UNIT V DOCUMENTING THE ARCHITECTURE

Good practices – Documenting the Views using UML – Merits and Demerits of using visual languages – Need for formal languages - Architectural Description Languages – ACME – Case studies. Special topics: SOA and Web services – Cloud Computing – Adaptive structures

5.1 GOOD PRACTICES

Importance of documenting an architecture:

Software architectures are represented through different views namely functional, operational, decisional and so on. No single view can represent the entire architecture. Not all views are required to represent the architecture of a system for a specific enterprise or problem of the domain. The architect determines the set of views that adequately represent the required dimensions of the software architecture.

By documenting the different views and capturing the development of each part, we can communicate information about the evolving system to the development team and to the business and IT stakeholders. A software architecture has to set of business and engineering goals that it is expected to meet. Documentation of the architecture communicates to the stakeholders how these goals will be achieved.

Documenting the various facets of a software architecture helps the architect bridge the proverbial gap between white-boarding the solution and representing the solution in a way that's meaningful to downstream design and implementation teams. The box-and-line diagram of the architecture leaves a lot of room for interpretation. Minutiae that need to be flushed out are often left hidden and confusingly intrinsic, behind those boxes and lines.

Documentation also fosters to the creation of architectural artifacts that are tangible and can be developed in a methodological fashion, like following a standard template. Software architecture's a discipline, is mature. We can leverage the best practices and guidelines to create standard templates for each view to represent a certain part or dimension of the architecture. Templates may educate the architect on what actually needs to be produced. They help the architect follow a regimen beyond the box-and-line technique. Templates define the architecture in more concrete terms so it can be traced directly to the business and IT goals that the solution is expected to meet.

Because of their complexity, typical systems-development initiatives can take around 18 months. Attrition(scratch) is common in design and development teams, causing a frantic struggle to find the right replacements. New team members usually are deterrents to progress, because they have to navigate a learning curve before they can become productive contributors.

A software architecture that has well-documented artifacts provides:

A perfect platform for educating new team members about what the solution entails.

An explanation of how the solution meets the business and engineering goals.

Various views of the solution architecture specific to the problem domain.

Focus on the view that an individual will work on.

Consider a hypothetical artifact called architecture decisions . This artifact identifies a problem to be solved and evaluates alternative mechanisms to address the problem. It provides justification for why a particular alternative is chosen over the others. A problem is identified concerning the mechanism to access mainframe IBM DB2. Two alternatives are evaluated: use IBM MQSeries or use a NEON Shadow Direct adapter. Though MQSeries has the capabilities and is less expensive, the latter is a much more stable, industry-strength product at the time the decision is made. Now imagine that the original architect has left the project after a architect comes on board. The new architect challenges the team as to why it's not using IBM MQSeries to access the mainframe DB2. The team quickly reverts to the architecture decision artifact and points out the reason for the choice. Because IBM MQSeries has been tested in the past year to be at par with the other solution and because it has a smaller price tag, the decision is revisited and changed to reflect the updated solution. This example illustrates why documenting various set of a system software architecture is the imperative to educate new team members and help them get started with minimum down time.

Uses of Architecture Documentation :

The axiom stated in the preceding section prescribes documenting the relevant views of the architecture. Which views are relevant? The answer, of course is it depends. In particular, what we put into software architecture documentation depends on what we wish to get out of it. So the question becomes How do we expect our documentation to be used? the answers will determine the form and content that our documentation must take. We'll begin by thinking about the uses of architecture corresponding to the times in a project's lifetime when the various roles come into play. A vehicle for communicating the system's design to interested stakeholders at each stage of its evolution. This perspective on architecture is forward-looking, involving steps of creation and refinement. Stakeholders include those involved in managing the project as well as consumers of the architecture that must write code to carry it out or design systems that must be compatible with it. Specific uses in this category include the following,

• For downstream designers and implementors, the architecture provides their marching(demo) orders. The architecture establishes inviolable(firm) constraints on the downstream development activities.

• For testers and integrators, the architecture dictates the correct black-box behavior of the pieces that must fit together.

•For technical managers, the architecture provides the basis for forming development teams corresponding to the work assignments identified.

• For project managers, the architecture serves as the basis for a work breakdown structure, planning, allocation of project resource and tracking of progress by the various teams.

• For designers of other systems with which this one must inter operate, the architecture defines the set of operations provided and required and the protocols for their operation, that allows the inter operation to the place.

