UNIT - III : SOFTWARE REQUIREMENTS AND SYSTEM MODELS

Functional and Non Functional Requirements

Requirements analysis is an essential process that enables the success of a system or software project to be assessed. Requirements are generally split into two types: Functional and Non-functional requirements. functional requirements define the specific behaviour or functions of a system. In contrast, non-functional requirements specify how the system performs its tasks, focusing on attributes like performance, security, scalability, and usability.

Functional Requirements

These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract.

- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.
- They are the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.
- **Testing:** Functional testing (e.g., API testing, integration testing) verifies that the system behaves as expected.

Example: Online Banking System

- Users should be able to log in with their username and password.
- Users should be able to check their account balance.
- Users should receive notifications after making a transaction.

Non-Functional Requirements

These are the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to another. They are also called non-behavioral requirements. They deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability

- Performance
- Reusability
- Flexibility

Example: Online Banking System

- The system should respond to user actions in less than 2 seconds.
- All transactions must be encrypted and comply with industry security standards.
- The system should be able to handle 100 million users with minimal downtime.

- **Testing:**
-
- Non-functional testing (e.g., performance testing, security testing, usability testing) verifies that the system meets these quality standards.
-

Importance of both types of requirements

Both functional and non-functional requirements are vital for a successful software product. Functional requirements ensure the system performs necessary tasks and meets business and user objectives, while non-functional requirements dictate the system's quality and efficiency, impacting user satisfaction and long-term viability. A system that functions correctly but is slow or difficult to use can lead to user dissatisfaction.

Balancing the two

Balancing these requirements is crucial, as improving one aspect, like security (non-functional), might affect another, like performance (non-functional). Teams need to make informed decisions and manage potential tradeoffs.

In essence, functional requirements define *what* the system does, while non-functional requirements define *how well* it does it. A well-designed system addresses both to create a product that is both correct and provides a positive user experience.

User requirements

User requirements in software engineering describe what the **end-user needs and expects from a software system.** These requirements are typically **expressed in natural language and focus on the user's perspective and desired functionalities**, rather than technical implementation details. They serve as a bridge between the user's vision and the technical development process.

Key aspects of user requirements

- **Focus on the "what" and "why":** User requirements describe **what the system needs to do from the user's perspective,** without **delving into the technical details of how it will be** implemented. They focus on the **user's goals and the value the system provides**.
- **Expressed in user-friendly language:** They are typically **written in natural language**, making them easily understandable by both **technical and non-technical stakeholders**.
- **Foundation for design and development:** User requirements serve as a essential starting **point for designing and developing the software**, ensuring that the **final product aligns with user expectations.**
- **Examples of documentation methods**: User requirements can be documented using techniques like **user stories, use cases, or within a User Requirements Document (URD).**

Importance of user requirements

Defining user requirements is important for several reasons:

- This approach ensures the software is **user-centric, leading to greater user satisfaction**.
- It **reduces rework and errors by minimizing uncertainty**.
- It **improves communication and collaboration** among **stakeholders.**
- It provides a basis for **testing and validating the system** against **user expectations**.

Example of User Requirements:

Consider a new online food ordering system. User requirements might include:

- **As a customer, I want to be able to browse a menu of available restaurants and their dishes.** This focuses on the user's action of browsing and the content they expect to see.
- **As a customer, I want to be able to add multiple dishes from different restaurants to a single order.** This specifies a desired functionality related to order creation.
- **As a customer, I want to be able to pay for my order using various payment methods, including credit card and mobile payment options.** This outlines the expected payment flexibility.
- **As a restaurant owner, I want to be able to update my menu items and prices in real-time.** This addresses the needs of a different user role within the system.

- **As a delivery driver, I want to be able to view my assigned deliveries and customer addresses on a map.** This highlights the need for location-based features for a specific user.

These examples illustrate how user requirements focus on the "what" (what the user wants to achieve) rather than the "how" (how the system will technically implement it). They are essential for ensuring the developed software meets the needs and expectations of its intended users.

Interface specification

An interface specification in software engineering is a detailed **description of how different software components, modules, or systems interact with each other**. It defines the **rules, protocols, and data formats for communication,** ensuring seamless integration and interoperability.

