Systems - Why?



Key reuse mechanisms:

- **objects**
  - **encapsulation**
  - **abstraction**
- **classes**
  - **inheritance**
- **mechanisms scaled-up to large objects**

Object-Oriented Systems - Why?

Objects facilitate modeling the world directly in software thus making a system easier to understand. They hide details (abstraction). Objects reduce coupling and therefore reduce the propagation of changes. Objects are more independent from the context of a system and therefore probably more reusable.

Classes group objects for ease of understanding. Inheritance reduces duplication of design/code and allows extension of existing classes into new subclasses.

In the context of architectures and mega-programming we are not talking about small data structure objects (code level). We are talking about large components or subsystems (e.g. stand-alone tools).

The disadvantage of OO systems is that objects have to know the names of the operations in other objects.

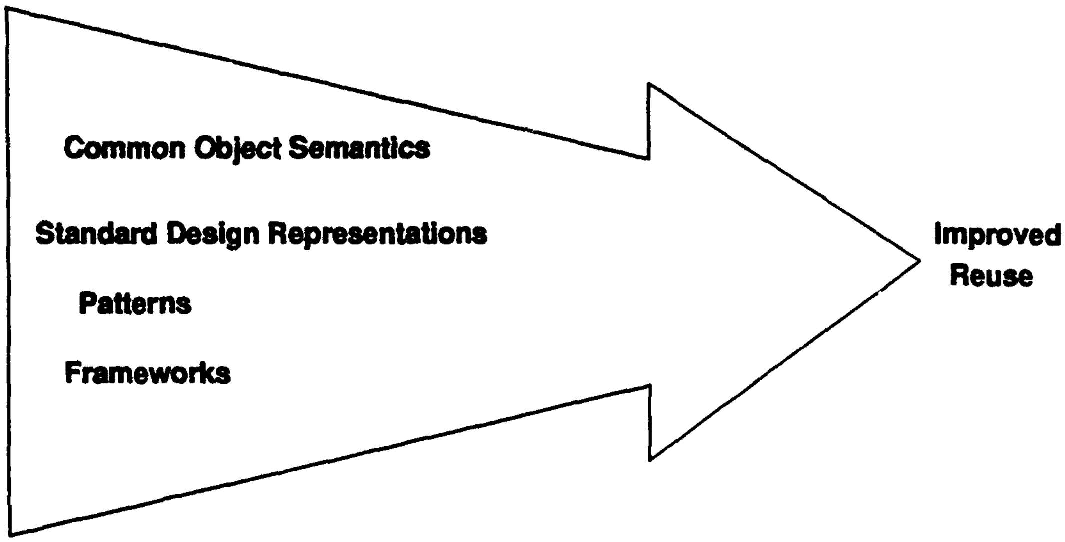
References:

Garlan, Shaw - "An Introduction to Software Architecture" to appear in Advances in Software Eng. and Knowledge Eng., vol.1 1993

Booch - "Object-Oriented Design with Applications" Benjamin Cummings 1991

Meyer B. "Object-Oriented Software Construction" Prentice-Hall 1988

Object-Oriented Systems - Trends



Common Object Semantics

Standard Design Representations

Patterns

Frameworks

Improved Reuse

Object-Oriented Trends

Common object semantics: The Object Management Group has developed an object model (as part of the Common Object Request Broker Architecture CORBA) which attempts to standardize object management services across heterogeneous platforms and establish common facilities (standard general utility objects - e.g. editors, help facilities, e-mail). This is done by establishing standard object interfaces(signatures) which include operations and parameters. The object model would promote extensive reuse of general objects. The SEI is pursuing the idea of common signatures in the context of a specific domain. This should prove to be a powerful reuse approach.

Standard design representations: Currently their is a proliferation of object-oriented design representations (graphics and text). Developing a standard representation would greatly facilitate the reuse of design/code.

Patterns: Researchers are beginning to identify and catalog patterns (micro-architectures) in object-oriented systems. These patterns are organized in a taxonomy and have a standard documentation template that may include: intent, motivation, applicability, participants, collaborations, diagrams, consequences, implementation, examples, and "see also". These patterns will help develop and facilitate understanding of software architectures for whole systems.

Frameworks: Object-oriented frameworks are flexible configurations of components (component classes) connected by data flow. Frameworks have many of the characteristics of a software architecture. Researchers are experimenting with the application of frameworks in various domains.

OMG "Object Management Architecture Guide" Sept. 1992

Peterson, Stanley "Mapping a Domain Model and Architecture to a Generic Design" CMU/SEI-TR draft

Booch "Next Generation Methods - Bringing Order out of the Chaos" Journal of Object Oriented Programming - Supplement on OO Analysis and Design July/August 1993

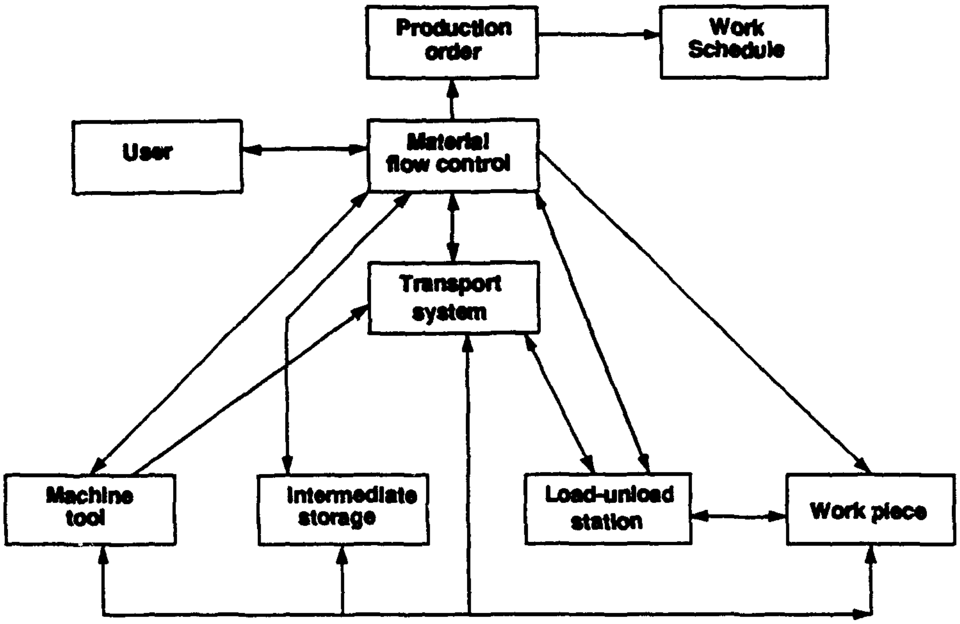
Taft "Ada 9X: A Technical Summary" Communications of the ACM, Nov. 1992

Gamma, E., Helm, R., Johnson, R., Vlissides, R., Design Patterns: "Abstraction and Reuse of Object Oriented Design"—unpublished paper. Contact Erich Gamma at Taligent, Inc., 10725 N. De Anza Blvd., Cupertino, CA 95014-2000

Nierstrasz, Gibbs, Tsichritzis "Component Oriented Software Development", Communications of the ACM, Vol. 35, No. 9 Sept. 1992.

Buschmann "Rational architectures for object-oriented software systems" Journal of Object-Oriented Programming, Sept. 1993

Object-Oriented Framework: Example



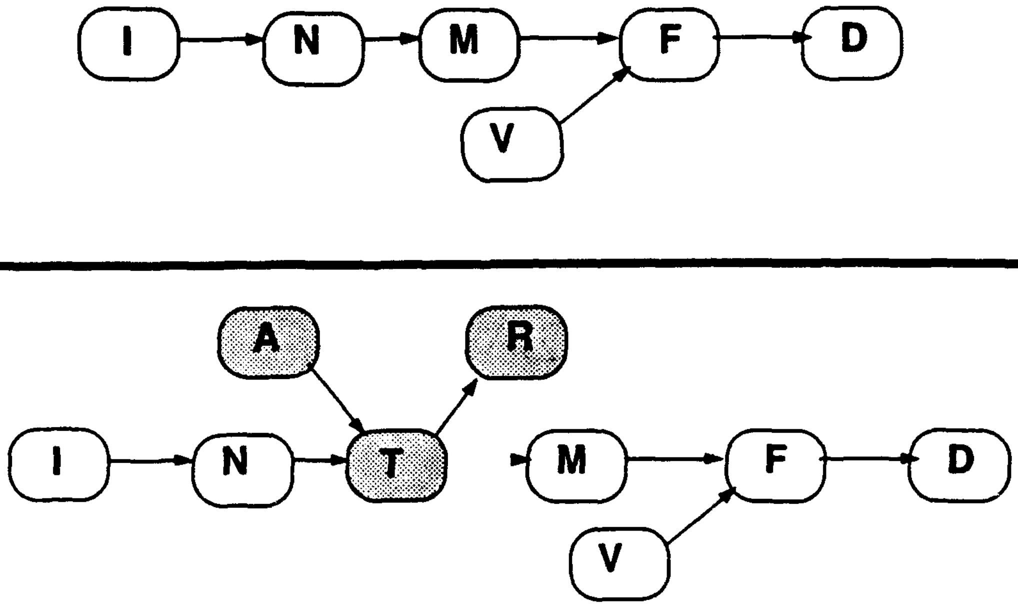
Object-Oriented Framework: Example

An OO framework is both a reusable architecture and an architecture that supports reuse of components. This particular framework is for a generic material flow control system which is part of a larger framework for flexible manufacturing systems. The basic structure and relationships between elements (component classes) can be reused regardless of the specific work pieces being transported. Basic operations and data (i.e. signatures) are defined at an abstract level for the domain.

References:

Buschmann "Rational architectures for object-oriented software systems" Journal of Object-Oriented Programming, Sept. 1993

Object-Oriented Framework: Example Adaptation



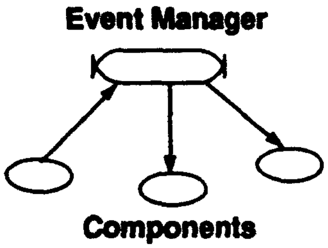
Object-Oriented Framework: Example Adaptation

An OO framework can be designed to be adaptable and flexible so that new objects or subsystems can be grafted in or removed. The top part of the slide shows the basic framework. The bottom part of the slide shows several new objects grafted in.

References:

Nierstrasz, Gibbs, Tsichritzis "Component Oriented Software Development", Communications of the ACM, Vol. 35, No. 9 Sept. 1992.

Event Systems - Key Ideas



- Components can announce (broadcast) events.

- Components can register for events of interest and associate operations with them.

- Upon event announcement the corresponding operations are automatically invoked (by the system).

- Hence, invocation is implicit, although explicit invocation is often still provided.

Event Systems - Key Ideas

Event systems are emerging as an important architecture for integrating diverse components (objects or modules). Many event systems are also object-oriented. They may also allow explicit invocation (direct calls) to control the flow of execution.

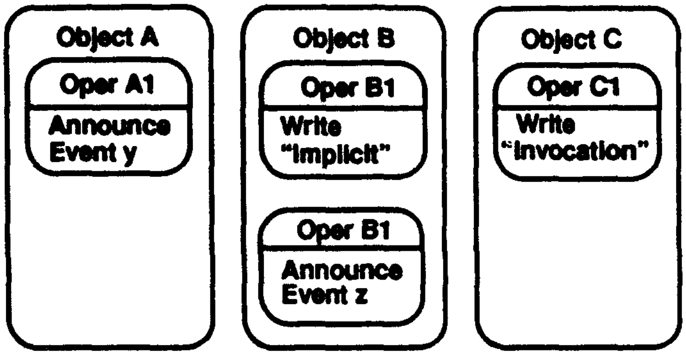
References:

David Garlan and Curtis Scott
Adding Implicit Invocation to Traditional Programming Languages
Proceedings of The 15th International Conference on Software Engineering
May 17-21, 1993 Baltimore, MD, pp. 447-455.

David Garlan and Mary Shaw
An Introduction to Software Architecture
To appear in *Advances in Software Engineering and Knowledge Engineering*, Volume I
World Scientific Publishing Co, 1993.

David Garlan, Gail E. Kaiser and David Notkin
Using Tool Abstraction to Compose Systems
IEEE Computer, June 1992, pp. 30-38

Event Systems Example



Object A

Oper A1

Announce Event y

Object B

Oper B1

Write "Implicit"

Oper B1

Announce Event z

Object C

Oper C1

Write "Invocation"

System Register		
Event	Object	Oper
w	A	A1
x	B	B1
y	B	B1
y	C	C1

Assume Operation A1 is called. This results in the announcement of event y.

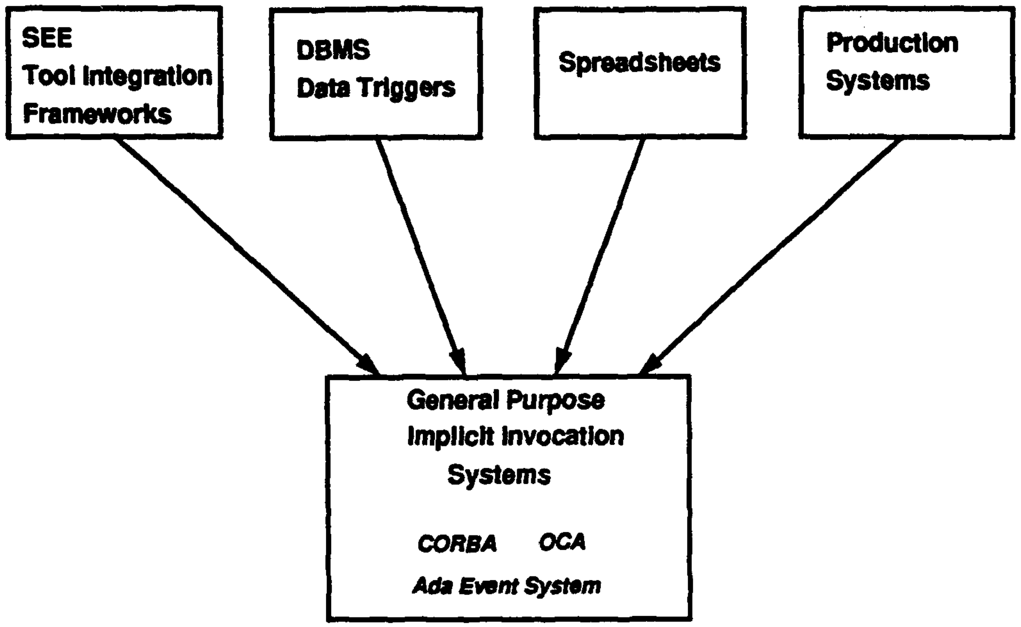
The system register (event manager) shows that both Object B and Object C can respond.

Object B would invoke Operation B1;
Object C would invoke Operation C1.

If the system does not choose one over the other, then "Implicit Invocation" will be output (in some order).

This page intentionally left blank.

Evolution of Implicit Invocation



SEE Tool Integration Frameworks

DBMS Data Triggers

Spreadsheets

Production Systems

General Purpose Implicit Invocation Systems

CORBA *OCA*

Ada Event System

Evolution of Implicit Invocation

A main source of ideas for event systems was research on (SEE) Tool Integration Frameworks.These SEE integrated frameworks are usually a collection of tools running as separate processes. Event are broadcast via separate dispatcher process. Communication channels are provided by host OS (e.g., Unix sockets).

The ideas behind event systems also show up in special purpose languages and application frameworks which provide access through special notations and runtime support. Examples include: active data triggers for a DRMS, spreadsheets (via dependency facts), and production systems for expert advice.

General purpose event systems are beginning to emerge. They are being built within general purpose language environments like Ada. The Common Object Request Broker Architecture (CORBA) is an emerging standard for event system architectures across heterogeneous platforms. The Object Connection Architecture (OCA) is a generalization of the Object Connection Update (OCU) model originally developed for the flight simulator domain (the OCA is related to the event system architecture).

References:

Garlan, Scott "Adding Implicit Invocation to Traditional Programming Languages" 15th ICSE

OMG "Object Management Architecture Guide" Sept. 1992

Peterson, Stanley "Mapping a Domain Model and Architecture to a Generic Design" CMU/SEI-TR draft

Lee, Rissman, D'Ippolito, Plinta, Van Scoy "An OOD Paradigm for Flight Simulators" CMU/SEI-88-TR-30

Event Systems: Advantages

- **Provides significant support for reuse:**
  - **Can integrate components simply by registering their interest in the events of the system.**
- **Eases system evolution:**
  - **Loose coupling helps eliminate name dependencies between components.**
  - **Can add / replace components without interfering with existing objects.**
  - **Changes localized to system register / event manager.**
- **Upward compatible.**
  - **Can still have explicit invocation.**

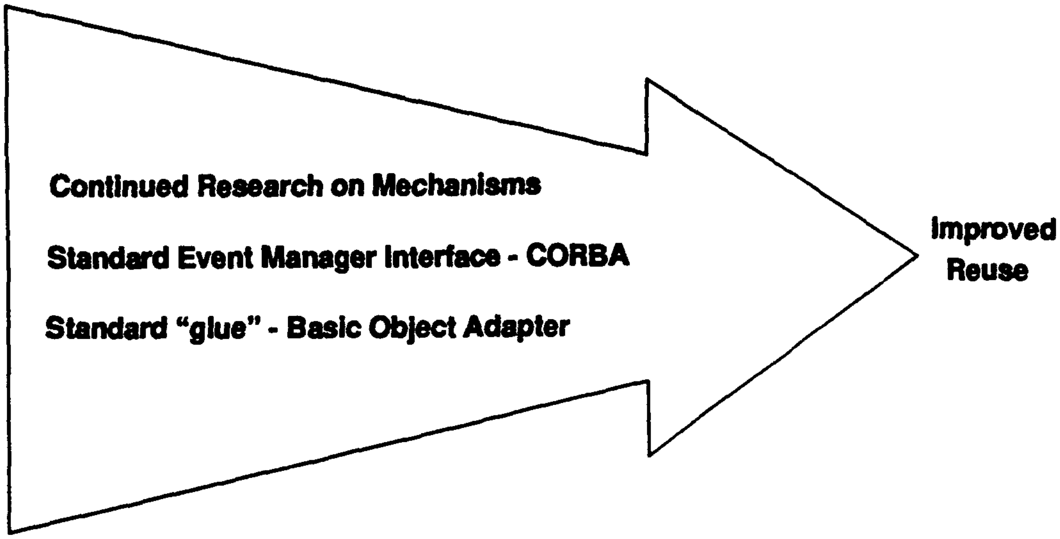
This page intentionally left blank.

Event Systems: Disadvantages

- **Indirection overhead may be high.**
- **Special purpose languages for event broadcast are limited by definition.**
- **Components relinquish control over the overall computation.**
- **A component does not know: "who" will respond or the order and completion of invocations, so cycles could result.**
- **Hard to reason about correctness.**

This page intentionally left blank.

Event Systems - Trends



Continued Research on Mechanisms

Standard Event Manager Interface - CORBA

Standard "glue" - Basic Object Adapter

Improved Reuse

Event Systems -Trends

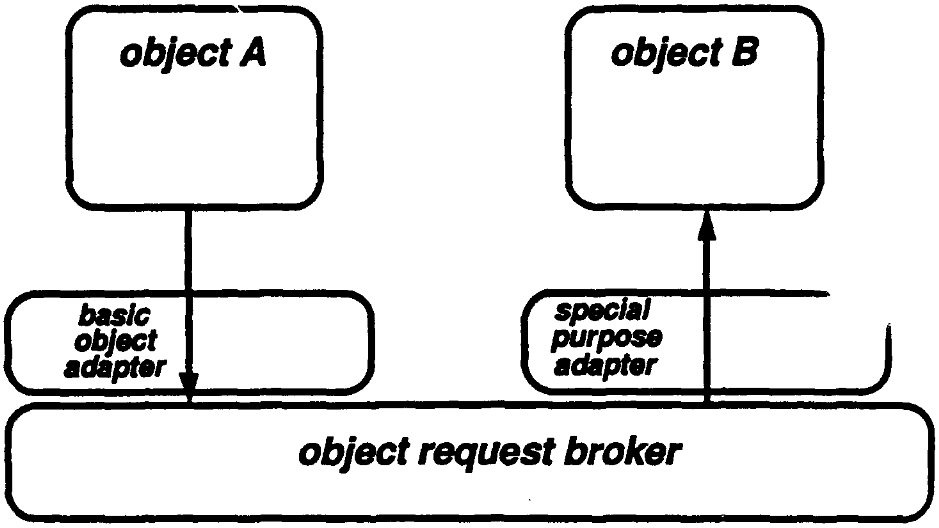
Continued research is needed to explore the design space of event system mechanisms and to fine tune them for specific classes of applications. Ongoing research is also addressing the process of developing systems based on the event system model.

References:

Garlan, Scott "Adding Implicit Invocation to Traditional Programming Languages" 15th ICSE

Peterson, Stanley "Mapping a Domain Model and Architecture to a Generic Design" CMU/SEI-TR draft

CORBA



object A

object B

basic object adapter

special purpose adapter

object request broker

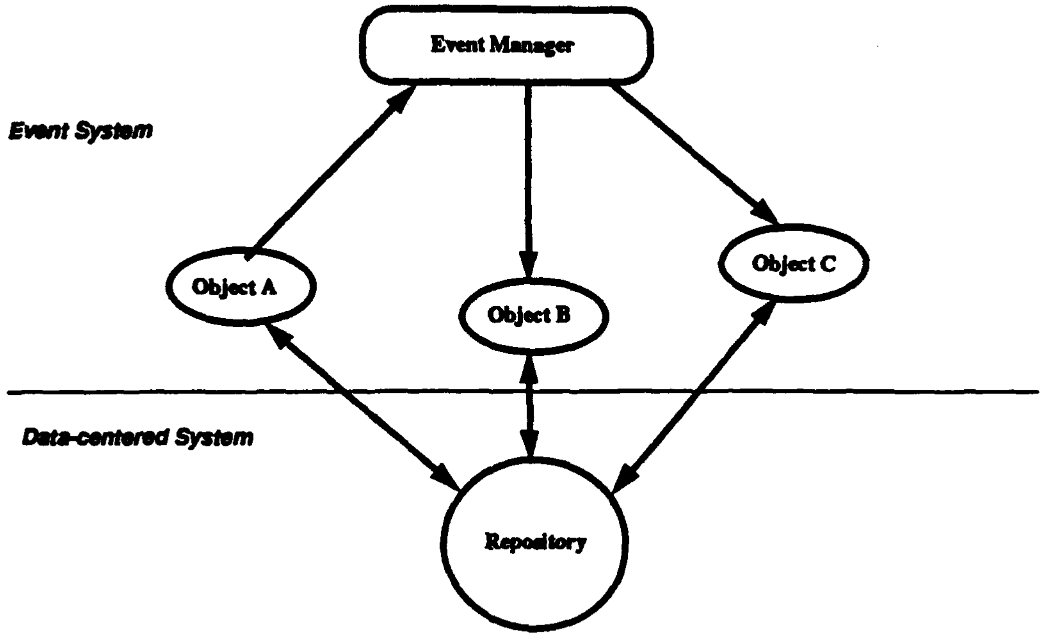
CORBA

The Object Management Group (OMG) is working on standardizing the interfaces to an object request broker within the Common Object Request Broker Architecture (CORBA). OMG has developed an interface definition language (IDL) that looks a lot like C++. Bindings to the IDL can be written in other languages (a C binding exists now). The OMG has also defined a Basic Object Adapter which provides standard "glue" (i.e. a wrapper) so that components can be integrated into a CORBA based heterogeneous system. Special purpose adapters can also be defined. CORBA is still evolving.

References:

OMG "The Common Object Request Broker: Architecture and Specification" 1992

Hybrid Architecture: Event/Data-centered System

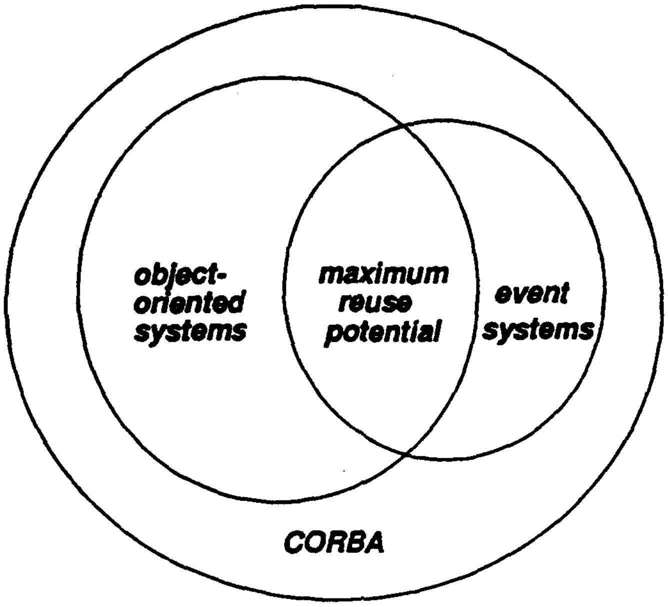


Hybrid Architectures: Event/Data-centered System

Large systems often are made up of components that have combined architecture styles. This diagram shows a popular hybrid architecture for software engineering environments where the two styles are complimentary. Control integration is achieved through event system mechanisms whereas a data-centered mechanism (repository) facilitates data integration.

Reference: Garlan, Shaw - "An Introduction to Software Architecture" to appear in Advances in Software Eng. and Knowledge Eng., vol.1 1993

Architectures for Reuse - Summary



Architectures for Reuse - Summary

As of late 1993 object-oriented and event systems appear to be the most promising architecture styles for accomplishing large scale reuse. CORBA is an important initiative that should facilitate the cost-effective adoption of a hybrid object-oriented event system architecture. CORBA is also attempting to address a few other important issues such as internationalization (multi-lingual and multi-cultural issues).

Roadmap for this Session

Architecture "Defined" →
- phenomenology
- externally visible qualities of architecture: a hypothesis

Towards a Science of Architecture →
- kinds of architectures

Trends in Architecture for Reuse →
- object-oriented architectures
- event-based architectures
- object-oriented/event hybrids

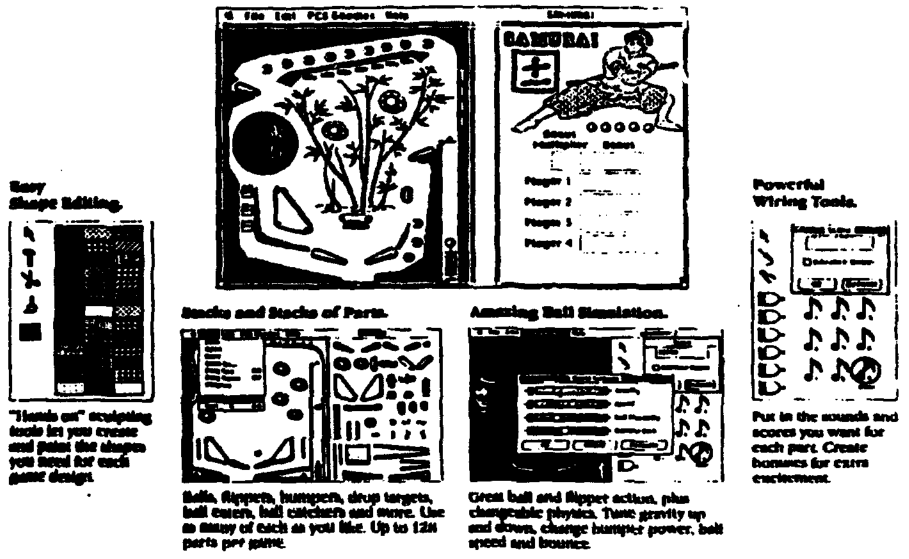
☞ Architecture-Based Reuse Systems →
- overview of concepts
- analogy with configuration management

This page intentionally left blank.

ARDS

Architecture-Based Reuse for the End User

PINBALL CONSTRUCTION SET



Easy Shape Editing.

"Hands on" sculpting tools let you create and paint the shapes you need for each game design.

Stacks and Stacks of Parts.

Balls, flippers, bumpers, drop targets, ball eaters, ball catchers and more. Use as many of each as you like. Up to 128 parts per game.

Amazing Ball Simulation.

Great ball and flipper action, plus changeable physics. Tune gravity up and down, change bumper power, ball speed and bounce.

Powerful Wiring Tools.

Put in the sounds and scores you want for each part. Create bonuses for extra excitement.

ARDS

Architecture-Based Reuse for the End User

Sometimes it is useful to imagine the extremities of a concept (e.g., *"reductio ad absurdum"*).

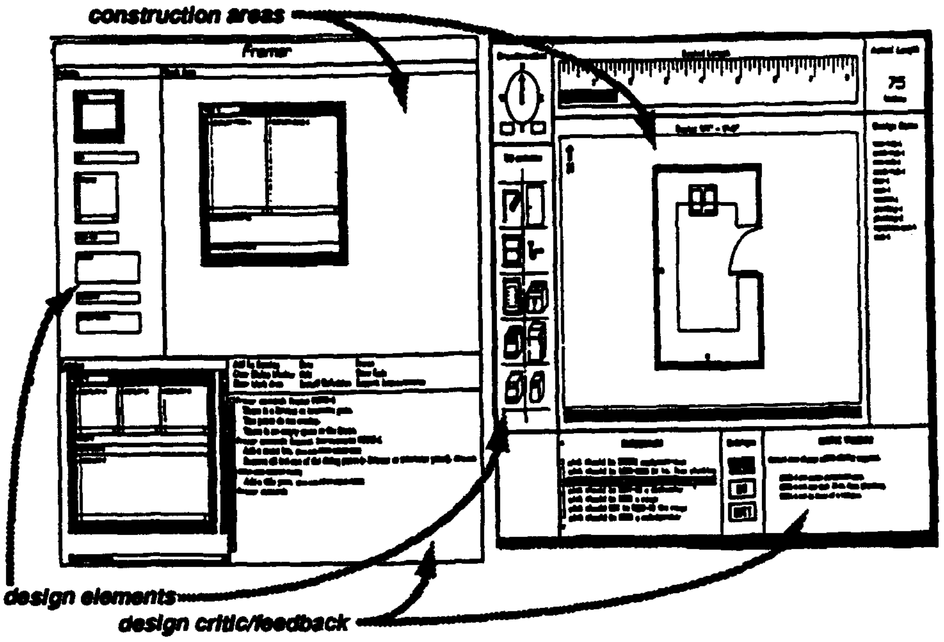
Is this pinball constructor kit perhaps the ultimate in architecture-based reuse environments? It seems to have many of the elements we would expect: a built-in application framework, sets of components, rules for construction, automated support for construction, mechanisms for connecting components, etc.

In this example, the user of the reuse system is the application end user. Would it be unreasonable to expect the end user of, say, a command and control command center to similarly "compose" the activity centers, screens and information flow among screens and activity centers within a command center? In the near term this may not be feasible due to the complexity of the application, the dependency of system function on events and time, the impact of mission and doctrine, etc., on the end application.

References

Pinball Constructor: photocopy of a product jacket for commercially-available personal computer application

Architecture-Based Systems for the System Designer



Architecture-Based Systems for the System Designer

The previous example illustrated architecture-based services for the end user of the application; we might view the previous example as more of a "tailorable application" than architecture-based reuse system.

But what if we target such a system not for the application end-user, but for its designer? In this case the system could move one notch closer to the implementation abstractions. In this illustration two systems from the Crack project demonstrate the concept nicely. The system depicted on the left is a design assistant for human-machine interfaces (HMI), while the system on the right is targeted to kitchen design. Although we are still not at the level of command center, these systems mirror some of the capabilities of the pinball constructor: a set of design elements, in these cases targeted to application designers; rules for composition; a composition/construction area, etc.

Note that in these examples the design elements are "domain-specific." This was true of the pinball constructor (flippers, bells, balls, etc.), the window design assistant (display, scrollers, etc.) and the kitchen design assistant (doors, sinks, stoves, etc.). But what if we substitute for domain-specific design elements the components, or design elements, of an architecture style (or architecture model)? We may find ourselves in an environment such as that provided by several CASE vendors (StateMate, UNAS/SALE).

References

Gerhard Fischer, "Human Computer Interaction Software: Lessons Learned, Challenges Ahead," IEEE Software, January 1989.

Architecture-Based Systems for the Programmer

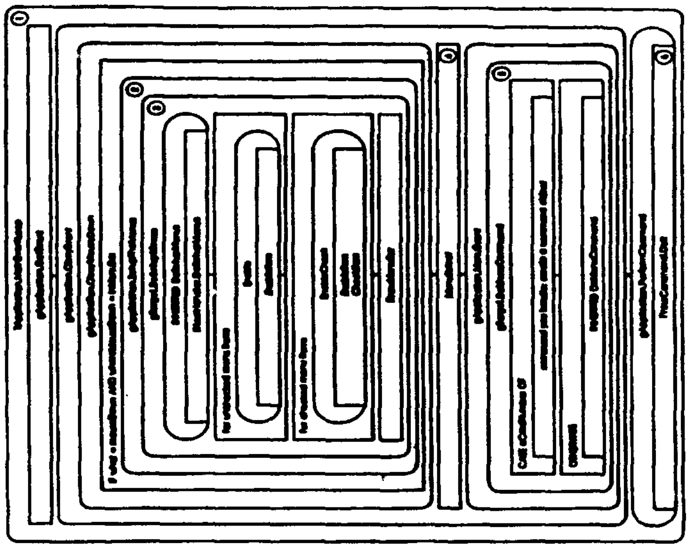


Figure 7-6
Choosing a menu command

204 Chapter 7: The Flow of Control in Response to Events

Architecture-Based Systems for the Programmer?

This final example illustrates yet another concept of architecture-based reuse system. Where the pinball constructor was targeted to end users, and the kitchen design assistant targeted to a system designer, the Apple Macintosh MacAPP represents an architecture-based reuse system targeted to programmers. This figure is copied from the MacAPP documentation, and illustrates the use of an architecture as a template, or framework, into which application-specific functionality are inserted. In this case the application architecture is (more or less) "fixed"—much of the hard design work has been encoded in the application template.

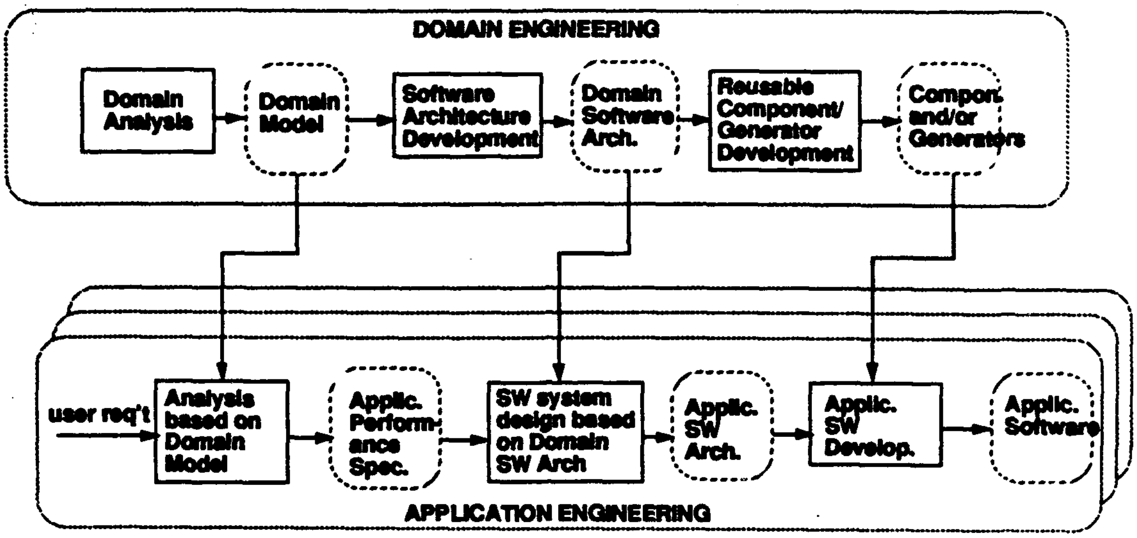
What this succession of examples illustrates is that there is a range of possible manifestations of "architecture-based reuse system." Moreover, these illustrates only varied the intended user of the system; many other dimensions of variability are possible.

A more general way of thinking about architecture-based reuse systems is to think of such systems as the *means of conveying the results of a domain-engineering life cycle to many possible application engineering life cycles*. Since the nature of each life cycle will vary depending upon domain, engineering infrastructure technologies (i.e., *software development environment tooling*), *local cultures*, etc., the associated reuse systems will also vary.

References

Apple Macintosh MacAPP Developer's Kit Documentation.

Reuse Environment: Integrating Domain and Application Engineering Life Cycles



DOMAIN ENGINEERING

Domain Analysis

Domain Model

Software Architecture Development

Domain Software Arch.

Reusable Component/ Generator Development

Compon. and/or Generators

user req't

Analysis based on Domain Model

Applic. Perform-ance Spec.

SW system design based on Domain SW Arch

Applic. SW Arch.

Applic. SW Develop.

Applic. Software

APPLICATION ENGINEERING

Reuse Environment: Integrating Domain and Application Engineering Life Cycles

The reuse environment is not simply an application building environment, it is a set of mechanisms and reusable products that allow us, in effect, to integrate domain engineering and application engineering processes.

Domain specific reuse is generally acknowledged to consist of two separate life cycles: the domain engineering life cycle, and the application engineering life cycle.

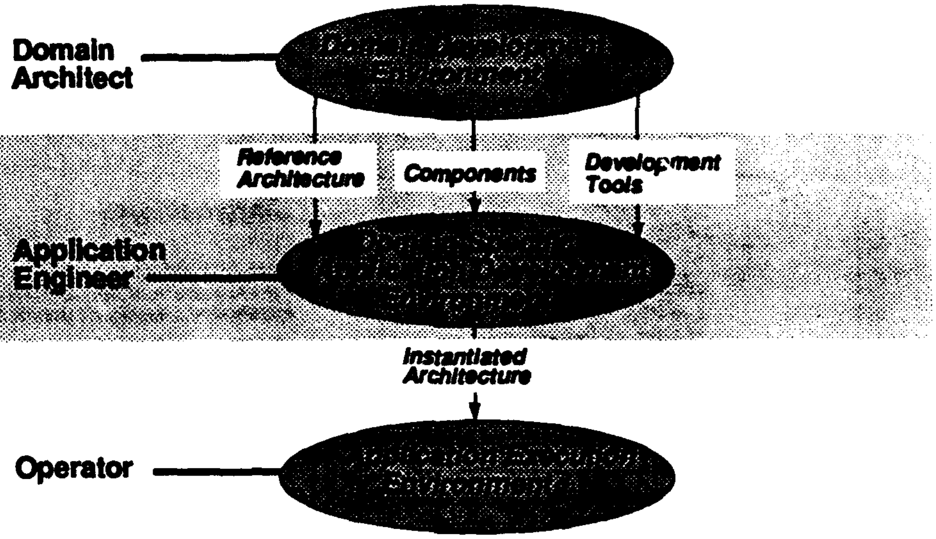
Some mechanisms must be present in order to transfer the results of domain engineering to application engineering—the packaging of reuse products, the tools and documentation needed to apply these products.

The chart illustrates the addition of a process dimension to this packaging. That is, the kinds of reusable products which flow from domain engineering to application engineering will depend upon the internal processes implied, or required, by each life cycle model. This chart illustrates just one of many possible models.

Reference:

T. Payton, "Domain-Specific Reuse," STARS 92 Annotated Briefing Chart, pp. 16-17. This chart is an interpreted rendering of one found in the STARS 92 proceedings.

A DSSA View of Architecture-Based Reuse Systems


Domain Architect
Domain Development Environment
Reference Architecture
Components
Development Tools
Application Engineer
Domain-Specific Application Development Environment
Instantiated Architecture
Operator
Application Execution Environment

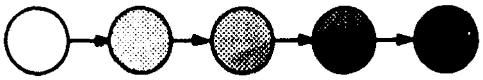
A DSSA View of Architecture-Based Reuse Systems

This chart illustrates the ARPA/DSSA view of architecture-based reuse systems. The chart is copied from a ARPA/DSSA presentation—the shaded box which highlights the application-specific development environment has been added to emphasize that in our discussions we are concerned with the tools and environments delivered to application developers, and not with the tools and environments necessary to conduct domain engineering activities.

References

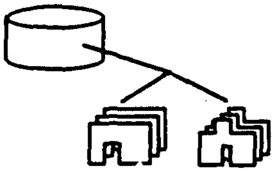
Lt.Col. Eric Matella, "Domain-Specific Software Architectures," STARS 92 annotated briefing, pp. 90-116.

Reuse Techniques and Architecture



Transformational

- formalisms and mappings
- generators—implicit transforms
- assistants—explicit transforms with guidance for human intervention



Compositional

- module building blocks
- mostly manual construction
- standard shapes or pre-fit by design

So where is the "architecture?" (a.k.a. reference architecture, application framework,...)
— pure transformational: in the transformation rules and languages
—pure compositional: in the form and function of components

Reuse Techniques and Architecture

We need to factor in another dimension in order to really understand the implications of the DSSA picture for domain-specific application engineering environments. The exact form and content of reference architectures, components and tools will be dependant upon the basic reuse technology approaches taken. Biggerstaff and others have defined taxonomies of reuse approaches. Without getting into needless detail, a top-level partitioning of approaches is the transformational/compositional dichotomy.

The transformational approach is characterized by a sequence of transformations among representations, with each transformation bringing the representation towards closer to some final state. Two major classes of transformational systems are:

1) generators: systems where the transformations are invisible/automatic (or, more commonly, there is only one automated transformation step).

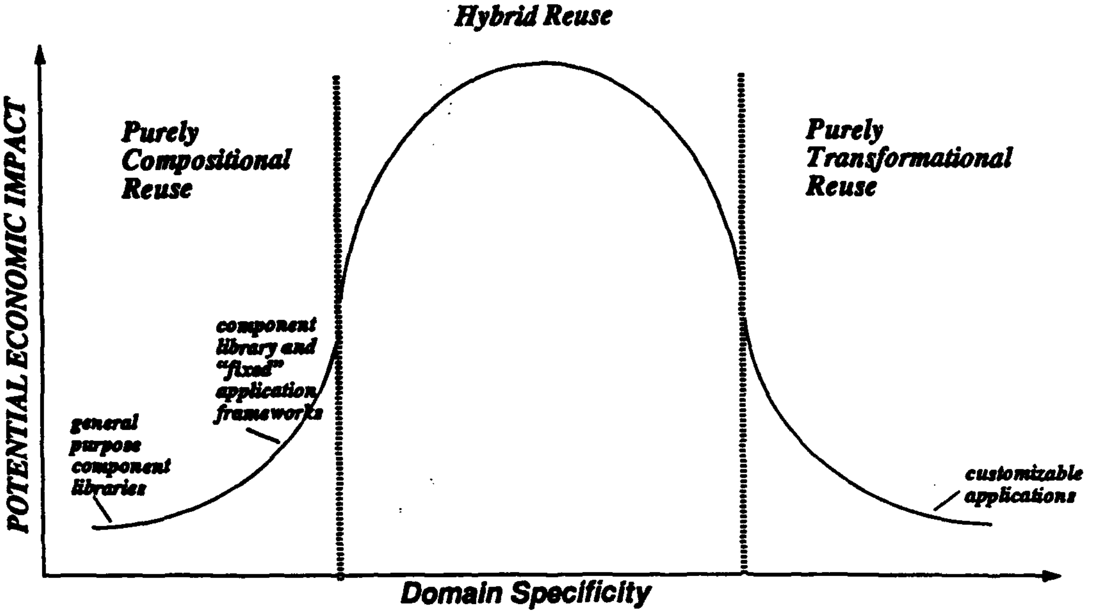
2) knowledge-based assistants: systems where there are multiple transformations, perhaps but not necessarily through different representations, and where the transformations are visible to the "user," and where there is guidance provided by the system to assist in performing the transformation.

The compositional approach is characterized by reuse through manual composition of concrete code components. Either families of components are developed (e.g., GRACE components) or highly-parameterized components are developed. In either case there is little scope for "automation."

Interestingly, in both "extremes" the question of "where is the architecture" is the same: the architecture is implicitly represented. In the case of the transformation approach the architecture is found in the patterns locked in code generators, in the terminology of the languages, and the rules for creating sentential forms. In the composition approach the architecture is again implicit, or, at best, reflected in the structure/form (i.e., interfaces and function) of the components.

The use of software architecture can help achieve the benefits of both approaches in a hybrid strategy.

Hypothetical Impact Analysis



Source: Martin Griss, Hewlett-Packard, Informal Workshop Presentation

Hypothetical Impact Analysis

Although there are no solid economic models to draw upon, there is general consensus within the reuse community that, all else being equal, generative reuse techniques will yield more dramatic reuse results than a purely compositional approach.

The chart illustrates a hypothetical curve, with the area under the curve being "economic impact." No scale or measures are intended, and the picture is not meant to imply any precision: the "shape" of the curve is a guess. However, a number of factors support the general hypothesis that hybrid reuse may provide, in practice, the biggest bang-for-the-buck:

1) While generative reuse would be ideal, such generators can be extremely expensive to develop, and may only be effective in highly stable application domains. The most frequently applied application of generational technology is through "application specific languages" for pieces ("subdomains") of application domains, e.g., message formal processing systems, human-machine interface subsystems, form/report generation subsystems, are just a few examples.

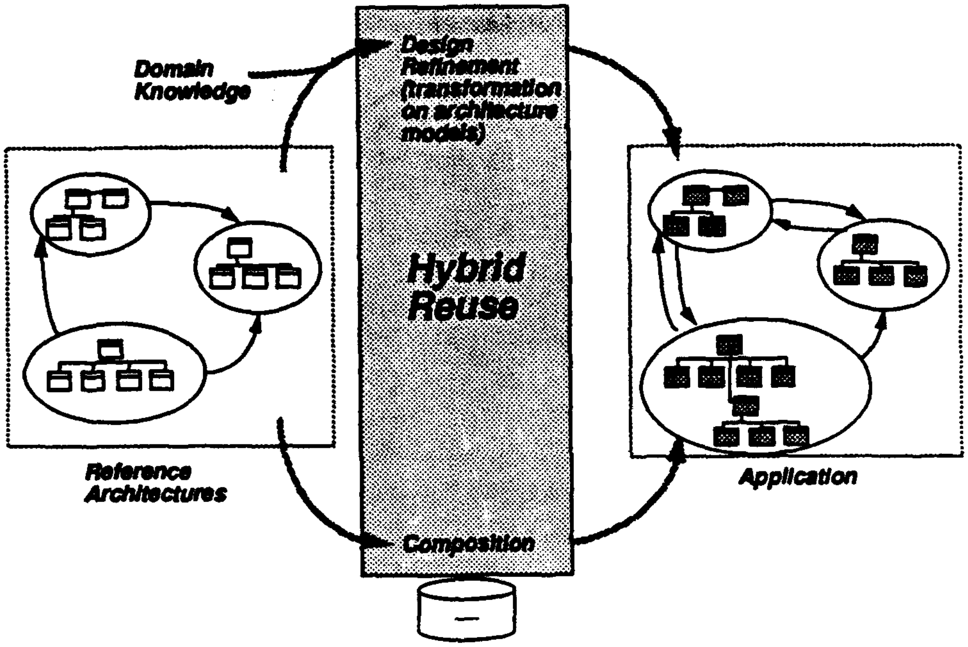
2) Compositional reuse is still a labor intensive activity, and it is difficult to develop a sufficiently "dense" population of components to satisfy diverse application requirements.

Ideally, then, we wish to develop reuse technologies which support the opportunistic hybridization of generative reuse with compositional reuse, whereever possible. Domain-specific software architectures can provide a mechanism for coherent integration of compositional and generational reuse, and, perhaps, a migration path towards increasing use of generative techniques within application domains.

References:

Martin Griss, Informal Presentation Charts, WISR6, Owego NY, Nov. 3-5 1993.

Hybrid-Reuse Strategies Centered on Architectures



Hybrid-Reuse Strategies Centered on Architectures

We've taken some liberties with illustrations from ARPA/DSSA presentations on this slide—we believe it reflects some important points of the ARPA/DSSA approach, but it should be noted that this picture is equal part "plagiarism" and "interpretation."

The basic tenant is that a reference architecture can be defined which represents a "partial application" in the domain. One analogy used to describe the reference architecture is as a "design with holes in it," with design refinement as the means of "filling the holes." In some cases the hole can be filled by generating a component, of selecting/adapting a component from a component library. In other cases the hole may be filled by selecting among various design alternatives, each alternative adding information to the design but, potentially, also introducing "new holes" which need to be filled.

It needs to be noted that this picture, though rich in concept, represents one common perspective from the DSSA program—different member projects each have refined the meaning of this picture using different technologies and processes. In at least one case the reference software architecture appears in the "middle" of a detailed system development process including hardware, controllers and software. The DSSA program has illustrated that the domain-specific application engineering environment does need to vary according to the problem domain, common engineering practice within the domain and cultural factors.

References

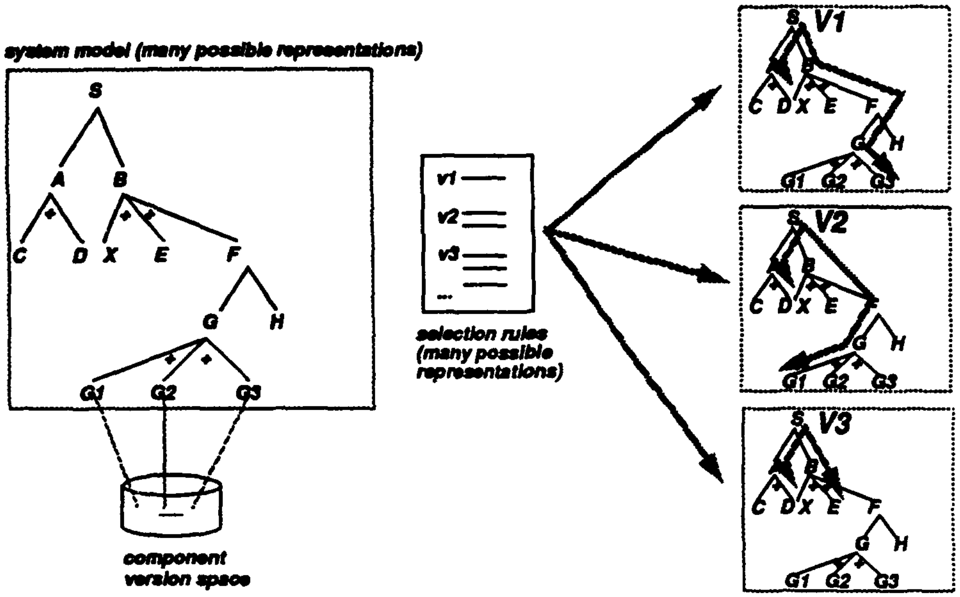
Lt.Col. Eric Matella, "Domain-Specific Software Architectures," in proceedings of STARS 92 Conference, annotated briefing, pp. 90-116.

Robert Balzer, "Model Management Examples," in proceedings of DSSA VII Workshop.

Robert Balzer, "Design Refinement in DSSAs," in proceedings of the JSGCC Software Initiative Strategy Workshop, December 1992.

Hybrid Architecture-Based Reuse and Compositional CM

Compositional CM = {system model+ version space + selection rules}


system model (many possible representations)

selection rules (many possible representations)

component version space

Hybrid Architecture-Based Reuse and Compositional CM

Even without getting into the details of specific domain-specific application engineering environments, it is possible to understand some of the information management requirements introduced by this model of hybrid, architecture-based reuse. One way to expose some of these issues is to compare the hybrid reuse approach with the relatively mature discipline of configuration management (CM).

Illustrated on this chart are some of the key principles behind a model of CM referred to as "compositional CM" in a paper by Feiler. The key elements of compositional CM are: 1) a system model, 2) a version space of sources, and 3) selection rules. There is great flexibility in the realization of this model (in fact, 1 and 2 can be combined). As illustrated by Feiler, the this CM model appears in a number of commercial products.

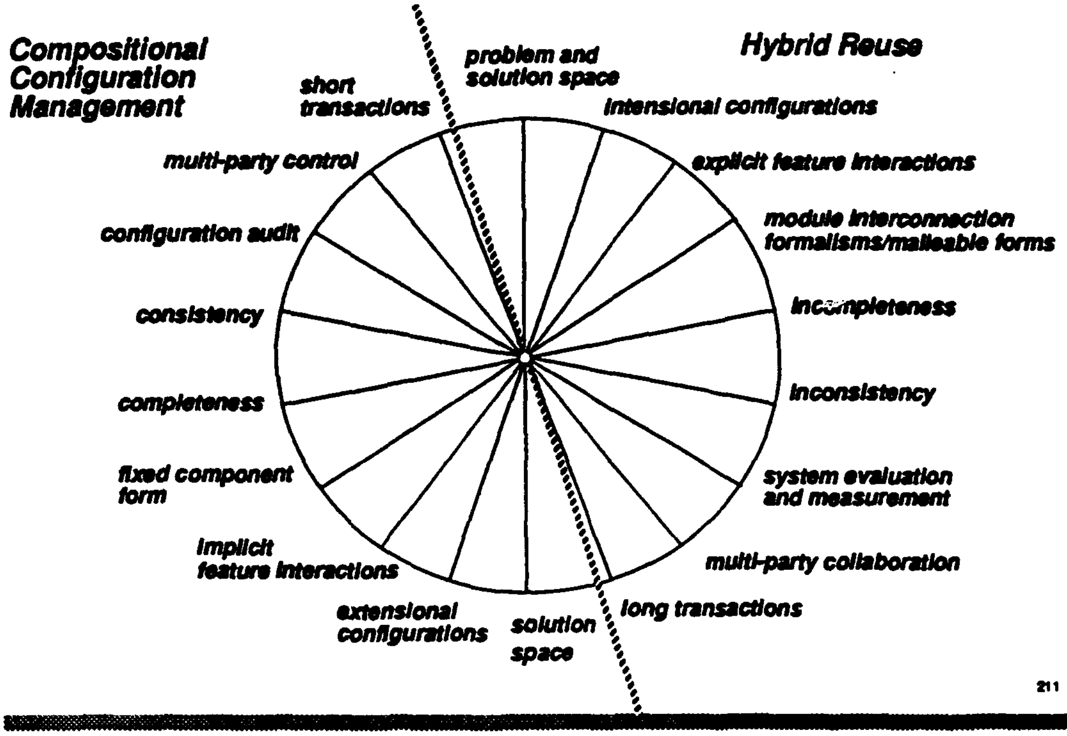
The system model reflects the structure of an application—here modeled as a simple "and/or" graph with "or" denoted as "+" and "and" denoted by the absence of a symbol. The interpretation is straightforward: a system is composed of A and B, with A composed either of variant C or D, etc. (It is important to note that we need not have such a representation, but it is convenient for the analogy.) Eventually, leaf nodes on the graph refer to concrete objects in the version space.

The selection rules can be primitive, e.g., an enumeration of the objects in the version space which belong to a configuration, or can be more elaborate. For example, one use of the and/or structure could be to mirror the hierarchical relationships in the system design, and could be "decorated" with attributes which could then be used in selection rules as predicates, e.g., configuration version 1 is such that we select versions where TESTED=TRUE and HOST=VAX. It is easy to see how a family of systems can be enumerated.

For the purposes of the analogy, we will equate system model with reference architecture, version space with component library, and selection rules with refinement rules.

Peter Feiler, Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Hybrid Architecture-Based Reuse and Compositional CM



Compositional Configuration Management

Hybrid Reuse

short transactions

problem and solution space

intensional configurations

multi-party control

explicit feature interactions

configuration audit

module interconnection formalisms/malleable forms

consistency

incompleteness

completeness

inconsistency

fixed component form

system evaluation and measurement

implicit feature interactions

multi-party collaboration

extensional configurations

solution space

long transactions

Hybrid Architecture-Based Reuse and Compositional CM

Our contention is that in many ways the hybrid architecture-based reuse approach mirrors that of compositional CM. The following dichotomies serve to illustrate these differences, and shed light on the technology considerations involved in architecture-based reuse systems:

Problem and solution space versus system build: Architecture-based reuse involves managing complex design trade-offs and making decisions regarding which design decisions to make, which components to integrate, etc. This requires detailed information about the problem space. In contrast, CM manages objects in the solution space.

Intensional configurations versus extensional configurations: For architecture-based reuse we do not want to have to enumerate all possible configurations, but rather define the rules for creating new instances.

Explicit feature interaction versus implicit feature interaction: CM is not a design discipline: there is no inherent need to capture all of the complex interdependencies among components (see "intensional configurations"). Note: there are gray areas, such as Tartan's Configuration Management Assistant.

Malleable component form versus fixed component form: If the context in which components are reused is changing, there is an increasing need for these components to be adaptable to these changing contexts. This is one motivation for research into module interconnection languages.

Incompleteness versus completeness: By definition, a reference architecture is incomplete. This implies that refinement and composition tools will need to accommodate, track and manage incompleteness.

Hybrid Architecture-Based Reuse and Compositional CM

Inconsistency versus consistency: See incompleteness. There are some differences with incompleteness, mainly concerned with the interaction of features and design refinements which may occur as a result of refining several aspects of a design simultaneously (as if on a design agenda). It will be necessary to manage inconsistencies; it may even be desirable to allow inconsistency. (Incompleteness can be thought of as a severe form of inconsistency)

System evaluation and measurement versus configuration audit: As designs are refined there must be a way of evaluating the partial and completed products. The engineering practices for evaluation will differ, by domain; the evaluation of configurations is more concerned with completeness and consistency.

Multi-party collaboration versus multi-party control: Both CM and architecture-based reuse systems address the existence of multiple parties making changes to shared representations. However, the emphasis in CM is on change control and change management, while architecture-based reuse systems will involve more elements of computer-supported cooperative work. This is not unexpected since the architecture-based reuse system is likely to emphasize aspects of collaborative and exploratory design.

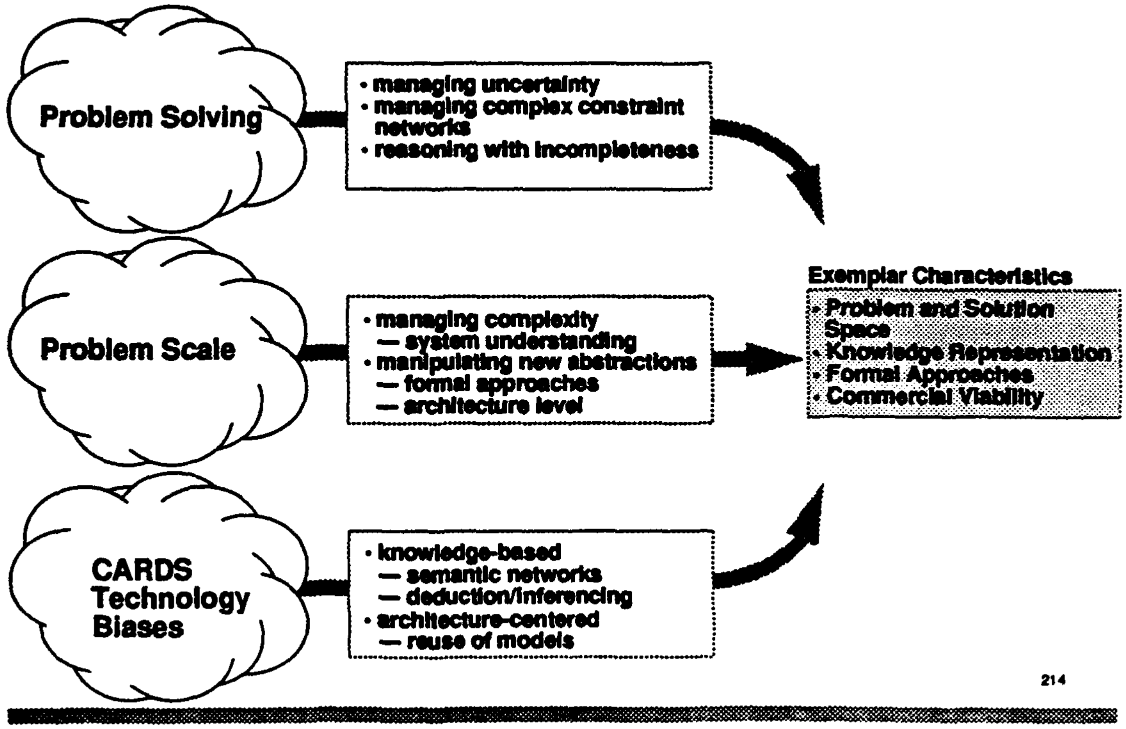
Long transactions versus short transactions: While CM may encompass some aspects of coordinating change among multiple parties, the construction of compositions from a CM system can be thought of as more or less atomic. However, real systems design takes place over an extended time, and may involve multiple parties, with backouts, checkpoints, etc.

References:

Peter Feiler, Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

R. Balzer, "Design Refinement in DSSAs," in proceedings of the JSGCC Software Initiative Strategy Workshop, Dec. 1992.

Influences on Selection of Exemplar Systems



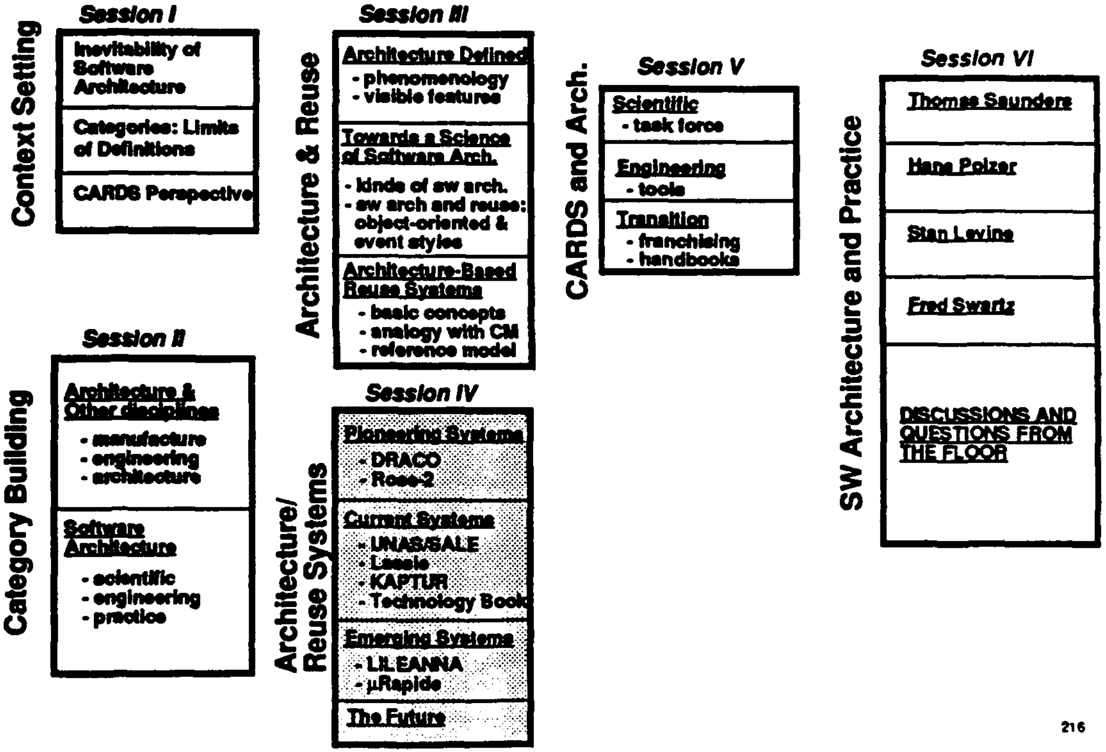
Influences on Selection of Exemplar Systems

A number of consequences on technology for architecture-based reuse systems can be discerned from the "wheel" diagram on the previous chart. Two key ideas emerge which differentiate architecture-based reuse systems from compositional CM systems: the focus on problem solving support and managing cognitive complexity from scale.