A basis for performing up-front analysis is to validate architectural design decisions and then refine or alter those decisions wherever necessary. This perspective on architecture is, in some sense inward-looking. It involves making prospective architectural decisions and then projecting the effect of those decisions on the system or systems that the architecture is driving. Where the effect is unacceptable, the relevant decisions are re-thought and the process repeats. This process that occurs in tight cycles (most architects project the effect of each of their decisions) and in large cycle. In particular, architecture provides the following,

• For the architect and requirements engineers who represent the customer, the architecture is a form for negotiating and making trade-offs among competing requirements.

• For the architect and component designers, architecture is a vehicle for arbitrating resource contention and establishing performance and other kinds of run-time resource consumption budgets.

• For those who want to develop using vendor-provided products from the commercial marketplace, the architecture establishes the possibilities for commercial off-the-shelf (COTS) component integration by setting system and component boundaries and establishing its requirements for the required behavior and quality properties of those components.

• For those interested in the ability of the design to meet the system's quality objectives, the architecture serves as the fodder for architectural evaluation methods like the Software Architecture Analysis Method and the Architecture Tradeoff Analysis Method (ATAMSM) and the Software Performance Engineering (SPE) as well as less ambitious (and less effective) activities like unfocused design walk through. For performance engineers, architecture provides the formal model that drives analytical tools like the rate monotonic schedulers, simulations and the simulation generators, theorem provers and model checking verifiers.

• For product line managers, the architecture determines whether a potential new member of a product family is in or out of scope and if out, by how much. The first artifact used to achieve the system understanding. This perspective on architecture is reverse-looking. It refers to the cases in which the system has been built and deployed and now the time has come to make a change to it or to extract resources from it for using elsewhere. Architecture mining and recovery fall into this category as do routine maintenance activities. In particular, architecture serves the following roles,

• For technical mangers, architecture is basis for conformance checking, for assurance that the implementations have in fact been faithful to the architectural prescriptions.

• For maintainers, architecture is a starting point for maintenance activities, revealing the areas a prospective change it will be affect.

.• For new project members, the architecture is usually the first artifact for familiarization with a system's design.

• For those inheriting the job of architect after the previous architect's untimely departure, the architecture is the artifact that preserves that architect's knowledge and rationale.

• For re-engineers, architecture is the often first artifact recovered from a program understanding activity or (in the event that the architecture is known or has already been recovered) .

The artifact that drives program understanding activities at component granularities. It must be clear from this discussion that architecture documentation is both prescriptive and descriptive. That is, it prescribes what must be described, what is true about a system's design. In that sense, the same documentation serves both purposes and rightly so. If the build-as documentation differs from the as-built documentation then clearly there was a breakdown in the development process.

However, the best architectural documentation, say performance analysis may well be different from the best architectural documentation then we would wish to hand to a module implementation. After we introduce architectural views and other architectural documentation, we will return to the roles of architecture and discuss what documentation strategies are well suited to each role.

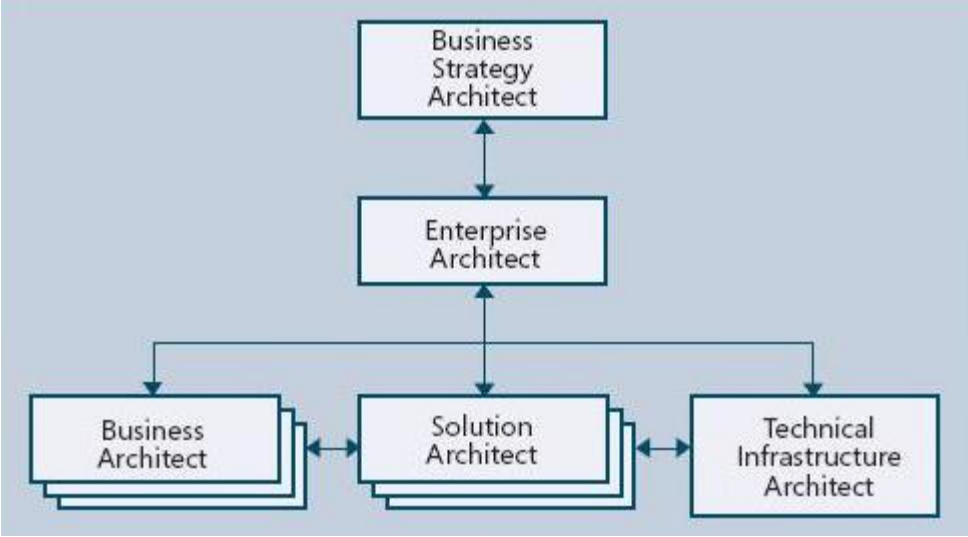
Architecture documentation is often a thorny issue in IT projects. It's common for there to be little or no documentation covering the architecture in many projects. Sometimes, if there is some, it's out-of-date, inappropriate and basically not very useful. At the other extreme there are projects that have masses of architecture related information captured in various documents and design tools. Sometimes this is invaluable, but at times it's out-of-date, inappropriate and not very useful clearly then, experience tells us that documenting architectures is not a simple job. But there are many good reasons why we want to document our architectures, for example, others can understand and evaluate the design. This includes any of the application stakeholders, but most commonly other members of the design and development team. We can understand the design when we return to it after a period of time. Others in the project team and development organization can learn from the architecture by digesting the thinking behind the design. We can do analysis on the design, perhaps to assess its likely performance, or to generate standard metrics like coupling and cohesion.