Key elements of an interface specification:

- **Interface Name:** A unique identifier for the interface.
- **Purpose/Description:** A high-level overview of the interface's function.
- **Inputs:** Description of data or parameters received by the interface, including data types, ranges, and constraints.
- **Outputs:** Description of data or results returned by the interface.
- **Pre-conditions:** Conditions that must be met before the interface can be invoke.
- **Post-conditions:** Conditions that are guaranteed to be true after the interface's execution.
- **Error Handling:** How the interface handles **errors or exceptions.**
- **Protocols/Communication Mechanisms:** The method used for interaction.

Example of an interface specification

Consider a basic user management system with an interface for creating new users. This interface could be defined as follows:

Interface Name: UserManagementService

Description: This interface defines operations related to user management within the system.

Methods:

- **userCreate** :**Description**: Creates a new user in the system.

  Parameters: firstName: String, representing the user's first name.

- **lastName:** String, representing the user's last name.

- **type:** String, representing the user's role or type (e.g., "admin", "regular").
- **email**: String, representing the user's unique email address.
- **password:** String, representing the user's password.

Returns: Integer, the unique identifier (ID) of the newly created user.

Exception Handling:

- UserAlreadyExists Exception:

  Thrown if a user with the provided first Name and last Name or email already exists.

- UserNotAuthorized Exception:

  Thrown if the attempting user does not have the required permissions to create a new user.

Example Implementation (Conceptual):

A component, such as a UserService class, would then implement this UserManagementService interface. It provides the concrete logic for creating users, handling exceptions, and ensuring data validity, based on the specifications defined in the interface.

Key elements of this example

- **Clear Method Signatures**: Each method clearly defines its name, parameters (with types), and return type.
- **Detailed Descriptions**: Each method's description explains its purpose and expected behavior.
- **Exception Handling**: Specifies the possible errors or exceptional conditions that might occur and how they should be handled.

By clearly defining the interface in this manner, different developers or teams can implement the UserManagementService independently. They know exactly how to interact with it to create new users and what to expect in terms of data and possible error scenarios.

Software Requirements Document (SRD)

A Software Requirements Document (SRD), also known as a Software Requirements Specification (SRS), is a detailed blueprint outlining the intended behavior, features, functionalities, and constraints of a software system. It's a foundational document created

early in the software development lifecycle to align all stakeholders on a common understanding of the project's goals, scope, and technical specifications.

Key sections of an SRD

While the structure might vary slightly, a typical SRD includes these essential sections:

1**. Introduction:**

- Purpose: States the document's objectives and the overall goals of the software project.
- Intended Audience: Defines who will use the software (e.g., end-users, internal teams).
- Intended Use: Explains the software's primary function and the problems it will solve.
- **Product Scope:** Delineates the boundaries and limitations of the project, including features and functions that will (and won't) be included.
- **Definitions and Acronyms:** Provides a glossary of terms used in the document to ensure shared understanding.

2. Overall Description:

- **User Needs**: Describes the needs and expectations of the intended users.
- Assumptions and Dependencies: Outlines any assumptions made during requirements gathering and identifies factors the project relies on.

3. System Features and Requirements:

- **Functional Requirements**: Details the specific features and functions the software *must* possess. These define what the software will *do*.
- **Example:** In an e-commerce application, a functional requirement might be: "Users should be able to add products to their shopping carts".
- These can be further detailed using formats like EARS ("When [event], the system shall [response]") or Behavior-Driven Development (BDD) scenarios.

Non-Functional Requirements: Specifies *how* the software should perform, focusing on quality attributes and constraints like performance, security, usability, and reliability.

- Example: "The e-commerce platform must load product pages in under two seconds".

External Interface Requirements: Describes how the software interacts with external components like user interfaces, hardware, other software systems, and communication protocols.

4. Other Requirements (Optional):

Database Requirements: Details the database structure and storage needs.

Legal and Regulatory Requirements: Outlines any legal or industry compliance mandates.

Internationalization and Localization: Specifies requirements for supporting multiple languages and cultural adaptations.

