Technical Report
CMU/SEI-96-TR-003
ESC-TR-96-003

# Software Architecture: An Executive Overview

Paul C. Clements
Linda M. Northrop

February 1996

Technical Report
CMU/SEI-96-TR-003
ESC-TR-96-003
February 1996

# Software Architecture: An Executive Overview

Paul C. Clements
Linda M. Northrop

Product Lines Systems



**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.



This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

# Table of Contents

## List of Figures

**Abstract:** Software architecture is an area of growing importance to practitioners and researchers in government, industry, and academia. Journals and international workshops are devoted to it. Working groups are formed to study it. Textbooks are emerging about it. The government is investing in the development of software architectures as core products in their own right. Industry is marketing architectural frameworks such as CORBA. Why all the interest and investment? What is software architecture, and why is it perceived as providing a solution to the inherent difficulty in designing and developing large, complex systems? This report will attempt to summarize the concept of software architecture for an intended audience of mid to senior level management. The reader is presumed to have some familiarity with common software engineering terms and concepts, but not to have a deep background in the field. This report is not intended to be overly-scholarly, nor is it intended to provide the technical depth necessary for practitioners and technologists. The intent is to distill some of the technical detail and provide a high level overview.

# 1. Introduction

Software architecture is an area of growing importance to practitioners and researchers in government, industry, and academia. The April 1995 issue of *IEEE Transactions on Software Engineering* and the November 1995 issue of *IEEE Software* were devoted to software architecture. Industry and government working groups on software architecture are becoming more frequent. Workshops and presentations on software architecture are beginning to populate software engineering conferences. There is an emerging software architecture research community, meeting and collaborating at special-purpose workshops such as the February 1995 International Workshop on Software Architectures held in Dagstuhl, Germany, or the April 1995 International Workshop on Architectures for Software Systems held in Seattle, Washington. The October 1996 ACM Symposium on the Foundations of Software Engineering will focus on software architecture. Textbooks devoted entirely to software architecture are appearing, such as the one by Shaw and Garlan [Shaw 95b]. The government is investing in the development of software architectures as core products in their own right; the Technical Architecture Framework for Information Management (TAFIM) is an example. The Common Object Request Broker Architecture (CORBA) and other computer-assisted software engineering environments with emphasis on architecture-based development are entering the marketplace with profound effect.

Why all the interest and investment? What is software architecture, and why is it perceived as providing a solution to the inherent difficulty in designing and developing large, complex systems?

This report will attempt to summarize the concept of software architecture for an intended audience of mid to senior level management. The reader is presumed to have some familiarity with common software engineering terms and concepts, but not to have a deep background in the field. This report is not intended to be overly-scholarly, nor is it intended to provide the technical depth necessary for practitioners and technologists. Software engineers can refer to the listed references for a more comprehensive and technical presentation. The intent here is to distill some of the technical detail and provide a high level overview.

Because software architecture is still relatively immature from both a research and practice perspective there is little consensus on terminology, representation or methodology. An accurate yet digested portrayal is difficult to achieve. All of the issues and all of the ambiguity in the area of software architecture have yet to be addressed. We have simplified based upon what we believe to be the best current understanding.

While software architecture appears to be an area of great promise, it is also an area ripe for significant investment in order to reach a level of understanding from which significant benefits can be reaped and from which a truly simple overview could be captured.

We invite feedback on the content, presentation, and utility of this report with regard to the intended audience.

The structure of the report is as follows:

- Section 2 discusses the concept of software architecture—its definition(s), its history, and its foundational underpinnings. It also suggests why there has been considerable confusion over the term and why we do not yet have a precise definition.
- Section 3 asks, and attempts to answer, the question “Why is software architecture important?” It discusses the importance of the concept of software architecture in system development from three vantages: as a medium of communication among a project’s various stakeholders; as the earliest set of design decisions in a project; and as a high-level abstraction of the system that can be reused in other systems.
- Section 4 discusses the concept of architectural views—the need for different views, a description of some accepted views, and the relationship among views.
- Section 5 explains how the architecture-based model of system development differs from the traditional programming-oriented development paradigms of the past.
- Finally, Section 6 lists some of the most promising research areas in software architecture.

# 2. What Is Software Architecture?

## 2.1 Definitions

What do we mean by software architecture? Unfortunately, there is yet no single universally accepted definition. Nor is there a shortage of proposed definition candidates. The term is interpreted and defined in many different ways. At the essence of all the discussion about software architecture, however, is a focus on reasoning about the *structural* issues of a system. And although architecture is sometimes used to mean a certain architectural style, such as client-server, and sometimes used to refer to a field of study, it is most often used to describe structural aspects of a particular system.

These structural issues are design-related—software architecture is, after all, a form of software design that occurs earliest in a system's creation—but at a more abstract level than algorithms and data structures. According to what has come to be regarded as a seminal paper on software architecture, Mary Shaw and David Garlan suggest that these

"Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives" [Garlan 93].

Each of the various definitions of software architecture emphasizes certain of these structural issues and corresponding ways to describe them. Each of these positions can usually be traced to an idea about what the proponent wishes to *do* with the software architecture—analyze it, evolve it, represent it, or develop from it. It is important to understand that though it may seem confusing to have multiple interpretations, these different interpretations do not preclude each other, nor do they represent a fundamental conflict about what software architecture is. We will address these different interpretations or views in Section 3. However, at this point it is important to realize that together they represent a spectrum in the software architecture research community about the emphasis that should be placed on architecture—its constituent parts, the whole entity, the way it behaves once built, or the building of it. Taken together, they form a consensus view of software architecture and afford a more complete picture.

As a sufficiently good compromise to the current technical debate, we offer the definition of software architecture that David Garlan and Dewayne Perry have adopted for their guest editorial in the April 1995 *IEEE Transactions on Software Engineering* devoted to software architecture:

*The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

Other definitions can be found in various documents [Perry 92, Garlan 93, Hayes-Roth 94, Gacek 95, Soni 95]. Diagrams are typically used to illustrate these components and their interrelationships. The choice of diagram is by no means standardized.

The bottom line is that software architecture is about structural properties of a system. Structural properties can be expressed in terms of components, interrelationships, and principles and guidelines about their use. The exact structural properties to consider and the ways to represent them vary depending upon what is of structural interest to the consumer of the architecture.

## 2.2 Roots of Software Architecture

The study of software architecture is in large part a study of software structure that began in 1968 when Edsger Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result [Dijkstra 68]. Dijkstra was writing about an operating system, and first put forth the notion of a layered structure, in which programs were grouped into layers, and programs in one layer could only communicate with programs in adjoining layers. Dijkstra pointed out the elegant conceptual integrity exhibited by such an organization, with the resulting gains in development and maintenance ease.

David Parnas pressed this line of observation with his contributions concerning information-hiding modules [Parnas 72], software structures [Parnas 74], and program families [Parnas 76].

A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for which it is profitable or useful to consider as a group. This avoids ambiguous concepts such as “similar functionality” that sometimes arise when describing domains. For example, software engineering environments and video games are not usually considered to be in the same domain, although they might be considered members of the same program family in a discussion about tools that help build graphical user interfaces, which both happen to use.[1]

[1] This example illustrates that the members of a program family may include elements of what are usually considered different domains.