Both of these together may imply some use of formal modeling—including both knowledge representation and more mathematical, i.e., algebraic, representations. Even if this implication is not accepted by the reader, it is certainly the case that the topic of architecture-based design assistants and domain-specific architectures is attracting the attention of researchers in artificial intelligence and formal methods. Naturally this has influenced our selection of tools for evaluation purposes.

> Note: The reason why we have endeavored to examine these exemplar systems, which will be discussed in the next session in detail, will be disclosed in Session V: CARDS and Software Architecture.

A second contributing factor is, of course, our own program biases, based in large part on the technology foundations used by CARDS to build architecture-based reuse library systems. This base technology draws heavily upon knowledge representation systems, and demonstrates the development of automated reuse assistants for DoD software-intensive applications.



Context Setting

Session I

Inevitability of Software Architecture

Categories: Limits of Definitions

CARDS Perspective

Category Building

Session II

Architecture & Other disciplines
- manufacture
- engineering
- architecture

Software Architecture
- scientific
- engineering
- practice

Architecture & Reuse

Session III

Architecture Defined
- phenomenology
- visible features

Towards a Science of Software Arch.
- kinds of sw arch.
- sw arch and reuse: object-oriented & event styles

Architecture-Based Reuse Systems
- basic concepts
- analogy with CM
- reference model

Architecture/ Reuse Systems

Session IV

Pioneering Systems
- DRACO
- Rose-2

Current Systems
- UNAS/SALE
- Lassie
- KAPTUR
- Technology Book

Emerging Systems
- LILEANNA
- µRapide

The Future

CARDS and Arch.

Session V

Scientific
- task force

Engineering
- tools

Transition
- franchising
- handbooks

SW Architecture and Practice

Session VI

Thomas Saunders

Hans Polzer

Stan Levine

Fred Swartz

DISCUSSIONS AND QUESTIONS FROM THE FLOOR

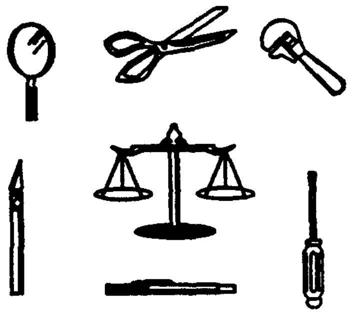
Central Archive for Reusable Defense Software (CARDS)

Session IV Architecture-Based Reuse Tools

16 November 1993

This page intentionally left blank.

Architecture Based Reuse Tools



- Pioneers:
  - Draco
  - ROSE-2
- Current:
  - LaSSIE
  - KAPTUR
  - UNAS
  - Technology Book
- Emerging:
  - LILEANNA
  - µRapide
- Future:
  - Integrated tools and libraries

Architecture Based Reuse Tools

The purpose of this session is to survey some representative tools which at least partially support architecture based reuse. This can be considered a mini-domain analysis of architecture based reuse tools. First we look at some early pioneers to give an historical context. Then we look at a sample of current tools (proprietary and available to the public). Next, tools emerging from the research community are examined because they may fill gaps in existing capabilities. Finally, we look at the vision for the future. For each tool we will describe key concepts, architecture representations, tool functionality, and lessons learned.

Draco: Key Concepts

- Early example of architecture based reuse tool
- A mixture of generation, assistance, and composition
- Reuse all aspects of software system development:
  - Requirements Information
  - Design Information
  - Source Code
- Application Architecture made up of multiple domains:
  - Application Domains (vertical)
  - Modeling Domains (horizontal)
- Multiple domain specific languages:
  - requirements/domains at different abstraction levels
  - transformations within domains
  - refinements between domains
- History mechanism:
  - tactics
  - pre-refined subsystems

Draco: Key Concepts

Draco embedded the notion of an application domain architecture made up of other, often more general purpose domains (horizontal domains). These horizontal domains could be reused in other application domains.Draco applied rules to transform(restate) specifications within one level of abstraction(domain) and refine specifications into a lower level of abstraction (until hopefully they reached code components). Draco also used a history mechanism to capture tactics for transformations/refinements and resulting re-occurring subsystems.

References:

Freeman, P., 1987. A conceptual analysis of the Draco approach to constructing software systems. IEEE Transactions on Software Engineering. SE-13, 7 (July), 830-844.

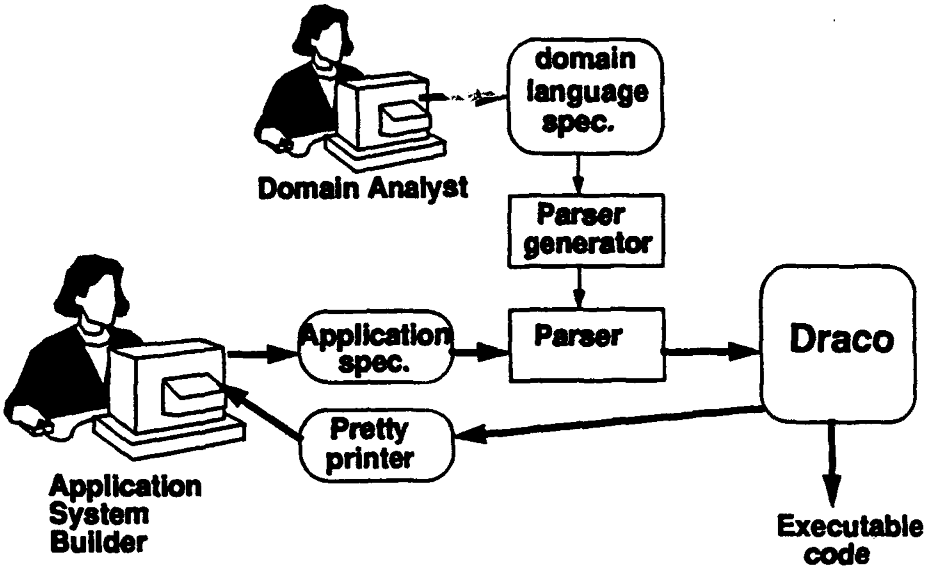
Neighbors, J.M., 1984. The Draco approach to constructing software from reusable components. IEEE Transaction on Software Engineering. SE-10, 5 (September), 564-574.

Neighbors, J.M., 1989. Draco: A method for engineering reusable software systems. In *Frontier Series: Software Reusability: Volume I - Concepts and Models.* Biggerstaff, T.J., and Perlis, A.J., Eds. ACM Press, New York, pp. 295-319, Chapter 12.

Neighbors, J.M., 1992. Draco: The evolution from software components to domain analysis. International Journal of Software Engineering and Knowledge Engineering. Volume 2, Number 3, September 1992; pp. 325-354.

Krueger, C. W., 1992. Software Reuse. ACM Computing Surveys. Volume 24, Number 2, June 1992, 131-183.

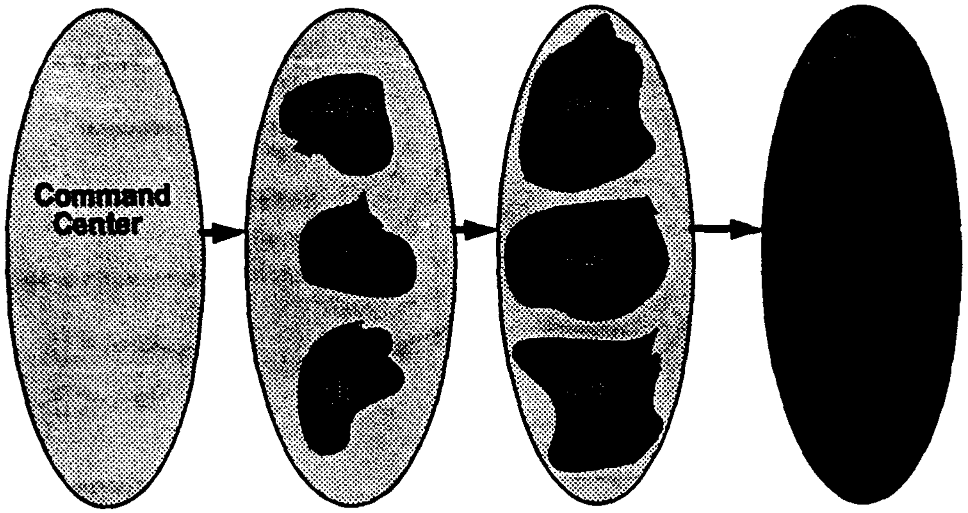
Draco



Draco

- Parser: Parses the application domain specification.
- Prettyprinter: Interacts with the system builder during the refinement process.
- Draco contains transformation and refinement rules and code components.
- Domain analyst: Develops domain specific language. This requires a significant amount of effort for either an application or modeling domain.

Draco: Refinement Process



application domain

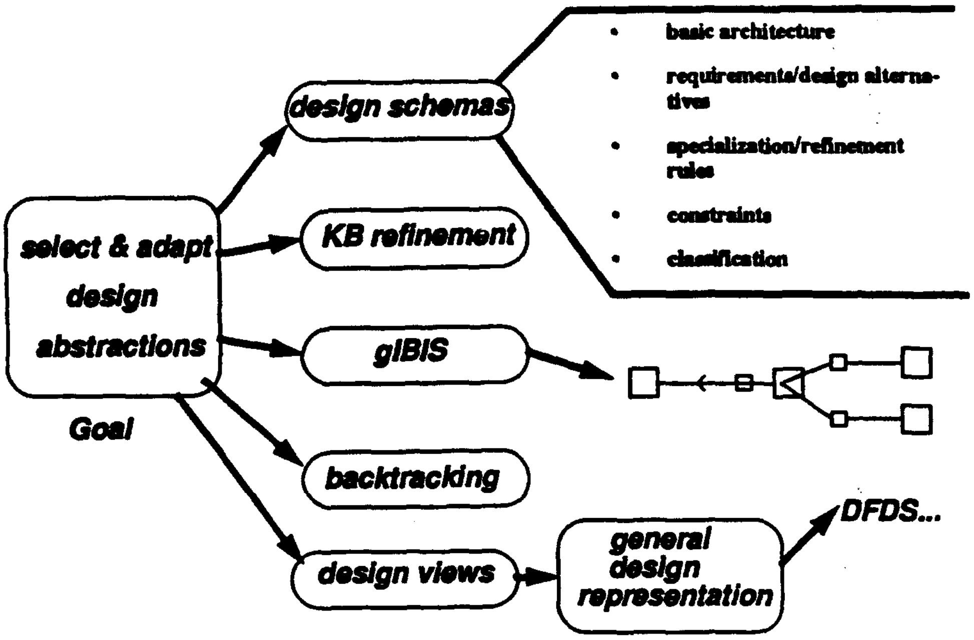
modeling domains

executable

Draco: Refinement Process

The Draco refinement process begins with specifications only in the application domain (e.g Command Center domain) and gradually fleshes out the design by refining/transforming components from the modeling domain (e.g. DBMSs, Geographical Information Systems, Message Processors...) until all requirements are fulfilled by executable code.

ROSE-2: Key Concepts



select & adapt design abstractions

Goal

design schemas

- basic architecture
- requirements/design alternatives
- specialization/refinement rules
- constraints
- classification

KB refinement

gIBIS

backtracking

design views

general design representation

DFDS...

Strategies

Rose-2: Key Concepts

Reuse of Software Elements (ROSE-2) developed by MCC: A key objective in software design reuse is to provide mechanisms that help the user select and adapt design abstractions to solve software problems. To achieve this objective the user should be presented with clear requirements and design alternatives that he/she can choose from to solve problems.

Five Strategies:

- use design schemas to represent abstract reusable design solutions
- organize requirements and design alternatives into issue-based structures (IBIS)
- develop and customize designs using the knowledge-based refinement paradigm
- use dependency-directed backtracking to support design exploration
- present multiple design views to enhance the reuse and evaluation of designs.

References:

- M. R. Lowry and R. D. McCartney.1991. *Automating Software Design*. California: AAAI Press. [Chapter 5]
- Lubars, M.D. 1989. A General Design representation, Technical Report STP-066-89, MCC Corp., Austin, Texas.
- Lubars, M. D. 1991. Representing Design Dependencies in an Issue-Based Style. *IEEE Software* July 1991: 81-89. Washington, D.C.: IEEE Computer Society Press.

Rose-2: Key Concepts

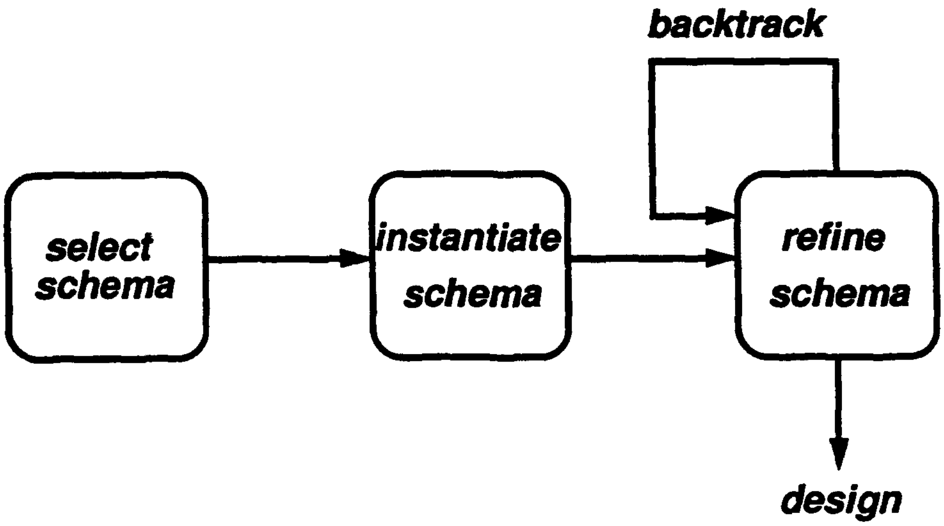
Design Schemas contain the following elements:

- basic architecture for constructing systems of a general form
- a set of requirements and design alternatives that specify which customizations can be applied to the design
- a set of specialization rules that select among alternative design customizations
- a set of refinement rules that perform specific design customizations
- a set of constraints that enforce dependencies between different requirement and design decisions
- classification information to assist in selecting design schemas from a reuse library.

Multiple Design Views

- General Design Representation (developed by MCC) is used as the base design representation from which the other design views can be displayed
- State-transition diagrams and state charts (to answer state- and event-oriented questions)
- Real-time structured analysis representations (to answer data-flow and control-flow oriented questions)
- Structural views (to answer questions about subsystems and lower-level system components)

ROSE-2: Process



backtrack

select schema

instantiate schema

refine schema

design

Rose-2: Process

Schema-based process of Reusing Design

- *Schema selection*
  - choose a design schema from a library that matches a given set of user requirements
- *Schema instantiation*
  - create an instance of a selected design schema based on the given user requirements
- *Schema refinement*
  - supply additional requirements and design decisions to further guide the refinement and customization of the design

Knowledge-Based Refinement Paradigm

The selection of design schemas and the application of refinement rules to semiautomatically customize design based on user requirements is a software development process called the Knowledge-Based Refinement Paradigm.

Rose-2: Process

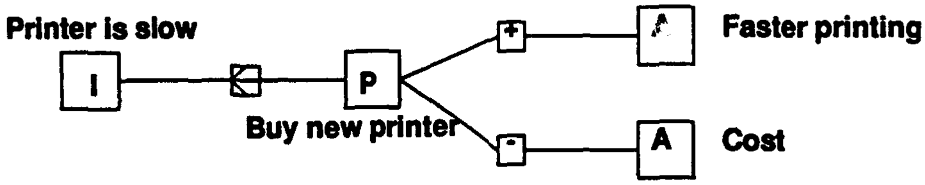
Advantages of Knowledge-Bases Refinement Paradigm:

- helps to reduce the size and complexity of user-supplied software requirements by supplementing them with detail from the design schemas
- helps assure that complete and consistent requirements are provided by checking them against constraints and issue structures in the design schema
- helps partially automate software design construction by applying the schema's refinement rules
- helps support software specification and design as parallel and complementary activities by refining design in direct response to user-supplied requirements
- helps support software design reuse as an integral part of the design process

Design Exploration and
Dependency-Directed Backtracking

Allow the user to supply and retract different requirements and design decisions and observe the effects as different sets of refinement rules are applied to customize the design

Rose-2: Issue-Based Information System Structures(IBIS)



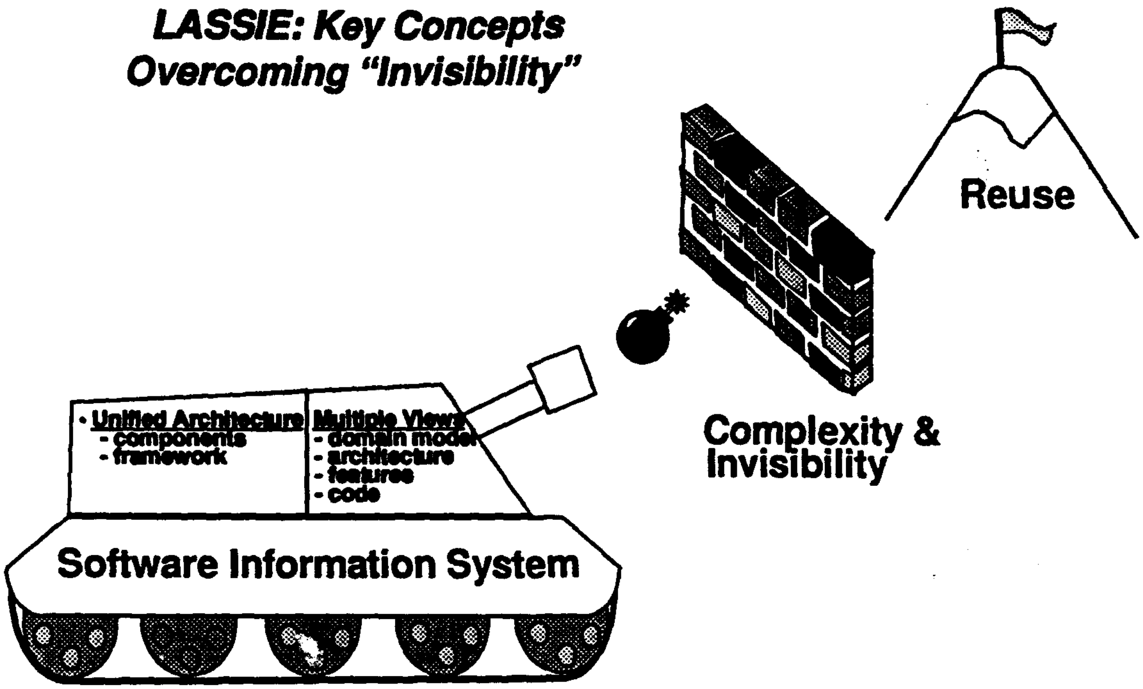
Rose-2: Issue-Based Information System Structures(IBIS)

- *Requirements and design questions are formulated as issues*
- Alternatives for resolving the issues (specific requirements or design decisions) are formulated as *positions*
- Each of these positions can be supported by, or objected to, by *arguments*.

Representational goal in design reuse is to incorporate the IBIS method into design schemas and design reuse mechanisms so that the following requirements are met:

- Requirements and design alternatives are clearly presented to the user as he/she attempts to reuse and customize designs
- The user can examine the relative benefits and disadvantages of the various alternatives.
- The design history can explicitly be recorded and examined as the user chooses alternatives, and the design is subsequently customized

LASSIE: Key Concepts
Overcoming "Invisibility"



Reuse

Complexity & Invisibility

• Unified Architecture
- components
- framework

Multiple Views
- domain model
- architecture
- features
- code

Software Information System

LASSIE: Key Concepts

Brooks identified two problems in software development: Complexity and Invisibility. LASSIE is intended to exploit knowledge-representation as a means of attacking these two problems.

- Complexity: Software is relatively complex compared to other constructs because no two parts are alike, and scale-up is non-linear
- Invisibility: The structure of software, unlike buildings or automobiles, is hidden and difficult to visualize. Execution behavior is the way we generally get behavioral data

The developer's burden is to determine whether something has been done before and how to make it conform to the architecture. But:

- Invisibility leads to violations of the architecture.
- Architecture violations create more irregularities, therefore more complexity.
- Increased complexity intensifies invisibility. And so on . . .

This also hampers reuse and fosters wasteful reimplementations, which in turn exacerbate invisibility and complexity and erode integrity.

Inivisibility is also manifested by a "discovery phenomina:"

- what a developer or maintainer must do to prepare for the actual task
- takes approximately 50% of *all* developer's time
- is a trail of inquiries to gain understanding of the system at hand.
- Visual displays are not effective; even graphs don't simplify things much. Documents are rarely up-to-date and correct and complete and available and oriented towards discovery. Knowledge largely resides in experts who may not be available or willing; may have to re-establish context; may not explain well.

LaSSIE Key Concepts

References:

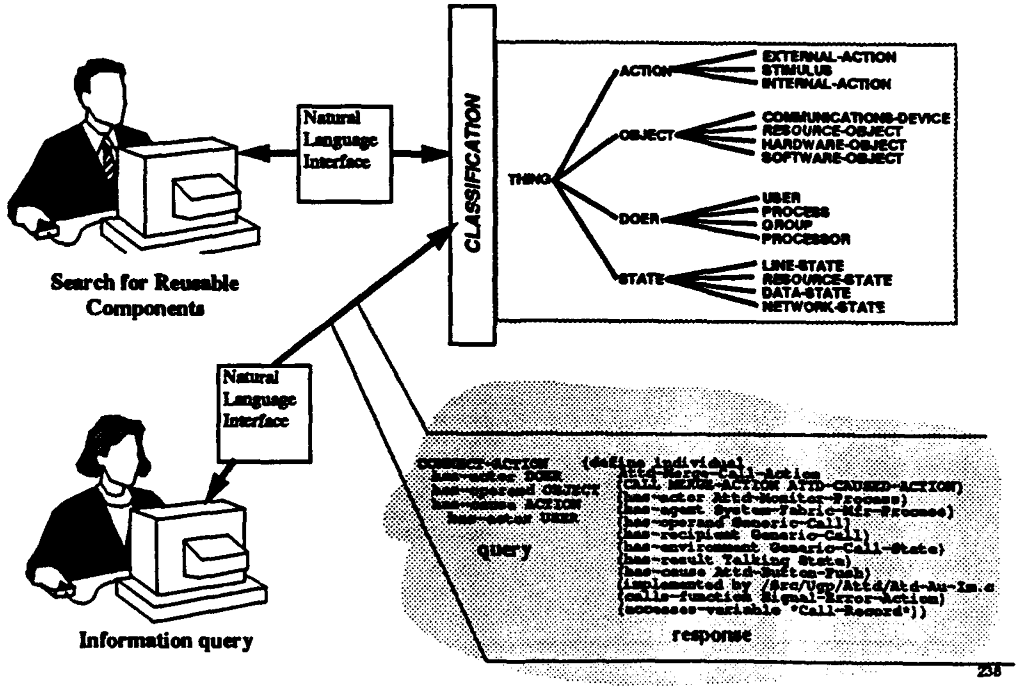
Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge and Bruce W. Ballard.
LaSSIE: A Knowledge-Based Software Information System
Communications of the ACM, May 1991, pp. 34-49.

Peter G. Selfridge
Knowledge Representation Support for a Software Information System
Proceedings of the 7th Conference on Artificial Intelligence Applications
February 24-28, 1991 Miami Beach, FL, Volume I: Technical Papers, pp. 134-140; Volume II: Visuals, pp. 271-291.

Peter G. Selfridge, Loren G. Terveen and M. David Long
Managing Design Knowledge to Provide Assistance to Large-Scale Software Development
Proceedings of the 7th Knowledge-Based Software Engineering Conference
September 20-23, 1992 McLean, VA, pp. 163-170.

P.F. Patel-Schneider, R.J. Brachman and H.J. Levesque
Argon: Knowledge representation meets information retrieval
Proceedings of the First Conference on Artificial Intelligence Applications
1984, pp. 280-286.

LaSSIE: Key Concepts



```
CONNECT-ACTION
  has-actor DOER
  has-operand OBJECT
  has-cause ACTION
    has-actor USER
```

query

```
(define individual
Attd-Merge-Call-Action
(CALL MERGE-ACTION ATTD-CAUSED-ACTION)
(has-actor Attd-Monitor-Process)
(has-agent System-Fabric-Mir-Process)
(has-operand Generic-Call]
(has-recipient Generic-Call)
(has-environment Generic-Call-State)
(has-result Talking State)
(has-cause Attd-Button-Push)
(implemented by (Src/Ugp/Attd/Atd-Au-Im.c
(calls-function Signal-Error-Action)
(accesses-variable *Call-Record*))
```

response

LaSSIE: Key Concepts

Knowledge Base: Emphasis is on capturing the semantics of the *actions* and *objects* of the architecture. Support is provided for complex questions involving architectural, conceptual and code views without knowing structure of Knowledge Base.

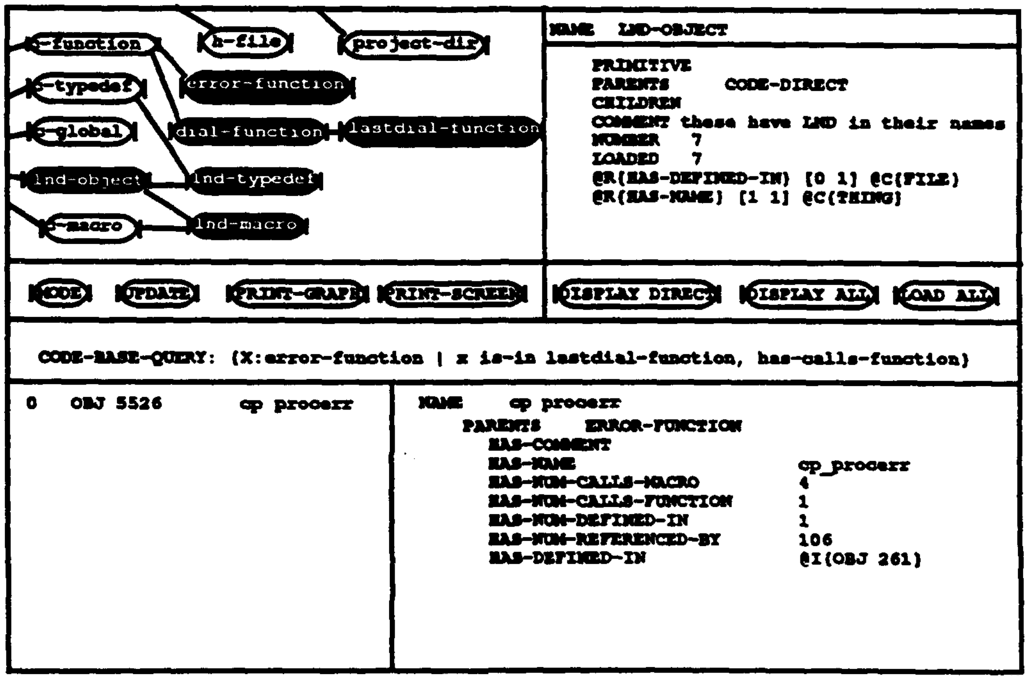
User Interface: Provides easy access at a conceptual level via a window/mouse interface. Provides "Query by Reformulation" (Patel-Schneider, Brachman & Levesque).

Knowledge Representation: Frame-based system with inheritance, which offers economy of representation and semantic integrity. Retrieval "hits" are instances of the frames subsumed by the query.

Example: System recognizes that `MERGE-ACTION` is a `CONNECT-ACTION` based on the descriptions of each. It also realizes from the description (not shown here) of `Attd-Button-Push` that this is an `ACTION` by an `ATTENDANT`, which is defined to be a specialization of `USER`. The Argon-like user interface displays the retrieved individuals, from which the user can select one for detailed display, with all its slots and fillers, each of which itself can be selected for further display.

Limitations: Action-based representation does not help the developer establish the contexts in which the actions are performed - no map of the territory. Plan-oriented questions like: "Why is this operation being performed?" are not supported. Knowledge acquisition is essentially manual.

LaSSIE: Related Tools - CODE-BASE



```
NAME  LND-OBJECT

    PRIMITIVE
    PARENTS      CODE-DIRECT
    CHILDREN
    COMMENT these have LND in their names
    NUMBER   7
    LOADED   7
    @R(HAS-DEFINED-IN) [0 1] @C(FILE)
    @R(HAS-NAME) [1 1] @C(THING)
```

```
CODE-BASE-QUERY: (X:error-function | x is-in lastdial-function, has-calls-function)
```

```
0   OBJ 5526       cp procerr
```

```
NAME     cp procerr
     PARENTS     ERROR-FUNCTION
       HAS-COMMENT
       HAS-NAME                          cp_procerr
       HAS-NUM-CALLS-MACRO               4
       HAS-NUM-CALLS-FUNCTION            1
       HAS-NUM-DEFINED-IN                1
       HAS-NUM-REFERENCED-BY             106
       HAS-DEFINED-IN                    @I(OBJ 261)
```

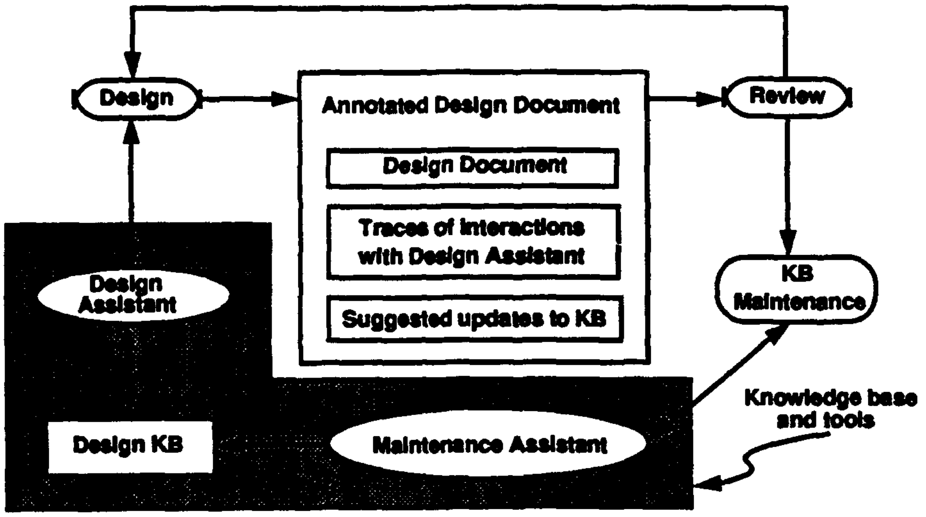
LaSSIE: Related Tools -CODE-BASE

Complementary to LaSSIE: LaSSIE supported semantic-based discovery in a hand-coded domain model.Goal is to extend conceptual model to incorporate a code model and provide meaningful links between them. CODE-BASF represents code-level information (at the level of a construct such as a procedure, function or declaration) which is automatically acquired, thus guaranteeing synchronization of KB with the code. The user interface allows posing of specific queries as well as "hypertext" style traversal. Thus we see a reverse engineering type tool supporting a reuse tool.

CODE-BASE: example:

- Upper-Left Panel : Browse the concept hierarchy
- Upper-Right Panel : Examine an individual concept
- Middle Panel : Where CODE-BASE queries are entered
- Lower-Left Panel : Display instances which match the query
- Lower-Right Panel : Display a selected instance

LaSSIE: Related Tools - Design Assistant



Design

Annotated Design Document

Design Document

Traces of interactions with Design Assistant

Suggested updates to KB

Review

KB Maintenance

Design Assistant

Design KB

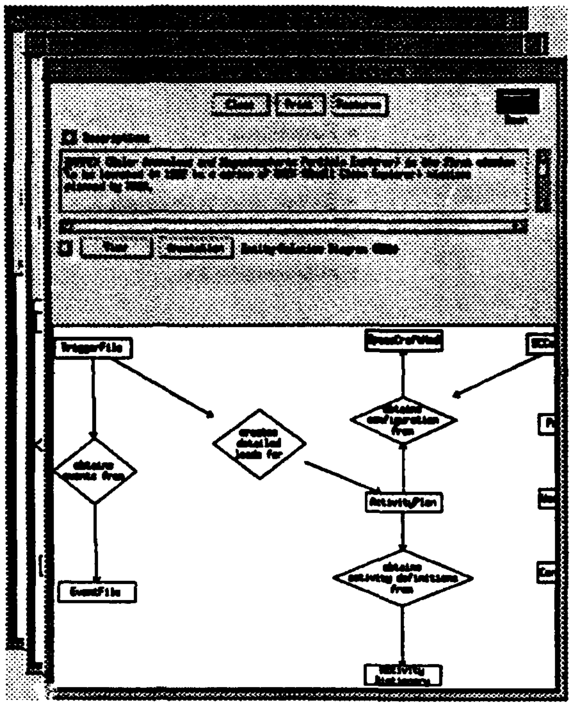
Maintenance Assistant

Knowledge base and tools

LaSSIE: Related Tools - Design Assistant

- Need to capture the folklore which is not documented and remains accessible only through human experts.
- To manage such knowledge with automation we must deal with: difficulty of acquisition, representation and accessibility, and maintainence of design knowledge
- Can't assume a "lump sum" single occurrence capture of all the knowledge. Need facilities to capture elaboration and evolution of design. Need facilities to capture new knowledge arising from normal design and review activities.
- Taxonomy of design problems with associated advice items which: removes redundancy and facilitates an advice exception (i.e., override) mechanism. KB is accessed by a design assistant program which manages the system/user dialog.
- Maintenance via the incorporation of design advice into the design,
  so it is also subject to the normal organizational review process.

KAPTUR: Key Concepts


TriggerFile
obtains events from
creates detailed loads for
obtains configuration from
ActivityPlan
obtains activity definitions from
EventFile

- Tool Supported Methodology Developed by CTA, Inc. with NASA Sponsorship
- Advocates a Case-Based Reasoning Approach to Domain Analysis (i.e. Case-Based Domain Analysis) Which Combines:
  - object-oriented modeling,
    - class, objects
    - inheritance
    - methods (services, functions)
    - variables (attributes)
    - message passing
  - feature modeling, and
    - extension of SEI FODA by including both visible and non-visible user features.
  - case-based reasoning
- Captures (hence KAPTUR) Domain Products, Legacy Systems, Features, Design Trade-offs and Rationale
- Follows a Supply Side <-> Demand-Side Cycle to Domain Analysis and Systems Analysis
- Supports Various Architectural Perspectives

KAPTUR was developed by CTA Incorporated under NASA sponsorship.

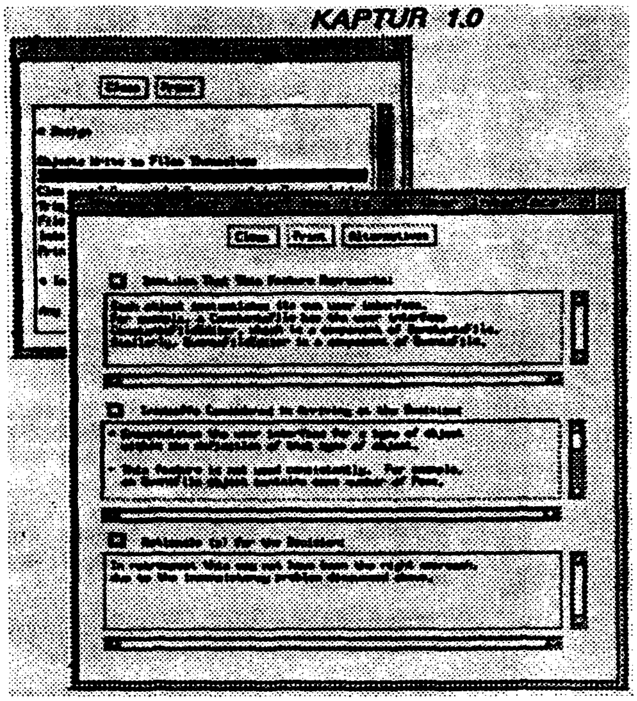
KAPTUR is a tool that is used in conjunction with an entire domain analysis process that begins with identifying and scoping a domain, capturing and analyzing domain information, creating a validated domain model, and using the knowledge captured and modeled to generate new systems in the domain using the knowledge gained from the legacy systems in the domain. KAPTUR is the tool used to organize and structure the information relative to the domain, as well as document decisions made in the development of domain systems.

The supply side of the KAPTUR process (and model) involves the accumulation of domain knowledge, organization of that knowledge, and knowledge placement in the KAPTUR tool. The supply side person is like a domain manager, a domain owner, or a domain developer - an expert in the domain and the person who creates the representations of the legacy systems in the domain. This person takes the perspective that components need to be reused and can distinguish the features or characteristics that make a component reusable. The demand side of the KAPTUR process (and model) involves the use of domain knowledge as it applies to a new system.

References

CTA Incorporated
6116 Executive Boulevard
Rockville, MD 20852

KAPTUR: Tool Functionality and Representations

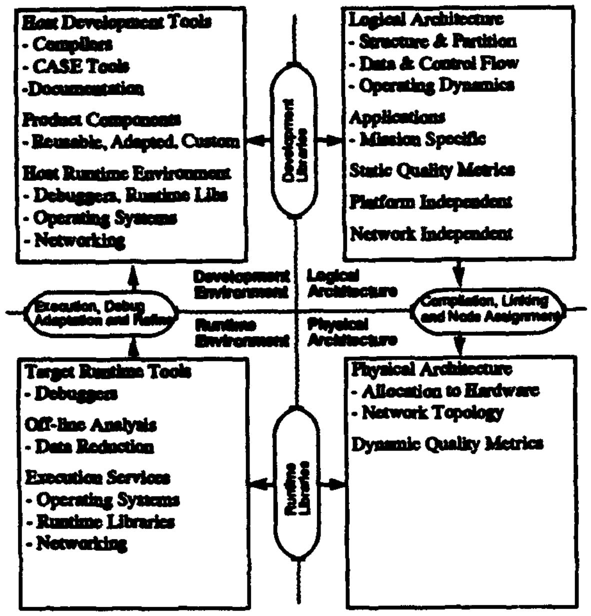


- Tool has direct support for capturing trade-offs and rationale for:
  - Operational Features
  - Interfaces
  - Functions
  - Performance
  - Development Methodology
  - Design
  - Implementation
- Alternative Architectural Views allow different perspectives of system via:
  - Entity-Relationship Diagram
  - Data Flow Diagram
  - Object Communication Diagram
  - Stimulus-Response Diagram
  - State Transition Diagram
  - Assembly Diagram
  - Classification Diagram

Descriptive information is available for each architecture and annotations (descriptive information) are available for each element in an architecture view. Associated with each architecture is a set of features and with each feature is information dealing with the decision that feature represents, the trade-off associated with the decision, and rationale for the decision made. Any feature may or may not be present in any of the architectures. To the extent that a feature is in one architecture and not in another indicates alternative implementations that a user may need to consider. Based on the presence or absence of a feature, the user may need to go and look at an alternative architecture, again looking at the features, decisions, trade-offs, and rationale information.

KAPTUR is a tool used to represent software architectures in support of object-oriented modeling. KAPTUR has several ways in which to represent the objects analyzed. There are various architecture views (or perspectives) currently available in KAPTUR.

UNAS: Key Concepts



Host Development Tools
- Compilers
- CASE Tools
-Documentation

Product Components
- Reusable, Adapted, Custom

Host Runtime Environment
- Debuggers, Runtime Libs
- Operating Systems
- Networking

Development Libraries

Logical Architecture
- Structure & Partition
- Data & Control Flow
- Operating Dynamics

Applications
- Mission Specific

Static Quality Metrics

Platform Independent

Network Independent

Execution, Debug Adaptation and Refine

Development Environment

Logical Architecture

Runtime Environment

Physical Architecture

Compilation, Linking and Node Assignment

Target Runtime Tools
- Debuggers

Off-line Analysis
- Data Reduction

Execution Services
- Operating Systems
- Runtime Libraries
- Networking

Runtime Libraries

Physical Architecture
- Allocation to Hardware
- Network Topology

Dynamic Quality Metrics

- **Universal Network Architecture Services (UNAS) — developed by TRW**
  - **A Process-based, Asynchronous, Message-driven Language Framework for Rapidly Developing Distributed Applications**
  - **A Collection of Integrated Tools to Support the Development and Management of Distributed Applications**
  - **A Software Architectural Design Paradigm**
- **Standard, Integrated Development Environment**
  - **Compilers, CASE Tools, Debuggers**
- **Software/System First Mentality**
  - **Promotes the development of the "logical" architecture first, which is later mapped to the "physical" architecture (architectural elements are allocated to hardware)**
- **Standard, Integrated Runtime Environment**
  - **Runtime and off-line analyzers, network resource management, runtime libraries**

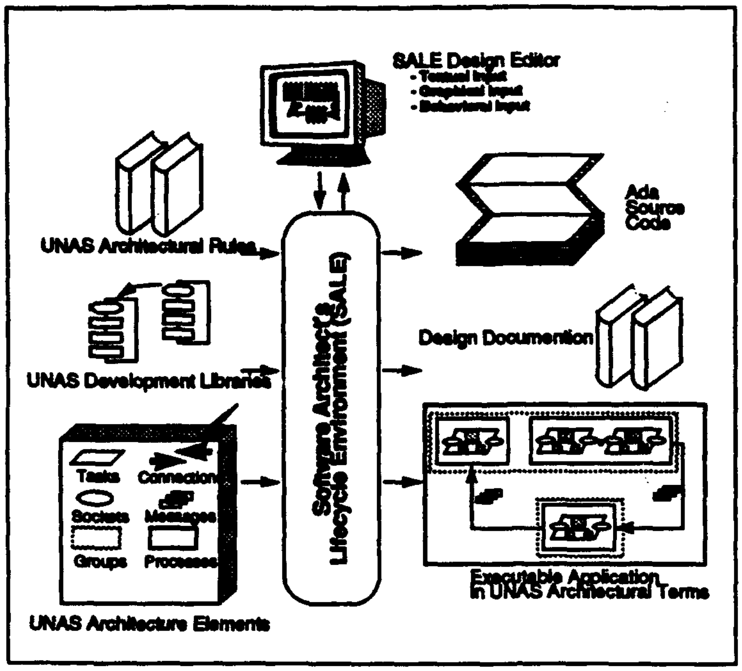
At it's core, UNAS can simply be defined as a high level language for building software architectures. This language is targeted for applications (potentially distributed and potentially heterogenous) based on a message driven paradigm. However, in addition to being a language there exists a highly integrated collection of tools and services which support the development of UNAS distributed applications and the runtime management of those applications. These tools and language, together, define UNAS's architectural design paradigm which is supported by architectural representation, rules for assembling elements of UNAS elements and tools to enforce that paradigm.

UNAS development environment permits the architect to built the system first without actually being concerned with the underlying physical implementation or hardware. This "logical" architecture can be defined in terms including performance, structure and control & data flow and then executed to establish metrics with which to perform comparative analysis against expected and actual results. Architectural elements can be assigned and allocated to the target hardware environment, thus instantiating the "physical" architecture from the "logical".

References

UNAS Training Class, July 7-9, 1993
TRW Systems Engineering & Development Division
DH2/1271
Carson, CA

UNAS: Tool Functionality



- UNAS:
  - defines basic architectural elements and rules for their interconnections
  - provides messaging between and control & management of those elements
- SALE — frontend: (UNAS CASE Tool Option)
  - GUI for building UNAS distributed applications
  - Enforces UNAS software architecture concepts — relationships and rules
- SALE — backend:
  - Ada code generation utilizing underlying UNAS services
  - Load and performance models from behavioral input to SALE
  - System documentation generated from textual and graphical input to SALE

UNAS's CASE Tool option, SALE — System Architect's Lifecycle Environment, enforces UNAS's methodology for build software architectures from UNAS elements. SALE's graphical user interface allows the architect to instantiate architectural elements and inter-connect and group them to form larger components in a consistent manner. As the architecture is created, SALE's accepts expected or required performance metrics and design considerations of tasks and processes in the system.

As a back-end tool, SALE will generate complete compilable source code which utilize underlying UNAS inter-task communication (ITC) and generic application controls (GAC) services. Once compiled, the application built can be executed as a skeleton which will exhibit performance behavior as presented to the tool. SALE will automatically generate design documentation which describes both mission-independent and mission-dependent (that which was entered into the CASE Tool) portions of the application.

The UNAS message product, provides the basic services for Inter Task Communication (ITC) capability known as ITC services and automatic heterogeneous data translation. Data structures written in Ada source are converted to meta-message format via UNAS off-line message registration tools. The meta-message format is the mechanism that permits data conversion between heterogeneous network nodes. Further, ITC services provides that Ada package generic to ensure Ada's strong type cohesion between the distributed process which write and read passed messages.

Other services of ITC include error reporting and propagation, task creation, interactive network management, SNMP interface to network management, and message interjection and recording.

UNAS's Generic Application Control (GAC) is a higher level of abstraction of the services provided by ITC. Pragmatically, GAC removes the application developer from many of the "quirks" and details of the ITC layer as well as adding buffered I/O to network message passing, message queuing, logical separation of nodes, processes and sockets from their physical implementation, and built-in performance and utilization. Additionally, exception handling, error reporting and logging is greatly enhanced and abstracted in the GAC layer.

UNAS: Representations

UNAS Architecture Paradigm

- **Basic elements:**

▱ ***Tasks*** **exchanging messages**

▭ ***Messages*** **are exchanged over interconnected sockets**

⬭ **A** ***Socket*** **is a names source/destination associated with a task**

➔ ***Connections*** **are paths between sockets controlling message flow**

▭ **Tasks are organized into** ***Processes*** **for control and re-configuration**

⬚ **Processes are combined into** ***Groups*** **for operational uniqueness**

These are the most basic architectural elements that can be used to build a UNAS application. Processes are made up of one or more tasks which communicate over one or more sockets connected to other tasks (or itself in the case of timer sockets). Messages are used to communicate data over interconnected sockets. Sockets can be connected together via connections (or circuits in earlier UNAS terminology) and can be either read-only, write-only, or read-write. The figure below shows an example of a simple UNAS applications in terms of these elements.

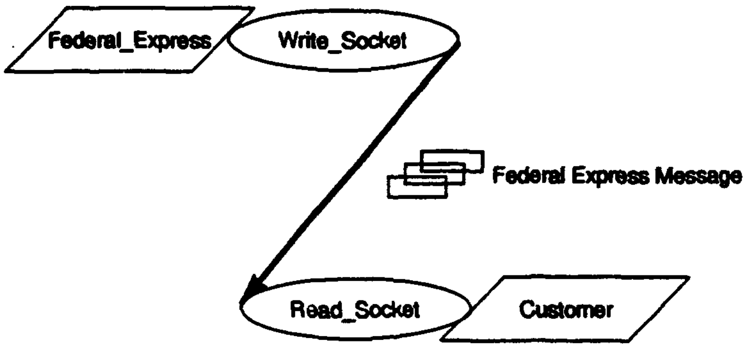
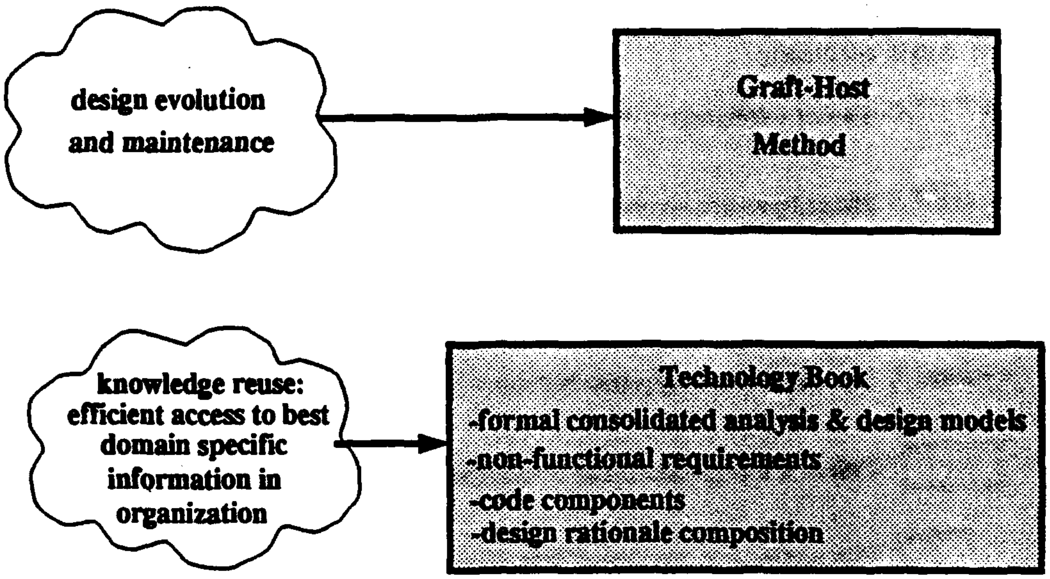


Figure: Sample UNAS Application

In the above figure, the task "Federal_Express", communicates via a "Write_Socket" with another task called "Customer". The message that is passed from the first task to the second task can only be of type "Federal Express Message". In this example, either task can belong to a different process, or they could be two tasks in the same process.

Technology Book: Key Concepts



design evolution and maintenance

Graft-Host Method

knowledge reuse: efficient access to best domain specific information in organization

Technology Book
-formal consolidated analysis & design models
-non-functional requirements
-code components
-design rationale composition

Technology Book: Key Concepts

The tools presented in this section are based on the approach used at Schlumberger.Design evolution and maintenance are the dominant activities in many software development organizations. Thus, the reuse of analyses and designs is of greater benefit than the reuse of software. To obtain this benefit the engineering environment should provide a Domain-specific information workspace and efficient access to the best information available in the organization. The approach is:

- *Domain analysis* to consolidate critical analysis and design information for product families.
- *Representation* of reusable information in structured form via 'Technology Books'.
- *'Graft-Host'* method for reusing design information and managing databases of constraints in a systematic and reliable way.

References: Guillermo Arango, Eric Schoen and Robert Pettengill
A Process for Consolidating and Reusing Design Knowledge
Proceedings of The 15th International Conference on Software Engineering
May 17-21, 1993 Baltimore, MD, pp. 231-242.

Guillermo Arango, Eric Schoen, Robert Petengill and Josiah Hoskins
The Graft-Host Method for Design Change
Proceedings of The 15th International Conference on Software Engineering
May 17-21, 1993 Baltimore, MD, pp. 243-254.

Guillermo Arango, Eric Schoen and Robert Pettengill
Design as Evolution and Reuse
Proceedings of the Second International Workshop on Software Reusability
March 24-26, 1993 Lucca, Italy

Technology Book Use: Finding an Algorithm

Technology Book Use: Finding an Algorithm

- In (1) the user finds there are three choices of algorithm for computing the CRC using the taxonomic relationship.
- Then in (2) she follows an analysis of their space and time properties.
- She identifies multiple combinations of generator polynomials and algorithms in (3).
- Finding that she must use CRC-16 to maintain backwards compatibility, inspects the CRC-16 Pattern Algorithm in (4).
- She finds in (5) the required polynomial coefficients via a uses relationship.
- Finally, she inspects a mathematical description of the algorithm in (6) via a documentation link.

Technology Book Use: Finding an Algorithm Implementation

Technology Book Use 2: Finding an Algorithm Implementation

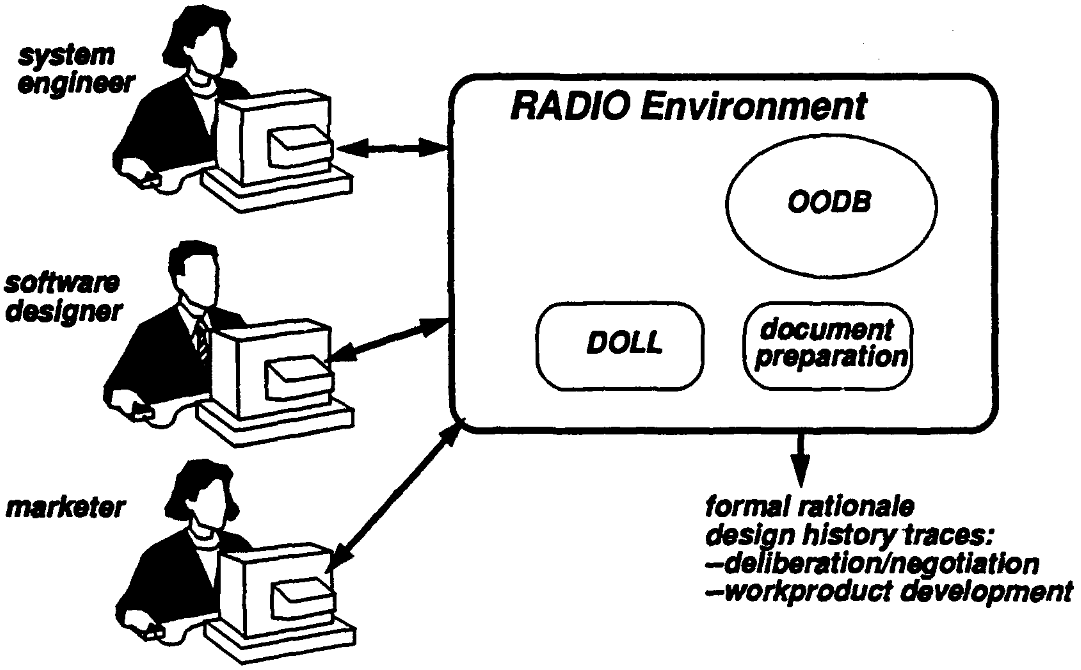
- User selects the algorithm in (1).

- The implementation relation graph (2) shows there are three implementations of the CRC-16 pattern algorithm.

- Designer selects a C-language implementation in (3) and browses the source code.

- She also views the detailed documentation in (4) for the chosen implementation.

Technology Book: Tools



formal rationale
design history traces:
–deliberation/negotiation
–workproduct development

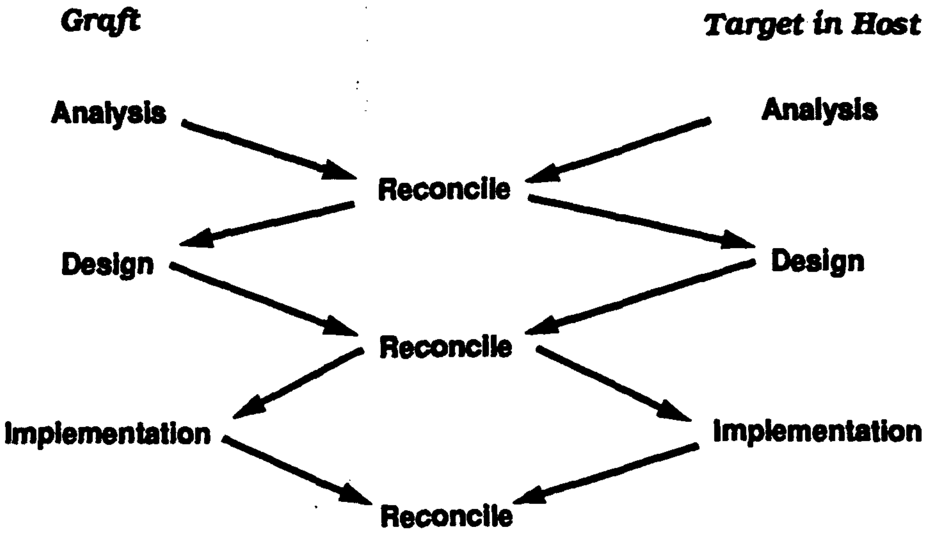
Technology Book : Tools

The representation is a compromise between usability and formality:

- **Semantic Tags:** issue, definition, assumption, imported constraint, exported constraint, position, design decision, unresolved, result.
- **Syntactic Tags:** authors, headings, equations, enumerations.
- **Information is stored in typed nodes and relations between them.**
- **Information nodes are organized into taxonomies by type:** domain entities, project entities, work products, resources, statements, analyses.
- **Relations include :** history, taxonomy, derivation, aggregation, use, justification, interconnection, ownership - as determined by domain.

- RADIO Environment (with Motif-based GUI) includes: Object oriented DBMS, DOLL (Modeling Language), and Document Preparation.
- RADIO: provides Browser / Editor for: depicting book contents, navigation, and updating.
- DOLL: emphasizes descriptiveness and runtime flexibility, not runtime speed or storage minimization. Nonetheless it provides subsecond response time.Informal elements (text, pictures, tables, equations) are stored as Framemaker attachments to DOLL objects.

Technology Book: Graft-Host Method



Graft

Target in Host

Analysis

Analysis

Reconcile

Design

Design

Reconcile

Implementation

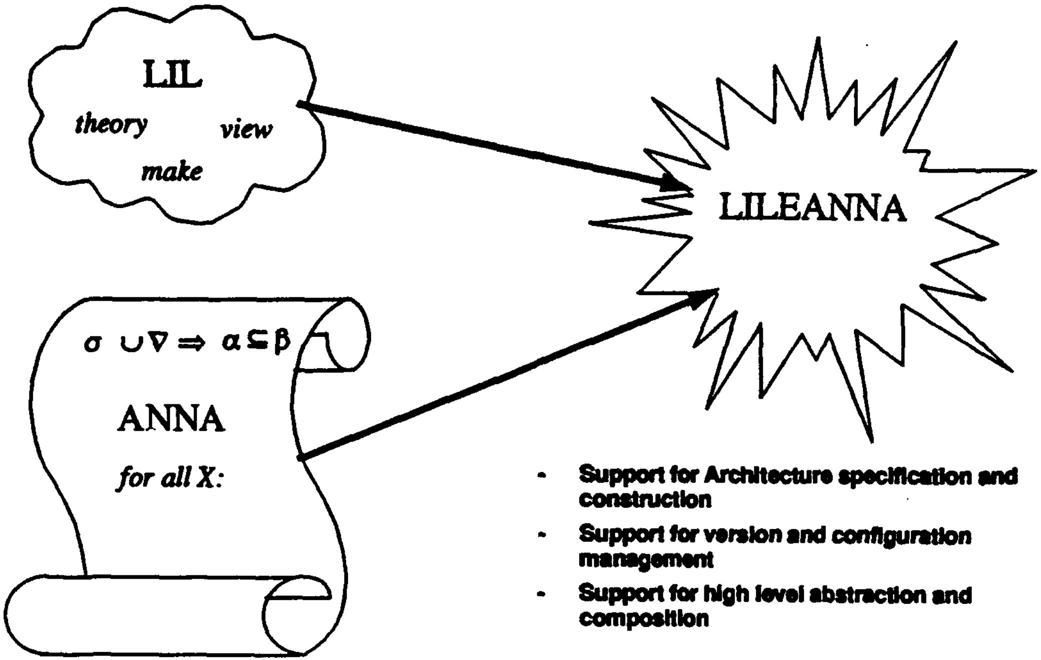
Implementation

Reconcile

Technology Book: Graft-Host Method

- Helps make design constraint management a systematic and reliable process.
- ***Host*** **: System to be changed.**
- ***Target*** **: Subset of the Host affected by the change.**
- ***Graft*** **: Proposed substitution for the target.**
- Reduces risk in change by providing guidance for developing change plans.
- Reduces need (via technology books) for designers to rediscover design rationales.
- Fewer design iterations; more errors caught more early.
- Shorter training times for engineers and maintainers.

LILEANNA: Key Concepts



- **Support for Architecture specification and construction**
- **Support for version and configuration management**
- **Support for high level abstraction and composition**

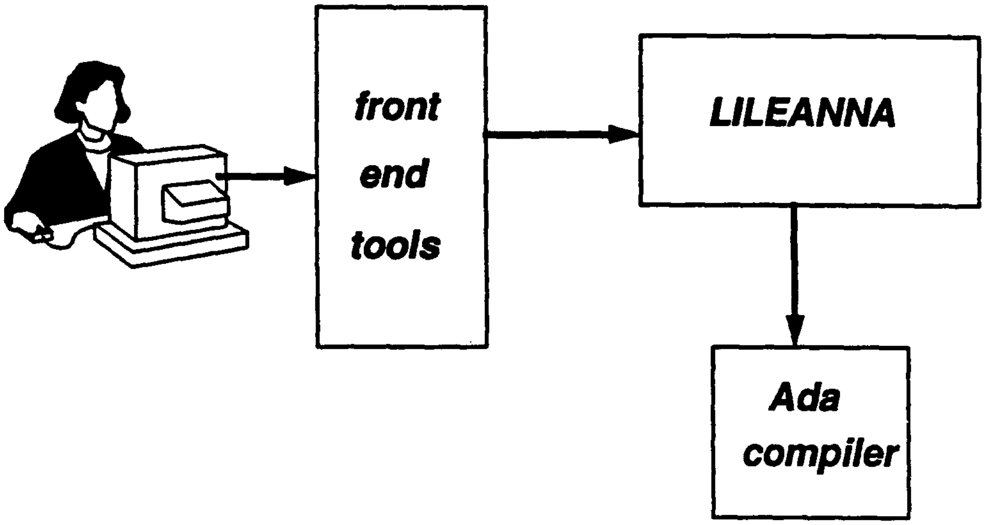
LILEANNA: Key Concepts

- LILEAnna is a Library Interconnect Language Extended with Annoted Ada, which is intended to support high level abstraction, composition and reuse of Ada Software. LILEANNA supports the design of parameterized components and software architectures.
- The language was designed to allow certain automated analyses based on formal specification of preconditions (using the Anna toolset); Automated selections, composition, tailoring and instantiation of Ada code from LILEAnna specification and pre-existing Ada code.
- LIL and Anna were pre-existing languages that have been refined and merged.
  - LIL is language for designing, structuring, composing, and generating software systems.
  - Anna is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. It is designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well-established formal methods of specification and documentation can be applied to Ada programs.
- References: Tracz, W. "A conceptual model for megaprogramming" ACM SIGSOFT SEN July 1991
- Tracz, W., "LILEANNA: A Parameterized Programming Language" in Proceedings of the Second International Workshop on Software Reusability, March 24-26, 1993, Lucca, Italy
- Goguen "Reusing and Interconnecting Software Components" in Domain Analysis and Software system Modeling Prieto-Diaz and Arango
- Luckham and von Henke "An overview of Anna a Specification Language" for Ada " IEEE Software March 1985

LILEANNA: Key Concepts

- LILEANNA provides mechanisms to specify abstraction and composition of Ada packages. It has the Look-and-Feel of a language that extends the existing Ada packages specifications, instantiations and dependency mechanisms. LILEAnna extends Ada by introducing two entities: theories and views, and enhancing a third, package specifications. It introduces [generic] theories, which provide a formal specification of functionality. It also introduces [generic] packages as abstraction for Ada [generic] packages, which can serve as an abstraction for multiple Ada packages or implementation
- Supports Architecture specification and construction of a executable Ada application with two features VIEWS and MAKES.
  - VIEWS allows users to specify how generic parameters, exported services, and (for LILEAnna packages) imported services are bound to (provide by) other LILEAnna theories, LILEAnna packages, Ada packages, and the objects exported by them.
  - MAKES allows users to specify how Ada packages can be composed and instantiated to form other Ada packages, where VIEWS can be used to refined and control this process.
- Both VIEWS and MAKES allow partial bindings, which if carried through the MAKES process results in Ada generic packages.
- Existing packages may be manipulated through packages expressions specify the instantiation, aggregation, renaming, additions, elimination or replacement of operations, types or exceptions.
- Provides support for version and configuration management
- Provides multiple controlled inheritance
- Supports the structuring and composition of software modules from existing modules.