Documenting architectures is problematic though, because,

• There's no universally accepted architecture documentation standard.

• An architecture can be complex and documenting it in a comprehensible manner is time consuming and non-trivial.

• An architecture has many possible views. Documenting all the potentially useful ones is time consuming and expensive.



Architecture Documentation

In larger teams and especially when groups are not located in the same offices or building, the architecture documentation becomes vital importance for describing design elements like,

• Component interfaces.

• Subsystems constraints.

• Test scenarios.

• Third party component purchasing decisions.

• Team structure and schedule dependencies.

• External services to be offered by the application.

Documentation takes time to develop and costs money. It's therefore important to think carefully about how the documentation is going to be most useful within the project context and produce and also maintain this as key reference documents for the project.

5.2 DOCUMENTING THE VIEWS USING UML

Documenting C&C Views with UML 1.4:

Components and component types: Represent the principle runtime elements of the computation and data storage like clients, servers, filters and databases .

Connectors and connector types: Represent the runtime pathways of the interaction between components like pipes, publish-subscriber and client-server channels.

Component interfaces (or ports): Represent points of interaction between a C&C component and its environment. A given component may have many such interfaces, even some of that provide or require identical services. Connector interfaces represent points of interaction between a C&C connector and the components to which it is connected.

Systems: The graphs represent the components and connectors in the system and the pathways of interaction among them.

Decomposition: It is a means of representing substructure and selectively hiding complexity. A component that appears as a single entity at one level of description might have an internal architectural structure that provides more detail on its design.

Properties: Additional information associated with structural elements of an architecture. For C&C views, such properties typically characterize attributes that allow us to analyze the performance or reliability of a system.

Styles: Defining a vocabulary of component and connector types together with rules for how instances of those types can be combined to form an architecture in a given style. Common styles include pipe and filter, client-server and publish-subscriber.

Documenting Components and Connectors with UML 1.4 :

Because components are the primary computational elements of a C&C view of a software architecture, they feature prominently in architectural documentation. In fact, how we choose to represent various C&C concepts, like connectors and ports, often depends on how components are represented. Consequently, for the first choice that must be made how to represent C&C components the documentation options for using **UML are organized around three broad strategies are,**

UML classes and objects.

UML subsystems .

The UML Real-Time (UML-RT) profile.

At first glance, it seems like a glaring omission to ignore UML components because, in fact, they are quite similar to C&C components. UML components have interfaces, that may be deployed on hardware and are visually distinct from classes. UML components are often used in diagrams that depict an overall topology and just as architectural components are mapped to hardware, components are assigned to nodes in UML deployment diagrams.

