UNIT 8

Uses of Architectural Documentation

As we saw earlier that there are several stakeholders in a system and each Stake holder's Viewpoint is different. So, we cannot have a one-size-fits-all document. Different stakeholders for the documentation have different needs. Each require different kinds of information, different levels of information and different treatments of information. Also, stakeholders can be experienced or could be a novice.

Architecture documentation is both prescriptive and descriptive. Prescriptive aspects detail the constraints on decisions to be made in the future and descriptive aspects will document decisions already made.

Let us now see how different stakeholders make use of Architectural Documents

1. Architect and customer use the document to negotiate competing requirements.
2. Architect and designers of constituent parts use it to resolve resource contention and perhaps to establish budgets for runtime resource consumption.
3. Implementers use it to provide constraints that cannot be violated, as also the exploitable freedoms they have when building the system
4. Testers and integrators are interested in aspects that specify black-box behavior of the elements that must fit together.
5. Maintainers use it to make clear the side-effects of a prospective change
6. Designers of other interoperating systems are interested in the parts of document that discussed the services provided and required by the system, and the protocols to use for such transactions.
7. Quality attribute specialists use it to provide the model that drives analytical tools, to help evaluate if all QA requirements have been met
8. Managers use it to create/manage development teams and also to track progress
9. Quality assurance team uses the documentation to provide a basis for conformance checking

Views

We understand the need for multiple views because of the multiple stakeholders and their varying needs and viewpoints. What all views to include in the architectural documentation can be done using these steps

1. Produce a candidate view list. First, prepare list of stakeholders, the different views they might be interested in and the level of detail that will be required by each stakeholder
2. Combine views: Next we need to eliminate views that carry redundant information. Also some views can be combined.

3. Prioritize: Next step we have to assign priorities to which views have to be done first. Usually, we can follow a breadth-first approach, i.e.,, first fill in all views with overview information and then flesh out the detail by order of priority. Also, we have to note that the Management stakeholder may need attention early in the process

Documenting a View

This is the layout of the architectural document recommended by Bass et al.

1. Primary presentation: The essential elements of the systems, relationships between them. This can be graphical or tabular
2. Element Catalog: Details of all elements, including elements that may not have fit into primary presentation is given here. In this section we need to discuss the Properties of the elements, relationship between elements, interfaces offered by each element, and the behavior of the elements
3. Context Diagram: This part relates the primary presentation to environment, in terms of various interfaces, protocols, etc. used in the system
4. Variability Guide: The decisions that have deliberately been postponed has to be documented in this section. It might also be appropriate to specify the time by which such decision has to be made.
5. Architecture Background: This section should give the rationale for the choices made and also, why the alternatives were rejected. Analysis results to justify the design should also be included here. Also, the assumptions made in making the design decisions should be made explicit
6. Glossary of all technical terms used in the document.
7. Other Information: This is a catch-all section to include any information that falls outside the purview of the prior sections

Documenting Behavior

Structural Views (modules, uses, etc.) are inadequate. Behavior needs to be documented as well. These should reveal matters of interest like, ordering of interactions, opportunities for concurrency and time dependencies between actions, etc. UML sequence diagrams or state charts might be used to document behaviors.

Documenting Interfaces

It is important document the interfaces, i.e., how elements within the system interact or communicate. Here is a template for documenting interfaces.

1. Interface Identity: Unique name of all interfaces to all the elements in the system has to be given.
2. Resources provided: There are three parts to this section. First part is the resource syntax or resource signature. This should include name of the resource, names of arguments, data types of arguments, etc. The next part is the resource semantics, i.e., behavior that happens when the resource is invoked. This can include events or messages that might be raised when the resource is invoked, how it will affect other behaviors, after this resource is used, etc. The last part is resource usage restrictions, if any.

3. Data type definitions. If any data type is used that is not provided by the underlying programming language that has described here. This should include declarations, operations, translation, etc.
4. Exception definitions: Exceptions that can be raised by all resources have to be listed in this section. Also, common exception-handling behavior should also be included here.
5. Variability provided by the interface: Configuration options provided by the interface to elicit alternate behavior should be included in this section.
6. Quality attribute characteristics of the interface. This is needed to inform of any significant QA characteristic about usage of resource (e.g., performance degradation due to too many fine-grained calls).
7. Element requirements: This section should specify the resource requirements of an element to provide its service should be described here.
8. Rationale and design issues: Similar to the rationale section in documenting view, this section should justify the choices made and significant alternatives rejected.
9. Usage guide: This is important when the usage is complex and cannot be completely explained by the prior sections.