LILEANNA: Tools



front end tools

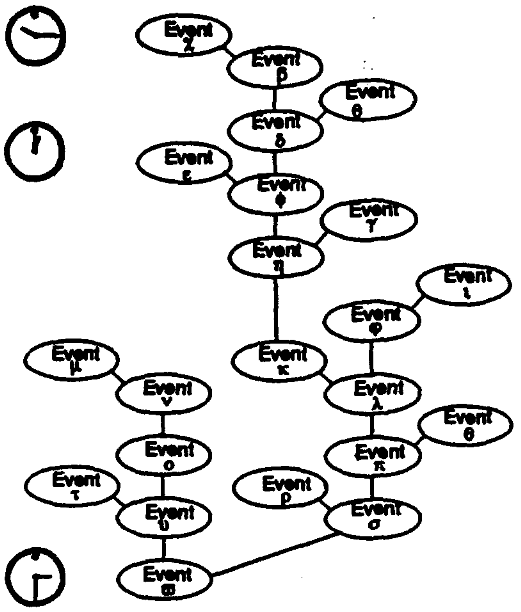
LILEANNA

Ada compiler

LILEANNA: Tools

LILEANNA can be programmed in directly or used with a variety of front end tools. The LILEANNA translator is part of the IBM DSSA Avionics Domain Application Generation Environment. A graphical composition front end tool has also been proposed.

μRapide: Key Concepts



- **Executable architecture definition language.**
- **Models time-sensitive, concurrent and distributed hardware and software systems.**
- **μRapide Features include**
  - **event patterns**
  - **interfaces**
  - **architectures**
  - **mappings**
- **Tool Supported:**
  - **CPL - Common Prototyping Language front-end compiler which translates μRapide source code into Ada.**
  - **IRS - Illustrated Run-time System for the viewing and printing of partially ordered event traces generated by μRapide computations.**
  - **POB - Partially Ordered event trace Browser for the viewing of μRapide computations as they occur.**

μRapide: Key Concepts

Event patterns are expressions that define sets of events and their dependency and timing relationships. An *event* signifies an activity during system execution. Event patterns contain information such as threads of control, data values, time interval; modelled as a tuple of values. Execution of a distributed system is modelled as a partially ordered set of events, called a *poset*, based on causality or timing relations.

An *interface* gives an external view of the behavior of a type of component and defines how components of its type react to events by changing state and generating new events. Members of a component type are called *objects*.

Architectures define the flow of events between interfaces. An architecture consists of a set of components (objects of some interface types) and a set of rules. These define how the components communicate by sending each other events or calling each other's functions. Communication rules are defined using event patterns.

Mappings define how architectures are related. One can then define how events in one system correspond to events in another. In the domain of design hierarchies refinement maps serve to express complex low level simulations as behaviors of a higher level. The mapped behavior is much smaller and simpler.

References

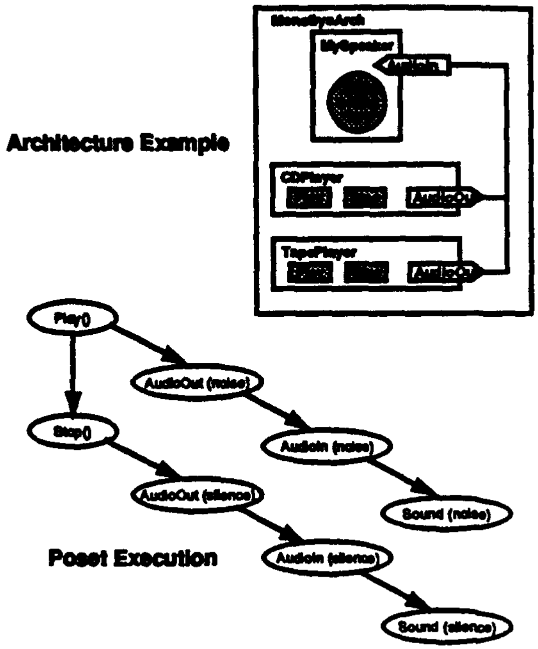
David C. Luckham and James Vera μRapide: An Executable Architecture Definition Language April 7, 1993

David C. Luckham and James Vera Event-Based Concepts and Language for System Architecture March 16, 1993

Rapide-0.2 Language and Tool-Set Overview Doug Bryan February, 1992

These and related papers are available via anonymous ftp to anna.stanford.edu. in /pub/Rapide

μRapide: Example



- MySpeaker, CDPlayer and TapePlayer are components.
    - Components can receive input events, generate output events (shaded regions) to communicate with the external environment.
    - Directed polygons indicate internal events.
- During execution, play caused a linear sequence of events, each depending on the previous, resulting in noise.
- Stop caused a linear sequence of events resulting in silence.
- There is no dependency between any event in the first sequence and any event in the second sequence.
- Given this particular execution poset, the system user invoked Play before Stop.

μRapide: Example

In this example, paths from `AudioOut` events to `AudioIn` event indicate communication from `CDPlayer` and `TapePlayer` to `MySpeaker`. So, they complete the definition of this architecture - since an architecture defines how components communicate by means of events.

Note that for this example, there are no timing constraints between any of the other events, which means there is a possible design flaw: A user could invoke Play then Stop, but still hear noise because the events depending upon Stop could overtake the events depending upon Play.

Using mapped behavior (from complex low-level systems to simpler high-level systems) yields these benefits:

- Facilitates understanding. (One application of mappings reduced the event space from 8073 to 5.)
- The formal constraints of high level architectures which capture design requirements can be automatically checked when low-level simulations are "mapped up".
- Errors in the mapped behavior can be traced back.

Architecture Based Reuse Tools: Summary

Features found in tools:

- easy access to large amounts of knowledge
  - problem space -domain specific semantics
  - solution space - architecture
- assistance - person in the loop
- method accompanies tool
- rationales and trade-offs
- composition - components and horizontal domains
- language and graphics oriented
- some tools biased toward an architectural style
- requirements, architecture, detailed design, code intermingled
- evaluation through automated analysis and simulation
- source code generation

Future?:

- integrated tool sets
- knowledge acquisition support
- cooperative design

Architecture Based Reuse Tools

It is difficult to draw a coherent picture from this or any set of tools because architecture based reuse is still an emerging area but some of the trends are clear. One major trend is the capture of large amounts of problem space (domain specific context) and solution space knowledge in organization wide knowledge bases which have user interfaces designed for easy browsing. They also provide some intelligent assistance to help apply that knowledge. There is still a tension between formal and informal representations.Another major trend is the emphasis on capturing rationales. These rationales provide clues that promote conformance to an architecture.

Some tools clearly work on the assumption of an underlying architecture style. Others allow the user to follow their own architecture style. The tools do not tend to limit themselves just to representing architecture. They often include detailed design, requirements and code.

It is difficult to predict winning trends. Tool integration will continue to be a major goal. Tools that require a lot of knowledge need to support acquisition and storage. Since designers do not work alone on large systems we should see increasing support for cooperative collaboration.

Central Archive for Reusable Defense Software (CARDS)

Session V CARDS Approach to Reuse & Software Architecture

16 November 1993

This page intentionally left blank.

Roadmap for this Session

☞ **CARDS Scientific** →
- Architecture Task Force
- Organizational Domain Modeling: Domain of Software Architecture Representation

CARDS Engineering →
- Component Qualification
- System Composition

CARDS Transition-to-Practice →
- Handbooks
- Franchising

Presentation Overview

During this portion of the seminar, the CARDS approach to Domain Engineering activities as they relate to software architectures will be discussed.

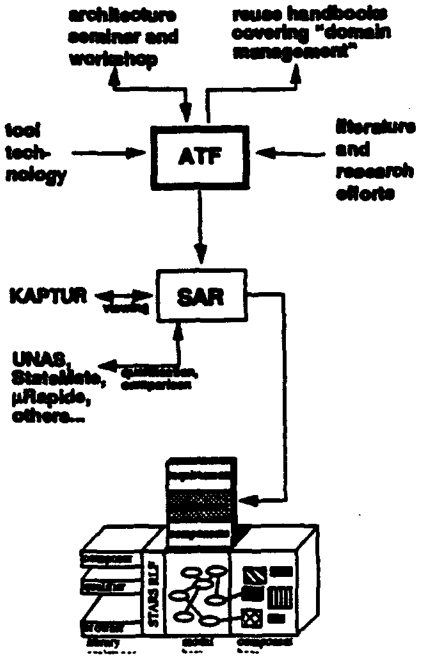
CARDS Phase 1 focused on the mechanics of Domain Engineering activities, making sure the infrastructure hardware and software and library modeling processes function correctly.

During Phase 2, CARDS focused on refining the processes of and developing prototype tools for domain specific component qualification and system composition.

For Phase 3, the CARDS focus is on architectures. An Architecture Task Force (ATF) was constituted with the goal of determining the best processes for capturing and representing architecture information in the library framework.

Throughout all the phases, CARDS has documented and transferred the information through its formal deliverables and franchising efforts.

Architecture Task Force Context & Goals



architecture seminar and workshop
reuse handbooks covering "domain management"
tool technology
ATF
literature and research efforts
SAR
KAPTUR
viewing
UNAS, Statemate, µRapide, others...
qualification, comparison
STARS RLF
library
model base
component base

ATF Goals

- Formalize the CARDS modeling approach for software architectures
  - facilitate franchise implementations
  - basis for reuse tools
- Gather and synthesize information for:
  - Reuse Adoption Handbooks
  - Evaluation of current technologies (e.g. UNAS, KAPTUR, etc.)
  - Architecture Seminar & Workshop

Architecture Task Force Context & Goals

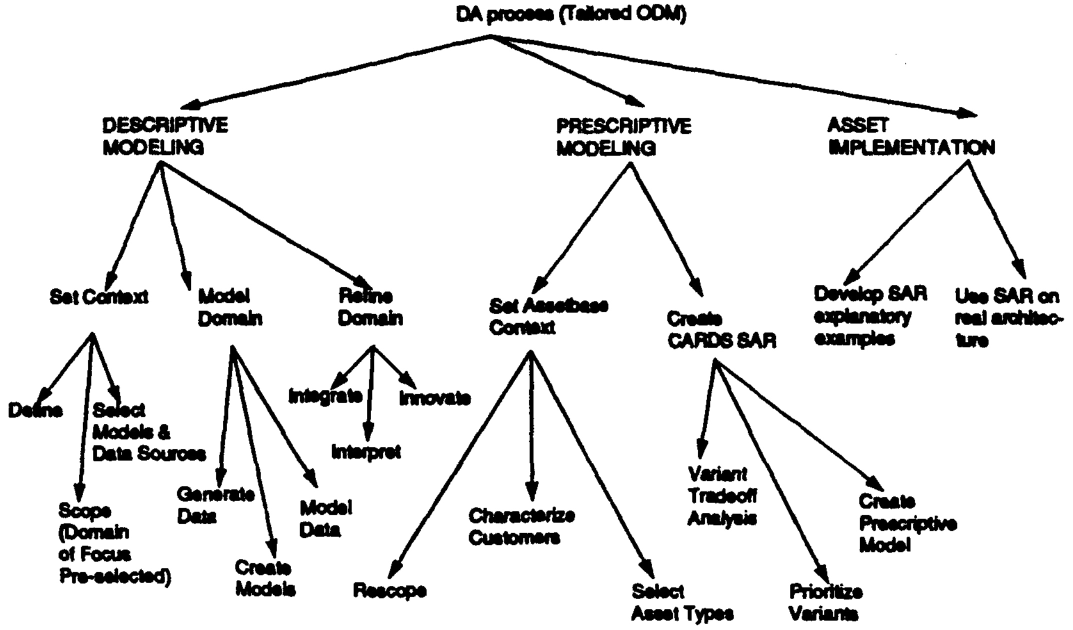
CARDS has: 1) Basic technology, model base with different views of the knowledge; tools (e.g. browser, composer, qualifier) that work off the base and 2) Process for certifying components for a domain (see later slides) and modeling the qualification information.

CARDS needs: 1) 'Good' Software Architecture Representation (SAR), and 2) Semantics for the integration of multiple architecture views.

Considerations:

- What abstractions are needed to support:
  - automated component qualification and
  - system composition?
- What information, technology is needed to support refinement and composition processes; system design and analysis processes; procurement, etc.?
- What technologies are available for architecture-centric reuse?
  - how do different technologies "fit?"
  - what are the invariants which allow representation and tooling diversity?
- What approach should CARDS adopt to
  - support systematic modeling without requiring an advanced degree in AI?
  - provide a conventional, non-AI interface to the CARDS model base?

Approach: ODM for Software Architecture Representations



DA process (Tailored ODM)

DESCRIPTIVE MODELING

PRESCRIPTIVE MODELING

ASSET IMPLEMENTATION

Set Context

Model Domain

Refine Domain

Set Assetbase Context

Create CARDS SAR

Develop SAR explanatory examples

Use SAR on real architecture

Define

Select Models & Data Sources

Scope (Domain of Focus Pre-selected)

Generate Data

Model Data

Create Models

Integrate

Innovate

Interpret

Rescope

Characterize Customers

Select Asset Types

Variant Tradeoff Analysis

Create Prescriptive Model

Prioritize Variants

Approach: ODM for Software Architecture Representations

Organization Domain Modeling (ODM) is a STARS Domain Engineering methodology. ODM is based on collaborative, team-based modeling *involving all the "stakeholders" of the domain.* ODM provides the ability to map points of commonality and difference without trying to work or resolve alternatives too rapidly. There is methodology support to model alternative "views" of the same information. ODM views the domain as the defined scope of reuse.

ODM has two distinct phases, *descriptive modeling in which commonalities and differences are modeled,* and a prescriptive phase where the modeling represents decisions and commitments to functionality to be supported and expresses the range of variability

Note: ODM presumes the definition of domain put forward by Arrango & Prieto Diaz that says: "A body of information is considered a problem domain if:

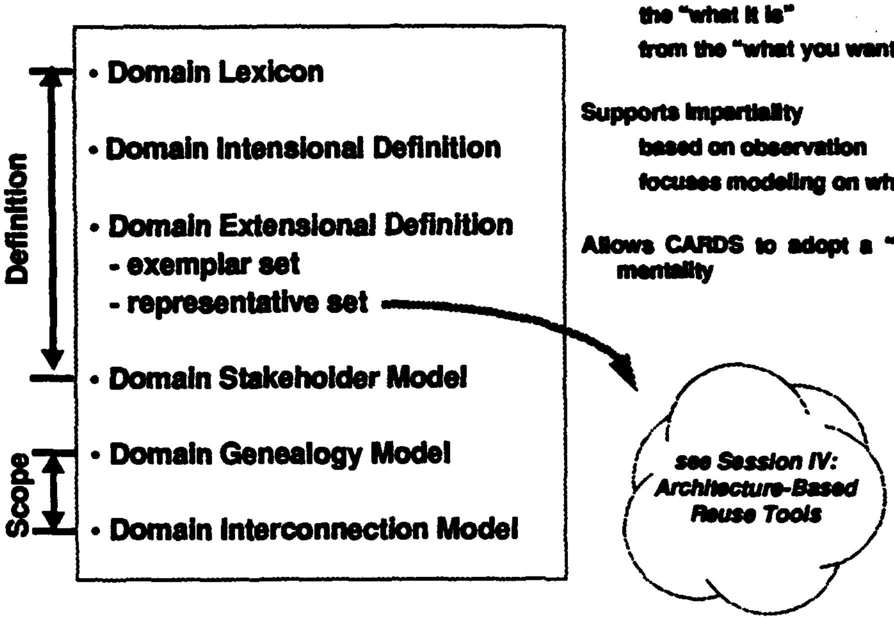
- Deep or comprehensive relationships are known or suspected with respect to some class of problems
- There is a community that has a stake (that is, stakeholders) in solving the problems
- The community seeks software intensive solutions to these problems; and
- The community has access to knowledge that can be applied to solving problems"
- And Software Architectures fits every one of their criteria.

References

Mark Simos, Organizational Domain Modeling, STARS Technical Report, Unisys Corporation.

Guillermo Arango, Prieto-Diaz, R., "Domain Analysis Concepts and Research Directions," in Domain Analysis and Systems Modeling, IEEE Computer Society Press, 1991. ISBN 0-8186-8996-X.

Why ODM? A Documentable Survey & Synthesis Method


Definition
Scope
• Domain Lexicon
• Domain Intensional Definition
• Domain Extensional Definition
- exemplar set
- representative set
• Domain Stakeholder Model
• Domain Genealogy Model
• Domain Interconnection Model
see Session IV: Architecture-Based Reuse Tools

Separates and Enforces
the "what it is"
from the "what you want it to be"

Supports Impartiality
based on observation
focuses modeling on what you can see

Allows CARDS to adopt a "*Don't* Invent Here" mentality

Why ODM? A Documentable Survey & Synthesis Method

The descriptive phase of ODM is intended to focus the analyst on describing what the system(s) is and discourages "creative" enhancements and personal bias (this is left to the prescriptive phase). Hence, observation of exemplars in the domain of analysis is focused on what exists and what can be seen. So rather than trying to observe and synthesize all at once, impartial observation allows an objective view of the domain exemplars. For CARDS, this approach is attractive insofar as we can collect as much information about software architecture representations and not have to try to re-invent, or invent on our own representation.

Why ODM? A Documentable Survey & Synthesis Method

Organization Domain Modeling (ODM), a STARS Domain Engineering methodology, was selected and tailored for determining the CARDS software architecture representation (SAR). The Domain of Focus (DOF) was pre-selected, an annotated bibliography compiled.

Domain Lexicon Sources include: IEEE Std 610.12; Garlan & Shaw; Perry & Wolf; Saunders, Horowitz, Mleziva; Wallnau, etc.

Intensional Domain Definition = Rules for inclusion and exclusion: A representation is included in the SAR domain if it is a design representation for at least some aspects of software architecture. A representation is not included in the SAR domain if it focuses primarily on requirements, detailed design, or algorithms.

Extensional Domain = Example SARS including the Exemplar (Core) set: KAPTUR; UNAS/SALE; Booch Object Oriented Design; Statemate, μ- Rapide; LILEANNA; OAD...; Borderline: Garlan & Shaw taxonomy of architecture styles; Counter set: requirements specification languages; PDL; programming languages. A *representative set* is a subset of the exemplar set which is analyzed in detail. It is the basis for the descriptive model of SAR features. The current representative set is: KAPTUR; UNAS/SALE; Garland & Shaw taxonomy of architectural styles; ROSE-2; μ- Rapide; LILEANNA; DCDS/RDD.

Domain Stakeholder Model provides context of how an SAR is related to roles of people in a software organization. The *reuse technology provider* develops/maintains the SAR, develops tools and provides technology transition. The *domain engineer* represents the domain specific software architecture (DSSA) n the SAR and qualifies/builds components based on the DSSA. The *application engineer* uses the DSSA and tools to build/maintain systems.

Domain Genealogy Model provides historical information on the development of the domain (useful in descriptive modeling).

Domain Interconnection Model shows the relations between domain of focus and related domains.

Descriptive Model is features of the members of the representative set modeled individually in RLF. These features are then synthesized into a prescriptive model. In this case, the prescriptive model = the software architecture representation (SAR).

Requirements for the Prescriptive Model (SAR): must facilitate architecture-centric reuse; must represent most DoD software architectures; must support interactive composition of systems; must assist component qualification; must be encodable in STARS Reuse Library Framework (RLF).

This page intentionally left blank.

ATF Summary and Status

- Domain Definition and Scope:
  - first step in formalizing the CARDS modeling approach
  - provides technical input to:
    - Architecture and Reuse Seminar
    - Reuse Adoption Handbooks
    - Evaluation of UNAS and other similar technologies
- Descriptive modeling in progress
- Plans
  - Complete ODM process on SAR domain
  - Develop explanatory examples
  - Use SAR on to describe real systems

ATF Summary and Status

To date, the CARDS Architecture Task Force has completed the Domain Definition and Scope for the software architecture representation domain. Early results of this work are being presented at this seminar and workshop and is also input to the CARDS Reuse Adoption Handbooks, to our tools evaluation, and to our domain engineering activities in the Command Center domain.

Descriptive modeling of the domain of software architecture is in progress. In addition to the coordination during this seminar, CARDS is in contact with numerous reuse organizations. The CARDS program welcomes the opportunity to collaborate with interested DoD, academic and industry partners.

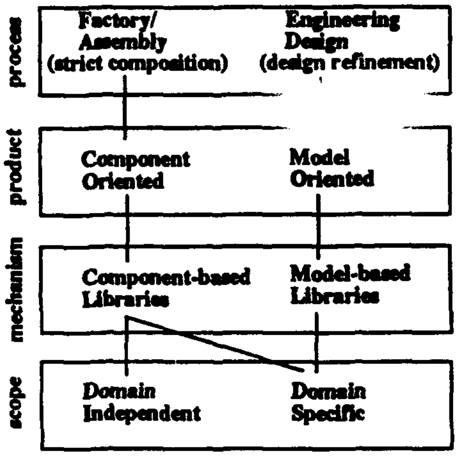
The ATF plans to complete the ODM process on SAR domain, develop explanatory examples and make the results part of the CARDS operational library.

Roadmap for this Session

CARDS Scientific	→	• Architecture Task Force • Organizational Domain Modeling: Domain of Software Architecture Representation
☞ CARDS Engineering	→	• Component Qualification • System Composition
CARDS Transition-to-Practice	→	• Handbooks • Franchising

This page intentionally left blank.

A Context for a Model-Based Approach to Reuse



process

Factory/ Assembly (strict composition)

Engineering Design (design refinement)

product

Component Oriented

Model Oriented

mechanism

Component-based Libraries

Model-based Libraries

scope

Domain Independent

Domain Specific

- Concept of "library assistance" varies by perspective

- Component-based libraries
  - organized for search and retrieval of individual reusable components
  - highly-developed, effective search mechanisms (relational)
  - weakness: parts-orientation loses context information

- Model-based libraries
  - organized for tailoring and adaptation of domain models
  - knowledge representation and conventional engineering notations
  - weakness: hard to build and get buy-in

- These approaches are complementary
  - need models and components
  - "certification" vs. "qualification"

A Context for a Model-Based Approach to Reuse

CARDS represents an alternative technology approach to reuse libraries, one which is more focused on describing the context of components (their interrelationships and their relationships to design, requirements—i.e., a domain model and an architecture). We have found it useful to distinguish two classes of reuse: a model-based approach, and a component-based approach. The model-based approach (CARDS) pursued by CARDS attempts to capture domain knowledge as formal models, and attempts to use this encoded knowledge to automate reuse through the use of knowledge-based assistants. It is also possible to consider model-based approaches based on formal methods.

Note that component-based libraries can be domain-specific or domain-independent, while model-based libraries tend to be domain-specific.

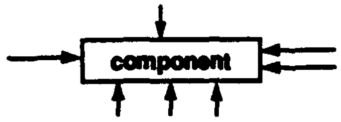
Also note that these approaches are complementary. Model-based libraries need components to work, while component-based libraries could develop large component populations in anticipation of encoding knowledge about their use in various contexts.

Examples of component-based libraries: STARS/ASSET, DISA/DSRS

Examples of model-based libraries: AT&T LaSSIE, CARDS, NASA/KAPTUR.

Model-Based Reuse: Certification and Qualification

Component-Based Perspective



Object Focus:
- what kind is it?
- what does it do?
- how good is it?

Faceted classification:
- powerful naming scheme
- equate "what it is" with "where to find it"

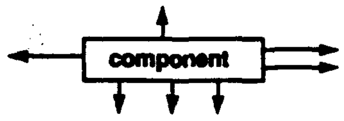
— FORM

Certification*:

- the process for determining whether a system or component is suitable for operational use.

*From: Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12 1990)

Model-Based Perspective



Context Focus
- what uses it?
- what does it use?
- when, why is it used?

Semantic classification:
- equate "how it is used" with "where to find it"

— CONTEXT

Qualification**:

- the process for determining the degree of "fit" between a component (form) and a particular design (context).

** CARDS Definition

Model-Based Reuse: Certification and Qualification

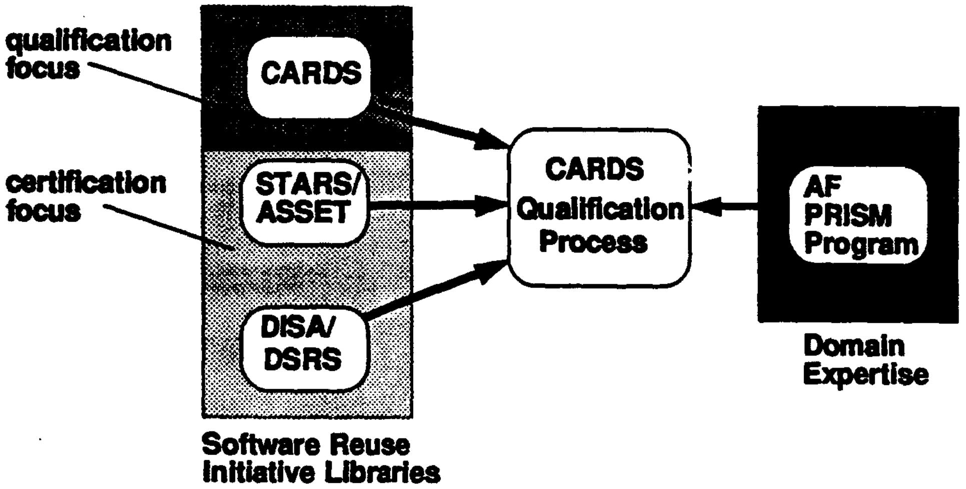
While Certification measures the goodness of a component in a rather generic fashion, Qualification of a component in a library provides assurance that this component is suitable for the domain.

The focus that component-based and model-based reuse libraries place on objects is fundamentally different. In a component-based approach, the emphasis on components in the library focus on "what" the component is and how "good" is it. This approach serves as a solid foundation in developing a rich and powerful classification scheme for equating "what the component is" to "where to find it" allowing the development of sophisticated mechanisms to search and retrieve components matching the search criteria.

In a model-based approach the focus is more on how the component fits in the application domain for which it is intended to be reused. In this approach the emphasis is on "what uses it" and "what does it use" which is an intent to preserve some or most of the context information lost in component-based approaches. Additionally, the model-based approach emphasizes on the "when" and "why" a component is used in the application domain where the intent is to tie the operational context or requirements for a components use. This approach serves as a foundation for semantic search classification schemes which relate how a component is used to "where to find it".

While CARDS Libraries adhere to a model-based paradigm in support of domain-specific reuse, CARDS believes that component- and model-based approaches are complementary, not diametrical.

Genealogy of CARDS Qualification Process



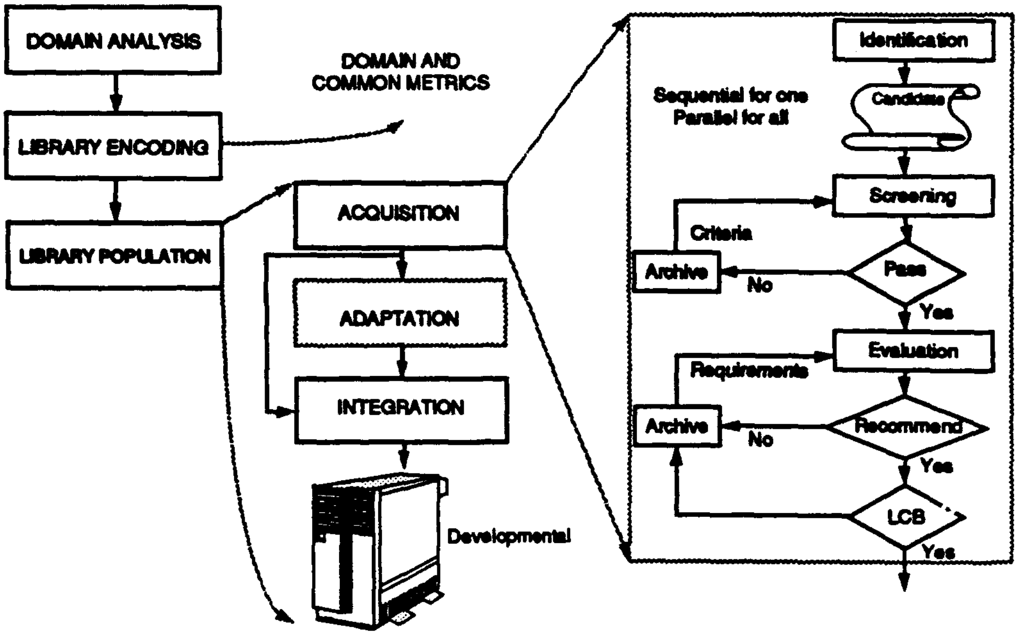
Genealogy of CARDS Qualification Process

The CARDS qualification process was synthesized from PRISM, ASSET and DSRS (RAPID) certification processes and refined to suit CARDS' domain specific library approach to reuse.

Reuse libraries, like DSRS and ASSET (which tend to focus more on components), are geared more towards a certification view of reusable assets. The CARDS library (which tend to focus more on the form and context of components) is geared more towards a qualification view of components. This does *not* imply that component-based libraries are purely certification oriented and conversely that model-based libraries are purely qualification oriented. The CARDS Qualification Process recognizes the complementary role that certification *and* qualification should play in the assessment of components for reuse libraries.

Clearly, a component-based library may not be so interested in the "domain", or context, that a component is intended to operate — for those qualification does not have a role. However, it would be ill conceived for a model-based library to totally ignore certification issues when qualifying a component for a domain.

Qualification Process



Qualification Process

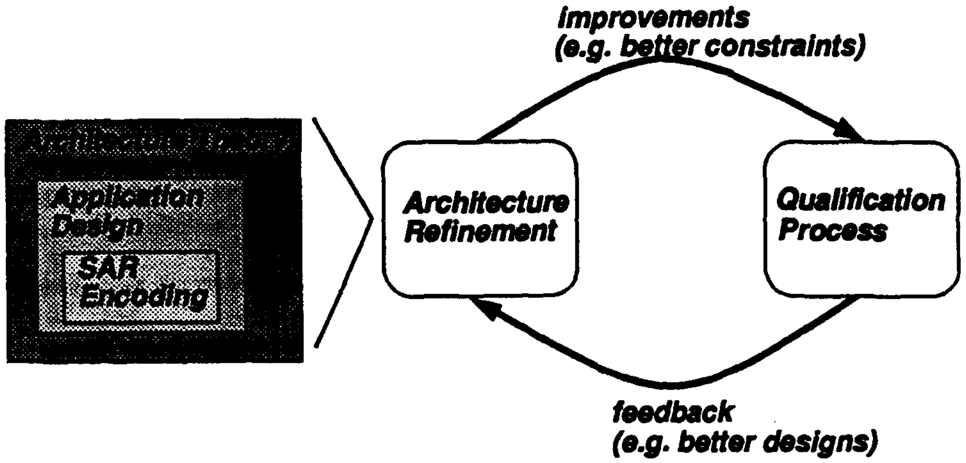
For **Qualification** the emphasis is on domain criteria and generic architecture. For **Certification**, the emphasis is on general characteristics such as reliability, maintainability, and portability.

The Qualification process was developed for the Command Center domain, but is applicable to other vertical domains with large grained COTS/GOTS/public domain components. The component classes represent horizontal domains. The qualification results are modeled in RLF and used in the qualification tool, system composition tool. Evaluation reports are also available in the CARDS library.

During the **Identification** phase a list of potential products suitable for the domain is compiled and information required for product screening is obtained. During **Screening** the list of potential products is prioritized so that more detailed evaluations can be performed with a high acceptance rate. Software Development Folders (SDF) are produced for products which pass screening. Products which do not pass screening are archived. The purpose of the **Evaluation** phase is to measure the selected Configuration Item against domain and common criteria and to produce the evaluation report.

Domain Criteria measure components against the domain and generic architecture (i.e. the "form, fit, and function") and are divided into component constraints, architectural constraints, and implementation con.-straints. Domain criteria are determined for each component class (horizontal domain), and selected domain criteria marked critical for vertical domain (command center). Criterion sources include: DISA Command Center Design Handbook, PRISM reports. **Common Criteria** measure domain independent evaluation of: reliability, maintainability, us[illegible]aturity, portability, and cost.

Qualification and Architecture



Qualification and Architecture

There is an interesting interdependency between architecture and qualification, which can be expressed as a positive feedback loop.

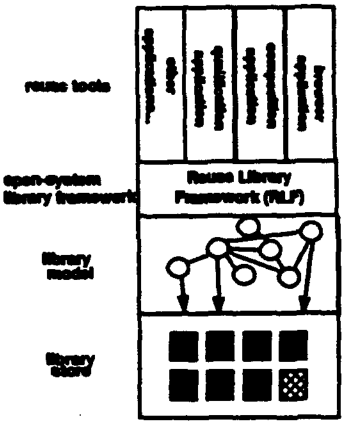
Architecture Refinement can be thought of as addressing three aspects of architecture: 1) the theory of software architecture (representation, evaluation, processes, etc.); 2) a specific application architecture/-design; and 3) the manner in which CARDS represents the application architecture in the CARDS library. As a result of undergoing qualification efforts, feedback can occur:

Architecture Theory: better understanding of the evaluation of non-functional characteristics of software architectures.

Application Design: a tuned design which expresses more trade-off information regarding the selection of components

SAR: a tuned representation which reflects advances in theory and capture of new and different kinds of trade-off information.

Model-Based Approach to Reuse



- Model contains information
  - about the domain
  - about the components
- Formal model of domain products supports
  - long-lived domains
  - reuse at different phases of systems-engineering life cycle
  - development of multiple reuse applications
  - 'components in, systems out'
- Different approaches to formal modeling
  - domain languages
  - module interconnection languages
  - expert system shells

Model-Based Approach to Reuse

The CARDS program has adopted a model-based approach to developing reuse technologies. More specifically, to be effective, our approach is to define our library model in the context of a domain, initially Command Centers. This slide depicts the architecture for a domain specific, model-based, reuse library.

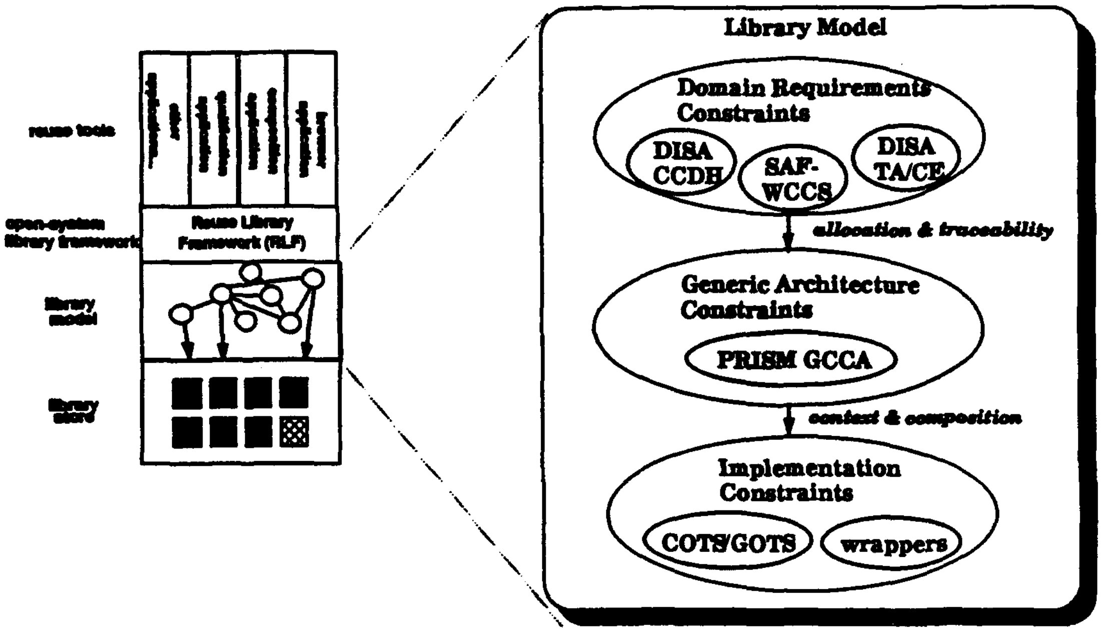
The library store "warehouses" the components stored in the library. The library model, or domain model, captures the products of domain analysis and defines the relationships between the components specific to a particular domain. Reuse tools, which leverage the information and relationships encoded in the library model to support a number of services available to CARDS library users.

The domain model gives us a formal encoding of the relationships between the requirements, architecture and implementation. The encoding can become the basis for a library framework in which to build applications to leverage those relationships and perform a variety of services. This is vitally important in post-deployment maintenance as those initially involved in building the domain model, defining the system architecture, and building the implementation are most likely not the individuals that will be making modifications to that system throughout its life cycle.

The focus of the domain model in a domain-specific reuse library is to bring together all the information that went into the architecture and implementation of the system. The library then becomes a vital tool in understanding the application domain being maintained.

Additionally, CARDS has employed the STARS product, Reusability Library Framework, (RLF), as the modeling paradigm to support the encoding of the domain model and the applications which access that model.

Model-Based Approach to Reuse


reuse tools
open-system library framework
Reuse Library Framework (RLF)
library model
library store
Library Model
Domain Requirements Constraints
DISA CCDH
SAF-WCCS
DISA TA/CE
allocation & traceability
Generic Architecture Constraints
PRISM GCCA
context & composition
Implementation Constraints
COTS/GOTS
wrappers

Command Center Domain Model

Model-Based Approach to Reuse

The CARDS approach to library modeling is to characterize the domain model in terms of three sub-models, each describing different kinds of constraints:

- *requirement* constraints - for instance, those imposed by the DISA[1] Command Center Design Handbook (CCDH);
- domain *architecture* constrains - those constraints imposed by a specific architecture implementation such as the PRISM[2] Generic Command Center Architecture (GCCA); and
- *implementation* constraints - those constraints imposed by a specific COTS[3] tool or software wrappers needed to integrate reuse components.

Also expressed are constraints which map between those sub-models:

- *allocation* constraints map between requirements and architecture - thus showing traceability how a specific part of the architecture satisfies portions of the requirements;
- *composition* constraints map between architecture and implementation - detailing how the architecture is satisfied by a particular implementation brought together from the library store.

The goal of domain-specific reuse library is to elicit reuse at higher levels of abstraction: requirements, architectures, systems and subsystems as well as components. This increases our ability to readily adopt reuse for a specific domain.

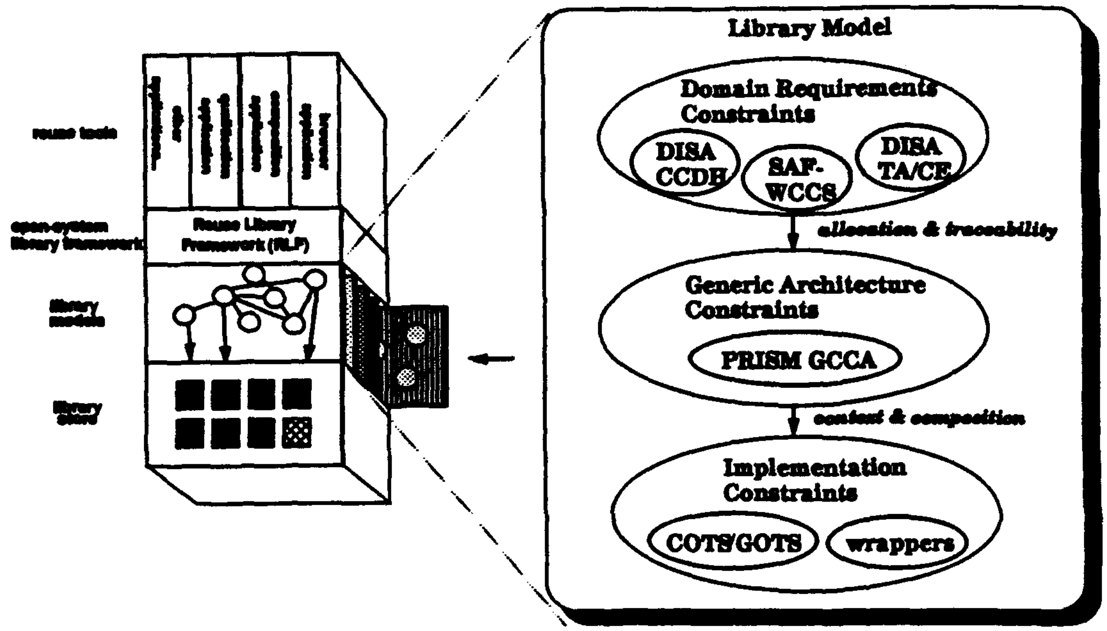
Significant benefits are achieved by focusing on model-based approach to capture the architecture and constraints to move the architecture along, avoiding the tendency of erosion and drift.

[1] Defense Information Systems Agency (DISA)

[2] Portable, Reusable, Integrated Software Modules (PRISM)

[3] Commercial off-the-shelf (COTS)

Model-Based Approach to Reuse


Library Model
Domain Requirements Constraints
DISA CCDB
SAF-WCCS
DISA TA/CE
allocation & traceability
Generic Architecture Constraints
PRISM GCCA
content & composition
Implementation Constraints
COTS/GOTS
wrappers
reuse tools
open-system library framework
Reuse Library Framework (RLF)
library models
library store

Command Center Domain Model

Model-Based Approach to Reuse

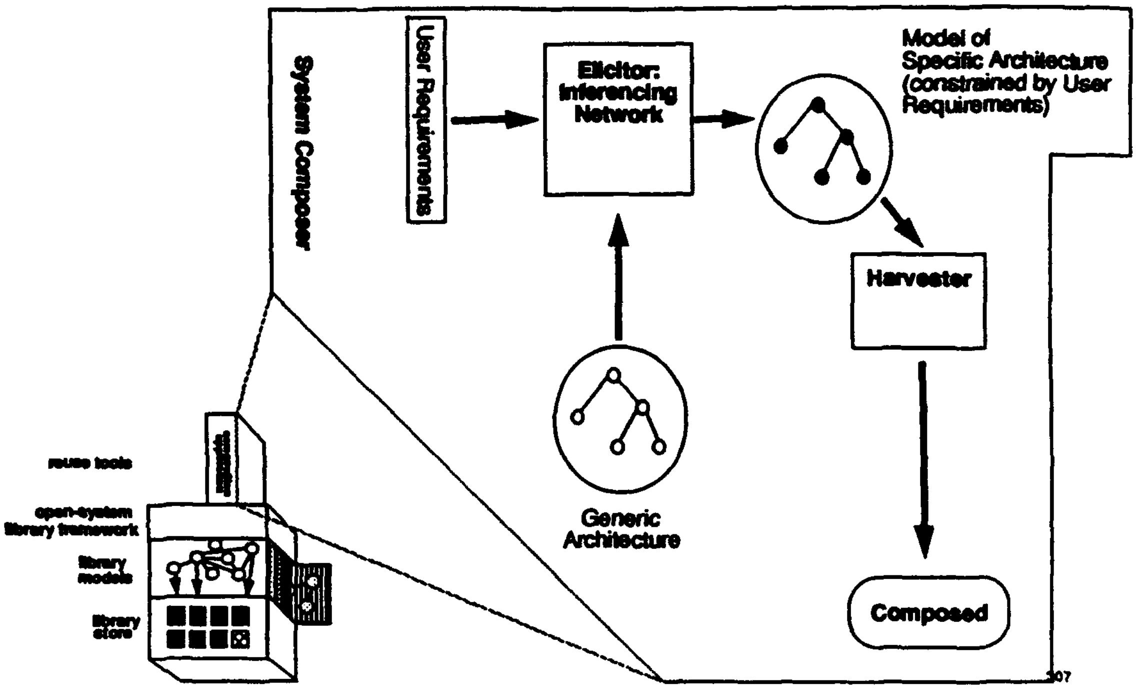
This slide provides a more realistic view of where CARDS technology is moving with respect to domain-specific reuse libraries. The Command Center domain is our initial domain, but other domains are planned. Therefore our reuse tools are designed to operate on any domain model.

The first prototype for the CARDS CC Library reuse tool, the system composition application, supports rapid integration. The system composition application works by eliciting input from the library user in order to identify the constraints that an operational command center must satisfy. The use of deductive inferencing in conjunction with the constraint network allows the system composition application to query the user for the minimal amount of information necessary to support automatic composition of a prototype system. Although the system composition application is targeted to the development phase, its computational model will apply to post-deployment support.

Other reuse tools envisioned, that to apply themselves to post-deployment maintenance, are a change impact analysis and component qualification application. The change impact analysis application built on the allocation constraint network would assess changes in requirements on the architecture. Further, on the composition constraint network, it would assess the impact of changes in the architecture on the implementation. The component qualification tool would also leverage the composition constraint network to suggest alternate components in an implementation during adaptive maintenance.

Also important to note about this slide is that there is not one library store for Command Centers and another library store for the next domain. Rather, each library model references those components in the store which are qualified and applicable for its' domain. It would be very likely that domains which both share the concept of a database management system would both "point" to the same component in the library store which satisfy the constraints placed on DBMSs.

System Composition


System Composer
User Requirements
Elicitor: Inferencing Network
Model of Specific Architecture (constrained by User Requirements)
Harvester
Generic Architecture
Composed
reuse tools
open-system library framework
library models
library store

System Composition

The objective of system composition is to provide command center library users with tools to automate the composition of new command centers, or portions thereof, based on user requirements from components in the library model. The approach is to apply user input to the library model to produce prototype demonstrations of systems, assist users in the decision making process of building new systems, and when possible, provide users with the actual software to build them.

This slide provides a top level view of the system composition application. There are three inputs to the "System Composer": a model of the Command Center Library, target system constraints elicited from the user, and a rule-base for system composition and heuristics for building the system. The outputs of the system composition tool are system demonstrations and composed systems (or portions of a system).

The System Composition Tool prototype has provided a reference for what we expect out of a software architecture representation, that is what are the products a software architecture should be helping to produce.

Roadmap for this Session

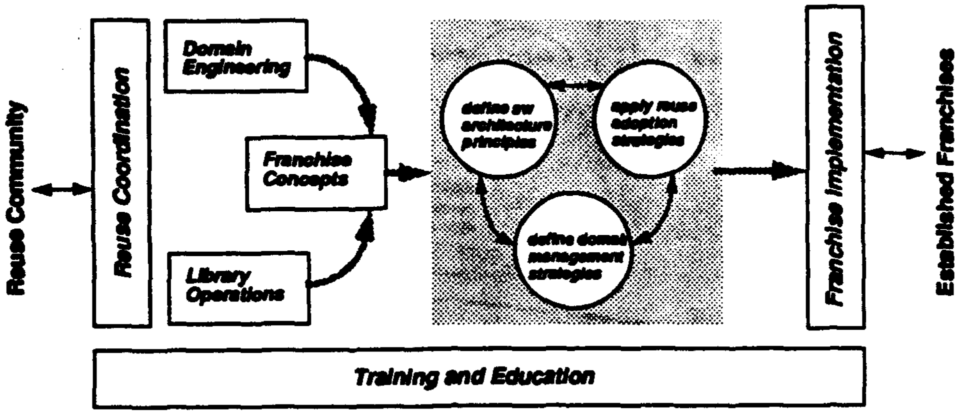
CARDS Scientific →
- Architecture Task Force
- Organizational Domain Modeling: Domain of Software Architecture Representation

CARDS Engineering →
- Component Qualification
- System Composition

☞ CARDS Transition-to-Practice →
- Handbooks
- Franchising

This page intentionally left blank.

CARDS Tech Transfer in Practice

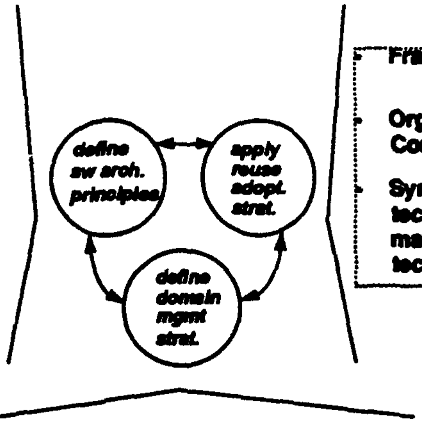


CARDS Tech Transfer in Practice

The rectangles in this slide represent the CARDS Phase 3 project areas and how they interact. Notice that Training and Education span our entire project. CARDS contact with the Reuse Community affects the other technical projects: Domain Engineering, Library Development and Franchise Concepts. The processes and products produced by these groups is then transferred to franchise organizations wishing to establish reuse capabilities.

CARDS Tech Transfer Approach

- Descriptive modeling of KAPTUR, ROSE-2, UNAS; others targeted
- Architecture seminar
- "Alternative views"


define sw arch. principles
apply reuse adopt. strat.
define domain mgmt strat.

- Franchising Strategy
- Organizational Analysis Concepts of Operation
- Synthesis/application of technology "change management" techniques

- Engineer's Handbook
- Component/Tool Developer's Handbook
- Acquisition Handbook
- Direction-Level Handbook
- Technical Concepts Document
- Legal and Acquisition Issues Document
- and many more...

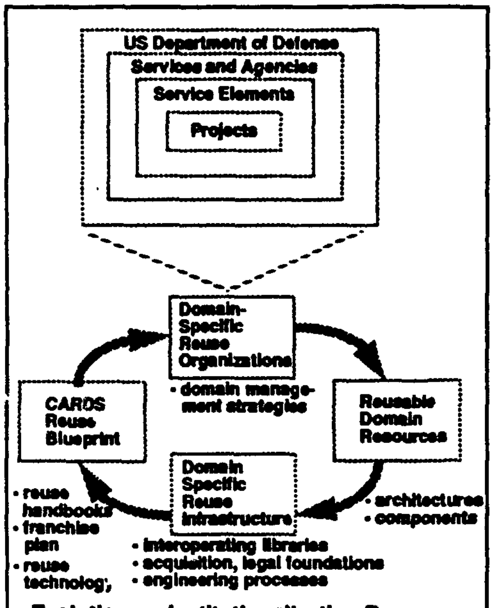
CARDS Tech Transfer Approach

CARDS has two main avenues for transfer of technical information, the Handbooks and Franchising activities.

The handbooks include: Engineering Handbook, Library Operations Policies and Procedures, Technical Concepts Document, Acquisition Handbook, Direction Level Handbook, Component and Tool Developer's Handbook. CARDS is also developing Model Contracts/Agreements and is conducting Market Studies.

CARDS reuse support services are available to Government organizations. These services include implementation of the CARDS blueprint according to the Handbooks and CARDS Franchise Plan.

The Franchise Approach to DoD-wide Reuse



Evolutionary Institutionalization Process

- **CARDS seeks to create permanently established reuse capabilities within DoD organizations (i.e., "franchises")**
- **Franchise Plan is CARDS tool to apply the reuse blueprint to DoD organizations**
- **Once established, franchises may create, manage and support use of domain-specific assets**
- **Franchise-developed reuse capabilities become part of a larger DoD reuse infrastructure**
- **Goal is to support, and integrate, reuse capabilities across multiple organizational levels, and across government and contractor boundaries**

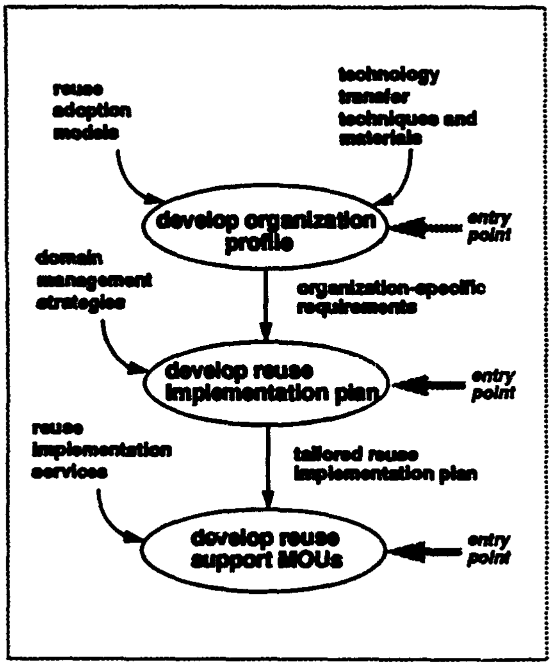
The Franchise Approach to DoD-wide Reuse

The CARDS approach to technology transfer includes a heavy emphasis on direct involvement between CARDS and technology adopting organizations. We refer to such organizations as "franchises."

We view Franchising as a process-feedback approach to incremental adoption of reuse in the DoD. The approach recognizes that reuse capabilities (domain expertise, product lines, etc.) exist within DoD product and logistics centers, and therefore reuse adoption must take place within these organizations. Thus, CARDS provides services to support an organization in adopting reuse, but, ultimately, the reuse products and experience generated from the use of reuse technology and methods must be generated by DoD organizations. CARDS views Franchising as a means to initiate the transition activity, and the channel the resulting products and lessons-learned into future franchising activities.

Note that CARDS joint-development activities are not restricted to only one kind of organization. We currently have development activities underway at the National Security Agency and Air Force Sacramento Air Logistics Center, as well as with the Air Force PRISM program (who serve as the domain experts and prototype developers for the major elements of the CARDS library).

Recognizing Franchise-Unique Context



reuse adoption models

technology transfer techniques and materials

develop organization profile

entry point

domain management strategies

organization-specific requirements

develop reuse implementation plan

entry point

reuse implementation services

tailored reuse implementation plan

develop reuse support MOUs

entry point

- Enormous variation among DoD software organizations
  - different levels of interest in, and commitment to, reuse
  - all of the usual socio/political variations
- Technology transition must recognize organizational diversity
  - different business objectives
  - different starting points (capabilities)
- Technology transition must also recognize technology diversity
  - domain-induced technology requirements
  - technology diversity among DoD contractors
  - continuous advances and evolution in software design and development technology

Recognizing Franchise-Unique Context

To be successful at technology transition, CARDS believes it is inevitable that organization-specific needs be addressed. Specifically, no two organizations are in a current state of "reuse maturity" (however one wishes to define this concept), nor do any two organizations share the same culture, business climate or strategic objectives. In short, the context into which a technology is being transferred shapes the approach taken to undertake the transfer.

Our approach to franchising is based upon a flexible, 3-tiered analysis, consulting and joint-development project model. CARDS has developed materials to conduct organizational analysis based upon organizational development principles, software process maturity principles, reuse principles and technology transfer principles. These materials are intended to be used to aid in identifying organization-specific requirements for the development of a reuse implementation plan. (Note: *these materials have only recently been developed, and have not yet been applied*).

It is possible, of course, to assist an organization in developing a reuse implementation plan—and we have provided services to the Air Force to do so in one instance—without having created an organizational profile. While CARDS believes that this may make the reuse implementation plan less effective (or at least increase the likelihood that the plan will be less than optimal), it is sometimes necessary to accept such planning limitations in the name of making even small progress in initiating organization and business practice changes to support reuse.

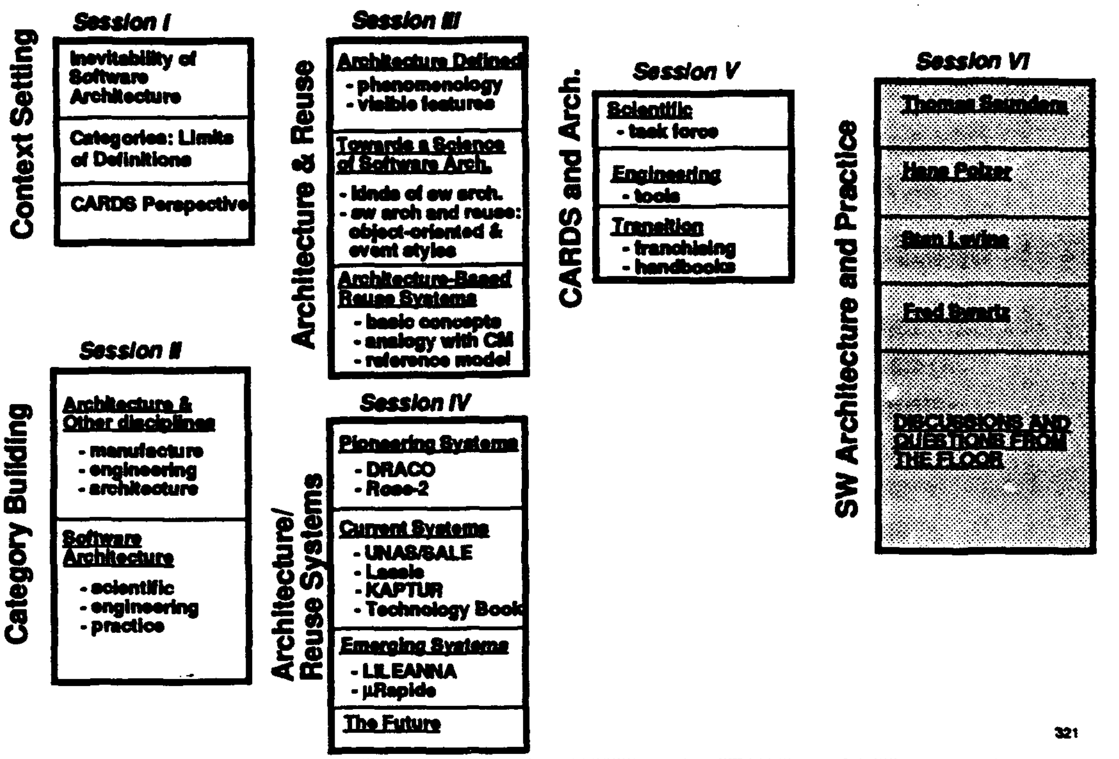
Finally, there is also scope for inserting reuse techniques into an organization at the "grass roots" level through direct prototype development efforts with organizations.

CARDS Team Members

- CARDS is managed by ESC/AVS
  - Mr. Robert Lencewicz, Program Manager (617) 377-9369
- Unisys Corporation is the prime contractor
- Subcontractors represent a highly diverse and skilled team
  - DSD Laboratories
  - Electronic Warfare Associates
    - Azimuth
    - DN American
    - Galaxy Global Corporation
    - Strictly Business Computer Systems, Inc.
  - HGO Technology, Inc.
  - AETech Inc.

This page intentionally left blank.

CARDS



Context Setting

Session I

Inevitability of Software Architecture

Categories: Limits of Definitions

CARDS Perspective

Category Building

Session II

Architecture & Other disciplines
- manufacture
- engineering
- architecture

Software Architecture
- scientific
- engineering
- practice

Architecture & Reuse

Session III

Architecture Defined
- phenomenology
- visible features

Towards a Science of Software Arch.
- kinds of sw arch.
- sw arch and reuse: object-oriented & event styles

Architecture-Based Reuse Systems
- basic concepts
- analogy with CM
- reference model

Architecture/ Reuse Systems

Session IV

Pioneering Systems
- DRACO
- Rose-2

Current Systems
- UNAS/SALE
- Lassie
- KAPTUR
- Technology Book

Emerging Systems
- LILEANNA
- µRapide

The Future

CARDS and Arch.