In UML 1.4, components are defined as concrete chunks of implementation or example, executables or dynamic link libraries that realize abstract interfaces. In the C&C view type, the

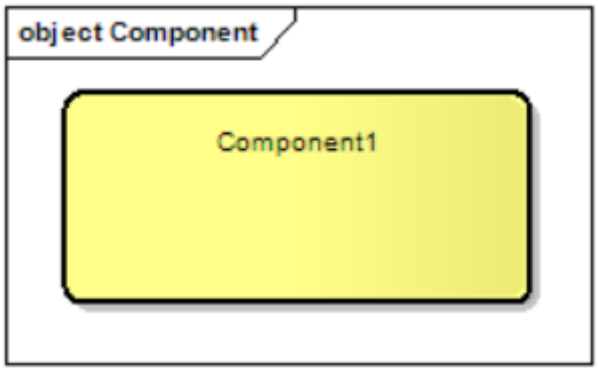
notion of a runtime component is more abstract, frequently having only an indirect relationship to a concrete unit of deployment. While architectural components in some systems may ultimately be realized as components in the UML sense, in many cases, their semantics simply do not match.

Modeling the Component-and-Connector Viewtype with UML 2.0:

In order to facilitate the design of the component-and connector views we have developed a UML which, by being a standard specification defined guarantees us that it will be able to be used by any tool that implements UML 2.0 or higher. The need to use the 2.0 or a higher UML version is due to the fact that in the 2.0 version some documentation constructs are introduced, like ports and roles, which are useful for our UML profile. Several of the available options for modeling a component-and-connector views use UML 2.0. Next, we will list the elements that make up the UML profile for the component-and-connector view type.

Components:

We decided to document the components that use the Component documentation construct defined in UML 2.0. Another option to document them was using the Classes documentation construct. There are some opinions that claim that the latter could vanish from the future UML versions since the semantics of both artifacts. We decided to give the component a similar visual aspect to the one used in documenting architecture diagrams, because we found it intuitive and practical for the purpose of documenting a software architecture.

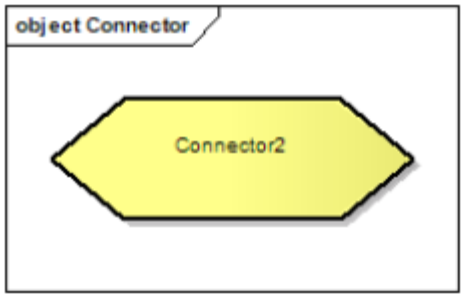


Component

Connectors:

The fact that a connector appears as a classifier has very deep implications in the expressive power of the connector and in the traceability of the artifacts for example, one could create a connector as a structured classifier and within that connector define the class diagram related to the design itself, its sequences, collaborations and quality requirements specifications, which would greatly facilitate the analysis of quality attributes that may or may not be reached by a software architecture. In spite of the fact that we take fundamental reference for our current work; the former is not included as an option in the use of a component construct to document a

connector. We believe that a connector's semantics is much more similar to the one of a component than to an association or an association class, We believe this because UML component are association and association class are not independent classifiers.



Connectors

5.3 MERITS AND DEMERITS OF USING VISUAL LANGUAGES

The trade-offs depend on the visual programming language in question, but the general rule is,

1.Text languages are better at the creation of new abstractions.

2.Visual languages are better at presenting and working with the existing abstractions.

An abstraction is basically a metaphor that lets us manipulate something related to the programming as the art of using abstractions to tell computers what to do.

For example, here are some classic programming abstractions,

1.A spreadsheet, which lets us represent various math operations as a bunch of rows + columns.

2.A function, which lets us represent operations in terms of the mapping from an input to an output.

3.An array, which lets us represent a collection of data as a numbered, traversable sequence.

Sometimes we have all the abstractions we need and the job is using them to solve a problem: Someone doing a data model in excel, for instance, is working within the spreadsheet abstraction. And sometimes we're tackling something we don't have the tools already, in which case we need to create new abstractions.

Text languages are great for creating new abstractions, because text is quick to work. We can define our new abstraction as some more text without having to write a bunch of complicated graphics code. However, if we have a pretty good set of abstractions already, visual languages that can save a ton of time by providing a nicer interface than text for working with them. The classic example of a user interface design, it's much faster to drag and drop a user interface than

to define it in text. Visual languages are often more beginner friendly, because the editor can do a lot of teaching the programmer how to use the language and showing them what their options are, whereas with text, a beginner programmer often flails around in an open-ended way.

The project is basically an example of collecting a set of common abstractions for building web and mobile apps designing user interfaces, submitted and sharing data, connecting to external services. Our goal is to lower the barrier to entry for app creation to anyone who's basically computer-savvy, regardless of their programming experience.

Advantages:

VB is not only a language but primarily an integrated, interactive development environment .

The structure of the Basic programming language is very simple, particularly as to the executable code.

