THE ROLE OF THE ARCHITECT

The architect’s role in the development of an EDM or PDM system is to maintain the integrity of the vision statement produced by the owners, users, and funders of the system.

This can be done by:

- ☑ Continually asking architecture questions in response to system requirements, requests for new features, alternative solutions, vendor’s offerings and the suggestions on how to improve the system – does this suggestion, product or feature request fit the architecture that has been defined? If not, does the requested item produce a change in the architecture? If so, does this item actually belong in the system?
- ☑ Continually asking the developers, integrators, and product vendors to describe how their system meets the architectural principles stated in. These questions are intended to maintain the integrity of the system for future and unforeseen needs, not the immediate needs of the users. Without this integrity, the system will not be able to adapt to these needs.
- ☑ Continually adapting to the needs of the users and the changing technology. The discipline of software architecture is continually changing. The knowledge of software architecture is expanding through research and practice. The architect must participant in this process as well.

ARCHITECTURE MANAGEMENT

The management of the system architecture is a *continuous improvement process* much in same way any quality program continually evaluates the design and production process in the search for improvement opportunities.

- ☑ Architectural evaluation – the architecture of the proposed system is continually compared with other architectural models to confirm its consistency.
- ☑ Architectural management – the architecture is actively managed in the same way software development is managed.
- ☑ System design processes – there are formal design processes for software architecture, just as there are formal processes for software development. These processes must be used if the architecture is to have an impact on the system integrity.
- ☑ Non–Functional design process – the non–functional requirements must be provided in the detailed system design. The design process will include the metrics needed to verify the non–functional requirements are being meet in the architecture.

Architecture Evaluation

Any framework used to manage the architecture of the system must include:

- ☑ Management – how are the various components of the system being defined, created, and managed? Are these components defined using some framework, in which their architectural structure can be validated?
- ☑ Coordination – are the various components and their authors participating in a rigorous process? Can the architectural structure of the system be evaluated across the various components with any consistency? Is there a clear and concise model of each coordinating function in the system?
- ☑ Transaction – are the various transactions in the system clearly defined? Are they visible? Are they recoverable? Do the transactions have permanence?
- ☑ Repository – is the data in the system isolated from the processing components?
- ☑ Type management – is there a mechanism for defining and maintaining the metadata for each data and process type?
- ☑ Security – have the security attributes of the system been defined before the data and processing aspects?

Architecture Management

The computational specifications of the system are intended to be distribution–independent. Failure to deal with this *transparency* is the primary cause of difficulty in the implementation of a physically distributed, heterogeneous system in a multi–organizational environment. Lack of transparency shifts the complexities from the applications domain to the supporting infrastructure domain. In the infrastructure domain, there are many more options available to deal with transparency issues. In the application domain,

- ☑ Access – hides the differences in data representation and procedure calling mechanisms to enable internetworking between heterogeneous systems.
- ☑ Location – makes the use of physical addresses, including the distinction between local and remote resource usage transparent.
- ☑ Relocation – hides the relocation of a service and its interface from other services and the interfaces bounded by it.
- ☑ Migration – masks the relocation of a service from that service and the services that interact with it.
- ☑ Persistence – masks the deactivation and reactivation of a service.
- ☑ Failure – masks the failure and possible recovery of services, to enhance the fault tolerance of the system.
- ☑ Transaction – hides the coordination required to satisfy the transactional properties of operations. Transactions have four critical properties: Atomicity, Consistency,

Isolation, and Durability. These properties are referred to as ACID. Atomicity means that the transaction execute to completion or not at all. Consistency means that the transaction preserves the internal consistency of the database. Isolation means the transaction executes as if it were running alone, with no other transactions. Durability means the transaction's results will not be lost in a failure.

System Design Process

The construction of software is based on several fundamental principals. These are called *enabling techniques*. All the enabling techniques are independent of a specific software development method, programming language, hardware environment, and largely the application domain. These enabling techniques have been known for years. Many were developed in the 1970's in connection with publications on structured programming.