Session V

Scientific
- task force

Engineering
- tools

Transition
- franchising
- handbooks

SW Architecture and Practice

Session VI

Thomas Saunders

Hans Polzer

Stan Levine

Fred Swartz

DISCUSSIONS AND QUESTIONS FROM THE FLOOR

Central Archive for Reusable Defense Software (CARDS)

Software Architecture Seminar

16 November 1993

Panel Discussion

Panel Discussion Architectures in Practice

Mr. T. F. "Skip" Saunders, Mitre Corporation

Mr. Hans Polzer, Unisys Corporation

Mr. Stan Levine, US Army Communications Electronics Command (CECOM)

Capt Frederick Swartz, Training System Program Office, ASC/YTE

Views on Architecture and Reuse

T. F. Saunders

16-27 Nov 93

MITRE

Outline

- **Goals:**
  Interoperability, Changeability, Cost Effectiveness
- **Views on Architecture**
- **Program Management Perspectives for Reuse**
- ***Acquisition management of Architectures*** **-**
  - **A strategy to promote Reuse**
  - **A strategy dependent on "Popular" Standards**
- **Interoperability**

MITRE

Emerging interest in "Architecture"

- Driven by desire for:
  - more changeability - "vertical" flexibility
  - more interoperability - "horizontal" flexibility
  - cheaper development - commercial product exploitation
- Technical solution is (and has been for a long time) envisioned (but not proven) to be associated with technology that is well ordered (i.e. well structured, modular,etc.)
  - Vertical flexibility comes from framework based system structure
    - "open" standards for components within the system
    - mix of proprietary and non-proprietary products
  - Horizontal flexibility comes from standard protocols
    - "open" protocols for exchanging bits
    - data element standardization for interpreting the bits exchanged or translators
  - Commercial trends are providing technology to support both vertical and horizontal flexibility

MITRE

"Open" Concepts -
An important distinction in definitions

- Open Systems:
  - A system is "open" if it has publicly known interfaces such that its components may be treated as "black boxes"
  - A system is a "desirable open" system if the interfaces are supported and used by a wide variety of vendors

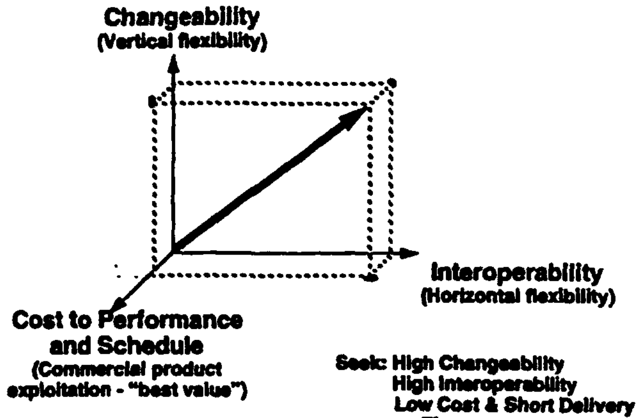
Note: Publicly known ≠ publicly owned,

i.e. an open system may have proprietary components

- "Proprietary" allows financial reward for:
  - achieving large market
  - improving products
  - maintaining backward (or forward) compatibility

MITRE

Three Objectives for "Information Architecture"



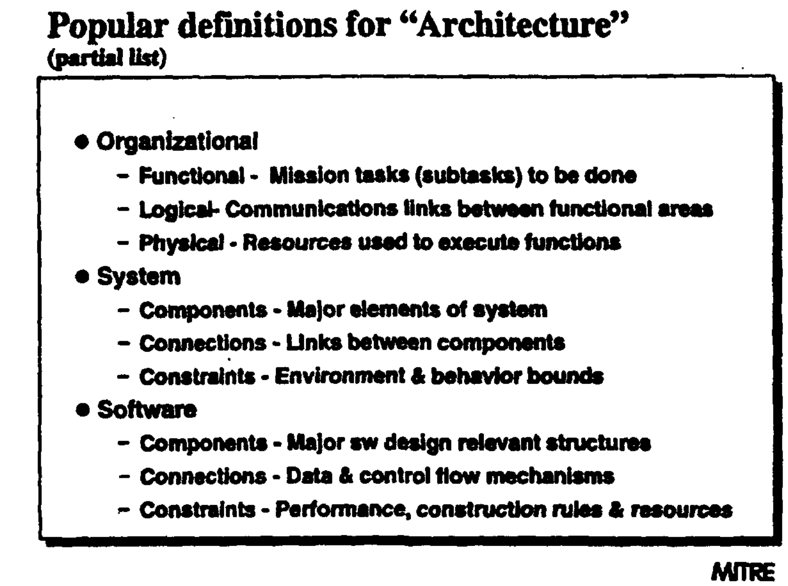
Seek: High Changeability
High Interoperability
Low Cost & Short Delivery
Times

MITRE

Outline

- Goals:
  Interoperability, Changeability, Cost Effectiveness
- Views on Architecture
- Program Management Perspectives for Reuse
- Acquisition management of Architectures -
  - A strategy to promote Reuse
  - A strategy dependent on "Popular" Standards
- Interoperability

MITRE

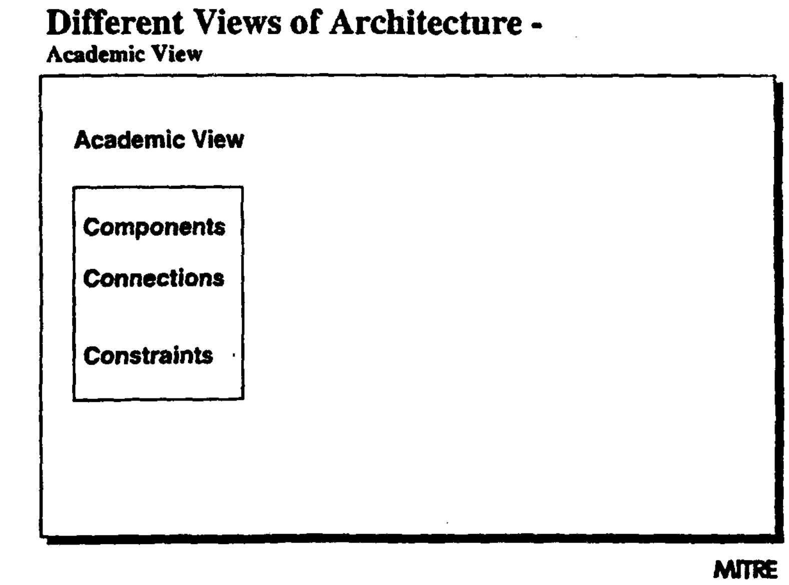


Popular definitions for "Architecture"

(partial list)

- Organizational
  - Functional - Mission tasks (subtasks) to be done
  - Logical- Communications links between functional areas
  - Physical - Resources used to execute functions
- System
  - Components - Major elements of system
  - Connections - Links between components
  - Constraints - Environment & behavior bounds
- Software
  - Components - Major sw design relevant structures
  - Connections - Data & control flow mechanisms
  - Constraints - Performance, construction rules & resources

MITRE



Different Views of Architecture -

Academic View

Academic View

Components

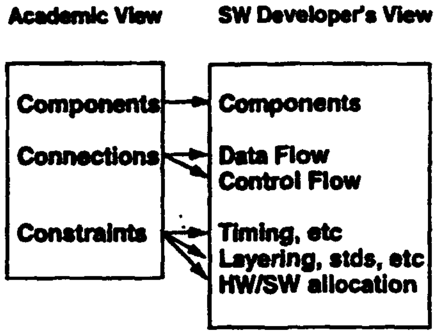
Connections

Constraints

MITRE

Different Views of Architecture -

Software Developer's View



Academic View

- Components
- Connections
- Constraints

SW Developer's View

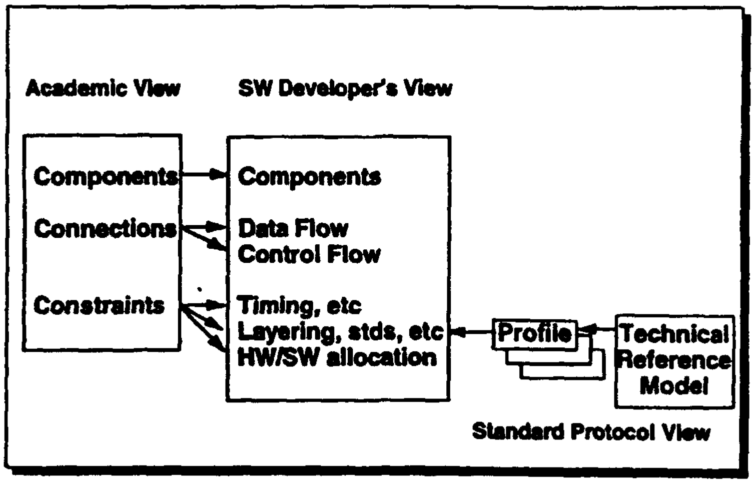
- Components
- Data Flow
- Control Flow
- Timing, etc
- Layering, stds, etc
- HW/SW allocation

Different Views of Architecture -

Software Developer's View (Notes continued)

Different Views of Architecture -

Standard Protocol Community View



Academic View

SW Developer's View

Components

Connections

Constraints

Components

Data Flow
Control Flow

Timing, etc
Layering, stds, etc
HW/SW allocation

Profile

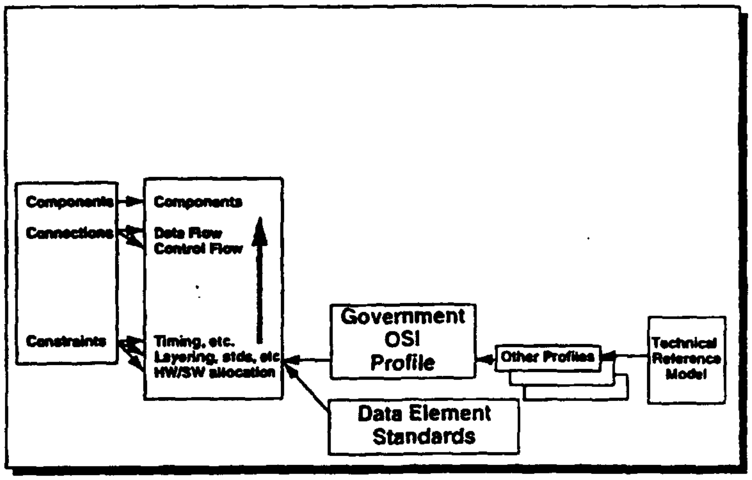
Technical Reference Model

Standard Protocol View

MITRE

Different Views of Architecture -

Government Standards Community View



Components

Connections

Constraints

Components

Data Flow
Control Flow

Timing, etc.
Layering, stds, etc
HW/SW allocation

Government OSI Profile

Other Profiles

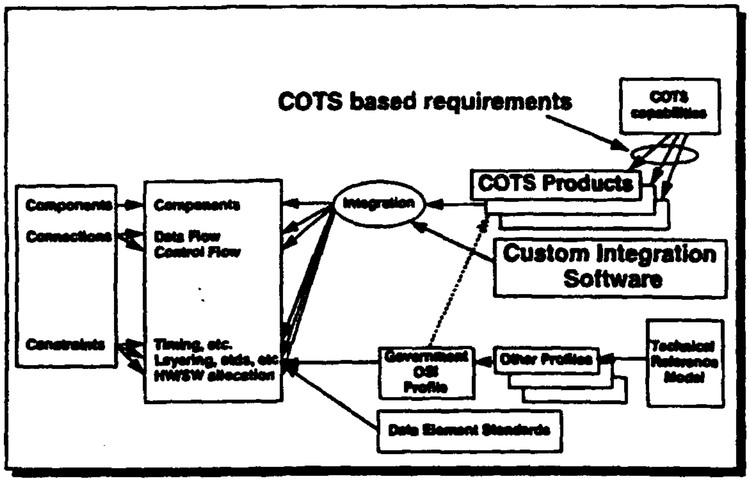
Technical Reference Model

Data Element Standards

MITRE

Different Views of Architecture -

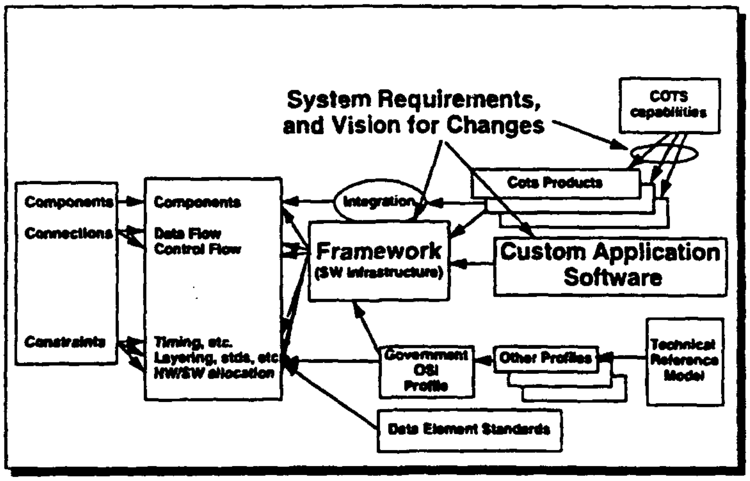
Rapid Prototyping Community View



MITRE

Different Views of Architecture -

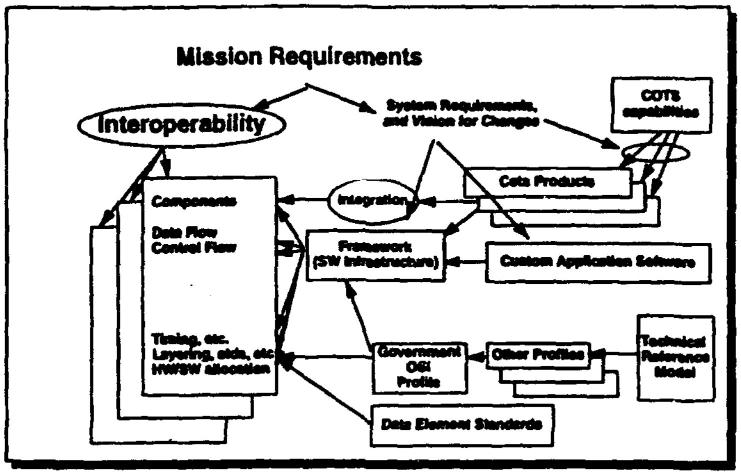
Architecture Preservation Community View



MITRE

Different Views of Architecture -

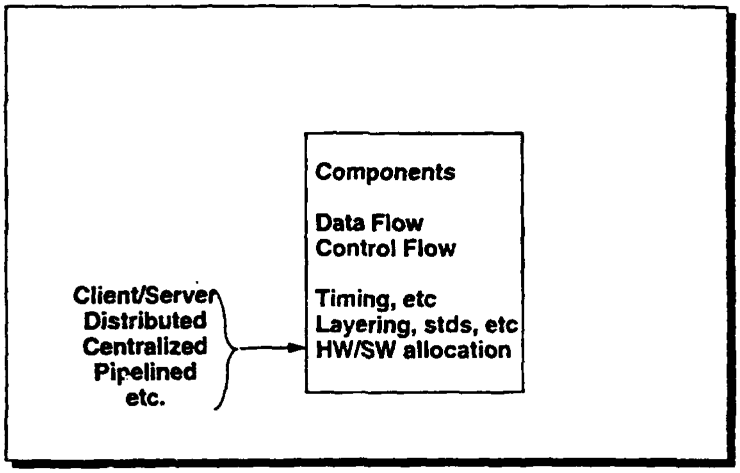
Mission Organization's View



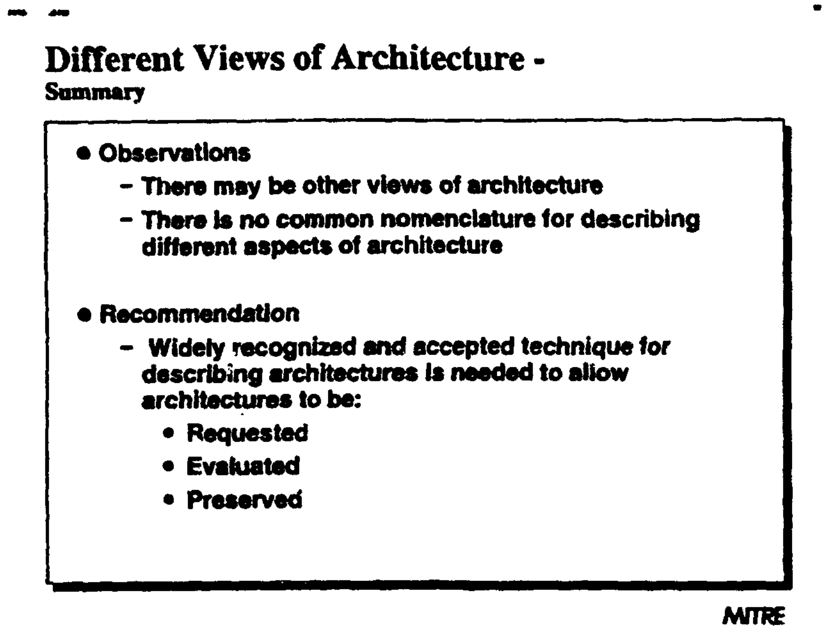
MITRE

Different Views of Architecture -

Hardware View



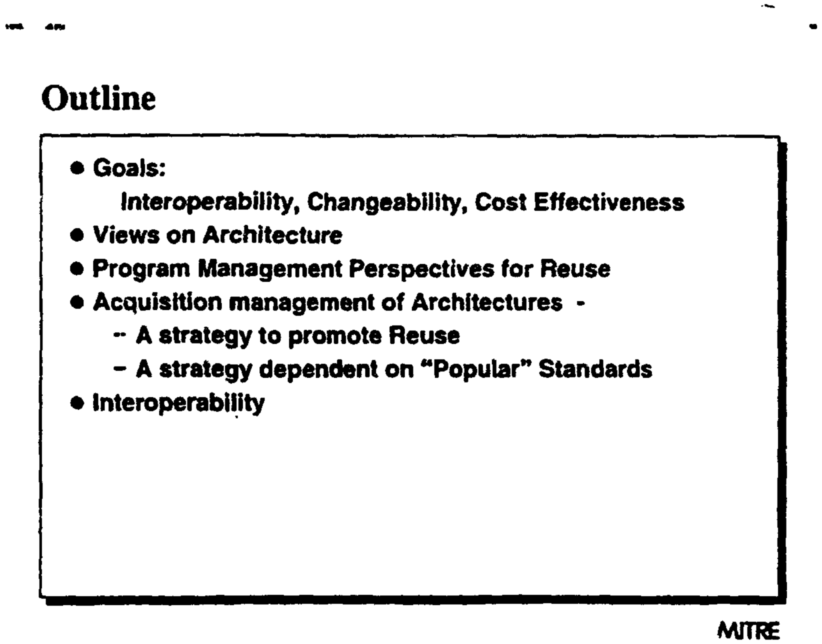
MITRE



Different Views of Architecture - Summary

- Observations
  - There may be other views of architecture
  - There is no common nomenclature for describing different aspects of architecture
- Recommendation
  - Widely recognized and accepted technique for describing architectures is needed to allow architectures to be:
    - Requested
    - Evaluated
    - Preserved

MITRE



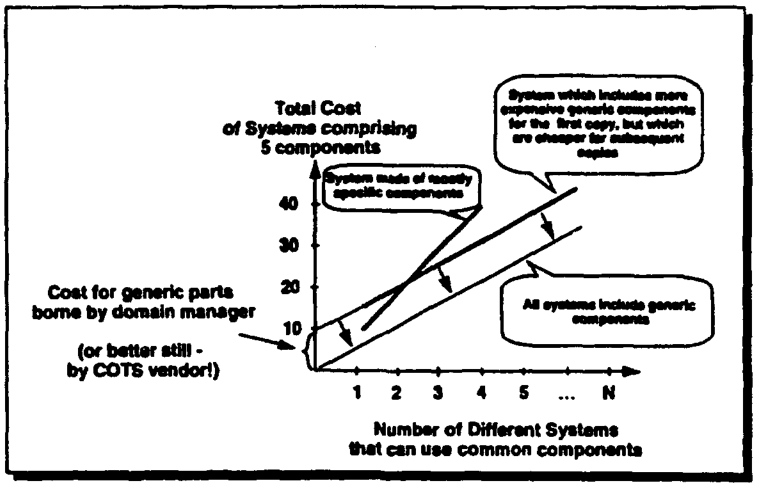
Outline

- Goals:
  Interoperability, Changeability, Cost Effectiveness
- Views on Architecture
- Program Management Perspectives for Reuse
- Acquisition management of Architectures -
  - A strategy to promote Reuse
  - A strategy dependent on "Popular" Standards
- Interoperability

MITRE

A Domain Managers Motivations -

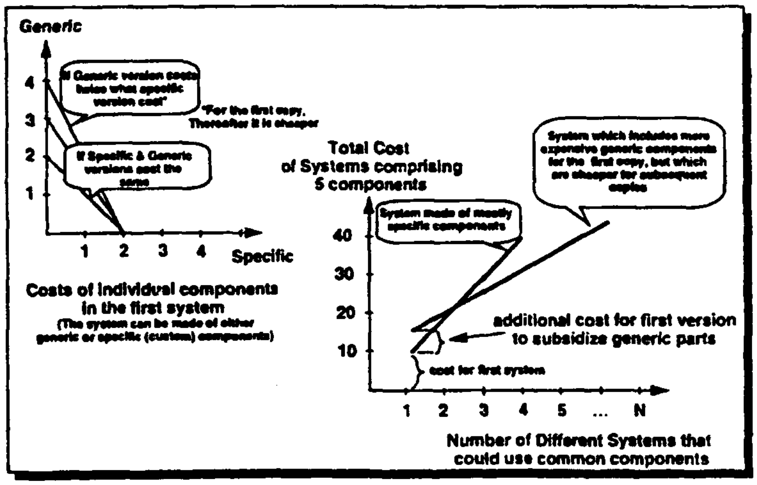
Reusable products to fulfill "corporate" perspective



MITRE

A Program Managers Motivations -

The missing "corporate" perspective



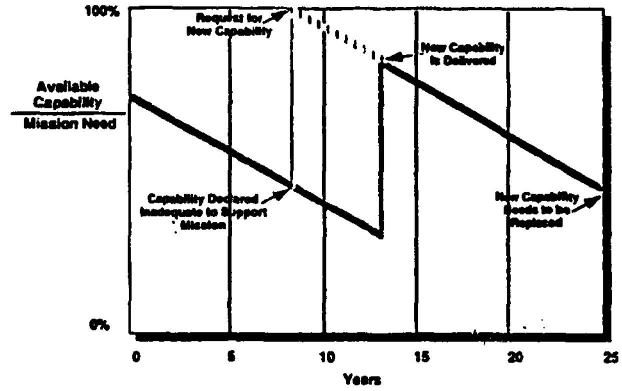
MITRE

Outline

- Goals:
  Interoperability, Changeability, Cost Effectiveness
- Views on Architecture
- Program Management Perspectives for Reuse
- Acquisition management of Architectures -
  - A strategy to promote Reuse
  - A strategy dependent on "Popular" Standards
- Interoperability

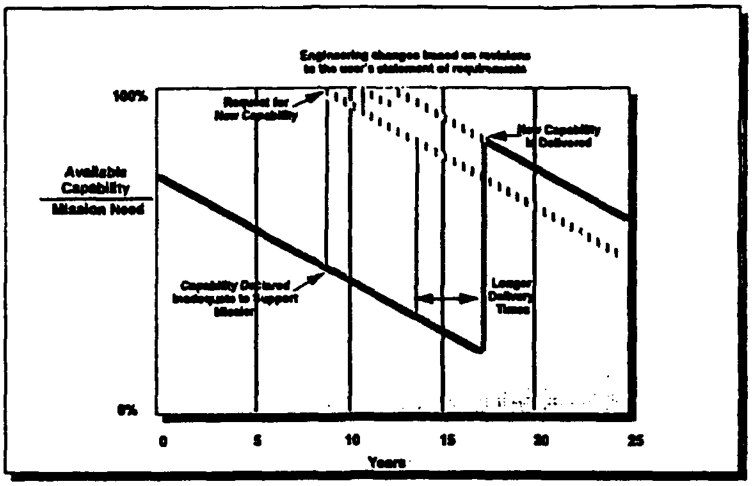
MITRE

Current Acquisition Approach



MITRE

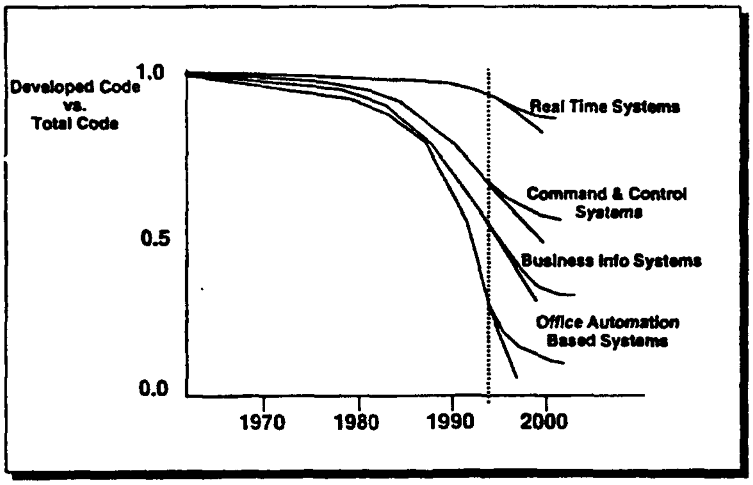
Current Acquisition Experience



MITRE

Trends in Software Development

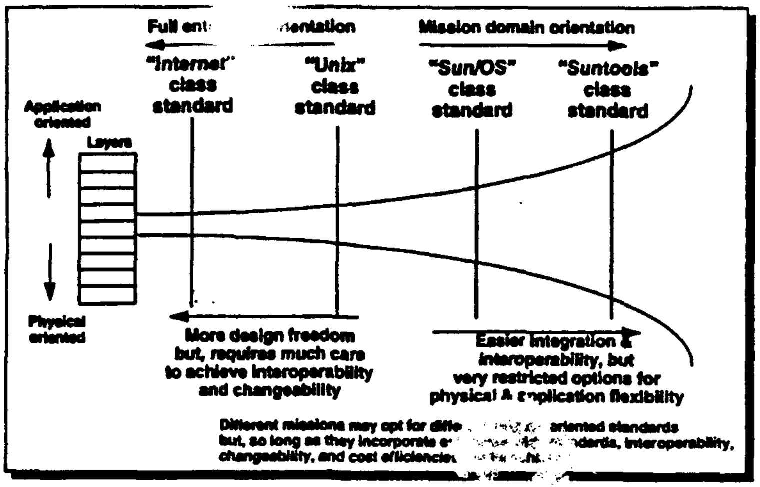
The shift towards integrated rather than developed products



MITRE

Information System Architecture -

A Spectrum of "Buil[illegible] [illegible]odes"



Full ent[illegible] [illegible]ientation

Mission domain orientation

"Internet" class standard

"Unix" class standard

"Sun/OS" class standard

"Suntools" class standard

Application oriented

Layers

Physical oriented

More design freedom but, requires much care to achieve interoperability and changeability

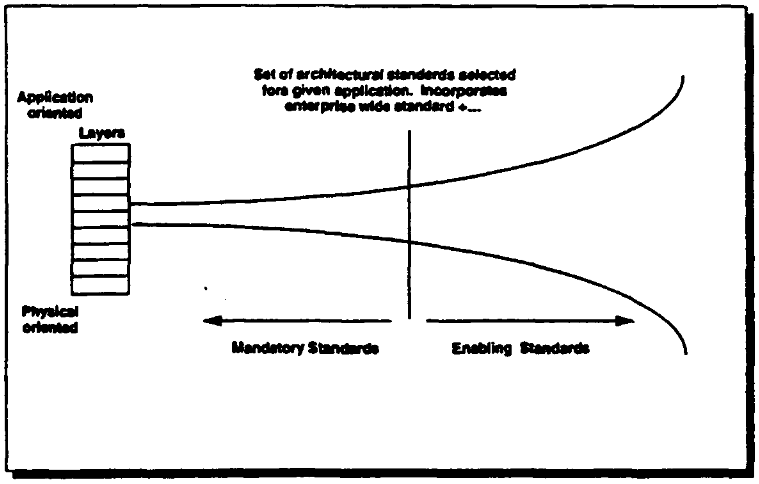
Easier integration & interoperability, but very restricted options for physical & [illegible]pplication flexibility

Different missions may opt for diffe[illegible] [illegible] oriented standards but, so long as they incorporate e[illegible] [illegible] [illegible]ndards, interoperability, changeability, and cost efficiencie[illegible] [illegible]

MITRE

Domain Specific Choices -

Mandatory vs. Enabling Standards



Set of architectural standards selected fora given application. Incorporates enterprise wide standard +...

Application oriented

Layers

Physical oriented

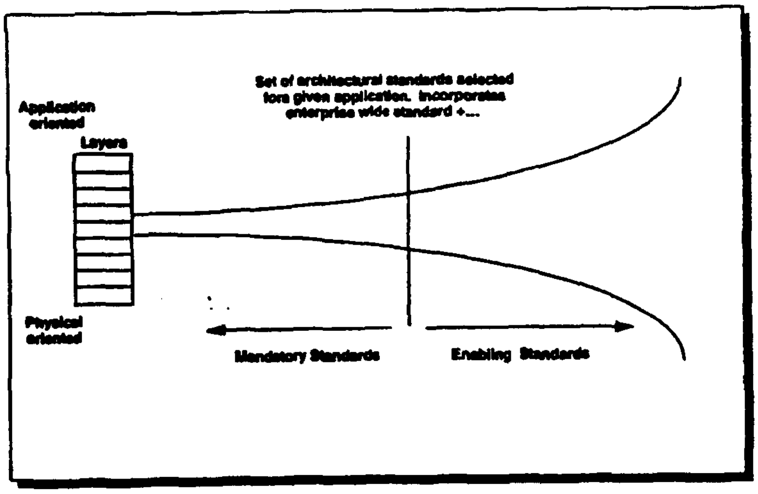
Mandatory Standards

Enabling Standards

MITRE

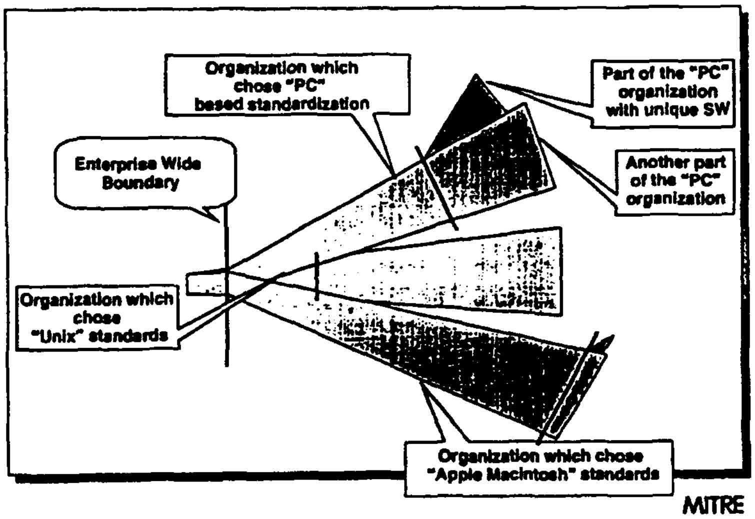
Domain Specific Choices -

Mandatory vs. Enabling Standards


Set of architectural standards selected
fora given application. Incorporates
enterprise wide standard +...
Application
oriented
Layers
Physical
oriented
Mandatory Standards
Enabling Standards

MITRE

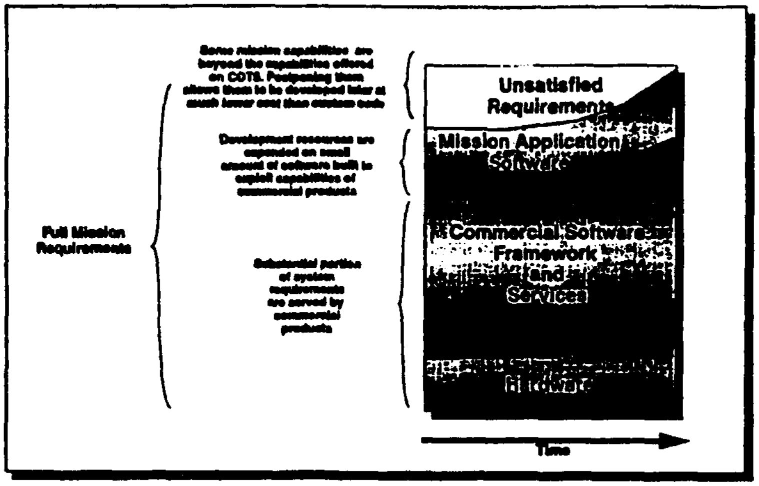
Family of options



MITRE

Leveraging Commercial Products -

The Promise of "Open" & "Structured" Architected Systems



Some mission capabilities are beyond the capabilities offered on COTS. Postponing them allows them to be developed later at much lower cost than custom code

Development resources are expended on small amount of software built to exploit capabilities of commercial products

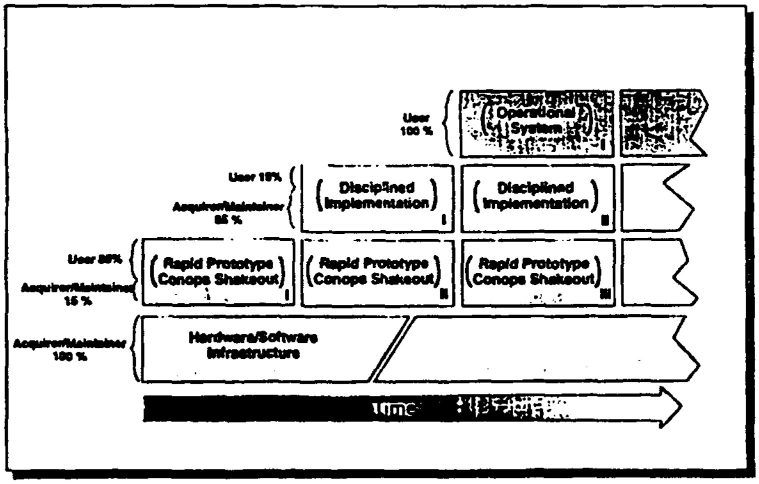
Full Mission Requirements

Substantial portion of system requirements are served by commercial products

Unsatisfied Requirements
Mission Application Software
Commercial Software Framework and Services
Hardware
Time

MITRE

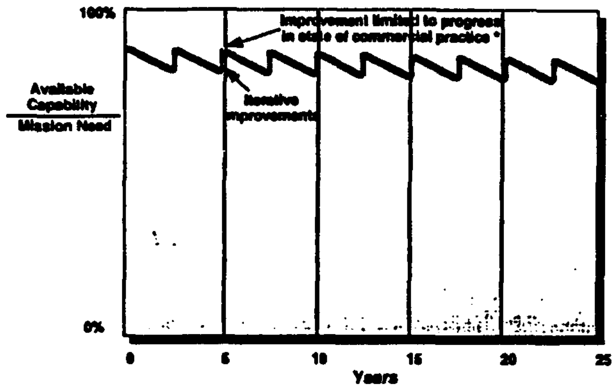
Progressive Acquisition



User 100 %
Operational System
User 15%
Acquirer/Maintainer 85 %
Disciplined Implementation
I
Disciplined Implementation
II
User 85%
Acquirer/Maintainer 15 %
Rapid Prototype Conops Shakeout
I
Rapid Prototype Conops Shakeout
II
Rapid Prototype Conops Shakeout
III
Acquirer/Maintainer 100 %
Hardware/Software Infrastructure
Time

MITRE

Future Acquisition Approach



* Not driven to meet full evolved mission needs

MITRE

Outline

- Goals:
  Interoperability, Changeability, Cost Effectiveness
- Views on Architecture
- Program Management Perspectives for Reuse
- Acquisition management of Architectures -
  - A strategy to promote Reuse
  - A strategy dependent on "Popular" Standards
- Interoperability

MITRE

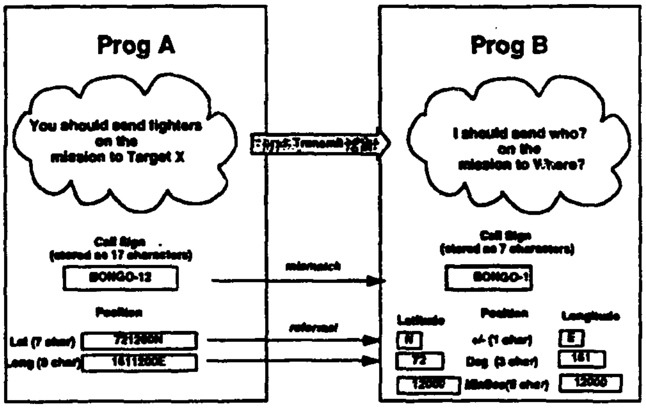
Interoperability =

Interconnectivity + Data Compatibility

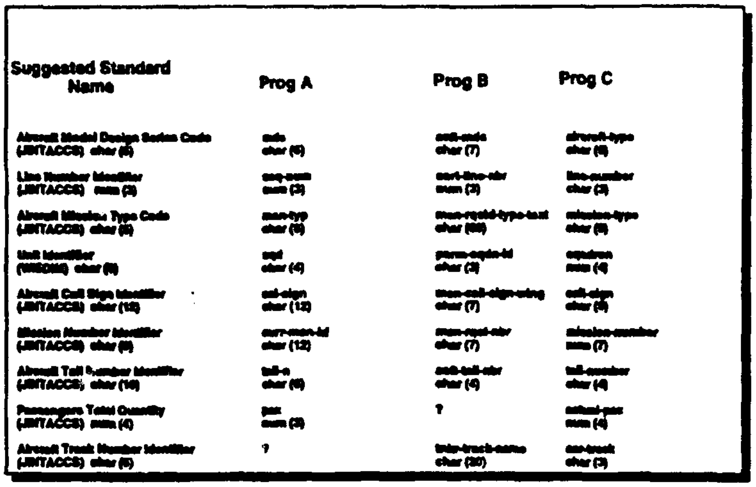
- C4I systems must exchange information for system interoperability
- Interoperability implies
  - Interconnection protocols allow systems to exchange bits
  - Systems within a user community have same representations for the same information, or else a means for translating between systems
- Existing C4I systems send and receive messages
  - directly when they have the same data standards and same internal definitions for data
  - by using translators, external or internal, when they do not have the same internal data representations

Data Element Format Mismatch -

Example



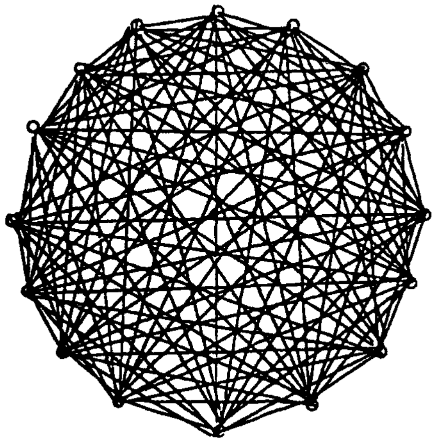
Data Element Format Mismatch -
Examples



Suggested Standard Name	Prog A	Prog B	Prog C
Aircraft Model Design Series Code (JINTACCS) char (6)	mds char (6)	acft-mds char (7)	aircraft-type char (8)
Line Number Identifier (JINTACCS) num (3)	seq-num num (3)	sort-line-nbr num (3)	line-number char (3)
Aircraft Missioa Type Code (JINTACCS) char (5)	msn-typ char (5)	msn-rqstd-type-text char (60)	mission-type char (5)
Unit Identifier (WSDMS) char (6)	sqd char (4)	parm-sqdn-id char (3)	squadron num (4)
Aircraft Call Sign Identifier (JINTACCS) char (12)	call-sign char (12)	msn-call-sign-wing char (7)	call-sign char (8)
Mission Number Identifier (JINTACCS) char (8)	curr-msn-id char (12)	msn-rqst-nbr char (7)	mission-number num (7)
Aircraft Tail Number Identifier (JINTACCS) char (10)	tail-n char (6)	acft-tail-nbr char (4)	tail-number char (4)
Passengers Total Quantity (JINTACCS) num (4)	pax num (3)	?	actual-pax num (4)
Aircraft Track Number Identifier (JINTACCS) char (5)	?	tndr-track-name char (20)	acr-track char (3)

MITRE

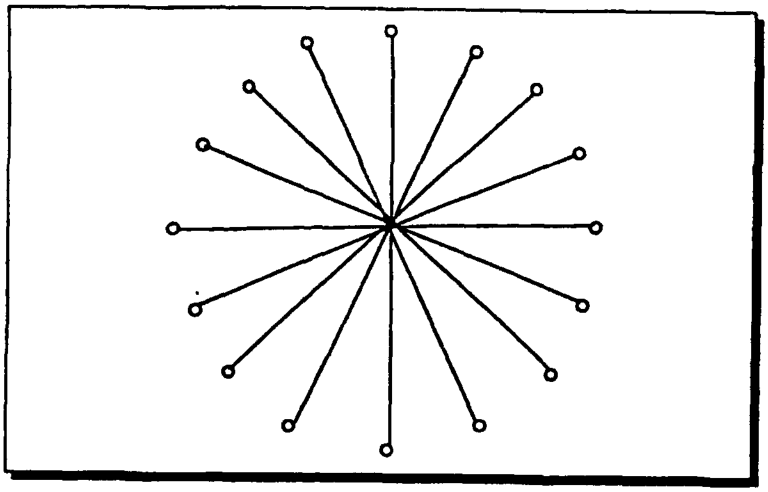
Data Element Standardization
Connection complexity without standards - Universal Interoperability



MITRE

Data Element Standardization

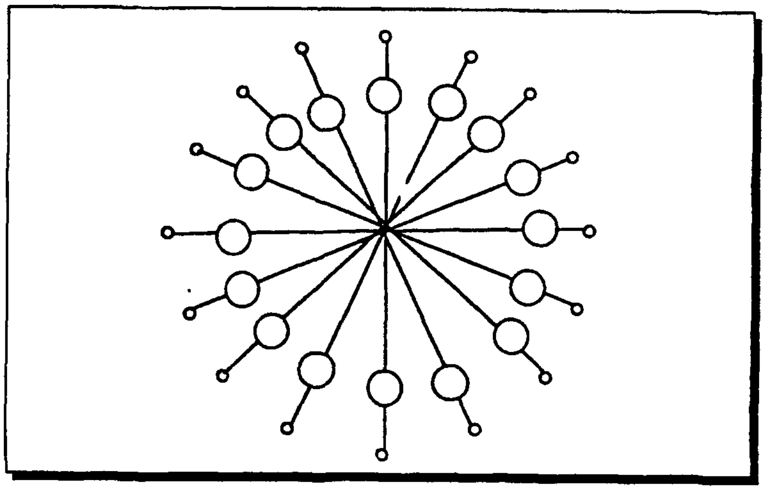
Connection complexity with standards - Universal Interoperability



MITRE

Data Element Standardization

Connection complexity with standards - Universal translator



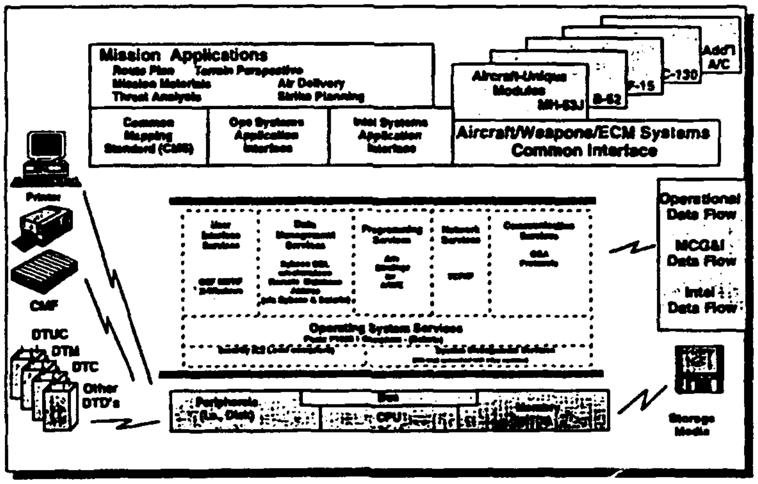
MITRE

BACKUP

MITRE

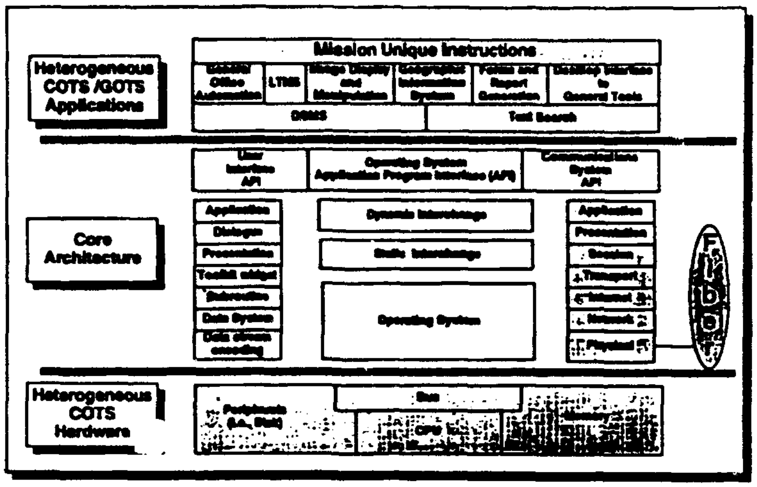
Program D Architecture -

Reusable components



MITRE

Technical Reference Model -
A Generic Version



Heterogeneous COTS /GOTS Applications

Mission Unique Instructions

Desktop Office Automation | LTMS | Image Display and Manipulation | Geographic Information System | Forms and Report Generation | Desktop Interface to General Tools

DBMS | Text Search

User Interface API | Operating System Application Program Interface (API) | Communications System API

Core Architecture

Application | Dialogue | Presentation | Toolkit widget | Subroutine | Data System | Data stream encoding

Dynamic Interchange | Static Interchange | Operating System

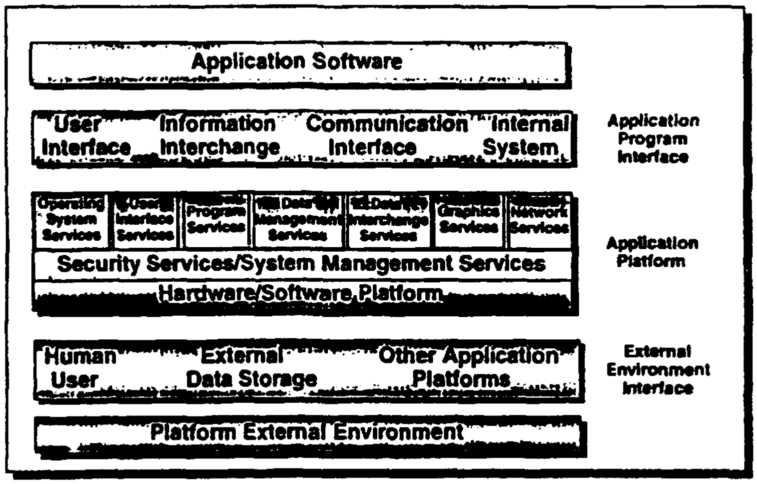
Application | Presentation | Session | Transport | Internet | Network | Physical

Heterogeneous COTS Hardware

Peripherals (i.e., Disk) | Bus | CPU | Memory

MITRE

Technical Reference Model -
NIST Application Portability Profile



Application Software

User Interface | Information Interchange | Communication Interface | Internal System

Application Program Interface

Operating System Services | User Interface Services | Program Services | Data Management Services | Data Interchange Services | Graphics Services | Network Services

Security Services/System Management Services

Hardware/Software Platform

Application Platform

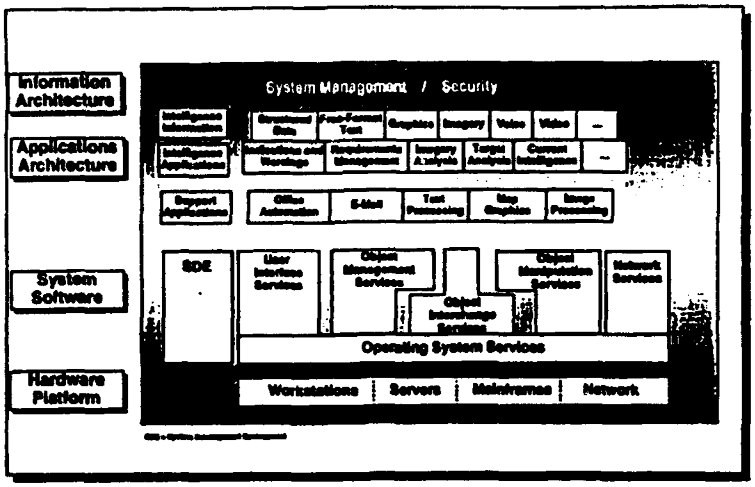
Human User | External Data Storage | Other Application Platforms

External Environment Interface

Platform External Environment

MITRE

Technical Reference Model -
DODIIS



Information Architecture

Applications Architecture

System Software

Hardware Platform

System Management / Security

Intelligence Information

Structured Data

Free-Format Text

Graphics

Imagery

Voice

Video

...

Intelligence Applications

Indications and Warnings

Requirements Management

Imagery Analysis

Target Analysis

Current Intelligence

...

Support Applications

Office Automation

E-Mail

Text Processing

Map Graphics

Image Processing

SDE

User Interface Services

Object Management Services

Object Interchange Services

Object Manipulation Services

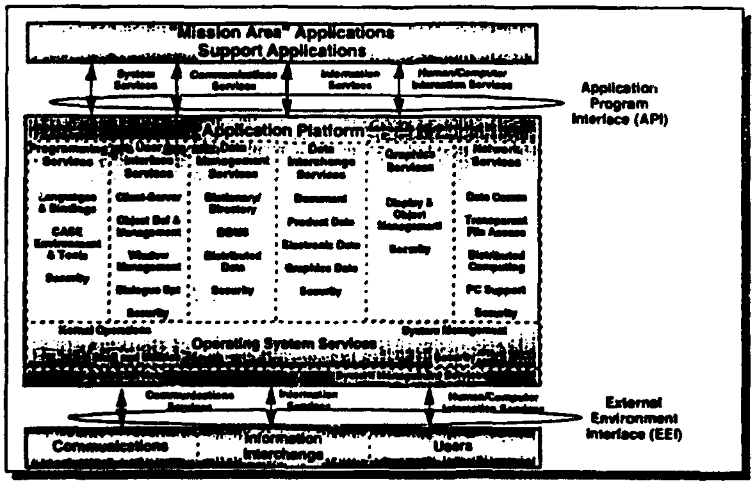
Network Services

Operating System Services

Workstations | Servers | Mainframes | Network

MITRE

Technical Reference Model -
DISA Technical Reference Model for Information Management (v 1.3)



"Mission Area" Applications

Support Applications

System Services

Communications Services

Information Services

Human/Computer Interaction Services

Application Program Interface (API)

Application Platform

Programming Services

User Interface Services

Data Management Services

Data Interchange Services

Graphics Services

Network Services

Languages & Bindings

CASE Environment & Tools

Security

Client-Server

Object Def & Management

Window Management

Dialogue Spt

Security

Dictionary/Directory

DBMS

Distributed Data

Security

Document

Product Data

Electronic Data

Graphics Data

Security

Display & Object Management

Security

Data Comm

Transparent File Access

Distributed Computing

PC Support

Security

Kernel Operations

System Management

Operating System Services

Communications Services

Information Services

Human/Computer Interaction Services

External Environment Interface (EEI)

Communications

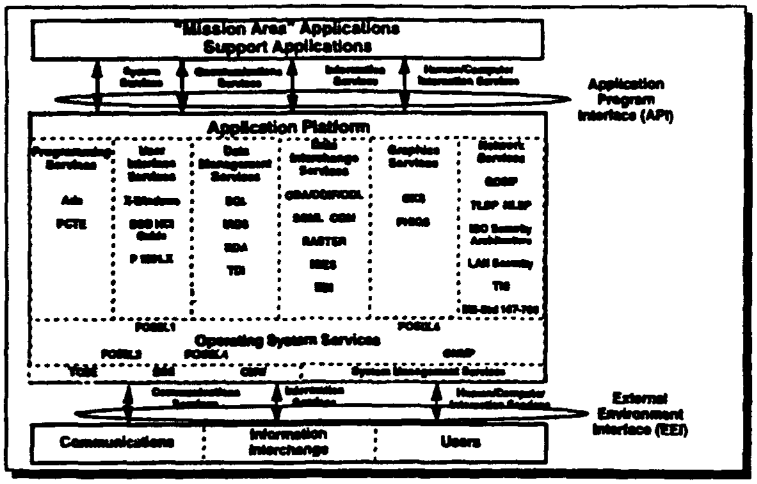
Information Interchange

Users

MITRE

Technical Reference Model - The Profile

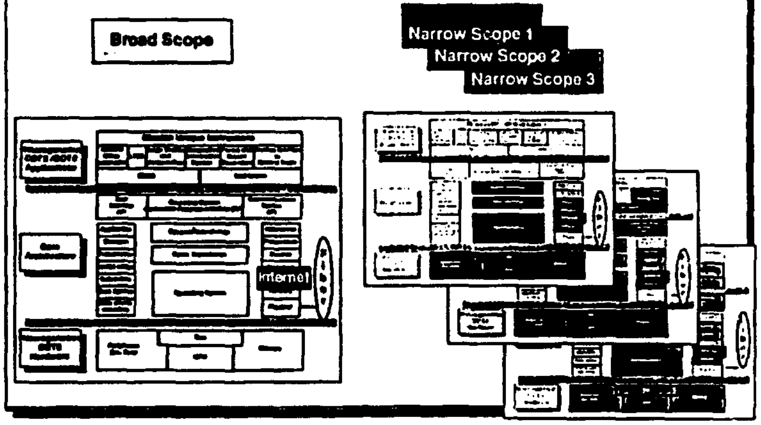
DISA The DoD Profile of Standards (v 1.3)



MITRE

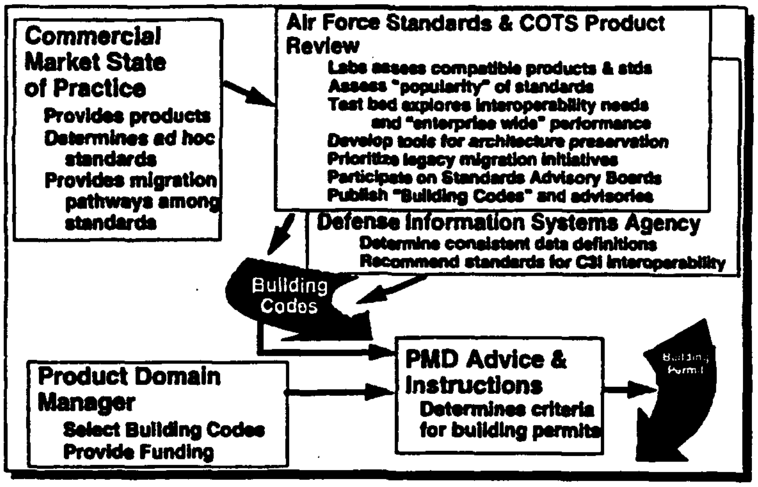
The "Building Codes"

Building codes should be established to correspond to different degrees of detail and content (as appropriate) depending upon the "scope of the enterprise"



MITRE

How to Derive the "Building Codes"



Commercial Market State of Practice

- Provides products
- Determines ad hoc standards
- Provides migration pathways among standards

Air Force Standards & COTS Product Review

- Labs asses compatible products & stds
- Assess "popularity" of standards
- Test bed explores interoperability needs and "enterprise wide" performance
- Develop tools for architecture preservation
- Prioritize legacy migration initiatives
- Participate on Standards Advisory Boards
- Publish "Building Codes" and advisories

Defense Information Systems Agency

- Determine consistent data definitions
- Recommend standards for C3I interoperability

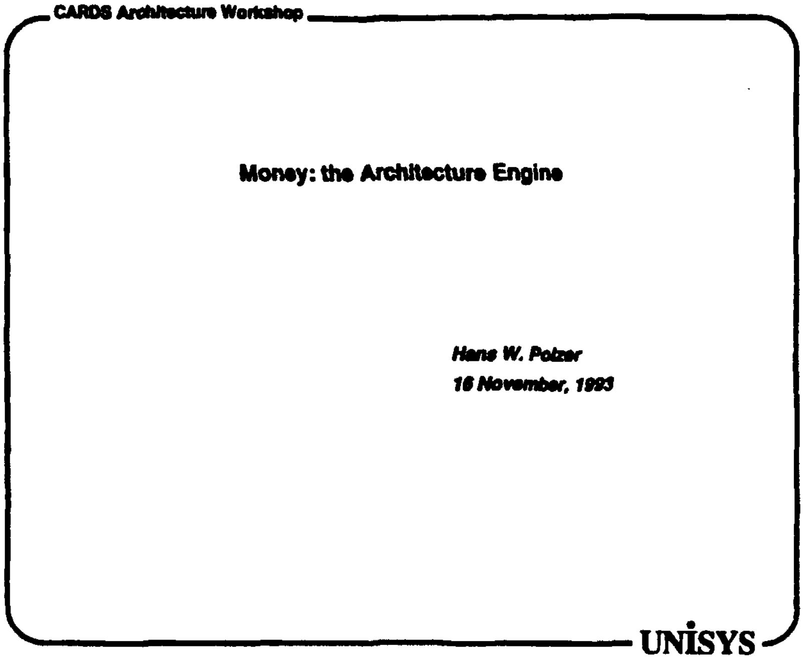
Product Domain Manager

- Select Building Codes
- Provide Funding

PMD Advice & Instructions

- Determines criteria for building permits

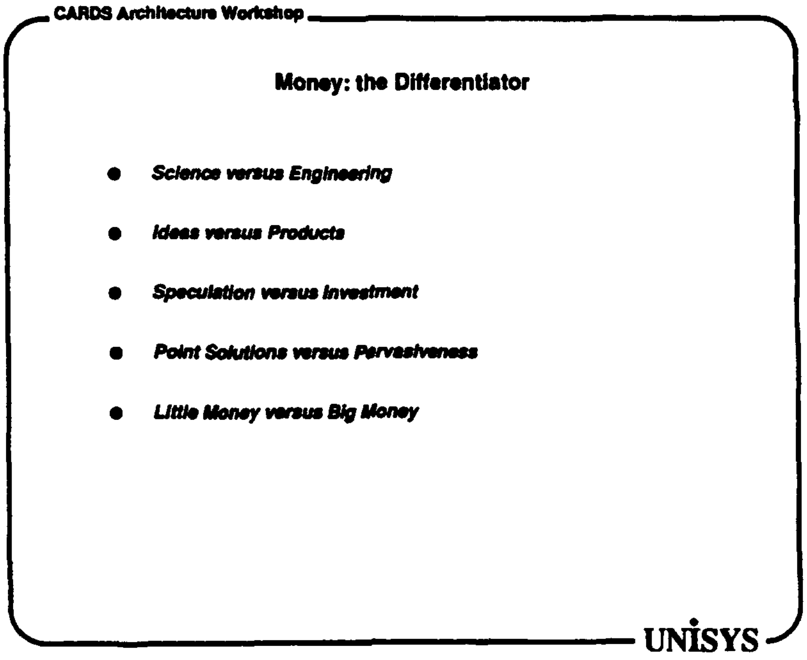
MITRE


CARDS Architecture Workshop

Money: the Architecture Engine

Hans W. Polzer
16 November, 1993

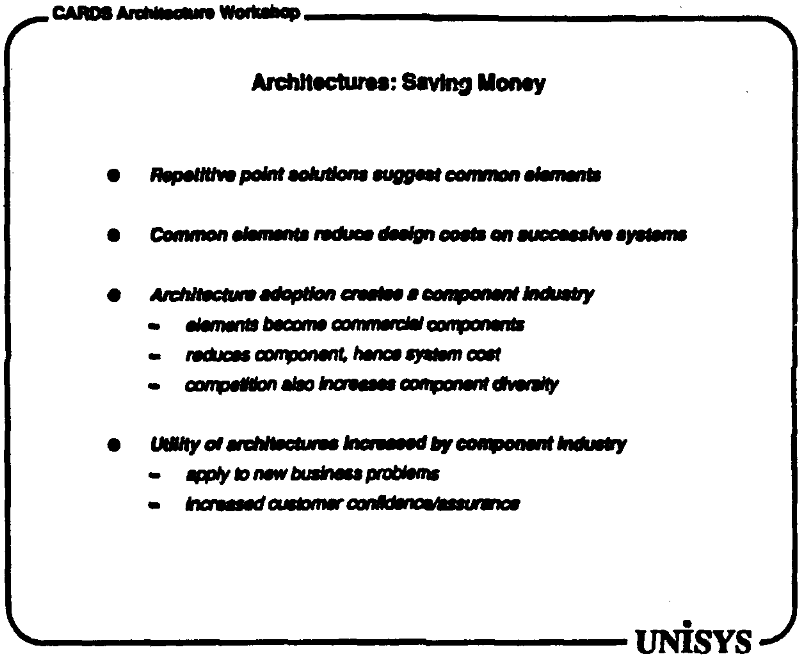
UNISYS


CARDS Architecture Workshop

Money: the Differentiator

- *Science versus Engineering*
- *Ideas versus Products*
- *Speculation versus Investment*
- *Point Solutions versus Pervasiveness*
- *Little Money versus Big Money*

UNISYS

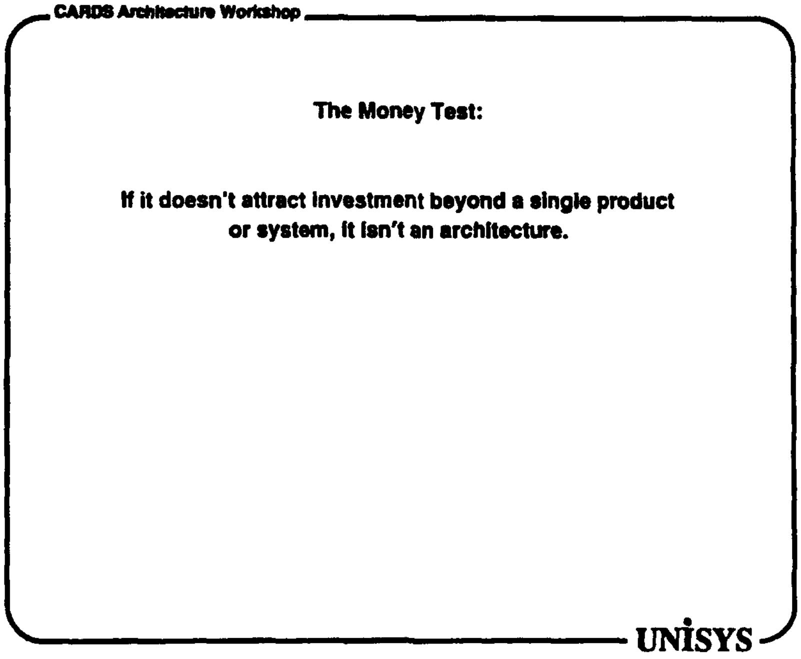


CARDS Architecture Workshop

Architectures: Saving Money

- *Repetitive point solutions suggest common elements*
- *Common elements reduce design costs on successive systems*
- *Architecture adoption creates a component industry*
  - *elements become commercial components*
  - *reduces component, hence system cost*
  - *competition also increases component diversity*
- *Utility of architectures increased by component industry*
  - *apply to new business problems*
  - *increased customer confidence/assurance*

UNISYS

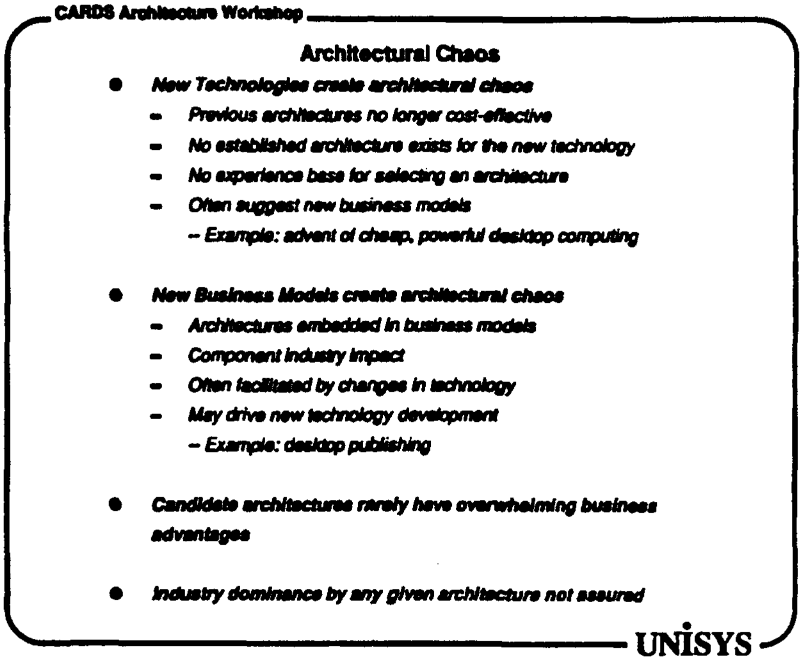


CARDS Architecture Workshop

The Money Test:

If it doesn't attract investment beyond a single product or system, it isn't an architecture.

