TRINITY INSTITUTE OF TECHNOLOGY AND RESEARCH BHOPAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Software Engineering & Project Management**

Subject Code: **CS-403**

Semester: **4th**

UNIT-II

REQUIREMENT ELICITATION ANALYSIS & SPECIFICATION

Requirement: -

The process to gather the software requirements from client, analyze and document them is known as requirement engineering.

The goal of requirement engineering is to develop and maintain sophisticated and descriptive '"system Requirements "specification' document.

Types of Requirements: -

- **User Requirements**: It is a collection of statement in natural language and description of the services the system provides and its operational limitation. It is written for customer.
- **System Requirement**: It is a structured document that gives the detailed description of the system services. It is written as a contract between client and contractor.

Software Requirement Specification: -

SRS is a document created by system analyst after the requirements are collected from various stakeholders. SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirements: -

We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.

Broadly software requirements should be categorized in two categories:

1. Functional Requirement
2. Non Functional Requirement

FUNCTIONAL REQUIREMENTS:

It should describe all requirement functionality or system services. The customer should provide statement of service. it should be clear how the system should be reacting to particular input and how a particular system

should behave in particular situation. Functional requirement is heavily depending upon he type of software expected users and the type of system where the software is used. It describes system services in detail.

NON-FUNCTIONAL REQUIREMENTS:

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software; which users make assumption of. Non -functional are more critical than functional requirement if the non-functional requirement do not meet then the complete system is of no use.

Some typical **non-functional requirements** are:

- **Product requirement**

Specify how a livered product should behave in particular way.

- Eg: efficiency, Usability, Reliability, portability

- **Organizational requirement**- The requirements which are unwelcome effect of organizational policies and procedures come under this category.
  - Eg: Delivery, implementation, standard
- **External requirement**-
  These requirements arise due to the factors that are external of the system and its developed process.
  - Eg: Interoperability, ethical, safety.

REQUIREMENT SOURCES AND ELICITATION TECHNIQUES:

Requirements Sources:-

In a typical system, there will be many sources of requirements and it is essential that all potential sources are identified and evaluated for their impact on the system. This subtopic is designed to promote awareness of different requirements sources and frameworks for managing them.

The main points covered are:

- **Goals: -**The term 'Goal' (sometimes called 'business concern' or 'critical success factor') refers to the overall, high-level objectives of the system. Goals provide the motivation for a. Requirements engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- **Domain knowledge**: - The requirements engineer needs to acquire or to have available knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements and sometimes to act as a 'user' champion.
- **System stakeholders: -**Many systems have proven unsatisfactory because they have stressed the requirements for one group of stakeholders at the expense of others. Hence, systems are delivered that are hard to use or which subvert the cultural or political structures of the customer organization. The requirements engineer needs to identify represent and manage the 'viewpoints' of many different types of stakeholder.
- **The operational environment: -**Requirements will be derived from the environment in which the software will execute. These may be, for example, timing constraints in a real-time system or interoperability constraints in an office environment. These must be actively sought because they can greatly affect system feasibility, cost, and restrict design choices.
- **The organizational environment: -**Many systems are required to support a business process and this may be conditioned by the structure, culture and internal politics of the organization. The requirements engineer needs to be sensitive to these since, in general, new software systems should not force unplanned change to the business process.

Elicitation techniques: -

When the requirements sources have been identified the requirements, engineer can start eliciting

requirements from them. It also means requirement discovery. This subtopic concentrates on techniques for getting human stakeholders to articulate their requirements. This is a very difficult area and the requirements engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity and that even if cooperative and articulate stakeholders are available, the requirements engineer has to work hard to elicit the right information. A number of techniques will be covered, but the principal ones are:

- **Interviews*:-***Interviews are a 'traditional' means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.

- **Scenarios**: - Scenarios are valuable for providing context to the elicitation of users' requirements. They allow the requirements engineer to provide a framework for questions about users' tasks by permitting 'what if?' and 'how is this done?' questions to be asked. (Conceptual modeling) because recent modeling notations have attempted to integrate scenario notations with object-oriented analysis techniques.
- **Prototypes*: -***Prototypes are a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing a context within which users better understand what information they need to provide. There is a wide range of prototyping techniques, which range from paper mock-ups of screen designs to beta-test versions of software products. There is a strong overlap with the use of prototypes for requirements validation.
- **Facilitated meetings*: -***The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight to their requirements than by working individually. They can brainstorm and refine ideas that may be difficult to surface using (e.g.) interviews.
- **Observation*: -***The importance of systems' context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. The requirements engineer learns about users' tasks by immersing themselves in the environment and observing how users interact with their systems and each other. These techniques are relatively new and expensive but are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