Importance of a clear SRD

- **Minimizes Misunderstandings:** Ensures all stakeholders (developers, clients, users) have a shared understanding of the project's objectives and functionalities, according to Requiment.
- **Reduces Project Risks**: Helps identify potential issues and conflicts early on, reducing the likelihood of costly rework and delays.
- **Facilitates Development and Testing**: Provides a clear roadmap for the development team and a basis for creating effective test cases.
- **Supports Future Maintenance and Enhancements:** Serves as a valuable reference for understanding the software's initial goals and requirements during future updates and maintenance efforts.

Feasibility Studies

A feasibility study in software engineering is a essential step taken in the initial planning and design phases of a proposed software project. It is essentially a systematic and comprehensive analysis designed to assess the viability and potential success of the proposed software solution. This evaluation helps stakeholders make informed decisions before allocating significant time, resources, and budget to the development process.

Types of Feasibility Study

The feasibility study mainly concentrates on below five mentioned areas. Among these Economic Feasibility Study is most important part of the feasibility analysis and Legal Feasibility Study is less considered feasibility analysis.

1. **Technical Feasibility**: In Technical Feasibility current resources both hardware software along with required technology are analyzed/assessed to develop project. This technical feasibility study gives report whether there exists correct required resources and technologies which will be used for project development. Along with this, feasibility study

also analyzes technical skills and capabilities of technical team, existing technology can be used or not, maintenance and up-gradation is easy or not for chosen technology etc.

2. **Operational Feasibility:** In Operational Feasibility degree of providing service to requirements is analyzed along with how much easy product will be to operate and maintenance after deployment. Along with this other operational scopes are determining usability of product, Determining suggested solution by software development team is acceptable or not etc.
3. **Economic Feasibility:** In Economic Feasibility study cost and benefit of the project is analyzed. Means under this feasibility study a detail analysis is carried out what will be cost of the project for development which includes all required cost for final development like hardware and software resource required, design and development cost and operational cost and so on. After that it is analyzed whether project will be beneficial in terms of finance for organization or not.
4. **Legal Feasibility:** In Legal Feasibility study project is analyzed in legality point of view. This includes analyzing barriers of legal implementation of project, data protection acts or social media laws, project certificate, license, copyright etc. Overall it can be said that Legal Feasibility Study is study to know if proposed project conform legal and ethical requirements.
5. **Schedule Feasibility:** In Schedule Feasibility Study mainly timelines/deadlines is analyzed for proposed project which includes how much time teams will take to complete final project which has a great impact on the organization as purpose of project may fail if it can't be completed on time.
6. **Cultural and Political Feasibility:** This section assesses how the software project will affect the political environment and organizational culture. This analysis takes into account the organization's culture and how the suggested changes could be received there, as well as any potential political obstacles or internal opposition to the project. It is essential that cultural and political factors be taken into account in order to execute projects successfully.
7. **Market Feasibility:** This refers to evaluating the market's willingness and ability to accept the suggested software system. Analyzing the target market, understanding consumer wants and assessing possible rivals are all part of this study. It assists in

identifying whether the project is in line with market expectations and whether there is a feasible market for the good or service being offered.

8. **Resource Feasibility:** This method evaluates if the resources needed to complete the software project successfully are adequate and readily available. Financial, technological and human resources are all taken into account in this study. It guarantees that sufficient hardware, software, trained labor and funding are available to complete the project successfully.

Requirements elicitation and analysis process

Requirements elicitation is a fundamental step in software engineering (SE) that involves the systematic gathering of information from stakeholders to understand and define the needs and expectations of a software system.

Software engineers interact with various stakeholders during requirement elicitation, including clients, end users, and domain experts. The goal is to collect, analyze, and document their inputs, which will serve as the foundation for designing and developing the software, as shown in the illustration below.

There are four stages in the requirements elicitation and analysis process, as shown in the diagram below.

Stages of requirements elicitation and analysis

1. **Requirements discovery:** Interact with stakeholders to uncover system needs and domain requirements, while also reviewing existing documentation. For example, interviewing customers and analyzing sales data to understand e-commerce platform needs.
2. **Classification and organization:** Group requirements by system architecture, identifying subsystems for clarity. For example, grouping car requirements into engine performance, safety features, and interior design categories.
3. **Prioritization and negotiation:** Resolve conflicts and prioritize requirements through stakeholder consensus, even if it requires compromise. For example, balancing

marketing's desire for new features with development's need for system optimization in software development.