The VB-IDE has been highly optimized to support rapid application development (“RAD”). It is particularly easy to develop graphical user interfaces and to connect them to handler functions provided by the application.

The graphical user interface of the VB-IDE provides intuitively appealing views for the management of the program structure in the large and the various types of entities .

VB provides a comprehensive, interactive and context-sensitive online help system.

When editing program, texts, the IntelliSense technology informs us in a little pop up window about the types of constructs that may be entered at the current cursor location.

VB is a component integration language which is attuned to Microsoft’s Component Object Model.

COM components can be written in different languages and then integrated using VB.

Interfaces of COM components can be easily called remotely via Distributed COM which makes it easy to construct distributed applications.

COM components can be embedded and linked to our application’s user interface and also in/to stored documents .

There is a wealth of readily available COM components for many different purposes.

Visual Basic is built around the .NET environment, used by all Microsoft Visual languages, so there is very little that can’t be done in Visual Basic that can be done in other languages .

Easy to "visualize" the logic. It's like looking at a flow chart.
Good for beginners to get an idea of how logical structures interact.

In some cases, they can aid rapid development.
There is basically no such thing as a syntax error.
Easier to do on a tablet in situations where no physical keyboard is available.
Disadvantages:

Visual basic is a proprietary programming language written by Microsoft, so programs written in Visual basic cannot, easily, be transferred to other operating systems.

There are some, fairly minor disadvantages compared with C. C has better declaration of arrays – its possible to initialize an array of structures in C at declaration time; this is impossible in VB.

Sometimes it's just easier to type to create a variable element and then link it to its source. As far as there's no visual equivalent to find/replace working with visual programming. Things can get really crowded and at some point we start to spend more time making room for things and encasing them in macros and rearranging them.

5.4 Architectural Description Languages

The software architecture is a new subject. Thus, there is not a universally accepted definition. The following is a definition in common use. Shaw and Garlan gave a definition from the software engineering point of view "Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns."

The above definition describes the software architecture from different points of view. However, there is not description about properties and behaviors of software architecture. Medvidovic and Taylor gave a description on classification framework for ADLs and comparison of various software architecture description languages. They view the architecture as a set of components and connectors.

In general methods of software development, informal graphs and texts are used to describe architecture. It is difficult to analyze their consistency and completeness etc. To solve this problem, some researchers have defined ways of formally describing architectures: Architecture Description Language (ADL). ADLs are used to present and analyze the design of architectures. And also, it provides some tools to display, analyze, and simulate software architectures.

Architecture Description Languages

Normally, if someone wants to develop a system, he/she would start to make diagrams, models, etc. at very beginning. The developer put everything about system into diagrams, like components, connectors, and restrictions. Hence, researchers created Architecture Description Language (ADL) to solve these problems. ADLs are a new way to help to model rather than the system implementation. Of course, there are also some disadvantages of ADLs, such as users need to be extensive trained, some of ADLs are only for a special type of systems, etc.

In the past ten years or more, many ADLs have been created and developed by some academic and industrial communities, and software architectures have been modeled by ADL. However, the architecture is a broad concept. What is meant with architecture and what sorts of data should be included, is different in different projects. So, choosing an ADL in a project depends on stakeholders and their needs. No ADL can be suitable for all projects. The research in this thesis also evaluates ADLs from a particular viewpoint: the software merge problem.

Although there is a big difference in the capability of different ADLs, all ADLs have similar conceptual basis or ontology. The ontology mainly includes: components, connectors, systems, properties, constraints, and styles. A survey of other ADLs describes most ADLs contain similar notions of components, connectors, and interfaces. In the following, we classify and analyze types of components and connectors mainly.

5.5 ACME

Goals for ACME

Interchange format for architectural development tools and environments

n * m problem -> m + n – tools graphical interface tools » animation » analysis for deadlock, well-formedness » architecture style-specific tools • Underlying representation for developing new tools for analyzing and visualizing architectures • Foundation for developing new, domain-specific ADLS

Language Features

1. an architectural ontology consisting of seven basic architectural design elements;

2. a flexible annotation mechanism supporting association of non-structural information using externally defined sublanguages;
3. a type mechanism for abstracting common, reusable architectural idioms and styles; and
4. an open semantic framework for reasoning about architectural descriptions.