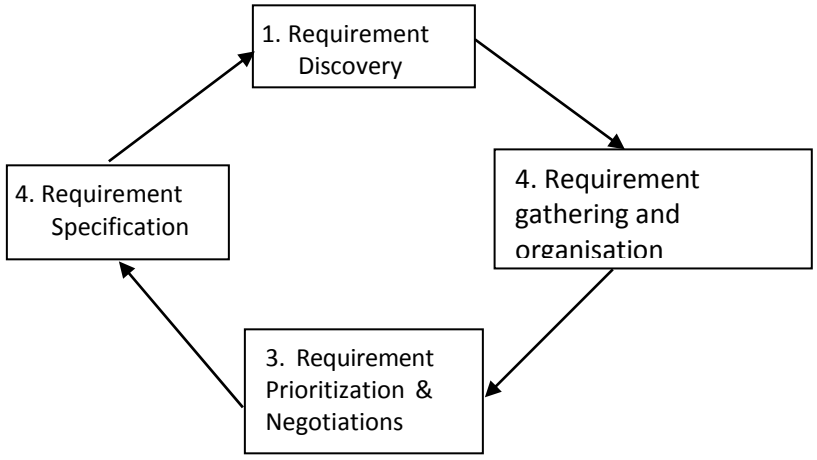


Figure 2.1 Requirement Prototype

ANALYSIS MODELING FOR FUNCTION ORIENTED AND OBJECT ORIENTED SOFTWARE DEVELOPMENT :

Analysis model: -

The analysis model must achieve three primary objectives:

- To describe what the customer requires(analysis)
- To establish a basis for the creation of software with a combination of text and design are used to represent the software requirement.
- To define a set of requirements that can be validated once the software is built.

The elements of analysis model: -

At the core of the model lies the data dictionary—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The entity relation diagram (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

The data flow diagram (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and sub functions) that transform the data flow.

The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a process specification (PSPEC).

The **state transition diagram** (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control flow in the **control specification** (CSPEC). **Process specification** describes each function in DFD. **Data object description** of various data object used.

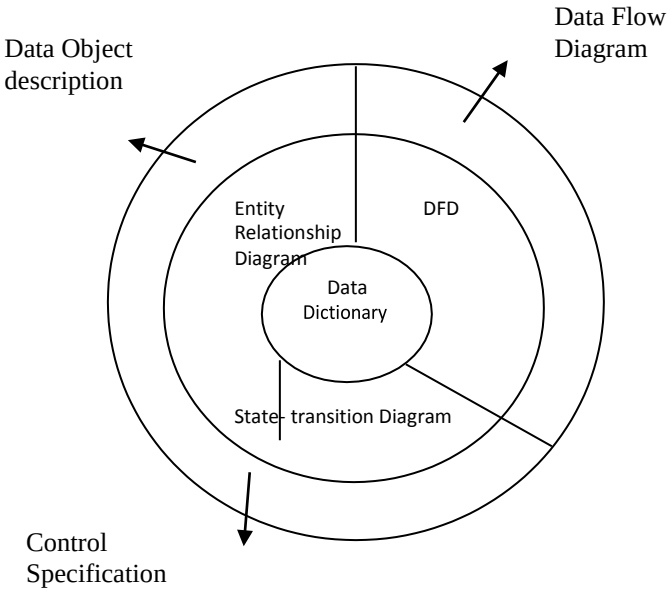


Figure 2.2: State Transition Diagram

Analysis and Modeling: -

Structured Approach

Data Modeling -> ERD

Functional Modeling -> DFD

Behavior Modeling -> State chart diagram

Object Oriented Approach (UML)

- Use case
- Class diagram
- Activity diagram
- Sequence diagram
- Deployment diagram
- Component diagram

The structured approach - plan the right way first

- plans to avoid crisis
- covers all eventualities
- useful for team working
- shorter in the end