Cross-view Documentation

This should start with a section that tells how the documentation is organized to help the stakeholder in reading the document.

The first item in this might be the View Catalog, which describes for each view, the name of the view, the description of the elements in it, the properties of the elements, relation types, etc. The reason that particular view was included in the documentation should also be mentioned here. The View template should specify the standard parts of a view document and the contents and rules for each, to help the reader in identifying significant aspects fast.

The next section describes what the architecture is. This should contain the system overview, which is a short description of the function of the system, targeted users, background, etc. Mapping between views should be made explicit here. To help in the reader navigate the documentation, the list of elements and where they appear in document should also be given. A project glossary might be useful, to help non-technical stakeholders or novice stakeholders.

The last section should explain why the architecture is the way it is. Rationale for the various design decisions made should be explained here. If any changes can be anticipated, the side-effect of the change should be documented in this section. This section should explain the constraints within which the developers need to work. Also, useful to include will be to list the significant alternatives that were rejected, and why they were rejected.

Documenting Architectures:

Need of Documenting Architecture

- Documenting the architecture is the crowning step to crafting it.
- Even a perfect architecture is useless if no one understands it, or
- If key stakeholders misunderstand it.
- The architecture for a system depends on the requirements levied on it, so too does the documentation for architecture depend on the requirements levied on it.
- Decidedly not a case of "one size fits all" !!!
- Sufficiently abstract and sufficiently detailed
- Different types of documentation needed for say, security and implementation
- Architecture documentation is both prescriptive and descriptive.
- different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.
- This might mean producing different documents for different stakeholders.
- No, more likely, it means producing a single documentation suite with a roadmap that will help different stakeholders navigate through it.
- Write from the point of view of the reader
- "easy to read" is in the eye of the beholder
- In addition, each stakeholder comes in two varieties: ***seasoned*** and ***new***.
- Perhaps one of the most avid consumers of architectural documentation is none other than the architect himself.

Documenting Views

- Software architecture is a complex entity that cannot be described in a simple one-dimensional fashion.
- ***Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.***
- This principle breaks the problem of architecture documentation into more tractable parts
  - Choosing the relevant views
  - Documenting a view
  - Documenting information that applies to more than one view

Choosing the Relevant Views

- This is where knowing your stakeholders and the uses they plan to make of the documentation will help you construct the documentation package they need.
- The many purposes that architecture can serve—as a mission statement for implementers, as the starting point for system understanding and asset recovery, as the blueprint for project planning, and so forth—are each represented by a stakeholder wanting and expecting to use the documentation to serve that purpose.
- Similarly, the quality attributes of most concern to you and the other stakeholders in the system's development will affect the choice of what views to document.
- In short, different views support different goals and uses.

	Module Views				**C&C Views**	**Allocation Views**	
Stakeholder	***Decomposition***	***Uses***	***Class***	***Layer***	***Various***	***Deployment***	***Implementation***
Project Manager	s	s		s		d	
Member of Development	d	d	d	d	d	s	s

Team							
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	
End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d
New Stakeholder	x	x	x	x	x	x	x
Current and Future Architect	d	d	d	d	d	d	s
Key: d = detailed information, s = some details, o = overview information, x = anything							

Stakeholders and Corresponding Views

- There are three types of views
    - Modules
    - Components and Connectors
    - Allocation
- This three-way categorization reflects the fact that architects need to think about their software in at least three ways at once

Choosing a View

1. Produce a candidate view list

- Begin by building a stakeholder/view table
- Be as comprehensive as you can.
- For the columns, enumerate the views that apply to your system.
- Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view:
    - none, overview only, moderate detail, or high detail.

2. Combine views.

- The candidate view list from step 1 is likely to yield an impractically large number of views.
- First look for views in the table that require only overview depth or that serve very few stakeholders.
- Next, look for views that are good candidates to be combined—that is, a view that gives information from two or more views at once.
- For small and medium projects, the implementation view is often easily overlaid with the module decomposition view etc

3. Prioritize

- Which view to be completed first depends upon details specific to your project
- you don't have to complete one view before starting another.
- People can make progress with overview-level information, so a breadth-first approach is often the best.

Also, some stakeholders' interests supersede others

Documenting the view

- First of all, whatever sections you choose to include, make sure to have a standard organization.
- Allocating specific information to specific sections will help the documentation writer attack the task and recognize completion, and
- It will help the documentation reader quickly find information of interest at the moment and skip everything else.
- Clements et. al. proposed seven-part standard organization which is working practically well.

Seven-part Standard Organization

1. *Primary presentation* shows the elements and the relationships among them that populate the view.