Parnas argued that early design decisions should be ones that will most likely remain constant across members of the program family that one may reasonably expect to produce. In the context of this discussion, an early design decision is the adoption of a particular architecture. Late design decisions should represent trivially-changeable decisions, such as the values of compile-time or even load-time constants.

All of the work in the field of software architecture may be seen as evolving towards a paradigm of software development based on principles of architecture, and for exactly the same reasons given by Dijkstra and Parnas: Structure is important, and getting the structure right carries benefits.

In tandem with this important academic understanding of program and system structure came a long series of practical experiences working with systems in several highly populated domains, such as compilers. Throughout the 1970s and 1980s, compiler design evolved from a series of distinct efforts, each one innovative and unprecedented, into one with standard, codified pieces and interactions. Today, textbooks about how to build a compiler abound, and the domain has matured to the point where no one today would think for a moment of building a compiler from scratch, without re-using and exploiting the codified experience of the hundreds of prior examples.

What exactly is reused and exploited? Those structural necessities that are common to all compilers. Compiler writers can talk meaningfully with each other about lexical scanners, parsers, syntax trees, attribute grammars, target code generators, optimizers, and call graphs even though the languages being compiled may look nothing at all alike. So, for instance, two compilers may have completely different parsers, but what is common is that both compilers *have* a component called a parser, which performs a function in both that (when viewed under at a high level) is exactly the same. Reusing the structural decisions and componentry for a system also allows reusing its work breakdown structures, estimates, team organization, test plans, integration plans, documentation, and many other labor-intensive assets.

Many other domains now exist that, through practice and repetition and sharing among the many early members of the family, now exhibit common structure, interconnection strategies, allocation of functionality to components, component interfaces, and an overall justifying rationale. The current study of software architecture can be viewed as an *ex post facto* effort to provide a structured storehouse for this type of reusable high level family-wide design information. Work in software architecture can be seen as attempting to codify the structural commonality among members of a program family, so that the high-level design decisions inherent in each member of a program family need not be re-invented, re-validated, and re-described.

## 2.3 Why Hasn't the Community Converged?

As noted above, the software engineering community has not settled on a universal definition for software architecture. The lack of a definition is perhaps not as significant as the reasons for lack of convergence. We suggest the following reasons for the current ambiguity in the term. We list them as issues to be aware of in any discussion of software architecture.

### 2.3.1 Advocates Bring Their Methodological Biases with Them

As we have noted, proposed definitions of architecture largely agree at the core, but differ seriously at the fringes. Some require that architecture must include rationale, others hold out for process steps for construction. Some require allocation of functionality to components; others contend that simple topology suffices. Each position depends upon the precise motivation for examining the structural issues in the first place. It is essential to understand that motivation prior to the study of either a definition of software architecture or an architecture artifact.

### 2.3.2 The Study Is Following Practice, not Leading It

The study of software architecture has evolved by observing the design principles and actions that designers take when working on real systems. It is an attempt to abstract the commonalities inherent in system design, and as such, it must account for a wide range of activities, concepts, methods, approaches, and results. This differs from a more top-down approach that defines software architecture and then maps compliant ongoing activities to the term. What we see happening is that people observe designers' many activities, and try to accommodate those activities by making the term software architecture more broad. Because this study is ongoing, the convergence of the definition hasn't happened.

### 2.3.3 The Study Is Quite New

Although it possesses long roots, the field of software architecture is really quite new, as judged by the recent flurry of books, conferences, workshops, and literature devoted to it.

### 2.3.4 The Foundations Have Been Imprecise

Beware: The field has been remarkable for its proliferation of undefined terms that can be land mines for the unsuspecting. For example, architecture defined as "the overall structure of the system" adds to rather than reduces confusion because this implies that a system has but a single "overall structure." Figure 1, taken from a system description for an underwater acoustic simulation system, purports to describe the top-level architecture of the system. Exactly what can we tell about the system

from this diagram? There are four components, three of which might have more in common with each other (MODP, MODR, and MODN) than with the fourth (CP).

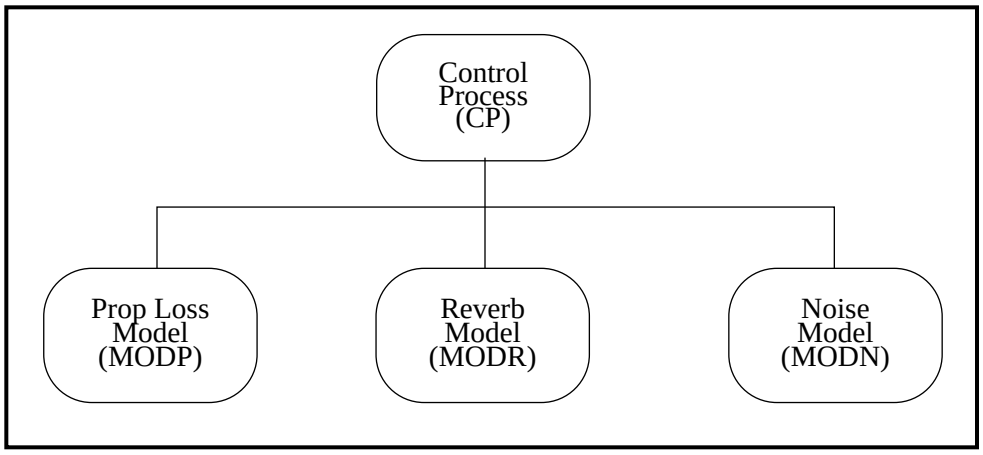


**Figure 1: Typical, but Uninformative, Model of a "Top-Level Architecture"**

Is this an architecture? Assuming that architecture consists of components (of which we have four) and connections among them (also present), then this would seem to suffice according to many definitions. However, even if we accept the most primitive definition, what can we *not* tell from the diagram?

- What is the nature of the components, and what is the significance of their separation? Do they run on separate processors? Do they run at separate times? Do the components consist of processes, programs, or both? Do the components represent ways in which the project labor will be divided, or do they convey a sense of runtime separation? Are they modules, objects, tasks, functions, processes, distributed programs, or something else?
- What is the significance of the links? Do the links mean the components communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, or some combination of these or other relations?
- What is the significance of the layout? Why is CP on a separate (higher) level? Does it call the other three components, and are the others not allowed to call it? Or was there simply not room enough to put all four components on the same row in the diagram?

We *must* raise these questions, for without knowing precisely what the components are, what the links mean, and what significance there is to the position of components and/or direction of links, diagrams are not much help and should be regarded warily.