ACME Design Element Types

Acme is built on a core ontology of seven types of entities for architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps. These are illustrated in Figures 3 and 4. Of the seven types, the most basic elements of architectural description are components, connectors, and systems.

- Components represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases.
- Connectors represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs, and intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application.
- Systems represent configurations of components and connectors.

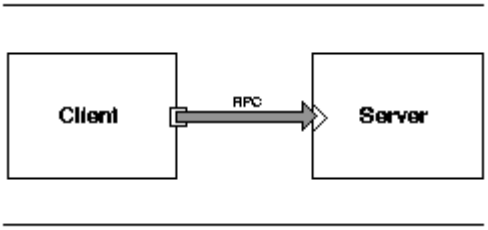


Figure 1: Simple Client-Server Diagram

```
System simple_cs = {
Component client = { Port send-request; };
Component server = { Port receive-request; };
Connector rpc = { Roels { caller, callee}};
Attachments {
client.send-request to rpc.caller;
server.receive-request to rpc.callee;
}
}
```

Figure2: Simple Client-Server System in Acme

As a simple illustrative example, Figure 1 shows a trivial architectural drawing containing a client and server component, connected by an RPC connector. Figure 2 contains its Acme description. The client component is declared to have a single send-request port, and the server has a single receive-request port. The connector has two roles designated caller and callee. The topology of this system is declared by listing a set of attachments.

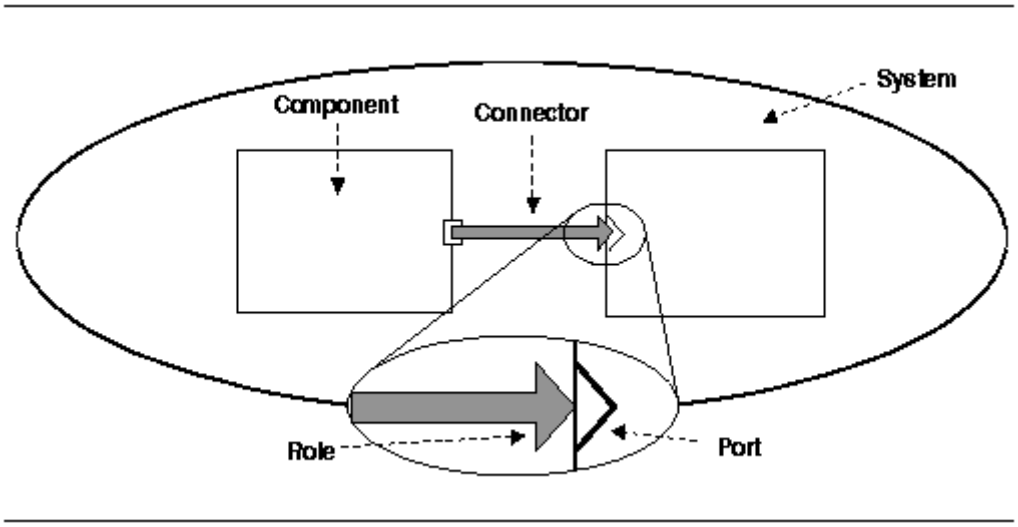


Figure 3: Elements of an Acme Description

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented.

ACME Properties

The seven classes of design element outlined above are sufficient for defining the structure of an architecture as a hierarchical graph of components and connectors. But there is clearly more to architectural description than structure. As discussed earlier, currently there is little consensus about exactly what should be added to the structural information: each ADL typically has its own set of auxiliary information that determines such things as the run-time semantics of the system, detailed typing information (such as types of data communicated between components), protocols of interaction, scheduling constraints, and information about resource consumption. To accommodate the wide variety of auxiliary information Acme supports annotation of architectural structure with lists of properties. Each property has a name, an optional type, and a value. Any of the seven kinds of Acme architectural design entities can be annotated.

ACME Families

The Acme features described thus far are sufficient to define an architectural instance, and, in fact, form the basis for the core capabilities of Acme parsing and unparsing tools. As a

representation that is good for humans to read and write, however, these features leave much to be desired. Specifically, they provide no facilities for abstracting architectural structure. As a result, common structures in complex system descriptions need to be repeatedly specified.

5.6 SOA and Web services

Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation.

Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

A service:

- Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
- Is self-contained
- May be composed of other services
- Is a "black box" to consumers of the service

SOA Architectural Style

An architectural style is the combination of distinctive features in which architecture is performed or expressed.