Although the importance of these techniques has been recognized in the software development community for some time, it is now becoming clear of the strong link between system architecture and these enabling principles. Patterns for software architecture are explicitly built on these principles.

- ☑ Abstraction – is a fundamental principle used to cope with complexity. Abstraction can be defined as the essential characteristic of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries relative to the perspective of the viewer. The word object can be replaced by component or module to achieve a broader definition.
- ☑ Encapsulation – deals with grouping the elements of an abstraction that constitute its stricture and behavior, and with separating different abstractions from each other. Encapsulation provides explicit barriers between abstractions.
- ☑ Information hiding – involves concealing the details of a component's implementation from its clients, to better handle system complexities and to minimize coupling between components.
- ☑ Modularization – is concerned with the meaningful decomposition of a system design and with its grouping into subsystems and components.
- ☑ Separation – different or unrelated responsibilities should be separated from each other within the system. Collaborating components that contribute to the solution of a specific task should be separated from components that are involved in the computation of other tasks.
- ☑ Coupling and Cohesion – are principles originally introduced as part of structured design. Coupling focuses on inter–module aspects. Cohesion emphasizes intra–module characteristics. Coupling is measure of strength of association established by the connection from one module to another. Strong coupling complicates a system architecture, since a module is harder to understand, change, or to correct if it is highly interrelated with other modules. Complexity can be reduced by architecting systems with weak coupling.

- ☑ Cohesion measures the degree of connectivity between the functions and elements of a single function. There are several forms of cohesion, the most desirable being functional cohesion. The worst is coincidental cohesion in which unrelated abstractions are thrown into the same module. Other forms of cohesion – logical, temporal, procedural, and informal cohesion are described in the computer science literature.
- ☑ Sufficiency, completeness, and primitiveness – sufficiency means that a component should capture those characteristics of an abstraction that are necessary to permit a meaningful and efficient interaction with the component. Completeness means that a component should capture all relevant characteristics of it abstraction. Primitiveness means that all the operations a component can perform can be implemented easily. It should be the major goal of every architectural process to be sufficient and complete with respect to the solution to a given problem.
- ☑ Separation of policy and implementation – a component of a system should deal with policy or implementation, but not both. A policy component deals with context–sensitive decisions, knowledge about the semantics and interpretation if information, the assembly of many disjoint computations into a result or the selection of parameter values. An implementation component deals with the execution of a fully–specified algorithm in which no context–sensitive decisions have to be made.
- ☑ Separation of interface and implementation – any component in a properly architected system should consist of an interface and an implementation. The interface defines the functionality provided by the component and specifies how to use it. The implementation includes the actual processing for the functionality.
- ☑ Single point references – any function within the system should be declared and defined only once. This avoids problems with inconsistency.
- ☑ Divide and conquer strategies – is familiar to both system architects and political architects. By dividing the problem domain into smaller pieces, the effort necessary to provide a solution can be lessened.

Non–Functional Architecture

The non–functional properties of a system have the greatest impact on its development, deployment, and maintenance. The overall *abilities* of the system are a direct result of the non–functional aspects of the architecture.

- ☑ Changability – since systems usually have a long life span, they will age. This aging process creates new requirements for change. To reduce maintenance costs and the workload involved in changing a system's behavior, it is important to prepare its architecture for modification and evolution.
- ☑ Interoperability – the software system that result from a specific architecture do not exist independently from other system in the same environment. To support interoperability, system architecture must be designed to offer well–defined access to externally–visible functionality and data structures.

- ☑ Efficiency – deals with the use of the resources available for the execution of the software, and how this influences the behavior of the system.
- ☑ Reliability – deals with the general ability of the software to maintain its functionality, in the face of application or system errors and in situations of unexpected or incorrect usage.
- ☑ Testability – a system needs support from its architecture to ease the evaluation of its correctness.