4. **Requirements specification:** Document requirements formally or informally to provide a clear foundation for system development. For example, documenting smartphone specifications, including screen size, camera details, and battery life for development guidance.

There are multiple techniques followed in eliciting the requirements. Let's explore them one by one.

Requirements elicitation techniques

1. Scenarios

- **Collaborate on scenarios:** Work closely with stakeholders to develop scenarios, incorporating their insights and details.
- **Scenario variety:** Explore different scenario formats, including text, diagrams, and screenshots, to convey requirements effectively.
- **Structured approach:** Consider structured methods like event scenarios or use cases for a more organized approach.
- **Visual aids:** Use visual elements like diagrams to enhance scenario understanding.
- **Detail capture:** Ensure that scenarios capture all the necessary details for a comprehensive requirement understanding.

2. Interviewing

- **Create a clear questionnaire:** Start with a detailed questionnaire.
- **Involve key people:** Include different stakeholder groups.
- **Ask open questions:** Use open-ended queries for free-flowing insights.
- **Flex your methods:** Adjust to their preferences (in-person, groups, video, email).
- **Listen actively:** Dig deeper with follow-up questions to uncover hidden needs.

3. User stories

- **Use conversational text:** Create brief, customer-focused user stories for initial requirements and agile planning.
- **Embrace agile:** Align user stories with agile methods for flexibility.
- **Customer's voice:** Let customers express system needs in their own words.
- **Variety of prototypes:** Prototypes can be working (e.g., software code or simulations) or nonworking (e.g., storyboards and mock-ups).
- **Keep it short:** Aim for two to four sentences on a compact card.
- **Adapt to project:** Adjust the number of user stories based on project size and methodology.

4. Requirement workshops

- **Focused workshops:** Highly structured sessions with selected stakeholders.
- **Clear objectives:** Define, refine, and finalize business requirements.
- **Swift documentation:** Quickly complete and share documentation for review.
- **Instant confirmation:** Get immediate feedback on requirements.
- **Efficient for large groups:** Ideal for efficiently gathering requirements from large groups.

5. Document analysis

- **Review existing materials:** Start by carefully examining the available documents to grasp the business environment and ensure current solutions are on track.
- **Validate implementation:** Use this approach to confirm the effectiveness of existing solutions.
- **Comprehend business needs:** Dive into documents to gain a clear understanding of business requirements.
- **Documents for evaluation:** Focus on business plans, technical documents, problem reports, and any existing requirement documents.

6. Focus groups

- **Focused discussions:** Focus groups facilitate discussions that target specific subjects, differing from individual interviews.
- **Optimal group size:** Typically, a focus group comprises 6 to 12 participants, with the option to form multiple groups for broader participation.
- **Dynamic interaction:** Active discussions foster an environment conducive to idea exchange.

7. Prototyping

1. **Explore new features:** Prototyping helps uncover and explore new system features.
2. **Architectural insights:** Architects use prototypes, like scale drawings and 3-D models, to uncover and validate customer requirements.
3. **Systems engineering benefit:** Systems engineers also employ prototypes to achieve similar goals in requirement discovery.

8. Brainstorming

1. **Idea generation:** Use brainstorming to generate solutions for specific issues involving domain and subject matter experts.
2. **Diverse ideas:** Encourage multiple ideas for a knowledge repository and a range of choices.
3. **Table discussions:** Hold table discussions—typically held as a roundtable discussion, with all participants given equal time to share their ideas.
4. **Key questions:** Brainstorming answers questions like system expectations, risk factors, rules, issue resolutions, and future issue prevention.

9. Ethnography

1. **Observational approach:** Ethnography immerses an analyst in the work environment to grasp operational processes and derive support needs.
2. **In-depth observation:** Analysts keenly observe daily tasks, uncovering implicit system requirements that mirror real-world work practices, not just formal processes.