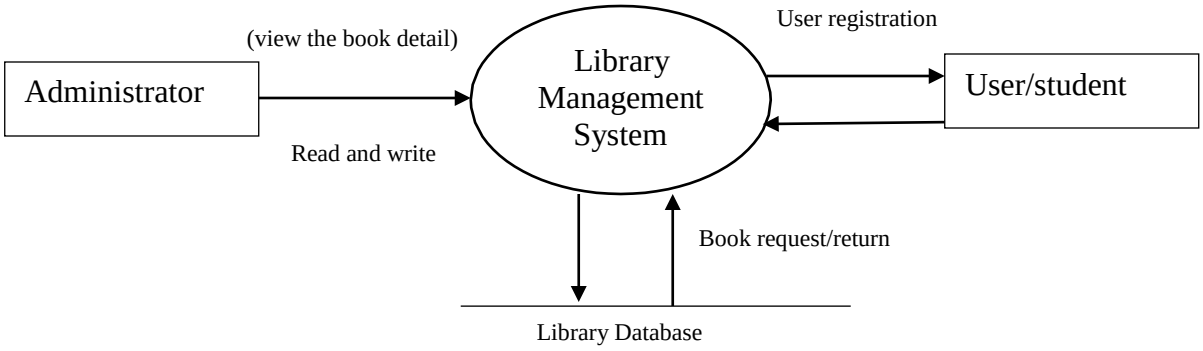
In SA data object are modeled in a way in which data attributes and their relationship is defined in structured approach.

Data Flow Diagram (DFD):

- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system
- Performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input
- Data to the system, various processing carried out on these data, and the output
- Data generated by the system. A DFD model uses a very limited number of primitive symbols as shown in below Figure to represent the functions performed by a system and the data flow among these functions.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.

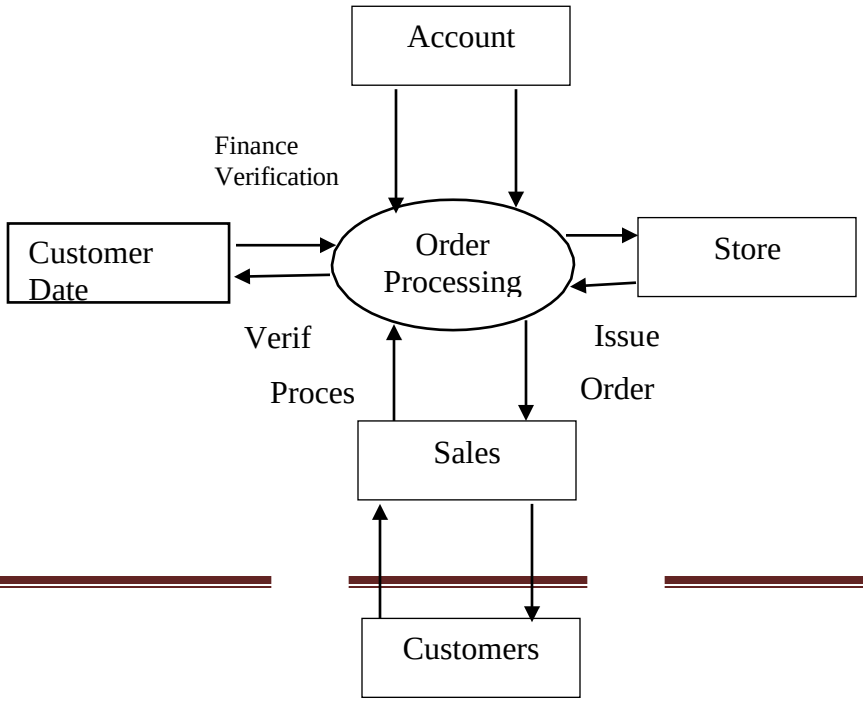


Figure 2.4: Level 1 DFD

Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1.
Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

Behavior Modeling: -

Structure Charts: -

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. The basic building blocks which are used to design structure charts are the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Transform Analysis: -

Transform analysis identifies the primary functional components (modules) and the high-level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists etc.). Each input portion is called an afferent branch. The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

Example: Structure chart for the RMS software

For this example, the context diagram was drawn earlier.

Object-oriented Software Development:-

Object-oriented design: -

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

What is a model: -

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models

are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model:

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following**:**

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

Unified Modeling Language (UML):-

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

UML Diagrams:

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external user's view of the system in terms of the functionalities offered by the system. The user's view is a black-box view of the system where the internal structure, the dynamic behavior of

different system components, the implementation etc. are not visible.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

USE-CASE MODELING:

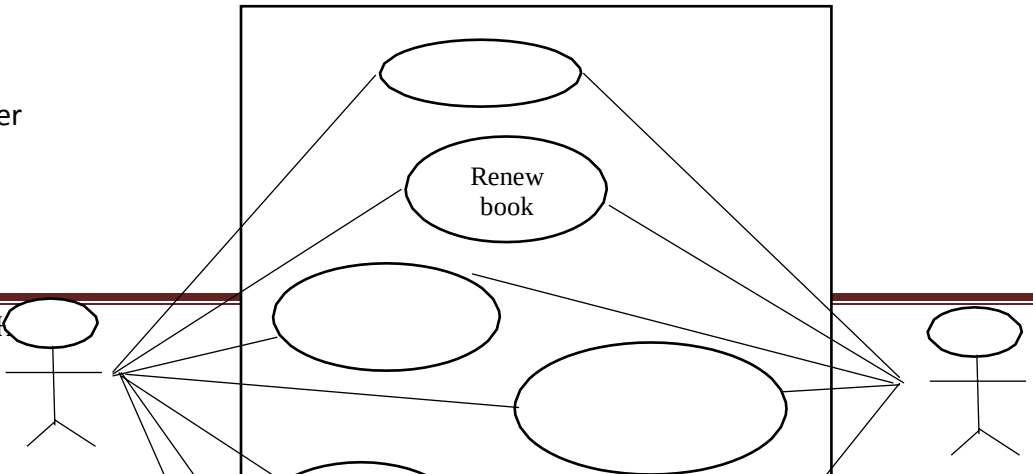
Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). Use case modeling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

Use-case Model: -

The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?"

Thus, for the Library Information System (LIS), the use cases could be**:**

- issue-book
- query-book
- return-book
- create-member
- add-book etc



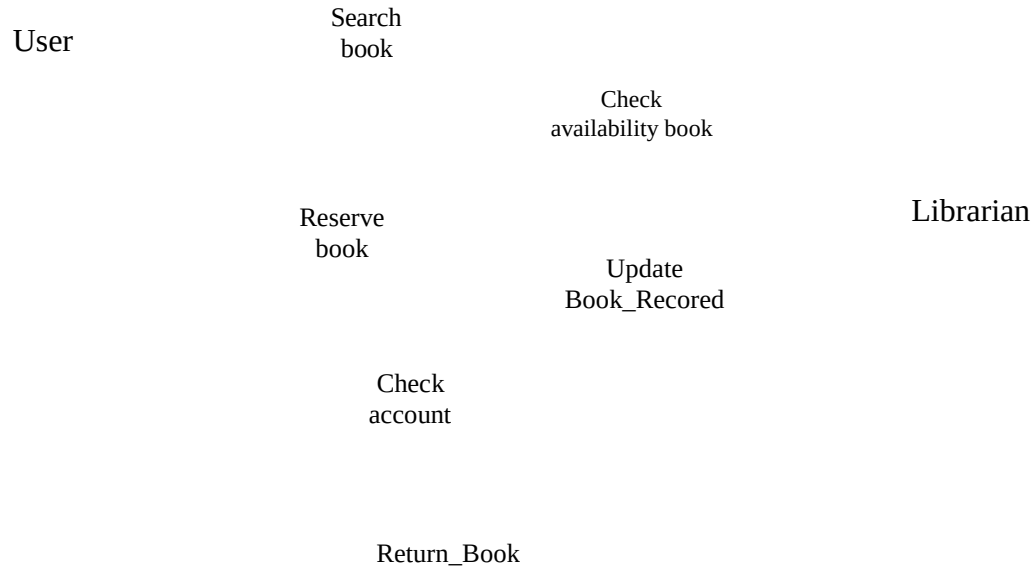


Figure 2.5: Use Case Diagram

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior

into transactions, such that each transaction performs some useful action from the user's point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

Purpose of use cases: -

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction.

Representation of use cases: -

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

Text Description: -

Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain related documents which may be useful to understand the system operation.

SYSTEM AND SOFTWARE REQUIREMENT SPECIFICATIONS:

Software Specification: -

Software Specification is an activity that is use to describe he thing you are trying to achieve to establish what services are required from the system and limitation on the system operation and development. His activity is often called Requirement Engineering. Requirement Engineering is a particularly critical stage of the software process as error at this stage certain to happen lead to later problem in the system design and implementation.

There are four main phases in the Requirement Engineering process:

- Feasibility study: user satisfaction, cost estimation.
- Requirement elicitation analysis: meeting for description of development.
- Requirement Specification: is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. 2 types of requirements may be including in his document.
- User Requirements (b) System Requirement
- Requirements Validation: this activity checks the requirement for realism, consistency and completeness.

Software Requirement Specification [SRS]: -

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

The Software Requirements Specification is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

- A description of each function required to solve the problem is presented in the Functional Description. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.
- The Behavioral Description section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.
- Validation Criteria is probably the most important and, ironically, the most often neglected section of the Software Requirements Specification. How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an

implicit review of all other requirements. It is essential that time and attention be given to this section.