The SOA architectural style has the following distinctive features:

- It is based on the design of the services – which mirror real-world business activities – comprising the enterprise (or inter-enterprise) business processes.
- Service representation utilizes business descriptions to provide context (i.e., business process, goal, rule, policy, service interface, and service component) and implements services using service orchestration.
- It places unique requirements on the infrastructure – it is recommended that implementations use open standards to realize interoperability and location transparency.
- Implementations are environment-specific – they are constrained or enabled by context and must be described within that context.
- It requires strong governance of service representation and implementation.
- It requires a "Litmus Test", which determines a "good service".

SOA is the overarching strategy for building software applications inside a company—think of an architectural blueprint—except that in this case, the architecture calls for all the pieces of software to be built using a particular software development methodology, known as service-oriented programming. Web services, meanwhile, are a set of standard communication

mechanisms built upon the World Wide Web. Web services are a linking and communications methodology. SOA is an overall IT strategy.

CLOUD COMPUTING

Cloud Computing is a technology that uses the internet and central remote servers to maintain data and applications. Cloud computing allows consumers and businesses to use applications without installation and access their personal files at any computer with internet access. This technology allows for much more efficient computing by centralizing data storage, processing and bandwidth.

A simple example of cloud computing is Yahoo email, Gmail, or Hotmail etc. All you need is just an internet connection and you can start sending emails. The server and email management software is all on the cloud (internet) and is totally managed by the cloud service provider Yahoo , Google etc. Cloud computing is broken down into three segments: "application" "storage" and "connectivity." Each segment serves a different purpose and offers different products for businesses and individuals around the world.

Cloud Computing Segments

Transparency Market Research [3]

Cloud computing is a technology which uses internet and one remote server to maintain data and various applications. Cloud computing provides significant cost effective IT resources as cost on demand IT based on the actual usage of the customer. Due to rapid growth, many companies are unable to handle their IT requirement even after having an in-house datacenter. Cloud services helps to improve IT capabilities without investing large amounts in new datacenters. This technology helps companies with much more efficient computing by centralizing storage, memory, processing and bandwidth.

Applications
Storage
Connectivity

Cloud Computing Deployment Models and Concepts

Community Cloud

Community cloud shares infrastructure between several organizations from a specific community with common concerns , whether managed internally or by a third-party and hosted internally or externally. The costs are spread over fewer users than a public cloud (but more than that of a private) to realize its cost saving potential.

Public Cloud

Definition

A public cloud is established where several organizations have similar requirements and seek to share infrastructure so as to appliance. In addition, it can be economically

attractive as the resources (storage, workstations) utilized and shared in the community are already exploited.

This is the cloud computing model where service providers make their computing resources available online for the public. It allows the users to access various important resources on cloud, such as:Software, Applications or Stored data. On of the prime benefits of using public cloud is that the users are emancipated from performing certain important tasks on their computing machines that they cannot get away with otherwise, these include:Installation of resources, their configuration; and Storage.

Advantages of using Public Cloud

Some of the most important ones are mentioned here:

1. Efficient storage and computing services
2. Inexpensive, since all the virtual resources whether application, hardware or data are covered by the the service provider.
3. Allow for easy connectivity to servers and information sharing.
4. Assures appropriate use of resources as the users are required to pay only for the services they require.
5. Highly reliable and redundant.
6. Widespread availability irrespective of geographical precincts.
7. Sets the business people free from the hassles of buying, managing and maintaining all the virtual resources at their own end, the cloud server does it all.
8. Public cloud, in today's advanced workplace, empowers employees and enable them to become productive even when outside the office. The SaaS model ensures that corporations save on IT expenditures while delivering the flexibility of productivity software on the cloud.

Private cloud

- iCylanAPP

iCylanAPP enables you to remote access the sensitive applications of enterprises by smartphones or tablet device anywhere and anytime. The cloud-based resources are delivered to one platform, which providing high performance, security, and user experience. You can access the desktop, run applications, change settings, and access data exactly as you are sitting in front of the local PC, using its keyboard and mouse.

iCylanAPP has three versions, such as Standard Edition, Advanced Edition, Enterprise Edition, which providing proven security of different class. It can connect to any Windows applications running a iCylanAPP Client on smartphones or tablets devices.Nowadays, it supports the current systems,such as google Android, Mac iOS,windows Phone 7 or BlackBerry.