Consider one more example, a "layered architecture," which is also a commonly-represented architectural paradigm [Garlan 93]:

*A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy... The connectors are defined by the protocols that determine how the layers will interact.*

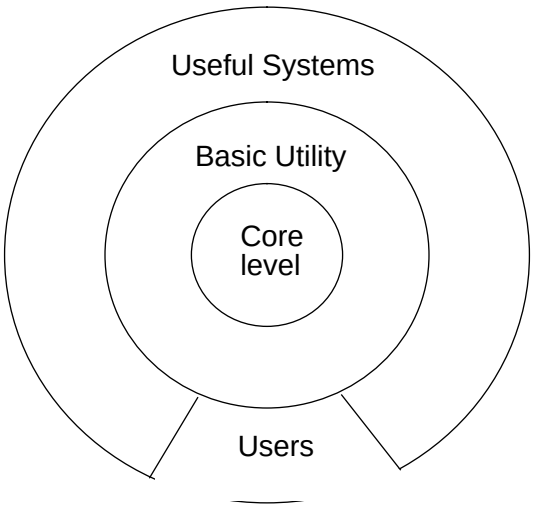


**Figure 2: A Layered System [Garlan93]**

Close examination of this description reveals that it mixes separate concerns. For one thing, “hidden” is a concept that has no meaning at runtime; it is purely a concept that applies at program-write time, and specifies what facilities a particular programmer is or is not allowed to use when writing his or her portion of the system. “Providing service” is the runtime interaction mechanism, but it could reasonably mean any of the following: calls, uses, signals, sends data to. It also fails to capture any notion of concurrency, real or potential. Can software in different layers run simultaneously, or are there mutual exclusion constraints between layers? If we are concerned about the feasibility of fielding our system on a multi-processor environment, shouldn’t we be able to discern this information as part of the answer to the question “What is the architecture of a layered system?”

### 2.3.5 The Term Is Over-Utilized

The meaning of the term architecture as it relates to software engineering is becoming increasingly dilute simply because it seems to be in vogue. It is possible to find references to the following “kinds” of architectures: domain-specific, megaprogramming, target, systems, information, information systems, functional, software, hardware, network, infrastructure, applications, operations, technical, framework,

conceptual, reference, enterprise, factory, C4I, manufacturing, building, machine-tool, etc. Often what differs is the nature of the components and connections (e.g., a network architecture specifies connectedness between processors at the endpoints of hardware communication paths); at other times the distinctions are less clear or the term “architecture” is simply an inappropriate reference. Some of these terms will be described in more detail in Section 4.

## 2.4 The Many Roles of Software Architecture

People often make analogies to other uses of the word architecture about which they have some intuition. They commonly associate architecture with physical structure (building, streets, hardware) and physical arrangement. A building architect has a perspective of architecture that is driven by the need to design a building that as an entity addresses needs and requirements including accessibility, aesthetics, light, maintainability, etc. [Alexander 77]. A software architect has a perspective that is driven by the need to design a system that addresses needs such as concurrency, portability, evolvability, usability, security, etc. Analogies between buildings and software systems should not be taken literally—they break down fairly soon—but rather used to help understand that perspective is important and structure can have different meanings depending upon the motivation for examining structure. What to glean from this discussion is that a precise definition of software architecture is not nearly as important as the concept and what its investigation allows us to do.

Software architecture usually refers to some combination of structural views of a system, with each view a legitimate abstraction of the system with respect to certain criteria, that facilitates a particular type of planning or analysis. This relatively simple concept has been co-opted by a wide variety of stakeholders and participants in software development; architecture has become a concept that represents many things to many people. In subsequent sections, we will explore some of these views and viewpoints.

# 3. Why Is Software Architecture Important?

> *If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.*
>
> — *Barry Boehm* [Boehm 95]

What is it about software architecture that warrants all the attention it is receiving? In this section we will suggest some reasons why software architecture is important, and why the practice of architecture-based development is worthwhile.

Fundamentally, there are three reasons:

1. **Mutual communication.** Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.
2. **Early design decisions**. Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life.
3. **Transferable abstraction of a system**. Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

We will address each in turn.

## 3.1 Architecture Is the Vehicle for Stakeholder Communication

Each stakeholder of a software system—customer, user, project manager, coder, tester, etc.—is concerned with different aspects of the system for which architecture is an important factor; or, they may be concerned with the same aspects, but from different perspectives. For example, the user is concerned that the system meets its availability and reliability requirements; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried (in addition to cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways. The developer is worried about strategies to achieve all of those goals. Architecture provides a common lan-

guage in which competing concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language it is difficult to understand large systems sufficiently to make well informed early decisions that greatly influence their quality and usefulness.

## 3.2 Architecture Embodies the Earliest Set of Design Decisions About a System

Architecture represents the earliest set of design decisions about a system. These early decisions are the most difficult to get right, are the hardest ones to change, and have the most far-reaching downstream effects, some of which we describe as follows.

### 3.2.1 Architecture Provides Builders with Constraints on Implementation

An architecture defines a set of constraints on an implementation; an implementation is said to exhibit an architecture if it conforms to the structural design decisions described by the architecture. The implementation must therefore be divided into the prescribed components, the components must interact with each other in the prescribed fashion, and each component must fulfill its responsibility to the other components as dictated by the architecture.

This constraining of the implementation is made on the basis of system- and/or project-wide allocation decisions that are invisible to implementors working on individual components, and permits a separation of concerns that allows management decisions that make best use of personnel. Component builders must be fluent in the specification of their individual components, but not in system trade-off issues; conversely, the architects need not be experts in algorithm design or the intricacies of the programming language.

### 3.2.2 The Architecture Dictates Organizational Structure for Development and Maintenance Projects

Not only does architecture prescribe the structure of the system being developed, but that structure becomes reflected in the work breakdown structure and hence the inherent development project structure. Teams communicate with each other in terms of the interface specifications to the major components. The maintenance activity, when launched, will also reflect the software structure, with maintenance teams formed to address specific structural components.

### 3.2.3 An Architecture Permits or Precludes the Achievement of a System's Targeted Quality Attributes

Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen.

Quality attributes may be divided into two categories. The first includes those that can be measured by running the software and observing its effects; performance, security, reliability, and functionality all fall into this category. The second includes those that cannot be measured by observing the system, but rather by observing the development or maintenance activities. This category includes maintainability in all of its various flavors: adaptability, portability, reusability, and the like.

Modifiability, for example, depends extensively on the system's modularization, which reflects the encapsulation strategies. Reusability of components depends on how strongly coupled they are with other components in the system. Performance depends largely upon the volume and complexity of inter-component communication and coordination, especially if the components are physically distributed processes.

It is important to understand, however, that an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an architectural framework. Decisions at all stages of the life cycle—from high level design to coding and implementation—affect system quality. Therefore, quality is not completely a function of an architectural design. A good architecture is necessary, but not sufficient, to ensure quality.

### 3.2.4 It Is Possible to Predict Certain Qualities About a System by Studying Its Architecture

If the architecture allows or precludes a system's quality attributes (but cannot ensure them), is it possible to tell that the appropriate architectural decisions have been made without waiting until the system is developed and deployed? If the answer were "no," then choosing an architecture would be a fairly hopeless task. Random architecture selection would perform as well as any other method. Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture.