10. Surveys/questionnaires

1. **Survey/questionnaire approach:** Stakeholders are given a set of questions to quantify their thoughts, with data analyzed to identify key areas of interest.
2. **Effective question crafting:** Craft questions based on high-priority risks, keeping them direct and unambiguous. Notify participants once the survey is ready and remind them to participate.
3. **Broad data collection:** Easily gather data from a large audience.

Requirements validation

Requirements validation is a crucial process in software development that ensures the gathered requirements accurately reflect the real needs and expectations of the stakeholders and will lead to a successful system.

It answers the question, "Are we building the right product?"

Importance of requirements validation

Requirements validation is important for a successful software project because it:

- **Minimizes errors and rework:** Identifying errors or misinterpretations of requirements early avoids costly and time-consuming rework in later development phases.
- **Ensures stakeholder satisfaction**: By confirming that the documented requirements align with the actual needs of the end-users and other stakeholders, the final product is more likely to meet their expectations, leading to higher satisfaction.
- **Reduces project risks**: It helps identify and address potential issues like technical limitations, resource constraints, or conflicting requirements before they impact the project schedule or budget.
- **Improves communication**: It fosters better communication and collaboration between stakeholders and the development team, leading to a shared understanding of project goals and deliverables.
- **Prevents scope creep**: By establishing clear and well-defined requirements, it helps prevent uncontrolled changes and ensures the project stays focused on its objectives.

Key techniques for requirements validation

Several techniques can be used for requirements validation, often in combination for optimal results:

Requirements Reviews:

This involves a systematic examination of the requirements document by a group of experts, stakeholders, and developers to identify errors, inconsistencies, uncertainty, and omissions.

Example: When developing an online banking application, a review of the requirement stating, "Users can transfer funds between their accounts," reveals the need for specifying the types of accounts involved (checking, savings), transaction limits, and security protocols like two-factor authentication.

Prototyping:

This involves creating a working model or simulation of the system based on the requirements, which stakeholders can interact with to provide feedback.

Example: For a new e-commerce website, a clickable prototype demonstrating the user journey from browsing products to adding to the cart and completing the purchase can be shared with potential customers. This allows gathering feedback on user-friendliness, checkout flow, and any missing features.

Test Case Generation: Developing test cases based on the requirements to assess their testability and identify potential issues during development.

Example: For a requirement stating, "The system must process payments securely," test cases would be created to cover various payment scenarios, including successful transactions, invalid card details, insufficient funds, and potential security threats. Difficulty in creating test cases for a particular requirement may indicate ambiguity or incompleteness in that requirement.

Automated Consistency Analysis: Utilizing tools to automatically check for inconsistencies and errors in requirements specifications, especially for complex systems.

Example: A tool might detect a conflict where one requirement states, "Users can access the system 24/7," while another states, "System will be unavailable for maintenance every Sunday from 2 AM to 4 AM." This discrepancy would then be flagged for resolution.

Walkthroughs: A less formal review where stakeholders and team members "walk through" the requirements document, discussing potential issues and concerns.

Example: For a project management tool, a walkthrough of the requirements related to task management might involve a discussion about how tasks are assigned, updated, and tracked, and whether the proposed workflow aligns with the users' actual practices.

Traceability: Ensuring each requirement can be traced back to its source and linked to design, implementation, and test cases.

Example: For a mobile banking app, a requirements traceability matrix could link the requirement "Users can view their transaction history" to specific design documents, code modules, and test cases that verify the accuracy and completeness of the displayed transaction data.

Checklists: Using predefined checklists to methodically assess requirements against quality standards and identify errors.

Example: A checklist for requirements validation could include items such as "Is the requirement clear and unambiguous?", "Is the requirement complete?", "Is the requirement testable?", and "Does the requirement align with business goals?".

By employing these techniques in conjunction and involving relevant stakeholders throughout the process, software development teams can significantly improve the quality of their requirements, minimize risks, and increase the likelihood of delivering a successful and user-satisfying product.

Requirements Management

Requirements management is the structured process of defining, documenting, tracking, prioritizing, and controlling changes to project requirements throughout the entire software development lifecycle. It is crucial for ensuring that the final software product meets the needs and expectations of all stakeholders involved.