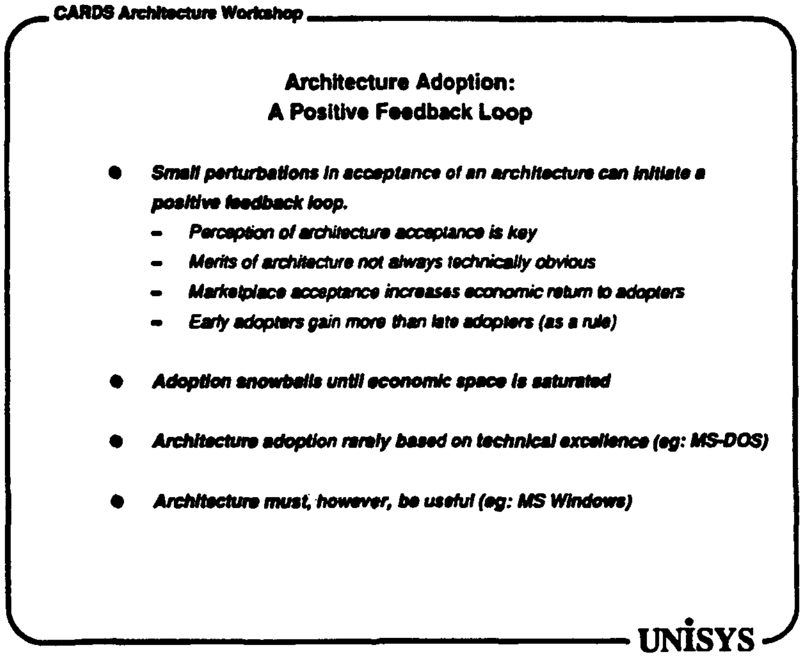
UNISYS


CARDS Architecture Workshop

Architectural Chaos

- *New Technologies create architectural chaos*
  - *Previous architectures no longer cost-effective*
  - *No established architecture exists for the new technology*
  - *No experience base for selecting an architecture*
  - *Often suggest new business models*
    - *Example: advent of cheap, powerful desktop computing*
- *New Business Models create architectural chaos*
  - *Architectures embedded in business models*
  - *Component industry impact*
  - *Often facilitated by changes in technology*
  - *May drive new technology development*
    - *Example: desktop publishing*
- *Candidate architectures rarely have overwhelming business advantages*
- *Industry dominance by any given architecture not assured*

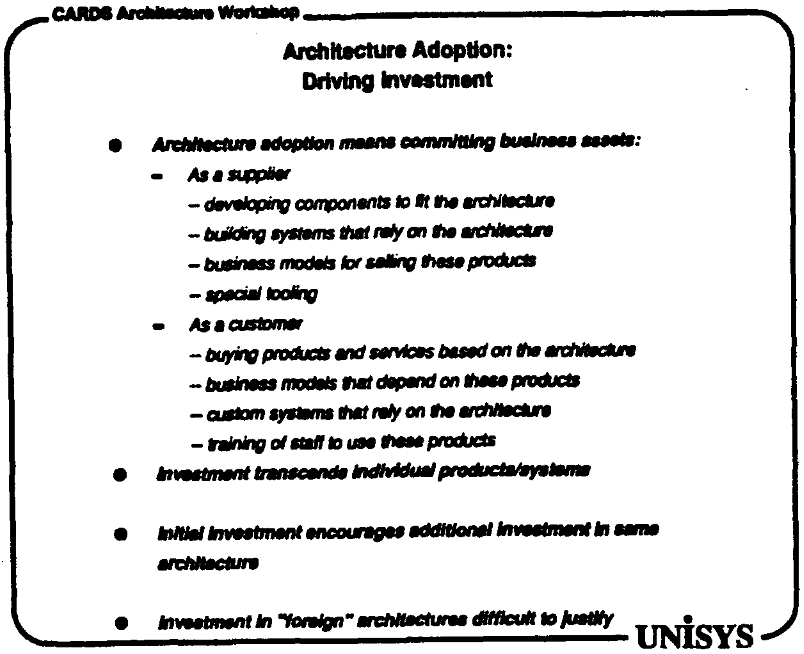
UNISYS


CARDS Architecture Workshop

Architecture Adoption: A Positive Feedback Loop

- *Small perturbations in acceptance of an architecture can initiate a positive feedback loop.*
  - *Perception of architecture acceptance is key*
  - *Merits of architecture not always technically obvious*
  - *Marketplace acceptance increases economic return to adopters*
  - *Early adopters gain more than late adopters (as a rule)*
- *Adoption snowballs until economic space is saturated*
- *Architecture adoption rarely based on technical excellence (eg: MS-DOS)*
- *Architecture must, however, be useful (eg: MS Windows)*

UNISYS

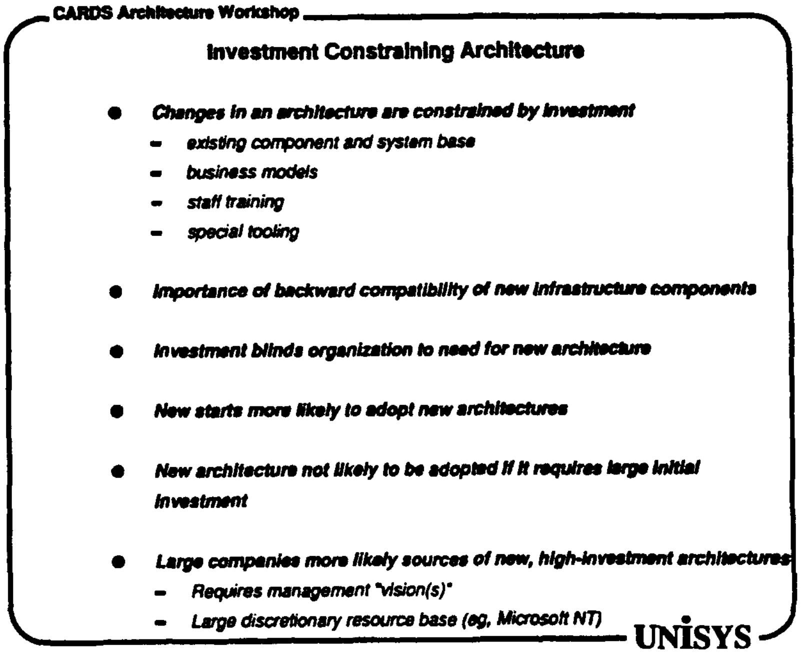


CARDS Architecture Workshop

Architecture Adoption: Driving Investment

- *Architecture adoption means committing business assets:*
  - *As a supplier*
    - *developing components to fit the architecture*
    - *building systems that rely on the architecture*
    - *business models for selling these products*
    - *special tooling*
  - *As a customer*
    - *buying products and services based on the architecture*
    - *business models that depend on these products*
    - *custom systems that rely on the architecture*
    - *training of staff to use these products*
- *Investment transcends individual products/systems*
- *Initial investment encourages additional investment in same architecture*
- *Investment in "foreign" architectures difficult to justify*

UNISYS

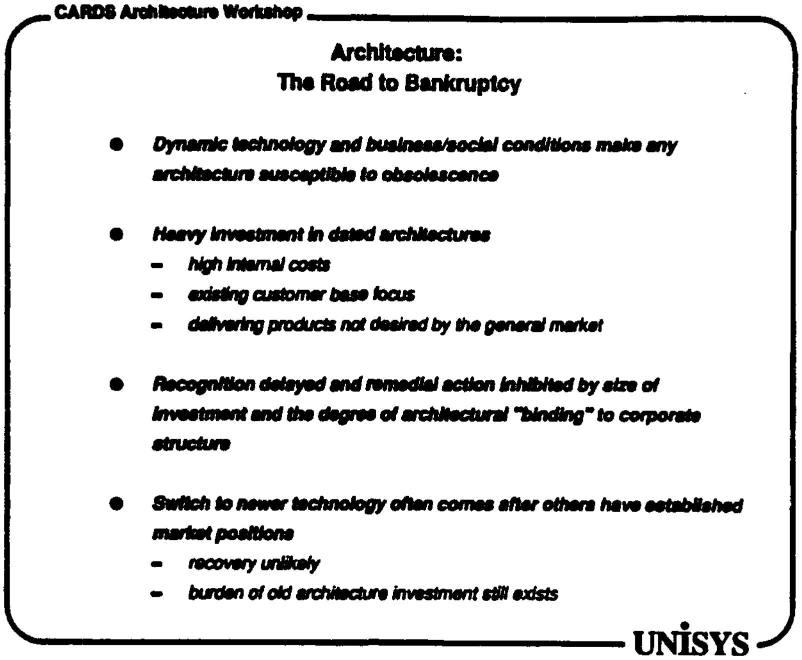


CARDS Architecture Workshop

Investment Constraining Architecture

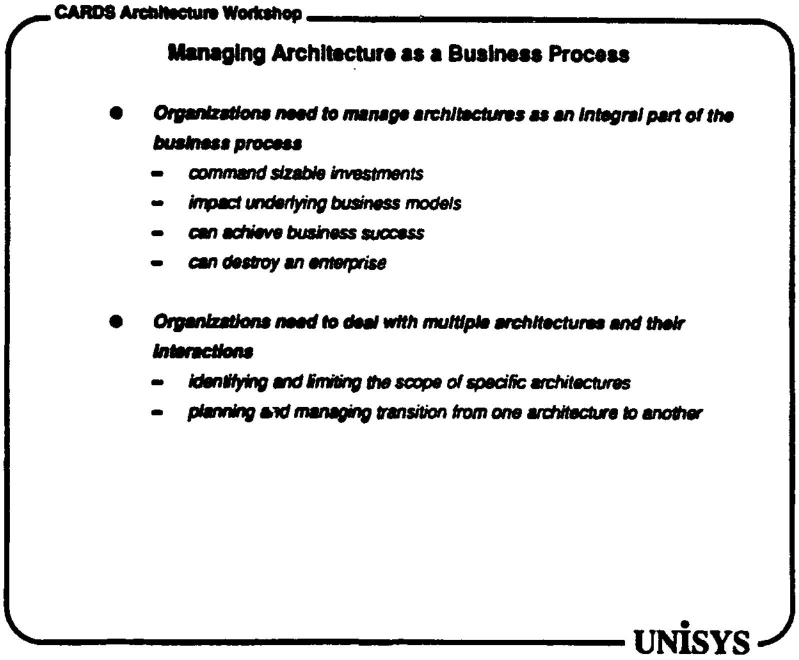
- *Changes in an architecture are constrained by investment*
  - *existing component and system base*
  - *business models*
  - *staff training*
  - *special tooling*
- *Importance of backward compatibility of new infrastructure components*
- *Investment blinds organization to need for new architecture*
- *New starts more likely to adopt new architectures*
- *New architecture not likely to be adopted if it requires large initial investment*
- *Large companies more likely sources of new, high-investment architectures*
  - *Requires management "vision(s)"*
  - *Large discretionary resource base (eg, Microsoft NT)*

UNISYS


CARDS Architecture Workshop

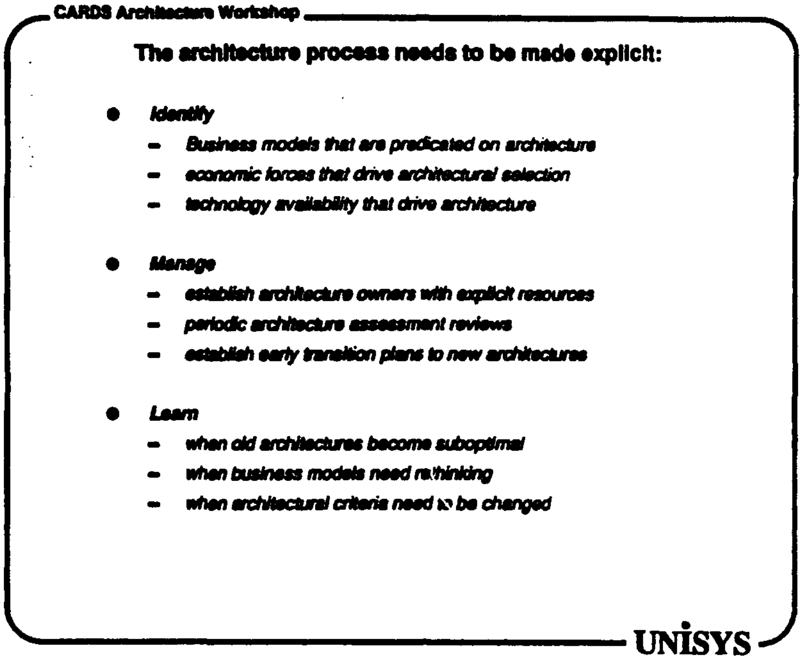
Architecture: The Road to Bankruptcy

- *Dynamic technology and business/social conditions make any architecture susceptible to obsolescence*
- *Heavy investment in dated architectures*
  - *high internal costs*
  - *existing customer base focus*
  - *delivering products not desired by the general market*
- *Recognition delayed and remedial action inhibited by size of investment and the degree of architectural "binding" to corporate structure*
- *Switch to newer technology often comes after others have established market positions*
  - *recovery unlikely*
  - *burden of old architecture investment still exists*


CARDS Architecture Workshop

Managing Architecture as a Business Process

- *Organizations need to manage architectures as an integral part of the business process*
  - *command sizable investments*
  - *impact underlying business models*
  - *can achieve business success*
  - *can destroy an enterprise*
- *Organizations need to deal with multiple architectures and their interactions*
  - *identifying and limiting the scope of specific architectures*
  - *planning and managing transition from one architecture to another*

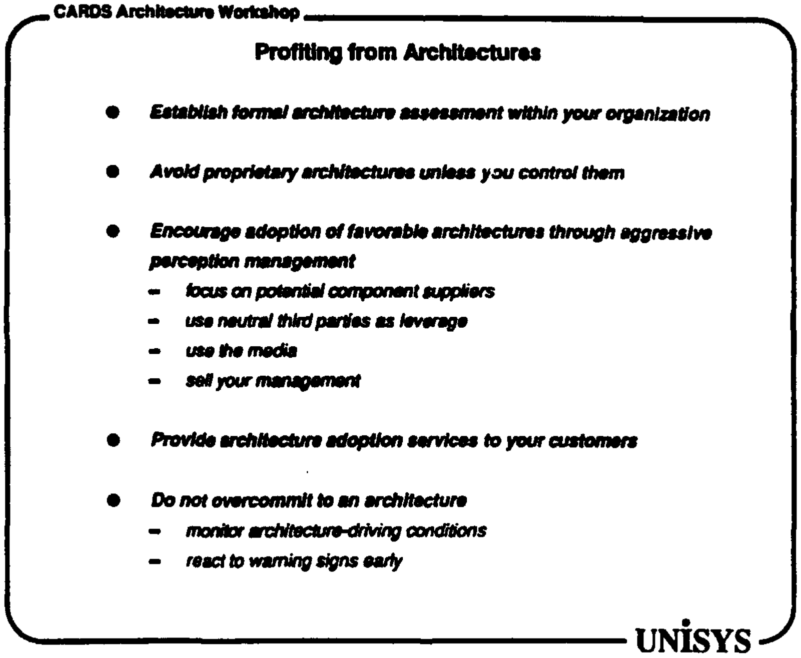


CARDS Architecture Workshop

The architecture process needs to be made explicit:

- *Identify*
  - *Business models that are predicated on architecture*
  - *economic forces that drive architectural selection*
  - *technology availability that drive architecture*
- *Manage*
  - *establish architecture owners with explicit resources*
  - *periodic architecture assessment reviews*
  - *establish early transition plans to new architectures*
- *Learn*
  - *when old architectures become suboptimal*
  - *when business models need rethinking*
  - *when architectural criteria need to be changed*

UNISYS



CARDS Architecture Workshop

Profiting from Architectures

- ***Establish formal architecture assessment within your organization***
- ***Avoid proprietary architectures unless you control them***
- ***Encourage adoption of favorable architectures through aggressive perception management***
  - *focus on potential component suppliers*
  - *use neutral third parties as leverage*
  - *use the media*
  - *sell your management*
- ***Provide architecture adoption services to your customers***
- ***Do not overcommit to an architecture***
  - *monitor architecture-driving conditions*
  - *react to warning signs early*

UNISYS

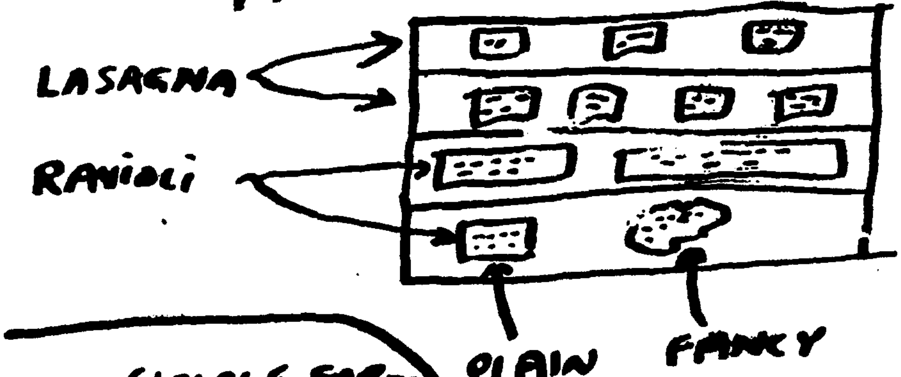
ARCHITECTURAL DEVELOPMENTS

ARMY
COMMAND AND CONTROL SYSTEM
COMMON SOFTWARE PROGRAM

STANLEY H. LEVINE
DEPUTY PROJECT MANAGER
COMMON HARDWARE SOFTWARE

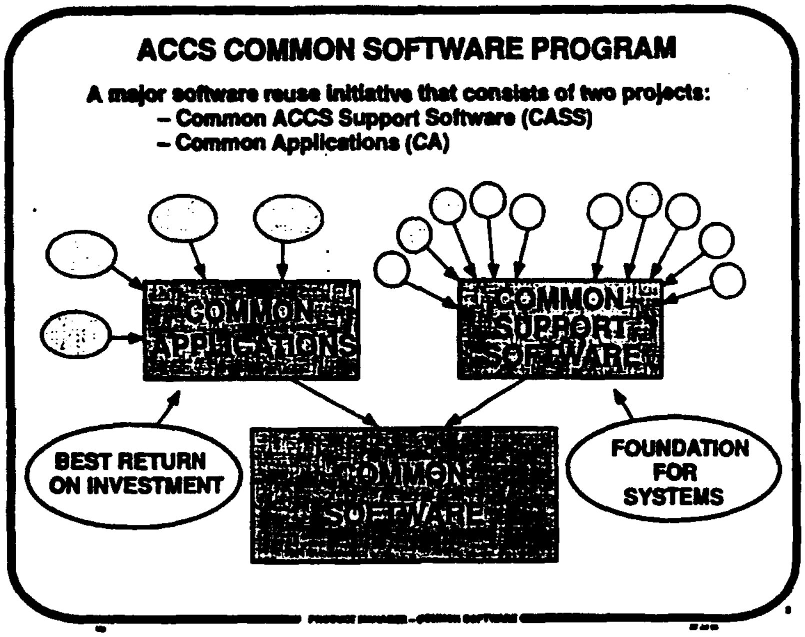
PRODUCT MANAGER - COMMON SOFTWARE

ARCHITECTURE



- SIMPLE FORM
- SHAPES
- LAYERS
- NO CONNECTIONS
- SIMPLE NAMES

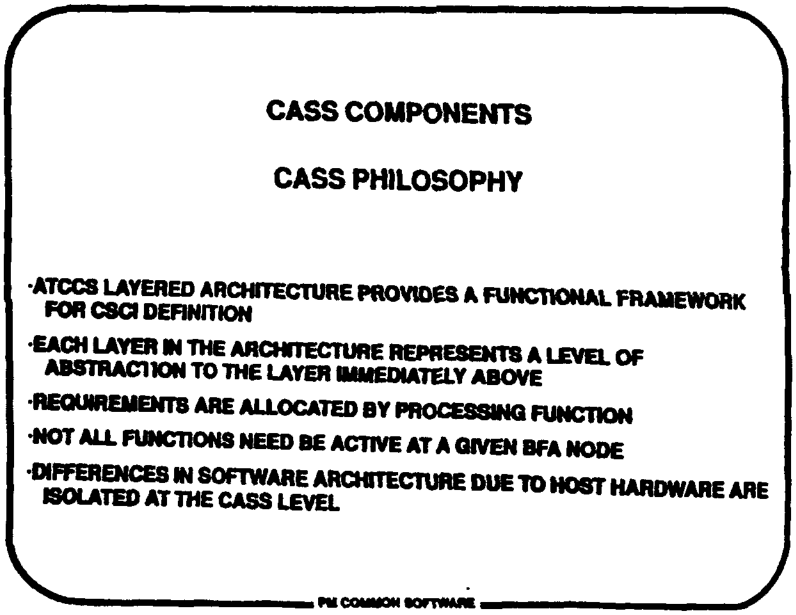
DEFINED IN 1 VIEWGRAPH



ACCS COMMON SOFTWARE PROGRAM

A major software reuse initiative that consists of two projects:
- Common ACCS Support Software (CASS)
- Common Applications (CA)

PRODUCT MANAGER—COMMON SOFTWARE

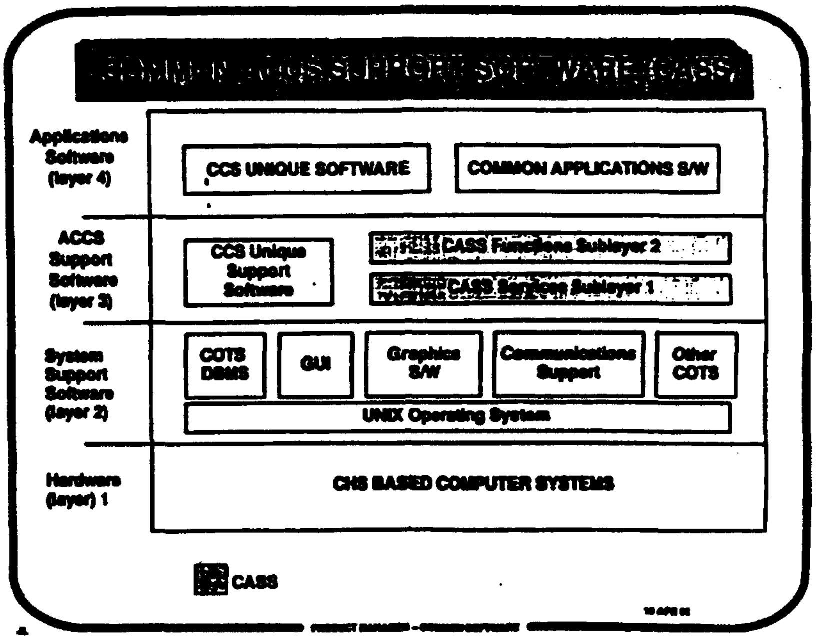


CASS COMPONENTS

CASS PHILOSOPHY

- ATCCS LAYERED ARCHITECTURE PROVIDES A FUNCTIONAL FRAMEWORK FOR CSCI DEFINITION
- EACH LAYER IN THE ARCHITECTURE REPRESENTS A LEVEL OF ABSTRACTION TO THE LAYER IMMEDIATELY ABOVE
- REQUIREMENTS ARE ALLOCATED BY PROCESSING FUNCTION
- NOT ALL FUNCTIONS NEED BE ACTIVE AT A GIVEN BFA NODE
- DIFFERENCES IN SOFTWARE ARCHITECTURE DUE TO HOST HARDWARE ARE ISOLATED AT THE CASS LEVEL

PM COMMON SOFTWARE



COMMON ACCS SUPPORT SOFTWARE (CASS)

Applications Software (layer 4)
- CCS UNIQUE SOFTWARE
- COMMON APPLICATIONS S/W

ACCS Support Software (layer 3)
- CCS Unique Support Software
- CASS Functions Sublayer 2
- CASS Services Sublayer 1

System Support Software (layer 2)
- COTS DBMS
- GUI
- Graphics S/W
- Communications Support
- Other COTS
- UNIX Operating System

Hardware (layer) 1
- CHS BASED COMPUTER SYSTEMS

CASS

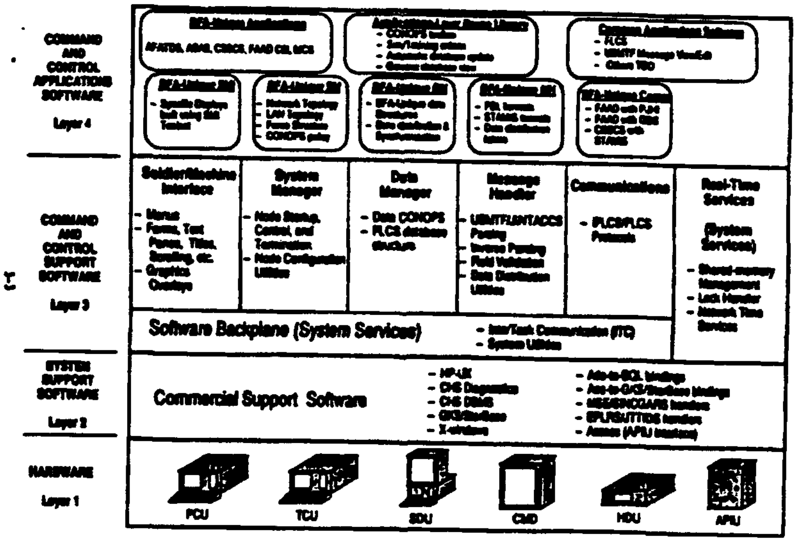

COMMAND AND CONTROL APPLICATIONS SOFTWARE
Layer 4
AFATDS, ASAS, CSSCS, FAAD C2I, MCS
- CONOPS
- FLCS
COMMAND AND CONTROL SUPPORT SOFTWARE
Layer 3
Soldier/Machine Interface
- Menus
- Forms, Text Panes, Titles, Scrolling, etc.
- Graphics Overlays
System Manager
- Node Status, Control, and Termination
- Node Configuration Utilities
Data Manager
- Data CONOPS
- FLCS database structure
Message Handler
- Parsing
- Field Validation
- Data Distribution Utilities
Communications
- FLCS/FLCS Protocols
Real-Time Services
(System Services)
- Shared-memory Management
- Lock Handler
- Network Time Services
Software Backplane (System Services)
- Inter/Task Communication (ITC)
- System Utilities
SYSTEM SUPPORT SOFTWARE
Layer 2
Commercial Support Software
- HP-UX
- CHS Diagnostics
- CHS DBMS
- X-windows
- Ada-to-SQL bindings
- Annex (APIU interface)
HARDWARE
Layer 1
PCU
TCU
SDU
CMD
HDU
APIU

Figure 3.1-1. ATCCS Common Software Architecture

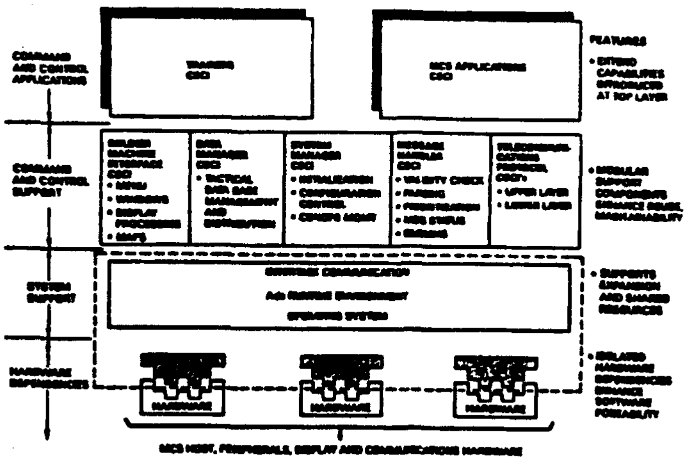

COMMAND AND CONTROL APPLICATIONS
TRAINING CSCI
MCS APPLICATIONS CSCI
FEATURES
• EXTEND CAPABILITIES INTRODUCED AT TOP LAYER
COMMAND AND CONTROL SUPPORT
SOLDIER MACHINE INTERFACE CSCI
• MENUS
• WINDOWING
• DISPLAY PROCESSING
• MAPS
DATA MANAGER CSCI
• TACTICAL DATA BASE MANAGEMENT AND DISTRIBUTION
SYSTEM MANAGER CSCI
• INITIALIZATION
• CONFIGURATION CONTROL
• [illegible]
MESSAGE HANDLER CSCI
• VALIDITY CHECK
• PARSING
• PRESENTATION
• MSG STATUS
• ROUTING
TELECOMMUNICATIONS PROTOCOL CSCIs
• UPPER LAYER
• LOWER LAYER
• MODULAR SUPPORT COMPONENTS ENHANCE REUSE, MAINTAINABILITY
SYSTEM SUPPORT
INTERTASK COMMUNICATION
Ada RUNTIME ENVIRONMENT
OPERATING SYSTEM
• SUPPORTS EXPANSION AND SHARED RESOURCES
HARDWARE DEPENDENCIES
HARDWARE
HARDWARE
HARDWARE
• ISOLATED HARDWARE DEPENDENCIES ENHANCE SOFTWARE PORTABILITY
MCS HOST, PERIPHERALS, DISPLAY AND COMMUNICATIONS HARDWARE

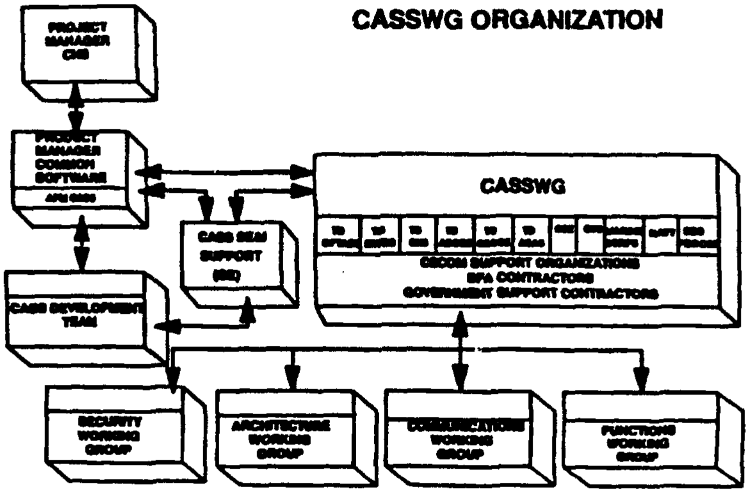
Figure 3.1-2. MCS Segment 11 Architecture

D1345
08 June 1990

CASS CODE SIZE ESTIMATES

CSCI	LINES OF CODE (X 1000)
SYSTEM SERVICES	15
SOLDIER-MACHINE INTERFACE	12
SYSTEM MANAGER	8
DATA MANAGER	10
MESSAGE HANDLER	16
COMMUNICATIONS	27
TOTAL	88

CASSWG ORGANIZATION


PROJECT MANAGER CHS
PROJECT MANAGER COMMON SOFTWARE
AFM CASS
CASS SEAM SUPPORT (SE)
CASS DEVELOPMENT TEAM
CASSWG
CECOM SUPPORT ORGANIZATIONS
SFA CONTRACTORS
GOVERNMENT SUPPORT CONTRACTORS
SECURITY WORKING GROUP
ARCHITECTURE WORKING GROUP
COMMUNICATIONS WORKING GROUP
FUNCTIONS WORKING GROUP

PM COMMON SOFTWARE

CASS ARCHITECTURE WORKING GROUP (ARCHWG)

PURPOSE: TO DEFINE CASS ARCHITECTURE AND THE SOFTWARE BACKPLANE REQUIREMENTS

ACTIVITIES

- DEFINE TOP–LEVEL CASS ARCHITECTURE FUNCTIONS
- PREPARE SOFTWARE REQUIREMENTS SPEC AND ADA SPECIFICATIONS FOR THE ITC
- DETERMINE CASS STANDARDS AND METRICS, DEFINE ADA BINDINGS, SELECT COMMON APPLICATIONS

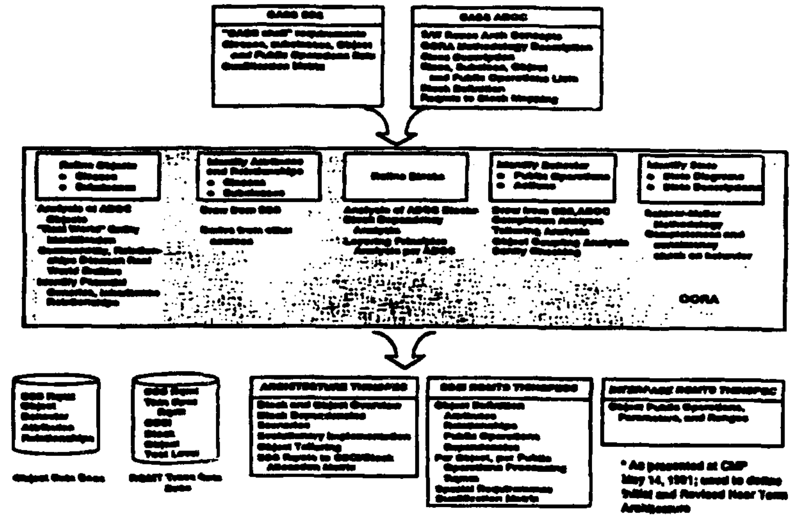
CHAIR: BRUCE GRAY, CSE

PM COMMON SOFTWARE

CDT

Near Term Architecture Definition Methodology*

TRW

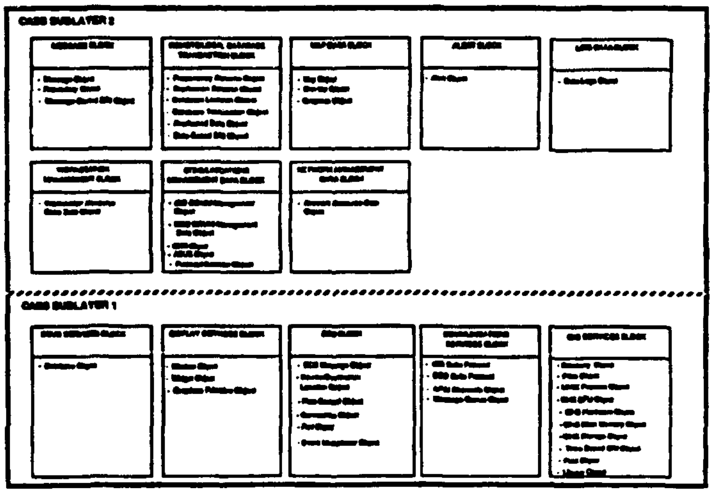


* As presented at CMP May 14, 1991; used to define Initial and Revised Near Term Architecture

CDT

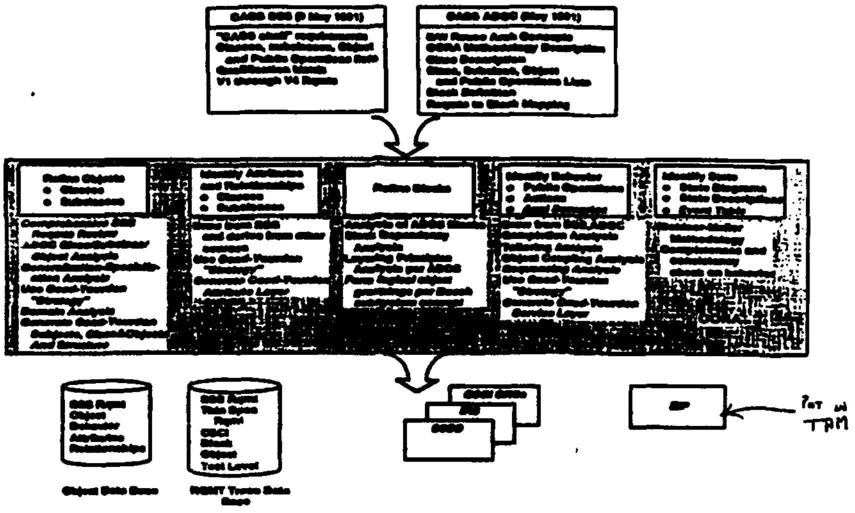
Revised Near Term Architecture *

TRW

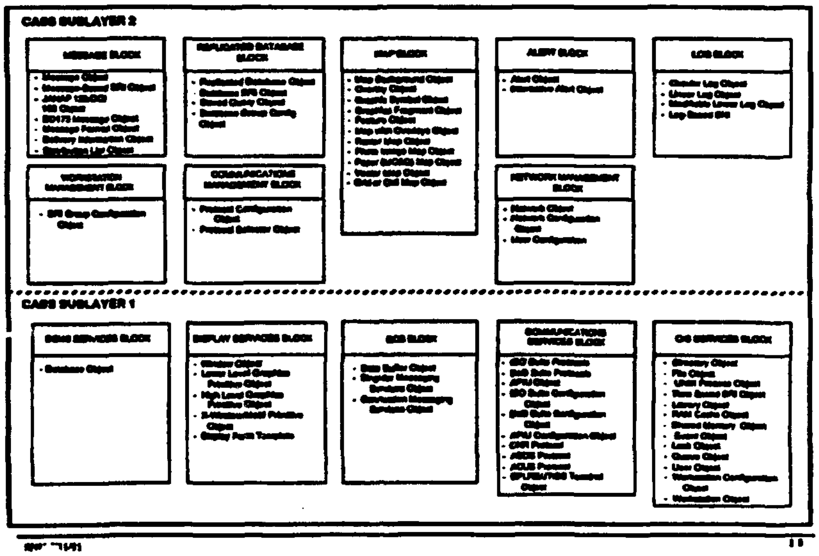


* As shown in the V6.1 Thin Spec Addendum

CDT | **Objective Architecture Definition Methodology** | TRW



CDT | **CASS Objective Architecture** | TRW



CDT	Comparison of Near Term Arch with Objective Arch	TRW

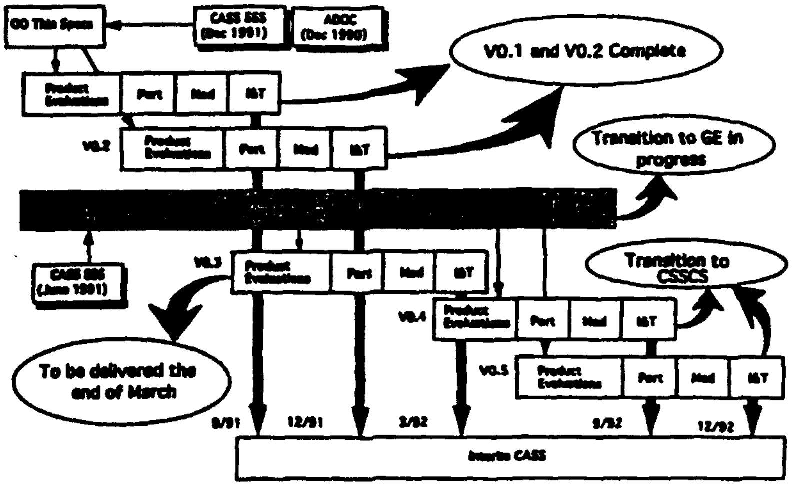
Architecture	Number of Blocks	Number of Objects	SSS Date	Versions Considered	Reason for differences
Initial Near Term Architecture	13	36	Dec 1991	V1/Some of V2	
Revised Near Term Architecture - used for V0.1 and V0.2 CDT activities	13	45	Dec 1991	V1/Some of V2	Broke BCS into separate objects, added two objects some object name changes and movement between blocks
Objective Architecture	13	66	May 1991	V1 through V4	18 Objects Added for V2 - V4 Rqmts 3 fewer Objects due to grouping into user selectable objects 6 added objects result from thin spec comments

CDT	Product Availability Issues	TRW

- Issue: MCS and AFATDS products appear to be the best match for CASS requirements but they will not be available when needed to complete product evaluations within schedule
- Resolution:
  - Meet with MCS regarding Loral CSCIs and AFATDS to develop workarounds (e.g., Beta Release, draft documentation, etc.) for as many objects as possible
  - Identify alternate products for near term and develop plans for upgrading CASS when AFATDS and MCS products become available

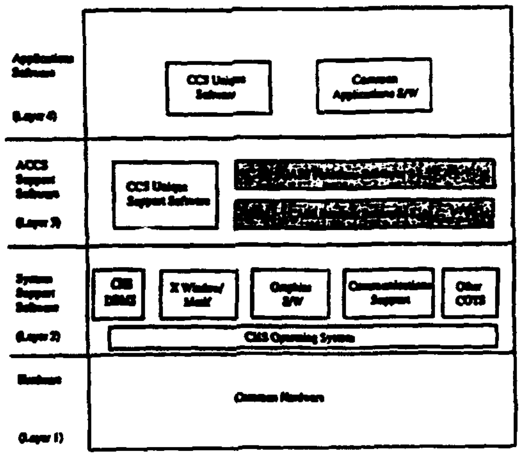
CDT

CDT Activities and Status

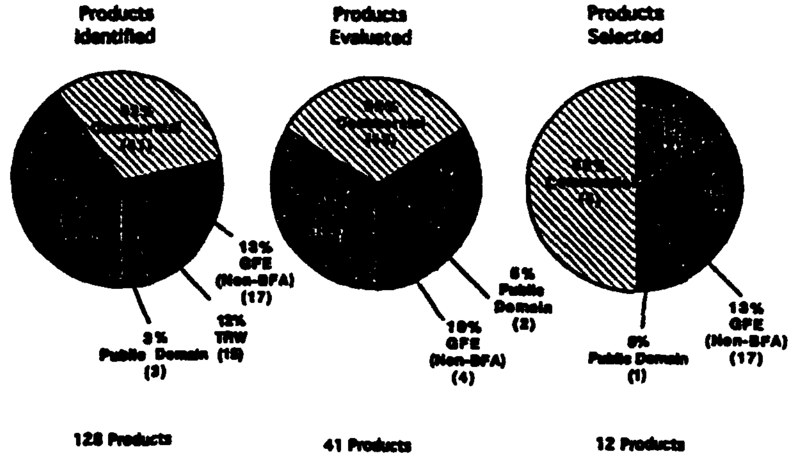


CDT

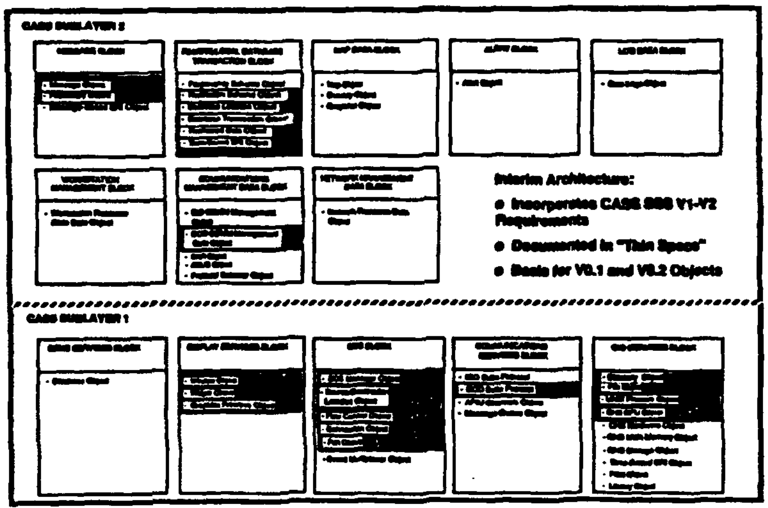
ACCS 4-Layer Architecture



CDT	Accomplishments (Continued) Product Summary	TRW



CDT	CASS Interim Architecture	TRW



Interim Architecture:

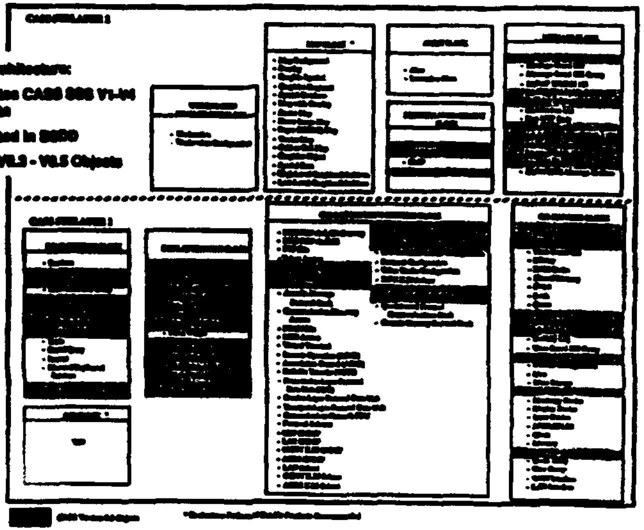
- Incorporates CASS SSS V1-V2 Requirements
- Documented in "Thin Specs"
- Basis for V0.1 and V0.2 Objects

CASS Version 0.2 Objects

CDT

CASS Objective Architecture

TRW



Objective Architecture:

- Incorporates CASS SSS V1-V4 Requirements
- Documented in SSDD
- Basis for V1.3 - V1.5 Objects

CDT

Lessons Learned Object Oriented Analysis

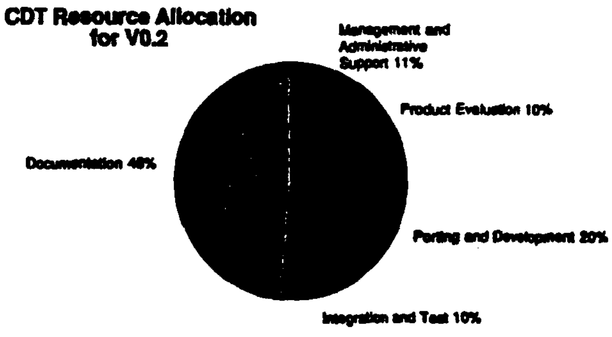
TRW

- Enhances Reusability
  - Focuses on Identification of Interfaces
  - Highlights Object Relationships and Dependencies
  - Good Toolkit Design Mechanism
- OOA Immature
  - Gap Between OOA and OOD
  - Requires Additional Techniques to Fully Depict System Design (e.g., Processing Sequences, Data Flow Diagrams, etc.)
  - Lacks a Good Decomposition Technique
- Documenting Results of OOA Produces Large Documents

CDT | Lessons Learned Software Reuse | TRW

- Recognize Schedule Risk in Your Planning When Using GFE Products
- Rigorous Product Evaluations Essential
  - Eliminate Immature Products
  - Reduce Integration Risk
  - Ensure Key Evaluation Criteria Met (Requirements, Architecture, Reuse)
- SLOC or Numbers of Objects Are Not Accurate Measures of the Porting and Development Effort Associated With a Release
  - e.g., it may require less effort to port a large product from one version of X-windows/MOTIF to another than to port a small product from the ICC to the ALSYS compiler
  - Some Objects Are Very Large, Some Are Small, and Some May Be Mostly COTS
- The Extent and Quality of the Documentation of Products to be Ported Has a Significant Impact on the Cost of Each Release
- Responsiveness of Product Developers to CDT Technical Questions and Extent Product Developed With Naming Conventions and Coding Standards Affects CDT Productivity

CDT | Lessons Learned Software Reuse (continued) | TRW


CDT Resource Allocation for V0.2

The extent and quality of the documentation of products to be ported has a significant impact on the cost of each release

CDT

Lessons Learned COTS Interface

TRW

- CASS Developed S/W Interface Preferable to COTS S/W Interface
  - Product Independence
  - Portability
- CASS Developed COTS S/W Interface Provides BFAs a S/W Layer Buffer from COTS Product Changes

GE

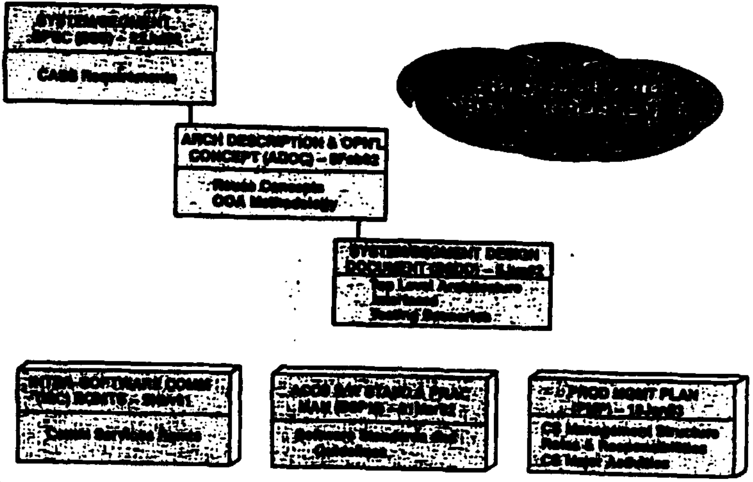
CHANGE OF APPROACH

ATCCS
SE&I

FINAL APPROACH	ORIGINAL APPROACH
CASS REQUIREMENTS DRIVEN	MCS DESIGN DRIVEN
EMBEDDED EXCESS MCS FUNCTIONALITY, LIMITED IMPLEMENTATION FLEXIBILITY AND PERFORMANCE OF CASS MODULES	
ATCCS MESSAGE PROCESSING	MCS MESSAGE PROCESSING
SUPPORTS PROCESSING DIFFERENT KINDS OF MESSAGES NOT JUST USMTF	
OBJECT ORIENTED	FUNCTION DECOMPOSITION
LESS DIFFICULTY IN SW MAINTENANCE & TRANSITION TO CHS2 AND CHANGES	
EXPLOIT MULTIPLE SOURCES	RELY ON SINGLE SOURCE
DISTRIBUTED RISK, MORE LIKELY TO GET SOME OF THE SOFTWARE	
ATCCS PRODUCT MANAGEMENT	BFA PRODUCT MANAGEMENT
SUPPORTS CASS' INDEPENDENCE FROM A NODAL PM	
CASS AS A TOOLKIT	CASS AS A SINGLE ENTITY
SUPPORTS CASS AS A SET OF OBJECTS THAT CAN BE USED INDEPENDENTLY	
ACCESS DISTRIBUTED PROCESSES	PROCESSES IN A COMPUTER
SUPPORTS PROCESSES DISTRIBUTED ACROSS MULTI WORK STATIONS, SEAMLESS TO USERS	

ACCS COMMON SOFTWARE PROGRAM

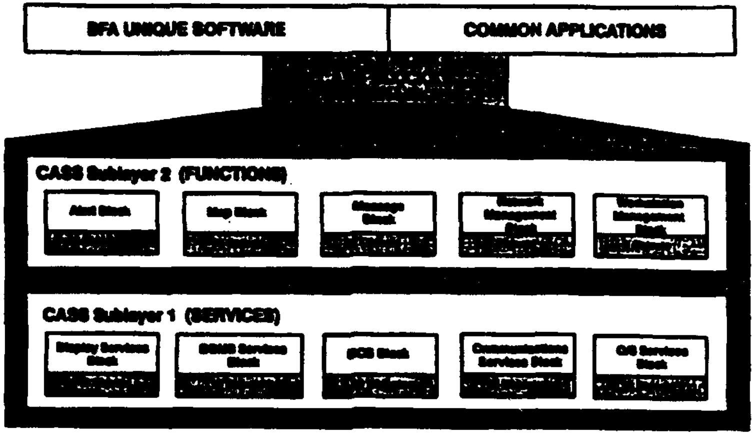
DOCUMENTATION



CASS INTER-SOFTWARE COMMUNICATIONS (ISC) REQUIREMENTS DOCUMENT

- THE ISC REQUIREMENTS DOCUMENT ESTABLISHES A DISTRIBUTED PROCESSING PARADIGM AND ARCHITECTURE FOR CASS
- THE CASS ISC IS ACCESSED BY MULTIPLE ADA PROGRAMS (Potentially on Multiple Processors) VIA ABSTRACT INTERFACES DEFINED FOR:
  - A LOWER-LEVEL DIRECT TRANSPORT INTERFACE (BCS)
  - AN UPPER-LEVEL RPC INTERFACE (Distributed Services)
- DEFINED BY THE CASS ARCHITECTURE WORKING GROUP IN 1989/ 1990
- INCLUDES ADA PACKAGE SPECS THAT DEFINE THE BCS, BASED ON THE AFATDS DESIGN
- REFERENCED AND BASELINED BY THE CASS SSS IN JUNE 1991
- UPCOMING REVISION PLANNED TO AMEND ADA SPECS WITH NEW AFATDS BCS DESIGN

CASS COMPONENT BREAKOUT



BFA UNIQUE SOFTWARE

COMMON APPLICATIONS

CASS Sublayer 2 (FUNCTIONS)

Alert Block

Map Block

Message Block

Network Management Block

Workstation Management Block

CASS Sublayer 1 (SERVICES)

Display Services Block

DBMS Services Block

ISC Block

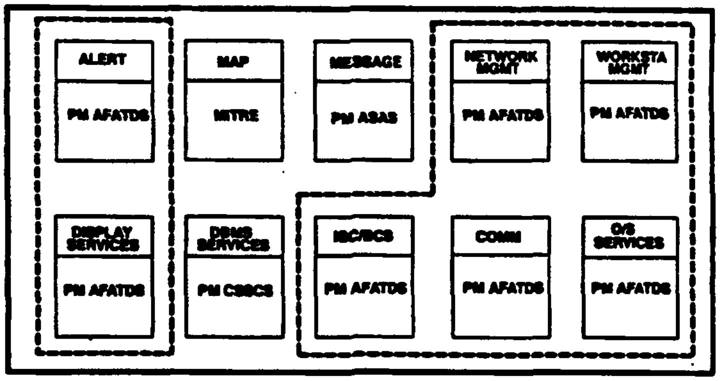
Communications Services Block

O/S Services Block

SUMMARY: 10 Blocks
137 Objects

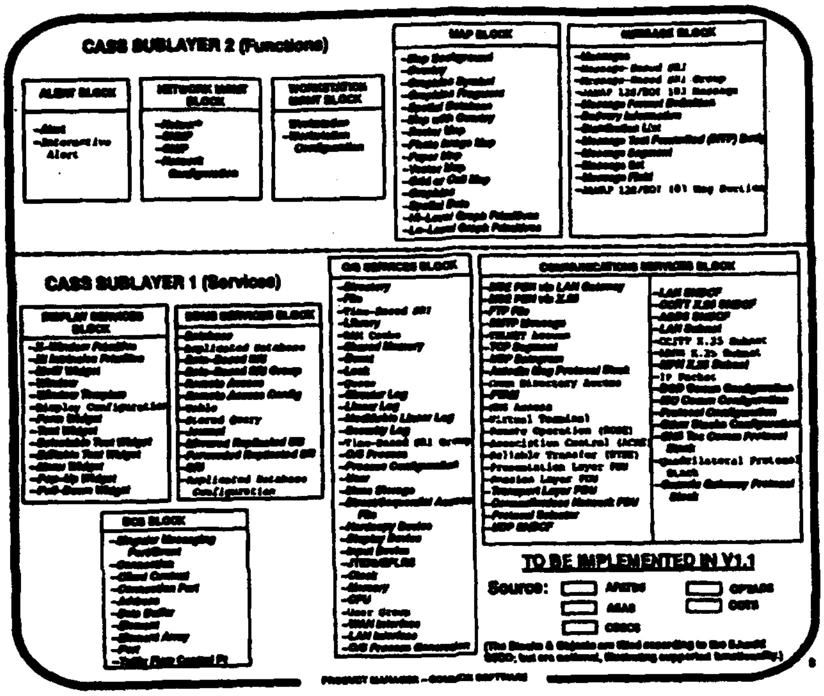
ACCS COMMON SOFTWARE PROGRAM

CASS ARCHITECTURE



ALERT	MAP	MESSAGE	NETWORK MGMT	WORKSTA MGMT
PM AFATDS	MITRE	PM ASAS	PM AFATDS	PM AFATDS
DISPLAY SERVICES	DBMS SERVICES	ISC/BCS	COMM	O/S SERVICES
PM AFATDS	PM CSSCS	PM AFATDS	PM AFATDS	PM AFATDS

INTEGRATED RELEASE FROM THE AFATDS PROGRAM



CASS SUBLAYER 2 (Functions)

ALERT BLOCK
- Alert
- Interactive Alert

NETWORK MGMT BLOCK
- Network
- [illegible]
- [illegible]
- Network Configuration

WORKSTATION MGMT BLOCK
- Workstation
- Workstation Configuration

MAP BLOCK
- Map Background
- Overlay
- Graphics Symbol
- Graphics Fragment
- Spatial Database
- Map with Overlay
- Raster Map
- Photo Image Map
- Paper Map
- Vector Map
- Grid or Cell Map
- Graphics
- Spatial Data
- Hi-Level Graph Primitives
- Lo-Level Graph Primitives

MESSAGE BLOCK
- Messages
- Message-Based SBI
- Message-Based SBI Group
- JANAP 128/DOI 103 Message
- Message Format Definition
- Delivery Information
- Distribution List
- Message Text Formatted (MTF) Body
- Message Segment
- Message Set
- Message Field
- JANAP 128/DOI 103 Msg Section

CASS SUBLAYER 1 (Services)

DISPLAY SERVICES BLOCK
- X-Window Primitive
- Xt Intrinsics Primitive
- Shell Widget
- Window
- Window Template
- Display Configuration
- Form Widget
- Text Widget
- Selectable Text Widget
- Editable Text Widget
- Menu Widget
- Pop-Up Widget
- Pull-Down Widget

DBMS SERVICES BLOCK
- Database
- Replicated Database
- Data-Based SBI
- Data-Based SBI Group
- Remote Access
- Remote Access Config
- Table
- Stored Query
- Journal
- Advanced Replicated DB
- Personalized Replicated DB
- [illegible]
- Replicated Database Configuration

SCS BLOCK
- Regular Messaging Participant
- Connection
- Client Control
- Connection Port
- Address
- Data Buffer
- Channel
- Channel Array
- Port
- Traffic Flow Control Pt

OS SERVICES BLOCK
- Directory
- File
- Time-Based SBI
- Library
- SBI Cache
- Shared Memory
- Event
- Lock
- Queue
- Circular Log
- Linear Log
- Modifiable Linear Log
- Security Log
- Time-Based SBI Group
- OS Process
- Process Configuration
- User
- Mass Storage
- Shared/Sequential Access File
- Hardcopy Device
- Display Device
- Input Device
- [illegible]
- Clock
- Memory
- CPU
- User Group
- WAN Interface
- LAN Interface
- OS Process Generator

COMMUNICATIONS SERVICES BLOCK
- MSE PDU via LAN Gateway
- MSE PDU via X.25
- FTP File
- SMTP Message
- TELNET Session
- TCP Segment
- UDP Datagram
- [illegible] Msg Protocol Stack
- Open Directory Service
- FTAM
- OSI Access
- Virtual Terminal
- Remote Operation (ROSE)
- Association Control (ACSE)
- Reliable Transfer (RTSE)
- Presentation Layer PDU
- Session Layer PDU
- Transport Layer PDU
- Connectionless Network PDU
- Protocol Selector
- UDP SNDCF
- LAN SNDCF
- CCITT X.25 SNDCF
- ARPA SNDCF
- LAN Subnet
- CCITT X.25 Subnet
- MSE X.25 Subnet
- DDN X.25 Subnet
- IP Packet
- DoD Comm Configuration
- OSI Comm Configuration
- Protocol Configuration
- Other Stacks Configuration
- [illegible] Comms Protocol Stack
- Quadrilateral Protocol Stack
- Gateway Protocol Stack

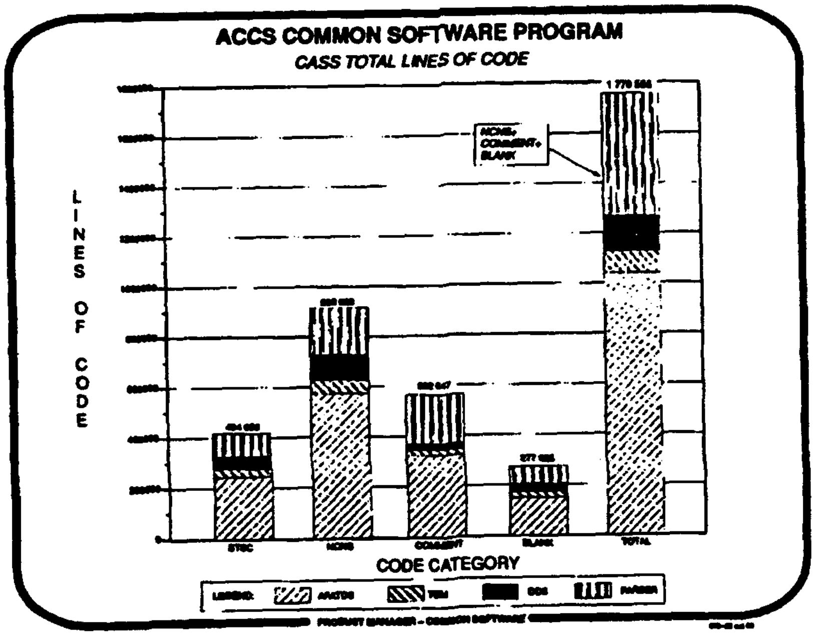
TO BE IMPLEMENTED IN V1.1

Source: □ AFATDS □ ASAS □ CSSCS □ OPFACS □ CGTS

(The Blocks & Objects are filed according to the [illegible] SDD, but are notional, illustrating supported functionality.)

8

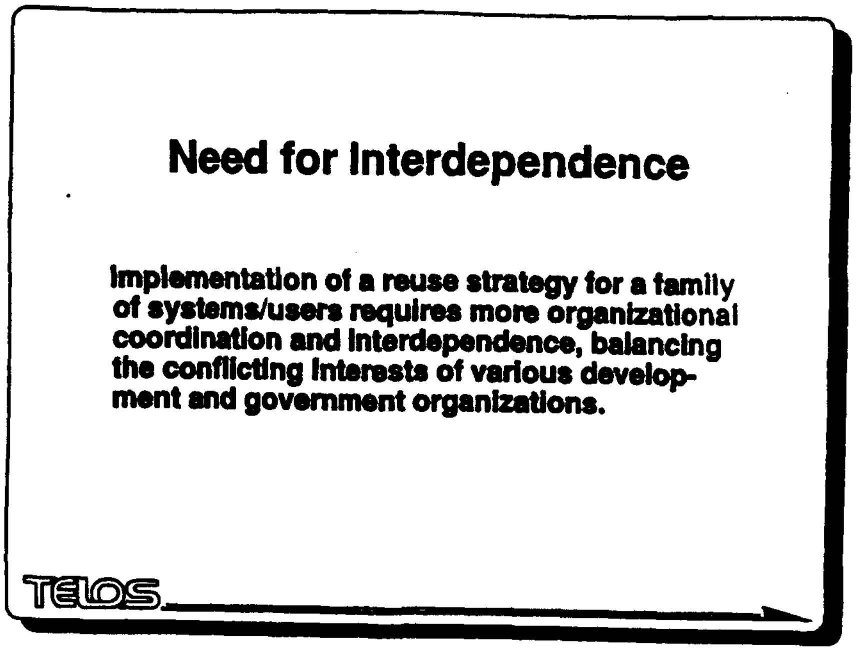
PRODUCT MANAGER – COMMON SOFTWARE



ACCS COMMON SOFTWARE PROGRAM

CASS TOTAL LINES OF CODE

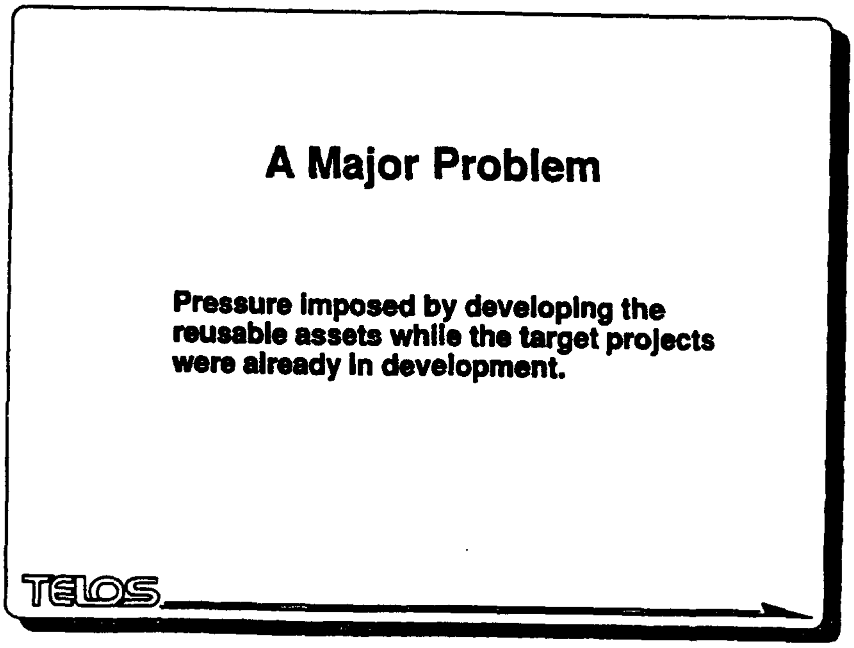
PRODUCT MANAGER – COMMON SOFTWARE



Need for Interdependence

Implementation of a reuse strategy for a family of systems/users requires more organizational coordination and interdependence, balancing the conflicting interests of various development and government organizations.