- The primary presentation should contain the information you wish to convey about the system first.
- Some elements can be omitted.
- For example, you may wish to show the elements and relations that come into play during normal operation, but relegate error handling or exceptional processing to the supporting documentation.
- Usually graphical, along with keys that explains notations
- Sometimes, tabular. Text presents a summary.

2. *Element catalog* details at least those elements and relations depicted in the primary presentation, and perhaps others.

- Producing primary presentation is of focus, but without backup information, it has little value.
- For instance, if a diagram shows elements A, B, and C, there had better be documentation that explains in sufficient detail what A, B, and C are, and their purposes or the roles they play, rendered in the vocabulary of the view.
- Introduce omitted elements here and detail.

3. Context diagram shows how the system depicted in the view relates to its environment in the vocabulary of the view.

For example, in a component-and-connector view you show which component and connectors interact with external components and connectors, via which interfaces and protocols.

4. Variability guide shows how to exercise any variation points that are a part of the architecture shown in this view.

An example of variability is found in software product lines where the product line architecture is suitable for multiple particular systems

It should include documentation regarding each point of variation including

- the options among which a choice is to be made
- the binding time of the option.

5. Architecture background explains why the design reflected in the view came to be.

An architecture background includes:

- *rationale,* explaining why the decisions reflected in the view were made and why alternatives were rejected.

- *analysis results,* which justify the design or explain what would have to change in the face of a modification.

- *assumptions* reflected in the design.

6. Glossary of terms used in the views, with a brief description of each.

7. Other information. The precise contents of this section will vary according to the standard practices of your organization.

They might include management information such as authorship, configuration control data, and change histories.

Strictly speaking, information such as this is not architectural.

Documenting Behavior

- Views present structural information about the system.
- Structural information is not sufficient to allow reasoning about some system properties.
- For example, Reasoning about deadlocks
- Behavior descriptions add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions
- Behavior can be documented either about an element or about an ensemble of elements.
- Exactly what to model will depend on the type of system being designed?
- Different modeling techniques and notations are used depending on the type of analysis to be performed.
- In UML we use
    - State charts
    - Sequence Diagrams
- State charts add a number of useful extensions to traditional state diagrams such as nesting of state and "and" states, which provide the expressive power to model abstraction and concurrency.
- It is possible to answer a question such as Will the response time to a stimulus always be less than 0.5 seconds?
- A sequence diagram documents a sequence of stimuli exchanges.
- It presents a collaboration in terms of component instances and their interactions and shows the interaction arranged in time sequence.
- The vertical dimension represents time and the horizontal dimension represents different components.
- They make it possible to answer a question such as what parallel activities occur when the system is responding to these specific stimuli under these specific conditions.

Documenting Interfaces

- An interface is a boundary across which two independent entities meet and interact or communicate with each other.
- Documenting an interface consists of naming and identifying it and documenting its syntactic and semantic information.
- The first two parts constitute an interface's "signature."
- Signatures are useful (for example, they can enable automatic build checking), but are only part of the story.
- Semantics actually tell what happens when resources are brought into play

- the architect chooses what information is permissible and appropriate for people to assume about the element, and what is unlikely to change.
- Documenting an interface is a matter of striking a balance between disclosing too little information and disclosing too much.
- The rule of thumb is to focus on how elements interact with their operational environments, not on how they are implemented.
- Elements that occur as modules often correspond directly to one or more elements in a component-and-connector view.
- The module and component-and-connector elements are likely to have similar, if not identical, interfaces, and
- Documenting them in both places would produce needless duplication.
- To avoid that, the interface specification in the component-and-connector view can point to the interface specification in the module view, and only contain the information specific to its view.

Where does software architecture fits in the SDLC
Requirements come first
But not all requirements are necessary to get started
Architecture shaped by (remember the ABC)
Functional requirements
Quality requirements
Business requirements
Expertise and experience of architects

Attribute Driven Design

Design architecture to support both functional requirements and quality requirements.