Architecture evaluation techniques such as the Software Architecture Analysis Method (SAAM), proposed at the Software Engineering Institute (SEI), obtain top down insight into the attributes of software product quality that are enabled (and constrained) by specific software architectures. SAAM proceeds from construction of a set of domain derived scenarios that reflect qualities of interest in the end-product software. This set includes direct scenarios (which exercise required software functionality) and indirect scenarios (which reflect non functional qualities). Mappings are

made between these domain scenarios and candidate architectures, and a score is assigned to the degree by which a candidate architecture satisfies the expectations of each scenario. Candidate architectures can then be contrasted in terms of their fulfillment of scenario-based expectations of them [Abowd 94, Clements 95a, Kazman 95].

### 3.2.5 Architecture Can Be the Basis for Training

The structure, plus a high-level description of how the components interact with each other to carry out the required behavior, often serves as the high-level introduction to the system for new project members.

### 3.2.6 An Architecture Helps to Reason About and Manage Change

The software development community is finally coming to grips with the fact that roughly 80% of a software system's cost may occur *after* initial deployment, in what is usually called the maintenance phase. Software systems change over their lifetimes; they do so often, and often with difficulty. Change may come from various quarters, including:

- The need to enhance the system's capabilities. Software-intensive systems tend to use software as the means to achieve additional or modified functionality for the system as a whole. Systems such as the Joint Stars battlefield surveillance radar and the MILSTAR network of telecommunication satellites are examples of systems that have achieved enhanced capability through software upgrades. However, with each successive change, the complexity of the system software has increased dramatically.
- The need to incorporate new technology, whose adoption can provide increased efficiency, operational robustness, and maintainability.

Deciding when changes are essential, determining which change paths have least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, dependencies, performance, and behavioral aspects of system software components. Reasoning at an architecture level can provide the insight necessary to make decisions and plans related to change.

More fundamentally, however, an architecture partitions possible changes into three categories: local, non-local, and architectural. A local change can be accomplished by modifying a single component. A non-local change requires multiple component modifications, but leaves the underlying architecture intact. An architectural change affects the ways in which the components interact with each other, and will probably require changes all over the system. Obviously, local changes are the most desirable,

and so the architecture carries the burden of making sure that the most likely changes are also the easiest to make.

## 3.3 Architecture as a Transferable Model

Greater benefit can be achieved from reuse the earlier in the life cycle it is applied. While code reuse provides a benefit, reuse at the architectural level provides a tremendous leverage for systems with similar requirements. When architectural decisions can be reused across multiple systems all of the early decision impacts we just described above are also transferred.

### 3.3.1 Entire Product Lines Share a Common Architecture

Product lines are derived from what Parnas referred to in 1976 as program families [Parnas 76]. It pays to carefully order the design decisions one makes, so that those most likely to be changed occur latest in the process. In an architecture-based development of a product line, the architecture is in fact the sum of those early design decisions, and one chooses an architecture (or a family of closely-related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early, and by making others that apply only to individual members late. The architecture defines what is fixed for all members of the product line and what is variable.

A family-wide design solution may not be optimal for all derived systems, but it is a corporate decision that the quality known to be associated with the architecture and the savings in labor earned through architectural-level reuse compensates for the loss of optimality in particular areas. The architecture for a product line becomes a developing organization's core asset, much the same as other capital investments.

The term *domain-specific software architectures* applies to architectures designed to address the known architectural abstractions specific to given problem domains. Examples of published domain-specific software architectures come from the ARPA Domain-Specific Software Architecture (DSSA) program [Hayes-Roth 94].

### 3.3.2 Systems Can Be Built by Importing Large Externally-Developed Components That Are Compatible with a Pre-Defined Architecture

Whereas former software paradigms have focused on *programming* as the prime activity, with progress measured in lines of code, architecture-based development often focuses on *composing or assembling components* that are likely to have been developed separately, even independently, from each other. This composition is possible because the architecture defines the set of components that can be incorporated into the system. The architecture constrains possible replacement (or additions)

in the way in which they interact with their environment, how they receive and relinquish control, the data that they work on and produce and how they access it, and the protocols they use for communication and resource sharing.

One key aspect of architecture is its organization of component structure, interfaces, and operating concepts. One essential value of this organization is the idea of interchangeability. In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, announced the dawn of the industrial age. In the days before physical measurements were reliable, this was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting, and just as significant. Commercial off-the-shelf components, subsystems, and compatible communications interfaces all depend on the idea of interchangeability.

There are still some significant unresolved issues, however, related to software development through composition. When the components that are candidates for importation and reuse are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications may increase the effort required to integrate their functions. David Garlan has coined the term "architectural mismatch" to describe this situation [Garlan 95]. Symptoms of architectural mismatch are the serious integration problems that occur when developers of independent subsystems have made architectural assumptions that differed from the assumptions of those who would employ these subsystems.

To resolve these differences, Garlan identifies the need to make explicit the architectural contexts for potentially reusable subsystems. Some design practices, such as information hiding, are particularly important for architectural consistency. Techniques and tools for developing wrappers[1] to bridge mismatches, and principles for composition of software are also needed. The most elemental need is for improved documentation practices, the inclusion of detailed pre-conditions for the use of interfaces, and conventions for describing typical architectural assumptions.

### 3.3.3 Architecture Permits the Functionality of a Component to be Separated from Its Component Interconnection Mechanisms

Traditional design approaches have been primarily concerned with the functionality of components. Architecture work seeks to elevate component relationships to the same

1. A wrapper is a small piece of software that provides a more usable or appropriate interface for a software component. Users of the component invoke it through the wrapper, which translates the invocation into the form required by the component. The wrapper "hides" the (less desirable) interface of the component, so that only the wrapper software has to deal with it.

level of concern. How components interact (coordinate, cooperate, communicate) becomes a first class design decision where the stated goal is to recognize the different fundamental qualities imparted to systems by these various interconnection strategies, and to encourage informed choices. The result is a separation of concerns, which introduces the possibility of building architectural infrastructure to automatically implement the architect's eventual choice of mechanism. The binding of this decision may be delayed and/or easily changed. Thus, prototyping and large-scale system evolution are both supported. Although proponents of this view speak of "first-class connectors" [Shaw 95c], they are actually making it possible for the question of connectors to be ignored in many cases. This contrasts to the programming paradigm, where connection mechanisms are chosen very early in the design cycle, are not given much thought, and are nearly impossible to change. Areas addressing this aspect include architecture description languages that embody connection *abstractions*, as opposed to mechanisms.

### 3.3.4 Less Is More: It Pays to Restrict the Vocabulary of Design Alternatives

Garlan and Shaw's work in identifying architectural styles [Garlan 93] teaches us that although computer programs may be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small set of choices when it comes to program cooperation and interaction. Advantages include enhanced reuse, more capable analysis, shorter selection time, and greater interoperability.

Properties of software design follow from the choice of *architectural style*. Architectural styles are patterns or design idioms that guide the organization of modules and subsystems into complete systems. Those styles that are more desirable for a particular problem should improve implementation of the resulting design solution, perhaps by enabling easier arbitration of conflicting design constraints, by increasing insight into poorly understood design contexts, and/or by helping to surface inconsistencies in requirements specifications. Client-server[1] and pipe-filter[2] are example of architectural styles.