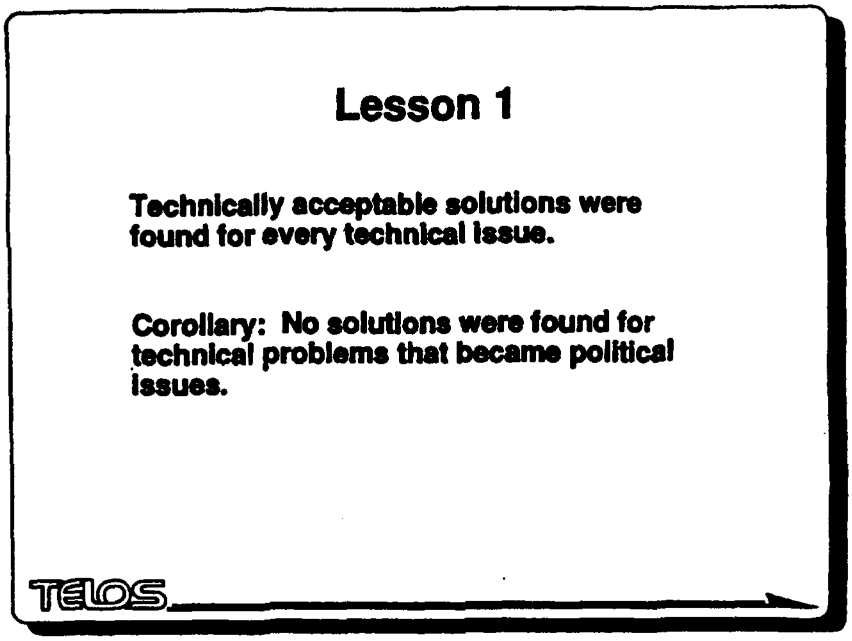
TELOS



A Major Problem

Pressure imposed by developing the reusable assets while the target projects were already in development.

TELOS

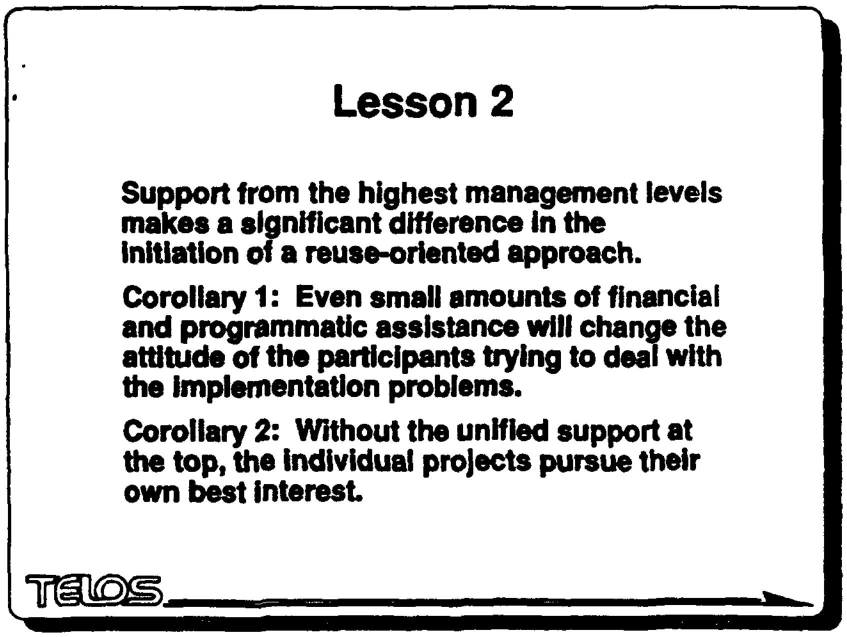


Lesson 1

Technically acceptable solutions were found for every technical issue.

Corollary: No solutions were found for technical problems that became political issues.

TELOS



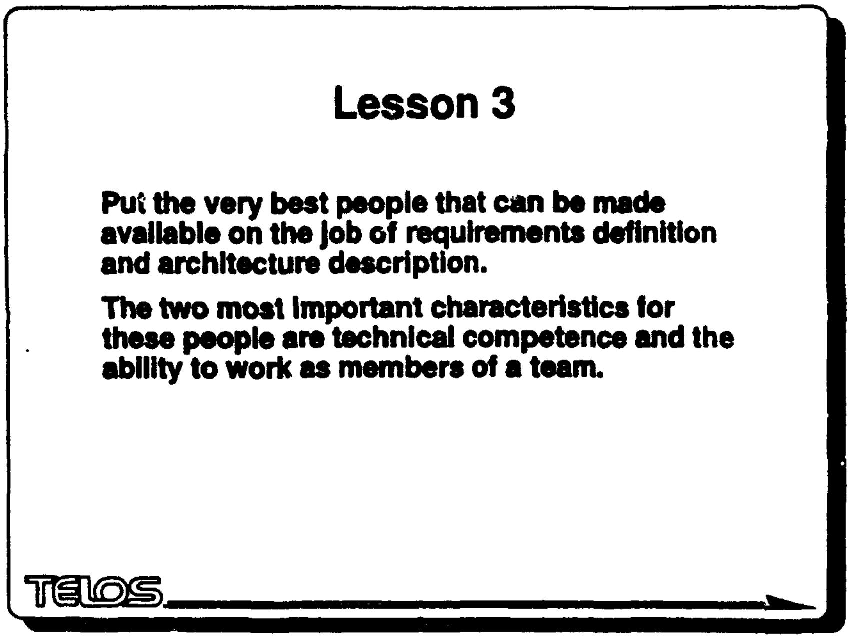
Lesson 2

Support from the highest management levels makes a significant difference in the initiation of a reuse-oriented approach.

Corollary 1: Even small amounts of financial and programmatic assistance will change the attitude of the participants trying to deal with the implementation problems.

Corollary 2: Without the unified support at the top, the individual projects pursue their own best interest.

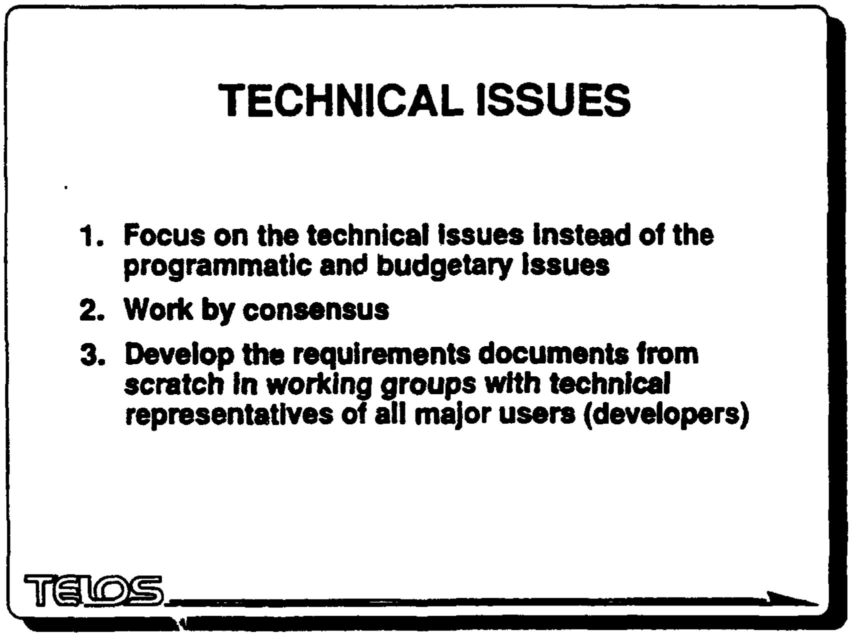
TELOS



Lesson 3

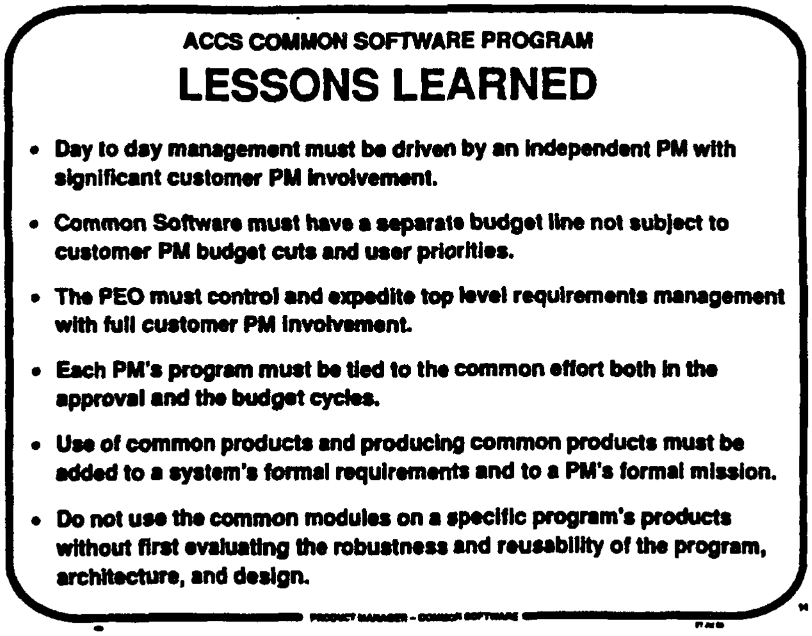
Put the very best people that can be made available on the job of requirements definition and architecture description.

The two most important characteristics for these people are technical competence and the ability to work as members of a team.



TECHNICAL ISSUES

1. Focus on the technical issues instead of the programmatic and budgetary issues
2. Work by consensus
3. Develop the requirements documents from scratch in working groups with technical representatives of all major users (developers)



ACCS COMMON SOFTWARE PROGRAM

LESSONS LEARNED

- Day to day management must be driven by an independent PM with significant customer PM involvement.
- Common Software must have a separate budget line not subject to customer PM budget cuts and user priorities.
- The PEO must control and expedite top level requirements management with full customer PM involvement.
- Each PM's program must be tied to the common effort both in the approval and the budget cycles.
- Use of common products and producing common products must be added to a system's formal requirements and to a PM's formal mission.
- Do not use the common modules on a specific program's products without first evaluating the robustness and reusability of the program, architecture, and design.

PRODUCT MANAGER - COMMON SOFTWARE

STRUCTURAL MODELS IN PROPOSALS

TRAINING SYSTEM PROGRAM OFFICE

FREDERICK J. SWARTZ
WRIGHT PATTERSON AFB OH

TRAINING SYSTEM PROGRAM OFFICE

The Training System Program Office is a wing-level organization singularly responsible for the planning, contracting, designing, testing, and delivery of sophisticated, multi-million dollar training aircraft and aircrew/maintenance training devices and systems to USAF frontline troops. Products enable USAF aircrews and maintainers to train like they fight.

OUTLINE

History of Structural Models

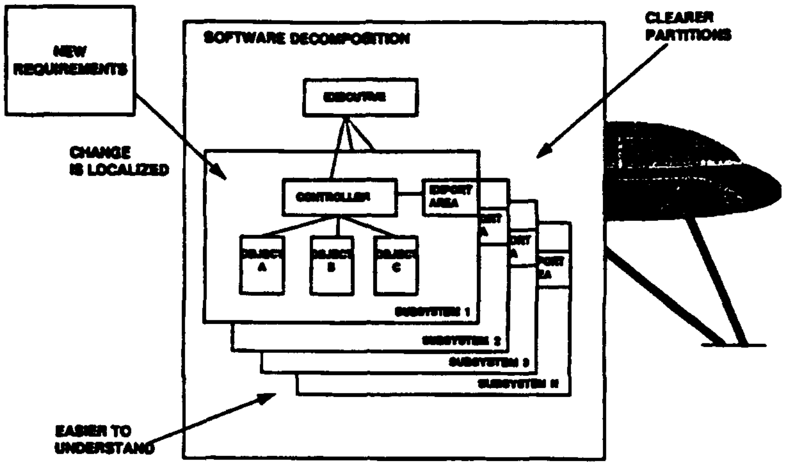
Overview of Structural Models

Use of Structural Models in RFPs

Structural Modeling

- A Structural Model provides a high level design
  - structure: classes of containers for functionality
  - coordination: captures coordination model which specifies communications, synchronization and time management
- Ability of the architecture to leverage development through structure
- Reusable software architecture - a high level embodiment of design decisions

STRUCTURAL MODELING



Structural Modeling Addresses

- Development Cost
  - simplifies and standardizes design
  - provides ability to make decisions early in process
  - minimizes assumptions built into designs
  - promotes reusability (architecture, design, implementations)
- Integration
  - clear picture of how system is constituted
  - early integration harness provides complete model of system
  - allows substitution of real parts for models in incremental fashion
  - reduced integration time, fewer surprises

Structural Modeling Addresses

- Maintenance Cost
  - "robust" under modification
  - more easily understood by maintainers
  - predictability in cost and performance
  - well defined expectations of structure, composition, and coordination
- Aircraft Currency
  - close mapping to aircraft design
  - well defined interfaces to avionics components
  - tolerance for data voids

STRUCTURAL MODELS IN PROPOSALS

- Instructions to Offerer (ITO)
  - Describe the structural Model(s)
  - Demonstrate model(s) is complete
  - Describe how model(s) will be applied
- Statement-of-Work (SOW)
  - Use object oriented methods
  - Ada structural modeling
  - SSR -- architectural guidelines
  - PDR -- incremental
  - CDR -- incremental

STRUCTURAL MODELS IN PROPOSALS

- System Requirements Documents
  - Modularity
  - Maintainability
  - P3I

STRUCTURAL MODELS IN PROPOSALS

- What else
  - New reviews
    - * Pre SRR -- Architecture Guidelines & SDP
    - * Pre PDR -- Structure Model Review I
    - * Pre CDR -- Structure Model Review II
- Guidebook
  - SEI produced
  - Part of bidder library
- White Paper on Structural Modeling

STRUCTURAL MODELS IN PROPOSALS SUMMARY

- Structural model is still maturing
- Based on Object Oriented methodology
- Very little specifics in ITO, SOW, and SRD
- Evaluating approach based on:
  - Risk
  - Performance
  - ilities
- Guidebook will give the basics

Central Archive for Reusable Defense Software (CARDS)

Software Architecture Workshop

16 November 1993

Architecture Forum Workshop - 17 November

Purpose:

- **Explore the current practice of software architectures and software re-use on actual projects**
- **Explore current research into architecture as a means of implementing reuse**

Overview:

- **Morning:**
  - **Short presentations by practitioners and researchers on their current work with architectures**
- **Afternoon:**
  - **Working session to identify common problems in reuse implementation and develop a common approach to solutions**

Workshop Schedule 17 November

8:00 AM	Transitioning from research to practice - T. Saunders, Mitre
8:30 AM	Architecture as the framework for realizing the benefits of reuse - W. Tracz, IBM
8:45 AM	Abstraction and layering within software architectures - M. Gerhard, ESL
9:00 AM	Overview of DISA Software Reuse Domain Analysis - D. Gary, DISA
9:15 AM	Software Architecture, Reuse, and Maintenance - Jim Baldo, Unisys
9:30 AM	Break
9:45 AM	The Object-Connection-Update Architecture - Charles Plinta, ACCEL

Workshop Schedule 17 November - Continued

10:00 AM	PRISM software architecture - P. Valdez, ESC/ENS
10:15 AM	NSA Unified INFOSEC Architecture (UIA) - B. Koehler, DIRNSA
10:30 AM	9LV Mk3 shipboard C2 architecture - U. Olsson, CelsiusTech Systems
10:45 AM	Architectures and the real world, based on the Army C2 common software program experience - S. Levine, Army
11:00 AM	Break
11:15 AM	Architectures in the CIS field - applying Christopher Alexander's work - J. Bonine, Design Metrics Technology
11:30 AM	OO-based architecture use at NUWC - S. Roodbeen, NUWC
11:45 AM	Capturing domain knowledge at NTF - T. Gill, NFT/ENS

Workshop Schedule 17 November - Continued

12:00 PM	STARS demo project architecture - G. Wickman, CECOM
12:15 PM	The STARS Air Force Demo Project - K. Spicer, SWSC/SMX
12:30 PM	Lunch - 4th Floor Antechamber
1:30 PM	Working Groups
4:30 PM	Working Group Report
5:00 PM	Wrap-up

Proposed Working Groups and Topics - 17 November

- WG 1: Evaluation and Measurement of Architectures
  - procurement issues: how can many proposed architectures be evaluated?
  - design issues: what are the "architecture-level" qualities which can and should be measured?
- WG 2: Software Architecture Technologies
  - what are the current and emerging technologies for software architecture?
  - where is the "low hanging fruit" (i.e., easily attained but useful technology)?
- WG 3: Software Architecture and Reuse
  - what does it mean for an architecture to be "reusable?"
  - what is needed for product-line architectures to sustain a commercial component provider industry?
- WG 4: Software Architecture and Standards
  - what is the relationship between architecture and open systems?
  - what are areas of architecture standardization, e.g., "building codes?"
- WG 5: Software Architecture and Strategic (Product-Line) Planning
  - where in the DoD should architectures be specified? maintained? implemented? What are the pros/cons of various approaches?
  - how can DoD architectures, if specified, be used prescriptively in procuring systems

DOMAIN-SPECIFIC SOFTWARE ARCHITECTURES AND SOFTWARE REUSE

Will Tracz
IBM Federal Systems Company
MD 0210
Owego, NY 13827-1298
TRACZ@VNET.IBM.COM

IBM

November 17, 1993

Outline

- What is DSSA
  - Definition
  - Goals and technical approach
  - Team members
- Observations on Software Architectures
  - The "Golden Gun/Silver Bullet" analogy
  - Low-hanging fruit
  - Dainty morsels just out of reach
  - The star on top of the tree

"Original" Definition of DSSA

An assemblage of software components[1],

- specialized for a particular type of task (domain),

- generalized for effective use across that domain,

- composed in a standardized structure (topology) effective for building successful applications.

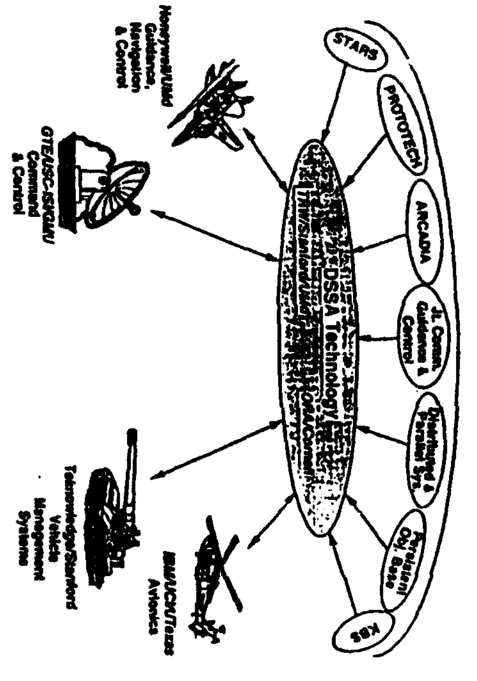
"Current" Definition of DSSA

- a domain model (several views)
- a reference (parameterised) architecture (expressed in an ADL),
- its supporting infrastructure/environment, and
- a process/methodology to instantiate/refine and evaluate it.

[1] Lee Erman, Teknowledge/Cimflex

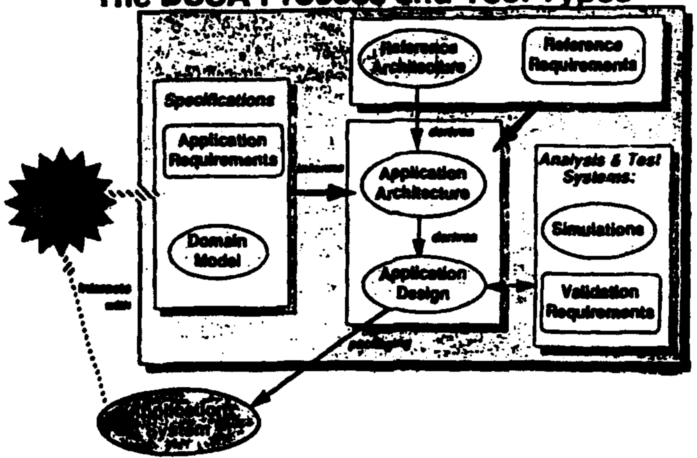
Will Tracz

DSSA Arena


STARS
PROTOTECH
ARCADIA
Jt. Comm. Guidance & Control
Distributed & Parallel Sys.
Persistent Obj. Base
KBS
DSSA Technology
TRW/Stanford/UMd/[illegible]/ORA/Cornell
Honeywell/UMd Guidance, Navigation & Control
GTE/USC-ISI/GMU Command & Control
Teknowledge/Stanford Vehicle Management Systems
IBM/UCI/UTexas Avionics

Will Tracz

The DSSA Process and Tool Types



Primary Tool Types: Modeling; Requirements Management; Architecture Specification, Refinement, & Evolution

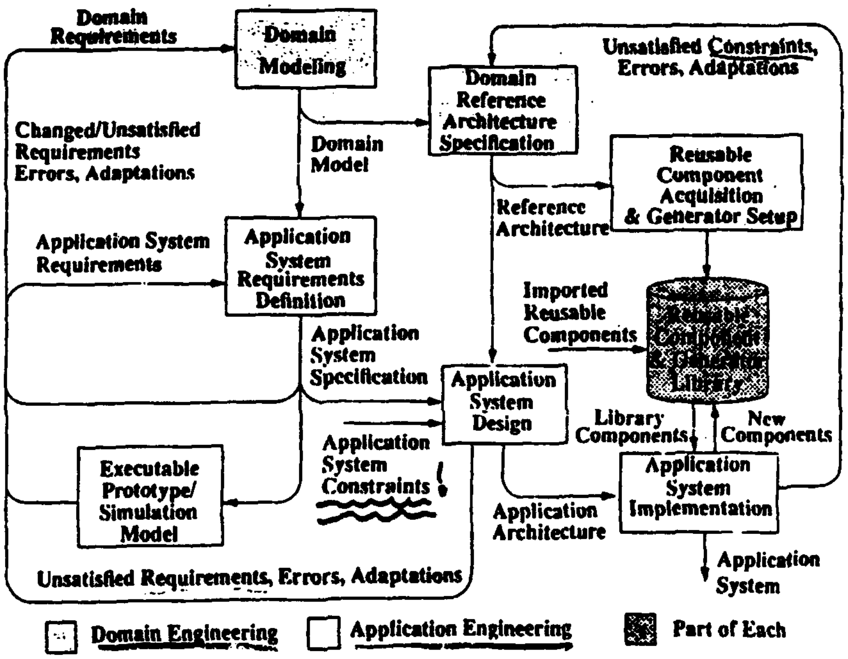
Secondary: Repository; Component Selection; Component Generation; Requirements Validation; Configuration Package, Load, & Exercise; Performance Evaluation

TFS/Cimflex Teknowledge

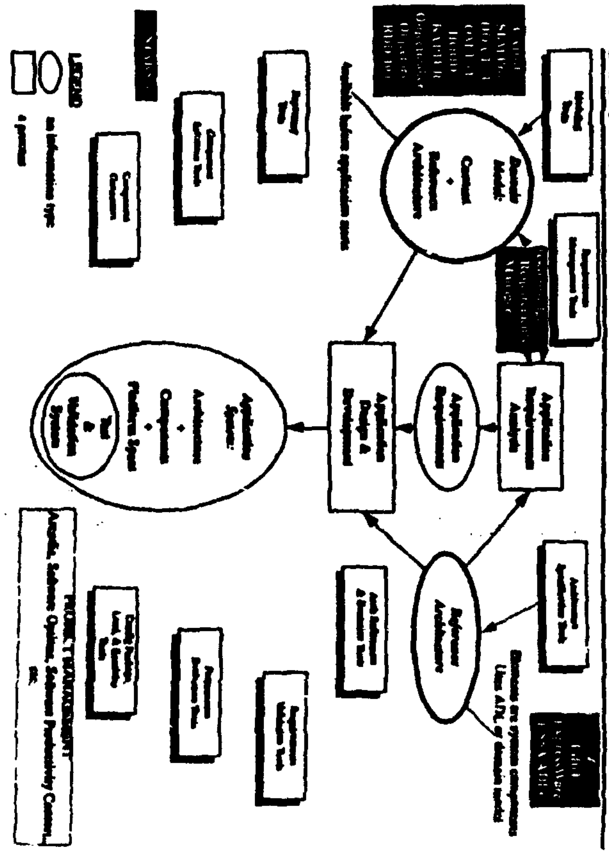
PROBLEM DOMAIN

DSSA Lifecycle

SOLUTION SPACE


Domain Requirements
Domain Modeling
Domain Model
Domain Reference Architecture Specification
Unsatisfied Constraints, Errors, Adaptations
Changed/Unsatisfied Requirements Errors, Adaptations
Reusable Component Acquisition & Generator Setup
Reference Architecture
Application System Requirements
Application System Requirements Definition
Imported Reusable Components
Reusable Component & Generator Library
Application System Specification
Application System Design
Library Components
New Components
Application System Constraints
Executable Prototype/ Simulation Model
Application System Implementation
Application Architecture
Application System
Unsatisfied Requirements, Errors, Adaptations
Domain Engineering
Application Engineering
Part of Each

DSSA - Tool Sources


Modeling Tools
Requirements Management Tools
KAPTUR
Organization Objects
RDD-100
Requirements Manager
Domain Model:
Context
+
Reference Architecture
Available before application starts
Application Requirements Analysis
Application Requirements
Application Design & Development
Architecture Specification Tools
ExpressWare
Elements are system components
Uses ADL or domain model
Reference Architecture
Arch Refinement & Evaluation Tools
Repository Tools
Requirements Validation Tools
Component Selection Tools
Performance Evaluation Tools
STARS
Component Generators
Config, Package, Load, & Execution Tools
Application System:
Architecture
+
Components
+
Platform Specs
Test
&
Validation
System
LEGEND
an information type
a process
PROJECT MANAGEMENT
Arcadia, Software Options, Software Productivity Consort.,
etc.

8 Sample Tool Usage

Note: This section is a rough cut and is subject to change based on future discussion. A more detailed list of available tool technology was compiled as part of the original Infrastructure Tiger Team effort.

Tool	IBM	Teknowledge	Honeywell	Other
Modeling Tools	RDD-100, OMTool, Circuit Compiler	Text Editor, Edge	HoneyX, HI Scheduling, ASSIST, SURE	MatLab, MatrixX, PERTS, CARE, HARP
Requirements Management Tools	RDD-100, Circuit Compiler	ProductTrack		Demo
Architecture Specification Tools	h, LILEANNA, VAG/AKRL	DADSE Architecture Tool	MetaH, ControlH, MCL	Rapide
Architecture Specialization and Application Evaluation Tools	GLUE, Design Record, Circuit Compiler	DADSE Architecture Tool	Demo, epoch	
Repository Tools	InQuisiX	ASCII File/editor, ?ReUse or RLF?		Dome, Columbus
Component Selection Tools	InQuisix, GLUE	DADSE Repository Browser,DADSE Query Browser		
Component Generators	CHGEN, MEGEN, LILEANNA	?Autocode?	ControlH, MPL, ISI Autocode	
Configuration, Package, Load and Execute Tools	GLUE	?QAD/Polylith?	MetaH, MCL, Adacope960	
Performance Evaluation Tools	Circuit Compiler		APH	MatLab, MatrixX, Perts, Scheduler 1-2-3
Requirements Validation Tools	RDD-100, Circuit Compiler	ProductTrack	ASID/VS	
Language Processing and Compiling	YACC/LEX	APLEX/AYACC		
User Interfaces	Chiron, Chimera, Interface Workbench, TC/TKL	ABE Windows	Demo	
Databases		Oracle		Ingress, Cobalos
Distributed and Real-time Computing	RNA Tool	?PERTS?	MetaH, Marcel	VxWorks, RT-Mach
Documentation	Frame, LaTeX			
Configuration Management	?CMVC/6000?	?CVS, Shape?	CVS	
Other	Amadeus, TrueWare			

Table 1: DSSA Tool Usage and Examples

LILEANNA / ADAGE Relationship

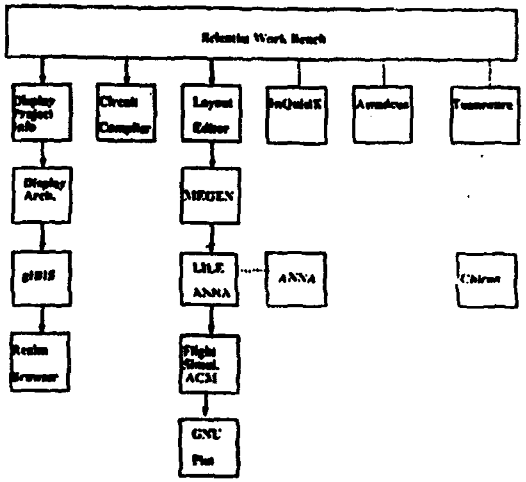
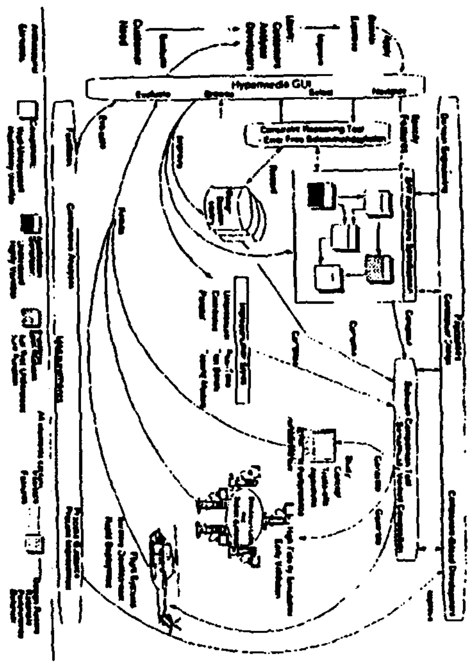
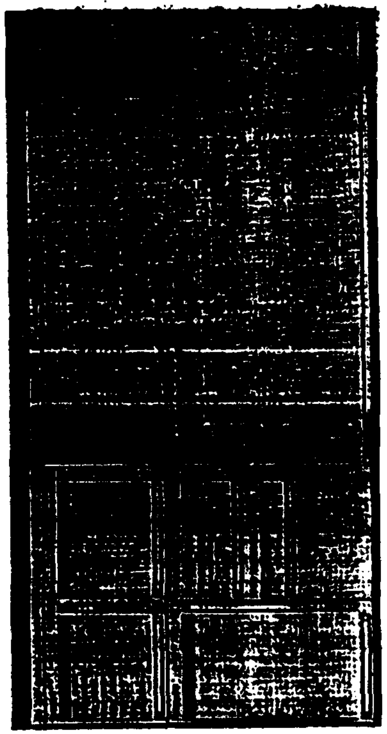


Figure 4: The DSSA-ADAGE Layout Editor and Module Expression Generator (MEGEN) provide inputs to LILEANNA based on user parameters.

Iterative Development Strategy





Golden Gun/Silver Bullet Analogy

- One's not good without the other
- The Price is Right
- Common Sense Prevails
- Do it anyway for complex systems
- Do it anyway for product lines
- Requires discipline and investment
  Catch-22

Will Tracz 2

Low-Hanging Fruit

- Architecture styles – POINT OF ANALYSIS
- Record rationale
- Software Bus – HP UPAS PURTILLO
- Alexander's Wisdom/
- Domain Modeling Processes
- Domain Engineering Processes

Dainty Morsels Just Out of Reach

- Architecture Description Languages
- DSSA infrastructure support
- More success stories
-
-
-

The Star on Top of the Tree

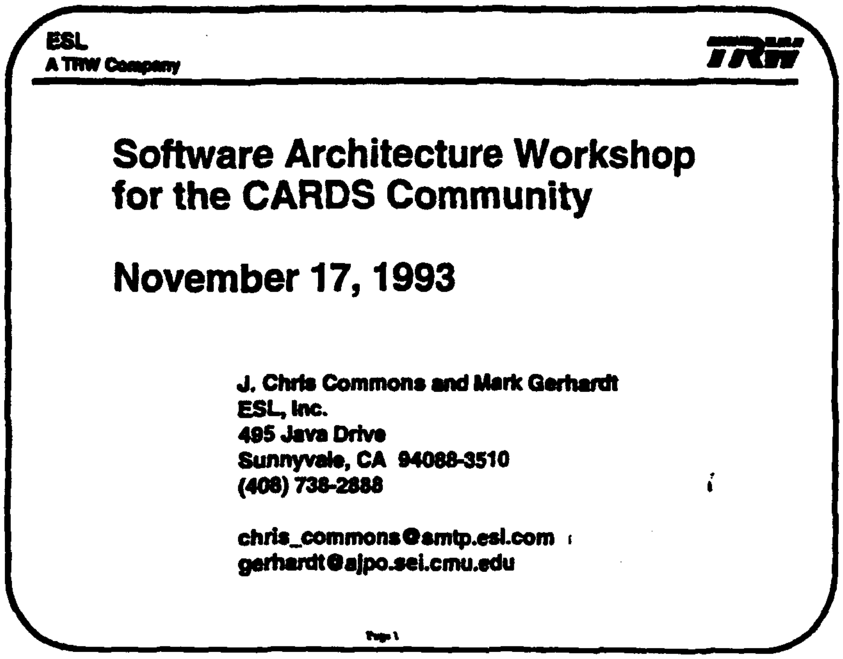
- Integrated environment
- Truly Heterogeneous Software Buses

•

•

•

•



ESL
A TRW Company

TRW

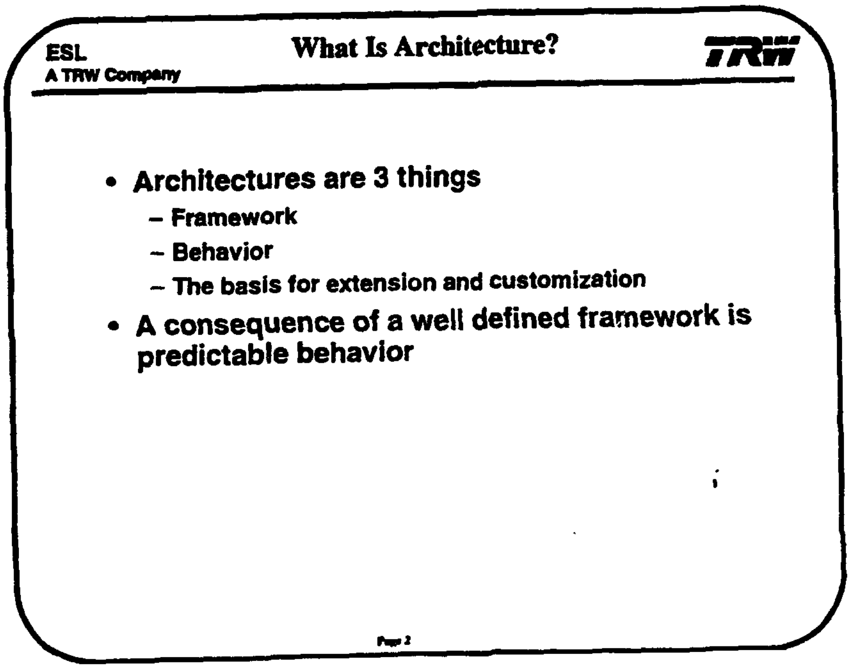
Software Architecture Workshop for the CARDS Community

November 17, 1993

J. Chris Commons and Mark Gerhardt
ESL, Inc.
495 Java Drive
Sunnyvale, CA 94088-3510
(408) 738-2888

chris_commons@smtp.esl.com
gerhardt@ajpo.sei.cmu.edu

Page 1



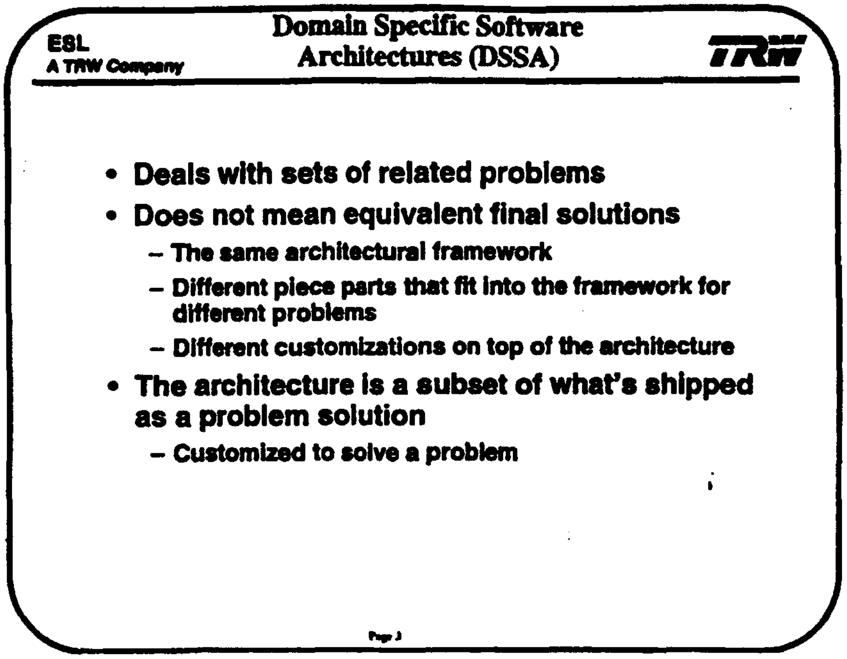
ESL
A TRW Company

What Is Architecture?

TRW

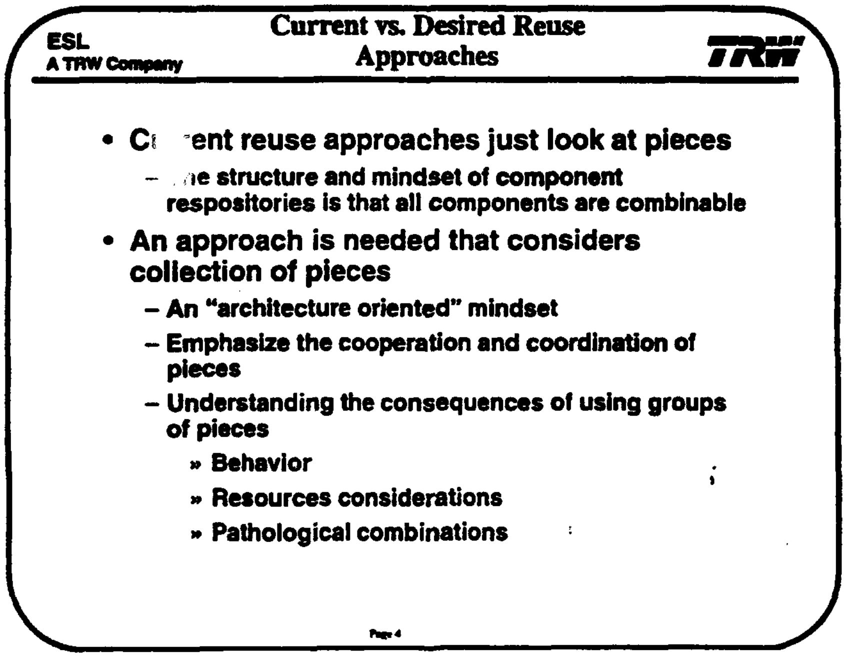
- Architectures are 3 things
  - Framework
  - Behavior
  - The basis for extension and customization
- A consequence of a well defined framework is predictable behavior

Page 2



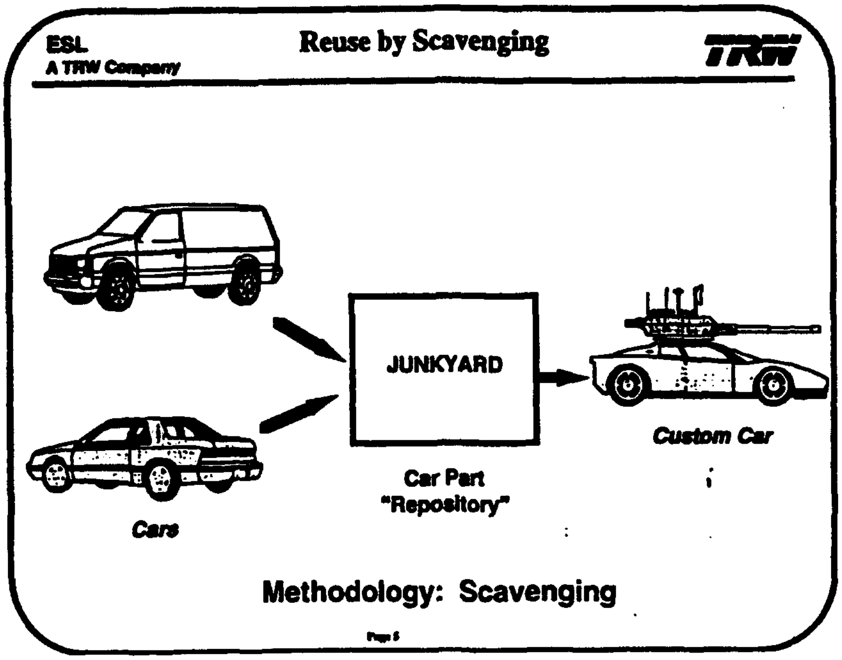
Domain Specific Software Architectures (DSSA)

- Deals with sets of related problems
- Does not mean equivalent final solutions
  - The same architectural framework
  - Different piece parts that fit into the framework for different problems
  - Different customizations on top of the architecture
- The architecture is a subset of what's shipped as a problem solution
  - Customized to solve a problem



Current vs. Desired Reuse Approaches

- Current reuse approaches just look at pieces
  - The structure and mindset of component respositories is that all components are combinable
- An approach is needed that considers collection of pieces
  - An "architecture oriented" mindset
  - Emphasize the cooperation and coordination of pieces
  - Understanding the consequences of using groups of pieces
    - » Behavior
    - » Resources considerations
    - » Pathological combinations



ESL
A TRW Company

Reuse by Scavenging

JUNKYARD

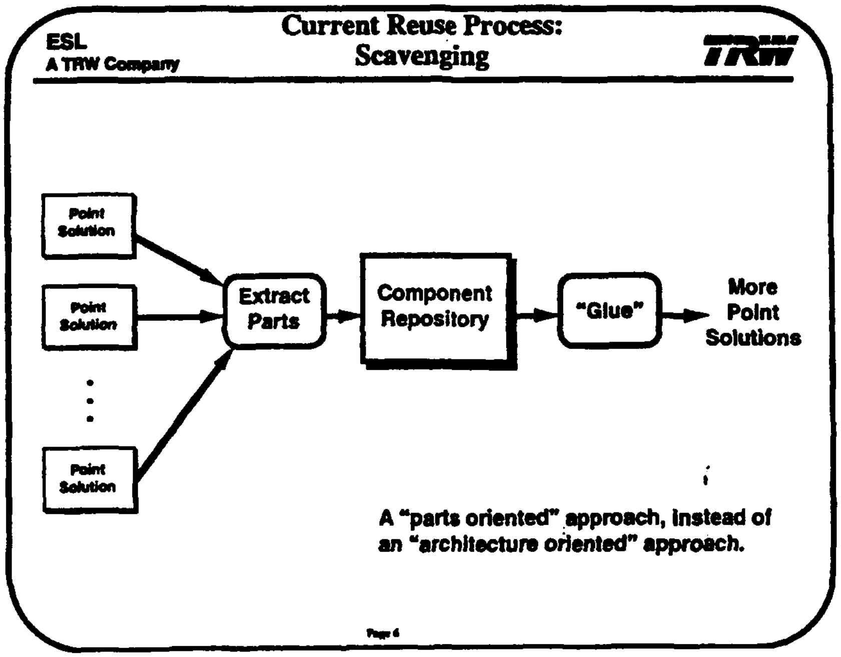
Car Part
"Repository"

Custom Car

Cars

Methodology: Scavenging

Page 5



ESL
A TRW Company

Current Reuse Process: Scavenging

Point Solution

Point Solution

Point Solution

Extract Parts

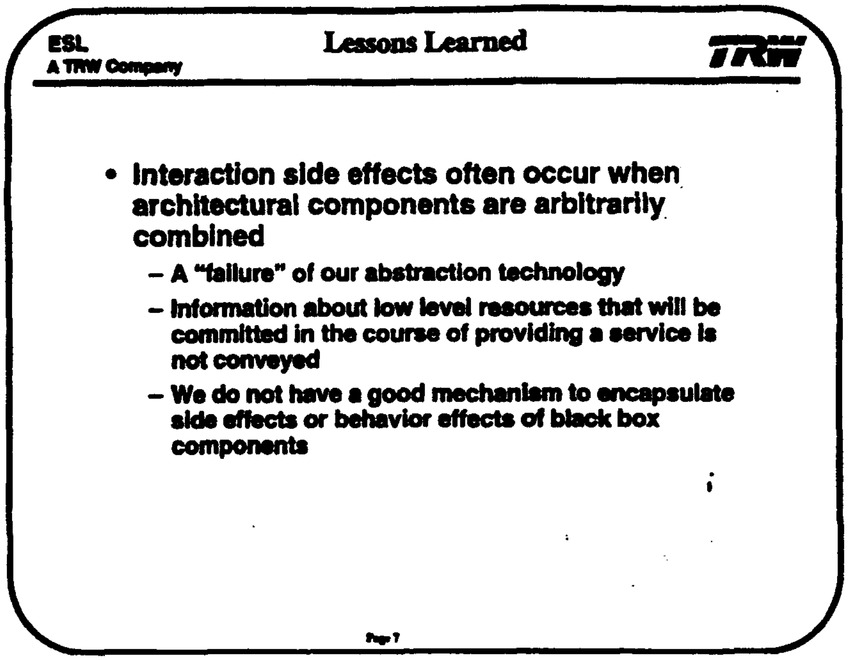
Component Repository

"Glue"

More Point Solutions

A "parts oriented" approach, instead of an "architecture oriented" approach.

Page 6

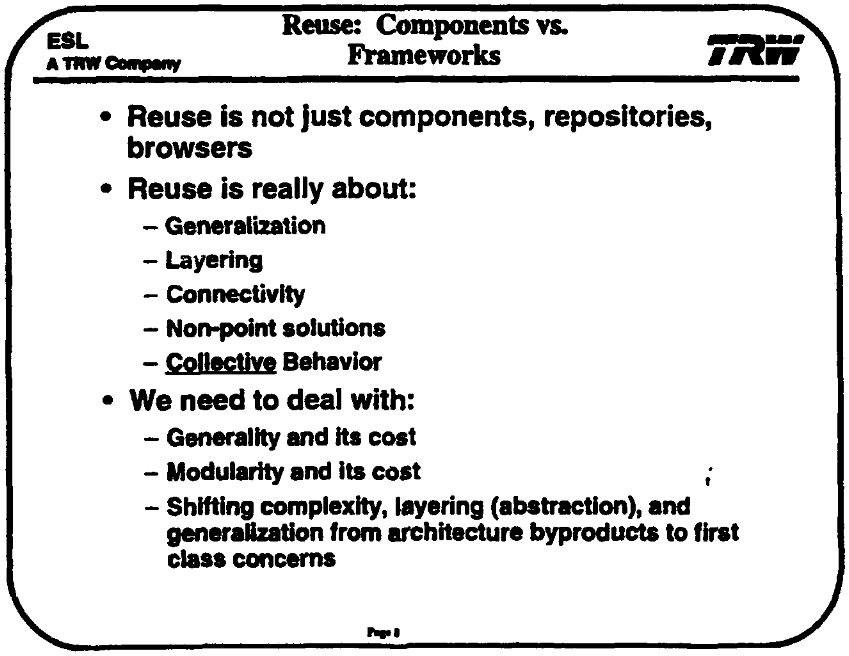


ESL
A TRW Company

Lessons Learned

- Interaction side effects often occur when architectural components are arbitrarily combined
  - A "failure" of our abstraction technology
  - Information about low level resources that will be committed in the course of providing a service is not conveyed
  - We do not have a good mechanism to encapsulate side effects or behavior effects of black box components

Page 7



ESL
A TRW Company

Reuse: Components vs. Frameworks

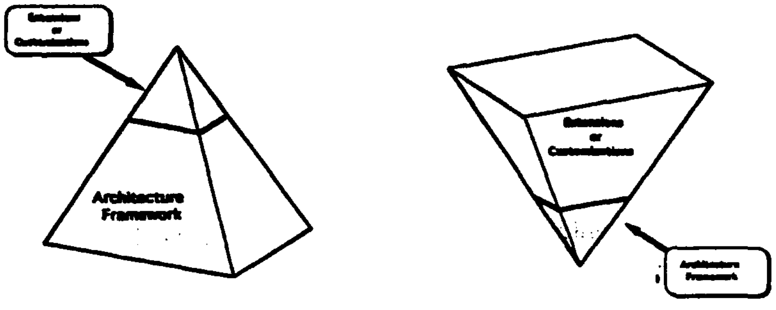
- Reuse is not just components, repositories, browsers
- Reuse is really about:
  - Generalization
  - Layering
  - Connectivity
  - Non-point solutions
  - Collective Behavior
- We need to deal with:
  - Generality and its cost
  - Modularity and its cost
  - Shifting complexity, layering (abstraction), and generalization from architecture byproducts to first class concerns

Page 8

ESL
A TRW Company

Architectures and Domains

TRW



Within The Domain

Outside The Domain

Page 9

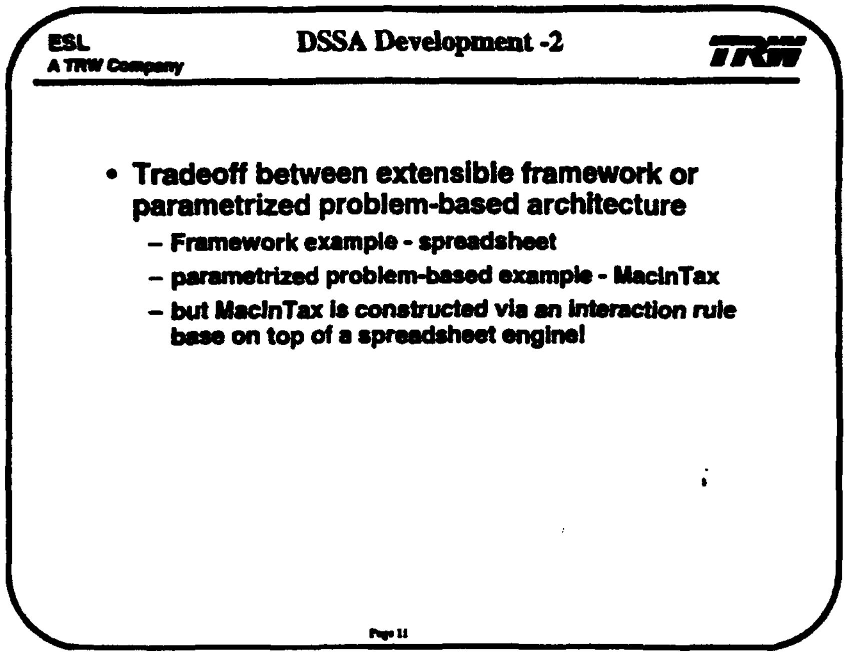
ESL
A TRW Company

DSSA Development

TRW

- A generalized approach for developing DSSA is difficult:
  - Generality can only be obtained from collections of specifics.
  - Bottom up approach
  - Factoring of commonality
    - » Recurring functionality (Common modules)
    - » Framework or infrastructure uniformity

Page 10



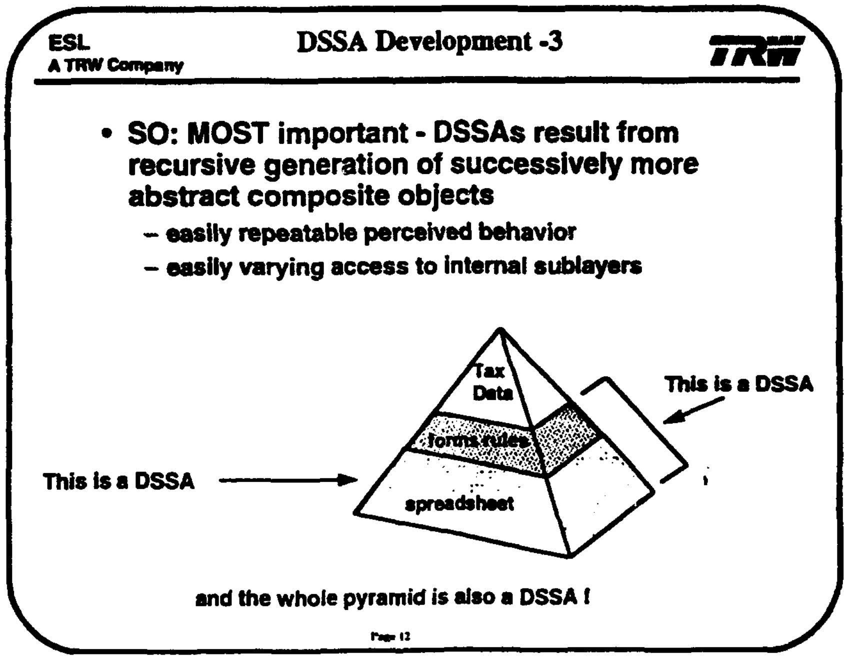
ESL
A TRW Company

DSSA Development -2

TRW

- Tradeoff between extensible framework or parametrized problem-based architecture
  - Framework example - spreadsheet
  - parametrized problem-based example - MacInTax
  - but MacInTax is constructed via an interaction rule base on top of a spreadsheet engine!

Page 11



ESL
A TRW Company

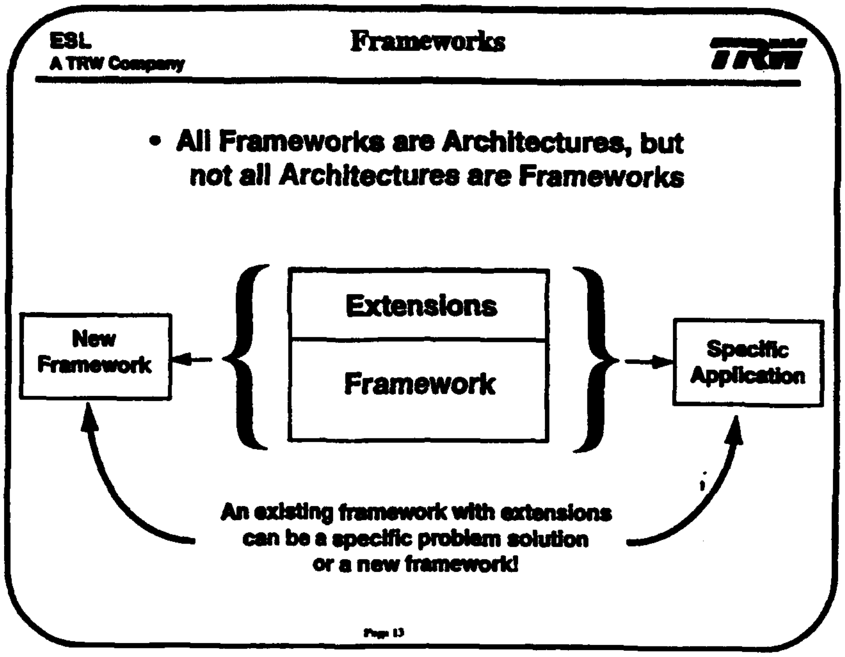
DSSA Development -3

TRW

- SO: MOST important - DSSAs result from recursive generation of successively more abstract composite objects
  - easily repeatable perceived behavior
  - easily varying access to internal sublayers

and the whole pyramid is also a DSSA !

Page 12



ESL
A TRW Company

Frameworks

TRW

- All Frameworks are Architectures, but not all Architectures are Frameworks

Extensions

Framework

New Framework

Specific Application

An existing framework with extensions can be a specific problem solution or a new framework!

Page 13

Domain Engineering

Deborah Gary (DISA/TXED)
(703) 536-[illegible]

Overall Concept

- Domain Engineering is the systematic identification of commonalities among a group of related software systems
- Domain Engineering is composed of three major parts:
  - Domain Analysis
  - Domain Design
  - Domain Implementation

Domain Engineering — The Products... Domain Model

- Object Oriented Domain Model
- Identifies Common Software Objects And Requirements For A Family Of Systems

Components:
- Domain Requirements Diagram
- Object/Class Specifications

Products of Domain Design

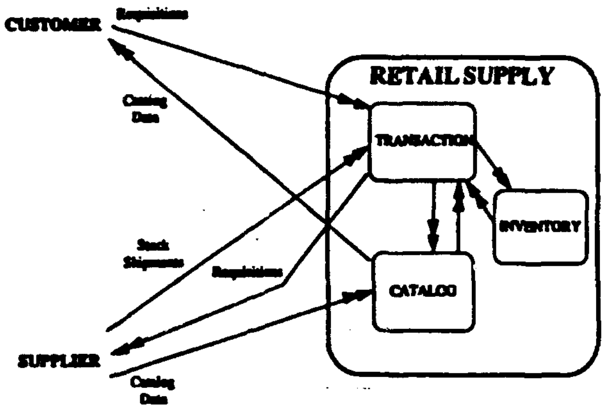
➤ Domain Specific Software Architecture (DSSA)

A specification for assemblage of software components that is:
- Specialized for a particular class of tasks (domain),
- Generalized for effective use across that domain,
- Composed in a standardized structure (topology),
- Effective for building successful applications.
- Minimally provides a framework for specifying the major components and the interfaces that satisfy the requirements. *[DARPA]*

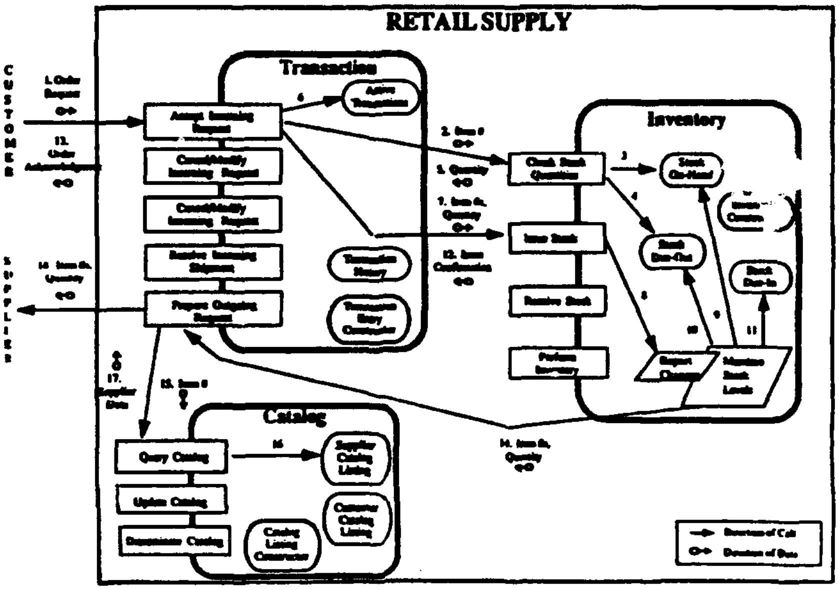
Components:
- Graphical Diagram
- Class/Object Design Specifications

➤ Domain Design Classification Terms

High-Level DSSA Diagram



DSSA - Execution Thread of an "Order Request"



Class/Object Design Specification - *Template*

Class/Object Name: *<text>*

a. This name should reflect the use of data standardization. If an existing standardized name is in use, it should be used here.

b. (If imported from another domain, prefix the name with the domain name, e.g., MIS:Database_Interface_Binding)>

***Required/Optional:** *<text>*

Description: *<text>*

Source(s): *<text (systems/prototypes employing this design)>*

Adaptation Requirements (Variants): *<text e.g.,generic_parameters)>*

* **Reuse Guidance:** *<text>*

* **Lessons Learned:** *<text>*

Page 1 of 4

Constraints:

- * **Directives/Standards:** *<text>*
- * **Software:** *<text (from SW/HW constraints)>*
- * **Hardware:** *<text (from SW/HW constraints)>*
- * **Memory Size Allocation:** *<text>*

Concurrency: *<object_name(s)>*

Structure:

Whole: *<aggregate_object_name(s)>*
--If this class/object has parts

Part-Of: *<object_name>*
--If this class/object is part of a larger object.