ADD for SA then following Rational Unified Process for the rest of the design

Steps involved in Attribute Driven Design (ADD)

Choose the module to decompose

start with entire system

Inputs for this module need to be available

Constraints, functional and quality requirements

Refine the module

Choose architectural drivers relevant to this decomposition

Choose architectural pattern that satisfies these drivers

Instantiate modules and allocate functionality from use cases representing using multiple views

Define interfaces of child modules

Verify and refine use cases and quality scenarios

Repeat for every module that needs further decomposition

System -> subsytem-> submodule

Start with the entire system as the module

Refine the module

Choose the architectural drivers from the quality scenarios and functional requirements
The drivers will be among the top priority requirements for the module.
Real-time performance requirement
Modifiability requirement
Requirements are not treated as equals

Less important requirements are satisfied within constraints obtained by satisfying more important requirements
This is a difference of ADD from other SA design methods
Recall an architectural pattern is determined by:
1.A set of elements
2.A topological layout of the elements indicating relationships
3.A set of semantic constraints
4.A set of interaction mechanisms
For each quality requirement there are identifiable tactics and then identifiable patterns that implement these tactics.
<u>Reexamine performance and modifiability tactics</u>
Modifiability Tactic categories
1.Localize changes
2.Prevent ripple effects
3.Defer binding time

Since the modifiability scenarios are primarily concerned with design time changes primary tactic is “localize changes”
<u>Performance Tactic categories</u>

1.Resource demand category
2.Resource arbitration category

<u>From each primary category select one tactic</u>
1.Localize changes
2.Semantic coherence and information hiding
3.Separate responsibilities dealing with
User interface

Next step is to verify whether decomposition achieves the desired functionality
and Allocate that functionality
Applying use cases may modify decomposition
<u>In the end every use case of the parent module must be representable by sequence of responsibilities within children</u>
Assigning these responsibilities to the children will also determine communications between children

<u>Views: module decomposition, concurrency, and deployment</u>
<u>Module decomposition view</u>
Provide containers for functionality
<u>Concurrency view</u>
Parallel activities such as resource contention, deadlock
Likely will lead to the discovery of new responsibilities
<u>Deployment view</u>

<u>Use cases Illustrating Concurrency</u>

Two users doing similar things at the same time
One user performing multiple activities simultaneously
Starting up the system
Shutting down the system

The three views provide:

<u>Module view</u>

patterns of communication

<u>Concurrency view</u>

Interactions among threads

Synchronization information

Deployment view

Hardware requirements

Timing requirements

Communication requirements

Verifying the decomposition by "running" the parent's use cases

Preparing children for their own decomposition by inheriting use cases/constraints from the parent.

We will examine this by looking at:

Functional requirements

Constraints

Quality Scenarios

Using Functional requirements to verify and refine

Decomposing functional requirements assigns responsibilities to child modules

Primary presentation – elements and their relationships

Element catalog -- explains the picture

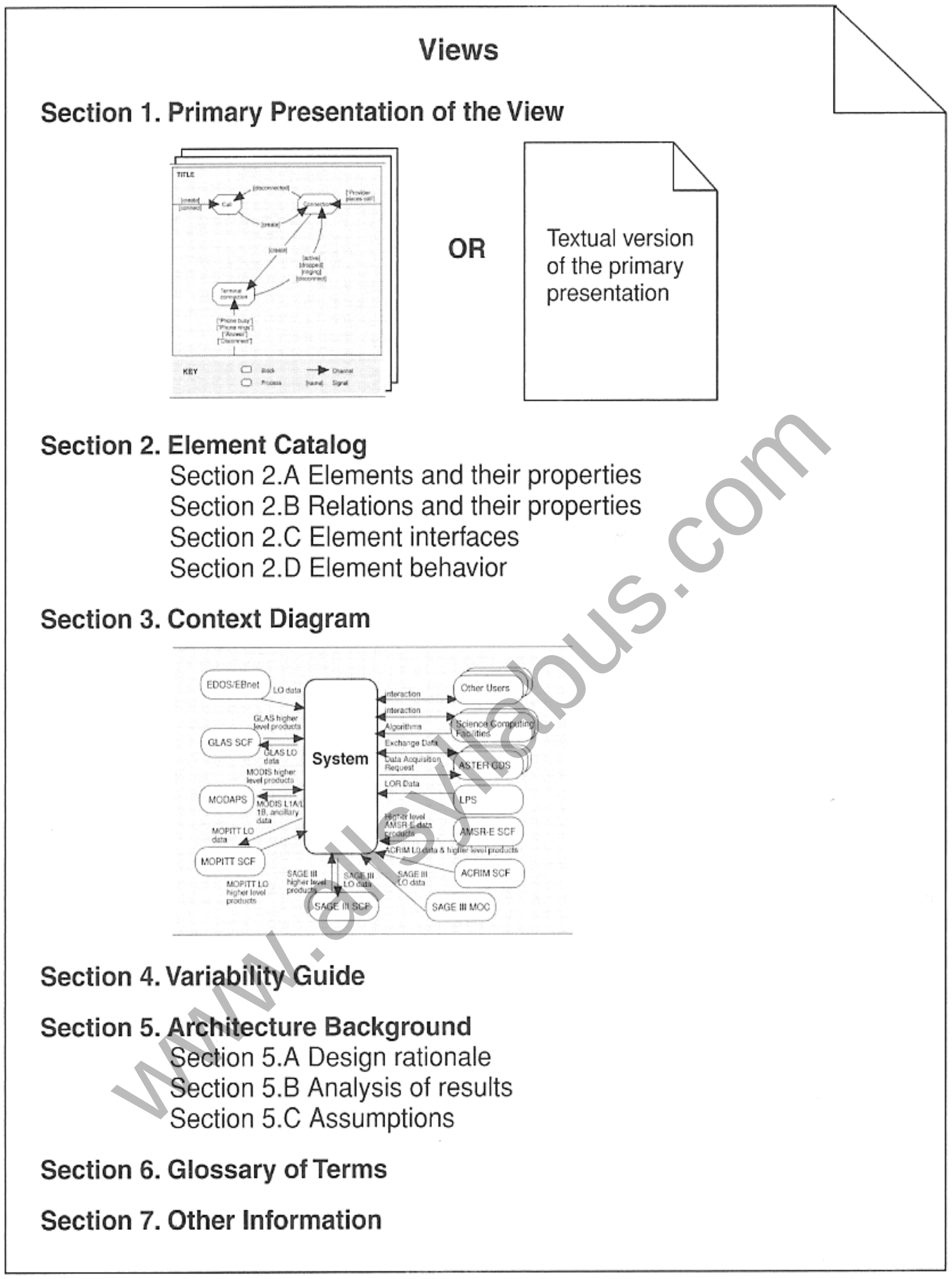
Context diagram -- how the system relates to its environment