Key aspects of requirements management

- **Requirements Elicitation**: This is the process of gathering information from stakeholders (users, customers, business teams, technical experts) to understand their needs and expectations for the system. Techniques include interviews, surveys, workshops, focus groups, and prototypes.

- **Requirements Analysis and Definition**: After elicitation, the gathered information is analyzed, clarified, and transformed into precise, unambiguous requirements. This involves identifying inconsistencies, conflicts, and missing information, and then specifying them clearly and unambiguously in a format like a Software Requirements Specification (SRS) document or user stories.

- **Requirements Prioritization**: Not all requirements have equal importance. Prioritization helps teams focus on the most critical needs first, especially when resources or time are limited. Methods like the MoSCoW method (Must-have, Should-have, Could-have, Won't-have) can be used for prioritization.

- **Requirements Traceability**: This involves establishing and maintaining links between requirements and other project artifacts, including design elements, code modules, and test cases. A Requirements Traceability Matrix (RTM) is often used for this purpose. Traceability helps to ensure that all requirements are being addressed during development and that the final product can be verified and validated against them.

- **Requirements Change Management**: Requirements are rarely static and often evolve throughout the project lifecycle. Change management defines the process for submitting, reviewing, analyzing the impact of, and approving or rejecting change requests, ensuring that changes are implemented in a controlled and systematic manner.

- **Requirements Validation and Verification**: Validation focuses on ensuring that the requirements reflect the true needs of the stakeholders and will lead to a successful system (i.e., building the right product), IBM states requirements validation ensures that product and development goals are met. Verification confirms that the developed software meets the defined requirements correctly (i.e., building the product right). Techniques like reviews, inspections, and testing are used in both processes.

Example

Imagine a company wants to develop a new online grocery ordering and delivery application.

1. Requirements Elicitation

- Interviews: Business analysts interview customers to understand their shopping preferences, delivery expectations, and pain points with existing services. They also interview store managers to understand inventory management, order fulfillment processes, and delivery logistics.
- Surveys: Customers could be surveyed about their preferred payment methods, delivery windows, and loyalty program features.
- Workshops: A workshop with key stakeholders (customers, delivery personnel, store managers, marketing team) could be conducted to brainstorm features and prioritize functionalities.
- Observation: Observing how customers currently shop and how store employees manage orders and deliveries can reveal implicit requirements.

2. Requirements analysis and definition

- Based on the collected information, the team defines functional requirements such as: "Users can search for groceries by category and keywords" or "The system must allow users to select a preferred delivery time slot."
- Non-functional requirements are also defined: "The system must be available 99.9% of the time," "The application must load product pages within 2 seconds," or "The system must secure customer payment information with encryption".
- User stories are also written, such as: "As a busy parent, I want to quickly reorder my weekly groceries, so I can save time on shopping".
- A Software Requirements Specification (SRS) document is created to formally document all defined requirements.

3. Requirements prioritization

- The team prioritizes requirements based on factors like business value (e.g., core features like adding items to a cart are high priority), risk (e.g., payment gateway security is critical), and feasibility.

- For instance, "secure payment processing" would be high priority, while "social media sharing of recipes" might be lower priority for the initial release.

4. Requirements traceability

- An RTM is created to link each requirement to its design elements (e.g., database schema for product catalog), code modules (e.g., functions for user registration), and test cases (e.g., test cases to verify user login functionality).
- This ensures that all parts of the application are designed, built, and tested to meet the specified requirements.

5. Requirements change management

- Mid-project, a new government regulation emerges, requiring specific data privacy protocols for handling customer information.
- A change request is submitted, documented, and analyzed for its impact on existing requirements, scope, timeline, and budget. StarAgile details the steps involved in a change control process.
- A formal change control board reviews the request and decides whether to approve or reject it.
- If approved, the relevant requirements are updated, design and code are modified, and new test cases are created to ensure compliance with the new regulation.

6. Requirements validation and verification

- Validation: Prototypes of the user interface are built and shown to target users to ensure the design meets their expectations and needs. Feedback is collected to refine the user experience.
- Verification: After the application is developed, it undergoes thorough testing to verify that each feature and function operates as specified in the requirements document. This includes unit testing, integration testing, and system testing. Acceptance testing by stakeholders confirms that the system meets the initially defined needs.

