FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

UNIT 8
DESIGNING AND DOCUMENTING SOFTWARE ARCHITECTURE

CHAPTER 7:
DESIGNING THE ARCHITECTURE

ARCHITECTURE IN THE LIFE CYCLE

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model shown in figure 7.1.

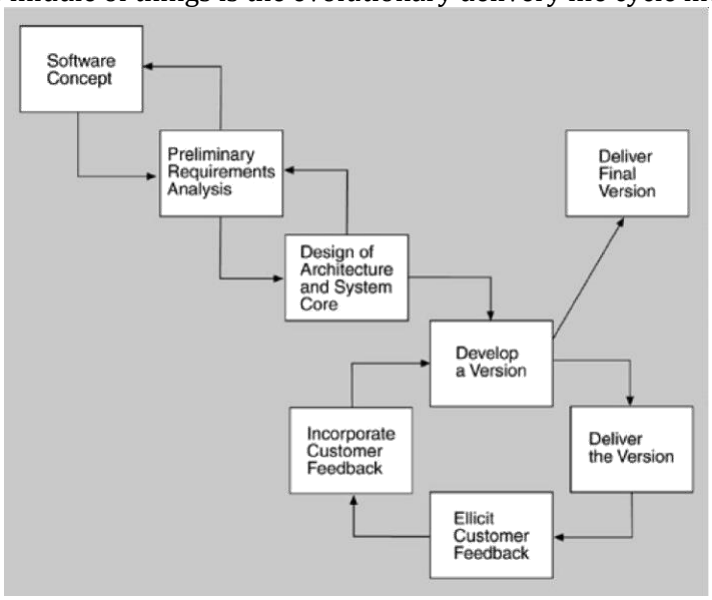


Figure 7.1. Evolutionary Delivery Life Cycle

The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

WHEN CAN I BEGIN DESIGNING?

- The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.
- An architecture is "shaped" by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.
- To determine the architectural drivers, identify the highest priority business goals. There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

7.2 DESIGNING THE ARCHITECTURE

A method for designing an architecture to satisfy both quality requirements and functional requirements is called attribute-driven design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture.

ATTRIBUTE DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

Garage door opener example

- Design a product line architecture for a garage door opener with a larger home information system the opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.
- **Input** to ADD: a set of requirements
  - Functional requirements as use cases
  - Constraints
  - Quality requirements expressed as system specific quality scenarios
- **Scenarios** for garage door system
  - Device and controls for opening and closing the door are different for the various products in the product line
  - The processor used in different products will differ
  - If an obstacle is (person or object) is detected by the garage door during descent, it must stop within 0.1 second
  - The garage door opener system needs to be accessible from the home information system for diagnosis and administration.
  - It should be possible to directly produce an architecture that reflects this protocol

ADD Steps:

Steps involved in attribute driven design (ADD)

1. *Choose the module to decompose*
   - Start with entire system
   - Inputs for this module need to be available
   - Constraints, functional and quality requirements
2. *Refine the module*
   a) Choose architectural drivers relevant to this decomposition
   b) Choose architectural pattern that satisfies these drivers
   c) Instantiate modules and allocate functionality from use cases representing using multiple views
   d) Define interfaces of child modules
   e) Verify and refine use cases and quality scenarios
3. *Repeat for every module that needs further decomposition*

Discussion of the above steps in more detail:

1. **Choose The Module To Decompose**
   - the following are the modules: system->subsystem->submodule
   - Decomposition typically starts with system, which then decompose into subsystem and then into sub-modules.
   - In our Example, the garage door opener is a system
   - Opener must interoperate with the home information system

2. **Refine the module**

1. Choose Architectural Drivers:

- choose the architectural drivers from the quality scenarios and functional requirements
- The drivers will be among the top priority requirements for the module.
- In the garage system, the 4 scenarios were architectural drivers,
- By examining them, we see
  - Real-time performance requirement
  - Modifiability requirement to support product line
- 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 architecture design methods