Variability guide – how to exercise any variation points

Architecture background – why the design reflected in the view came to be

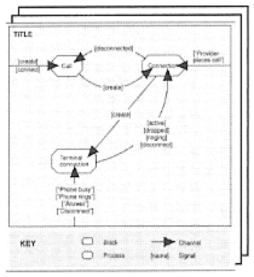
Glossary of terms used

Other information



Views

Section 1. Primary Presentation of the View



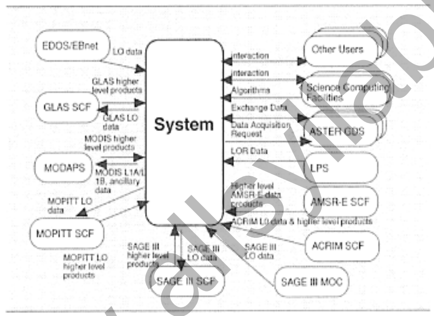
OR

Textual version of the primary presentation

Section 2. Element Catalog

Section 2.A Elements and their properties
Section 2.B Relations and their properties
Section 2.C Element interfaces
Section 2.D Element behavior

Section 3. Context Diagram



Section 4. Variability Guide

Section 5. Architecture Background

Section 5.A Design rationale
Section 5.B Analysis of results
Section 5.C Assumptions

Section 6. Glossary of Terms

Section 7. Other Information

An interface is a boundary across which tow independent entities meet and interact or communicate with each other.

Documenting interface is an important part of documenting architecture.

Documenting interfaces

Interface identity - unique name

Resources provided

Locally defined data types - if used

Exception definitions - including handling

Variability provided - for product lines

Quality attributes of the interface

Element requirements

Design issues

Usage guide

<u>How the documentation is organized</u>?

1.View catalog– Introduction to view

2.View Template – Diagram as discussed

<u>Resources provided</u>

Syntax - signature

<u>Semantics</u>

assignment of values to data

changes in state

events signaled or messages sent

how other resources will behave differently in future

humanly observable results

<u>Usage restrictions</u>

initialization requirements

limit on number of actors using resource

- **<u>WHAT THE ARCHITECTURE IS</u>?**
- **SYSTEM OVERVIEW**
- **MAPPING BETWEEN VIEWS**

- **ELEMENT LIST**
- **PROJECT GLOSSARY**
- **THE IMLICATIONS OF SYSTEM WIDE DESIGN CHOICES ?**
- **MODIFIABILITY AND EXTENSION EFFECTS**
- **CONSTRAINTS ON THE DESIGNER DURING IMPLEMENTATION**
- **DECISION ALTERNATIVES THAT WERE REJECTED.**
- **WHY A DECISION WAS MADE ,** , Decision making by using abstractions itself is software architecture
- **END OF ARCHITECTURAL DISCUSSION**