---

1. "Client-server" refers to two software components, usually implemented as separate processes, that serve as requestor and provider, respectively, of specific information or services.

2. A pipe is a mechanism for transporting data, unchanged, from one program to another. A filter is a program that applies a data transformation to a data set. This is a familiar style to programmers who use the Unix operating system; commands such as "cat data | grep 'keyword' | sort | fmt -80" are pipe-and-filter programs.

### 3.3.5 An Architecture Permits Template-Based Component Development

An architecture embodies design decisions about how components interact that, while reflected in each component at the code level, can be localized and written just once. Templates may be used to capture in one place the interaction mechanisms at the component level. For example, a template may encode the declarations for a component's public area where results will be left, or encode the protocols that the component uses to engage with the system executive. An example of a set of firm architectural decisions enabling template-based component development may be found in the Structural Modeling approach to software [Abowd 93].

# 4. Architectural Views and Architecture Frameworks

## 4.1 The Need for Multiple Structures or Views

The contractor, the architect, the interior designer, the landscaper, and the electrician all have a different architectural view of a building. These views are pictured differently, but all are inherently related and together describe the building's architecture.

In Section 2 we said that software architecture is about software structure, but we also explained that defining "the overall structure" of a system was an inherently ambiguous concept. So, just as the "structure of a building" has many meanings depending upon one's motive and viewpoint, software exhibits many structures and we cannot communicate meaningfully about a piece of software unless it is clear which structure we are describing.

Moreover, when designing the software for a large, complex system, it will be necessary to consider more than one structural perspective as well as the relationships among them. Though one often thinks about structure in terms of system functionality, there are system properties in addition to functionality, such as physical distribution, process communication, and synchronization, that must be reasoned about at an architectural level. These other properties are addressed in multiple structures often referred to as architectural views. They are also sometimes referred to as architectural models, but again the terminology has not settled enough yet to be dependable.

Each different view reflects a specific set of concerns that are of interest to a given group of stakeholders in the system. Views are therefore abstractions, each with respect to different criteria. Each abstraction "boils away" details about the software that are independent of the concern addressed by the abstraction. Each view can be considered to be a software blueprint and each can use its own notation, can reflect its own choice of architectural style, and can define what is meant in its case by components, interrelationships, rationale, principles, and guidelines.

Views are not fully independent, however. Elements in one can relate to elements in another so while it is helpful to consider each separately, it is also necessary to reason rigorously about the interrelations of these views. Views may be categorized as follows:

- Whether or not the structures they represent are discernible at system runtime. For example, programs exist at runtime; one can determine the calls structure of a system by observing the execution of the system. Modules, however, disappear; modules are a purely static (pre-runtime) phenomenon.

- Whether the structures describe the product, the process of building the product, or the process of using the product to solve a problem. All the views discussed in this section are of the product. A model of the user interaction presents another view of the architecture of the system, and is typically represented via entity-relation diagrams. Still other views model the problem area or domain.

Some authors differentiate views by what kind of information they show. For instance, Budgen distinguishes between functional, behavioral, structural, and data-modeling viewpoints [Budgen 93]. These all map, more or less, to the previous two categories.

Note that each view may be interpreted either as a description of the system that has been built, or a prescription that is engineered to achieve the relevant quality attributes.

## 4.2 Some Representative Views

There is not yet agreement on a standard set of views or terms to refer to views. In this section we list a typical and useful set. It should be noted that these views are given different names by various technologists. The perspective they represent is more important than the associated name.

### 4.2.1 Conceptual (Logical) View

The conceptual, or logical, architectural view includes the set of abstractions necessary to depict the functional requirements of a system at an abstract level. This view is tightly connected to the problem domain and is a useful communication vehicle when the architect interacts with the domain expert. The conceptual view is independent of implementation decisions and instead emphasizes interaction between entities in the problem space. This view is usually described by an informal block diagram, but in the case where object technology is utilized it may be expressed using class diagrams and class templates or in complex systems class categories.

Frameworks are similar to conceptual views but target not just systems but specific domains or problem classes. Frameworks are therefore close in nature to domain-specific architectures, to CORBA-based architecture models, and to domain-specific component repositories such as PRISM.

### 4.2.2 Module (Development) View

The module, or development, view is a frequently developed architectural structure. It focuses on the organization of actual software modules. Depending on how the modules are organized in the system, this view can take different forms. One form groups modules into identifiable subsystems, reflecting the software's organization into

chunks of related code in the system, and often the basis for allocating development or maintenance work to project teams. For example, a module view might use the principle of information-hiding as the grouping criterion to facilitate maintainability [Clements 85]. A much different grouping would result by collecting modules together that interact with each other heavily at runtime to perform related tasks.

Another form groups the modules into a hierarchy of layers that reflect design decisions about which modules can communicate as well as predictions about the generality or application speciality of each module. In a layered system, modules within a given layer can communicate with each other. Modules in different layers can communicate with each other only if their respective layers are adjacent[1]. In this way each layer has a well defined and narrowly scoped interface to the software that makes use of it. In addition, traversing down the hierarchy shows modules of greater generality that are less likely to change in response to an application-specific requirements change. Layered systems are often comprised of four to six layers.

Figure 3 shows the five layers of an air traffic control system [Kruchten 95].

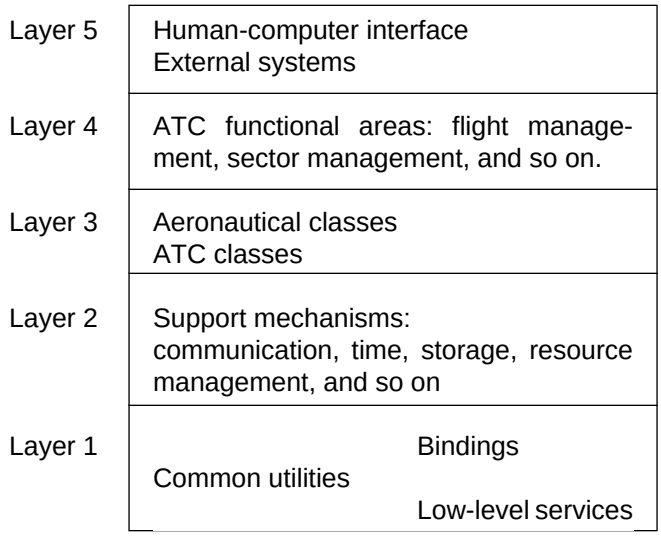


**Figure 3: Layers in an Air Traffic Control System [Kruchten 95]**

The module or layered organizations may or may not reflect the organization of the code as the compiler sees it in program libraries or compilation units. If not, then this compile-time structure is yet another view, which facilitates planning about the system build procedures.

1. In some systems, the rule is that a module can only access modules in the same or lower layers. In other systems, the communication is limited to the same or immediately lower layer.

Unlike the conceptual view, the module view is closely tied to the implementation. It is usually represented by module and subsystem diagrams that show interface imports and exports. As an architecture, this view has components that are either modules, subsystems, or layers, and the interrelationships are determined by import/export relations between modules, subsystems, or layers, respectively.