2. Choose Architectural Pattern

- For each quality requirement there are identifiable tactics and then identifiable patterns that implement these tactics.
- The goal of this step is to establish an overall architectural pattern for the module
- The pattern needs to satisfy the architectural pattern for the module tactics selected to satisfy the drivers
- Two factors involved in selecting tactics:
  - ✓ Architectural drivers themselves
  - ✓ Side effects of the pattern implementing the tactic on other requirements ○ This

yields the following tactics:

- *Semantic coherence* and *information hiding.* Separate responsibilities dealing with the user interface, communication, and sensors into their own modules.
- *Increase computational efficiency.* The performance-critical computations should be made as efficient as possible.
- *Schedule wisely.* The performance-critical computations should be scheduled to ensure the achievement of the timing deadline.

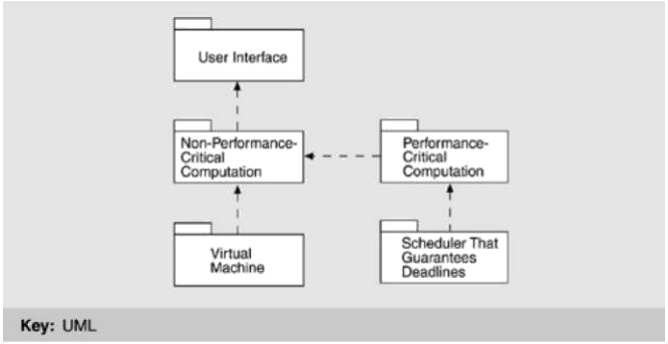


Figure 7.2. Architectural pattern that utilizes tactics to achieve garage door drivers

3. Instantiate Modules And Allocate Functionality Using Multiple Views

♥ *Instantiate modules*

The non-performance-critical module of Figure 7.2 becomes instantiated as diagnosis and raising/lowering door modules in Figure 7.3. We also identify several responsibilities of the virtual machine: communication and sensor reading and actuator control. This yields two instances of the virtual machine that are also shown in Figure 7.3.

♥ *Allocate functionality*

Assigning responsibilities to the children in a decomposition also leads to the discovery of necessary information exchange. At this point in the design, it is not important to define how the information is exchanged. Is the information pushed or pulled? Is it passed as a message or a call parameter? These are all questions that need to be answered later in the design process. At this point only the information itself and the producer and consumer roles are of interest

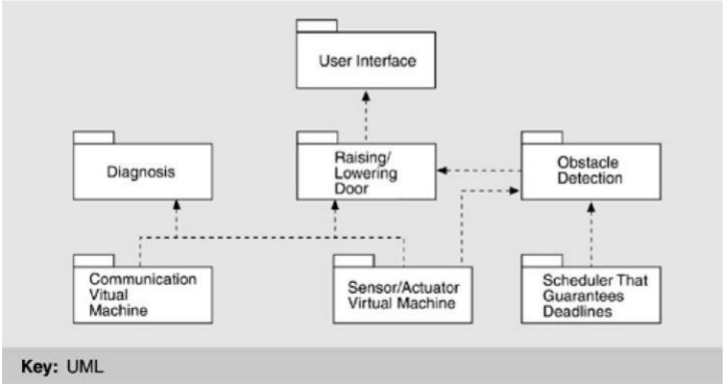


Figure 7.3. First-level decomposition of garage door opener

- ♥ *Represent the architecture with multiple views*
  - ♣ *Module decomposition view*
  - ♣ *Concurrency view*
    - Two users doing similar things at the same time
    - One user performing multiple activities simultaneously
    - Starting up the system
    - Shutting down the system
  - ♣ *Deployment view*

4. Define Interfaces Of Child Modules

- It documents what this module provides to others.
- Analyzing the decomposition into the 3 views provides interaction information for the interface
  - *Module view:*
    - ✓ Producers/consumers relations
    - ✓ patterns of communication
  - *Concurrency view:*
    - ✓ Interactions among threads
    - ✓ Synchronization information
  - *Deployment view*
    - ✓ Hardware requirement
    - ✓ Timing requirements
    - ✓ Communication requirements

