RGPVNOTES.IN

Please do not share these notes on apps like WhatsApp or Telegram.

The revenue we generate from the ads we show on our website and app funds our services. The generated revenue **helps us prepare new notes and improve the quality of existing study materials**, which are available on our website and mobile app.

If you don't use our website and app directly, it will hurt our revenue, and we might not be able to run the services and **have to close them**. So, it is a humble request for all to **stop sharing the study material** we provide on various apps. Please **share the website's URL instead.**

Course Contents:

Unit 5. Software Architecture documentation: principles of sound documentation, refinement, context diagrams, variability, software interfaces. Documenting the behavior of software elements and softwaresystems, documentation package using a seven-part template.

Unit-5

Software Architecture Documentation

The software architecture document provides a comprehensive overview of the architecture of the software system or may be consider a map of the software. We can use it to see, how the software is structured. It mainly helps to understand the software's modules and components without digging into the code. It serves as a tool to communicate with others like developers and non-developers—about the software.

Below are the three primary goals for architectural documentation:

- **Knowledge sharing-** It is suitable to transfer knowledge between people working in different functional areas of the project, as well as for knowledge transfer to new participants.
- **Communication-** Documentation is the starting point for interaction between different stakeholders. It helps to share the ideas of the architect to the developers.
- **Analyses-** Documentation is also a starting point for future architectural reviews of the project.

Two approaches to create software architecture

There are two well-known approaches- top-down and bottom-up to create software and its architecture. When we start from general idea and iteratively decompose system into smaller components, this is called the top-down approach.

Alternatively, we can start with defining project goals and iteratively add more and more small pieces of functionality to the system. All those small elements and their relations will compose system with its architecture, this is called the bottom-up approach.

Principles of Sound Documentation

Principle 1: Write from the point of view of the reader

The Web-based design includes a component to support role-based user login and access control. Within the context of a particular document, once a stakeholder logs in, the system generates a view tailored to that stakeholder's needs and access rights. The view might provide a part of the document as a Web form to the architect and that same part as a Portable Document Format (PDF) file for a developer; a junior architect is likely to benefit from guidance at a level that would cause a senior architect frustration. Clickable pop-up windows show tips relevant to the role of the currently logged-in stakeholder. For instance, if a junior architect is creating a primary presentation, the tip might provide guidance on organizational standards or notations, while the tip for a developer would provide the semantics of the notation used by the architect.

Principle 2: Avoid unnecessary repetition

The database back end of the system allows each piece of information to be created and maintained as individual files and accessed by multiple users in a variety of contexts. Hyperlinking allows the illusion of redundancy while completely removing its need. A glossary captures definitions in a single place and provides a readily accessible and consistent reference for all stakeholders. Similarly, an acronym list is

created and is readily accessible at any time. Diagrams and other files can be created once and referenced from multiple locations throughout the document.

Principle 3: Avoid ambiguity

While the use of natural language always allows for varied interpretation, the design reinforces the need for precision and clarity. In addition, the use of Unified Modelling Language (UML) and other more formal languages is supported by way of file upload.

Principle 4: Use a standard organization

A template, by nature, enforces the use of a standard organization. With this system's design, we use Web forms as templates that provide rigid structure to the document's various elements.

Principle 5: Record rationale

The design provides entries specific to recording rationale in appropriate places and encourages their use by querying the author if this section of the form is not used.

Principle 6: Keep documentation current but not too current

Keeping the documentation current is simple in Web-based documentation; however, issues associated with "too current" are more pressing in this type of environment. The Web-based documentation system is best backed with a configuration management system that makes available various levels of currency based on the user's role.

Principle 7: Review documentation for fitness of purpose

The Web-based design supports easy and immediate feedback on both the documentation's form and content as email links on every Web page.

Refinement

Refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

At the early steps of the refinement process the software engineer does not necessarily know how the software will perform what it needs to do. This is determined at each successive refinement step, as the design and the software is elaborated upon.

