Guidelines for Preparing Detailed System Design Documentation

A Detailed System Design document is a critical part of the software engineering process, providing a comprehensive blueprint for the system's architecture, components, and interactions. It serves as a guide for developers during implementation and ensures that the design follows the functional and non-functional requirements defined in the Software Requirements Specification (SRS). A well-prepared system design document includes detailed UML diagrams that visually represent the system's structure and behavior.

Here's a breakdown of what should be included in the Detailed System Design documentation, along with guidelines for creating each section.

1. Introduction

This section gives an overview of the project's purpose and sets the foundation for understanding the system design.

- **Objective**:
  Explain the goal of the system in one or two sentences, aligned with the requirements from the SRS (Software Requirements Specification). For example, "The objective of this system is to provide a secure and efficient platform for managing user accounts and processing customer orders."
- **Scope**:
  Clearly state what the system covers in terms of functionality. This may include modules like user management, product catalog, order processing, etc. The scope should also define boundaries: what the system will and will not include.

Example:
"The scope of this system covers functionalities such as user authentication, role-based access control, product browsing, and order management. It does not cover external payment gateway integration or third-party logistics tracking."

- **Assumptions**:
  Mention any assumptions about technology, resources, or user roles that will affect the design. For instance, "It is assumed that users will access the system through a desktop environment, and the system will use a SQL-based relational database for data storage."

2. System Architecture Design

This section provides a high-level overview of the entire system, describing how different components interact and the overall structure of the system.

- **Architectural Overview**:
  Provide a general description of the system's architecture. This should include an outline of the system's components, such as the client application, business logic layer, database, and any external systems or APIs the software will interact with.

Example:
"The system follows a three-tier architecture with the presentation layer (user interface), business logic layer (application services), and data layer (database)."

- **UML Diagram – Component Diagram**:
  Use a **Component Diagram** to visually represent the system's components and their relationships. Each component should represent a distinct part of the system, such as the database, user interface, or external systems. Make sure to show the connections between these components and their interaction.

Key Elements:

  - **Components**: Represent each module/subsystem.
  - **Connectors**: Show dependencies between components (e.g., how the UI interacts with the business logic layer).
  - **External Interfaces**: Highlight any APIs or external systems that the software interacts with.

Example:
If you're building an e-commerce system, the component diagram might show interactions between the web front-end, product catalog service, order processing system, and the database.

3. Detailed Module Descriptions

Each module should be described in detail, including its responsibilities, interactions, and internal structure.

- **Module Overview**:
  Provide a detailed description of each module’s purpose within the system. Explain what each module does and how it contributes to the system as a whole.

Example:
For an "Order Management" module:

  - **Responsibilities**: "This module is responsible for handling customer orders, including creation, validation, processing, and status tracking."
  - **Dependencies**: "This module interacts with the Payment module for payment processing and with the Inventory module to check product availability."

- **Input/Output**:
  Explain what data the module accepts as input and what it produces as output. This section helps developers understand what data needs to be passed between modules.

Example:

  - **Input**: "The Order Management module receives customer information, product details, and payment confirmation."
  - **Output**: "The module generates an order confirmation and updates the product inventory."

- **UML Diagram – Class Diagrams**:
  For each module, create a **Class Diagram** that outlines the internal structure of the module in terms of objects, classes, and their relationships.

Key Elements:

  - **Classes**: Include important classes with their attributes and methods.
  - **Relationships**: Show inheritance, composition, aggregation, and associations between classes.
  - **Visibility**: Clearly define whether attributes and methods are public, private, or protected.

Example:
In the "Order Management" module, the class diagram might include classes such as Order, OrderItem, Customer, and OrderProcessor, with relationships like Customer places Order or Order contains OrderItem.

4. Database Design

This section describes the database structure, including entity relationships and normalization.

- **Entity-Relationship (ER) Diagram**:
  Create an **ER Diagram** to illustrate how the system's data will be structured in the database. This diagram should include all relevant entities (database tables) and their relationships (e.g., one-to-one, one-to-many, many-to-many).

Key Elements:

  - **Entities**: Represent database tables with their primary attributes.
  - **Relationships**: Define how entities are related (e.g., each Customer can have multiple Orders, and each Order contains multiple OrderItems).
  - **Keys**: Specify primary and foreign keys that ensure data integrity.

Example:
For an online store, the ER diagram may include entities such as Customer, Order, Product, and OrderItem, with relationships showing how customers place orders and orders contain products.

- **Normalization**:
  Ensure that your database design adheres to the principles of **Normalization** (typically up to 3rd Normal Form) to minimize redundancy and improve data integrity.

Example:
In a normalized design, the customer information would be stored separately in the Customer table, rather than duplicating this data in each order.

5. System Flow Design

Describe how the system handles workflows and specific use cases.

- **Sequence Diagrams**:
  Use **Sequence Diagrams** to describe how different objects or components interact over time to fulfill a specific use case. Sequence diagrams focus on the exchange of messages between objects.

Key Elements:

  - **Actors**: Define the user or external system interacting with the software.
  - **Messages**: Represent the communication between objects (e.g., method calls, return values).
  - **Lifelines**: Show the lifespan of objects as they interact.