Generalization-of: *<class/object_name>*

Specialization-of: *<class_name>*

Page 2 of 4

* additional specs not in the requirement specs

Class/Object Design Specification - *Template (cont)*

Connection:

Instance: *<class/object_name with cardinality>*

Message: *<object_name with associated service>*

External Interfaces: *<object_name with associated attribute>*

* **State Space:** *<state transition diagram/matrix>*

Attributes:

Description: *<text>*

Source(s): *<text>*

* **Traceability to Domain Model:** *<attribute_name>*

Adaptation: *<text>*

Traceability:

* **Down to Detailed Design/Code:** *<compilation units (e.g., Package specifications)>*

* **Up to Domain Model):** *<problem_space_objects, derivations>*

Page 3 of 4

Operations:

Description: *<text>*

Source(s): *<text>*

* **Traceability to Domain Model:** *<operation_name>*

Adaptation: *<text>*

* **Preconditions:** *<program_design_language>*

Algorithms: *<program_design_language>*

* **Postconditions:** *<program_design_language>*

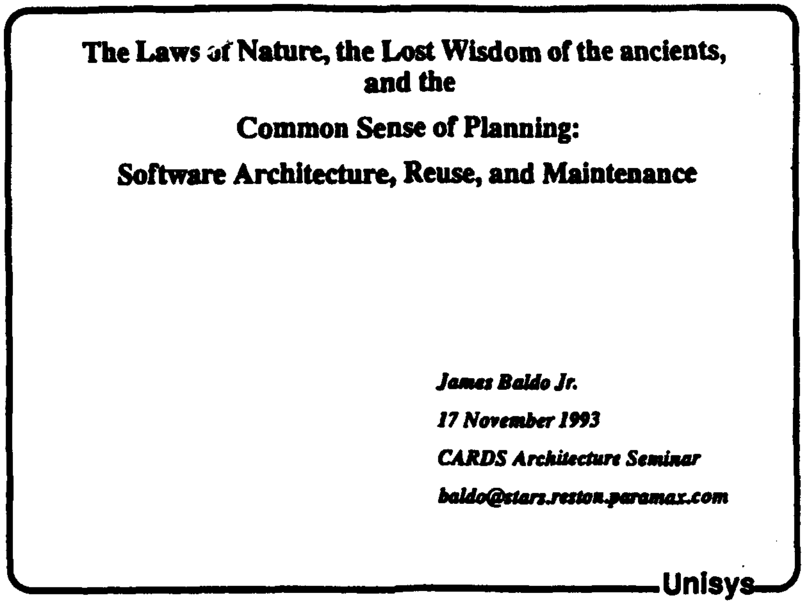
* **Timing Allocation:** *<text>*

Rationale: *<text>*

Tradeoffs: *<text>*

* additional specs not in the requirement specs

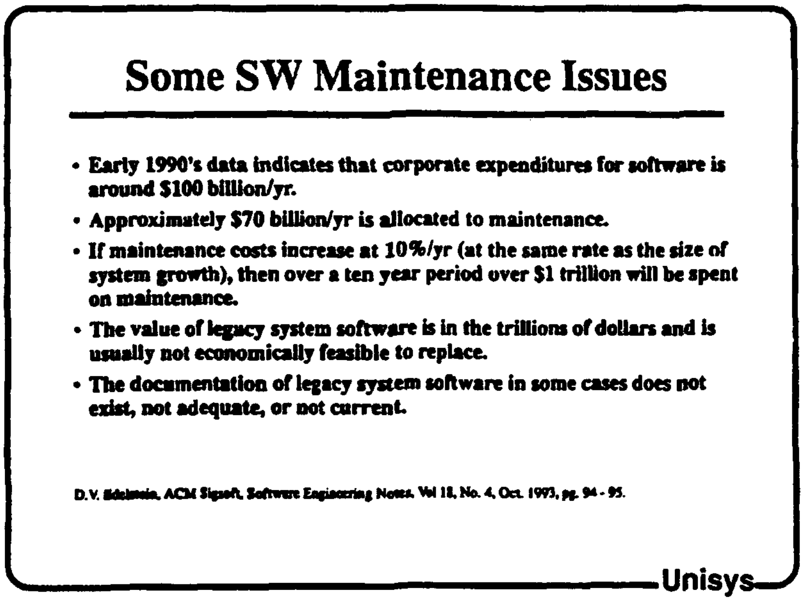
Page 4 of 4



The Laws of Nature, the Lost Wisdom of the ancients, and the Common Sense of Planning: Software Architecture, Reuse, and Maintenance

James Baldo Jr.
17 November 1993
CARDS Architecture Seminar
baldo@stars.reston.paramax.com

Unisys

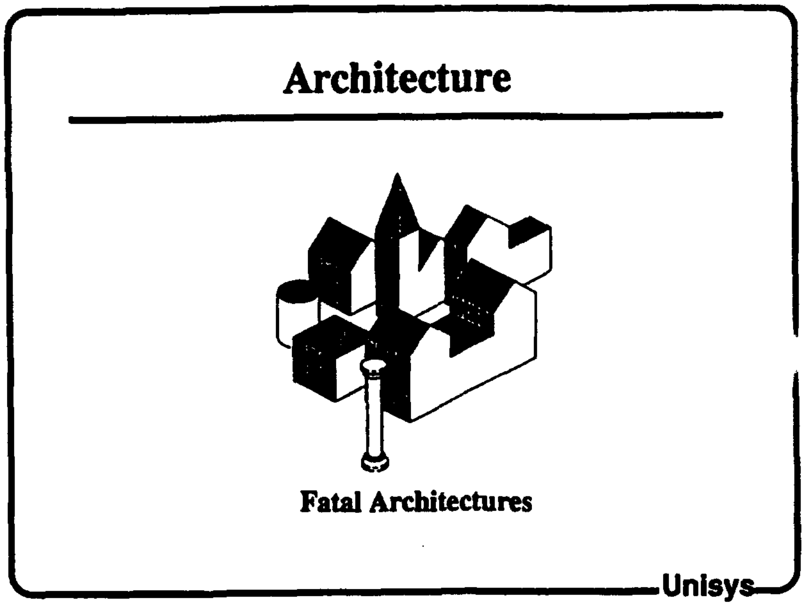


Some SW Maintenance Issues

- Early 1990's data indicates that corporate expenditures for software is around $100 billion/yr.
- Approximately $70 billion/yr is allocated to maintenance.
- If maintenance costs increase at 10%/yr (at the same rate as the size of system growth), then over a ten year period over $1 trillion will be spent on maintenance.
- The value of legacy system software is in the trillions of dollars and is usually not economically feasible to replace.
- The documentation of legacy system software in some cases does not exist, not adequate, or not current.

D.V. Edelstein, ACM Sigsoft, Software Engineering Notes, Vol 18, No. 4, Oct. 1993, pg. 94 - 95.

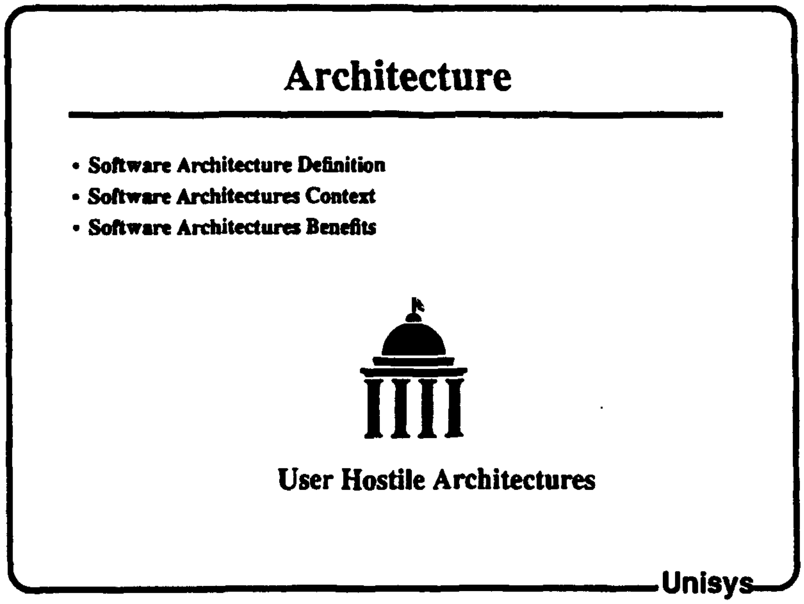
Unisys



Architecture

Fatal Architectures

Unisys

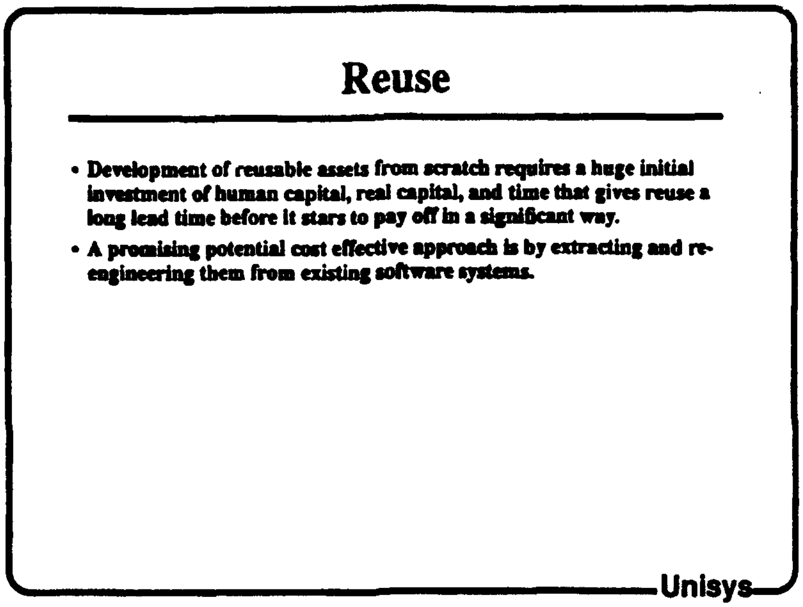


Architecture

- Software Architecture Definition
- Software Architectures Context
- Software Architectures Benefits

User Hostile Architectures

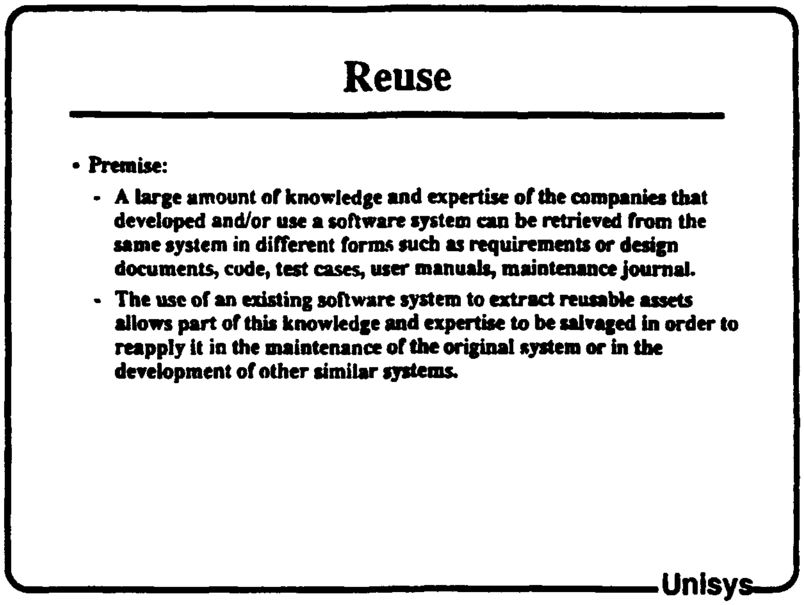
Unisys



Reuse

- Development of reusable assets from scratch requires a huge initial investment of human capital, real capital, and time that gives reuse a long lead time before it stars to pay off in a significant way.
- A promising potential cost effective approach is by extracting and re-engineering them from existing software systems.

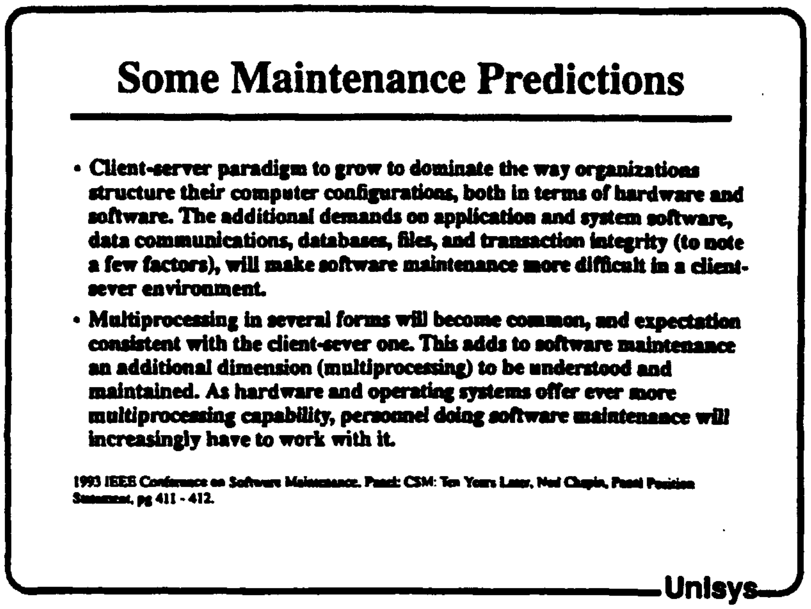
Unisys



Reuse

- Premise:
  - A large amount of knowledge and expertise of the companies that developed and/or use a software system can be retrieved from the same system in different forms such as requirements or design documents, code, test cases, user manuals, maintenance journal.
  - The use of an existing software system to extract reusable assets allows part of this knowledge and expertise to be salvaged in order to reapply it in the maintenance of the original system or in the development of other similar systems.

Unisys

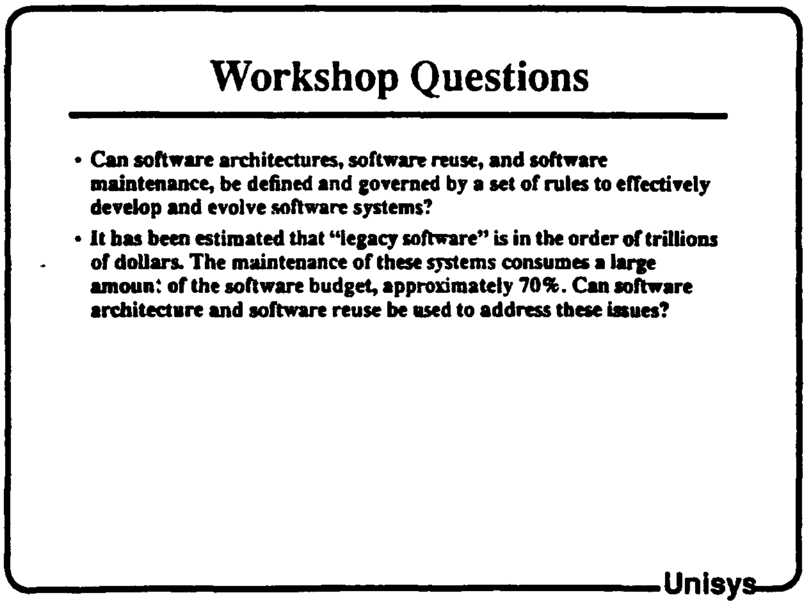


Some Maintenance Predictions

- Client-server paradigm to grow to dominate the way organizations structure their computer configurations, both in terms of hardware and software. The additional demands on application and system software, data communications, databases, files, and transaction integrity (to note a few factors), will make software maintenance more difficult in a client-sever environment.
- Multiprocessing in several forms will become common, and expectation consistent with the client-sever one. This adds to software maintenance an additional dimension (multiprocessing) to be understood and maintained. As hardware and operating systems offer ever more multiprocessing capability, personnel doing software maintenance will increasingly have to work with it.

1993 IEEE Conference on Software Maintenance. Panel: CSM: Ten Years Later, Ned Chapin, Panel Position Statement, pg 411 - 412.

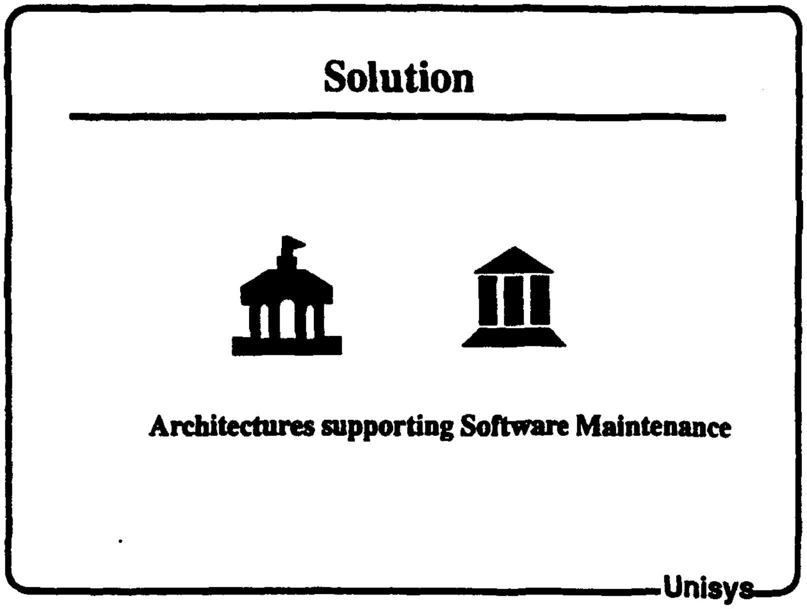
Unisys



Workshop Questions

- Can software architectures, software reuse, and software maintenance, be defined and governed by a set of rules to effectively develop and evolve software systems?
- It has been estimated that "legacy software" is in the order of trillions of dollars. The maintenance of these systems consumes a large amoun: of the software budget, approximately 70%. Can software architecture and software reuse be used to address these issues?

Unisys



Solution

Architectures supporting Software Maintenance

Unisys

Accel Software Engineering

Accel Software Engineering

Providing advanced technologies for engineering software solutions

Presentation for the CARDS Seminar on Software Architectures and Reuse

Charles Plinta

November 16, 1993 [illegible]

[illegible]

Notes for Slide : ""

Abstract:

Accel Software Engineering was founded by Richard D'Ippolito, Kenneth Lee, and Charles Plinta to bring to a wider audience advanced technology in domain-specific software architectures for engineering solutions to complex system problems. This technology was developed by the principals during their seven-year tenure at the Software Engineering Institute (SEI), a federally-funded research and development center in Pittsburgh, Pennsylvania.

For organizations specializing in software-intensive systems and seeking the capability to routinely engineer the software from a set of domain models, Accel can provide the three key elements for software success:

- Technology
- Training
- Technical Assistance

These integrated elements will reduce development time and cost, increase product quality, and extend software capabilities, all of which will increase profit.

Accel Software Engineering

What is an Architecture?

[Webster's, n (1555): a method or style of building]

Traditionally, architecture is a style of design.

- **designs (and implementations) exhibit a style resulting from the selection of a compatible set of models and rules for composition.**
- **this selection defines the structure, performance, and use of a system relative to its context and a set of engineering goals.**

Examples of cathedral architectures include: Gothic, Byzantine, and Romanesque.

Examples of models used for designing cathedrals include: windows, doors, a large open space, etc.

Notes for Slide 1: "What is an Architecture?"

Accel Software Engineering

What is a Software Architecture?

We prefer to define software architecture in a more traditional way, i.e., modeled after the definition on the previous page.

Software architecture is a style of design embodied in a set of compatible models, used to form, set, and solve software dependent systems.

- **(i.e., specify, design, and implement)**

***Accel*'s Object-Connection-Update model (OCU) is an example of a software architectural model.**

<u>*Notes for Slide 2: "What is a Software Architecture?"*</u>

Accel Software Engineering

What is the OCU Architectural Model?

The OCU is a key architectural structure for organizing software activity centers (subsystems), reducing complexity and documentation, and increasing *maintainability*.

- **Its organizing principle is based on the desired separation of what you want to do (missions) and how you do it (service providers).**
- **It provides a uniform set of packaging and communication techniques for structuring the missions and service providers into working groups.**
- **It provides activation and control of the working groups through a single software executive.**

<u>*Notes for Slide 3: "What is the OCU Architectural Model?"*</u>

We view all software systems as a collection of communicating subsystems, each subsystem [illegible] a clearly defined and separate task or mission. The mission can be expressed as a sequence of actions that the mission support objects will take.

Packaging the mission control (the "what") separately from the service providing objects (the "how") means that they can be upgraded independently, resulting in a more flexible and less brittle system. It also means that "what" and "how" experts can work independently.

The software architecture is a set of adaptable, reusable components consisting of matching pairs of specification forms and code templates. These have been used and refined in various applications for over five years. Placeholders in the forms and code templates are filled in at the [illegible] level [illegible] the problem definition stage.

Accel Software Engineering

How Do We Represent the OCU?

Representations:

- icons
- *specification forms*
- code templates

CASE tools used:

- MacDraw and editors (not very elegant, but did the job)

CASE tool dream:

- a tool that understands the relationship between icons, forms, and templates and allows customization.
- to replace application programming with graphical application composition.

Notes for Slide 4: "How Do We Represent the OCU?"

Icons are graphical representations with semantic meaning. Icons have specification forms and code templates associated with them.

Specification forms are text-based form [illegible] which capture and provide information to [illegible] each instance of a model.

Code templates are syntactically correct Ada code modules with placeholders for the information captured on a specification form.

MacDraw and other graphical drawing tools have been used by our clients to graphically represent design information. Word processing tools have been used to capture the structure of specification forms and the information associated with each OCU instance. Text editors and Ada language sensitive editors have been used to capture the code templates. The editor's substitution capabilities are used to "instantiate" the templates. Several clients have produced alpha-versions of tools that automate the process of transferring information from the forms to the templates. Given today's tooling technologies, more effective solutions are possible.

It is not necessary to be a software specialist to understand and apply the architecture, or to be a system designer. This leads us to the dream of having end users creating their own applications with graphical application programming tools.

Accel Software Engineering

How Did We Develop the OCU?

Looked at several parts of the same application for necessary, common functionality

Designed and implemented several parts depth-first

Stepped back, looked at commonalities across parts, and created a general model

Applied resulting general model to remainder of subsystems in an incremental fashion, continuously reapplying the lessons learned

Evolved form (minor) through repeated application in several application areas over a 5 year period

NOTE: Creating an architecture is a very difficult, time consuming, and evolutionary task

Notes for Slide 5: "How Did We Develop the OCU?"

Factors in the application and evolution of the OCU Architectural Model

Aeronautical Systems Center (ASC) at Wright-Patterson AFB has adopted and evolved the technology under the name of Structural modeling as a standard for acquisition, analysis, and development of aircrew training simulators.

Air Force Institute of Technology (AFIT), recognizing that this is a key technology for a part the [illegible] software engineering, has adopted it as the foundation of the [illegible] graduate program in software engineering; to date, four Master's theses have been completed on the new program.

Joint Modeling and Simulation System (J-MASS) Program has adopted and tailored the modeling technology to be a standard for the design of electronic combat simulators and simulation systems.

Comanche Systems Section (CSS) has adopted and extended the technology for the Comanche System Integrated Training Equipment (CSITE) Program, an extensible, stand-alone [illegible] on-board training [illegible] helicopter crews.

NORAD's Cheyenne Mountain Upgrade Program used the Message Translation and Validation (MTV) model and an automated development approach to develop and implement part of Granite Sentry, a command and control system.

The Portable, Reusable, Integrated Software Modules (PRISM) Program out of Electronic Systems Center (ESC/AVS) at HANSCOM AFB is developing a generic command center and has included the MTV model in their baseline reusable architecture library, CARDS.

The Advanced [illegible] Wide Surface System (AMWSS) [illegible] Program out of the Software Technology for Adaptable, Reliable Systems (STARS) Program has used and applied the technology to the design of the [illegible] embedded vehicle component of [illegible], capable of providing [illegible] returns at a [illegible] rate.

Accel Software Engineering

What Benefits Have Been Realized From Using the OCU?

Best practice is accessible and easily replicated.

Application complexity is *reduced.*

Components of risk are isolated in models.

Coding, testing, and documentation costs are reduced.

Domain expertise is isolated, captured, and given a common software structure in models.

Maintainability and enhanceability are increased.

Gaps between analysis and design and between design and implementation are bridged

Notes for Slide 6: "What Benefits Have Been Realized From Using the OCU?"

Benefits Obtained with the OCU Architectural Model

Best practice is accessible and easily replicated

- [illegible] specification forms and software templates exist for objects, controllers, and [illegible].

Application complexity is reduced

- Object, controllers, and [illegible] are easily [illegible] building blocks
- Subsystem [illegible] of complexity.

Components of risk are isolated in models.

- [illegible]
- Risk is reduced [illegible] of implementation.
- [illegible]

Coding, testing, and documentation costs are reduced.

- Specification forms and [illegible] for objects, controllers, and [illegible]
- Specification forms and [illegible] as needed

Domain expertise is isolated, captured, and given a common software structure in models.

- Software expertise is captured in software templates for objects, controllers, and [illegible]
- Engineering expertise, in the form of algorithms of implementation, [illegible]

Maintainability and enhanceability are increased

- [illegible]

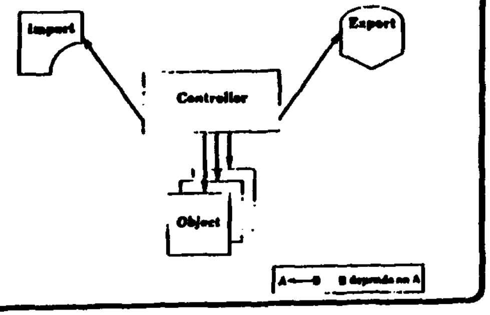
Bridging gaps between analysis and design and between design and implementation

- Specification forms and software templates are the bridges between [illegible]

[illegible]

Accel Software Engineering

OCU (Subsystem) Iconic Diagram



Notes for Slide 7: "OCU (Subsystem) Iconic Diagram"

[illegible]

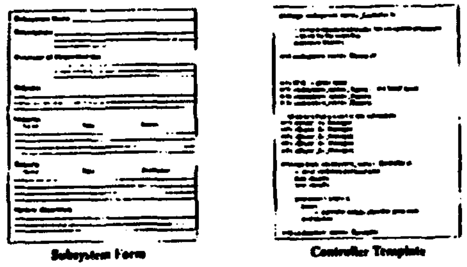
Subsystem elements:

- Objects   service providing elements
- Controller   elements that [illegible]
- Import   elements that [illegible]
- Export   elements that provide data [illegible] subsystem's solution

[illegible]

Accel Software Engineering

Forms and Templates – Incomplete



Subsystem Form

Controller Template

Notes for Slide 8: "Forms and Templates – Incomplete"

The OCU is applied with the aid of the subsystem specification form and the element code templates.

The subsystem form provides a standard way for the systems analyst and domain engineers to record the specifications of subsystems in terms of the known components/elements of subsystems.

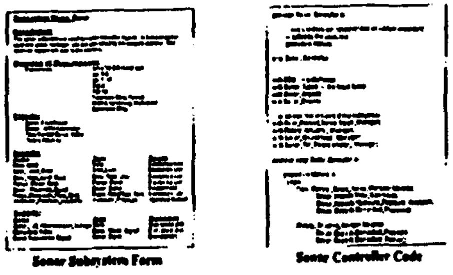
The templates provide a standard way for the software engineer to map the design of the models, captured on the forms, directly into an Ada implementation of the elements.

The systems analyst, in consultation with the customers and users, analyzes the application to identify subsystems consistent with the concept of operation and partitioning of the application. Each of the subsystems is assigned a specification form and passed to the appropriate domain engineer for completion.

For example, in the flight training simulator application some typical subsystems would be electrical, hydraulic, and fuel. In addition to the identification of subsystems, the systems analyst will provide the domain engineer with a mapping of the training requirements expressed in terms of model fidelity, operations, modes, and malfunctions.

Accel Software Engineering

Forms and Templates – Completed



Sonar Subsystem Form

Sonar Controller Code

Notes for Slide 9: "Forms and Templates – Completed"

The domain engineer receives a partially completed specification form and some physical subsystem information from the systems analyst. The domain engineer then maps the physical subsystem elements drawn up for the subsystem to OCU elements, basing the selections on the specified fidelity requirements. The OCU elements are provided with placeholders for the specific update and control algorithms, which are filled in as they become available. The result is a logical mapping from physical objects to software objects.

The domain engineer completes the specification form by recording the mapping and forwarding the form, containing the natural language descriptions of the parameterized OCU elements, to the software engineer for completion and use.

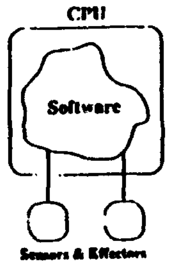
Having a predetermined set of code templates for each subsystem element means that the interface mechanisms between elements is known ahead of and is independent of the subsystem being modeled. All modeled subsystems look structurally alike, and each subsystem can be made to fit within a single template, with communication between subsystems also being handled by common structures.

This enables the software engineers to proceed with executive and test harness design in parallel with the subsystem modeling. It also means that the model specifiers can work independently in their own domains, knowing that their models will fit into the completed system.

Thus, a completed application will consist of as many instances of the OCU subsystem model as required by the use and fidelity requirements.

Accel Software Engineering

General Application Notes - 1



Notes for Slide 10: "General Application Notes - 1"

Many problems, as seen earlier, can be separated into the following three categories: problem context, problem information, and other stuff.

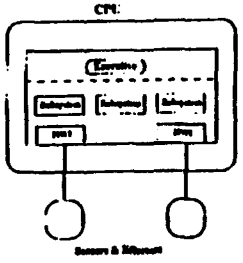
When presented with a problem that you feel can be solved using a computing platform, the [illegible] above depicts your first thoughts.

The sensors and effectors [illegible] to the application fall into the category of problem context, while the amorphous software blob to be created will somehow accomplish something based on the problem information.

We use the problem information as focus and on the software portion of the application. The OCU-style models will be the basis for focusing and working.

Accel Software Engineering

General Application Notes - 2



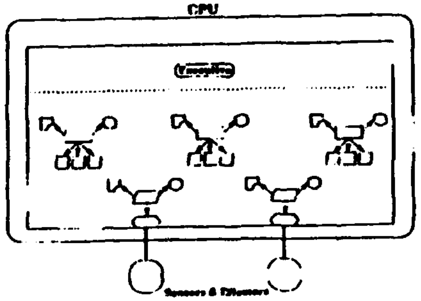
Notes for Slide 11: "General Application Notes - 2"

Based on the problem information and OCU models, we can begin to identify subsystems and hardware interfaces based on the concepts that subsystems will organize information around cohesion, and hardware interfaces will localize information regarding communication to external devices. The information is then organized by the subsystems and hardware interfaces identified.

Accel Software Engineering

General Application Notes - 3

CPU



Notes for Slide 12: "General Application Notes - 3"

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Accel Software Engineering

What is the OCU Architectural Model?

The OCU is a key architectural structure for organizing software activity centers (subsystems), reducing complexity and documentation, and increasing maintainability.

- **Its organizing principle is based on the desired separation of what you want to do (missions) and how you do it (service providers).**
- **It provides a uniform set of packaging and communication techniques for structuring the missions and service providers into working groups.**
- **It provides activation and control of the working groups through a single software executive.**

Notes for Slide 3: "What is the OCU Architectural Model?"

[illegible]

[illegible]

[illegible]

Accel
Software Engineering

Accel Software Engineering

Providing advanced technologies for engineering software solutions

Notes for Slide 14:

Recent References

The following papers record current thinking with respect to model-based software engineering:

- "The Context of Model-Based Software Engineering"
  Richard D'Ippolito
  Proceedings of the Workshop on Domain-Specific Software Architectures, July 1990
- "Modeling Software Systems by Domains"
  Richard D'Ippolito and Ken Lee
  AAAI 92 Workshop: Automating Software Design, Theme: Domain-Specific Software Architectures
- "Structural Modeling and Model-Based Software Development"
  Richard D'Ippolito, Ken Lee, Chuck Plinta, Jeff Stewart, Mike Rissman, Larry Howard, Tim Hansen
  Course prepared for and delivered at [illegible], April 1992
- "Putting the Engineering into Software Engineering"
  Richard D'Ippolito, Ken Lee, Chuck Plinta, Jeff Stewart, Paul [illegible]
  Tutorial prepared for and delivered at 6th Conference on Software Engineering Education, October 1992
- "Model-Based Software Development and the Engineering of Software"
  Chuck Plinta, Richard D'Ippolito, Ken Lee
  Tutorial given at the Tri-Ada '92, November 1992

PRISM

PRISM

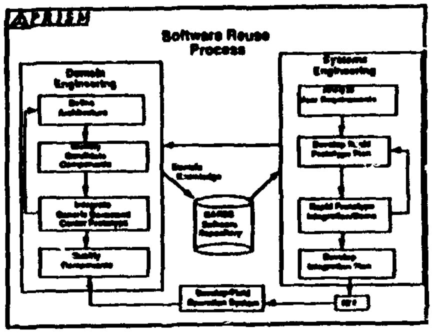
(Portable, Reusable, Integrated Software Modules)

GENERIC COMMAND CENTER

SOFTWARE ARCHITECTURE

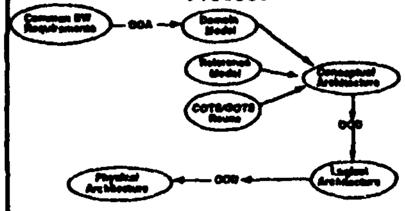
PRISM

Software Reuse Process



PRISM

Architecture Development Process



PRISM

What is a Software Architecture?

- A representation of the large-scale structural and performance characteristics of a software system.
  - Identifies system services, components and their interfaces
  - Describes component functionality, behavior and interoperation

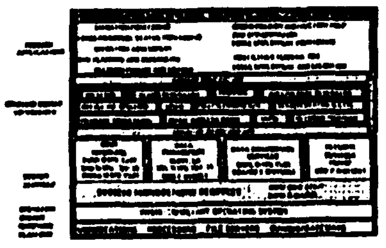
What is a Generic Software Architecture?

- A software architecture that meets a common set of requirements drawn from a specific domain
  - Addresses requirements common to the domain
  - Structural design dimensions identify recurring patterns within the domain
- A requirement may be considered common if it occurs in at least some percentage (e.g. 80%) of systems
- Represents a subset of a software architecture of a specific system within the domain

Characteristics of the PRISM GCCA

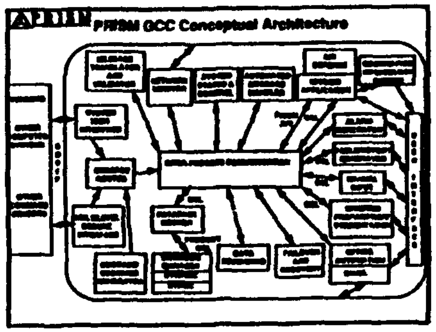
- Based on Software Components
  - Multiple implementations per component, where each is "plug-and-play" compatible
  - Each component implemented by integrating one or more COTS/GOTS software products
- Allows for the use of multiple "styles" (interaction mechanisms)
- Expressed at multiple levels of abstraction
  - Conceptual architecture
  - Logical architecture
  - Physical architecture

PRISM Reference Model



Conceptual Architecture

- Most abstract of the architecture representations
- Composed of three parts
  - Identification of constituent components
  - Identification of component interfaces
  - mapping of software requirements to components
- Provides basis for component classification scheme



PRISM GCC Conceptual Architecture

What is an API?

PRISM

- A procedural interface through which applications may invoke a service
- Types of API's
  - Interfaces with platform services (e.g. POSIX 10003.0)
  - Interfaces with standard middleware services (e.g. Motif)
  - Interfaces with domain-specific components (e.g. mapping system)
- Primary purpose is to provide application portability

Interface Development

PRISM

- Approach taken by PRISM:
  - Attempt to first make use of industry-standard interfaces
  - Define domain-specific interfaces where standards do not exist
  - Participate in relevant working groups (e.g., COE)
  - Develop generic wrapper for each component
- Experience
  - Attempts at "top-down" definition need to address existing COTS/GOTS products
  - Lack of standards results in component integration via ad hoc "wrapper" code
  - Standards still evolving

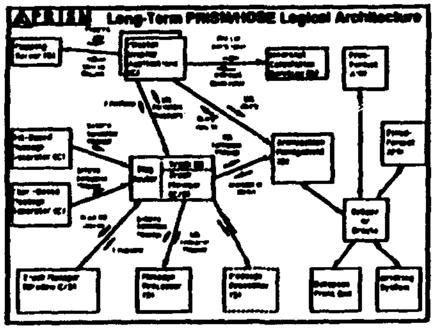
Logical Architecture

PRISM

- Illustrates relationships between components
- Illustrates logical flow of data
  - Format of data transferred
  - Protocols for data transfer
- Illustrates control flow
  - Identifies principle events and conditions
  - Identifies operational modes/states
  - Describes component behavior (as viewed from the outside)
  - Represents stimulus-response threads
- Relates to product-independent design aspects

PRISM

Long-Term PRISM/HOBE Logical Architecture



PRISM

Notation

- Components in logical architecture represented by rectangles
  - C designates a client
  - S designates a server
  - C/S indicates a component may function as a client for some components and as a server for others
- Lines connecting components represent communication paths
  - Lines with arrow at one end represent RPC calls (arrow indicates source of call)
  - RPC parameters denoted via appended text and "floating" arrows
  - Lines with arrows at both ends represent vendor-provided client-server implementation
  - Lines without arrows at end represent message passing communication

PRISM

Physical Architecture

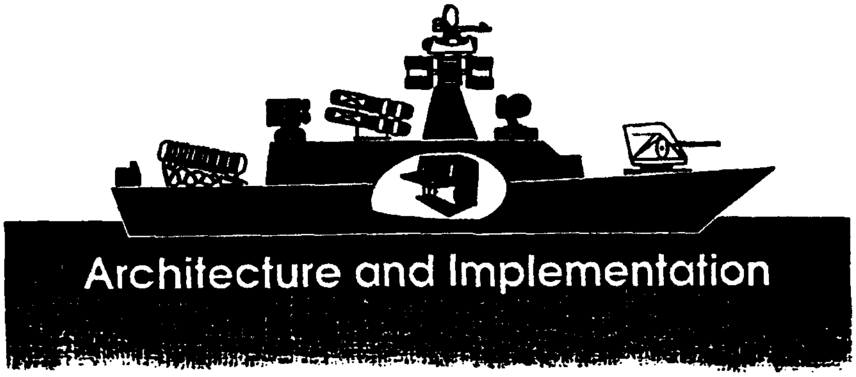
- Mapping of components to processes and tasks
- Allocation of software to hardware
- Representation of component interfaces
  - Procedures
  - OS services
  - Files
  - Environmental variables
- Description of API's
- Includes environment-specific design elements
  - Performance characteristics
  - Resource constraints
  - Mapping of processes, tasks, and data files to workstations and file servers

PRISM

Security

- Architectural attribute rather than a component
- Selection of a security policy may involve trade-offs relative to other attributes
  - Performance
  - Cost
  - Dependability
  - Ease of use
  - Speed and ease of system development

Ship System 2000



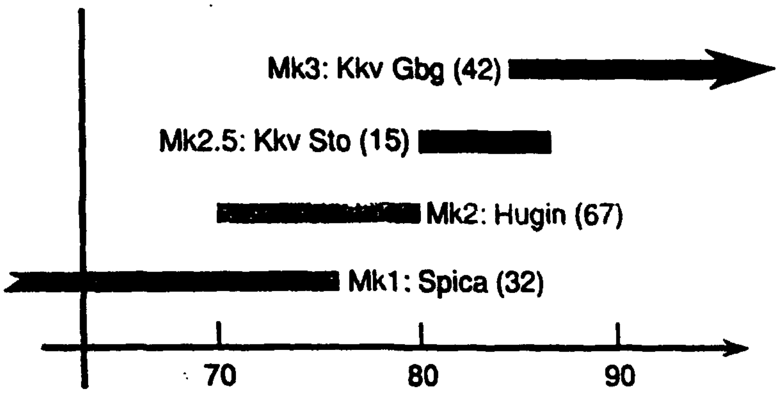
Architecture and Implementation

CelsiusTech

The Projects

		Displacement (t)	Length (m)	Armament
Göteborg (4)		380	57	Guns SSM ASW
SF300 (7+6+3)		300	54	Gun (+ role weapons)
IS/86 (4)		2700	112	Gun
Rauma (4)		200	48	Gun SSM
ANZAC (10)		3225	118	Gun SAM
Gotland (3+1)		1250	52	Torpedo
StriC		Multi-site national Air Defence System		

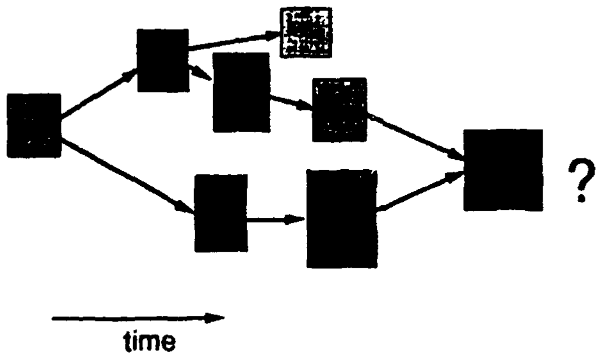
The Background



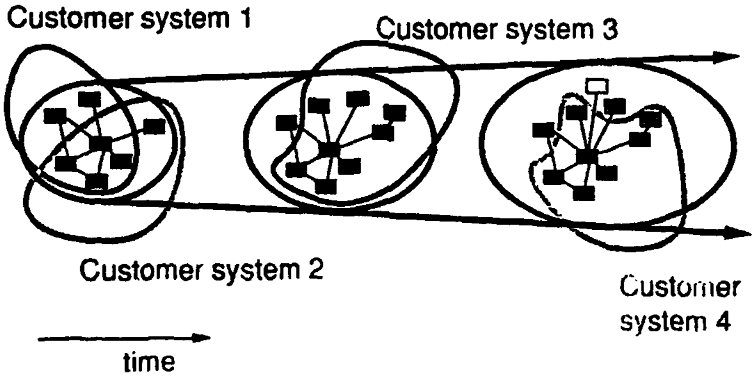
Strategy

- Structure for reuse
- Use recognized standards (open systems)
- Emphasis on applications
- Produce family of components
- Integrate components into a system

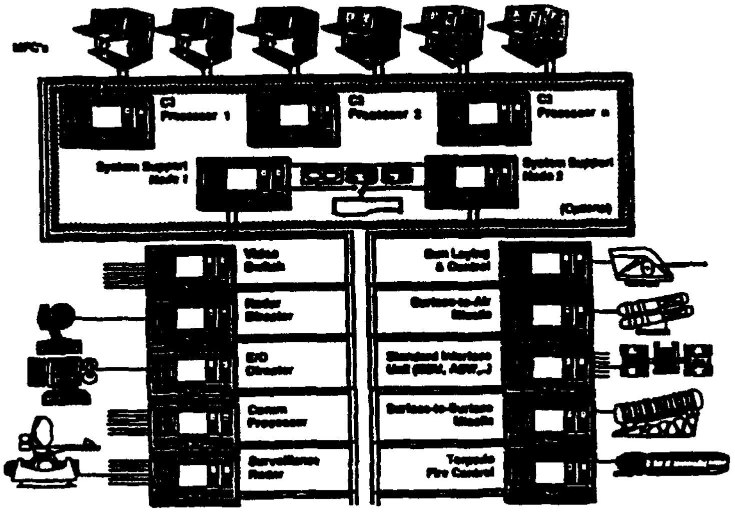
Classical Multi-Project Development



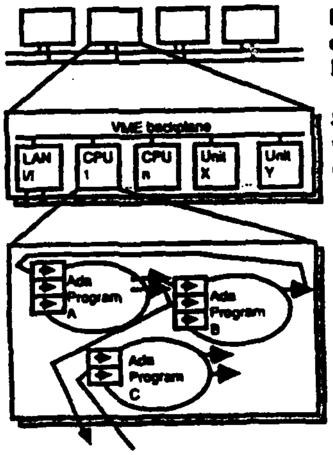
Creating a Set of Components



Ship System 2000



Structure in a Node

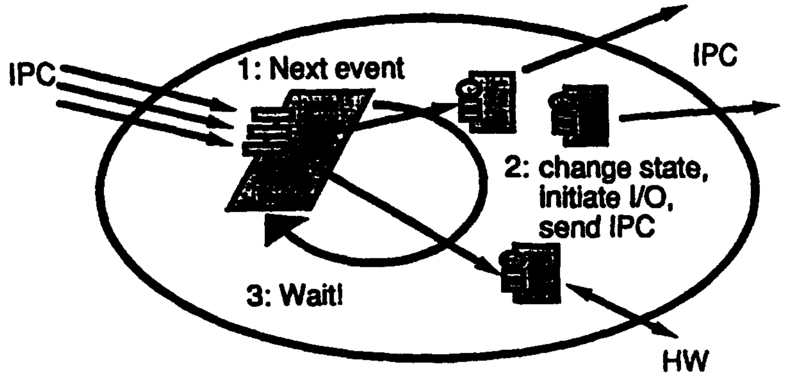


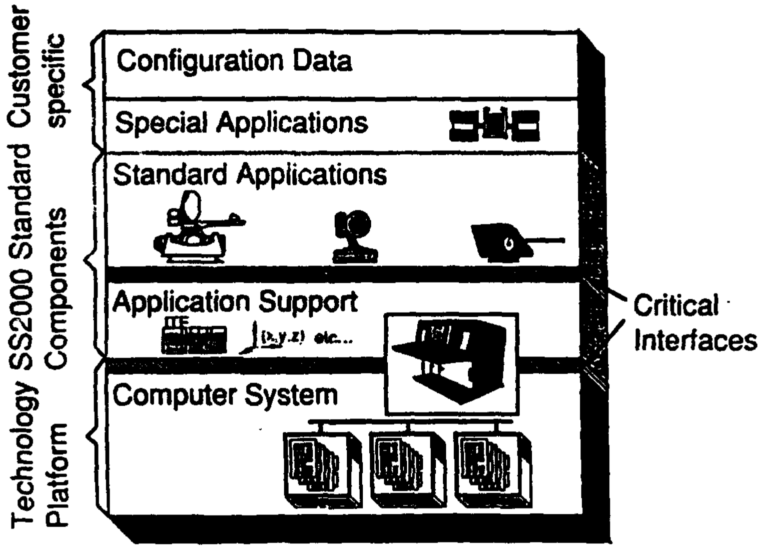
Nodes on the LAN

Structure within a node

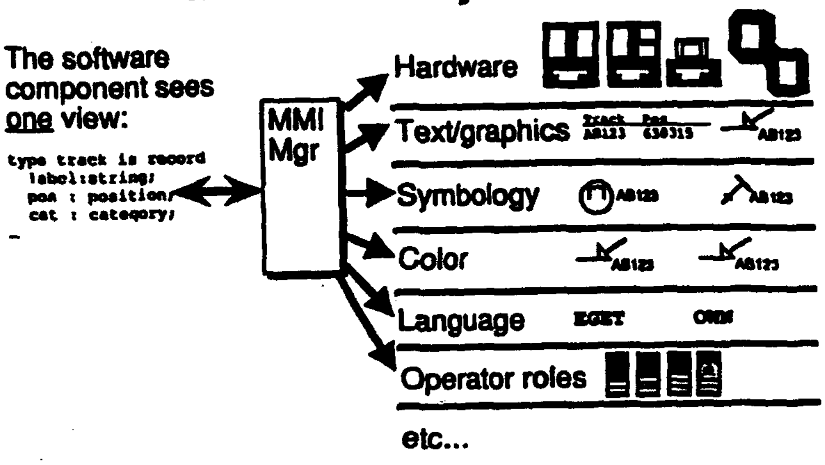
Software structure within a CPU

Life in an Ada Program





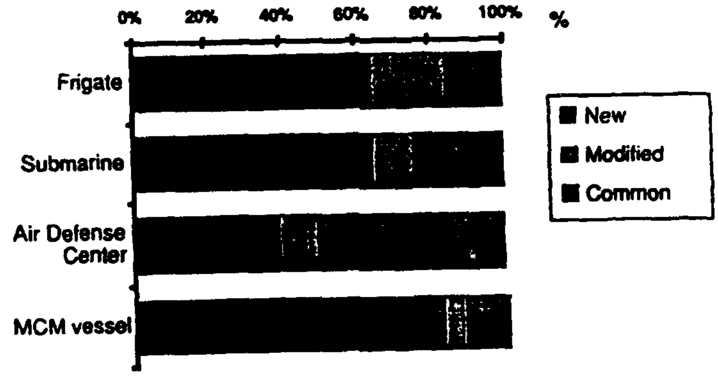
MMI Flexibility



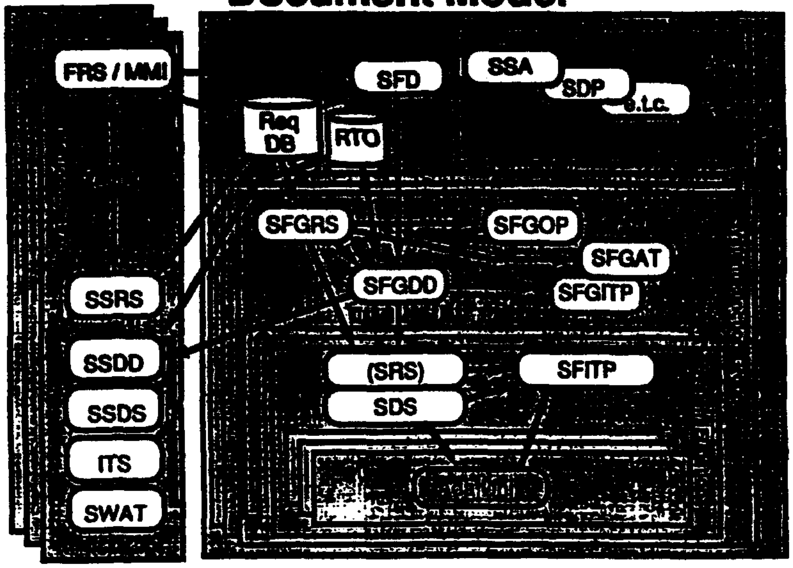
The software component sees <u>one</u> view:

```
type track is record
  label:string;
  pos : position;
  cat : category;
-
```

SS2000 Software Commonality



Document Model



The situation 1993

- Several systems operational with several customers
- Highly successful firing tests
- ≈2 MDSI operational
- Stable architecture
- High quality in the delivered software
- Demonstrated portability

DesignMetrics™ A system architecture tool.

DesignMetrics™ Technology
2 Cedartree Lane
Stamford, CT 06903
Tel. (203) 968-0594

DesignMetrics Inputs.

Requirement Records

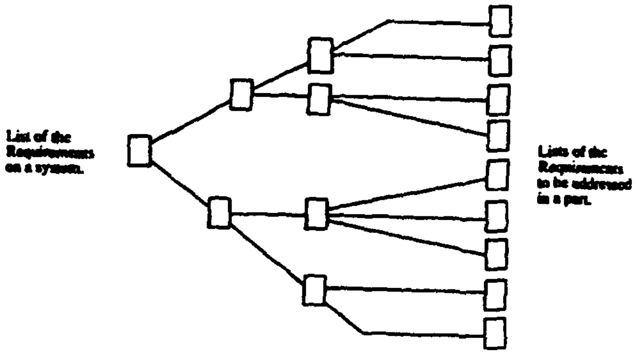
Req 1. User access to bulkCustomerSetup and Data Base Administration will be limited by user privileges at login.

Req. 53. If the customer record already exists then display the existing customer.

Requirement Interdependency Record

Req 1   8 23 147 end

The Decomposition Process

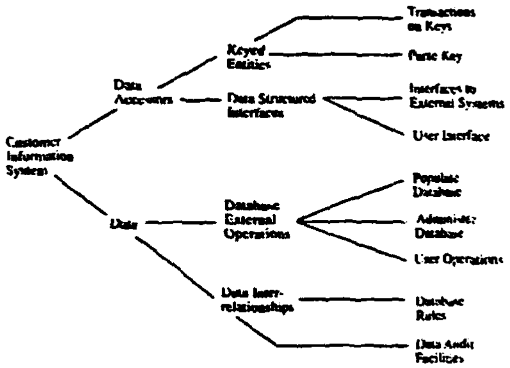


List of the Requirements on a system.

Lists of the Requirements to be addressed in a part.

DesignMetrics © 1993

Customer Information System Parts.



Customer Information System

Data Accessors

Keyed Entities

Transactions on Keys

Parse Key

Data Structured Interfaces

Interfaces to External Systems

User Interface

Data

Database External Operations

Populate Database

Administer Database

User Operations

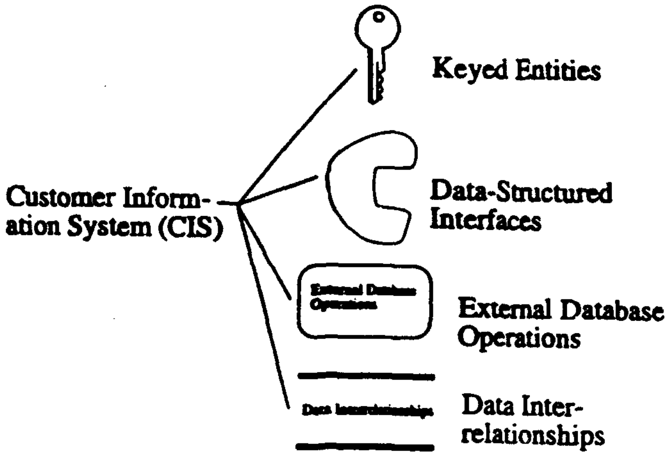
Data Inter-relationships

Database Rules

Data Audit Facilities

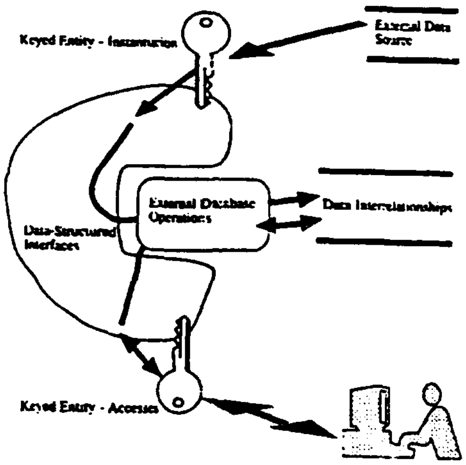
DesignMetrics © 1993

Customer Information System Parts.


Keyed Entities
Customer Information System (CIS)
Data-Structured Interfaces
External Database Operations
External Database Operations
Data Interrelationships
Data Interrelationships

DesignMetrics © 1993

The Operational Principal Of A CIS.


Keyed Entity - Instantiation
External Data Source
External Database Operations
Data Interrelationships
Data-Structured Interfaces
Keyed Entity - Accesses

DesignMetrics © 1993

NUWC Division Newport

Software Architectures

Steve Roodbeen
Naval Undersea Warfare Center
Division Newport, RI
17 November 1993

NUWC Division Newport

Architecture

- The Science, Art, Or Profession Of Designing And Constructing Buildings, Bridges, Etc.
- The Design And Integration Of Components Of A Computer Or Computer System.

NUWC Division Newport

Software Architecture

- The Science, Art, Or Profession Of Designing And Constructing Software, Software Systems, Etc.
- The Design And Integration Of Software Components In A Computer Or Computer System.

NUWC Division Newport

Current Emphasis

- Analysis, Acquisition, And Integration Of Several Heterogeneous Support Software Tools.
    - All Support Software Tools Accessible Through A Central Interface.
    - All Software System Information Accessible Through A Central Interface.
    - List Of Tools Includes: CARDS RLF, SEE-Ada, Rational Rose, AdaMAT, and Objectmaker.

NUWC Division Newport

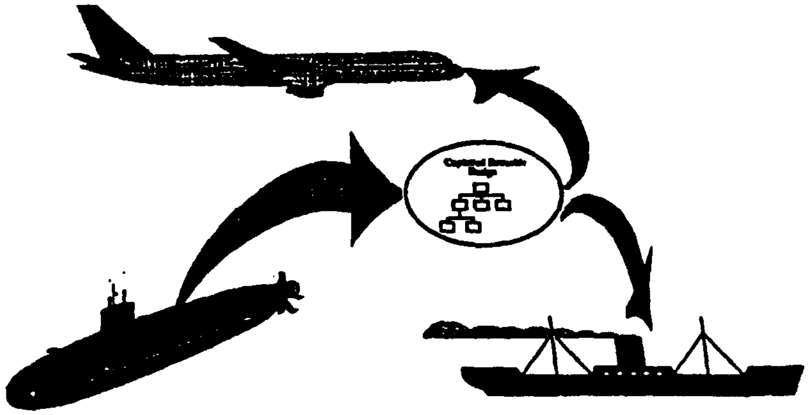
Current Goal

- Analyze Legacy Software Systems And Extract Design Information.

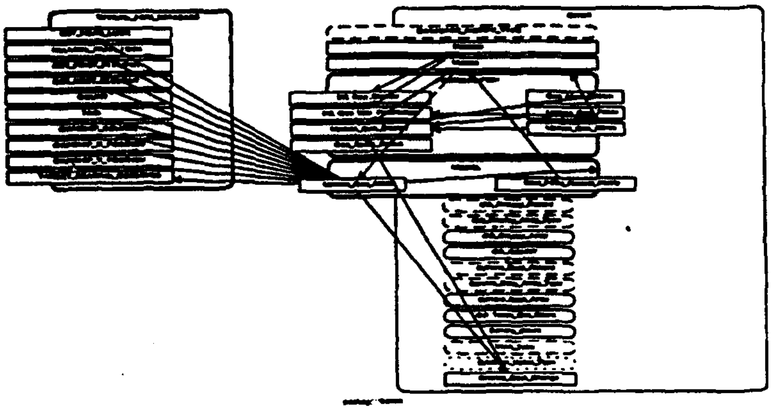
NUWC Division Newport

Design Capture

- Analysis And Extraction Of Design Information From Legacy Software.