5. Verify And Refine Use Cases And Quality Scenarios As Constraints For The Child Modules

- *Functional requirements*

Using functional requirements to verify and refine

- ➢ Decomposing functional requirements assigns responsibilities to child modules
- ➢ We can use these responsibilities to generate use cases for the child module
  - ✓ *User interface:*
    - ♣ Handle user requests
    - ♣ Translate for raising/lowering module
    - ♣ Display responses
  - ✓ *Raising/lowering door module*
    - ♣ Control actuators to raise/lower door
    - ♣ Stop when completed opening or closing
  - ✓ *Obstacle detection:*
    - ♣ Recognize when object is detected
    - ♣ Stop or reverse the closing of the door
  - ✓ *Communication virtual machine*
    - ♣ Manage communication with house information system(HIS)

✓ *Scheduler*
- ♣ Guarantee that deadlines are met when obstacle is detected

✓ *Sensor/actuator virtual machine*
- ♣ Manage interactions with sensors/actuators

✓ *Diagnosis:*
- ♣ Manage diagnosis interaction with HIS

- *Constraints:*

✓ The decomposition satisfies the constraint
- OS constraint-> satisfied if child module is

OS ♥ The constraint is satisfied by a single module
- Constraint is inherited by the child module

♥ The constraint is satisfied by a collection of child modules
- E.g., using client and server modules to satisfy a communication constraint

- *Quality scenarios:*
  - Quality scenarios also need to be verified and assigned to child modules
  - A quality scenario may be satisfied by the decomposition itself, i.e, no additional impact on child modules
  - A quality scenario may be satisfied by the decomposition but generating constraints for the children
  - The decomposition may be "neutral" with respect to a quality scenario
  - A quality scenario may not be satisfied with the current decomposition

7.3 FORMING THE TEAM STRUCTURES

- ♥ Once the architecture is accepted we assign teams to work on different portions of the design and development.
- ♥ Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.
- ♥ Each team then creates its own internal work practices.
- ♥ For large systems, the teams may belong to different subcontractors.
- ♥ Teams adopt "work practices' including
  - Team communication via website/bulletin boards
  - Naming conventions for files
  - Configuration/revision control system
  - Quality assurance and testing procedure

The teams within an organization work on modules, and thus within team high level of communication is necessary

7.4 CREATING A SKELETAL SYSTEM

✓ Develop a skeletal system for the incremental cycle.

✓ Classical software engineering practice recommends -> "stubbing out"

✓ Use the architecture as a guide for the implementation sequence

✓ First implement the software that deals with execution and interaction of architectural components
- Communication between components
- Sometimes this is just install third-party middleware

✓ Then add functionality
- By risk-lowering
- Or by availability of staff

Once the elements providing the next increment of functionality have been chosen, you can employ the uses structure to tell you what additional software should be running correctly in the system to support that functionality. This process continues, growing larger and larger increments of the system, until it is all in place.

CHAPTER 9: DOCUMENTING SOFTWARE ARCHITECTURES

9.1 USES OF ARCHITECTURAL DOCUMENTATION

- Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design.
- All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.
- One of the most fundamental rules for technical documentation in general, and software architecture documentation in particular, is to write from the point of view of the reader. Documentation that was easy to write but is not easy to read will not be used, and "easy to read" is in the eye of the beholder—or in this case, the stakeholder.
- Documentation facilitates that communication. Some examples of architectural stakeholders and the information they might expect to find in the documentation are given in Table 9.1.
- In addition, each stakeholders come in two varieties: seasoned and new. A new stakeholder will want information similar in content to what his seasoned counterpart wants, but in smaller and more introductory doses. Architecture documentation is a key means for educating people who need an overview: new developers, funding sponsors, visitors to the project, and so forth.