### 4.2.3 Process (Coordination) View

While the conceptual and module views we've seen so far deal with static aspects of the system, the process, or coordination, view takes an orthogonal perspective; it focuses on the runtime behavior of the system. As such, the process view is not as much concerned with functionality as it is with how entities are created, with communications mechanisms such as concurrency and synchronization. Clearly the process view deals with the system's dynamic aspects.

The structural components in the process view are usually *processes*. A process is a sequence of instructions (statements) with its own thread of control. During system execution, a process can be started, shut down, recovered, reconfigured, etc., and can communicate and synchronize as necessary with other processes.

This view facilitates reasoning about a system's performance and runtime scheduling based on inter-process communication patterns.

### 4.2.4 The Physical View

The physical view shows the mapping of software onto hardware. Software that executes on a network of computers must be partitioned into processes that are *distributed* across these computers. That distribution scheme is a structure that affects (and allows reasoning about) system availability, reliability, performance, and scalability.

This mapping of the software on to the hardware needs to be flexible and have as little impact on the actual code as possible since physical configurations can actually vary depending upon whether or not the system is in test or deployment and depending on the exact deployment environment.

### 4.2.5 Relating the Views to Each Other

Each one of these views provides a different structure of a system, each valid and useful in its own right. The conceptual and module views show the static system structure while the process and physical views give us the dynamic or runtime system structure. The conceptual and module views, though very close, address very different concerns as described above. Some argue that the process and physical view should be combined [Soni 95]. Further, there is no requirement or implication that these structures bear any topological resemblance to each other.

While the views give different system structural perspectives they are not fully independent. Elements of one view will be “connected” to elements of other views, and one needs to reason about those connections. Scenarios, as described below, are useful for exercising a given view as well as these inter-view connections.

All systems do not warrant multiple architectural views. Experience has shown that the larger the system, the more dramatic the difference between these views [Kruchten 95], but for very small systems the conceptual and module views may be so similar that they can be described together. If there is only one process or program there is clearly no need for the process view. If there is to be no distribution (that is, if there is just one processor) there is no need for the physical view. However, in the case of most systems of significant size and difficulty, if we were to attempt to combine these views into one structure we would limit evolvability, and reconfigurability, and add detailed complexity that would cripple the usefulness of the architecture as an artifact. This separation of concerns afforded by multiple views proves extremely beneficial in managing the complexity of large systems.

Figure 4 illustrates the views we have described, as well as their primary intended audiences and issues. It was adapted from a similar diagram shown in Kruchten [Kruchten 95].

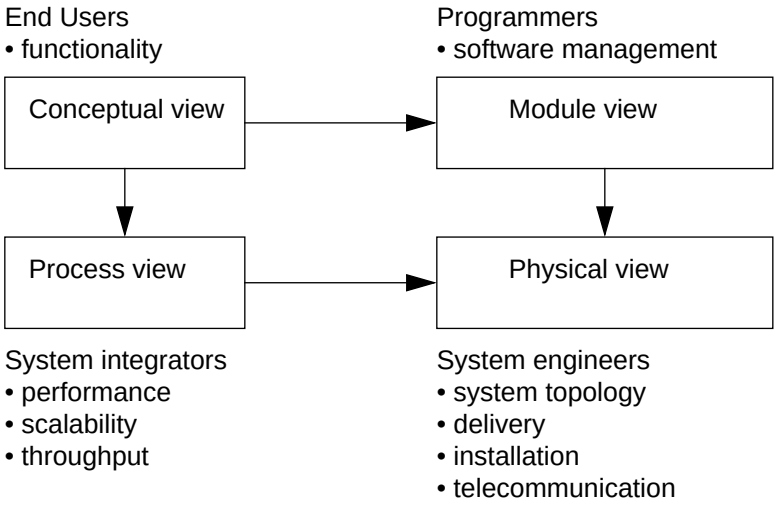


**Figure 4: Architectural Views**

**Scenarios**

Scenarios are scripts of individual pieces of a system’s functionality. Scenarios are useful in analyzing a given view [Clements 95a] or in showing that the elements of multiple views work together properly [Kruchten 95]. We can think about the scenar-

ios as an abstraction of the most important system requirements. Scenarios are described in text using what is called a script and are sometimes described in pictures, for example object interaction diagrams. Scenarios are an important tool to relate different architectural views, because walking through a scenario can show how parts of different architectural views are related to each other.

## 4.3 Architecture Frameworks

While the conceptual view is often generalized for a problem domain into what was described as a framework, there has been some effort to actually make frameworks for architectures of systems from a given broad domain. The Technical Architecture Framework for Information Management (TAFIM) described below is one such effort.

### 4.3.1 Technical Architecture Framework for Information Management (TAFIM)

The context for the management, use, and evolution of an information system is today rarely a collection of manual activities as was the case forty years ago. Current information systems function as entities within larger systems of related automated processes; they support the operations of independent, but interoperating mission critical systems; they are intrinsic elements of embedded computer systems that control, assimilate, distribute, activate, and/or monitor the operations of complex hybrids of digital and electronic hardware. Partly as a result of this increasing integration with other mechanical, digital and electronic systems, it is increasingly important to recognize the common elements shared by all information systems. Whether an information system accumulates stand-alone data on financial trends, provides interactive feedback to process control systems developers on the performance of their designs, interprets and recognizes patterns as part of a distributed surveillance system, calculates navigation and guidance parameters as an embedded subsystem of a military weapons platform, or computes and distributes robotic machine control instructions at the heart of a factory automation system, it typically has three kinds of elements, each with its own typical life cycle of management and technology decisions. The three typical elements of any information system are (a) its data; (b) its mission-specific applications; and (c) its infrastructure of support applications, computing platforms and communications networks.

The TAFIM [DoD 94], being developed by the Department of Defense (DoD) advances a descriptive pattern for information systems architectures (ISA). This pattern recognizes the pervasiveness of the above three information systems elements. It thereby provides a means for tailoring common life cycle management and technology decisions appropriately for different kinds of information systems applications.

A technical architecture such as TAFIM is *not* a complete architecture in that it does not fully specify components and the connections between them. Rather it is a specification that constrains the architectural choices that can be made. Often it does this by specifying consensus-based standards to which the system and its architecture must adhere. Thus, the TAFIM expresses a pattern for describing an ISA, and recognizes that there are necessary and proper variations in the content of ISAs within different types of information systems applications, enterprises, mission areas, computing communities, etc. For instance, an ISA that is tailored for information systems supporting weather processing, where parallel processing computations are intrinsic and essential, will have unique content and features. However, its overall form will be intelligible to systems managers and executives via its consonance with the TAFIM pattern for describing ISAs. This pattern is the so-called technical architectural “framework.”

# 5. Architecture-Based Development

## 5.1 Architecture-Based Activities for Software Development

What activities are involved in taking an architecture-based approach to software development? At the high level, they include the following that are in addition to the normal design-program-and-test activities in conventional small-scale development projects:

- understanding the domain requirements
- developing or selecting the architecture
- representing and communicating the architecture
- analyzing or evaluating the architecture
- implementing the system based on the architecture
- ensuring that the implementation conforms to the architecture

Each of these activities is briefly discussed below.