Refinement can be seen as the compliment of abstraction. Abstraction is concerned with hiding lower levels of detail; it moves from lower to higher levels. Refinement is the movement from higher levels of detail to lower levels. Both concepts are necessary in developing software.

Context Diagram

The system context diagram (also known as a level 0 DFD) is the highest level in a data flow diagram and contains only one process, representing the entire system, which establishes the context and boundaries of the system to be modelled. It identifies the flows of information between the system and external entities (i.e. actors). A context diagram is typically included in a requirements document. It must be read by all project stakeholders and thus should be written in plain language, so the stakeholders can understand items.

Purpose of a System Context Diagram

The objective of the system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of systems requirements and constraints. A system context diagram is often used early in a project to determine the scope under investigation. Thus, within the document. A system context diagram represents all external entities that may interact with a system. The entire software system is shown as a single process. Such a diagram pictures the system at the centre,

with no details of its interior structure, surrounded by all its External entities, interacting systems, and environments.

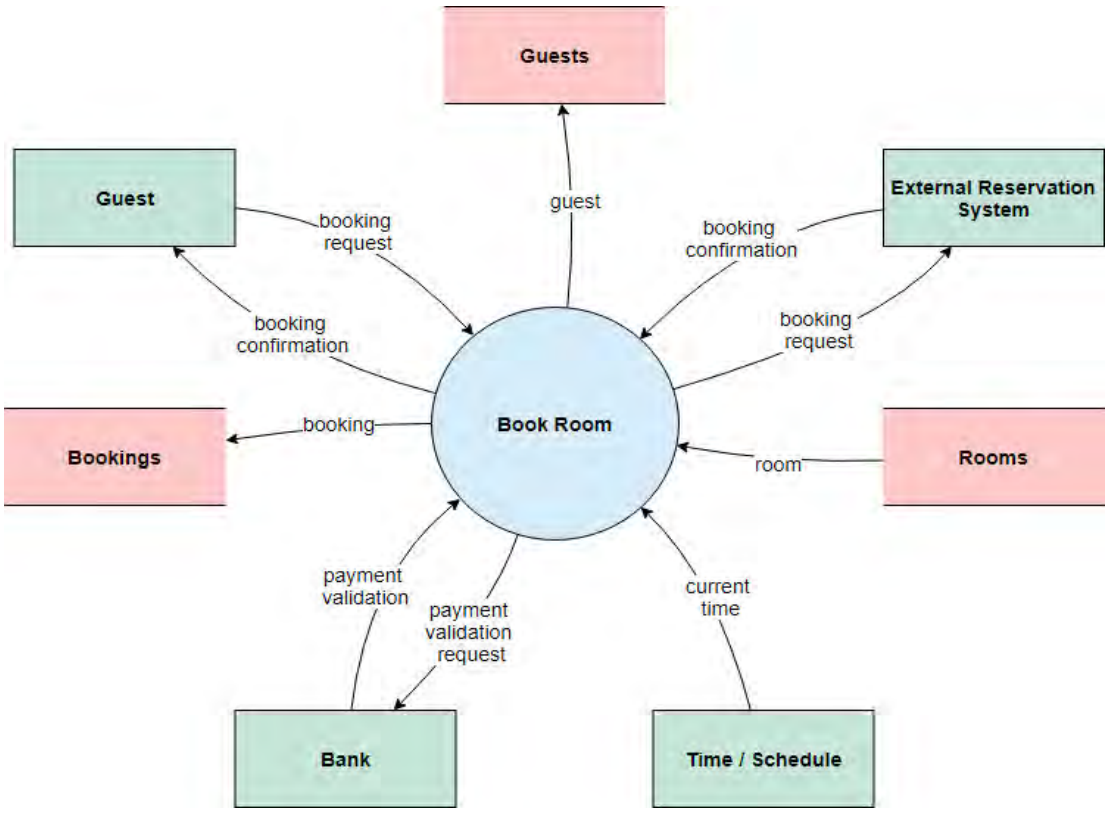

Figure 5.1: Example of context diagram for hotel reservation system

Variability

Variability is the ability of a software artifact to be changed (e.g., configured, customized, extended, adapted) for a specific context, in a pre-planned manner. This means, variability can be understood as "anticipated change".