Table 9.1. Stakeholders and the Communication Needs Served by Architecture

Stakeholder	Use
Architect and requirements engineers who represent customer(s)	To negotiate and make tradeoffs among competing requirements
Architect and designers of constituent parts	To resolve resource contention and establish performance and other kinds of runtime resource consumption budgets
Implementors	To provide inviolable constraints (plus exploitable freedoms) on downstream development activities
Testers and integrators	To specify the correct black-box behavior of the pieces that must fit together
Maintainers	To reveal areas a prospective change will affect
Designers of other systems with which this one must interoperate	To define the set of operations provided and required, and the protocols for their operation
Quality attribute specialists	To provide the model that drives analytical tools such as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, verifiers, etc. These tools require information about resource consumption, scheduling policies, dependencies, and so forth. Architecture documentation must contain the information necessary to evaluate a variety of quality attributes such as security, performance, usability, availability, and modifiability. Analyses for each attributes have their own information needs.
Managers	To create development teams corresponding to work assignments identified, to plan and allocate project resources, and to track progress by the various teams
Product line managers	To determine whether a potential new member of a product family is in or out of scope, and if out by how much
Quality assurance team	To provide a basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions

Source: Adapted from [Clements 03]

9.2 VIEWS

The concept of a view, which you can think of as capturing a structure, provides us with the basic principle of documenting software architecture

> *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*

This principle is useful because it breaks the problem of architecture documentation into more tractable parts, which provide the structure for the remainder of this chapter:

- Choosing the relevant views
- Documenting view
- Documenting information that applies to more than one view

9.3 CHOOSING THE RELEVANT VIEWS

A view simply represents a set of system elements and relationships among them, so whatever elements and relationships you deem useful to a segment of the stakeholder community constitute a valid view. Here is a simple 3 step procedure for choosing the views for your project.

1. **Produce a candidate view list:**

   Begin by building a stakeholder/view table. Your stakeholder list is likely to be different from the one in the table as shown below, but be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views apply to every system, while others only apply to systems designed that way. Once you have 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.

Table 9.2. Stakeholders and the Architecture Documentation They Might Find Most Useful

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

2. **Combine views:**

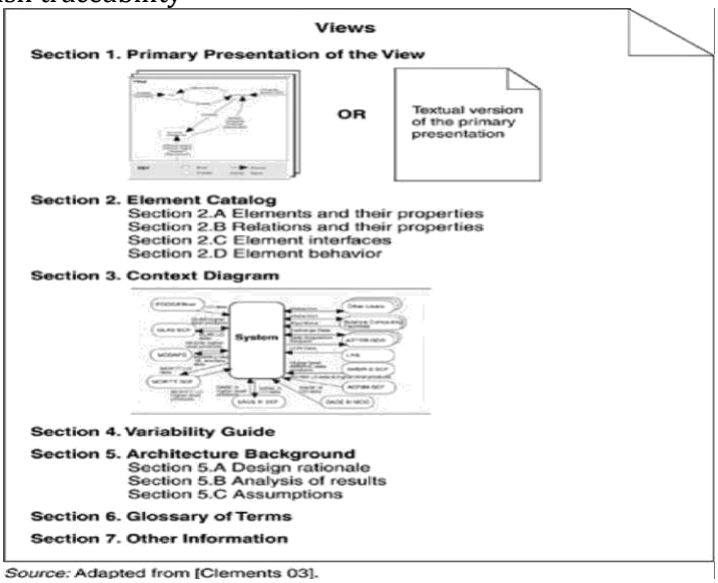
   The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger consistency. Next, look for the 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. The module decomposition view also pairs well with users or layered views. Finally, the deployment view usually combines well with whatever component-and-connector view shows the components that are allocated to hardware elements.

3. **Prioritize:**

   After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific project. But, remember that 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.

9.4 DOCUMENTING A VIEW

- **Primary presentation**- elements and their relationships, contains main information about these system , usually graphical or tabular.
- **Element catalog**- details of those elements and relations in the picture,
- **Context diagram**- how the system relates to its environment
- **Variability guide**- how to exercise any variation points a variability guide should include documentation about each point of variation in the architecture, including
  - The options among which a choice is to be made
  - The binding time of the option. Some choices are made at design time, some at build time, and others at runtime.
- **Architecture background** –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
- **Glossary of terms** used in the views, with a brief description of each.
- **Other information** includes management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability



Source: Adapted from [Clements 03].