### 5.1.1 Understanding the Domain Requirements

Since a primary advantage of architectures is the insight they lend across whole families of systems, it is prudent to invest up front time in the life cycle to conduct an in depth study of requirements not just for the specific system but for the whole family of systems of which it is or will be a part. These families can be:

- a set of related systems, all fielded simultaneously, that differ incrementally by small changes
- a single system that exists in many versions over time, the versions differing from each other incrementally by small changes

In either case *domain analysis* is recommended. Domain analysis is an independent investigation of requirements during which changes, variations, and similarities for a given domain are anticipated, enumerated, and recorded for review by domain experts. An example of a domain is command and control. The result of domain analysis is a domain model which identifies the commonalities and variations among different instantiations of the system, whether fielded together or sequentially. Architectures based on domain models will yield the full advantage of architecture-based development because domain analysis can identify the potential migration paths that the architecture will have to support.

### 5.1.2 Developing (Selecting) the Architecture

While some advocate a phased approach to architectural design [Witt 94], experience has shown that architecture development is highly iterative and requires some prototyping, testing, measurement and analysis [Kruchten 95].

Ongoing work at the Software Engineering Institute posits that architects are influenced by factors in three areas [Clements 95b]:

- requirements (including required quality attributes) of the system or systems under development
- requirements imposed (perhaps implicitly) by the organization performing the development. For example, there may be requirements, or encouragement, to utilize a component repository, object-oriented environment, or previous designs in which the organization has significantly invested.
- experience of the architect. The results of previous decisions, whether wildly successful, utterly disastrous, or somewhere in between, will affect whether the architect reuses those strategies.

Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design, and that conceptual integrity can only be had by a very small number of minds coming together to design the system's architecture [Brooks 75].

### 5.1.3 Representing and Communicating the Architecture

In order for the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments it requires of them; testers must understand the task structure it imposes on them; management must understand the scheduling implications it suggests. The representation medium should therefore be informative and unambiguous, and should be readable by many people with varied backgrounds.

The architects themselves must make sure that the architecture will meet the behavioral, performance, and quality requirements of the system(s) to be built from the architecture. Therefore, there is an advantage if the representation medium can serve as input to formal analysis techniques such as model-building, simulation, verification, or even rapid prototyping. Towards this end, the representation medium should be formal and complete.

The field of architecture description languages (ADLs) is young but growing prolifically [Clements 96a].

### 5.1.4 Analyzing or Evaluating the Architecture

The analysis capabilities that many ADLs bring to the table are valuable, but tend to concentrate on the runtime properties of the system—its performance, its behavior, its communication patterns, and the like. Less represented are analysis techniques to evaluate an architecture from the point of view of non-runtime quality attributes that it imparts to a system. Chief among these is maintainability, the ability to support change. Maintainability has many variations: portability, reusability, adaptability, extensibility; all are special perspectives of a system's ability to support change. There is emerging a consensus on the value of scenario-based evaluation to judge an architecture with respect to non-runtime quality attributes [Kazman 95], [Kruchten 95].

### 5.1.5 Implementing Based on the Architecture and Assuring Conformance

This activity is concerned with ensuring that the developers adhere to the structures and interaction protocols dictated by the architecture. An architectural environment or infrastructure would be beneficial here. However, work in this area is still quite immature.

# 6. Current and Future Work in Software Architecture

As has been the sub-theme of this report, the study and practice of software architecture is still immature. Rigorous techniques need to be developed to describe software architecture so that they can be analyzed to predict and assume nonfunctional system properties. Moreover, architecture-based activities need to be more precisely defined and supported with processes and tools, and need to be smoothly incorporated into existing development processes.

There is an active research community working on technologies related to software architecture. This section examines ongoing technology work in software architecture and predicts areas of the most promising work that will bear fruit over the next five to ten years.

Problem areas in architecture and work that addresses them tend to be clustered around the following themes, arranged in terms of how and when the architecture is used during a system's life cycle:

- **Creation/selection:** How to choose, create, or select an architecture, based on a set of functional, performance, and quality requirements.
- **Representation:** How to communicate an architecture. This problem has manifested itself as one of representing architectures with linguistic facilities, but the problem also includes selecting the set of information to be communicated (i.e., represented with a language).
- **Analysis:** How to analyze an architecture to predict qualities about systems that manifest it, or how to evaluate an architecture for fitness. A similar problem is how to compare and choose between competing architectures.
- **Development:** How to build a system given a representation of its architecture.
- **Evolution:** How to evolve a legacy system when changes may affect its architecture; for systems lacking trustworthy architectural documentation, this will first involve "architectural archaeology" to extract its architecture.

See the Clements report[1] for a discussion of research opportunities in these areas.

1. The pending SEI technical report authored by Paul Clements, *Coming Attractions in Software Architecture*, should be available by April 1996.

# References


[Abowd 93] Abowd, G.; Bass, L.; Howard, L.; & Northrop, L. *Structural Modeling: An Application Framework and Development Process for Flight Simulators*. (CMU/SEI-93-TR-14, ADA 271348) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Abowd 94] Abowd, G.; Bass, L.; Kazman, R.; & Webb, M. "SAAM: A Method for Analyzing the Properties of Software Architectures," 81-90. *Proceedings of the 16th International Conference on Software Engineering.* Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Alexander 77] Alexander, Christopher; Ishikawa, Sara; & Silverstein, Murray. *A Pattern Language.* New York City: Oxford University Press, 1977.

[Batory 92] Batory, D. & O'Malley, S. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Transactions on Software Engineering and Methodology 2,* 4 (1992): 355-398.

[Beck 94] Beck, K. & Johnson, R. "Patterns Generate Architectures," 139-49. *European Conference on Object-Oriented Programming.* Bologna, Italy, July 4-8, 1994. Berlin, Germany: Springer-Verlag, 1994.

[Boehm 95] Boehm, B. "Engineering Context (for Software Architecture)." Invited talk, First International Workshop on Architecture for Software Systems. Seattle, Washington, April 1995.

[Brooks 75] Brooks, F. *The Mythical Man-Month—Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

[Brooks 95] Brooks, F. *The Mythical Man-Month—Essays on Software Engineering(20th Anniversary Edition).* Reading, MA: Addison-Wesley, 1995.

[Brown 95] Brown, A.; Carney, D.; & Clements, P. “A Case Study in Assessing the Maintainability of Large, Software-Intensive Systems,” 240-247. *Proceedings, International Symposium and Workshop on Systems Engineering of Computer Based Systems.* Tucson, AZ, March 6-9, 1995. Salem, MA: IEEE Computer Society Press, 1995.

[Budgen 95] Budgen, David. *Software Design.* Reading, MA: Addison-Wesley, 1993.

[Clements 85] Clements, P.; Parnas, D.; & Weiss, D. “The Modular Structure of Complex Systems.” *IEEE Transactions on Software Engineering SE-11*, 1 (1985): 259-266.