- Finally, the specification includes a Bibliography and Appendix. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the Software Requirements Specification may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a Preliminary User's Manual. The Preliminary User's Manual presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

Characteristics of SRS:

- **Correct**: Requirement must be correctly mentioned and realistic by nature.
- **Unambiguous**: Transparent and plain SRS must be written.
- **Complete**: To make the SRS complete I should be specified what a software designer wants to create on software.
- **Consistent**: If there are not conflicts in the specified requirement then SRS is said to be consistent.
- **Stability**: The SRS must contain all the essential requirement. Each requirement must be clear and

explicit.

- **Verifiable**: the SRS should be written in such a manner that the requirement that is specified within it must be satisfied by the software.
- **Modifiable**: It can easily modify according to user requirement.
- **Traceable**: If origin of requirement is properly given of the requirement are correctly mentioned then such a requirement is called as traceable requirement.

REQUIREMENTS VALIDATION:

The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review. The review team includes system engineers, customers, users, and other stakeholders who examine the system specification 5 looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies , conflicting requirements, or unrealistic (unachievable) requirements.

Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions. The following questions represent a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives? • Is the system specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with system performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Requirements Management: -

Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds

Once requirements have been identified, traceability tables are developed. Shown schematically in Figureure, each traceability table relates identified requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

- Features traceability table. Shows how requirements relate to important customer observable system/product features.
- Source traceability table. Identifies the source of each requirement
- Dependency traceability table. Indicates how requirements are related to one another.
- Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.

- Interface traceability table. Shows how requirements relate to both internal and interfaces.

In many cases, these traceability tables are maintained as part of a requirements database so that they may be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

TRACEABILITY:

Traceability is a property of an element of documentation or code that indicates the degree to which it can be traced to its origin or "reason for being". Traceability also indicates the ability to establish a predecessor-successor relationship between one work product and another.

A work product is said to be traceable if it can be proved that it complies with its specification. For example, a software design is said to be traceable if it satisfies all the requirements stated in the software requirements specification. Examples of traceability include:

- External source to system requirements
- System requirements to software requirements
- Software requirements to high level design
- High level design to detailed design
- Detailed design to code
- Software requirement to test case.

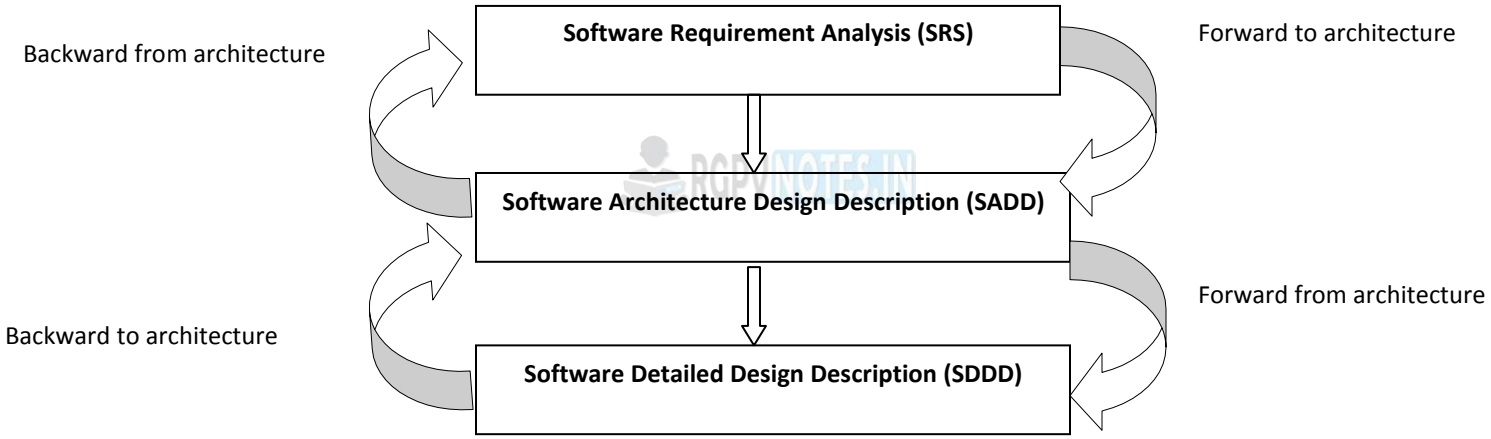


Figure 2.6: Tracing a Software Architecture Design Description