DOCUMENTING BEHAVIOR

★ Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties .behavior description 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 ensemble of elements working in concert. 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, sequence diagrams and state charts are examples of behavioral descriptions. These notations are widely used.

DOCUMENTING INTERFACES

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

1. Interface identify

When an element has multiple interfaces, identify the individual interfaces to distinguish them. This usually means naming them. You may also need to provide a version number.

2. Resources provided:

The heart of an interface document is the resources that the element provides.

- ★ *Resource syntax* – this is the resource's signature
- ★ *Resource Semantics:*
  - Assignment of values of data
  - Changes in state
  - Events signaled or message sent
  - how other resources will behave differently in future
  - humanly observable results
- ★ *Resource Usage Restrictions*
  - initialization requirements
  - limit on number of actors using resource

3. Data type definitions:

If used if any interface resources employ a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that type. If it is defined by another element, then reference to the definition in that element's documentation is sufficient.

4. Exception definitions:

These describe exceptions that can be raised by the resources on the interface. Since the same exception might be raised by more than one resource, if it is convenient to simply list each resource's exceptions but define them in a dictionary collected separately.

5. Variability provided by the interface.

Does the interface allow the element to be configured in some way? These configuration parameters and how they affect the semantics of the interface must be documented.

6. Quality attribute characteristics:

The architect needs to document what quality attribute characteristics (such as performance or reliability) the interface makes known to the element's users

7. Element requirements:

What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same as for resources provided: syntax, semantics, and any usage restrictions.

8. Rationale and design issues:

Why these choices the architect should record the reasons for an elements interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternatives designs were considered.

9. Usage guide:

Item 2 and item 7 document an element's semantic information on a per resource basis. This sometimes falls short of what is needed. In some cases semantics need to be reasoned about in terms of how a broad number of individual interactions interrelate.

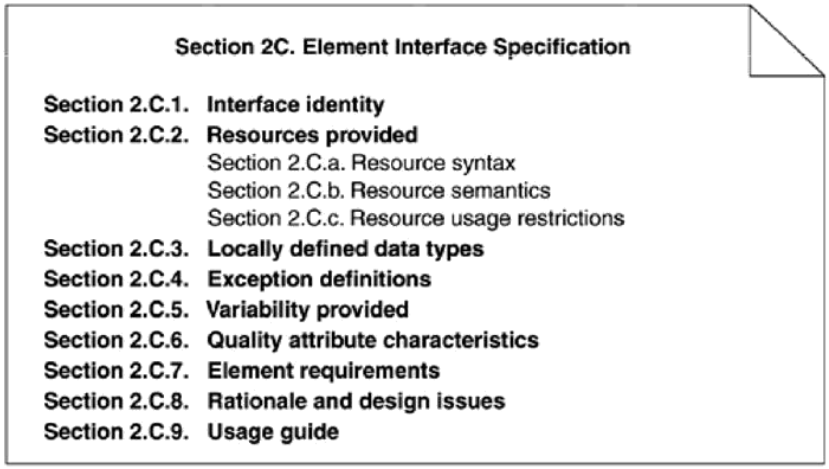
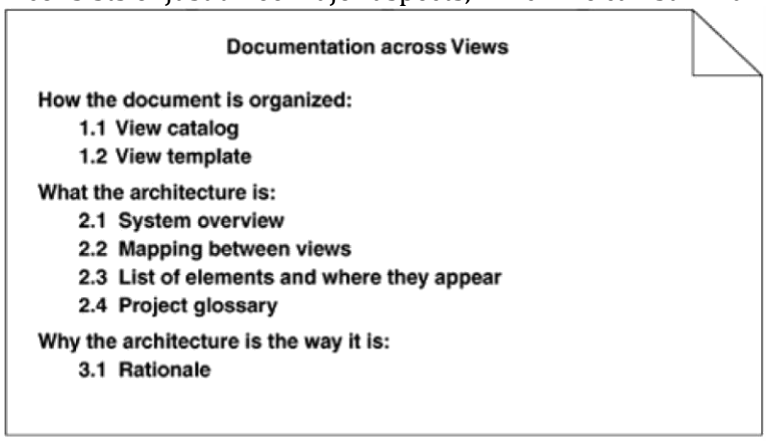


Figure 9.2. The nine parts of interface documentation

9.5 DOCUMENTATION ACROSS VIEWS

Cross-view documentation consists of just three major aspects, which we can summarize as how-what-why:



Source: Adapted from [Clements 03].