- It helps manage commonalities and differences between systems.
- It supports the development of different versions of software by allowing the implementation of variants within systems.
- It facilitates planned reuse of software artifacts in multiple products,
- It allows the delay of design decisions to the latest point that is economically feasible.
- It supports runtime adaptations of deployed systems.

Reasons for variability include the deferral of design decisions, multiple deployment / maintenance scenarios.

Software Interfaces

Software interfaces (programming interfaces) are the languages, codes, and messages that programs use to communicate with each other and to the hardware. An interface is a boundary across which two independent entities meet and interact or communicate with each other. The characteristics of an interface depend on the view type of its element. If the element is a component, the interface represents a specific point of its potential interaction with its environment. If the element is a module, the interface is a definition of services. There is a relation between these two kinds of interfaces, just as there is a relation between components and modules.

By the element's environment, we mean the set of other entities with which it interacts. We call those other entities actors: An element's actors are the other elements, users, or systems with which it interacts. In general, an actor is an abstraction for external entities that interact with the system. Interaction is part of the element's interface. Interactions can take a variety of forms. Most involve the transfer of control

and/or data. Some are supported by standard programming-language constructs, such as local or remote procedure calls (RPCs), data streams, shared memory, and message passing.

Some principles about interfaces:

- All elements have interfaces. All elements interact with their environment.
- An element's interface contains view-specific information.
- Interfaces are two ways.
- An element can have multiple interfaces.
- An element can interact with more than one actor through the same interface.

An interface is documented with an interface specification: An interface specification is a statement of what an architect chooses to make known about an element in order for other entities to interact or communicate with it.

Documenting the behaviour of software elements and software systems

A software component simply cannot be differentiated from other software elements by the programming language used to implement the component. The difference must be in how software components are used. Software comprises many abstract, quality features, that is, the degree to which a component or process meets specified requirement for example, an efficient component will receive more use than a similar, inefficient component. It would be inappropriate, however, to define a software component as "an efficient unit of functionality." Elements that comprise the following definition of the term software component are described in the "Terms" sidebar. A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model. A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

The main goal of effective documentation is to ensure that developers and stakeholders are headed in the same direction to accomplish the objectives of the project. To achieve them, plenty of documentation types exist.

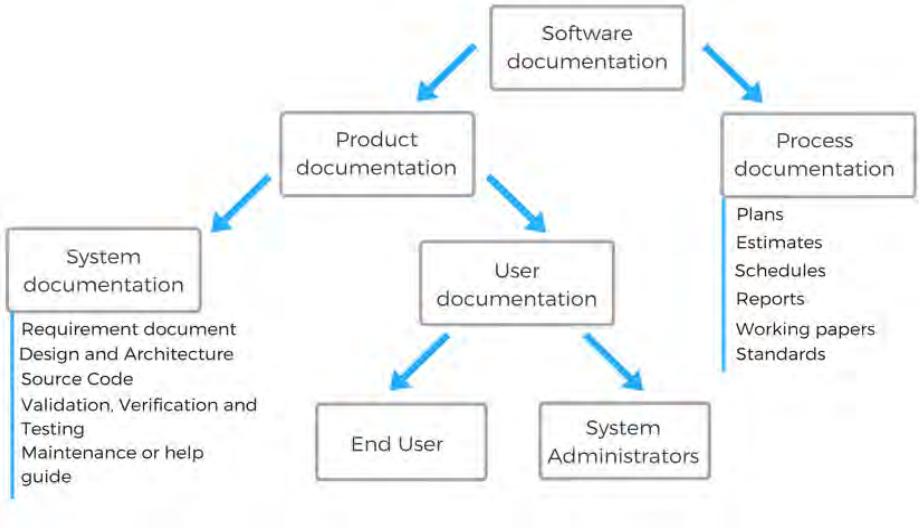

Figure 5.2: Types of Documentation

All software documentation can be divided into two main categories:

- Product documentation
- Process documentation

Product documentation describes the product that is being developed and provides instructions on how to perform various tasks with it. Product documentation can be broken down into:

- System documentation and
- User documentation

System documentation represents documents that describe the system itself and its parts. It includes requirements documents, design decisions, architecture descriptions, program source code, and help guides.