By diligently managing requirements throughout the software development process, the company developing the online grocery application increases the chances of creating a product that satisfies customer needs, complies with regulations, and is delivered within the projected budget and timeline.

Context model

In software engineering, a context model is a high-level representation of a system's boundaries and its interactions with the external environment. It provides a big-picture view that shows the system, external entities that interact with it, and the data or information that flows between them.

Context models are particularly valuable during the initial phases of a project, such as system analysis and requirement gathering.

Key components of a context model

- **System:** The software being developed, this is typically placed at the center of the diagram.
- **External entities**: These are elements outside the system boundary that interact with it. **Examples** include users, other software systems, databases, and external hardware devices.
- **Data flows**: These are represented by arrows showing the movement of data or information between the system and the external entities. The arrows can be unidirectional or bidirectional.

Purpose and uses

- **Defines scope**: By illustrating the system's boundaries and its interactions, a context model helps define and focus the project's scope, ensuring that the analysis and development concentrate on the relevant components.
- **Clarifies requirements**: It helps in identifying both functional and non-functional requirements by visualizing how the system interacts with its environment and what data it exchanges.
- **Facilitates communication**: Context models are simple and easy for both technical and non-technical stakeholders to understand, making them an excellent communication tool.
- **Manages dependencies and risks**: By mapping out the system's relationships with external entities, the model helps identify potential dependencies and risks. This allows for proactive planning and mitigation.

Example: A hotel reservation system

Imagine you're designing a new online hotel reservation system. A context model would illustrate the following:

- **Central system**: The Hotel Reservation System.
- **External entities:**
  - Customer: Interacts with the system to search for rooms, make reservations, and pay.
  - Hotel staff: Uses the system to manage bookings, check guests in and out, and view reports.
  - Payment gateway: A third-party system that processes customer payments.
  - Hotel database: An external database that stores information on rooms, bookings, and customer details.
- **Data flows:**
  - The "Customer" sends payment and reservation information to the system.
  - The system sends booking confirmations and payment requests to the "Customer" and "Payment Gateway," respectively.
  - The system sends and receives booking information from the "Hotel staff" and "Hotel database."

This type of diagram provides a clear overview of the system's purpose and its place in the larger operational environment.

Context models can be visualized using simple graphical notation, often referred to as context diagrams.

Behavioral models

A behavioral model in software engineering describes how a system responds to events, focusing on the sequence of actions and interactions between its components.

Common behavioral models include State Machine Diagrams, which show system states and transitions, and Activity Diagrams, which illustrate ordered sets of actions.

An example is an online banking system's behavioral model, which uses a Sequence Diagram to show how a user's login attempt triggers events, leading to a verification process that determines access to their account.

- It focuses on the dynamic aspect of a system: its behavior over time and in response to stimuli.
- It helps understand how the system's components interact to achieve functions.
- It's used in requirements elicitation, design, and testing to ensure the system behaves as expected.

Types of Behavioral Models

- **Use Case Diagrams**: Show user goals and how they interact with the system.
- **Sequence Diagrams**: describe interactions between objects over time, showing the order of message exchanges.
- **Activity Diagrams**: Model the flow of activities and decisions within a process or function.
- **State Machine (or State) Diagrams**: Illustrate the different states a system can be in and the events that cause transitions between these states.

Example: Online Banking Login

Imagine a user trying to log in to their online banking account. A behavioral model, specifically a Sequence Diagram, could illustrate this process:

1. **User Action**: The user enters their username and password in the web browser.
2. **System Event**: The browser sends a "Login Request" event to the banking application.
3. **Component Interaction**:
    - The application receives the request and sends a "Verify Credentials" message to the authentication service.
    - The authentication service checks the credentials against its database.
4. **System Response**:
    - If successful, the authentication service sends a "Credentials Valid" message back to the application.
    - The application then sends an "Access Granted" event, leading the user to their account dashboard.
    - If the credentials are invalid, an "Access Denied" message is sent instead.

This model helps visualize the interactions and sequences of events, ensuring that the system correctly handles both successful and unsuccessful login attempts.