Figure 9.3. Summary of cross-view documentation

HOW THE DOCUMENTATION IS ORGANIZED TO SERVE A STAKEHOLDER

Every suite of architectural documentation needs an introductory piece to explain its organization to a novice stakeholder and to help that stakeholder access the information he or she is most interested in. There are two kinds of "how" information:

★ **View Catalog**

A view catalog is the reader's introduction to the views that the architect has chosen to include in the suite of documentation.

There is one entry in the view catalog for each view given in the documentation suite. Each entry should give the following:

- The name of the view and what style it instantiates
- A description of the view's element types, relation types, and properties
- A description of what the view is for
- Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document

★ **View Template**

A view template is the standard organization for a view. It helps a reader navigate quickly to a section of interest, and it helps a writer organize the information and establish criteria for knowing how much work is left to do.

WHAT THE ARCHITECTURE IS

This section provides information about the system whose architecture is being documented, the relation of the views to each other, and an index of architectural elements.

♣ **System Overview**

This is a short prose description of what the system's function is, who its users are, and any important background or constraints. The intent is to provide readers with a consistent mental model of the system and its purpose. Sometimes the project at large will have a system overview, in which case this section of the architectural documentation simply points to that.

♣ **Mapping between Views**

Since all of the views of an architecture describe the same system, it stands to reason that any two views will have much in common. Helping a reader of the documentation understand the relationships among views will give him a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the relationship by providing mappings between views is the key to increased understanding and decreased confusion.

♣ **Element List**

The element list is simply an index of all of the elements that appear in any of the views, along with a pointer to where each one is defined. This will help stakeholders look up items of interest quickly.

- **Project Glossary**

  The glossary lists and defines terms unique to the system that have special meaning. A list of acronyms, and the meaning of each, will also be appreciated by stakeholders. If an appropriate glossary already exists, a pointer to it will suffice here.

WHY THE ARCHITECTURE IS THE WAY IT IS: RATIONALE

Cross-view rationale explains how the overall architecture is in fact a solution to its requirements. One might use the rationale to explain:

- The implications of system-wide design choices on meeting the requirements or satisfying constraints.
- The effect on the architecture when adding a foreseen new requirement or changing an existing one.
- The constraints on the developer in implementing a solution.
- Decision alternatives that were rejected.

In general, the rationale explains why a decision was made and what the implications are in changing it.

UNIT 8 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1.	What are the three steps for choosing views for a project?	Dec 09	6
2.	Write a note on view catalog	Dec 09	4
3.	What are the options for representing connectors and systems in UML? ★ **OUT OF SYLLABUS**	Dec 09	10
4.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 10	8
5.	What are the suggested standard organization points for interface documentation?	June 10	12
6.	List the steps of ADD	Dec 10	4
7.	Write a note on creating a skeletal system	Dec 10	6
8.	What are the uses of architectural documentation? Bring out the concept of view as applied to architectural documentation.	Dec 10	10
9.	Briefly explain the different steps performed while designing an architecture using the ADD method	June 11	10
10.	write short notes on: i)forming team structures ii)documenting across views iii)documenting interfaces	June 11	10
11.	Explain the steps involved in designing an architecture, using the attribute driven design	Dec 11	10
12.	"Architecture serves as a communication vehicle among stakeholders. Documentation facilitates that communication." Justify.	Dec 11	10
13.	List the steps of ADD method of architectural design	June 12	6
14.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 12	6
15.	What are the suggested standard organization points for view documentation?	June 12	8