User documentation covers manuals that are mainly prepared for end-users of the product and system administrators. User documentation includes tutorials, user guides, troubleshooting manuals, installation, and reference manuals.

Process documentation represents all documents produced during development and maintenance that describe well, process. The common examples of process documentation are project plans, test schedules, reports, standards, meeting notes, or even business correspondence.

The main difference between process and product documentation is that the first one record the process of development and the second one describes the product that is being developed.

Documentation Package using a seven-part template

Each ECS view is presented as a number of related view packets. A view packet is a small, relatively self-contained bundle of information about the system or a particular part of the system, rendered in the languageelement and relation typesof the view to which it belongs. Two view packets are related to each other as either parent/childbecause one shows a refinement of the information in the otheror as siblingsbecause both are children of another view packet.

1. A primary presentation that shows the elements and their relationships that populate the view packet. The primary presentation contains the information important to convey about the system, in the vocabulary of that view, first.
   The primary presentation is usually graphical. If so, the presentation will include a key that explains the meaning of every symbol used. The first part of the key identifies the notation, If a defined notation is being used, the key will name it and cite the document that defines it or defines the version of it being used. If the notation is informal, the key will say so and proceed to define the symbols and the meaning, if any, of colours, position, or other information-carrying aspects of the diagram.
2. Element catalog detailing at least those elements depicted in the primary presentation and others that were omitted from the primary presentation. Specific parts of the catalog include-
   - Elements and their properties- This section names each element in the view packet and lists the properties of that element. For example, elements in a module decomposition view have the property of "responsibility," an explanation of each module's role in the system, and elements in a communicating-process view have timing parameters, among other things, as properties.
   - Relations and their properties -Each view has a specific type of relation that it depicts among the elements in that view. However, if the primary presentation does not show all the relations or if there are exceptions to what is depicted in the primary presentation, this section will record that information.
   - Element interface - An interface is a boundary across which elements interact or communicate with each other. This section is where element interfaces are documented.
   - Element behaviour- Some elements have complex interactions with their environment and for purposes of understanding or analysis, the element's behaviour is documented.
3. Context diagram showing how the system depicted in the view packet relates to its environment.
4. Variability guide showing how to exercise any variation points that are a part of the architecture shown in this view packet.
5. Architecture background explaining why the design reflected in the view packet came to be. Itexplains why the design is as it is and to provide a convincing argument that it is sound. Architecture background includes-

- Rationale- It explains why the design decisions reflected in the view packet were made and gives a list of rejected alternatives and why they were rejected. This will prevent future architects from pursuing dead ends in the face of required changes.
- Analysis results -This document the results of analyses that have been conducted, such as the results of performance or security analyses, or a list of what would have to change in the face of a particular kind of system modification.
- Assumptions - This document any assumptions the architect made when crafting the design. Assumptions are generally about either environment or need.
- Environmental assumptions document what the architect assumes is available in the environment that can be used by the system being designed. They also include assumptions about invariants in the environment.

6. Other information – It includesno architectural and organization-specific information. "Other information" will usually include management or configuration control information, change histories, bibliographic references or lists of useful companion documents, mapping to requirements, and the like.
7. Related view packets - It will name other view packets that are related to the one being described in a parent/child or sibling capacity.

RGPVNOTES.IN

Thank you for using our services. Please support us so that we can improve further and help more people.
https://www.rgpvnotes.in/support-us

If you have questions or doubts, contact us on
WhatsApp at +91-8989595022 or by email at hey@rgpvnotes.in.

For frequent updates, you can follow us on
Instagram: https://www.instagram.com/rgpvnotes.in/.