Data models

In software engineering, a data model is a blueprint that describes the structure of data, including its types, attributes, and the relationships between them, within a system or application. It is a key part of the design phase, providing a common understanding for business and technical stakeholders about how data will be organized, stored, and used.

Data models serve as a visual roadmap, facilitating communication and helping to identify and prevent errors before code is written.

Key Aspects of Data Models

- Purpose:

To define business and application data requirements, create a visual representation of data elements, and provide a structure for database design and implementation.

- Components:

Data models typically include:

- Entities: The main objects or concepts in the system (e.g., customers, orders, products).
- Attributes: The properties or characteristics of each entity (e.g., a customer's name, an order's date).
- Relationships: How entities are connected to each other (e.g., a customer places an order).

Levels of Abstraction:

Data models are often developed in stages:

- **Conceptual Data Model**: A high-level overview of the data that focuses on business requirements and user needs.
- **Logical Data Model**: A more detailed description of data structures, including entities, attributes, and relationships, but without specifying the physical database implementation.
- **Physical Data Model**: The actual database design, specifying table types, data types, constraints, and other technical details for implementation.

Benefits:

- **Improved Communication**: Provides a common language for business analysts, developers, and other stakeholders.
- **Better Design**: Acts as a blueprint for creating efficient and effective databases.

- **Reduced Errors**: Helps uncover flaws and inconsistencies in data requirements and development plans early on.
- **Enhanced Data Management**: Supports better data organization, accessibility, and quality throughout the system.

How Data Models are Used

- **Requirements Analysis**: They start by capturing and visualizing the data requirements of a software application or business process.
- **Database Design**: The data model serves as the foundation for designing the database structure.
- **Documentation**: They create a clear and concise record of the data within a system.

In software engineering, system models are abstract representations used to understand, design, and communicate different views of a system. Context, behavioral, and data models each provide a distinct perspective.

Structured methods

Structured methods in software engineering are systematic, top-down approaches that break down complex systems into manageable parts using specific notations and graphical tools, such as Structured Analysis (SA) and Structured Design (SD), to improve clarity, maintainability, and quality.

These methods focus on logical thinking, functional decomposition, and process-oriented techniques like Data Flow Diagrams (DFDs) to ensure requirements are accurately captured and software is designed to meet those needs effectively.

Key Principles and Concepts

- **Top-Down Decomposition:**

Complex systems are broken into smaller, simpler, and more manageable components, starting from a high-level overview and progressively detailing lower-level functions.

- **Logical vs. Physical Models:**

Structured methods emphasize the logical structure of a system, separating it from physical implementation details (like specific hardware or software) to allow for greater flexibility and ease of maintenance.

- **Modularity and Hierarchy:**

Programs are designed as a hierarchy of functional, interconnected modules, making them easier to understand, develop, and maintain.

- **Structured Programming:**

This involves a disciplined approach to writing code, focusing on clarity and simplicity through the use of control constructs rather than unrestricted jumps (like goto).

Core Techniques and Tools

Structured Analysis (SA):

- **Purpose:** To understand and document system requirements by dividing a complex system into smaller parts.
- **Tools:** Uses graphical tools like Data Flow Diagrams (DFDs) to show how data moves through the system.
- **Outcome:** Produces user-friendly system specifications that clearly define user needs.

Structured Design (SD):

- **Purpose:** To influence the structure of the final program, creating a hierarchical design from a functional decomposition.
- **Tools:** Uses techniques like top-down design and structure charts to visualize the system's components and their relationships.
- **Outcome:** A well-structured and hierarchical design that is easier to implement and maintain.

Data Flow Diagrams (DFDs):

- **Purpose:** To represent the flow of data within a system, showing how data enters, is processed, and exits.
- **Visuals:** Consist of "bubbles" (processes) and arrows (data flow) to illustrate system functions.

Data Dictionary:

A centralized repository for data definitions and structures used in the system, ensuring consistency in data representation.

Benefits

- **Improved Software Quality:** Leads to more understandable, reliable, and maintainable software.
- **Enhanced Clarity:** Graphical representations and systematic documentation make complex systems easier to grasp.
- **Better Resource Utilization:** Helps design systems that make the best use of available resources.
- **Facilitated Modifications:** Creates flexible systems that can be more easily adapted to future changes.