An Object-Based View Of Functionally Designed Code



Architecture Representation

- Primary Representation Vehicle CARDS RLF
  - RLF Selected Due To Its Robustness (e.g., Its Ability To Provide Access To A Variety Of Information).
  - All Other Representation Tools Can Be Launched From The RLF. Basically, RLF Provides An Open Interface To Other Tools

NUWC Division Newport

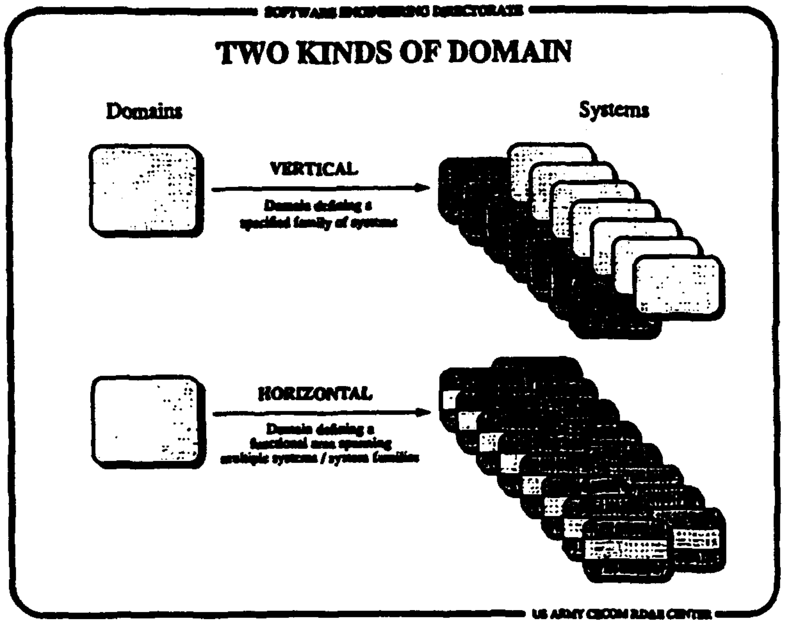
Lessons Learned

- Developer's Reluctant To Provide Design Information
- Design Information May No Longer Be Available
- Information That Is Available Is Incorrect Or Obsolete
- It Is Difficult To Incorporate The New Software Engineering Paradigm Into The Design Process (i.e., Now Is A Tuff Time To Change The Way We Do Business)

NUWC Division Newport

The Ultimate Goal

- Define Process Which Will Result In The Generation Of Reusable Software Systems/ Subsystems/Components
  - Object Oriented Technology
  - New Tools Emerging To Support This Approach
- Expand Software Architecture To Include Everything Known About A Given System



SOFTWARE ENGINEERING DIRECTORATE

TWO KINDS OF DOMAIN

Domains

Systems

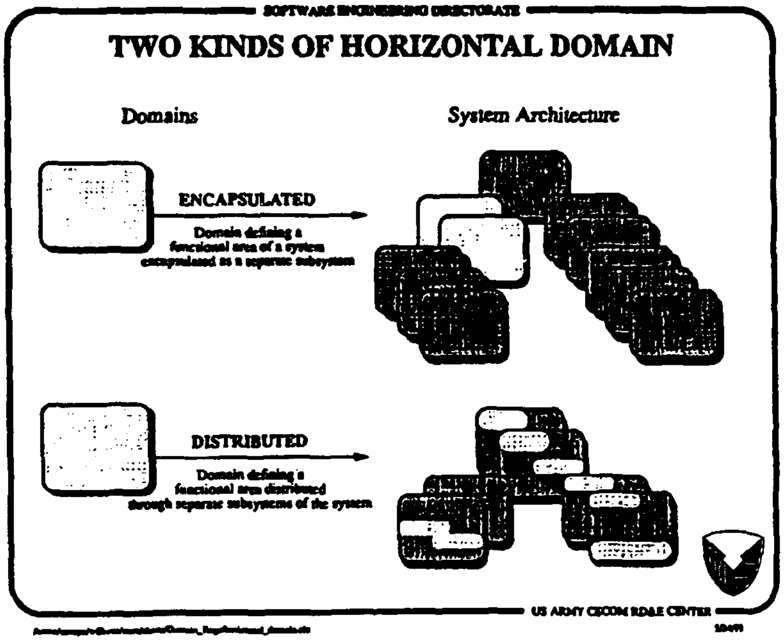
VERTICAL

Domain defining a
specified family of systems

HORIZONTAL

Domain defining a
functional area spanning
multiple systems / system families

US ARMY CECOM RD&E CENTER



SOFTWARE ENGINEERING DIRECTORATE

TWO KINDS OF HORIZONTAL DOMAIN

Domains

System Architecture

ENCAPSULATED

Domain defining a
functional area of a system
encapsulated as a separate subsystem

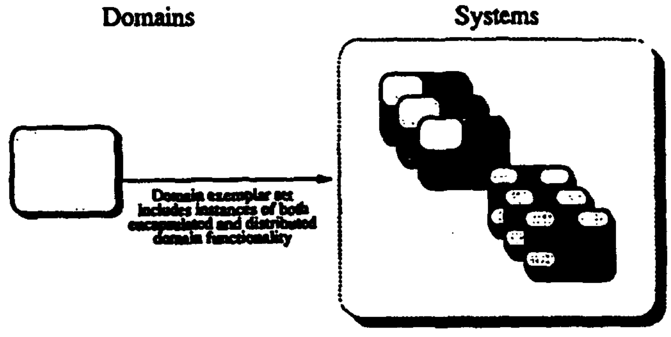
DISTRIBUTED

Domain defining a
functional area distributed
through separate subsystems of the system

US ARMY CECOM RD&E CENTER

SOFTWARE ENGINEERING DIRECTORATE

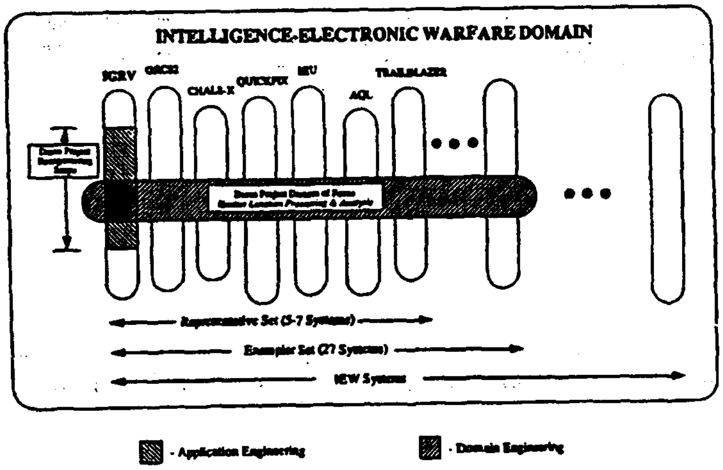
MAXIMUM DIVERSITY DOMAIN / SYSTEM INTERACTION


Domains
Systems
Domain exemplar set includes instances of both encapsulated and distributed domain functionality

US ARMY CECOM RD&E CENTER

SOFTWARE ENGINEERING DIRECTORATE

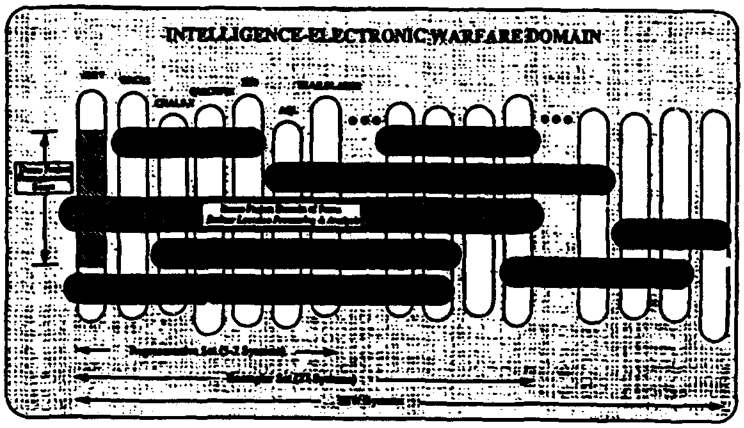
SYSTEMATIC APPROACH TO IEW REUSE


INTELLIGENCE-ELECTRONIC WARFARE DOMAIN
IGRV
GRCS
CHALS-X
QUICKFIX
MIU
AQL
TRAILBLAZER
Demo Project Reengineering Scope
Demo Project Domain of Focus
Emitter Location Processing & Analysis
Representative Set (5-7 Systems)
Exemplar Set (27 Systems)
IEW Systems
▨ - Application Engineering
▨ - Domain Engineering

US ARMY CECOM RD&E CENTER

SOFTWARE ENGINEERING DIRECTORATE

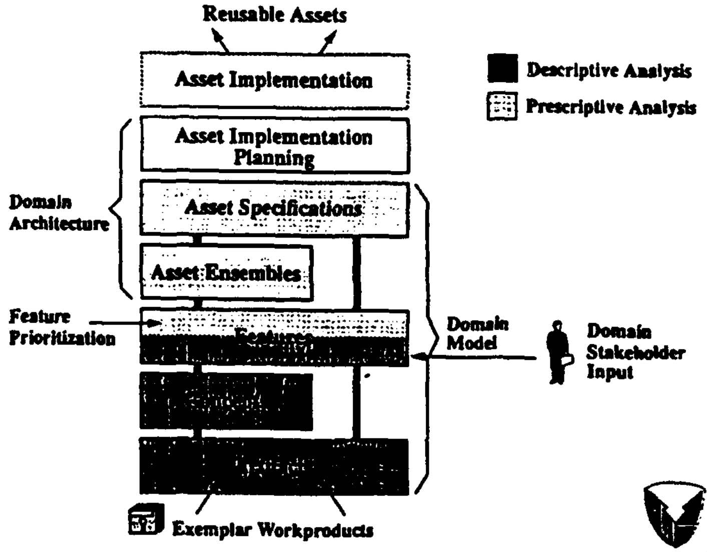
SYSTEMATIC APPROACH TO IEW REUSE (CONT'D)



US ARMY CECOM RD&E CENTER

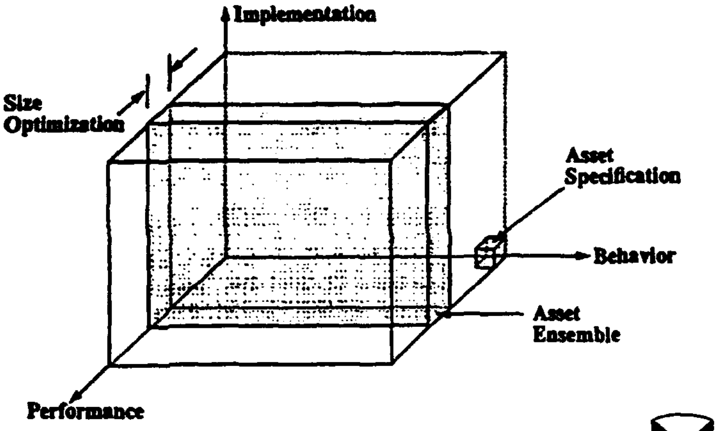
SOFTWARE ENGINEERING DIRECTORATE

ODM DOMAIN ANALYSIS REFERENCE MODEL



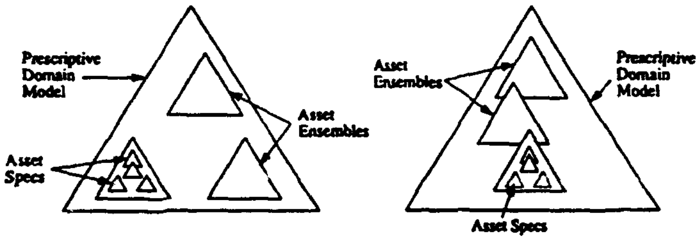
US ARMY CECOM RD&E CENTER

PRESCRIPTIVE ANALYSIS SOLUTION SPACE



"n" DIMENSIONAL FEATURE SPACE

DOMAIN ARCHITECTURAL MODEL VARIANTS



Separately Selectable Ensembles

Layered Ensembles

- A Domain Architectural Model will be some combination of a layered and separately selectable set of Asset Ensembles
- Asset Base Architecture underlys the domain architecture

STARS/AIR FORCE DEMONSTRATION PROJECT
DOMAIN ENGINEERING - SWSC PRODUCT-LINE APPROACH

Air Force/STARS Demonstration Project Space Command & Control Architectural Infrastructure (SCAI)

Capt Kelly L. Spicer, USAF
Lead, Domain Engineering & Reuse Working Group
17 November 1993
Space and Warning Systems Center
Air Force Space Command

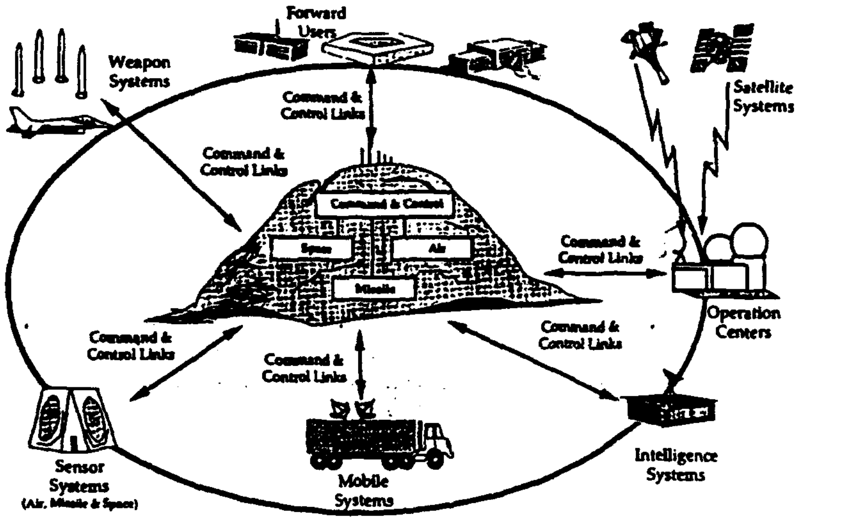
SWSC/SMX, Stop 2320, 130 W. Paine St
Peterson AFB, Co, 80914-2320

(719)554-6675
kspicer@spacecom.af.mil

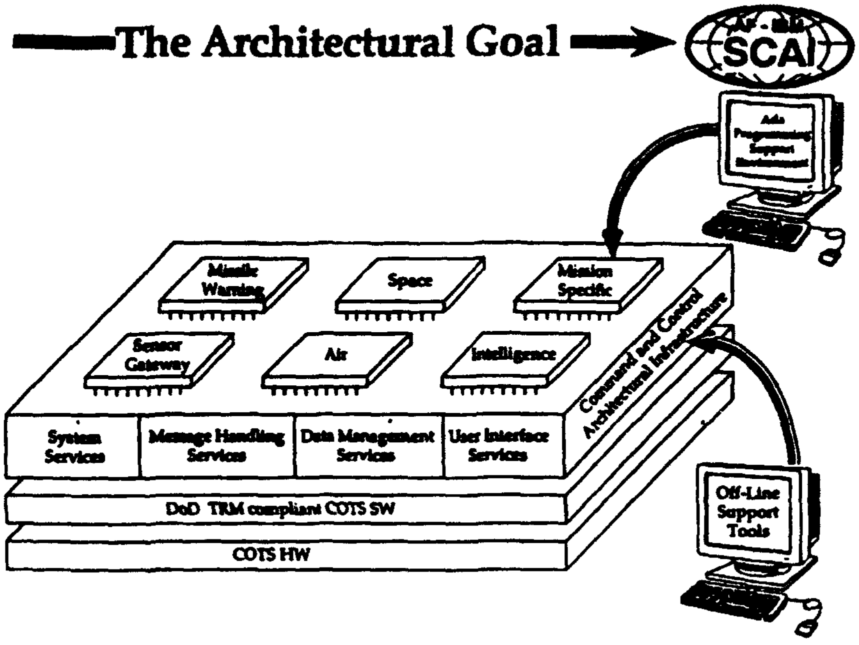
[illegible]CARDS_[illegible]

Major Systems

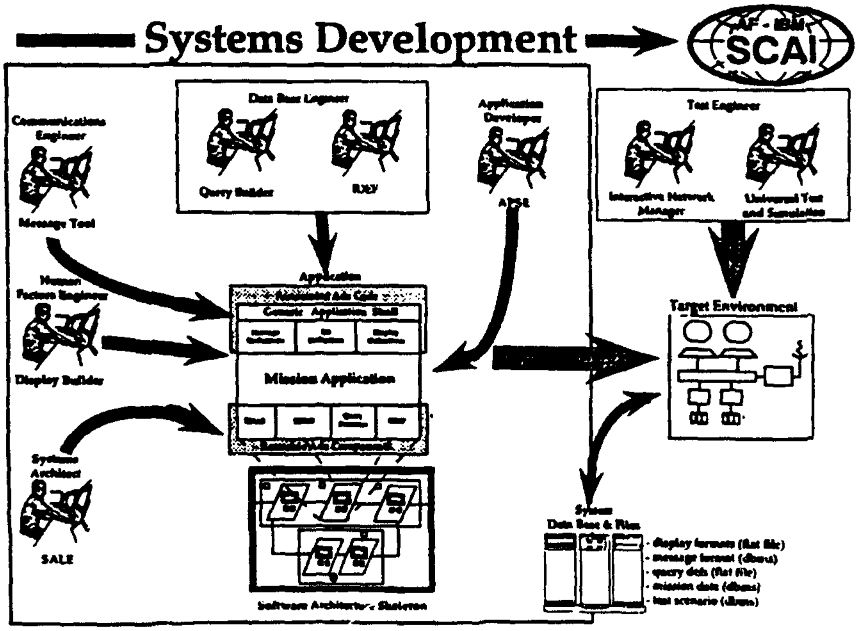
AF - IBM SCAI



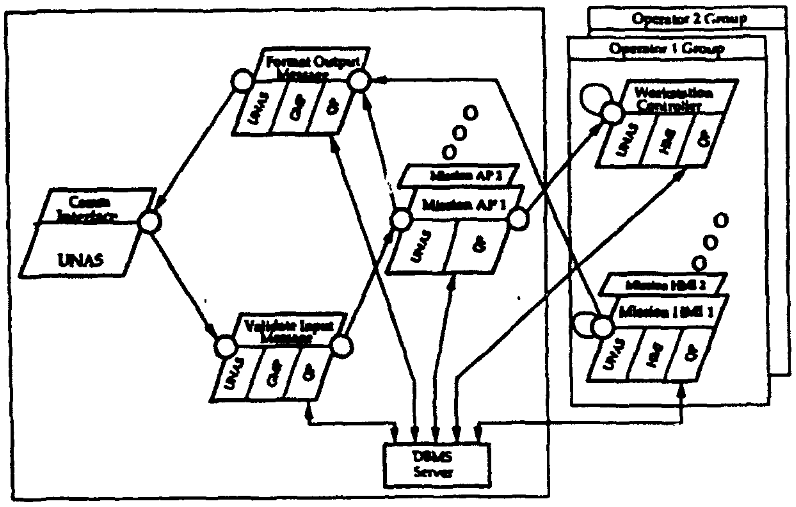
The Architectural Goal



Systems Development

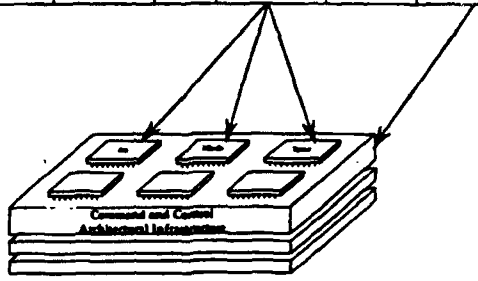


Command Center Architecture



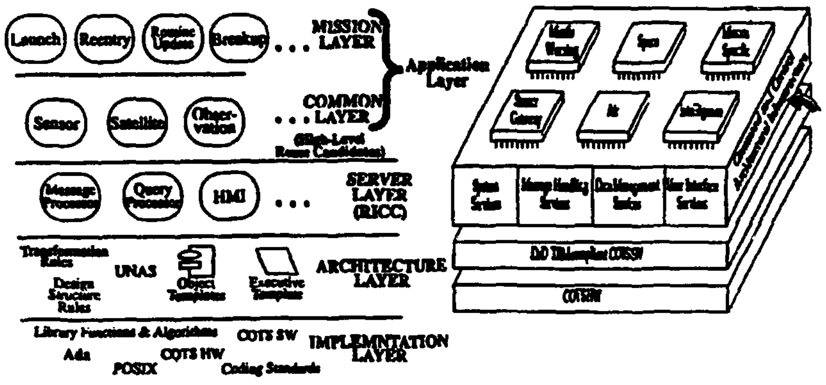
Results

Mission	Platform	Duration	Effort	Application	Automatic	Reused	Total
Air	DEC/VMS	4.3	8.5	1,506	4,484	64,440	70,430
Missile	SUN/HP/DEC/UNIX	10.0	97.0	66,323	26,714		157,477
Space	IBM/AIX	10.0	52.0	17,558	42,878		124,876



STARS/AIR FORCE DEMONSTRATION PROJECT
DOMAIN ENGINEERING - SWSC PRODUCT-LINE APPROACH

TARGET: REFINE LAYERS TO SCAI ARCHITECTURE



STARS/AIR FORCE DEMONSTRATION PROJECT
DOMAIN ENGINEERING - SWSC PRODUCT-LINE APPROACH

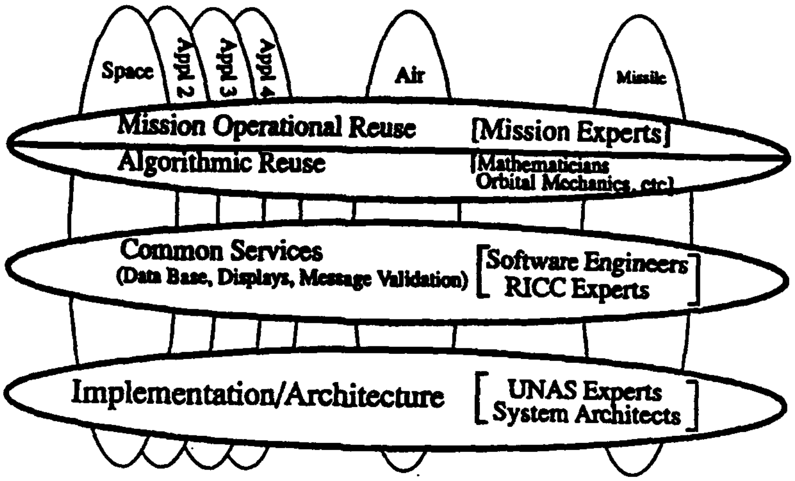
REFINE LAYERS TO SCAI ARCHITECTURE

- Abstract Display-User Interaction Classes Into Mission Objects/ Classes
- Define Standard Structure For Mission Objects
- Continue to Refine Layering Scheme:
    - •Standardize Layer Interfaces (e.g. Common Layer)
    - •Define Standardized Interface to RICC Tools
- Define Consistent Display Interface Paradigms
- Extend Scope of OO Analysis to Other Missions Besides Space
- Extend RICC "Layer" To Include Additional Tools

STARS/AIR FORCE DEMONSTRATION PROJECT
DOMAIN ENGINEERING - SWSC PRODUCT-LINE APPROACH

Building the Product-Line Organization

[Functional Organization Mimics Architecture Layering]



Mission Operational Reuse [Mission Experts]

Algorithmic Reuse [Mathematicians
Orbital Mechanics, etc]

Common Services
(Data Base, Displays, Message Validation) [Software Engineers
RICC Experts]

Implementation/Architecture [UNAS Experts
System Architects]

[illegible]/USAF/[illegible]/CARDS [illegible]

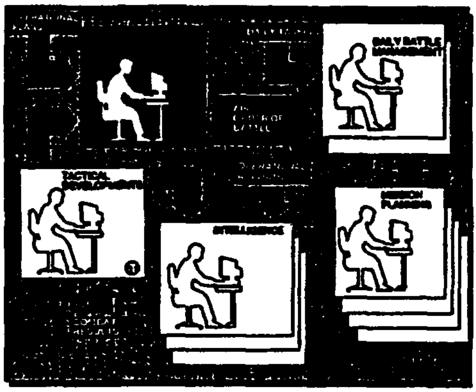
APPLICATION
BASED
SYSTEM

APPLICATION BASED SYSTEM

INTRODUCTION:
A SOFTWARE ARCHITECTURE SUITABLE FOR COMPLEX INTERACTIVE C^3I SYSTEMS

PRESENTED BY:

STELLAN KÄRNEBRO
DEFENCE MATERIEL ADMINISTRATION
SWEDEN



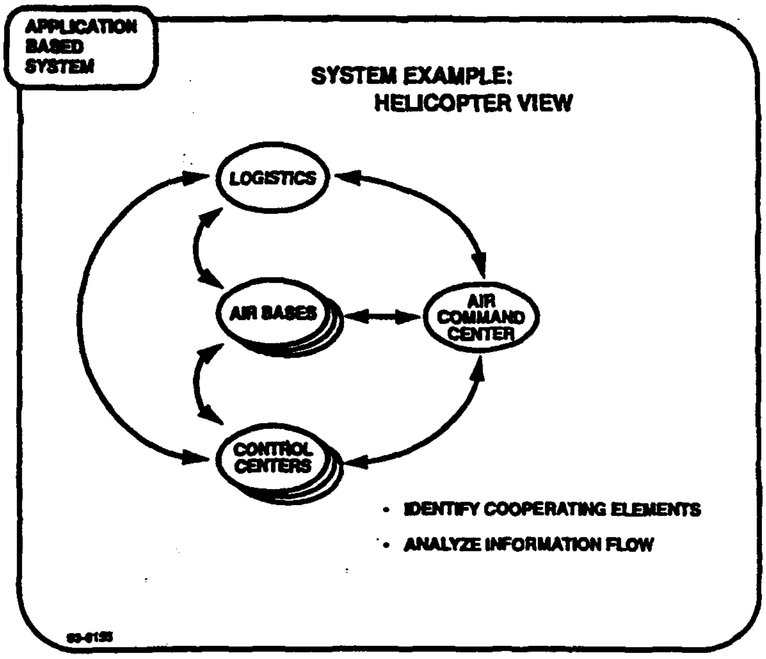
93-0153

APPLICATION
BASED
SYSTEM

TARGETED C^3I SYSTEM CHARACTERISTICS

- MULTIPLE COOPERATING APPLICATIONS
- GEOGRAPHICALLY DISTRIBUTED
- TOLERANT TO COMMUNICATION LINK INTERRUPTION
- REPLICATED DISTRIBUTED DATABASES
- VARYING SECURITY LEVELS AND REQUIREMENTS
- PORTABLE AND REUSABLE APPLICATION SOFTWARE

93-0154

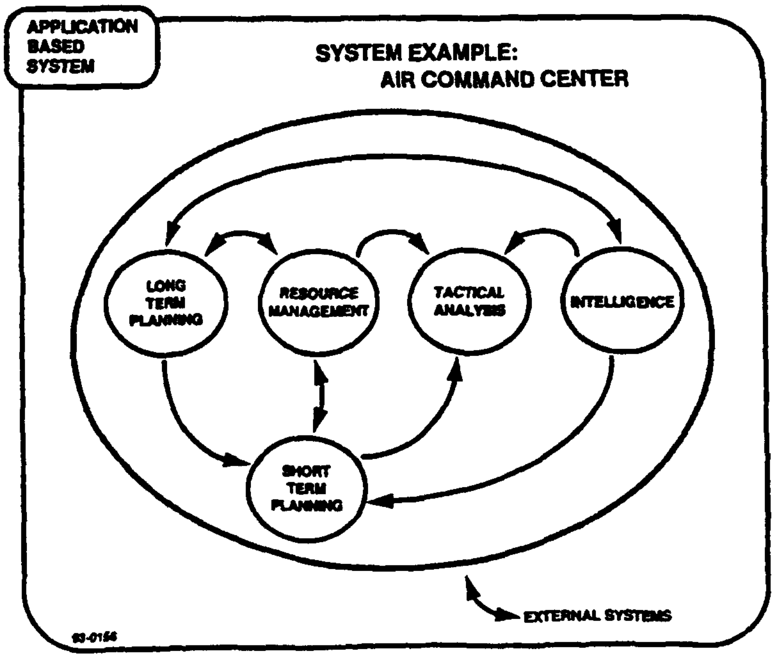


APPLICATION BASED SYSTEM

SYSTEM EXAMPLE: HELICOPTER VIEW

- IDENTIFY COOPERATING ELEMENTS
- ANALYZE INFORMATION FLOW

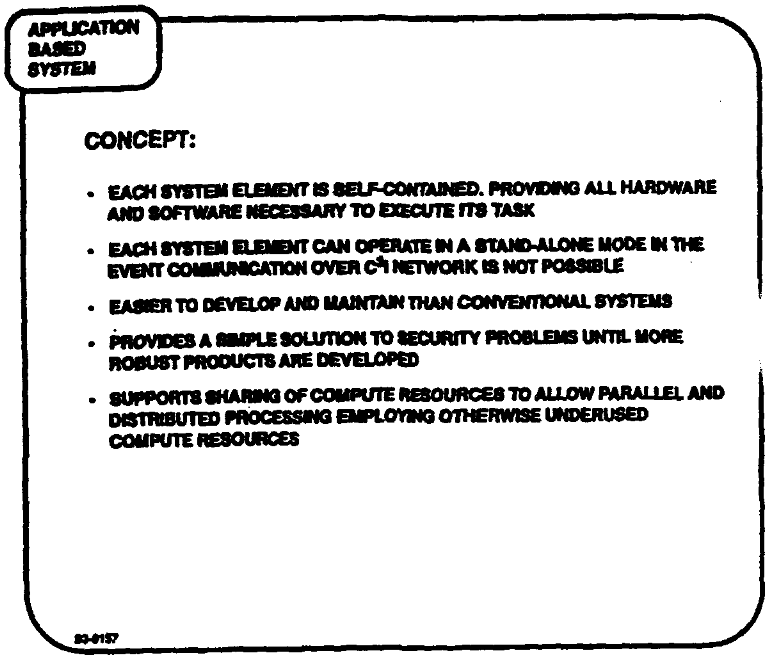
93-0155



APPLICATION BASED SYSTEM

SYSTEM EXAMPLE: AIR COMMAND CENTER

93-0154

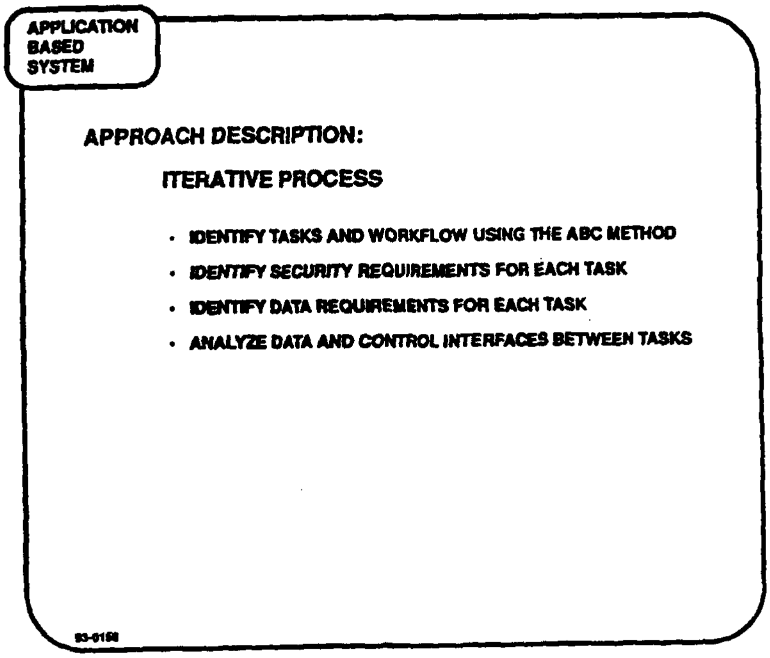


APPLICATION
BASED
SYSTEM

CONCEPT:

- EACH SYSTEM ELEMENT IS SELF-CONTAINED. PROVIDING ALL HARDWARE AND SOFTWARE NECESSARY TO EXECUTE ITS TASK
- EACH SYSTEM ELEMENT CAN OPERATE IN A STAND-ALONE MODE IN THE EVENT COMMUNICATION OVER C^3I NETWORK IS NOT POSSIBLE
- EASIER TO DEVELOP AND MAINTAIN THAN CONVENTIONAL SYSTEMS
- PROVIDES A SIMPLE SOLUTION TO SECURITY PROBLEMS UNTIL MORE ROBUST PRODUCTS ARE DEVELOPED
- SUPPORTS SHARING OF COMPUTE RESOURCES TO ALLOW PARALLEL AND DISTRIBUTED PROCESSING EMPLOYING OTHERWISE UNDERUSED COMPUTE RESOURCES

93-0157



APPLICATION
BASED
SYSTEM

APPROACH DESCRIPTION:

ITERATIVE PROCESS

- IDENTIFY TASKS AND WORKFLOW USING THE ABC METHOD
- IDENTIFY SECURITY REQUIREMENTS FOR EACH TASK
- IDENTIFY DATA REQUIREMENTS FOR EACH TASK
- ANALYZE DATA AND CONTROL INTERFACES BETWEEN TASKS

93-0158

APPLICATION
BASED
SYSTEM

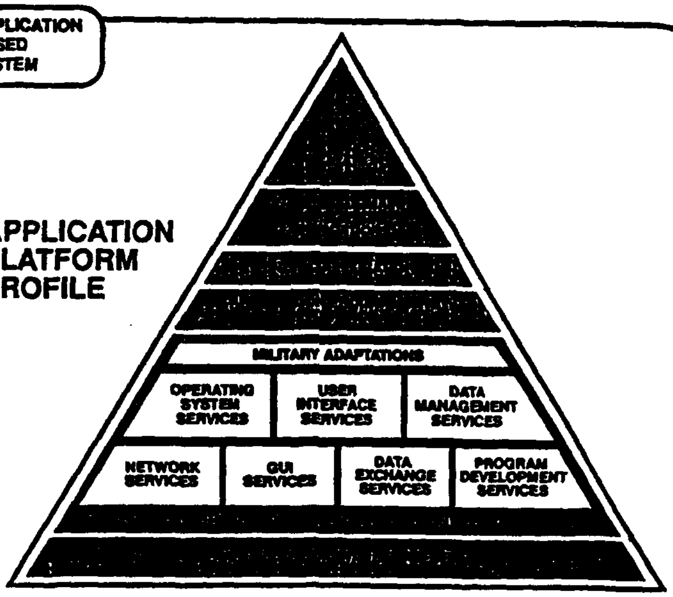
APPROACH DESCRIPTION:

- DECOMPOSE SYSTEM INTO FUNCTIONAL TASK GROUPS BASED UPON ANALYSIS PERFORMED IN THE PREVIOUS STEPS. THESE TASKS GROUPS SHOULD BE ORGANIZED SUCH THAT COMMUNICATION BETWEEN THEM MAY BE ACCOMPLISHED BY SIMPLE MESSAGES. THIS RESULTS IN A *SYSTEM SEGMENT SPECIFICATION* FOR EACH TASK GROUP
- DEFINE DATA BASE STRUCTURES REQUIRED FOR EACH TASK GROUP. DATABASES COMMON AMONG TASK GROUPS SHOULD SHARE THE SAME INFORMATION CONTENT IN ORDER TO SUPPORT RELOCATION OF APPLICATION CODE. THIS RESULTS IN A *DATABASE DESCRIPTION DOCUMENT*
- DEFINE MESSAGES USED TO COMMUNICATE BETWEEN TASK GROUPS. THIS RESULTS IN AN *INTERFACE DESIGN DOCUMENT*
- ITERATE OVER THE ABOVE STEPS UNTIL A SUITABLE ARCHITECTURE IS ACHIEVED

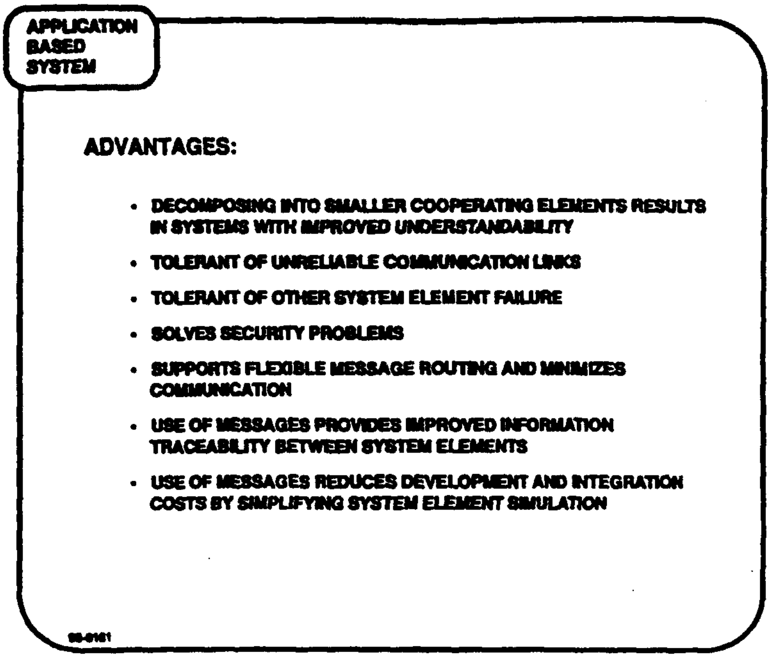
93-0189

APPLICATION
BASED
SYSTEM

APPLICATION PLATFORM PROFILE



93-0190

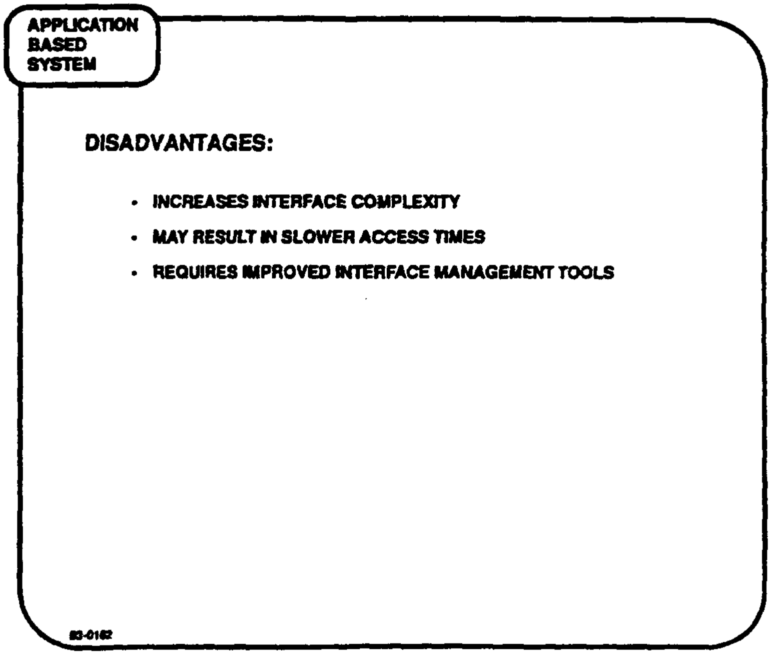


APPLICATION
BASED
SYSTEM

ADVANTAGES:

- DECOMPOSING INTO SMALLER COOPERATING ELEMENTS RESULTS IN SYSTEMS WITH IMPROVED UNDERSTANDABILITY
- TOLERANT OF UNRELIABLE COMMUNICATION LINKS
- TOLERANT OF OTHER SYSTEM ELEMENT FAILURE
- SOLVES SECURITY PROBLEMS
- SUPPORTS FLEXIBLE MESSAGE ROUTING AND MINIMIZES COMMUNICATION
- USE OF MESSAGES PROVIDES IMPROVED INFORMATION TRACEABILITY BETWEEN SYSTEM ELEMENTS
- USE OF MESSAGES REDUCES DEVELOPMENT AND INTEGRATION COSTS BY SIMPLIFYING SYSTEM ELEMENT SIMULATION

93-0161



APPLICATION
BASED
SYSTEM

DISADVANTAGES:

- INCREASES INTERFACE COMPLEXITY
- MAY RESULT IN SLOWER ACCESS TIMES
- REQUIRES IMPROVED INTERFACE MANAGEMENT TOOLS

93-0162

APPLICATION
BASED
SYSTEM

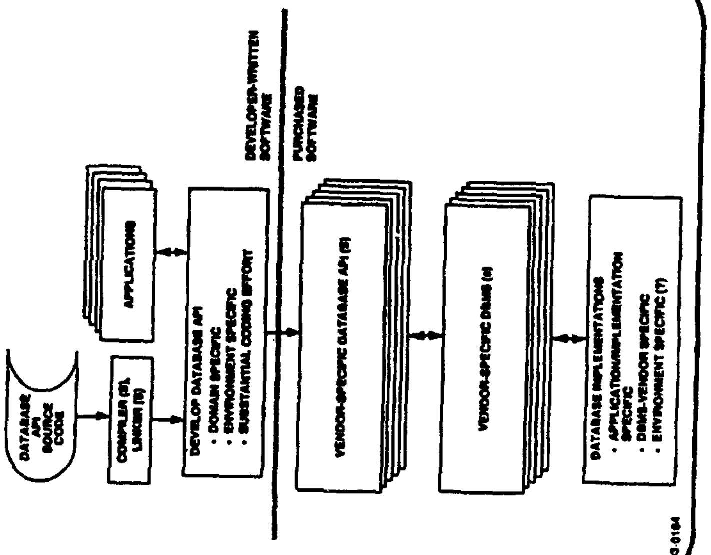
CONCLUSION:

THE APPLICATION BASED SYSTEM ARCHITECTURE PROVIDES A METHODOLOGY FOR ADDRESSING AND SOLVING MANY OF THE ISSUES FACING SWEDEN FOR DEVELOPING A COMPLEX C^3I CAPABILITY

93-0183

DEFENCE MATERIEL ADMINISTRATION

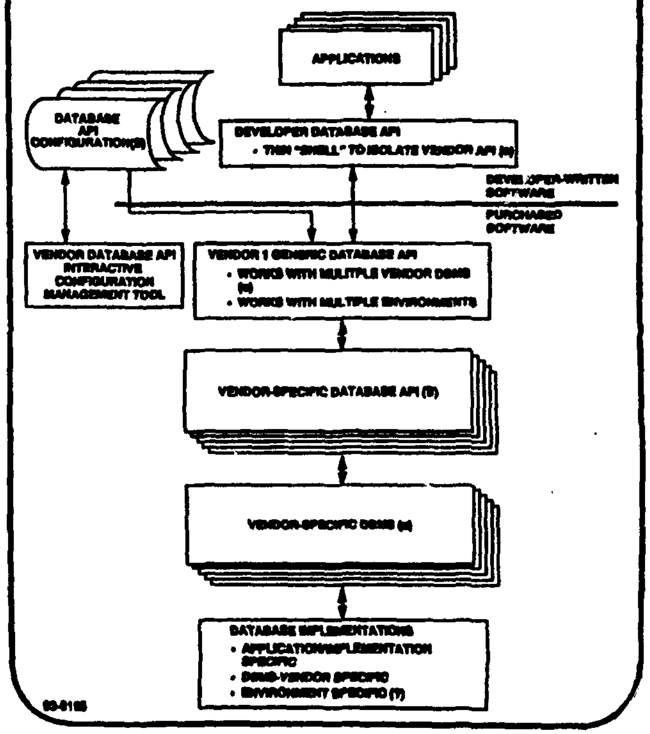
CLASSICAL DEVELOPER-WRITTEN DATABASE APPLICATION PROGRAM INTERFACE (API)



93-0184

DEFENCE MATERIEL ADMINISTRATION

EVOLVING VENDOR-PROVIDED GENERIC DATABASE APPLICATION PROGRAM INTERFACE (API)


APPLICATIONS
DATABASE API CONFIGURATION(S)
DEVELOPER DATABASE API
• THIN "SHELL" TO ISOLATE VENDOR API (n)
DEVELOPER-WRITTEN SOFTWARE
PURCHASED SOFTWARE
VENDOR DATABASE API INTERACTIVE CONFIGURATION MANAGEMENT TOOL
VENDOR 1 GENERIC DATABASE API
• WORKS WITH MULTIPLE VENDOR DBMS (n)
• WORKS WITH MULTIPLE ENVIRONMENTS
VENDOR-SPECIFIC DATABASE API (5)
VENDOR-SPECIFIC DBMS (n)
DATABASE IMPLEMENTATIONS
• APPLICATION/IMPLEMENTATION SPECIFIC
• DBMS-VENDOR SPECIFIC
• ENVIRONMENT SPECIFIC (7)
93-0165

APPENDIX A - PARTICIPANTS

Dr. Dennis Ahern	Westinghouse Electric
Mr. Robert Allen	Carnegie Mellon University
Capt Emily Andrew	National Test Facility
Ms. Rose Armstrong	DSD Laboratories/CARDS
Ms. Pam Arya	General Research
Mr. Ali Babadi	CERC
Major Paul Bailor	Air Force Institiute of Technology/ENG
Mr. James Baldo	Unisys
Mr. Eric Beser	Unisys
Mr. Christopher Bengtsson	C3I, Sweden
Mr. Vincent Bia	National Test Facility
Mr. James Bonine	Design Metrics
Mr. Wayne Brandt	CERC
Ms. Linda Brown	OASD
Mr. J. Chris Commons	ESL, Inc.
Mr. Dick Creps	Unisys
Mr. Paul Dumanoie	DOD/Army STRICOM
Mr. Jim Estep	Unisys/CARDS
Mr. Jeff Facemire	Azimuth/CARDS
Dr. Peter Feiler	Software Engineering Institute/CMU
Ms. Karen Fleming	Strictly Business Computer Systems/CARDS
Ms. Deborah Gary	DISA/CSRO
Mr. Mark Gerhardt	ESL, Inc.
Mr. Mark Gerken	AFIT/ENG
Mr. Terry Gill	National Test Facility
Dr. Robert Gillespie	WVT
Mr. Chandra Gollypudy	CERC
Mr. Nicholay Gradetsky	CERC
Mr. Paul Gregory	Unisys/CARDS
Ms. Kerri Haines	Unisys/CARDS
Ms. Kammi Hefner	Electronic Warfare Associates
Mr. Scott Hissam	Unisys/CARDS
Mr. John James	Intermetrics
Mr. Dan Juttlestadt	NUWC
Mr. Erik Karikoski	Unisys Sweden
Mr. Stellan Karnebro	Syst. Tech.
Mr. Perry Koger	Electronic Warfare Associates/CARDS
Mr. Paul Kogut	Unisys/CARDS
Mr. Jim Law	D.N. American/CARDS
Mr. Roy Lawson	CERC
Mr. Bob Lencewicz	ESD/ENS
Mr. Stanley Levine	CECOM
Mr. Ed Liebhardt II	MountainNet

Mr. Quiang Lin	Galaxy Global Corporation/CARDS
Mr. Bill Loftus	WPL Labs, Inc.
Mr. Pete Maravelias	USAF
Ms. Lorraine Martin	Unisys/CARDS
Mr. Dan McCaugherty	Intermetrics
Mr. Steven Merritt	DISA
Mr. Mike Nichol	ASC/EN(CR)
Mr. Dan Nichols	Electronic Warfare Associates/CARDS
Mr. Ulf Olsson	CelsiusTech
Mr. A. Spencer Peterson	SEI/CMU
Ms. Aleisa Petracca	Azimuth/CARDS
Mr. Jim Petro	Electronic Warfare Associates/CARDS
Mr. Charles Plinta	Accel
Mr. Hans Polzer	Unisys
Mr. Jay Reddy	Strictly Business Computer Systems/CARDS
Mr. Stephen Riesbeck	Azimuth/CARDS
Mr. Steve Roodbeen	NUWC
Mr. Robert Rutherford	SofTech, Inc.
Mr. Skip Saunders	Mitre Corp.
Mr. Evan Schmidt	Electronic Warfare Associates
Mr. Bill Schwartz	DoD
Ms. Jennie Shipe	SofTech, Inc.
Mr. Mark Simos	Organon Motives
Dr. Thomas J. Smith	Mitre Corp.
Ms. Catherine Smotherman	Unisys
Mr. Charlie Snyder	Unisys/CARDS
Mr. Michael Sobolewski	CERC
Dr. Nancy Solderitsch	Unisys/CARDS
Capt Kelly Spicer	SWSC/SMX
Major Frederick Swartz	ASC/YTEC
Mr. Robert Terry	MountainNet
Mr. Will Tracz	IBM FSC
Capt Paul Valdez	ESC/ENS
Mr. Kurt Wallnau	Unisys/CARDS
Mr. Mike Webb	SEI
Mr. Bob Webster	ESC/ENS
Mr. Roger Whitehead	DSD Laboratories/CARDS
Major Grant Wickman	CECOM

Mr. Dennis Ahern
Westinghouse Electric
P.O. Box 746, MS 432
Baltimore, MD 21203-0746

Mr. Robert Allen
CMU
Science Hall 8214
Pittsburgh, PA 15217-3890

Capt. Emily Andrew
National Test Facility
730 Irwin Ave.
Falcon AFB, CO 80912-7300

Ms. Rose Armstrong
DSD
1401 Country Club Rd.
Fairmont, WV 26554

Ms. Pam Arya
General Research
1900 Gallows Road
Vienna, VA 22182

Mr. Ali Babadi
CERC
P.O. Box 6506
Morgantown, WV 26506

Major Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-6583

Mr. James Baldo
CARDS
2010 Sunrise Valley Drive
Reston, VA 22091

Mr. Christopher Bengtsson
C3I
S-115 88 Stockholm
Sweden

Mr. Eric Beser
12344 Greenspring Ave.
Owings Mills, MD 21117

Mr. Vincent Bia
NTF
730 Irwin Ave., MS N9000
Falcon AFB, CO 80909

Mr. James Bonine
Design Metrics
2 Cedar Tree Lane
Stamford, CT 062903

Mr. Wayne Brandt
CERC
P.O. Box 6506
Morgantown, WV 26506

Ms. Linda Brown
OASD
1225 Jefferson Davis Highway
Arlington VA 22202

Mr. J. Chris Commons
ESL, Inc.
495 Java Drive
Sunnyvale, CA 94088-3510

Mr. Dick Creps
Unisys
12010 Sunrise Valley Drive
Reston, VA 22091

Mr. Paul Dumanoie
DoD/Army STRICOM
12350 Research Parkway
Orlando, FL 32826-3276

Mr. Jim Estep
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. Jeff Facemire
Azimuth
1401 Country Club Rd., Suite 204
Fairmont, WV 26554

Dr. Peter Feiler
SEI/CMU
Carnegie Mellon Univ.
Pittsburgh, PA 15213-3890

Ms. Karen Fleming
SBI
12 Moran Circle
Fairmont, WV 26554

Ms. Deborah Gary
CSRO
500 N. Washington St., Suite 200
Falls Church, VA 22046

Mr. Mark Gerhardt
ESL, Inc.
495 Java Drive
Sunnyvale, CA 94088-3510

Mr. Mark Gerken
AFIT/ENG
2950 P. Street
Wright-Patterson AFB, OH 45433-7765

Mr. Terry Gill
National Test Facility
730 Irwin Avenue
Falcon AFB, CO 80912-7300

Dr. Robert Gillespie
WVT
West Virginia Tech
Montgomery, WV 25136

Mr. Chandra Gollypudy
CERC
P.O. Box 6506
Morgantown, WV 26506

Mr. Nicholay Gradetsky
CERC
P.O. Box 6506
Morgantown, WV 26506

Mr. Paul Gregory
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Ms. Kerri Haines
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Ms. Kammi Hefner
EWA
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Scott Hissam
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. John James
Intermetrics

Mr. Dan Juttlestadt
NUWC
Building 1171, 3rd Floor
Newport, RI 02841-4612

Mr. Erik Karikoski
Unisys Sweden

Mr. Stellan Karnebro
Syst. Tech
S-115 88 Stockholm
Sweden

Mr. Perry Koger
EWA
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Paul Kogut
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. Jim Law
DNA
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Roy Lawson
CERC
P.O. Box 6506
Morgantown, WV 26506

Mr. Bob Lencewicz
ESD/ENS
Bldg. 1704
Hanscom AFB, MA 01731-5000

Mr. Stanley Levine
CECOM
710 Carol Avenue
Ocean, NJ 07712

Mr. Ed Liebhardt II
MountainNet
2705 Cranberry Sq.
Morgantown, WV 26505-9286

Mr. Quiang Lin
Galaxy Global
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Bill Loftus
WPL Labs, Inc.
410 Lancaster Ave., Suite 6
Haverford, PA 19041

Mr. Pete Maravelias
USAF
ESD/AVS, Bldg. 1704
Hanscom AFB, MA 01731-5000

Ms. Lorraine Martin
CARDS
4 Militia Dr., Suite 11
Lexington, MA 02173

Mr. Dan McCaugherty
Intermetrics

Mr. Steven Merritt
DISA
500 N. Washington St.
Falls Church, VA 22046

Mr. Mike Nichol
ASC/EN(CR)
1865 4th St., Suite 11
Wright Patterson AFB, Ohio 45433-7126

Mr. Dan Nichols
EWA
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Ulf Olsson
CelsiusTech
S-175 88 Jarfalla
Sweden

Mr. A. Spencer Peterson
SEI/CMU
Carnegie Mellon Univ.
Pittsburgh, PA 15213-3890

Ms. Aleisa Petracca
Azimuth
1401 Country Club Rd., Suite 204
Fairmont, WV 26554

Mr. Jim Petro
EWA
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Charles Plinta
Accel
449 Maple Avenue
Pittsburgh, PA 15218

Mr. Hans Polzer
Unisys
12010 Sunrise Valley Dr.
Reston, VA 22091

Mr. Jay Reddy
SBI
12 Moran Circle
Fairmont, WV 26554

Mr. Stephen Riesbeck
Azimuth
1401 Country Club Rd.
Fairmont, WV 26554

Mr. Steve Roodbeen
NUWC
Bldg. 1171-3, Code 2221
Newport, RI 002841-1708

Mr. Robert Rutherford
SofTech, Inc.
P.O. Box 210386
Montgomery, AL 36121-0386

Mr. Skip Saunders
Mitre Corp.
202 Burlington Rd.
Bedford, MA 01730

Mr. Evan Schmidt
CARDS
1401 Country Club Rd., #201
Fairmont, WV 26554

Ms. Jennie Shipe
SofTech
Alexandria, VA

Mr. Mark Simos
Organon Motives
36 Warwick Road
Watertown, MA 02172

Dr. Thomas J. Smith
Mitre Corp.
752S Colshire Drive MS:W197
McLean, VA 22102

Ms. Catherine Smotherman
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. Charlie Snyder
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. Michael Sobolewski
CERC
P.O. Box 6506
Morgantown, WV 26506

Dr. Nancy Solderitsch
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Captain Kelly Spicer
SWSC/SMX
130 W. Paine St.
Peterson AFB, CO 80914-2320

Major Frederick Swartz
ASC/YTEC
2240 B St., Suite 7
Wright-Patterson AFB, OH 45433-7111

Mr. Robert Terry
MountainNet
2705 Cranberry Sq.
Morgantown, WV 26505

Mr. Will Tracz
IBM, FSC MD 0210
1801 State Route 17c
Owego, NY 13827-3994

Capt. Paul Valdez
ESC/ENS
Bldg. 1704, Rm 107
Hanscom AFB, MA 01731-2116

Mr. Kurt Wallnau
Unisys
1401 Country Club Rd., Suite 102
Fairmont, WV 26554

Mr. Mike Webb
SEI
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Mr. Bob Webster
ESD/ENS, Bldg. 1704
Hanscom AFB, MA 01731-5000

Mr. Roger Whitehead
CARDS
75 Union Avenue
Sudbury, MA 01776

Major Grant Wickman
CECOM
AMSEL-RD-SE-R-ESD-SPT
Ft. Monmouth, NJ 07703

APPENDIX B - BIBILIOGRAPHY

The following sources were used for the development of Seminar materials.

Abowd, et. al., "Structural Modeling: An Application Framework and Development Process for Flight Simulators." Technical Report CMU/SEI-93-TR-14, Software Engineering Institute, 1993.

Air Force Institute of Technology and the Software Engineering Institute, "Putting the Engineering in Software Engineering." Annotated briefing, Carnegie Mellon University.

Alexander, C., "Notes on the Synthesis of Form." Harvard University Press, ISBN 0-674-62750-4, 1964.

Alexander, C., "The Timeless Way of Building." Oxford University Press, ISBN 0-19-502402-8.

Apple Macintosh "MacAPP Developer's Kit Documentation."

Arango, G., Prieto-Diaz, R., "Domain Analysis Concepts and Research Directions." Domain Analysis and Systems Modeling, IEEE Computer Society Press, ISBN 0-8186-8996-X, 1991.

Arango, G., Schoen, E., Pettengill, R., "A Process for Consolidating and Reusing Design Knowledge." Proceedings of The 15th International Conference on Software Engineering, May 17-21, 1993.

Arango, G., Schoen, E., Pettengill, R., "Design as Evolution and Reuse." Proceedings of the Second International Workshop on Software Reusability, March 24-26, 1993.

Arango, G., Schoen, E., Pettengill, R., Hoskins, J., "The Graft-Host Method for Design Change." Proceedings of The 15th International Conference on Software Engineering, May 17-21, 1993.

Balzer, R., "Model Management Examples." Proceedings of DSSA VII Workshop.

Balzer, R., "Design Refinement in DSSAs." Proceedings of the JSGCC Software Initiative Strategy Workshop, December 1992.

Barr, Feigenbaum, Cohen, "The Handbook of Artificial Intelligence." Vols. I-IV, 1981-89.

Batory, D., O'Malley, S., "The Design and Implementation of Hierarchical Software Systems with Reusable Components." Technical Report TR-91-22, University of Texas at Austin, Texas, January 1992.

Booch, G.,"Software Components with Ada." 1987.

Booch, G., "Object-Oriented Design with Applications." 1991.

Booch, G., "Next Generation Methods - Bringing Order Out of the Chaos." Journal of Object Oriented Programming, Supplement on OO Analysis and Design, July/August 1993.

Braun, "DSSAs: Approaches to Specifying and Using Architectures." STARS 92, December 1992.

Bryan, D., Rapide-0.2 Language and Tool-Set Overview. February 1992.

Buschmann, "Rational Architectures For Object-Oriented Software Systems." Journal of Object-Oriented Programming, September 1993.

Callahan, J., Purtillo, J., "A Packaging System for Heterogeneous Execution Environments." IEEE Transactions on Software Engineering, Vol. 17, No. 6, June 1991.

Coad, P., "Object-Oriented Patterns." Communications of the ACM, Vol. 35, No. 9, September 1992.

Commons, J.C., Gerhardt, M., "A Model for Analyzing Megaprogramming, Reuse, and Domain Specific Software Architectures." TRI-Ada, September 1993.

Cox, "Planning the Software Industrial Revolution." IEEE Software, November 1990.

Datapro "Reports on...", updated periodically.

Devanbu, P., Brachman, R.J., Selfridge, P.G., Ballard, B.W., "LaSSIE: A Knowledge-Based Software Information System." Communications of the ACM, May 1991.

Dumas, "Designing User Interfaces for Software." 1988.

Estrin, G., Fenchel, R., Razouk, R., Vernon, M., "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems." IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.

Feiler, P., "Configuration Management Models in Commercial Environments." Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, 1991.

Fischer G., "Human Computer Interaction Software: Lessons Learned, Challenges Ahead." IEEE Software, January 1989.

Freeman, P., "A Conceptual Analysis of the Draco Approach to Constructing Software Systems." IEEE Transactions on Software Engineering, SE-13, July 1987.

Gamma, E., Helm, R., Johnson, R., Vlissides, R., "Design Patterns: Abstraction and Reuse of Object Oriented Design." Unpublished paper. Contact Erich Gamma at Taligent, Inc., 10725 N. De Anza Blvd., Cupertino, CA 95014-2000.

Garlan, D., Shaw, M., "An Introduction to Software Architecture." To appear in Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

Garlan, D., Scott, C., "Adding Implicit Invocation to Traditional Programming Languages." Proceedings of The 15th International Conference on Software Engineering. May 17-21, 1993.

Garlan, D., Kaiser, G.E., Notkin, D., "Using Tool Abstraction to Compose Systems." IEEE Computer, June 1992.

Gelerator, D., Carriero, N., "Coordination Languages and Their Significance." Communications of the ACM, Vol. 35, No. 2, 1992.

Goguen, "Reusing and Interconnecting Software Components."

Griss, M., Informal Presentation Charts. WISR6, November 3-5, 1993.

Harel, D., et.al., "STATEMATE: A Working Environment for the Development of Complex, Reactive Systems." Technical Report, 10th ICSE, 1988.

IEEE Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology. December 1990.

Journal of Object-Oriented Programming, September 1993.

Kazman, R., Bass, L., Abowd, G., Webb, M., "Analyzing Properties of User Interface Software." To be released as a Technical Report, Software Engineering Institute, Carnegie Mellon University.

Knuth, "The Art of Computer Programming." Vols. I-III, 1973.

Krueger, C. W., "Software Reuse." ACM Computing Surveys, Volume 24, Number 2, June 1992.

Lakoff, G., "Women, Fire and Dangerous Things: What Categories Reveal About The Mind." University of Chicago Press, ISBN 0-226-46803-8, 1991.

Lane, T. G., "Studying Software Architectures Through Design Spaces and Rules." Technical Report CMU/SEI-90-TR-18, Software Engineering Institute, 1990.

Lee, Rissman, D'Ippolito, Plinta, Van Scoy, "An OOD Paradigm for Flight Simulators." Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, 1988.

Lowry, "Software Engineering in the Twenty-First Century." AI Magazine, Fall 1992.

Lowry, M. R., McCartney, R. D., "Automating Software Design." AAAI Press, 1991.

Lubars, M.D., "A General Design Representation." Technical Report STP-066-89, MCC Corp., Austin, Texas, 1989.

Lubars, M. D., "Representing Design Dependencies in an Issue-Based Style." IEEE Software, July 1991.

Luckham, D.C., von Henke, "An Overview of Anna: a Specification Language for Ada." IEEE Software, March 1985.

Luckham, D.C., Vera, J., "μRapide: An Executable Architecture Definition Language." April 1993.

Luckham, D.C., Vera, J., "Event-Based Concepts and Language for System Architecture." March 1993.

Metalla, E.,"Domain-Specific Software Architectures." STARS 92 Annotated Briefing, 1992.

Mettala, E., "The Domain Specific Software Architecture Program." DARPA Software Technology Conference, April 1992.

Meyer, B., "Object-Oriented Software Construction." Prentice-Hall, 1988.

Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components." IEEE Transaction on Software Engineering, SE-10, September 1984.

Neighbors, J.M., "Draco: A Method for Engineering Reusable Software Systems." Frontier Series: Software Reusability: Volume I - Concepts and Models, ACM Press, 1989.

Neighbors, J.M., "Draco: The Evolution From Software Components to Domain Analysis." International Journal of Software Engineering and Knowledge Engineering. Vol. 2, No. 3, September 1992.

Nierstrasz, O., Gibbs, Tsichritzis, "Component Oriented Software Development." Communications of the ACM, Vol. 35, No. 9, September 1992.

OMG, "The Common Object Request Broker: Architecture and Specification." 1992.

OMG, "Object Management Architecture Guide." September 1992.

Patel-Schneider, P.F., Brachman, R.J., Levesque., H.J., "Argon: Knowledge Representation Meets Information Retrieval." Proceedings of the First Conference on Artificial Intelligence Applications, 1984.

Payton, T., "Domain-Specific Reuse." STARS 92 Annotated Briefing, 1992.

Perry, Chilton, "Chemical Engineers' Handbook." 5th ed., 1973.

Perry, D.E., Wolf, A., "Foundations for the Study of Software Architecture." ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4, October 1992.

Peterson, S., "Mapping a Domain Model and Architecture to a Generic Design." CMU/SEI-Technical Report, draft.

Peterson, S., "Coming to Terms with Software Reuse Terminology: A Model-Based Approach." ACM SIGSOFT Software Engineering Notes, April 1991.

Purtilo, J., "Software Bus Organization: Reference Model and Comparison of Two Existing Systems." ARPA Module Interconnection Formalism Working Group Technical Note Series, TN No. 8, November 1991.

Royce, W., Brown, D., "Architecting Distributed Realtime Ada Applications: The Software Architect's Lifecycle Environment." Ada IX, 1991.

Salasin, J., Waugh, D., "An Approach to Analyzing Non-Functional Aspects During System Definition." Draft Technical Paper, Proceedings of the ARPA/DSSA VII Workshop.

Saunders, Horowitz, Mleziva, "A New Process for Acquiring Software Architecture." MITRE Corporation, 1993.

Sedgewick, "Algorithms in C." 1990.

Sedgewick, "Algorithms in C++." 1992.

Selfridge, P.G., "Knowledge Representation Support for a Software Information System." Proceedings of the 7th Conference on Artificial Intelligence Applications, February 24-28, 1991.

Selfridge, P.G., Terveen, L.G., Long, M.D., "Managing Design Knowledge to Provide Assistance to Large-Scale Software Development." Proceedings of the 7th Knowledge-Based Software Engineering Conference, September 1992.

Shaw, M. "Prospects for an Engineering Discipline of Software." IEEE Software, November 1990.

Shaw, M., "Larger Scale Systems Require Higher Level Abstractions." 5th International Workshop on Software Specification and Design, May 1989.

Simos, M., "Organizational Domain Modeling." STARS Technical Report, Unisys Corporation.

Singhal, V., Batory, D., "P++: A Language for Software System Generators." Technical Report TR-93-16, Department of Computer Science, University of Texas at Austin, 1993.

Taft, "Ada 9X: A Technical Summary." Communications of the ACM, November 1992.

Tracz, W., "LILEANNA: A Parameterized Programming Language." Proceedings of the Second International Workshop on Software Reusability, March 24-26, 1993.

Tracz, W., "A Conceptual Model for Megaprogramming." ACM SIGSOFT Software Engineering Notes, July 1991.

UNAS Training Class, TRW Systems Engineering & Development Division, DH2/1271, Carson, CA, July 7-9, 1993.

Zachman, J., "A Framework for Information Systems Architecture." IBM Systems Journal, Vol 26, No. 3, 1987.

The following sources are recommended for those interested in additional information.

Agrawala, Jackson, Vestal, "Domain-Specific Software Architectures for Intelligent Guidance, Navigation and Control." DARPA Software Technology Conference, April 1992.

Bailin, S., "KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales." Unpublished, 1993.

Belz, Luckham, Purtilo, "Application of ProtoTech Technology to the DSSA Program." DARPA Software Technology Conference, April 1992.

Bhansali, Nii, "Software Design by Reusing Architectures." Proceedings of the 7th Knowledge-Based Software Engineering Conference, September 1992.

Braun, Hatch, Ruegsegger, Balzer, Feather, Goldman, Wile, "Domain Specific Software Architectures - Command and Control." DARPA Software Technology Conference, April 1992.

Coglianese, Goodwin, Smith, Tracz, Batory, Bellman, Gries, McAllester, Selby, Taylor, "An Avionics Domain-Specific Software Architecture." DARPA Software Technology Conference, April 1992.

Coglianese, Tracz, Newton, McAllester, Goguen, Taylor, Selby, Batory, "DSSA-ADAGE." DSSA VII Briefing, July 1993.

Dasgupta, S., "A Hierarchical Taxonomic System for Computer Architectures." IEEE Computer, March 1990.

Davis, A., "A Comparison of Techniques for the Specification of External System Behavior." CACM, September 1988.

Fichman, Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique." IEEE Computer, October 1992.

Fowler, M., "OO Methods: A Comparative View." Journal of Object Oriented Programming, Supplement on OO Analysis and Design, July/August 1993.

Graham, I., "Object-Oriented Methods." Addison Wesley, 1991.

Gruber, T., "Toward principles for the design of ontologies used for knowledge sharing." Unpublished report, January 1993.

Guindon, R., "The Knowledge Exploited by Experts During Software System Design." MCC STP-032-90, 1990.

Hayes-Roth, F., Erman, Terry, Hayes-Roth, B., "DSSA: Distributed Intelligent Control and Management Applications and Development Support Environment." DARPA Software Technology Conference, April 1992.

Jullig, R., "Applying Formal Software Synthesis." IEEE Software, May 1993.

Lalum, C., "Analysis of DCDS Data Model." STARS CDRL 3048R, January 1991.

Lee, J., "The 1992 Workshop on Design Rationale Capture and Use." AI Magazine, Summer 1993.

Long, Morris, "An Overview of PCTE: A Basis for a Portable Common Tool Environment." CMU/SEI-TR-93-1, 1993.

Lubars, M., "The ROSE-2 Strategies for Supporting High Level Software Design Reuse." Automating Software Design, 1991.

Lubars, M., "Domain Specific Software Architectures." MCC STP-RU-043-91, February 1991.

Meadow, C. L., Latour, L., "Layered Generic Architectures: A Methodology for the Construction of Reusable Software Components." Prepared for the US Army CECOM Center for Software Engineering, July 1991.

Monarchi, Puhr, "A Research Typology for Object-Oriented Analysis and Design." CACM, September 1992.

Neches, Fikes, Finin, Gruber, Patil, Senator, Swartout, "Enabling Technology for Knowledge Sharing." AI Magazine, Fall 1991.

Platek, R., "DSSA's for Hybrid Control." DARPA Software Technology Conference, April 1992.

Schwanke, Altucher, Platoff, "Discovering, Visualizing, and Controlling Software Structure." 5th International Workshop on Software Specification and Design, May 1989.

Software Technology Support Center, "Requirements Analysis and Design Tools Report." April 1992.

Tracz, Coglianese, Young, "Domain-specific SW Architecture Engineering." ACM SIGSOFT Software Engineering Notes, October 1992.

Tracz, W., "Megaprogramming and Domain Engineering Tutorial." ICSE 15, May 1993.

Tracz, Shafer, Coglianese, "DSSA-ADAGE Design Records." ADAGE-IBM-93-05, July 1993.

Vestal, S., "A Cursory Overview and Comparison of Four Architectural Description Languages." Informal technical report, February 1993.

Vestal, S., "Host Environment Support for Architecture-Oriented Toolsets." Informal technical report, March 1993.

Webster, D., "Mapping the Design Information Representation Terrain." IEEE Computer, December 1986.

Wiederhold, Wegner, Ceri, "Toward Megaprogramming." CACM, November 1992.

Wood, Pethia, Gold, Firth, "A Guide to the Assessment of Software Development Methods." Technical Report CMU/SEI-88-TR-8, 1988.