[Clements 95a] Clements, P.; Bass, L.; Kazman, F; & Abowd, G. “Predicting Software Quality by Architecture-Level Evaluation,” 485-498.*Proceedings, Fifth International Conference on Software Quality.* Austin, TX, Oct. 23-26, 1995. Austin, TX: American Society for Quality Control, Software Division, 1995.

[Clements 95b] Clements, P. “Understanding Architectural Influences and Decisions in Large-System Projects.” Invited talk, First International Workshop on Architectures for Software Systems. Seattle, WA, April 1995.

[Clements 96a] Clements, P. “A Survey of Architectural Description Languages,” *Proceedings of the Eighth International Workshop on Software Specification and Design*. Paderborn, Germany, March 1996.

[Dijkstra 68] Dijkstra, E.W. “The Structure of the ‘T.H.E.’ Mulitprogramming System.” *Communications of the ACM 18*, 8 (1968): 453-457.

[DoD 94] DoD Technical Architecture Framework for Information Management (TAFIM), Defense Information Systems Agency (DISA) Center for Information Management (CIM), Vol I (Concept) and Vol II (Guidance), V2.0. Reston, VA, October 1992.

[Gaycek 95] Gaycek, C.; Abd-Allah, A.; Clark, B.; & Boehm, B. “On the Definition of Software System Architecture.” Invited talk, First International Workshop on Architectures for Software Systems. Seattle, WA, April 1995.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns, Elements of Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Garlan 93] Garlan, D. & Shaw, M. “An Introduction to Software Architecture.” *Advances in Software Engineering and Knowledge Engineering. Vol 1.* River Edge, NJ: World Scientific Publishing Company, 1993.

[Garlan 95] Garlan, D. et al. “Architectural Mismatch (Why It’s Hard to Build Systems Out of Existing Parts),” 170-185. *Proceedings, 17th International Conference on Software Engineering.* Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995.

[Hayes-Roth 94] Hayes-Roth. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program* [online]. Available WWW <URL:http://www.sei.cmu.edu/arpa/dssa/dssa-adage/dssa.html>(1994).

[Kazman 95] Kazman, F.; Basas, l.; Abowd, G.; & Clements, P. “An Architectural Analysis Case Study: Internet Information Systems.” Invited talk, First International Workshop on Architectures for Software Systems. Seattle, WA, April 1995.

[Kruchten 95] Kruchten, Philippe B. “The 4+1 View Model of Architecture.” *IEEE Software 12,* 6 (November 1995): 42-50.

[Martin 92] Martin, C.; Hefley, W. et al. “Team-Based Incremental Acquisition of Larger Scale Unprecedented Systems.” *Policy Sciences.* Vol 25. The Netherlands: Kluwer Publishing, 1992.

[McMahon 95] McMahon, P. “Pattern-Based Architecture: Bridging Software Reuse and Cost Management.” *CROSSTALK 8,* 3 (March 1995): 10-16.

[OMG 91] Object Management Group. *Common Object Request Broker: Architecture and Specification.* Document 91.12.1. Framingham, MA, 1991.

[Parnas 72] Parnas, D. “On the Criteria for Decomposing Systems into Modules.” *Communications of the ACM 15,* 12 (December 1972): 1053-1058.

[Parnas 74] Parnas, D. “On a ’Buzzword’: Hierarchical Structure.” 336-3390. *Proceedings IFIP Congress 74.* North Holland Publishing Company, 1974.

[Parnas 76] Parnas, D. “On the Design and Development of Program Families.” *IEEE Transactions on Software Engineering SE-2*, 1 (1976): 1-9.

[Parnas 79] Parnas, D. “Designing Software for Ease of Extension and Contraction.” *IEEE Transactions on Software Engineering SE-5*, 2 (1979): 128-137.

[Perry 92] Perry, D.E. & Wolf, A.L. “Foundations for the Study of Software Architecture.” *Software Engineering Notes, ACM SIGSOFT 17,* 4 (October 1992): 40-52.

[Saunders 92] Saunders, T.F. et al. *A New Process for Acquiring Software Architecture*. MITRE Corporation Paper, M92B-126. Bedford, MA, November 1992.

[Schultz 95] Schultz, Charles. “Rome Laboratory Experience with Standards Based Architecture (SBA) Process.” Presentation at 7th Annual Software Technology Conference. Software Technology Systems Center (STSC), Hill AFB, UT, April 1995.

[Shaw 94] Shaw, M. “Making Choices: A Comparison of Styles for Software Architecture.” *IEEE Software 12*, 6 (November 1995):27-41.

[Shaw 95a] Shaw, M. “Conceptual Basis for Software Architecture.” Invited talk, First International Workshop on Architecture for Software Systems. Seattle, WA, April 1995.

[Shaw 95b] Shaw, M. & Garlan, D. *Software Architectures*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[Shaw 95c] Shaw, M. *Procedure Calls Are the Assembly Language of Software Interconnection; Connectors Deserve First-class Status*. (CMU-CS-94-107) Pittsburgh, PA: Carnegie Mellon University, 1994.

[Soni 95] Soni, D.; Nord, R.; & Hofmeister, C. “Software Architecture in Industrial Applications.” 196-210. *Proceedings, 17th International Conference on Software Engineering.* Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, 1995.

[Witt 94] Witt, B.I.; Baker, F.T.; & Merritt, E.W. *Software Architecture and Design Principles, Models and Methods*. New York, New York: Van Nostrand Reinholt, 1994.

## Acknowledgments

This study and report were funded by the Office of the Assistant Secretary of Defense for Command, Control, Communications, and Intelligence. John Leary was a contributor to an earlier version of this document. Our thanks go to Nelson Weiderman, who provided excellent comments.

# REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-96-TR-003

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-96-003

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F19628-95-C-0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

| PROGRAM ELEMENT NO | PROJECT NO. | TASK NO | WORK UNIT NO. |
|---|---|---|---|
| 63756E | N/A | N/A | N/A |

11. TITLE (Include Security Classification)
Software Architecture: An Executive Overview

12. PERSONAL AUTHOR(S)
Paul C. Clements, Linda M. Northrop

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (year, month, day) | 15. PAGE COUNT |
|---|---|---|---|
| Final | FROM TO | February 1996 | 38 |

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

| FIELD | GROUP | SUB. GR. |
|---|---|---|
| | | |
| | | |
| | | |

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
software architecture, architecture, architecture framework, framework, TAFIM, product lines, Technical Architecture Framework for Information Management

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Software architecture is an area of growing importance to practitioners and researchers in government, industry, and academia. Journals and international workshops are devoted to it. Working groups are formed to study it. Textbooks are emerging about it. The government is investing in the development of software architectures as core products in their own right. Industry is marketing architectural frameworks such as CORBA.

Why all the interest and investment? What is software architecture, and why is it perceived as providing a solution to the inherent difficulty in designing and developing large, complex systems? This report will attempt to summarize the concept of software architecture for an intended audience of mid- to senior-level management. The

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

reader is presumed to have some familiarity with common software engineering terms and concepts, but not to have a deep background in the field. This report is not intended to be overly-scholarly, nor is it intended to provide the technical depth necessary for practitioners and technologists. The intent is to distill some of the technical detail and provide a high level overview.