Example:
For a user login process, the sequence diagram might show the user interacting with the UI, which sends credentials to the authentication system, which in turn interacts with the database to validate the user.

- **Activity Diagrams**:
  Create **Activity Diagrams** to represent workflows or processes within the system, focusing on the flow of control.

Key Elements:

  - **Activities**: Define the steps that occur during a specific process.
  - **Decisions**: Represent points in the process where alternative paths can be taken.
  - **Synchronization Bars**: Indicate parallel processes or workflows.

Example:
An activity diagram for the order fulfillment process might show activities such as "Create Order," "Process Payment," and "Ship Order," with decisions like “Payment Approved?”

6. User Interface Design (Optional in Detailed Design)

In some cases, UI design may be a part of system design documentation. Include high-level design elements to show how the user will interact with the system.

- **Wireframes or Mockups**:
  Include basic wireframes or mockups to show the layout of user interface screens. This helps developers understand how users will navigate through the application.

Example:
A wireframe for the order management screen might show sections for order details, product listings, and customer information.

- **Navigation Flow**:
  Provide a flowchart that shows how different screens or pages are connected, representing the user's navigation paths through the system.

7. Interaction Between Components

This section describes how system components collaborate to achieve the functionality.

- **Collaboration Diagrams (Communication Diagrams)**:
  Use **Collaboration Diagrams** to show how different objects or components interact and work together to achieve a specific task. This diagram emphasizes the structure of the system.

Example:
For a "User Registration" process, the diagram may show how the User, ValidationService, Database, and NotificationService collaborate to register a user and send a confirmation email.

8. Non-Functional Requirements

Describe how the system design meets non-functional requirements, such as performance, security, scalability, and reliability.

- **Performance Considerations**:
  Describe how the system is designed to handle performance concerns such as response time, throughput, or latency. This may include database indexing, caching mechanisms, or load balancing strategies.
- **Security Measures**:
  List security features designed into the system, such as encryption, authentication methods, secure communication protocols (e.g., HTTPS), and access control measures.
- **Scalability**:
  Explain how the system is designed to scale as the user base or data grows. This may involve horizontal scaling (adding more servers) or vertical scaling (upgrading server resources).
- **Reliability**:
  Describe how the system is designed for reliability, including failover mechanisms, backups, and recovery procedures.

9. System Interfaces

Detail the internal and external interfaces that the system interacts with.

- **External Interfaces**:
  Describe how the system will interact with external systems, such as third-party services or APIs. Specify the protocols, data formats, and security considerations.

Example:
The system may connect to an external payment gateway through a REST API, sending payment details and receiving payment confirmation in JSON format.

- **Internal Interfaces**:
  Define how different modules within the system will communicate with each other. This could include service calls, method invocations, or data transfers between layers.

10. Design Patterns (If Applicable)

Mention any design patterns used in the system and justify their usage.

- **Patterns Used**:
  List any design patterns applied, such as **Model-View-Controller (MVC)** for structuring the UI, **Singleton** for managing resources, or **Observer** for event-driven interactions.
- **Justification**:
  Explain why these patterns were chosen and how they benefit the system. For instance, MVC is beneficial for separating concerns between the UI and business logic, making the system more modular and maintainable.

11. Detailed Algorithm Descriptions

Provide detailed descriptions of any key algorithms used in the system, particularly those implementing core business logic.

- **Pseudocode**:
  Where necessary, include pseudocode to describe the steps of complex algorithms, such as sorting, searching, or validation processes.

Example:
For an order validation algorithm:

```
If product is in stock:
   Check customer credit
   If credit is sufficient:
      Approve order
   Else:
      Reject order
Else:
   Notify customer of stock issues
```

12. Hardware and Software Requirements

List the hardware and software environments necessary for the system to operate.

- **Hardware Requirements**:
  Specify the hardware resources needed, such as minimum CPU, memory, storage, and any specific hardware (e.g., servers, network equipment).

- **Software Requirements**:
  List the required software environments, libraries, frameworks, and development tools. This may include operating systems, databases, application servers, and external dependencies.

Example:
"The system will run on Windows 10, with MySQL 8.0 as the database, and Java 11 for application development."

13. Appendices

Include any additional material that supports the design documentation, such as glossaries, references, or external links to further information.

- **Glossary**:
  Provide definitions for any terms or acronyms used in the document to ensure clarity for all readers.
- **References**:
  List any external references, such as API documentation, technical papers, or standards, that were used to create the design.

Detailed UML Diagram Guidelines

1. **Component Diagram**
    - Shows the high-level structure of the system.
    - Include components (subsystems, databases, external interfaces).
    - Define relationships and dependencies between components.
2. **Class Diagram**
    - Use to detail each module’s structure.
    - Include all classes with attributes and methods.
    - Clearly show relationships between classes (inheritance, aggregation, composition, and association).
3. **Sequence Diagram**
    - For showing interaction flows between objects.
    - Include actors, objects, and messages exchanged.
    - Use to describe dynamic interactions within a specific use case.
4. **ER Diagram**
    - For the database structure.
    - Show entities, attributes, and relationships.
    - Define primary and foreign keys.
5. **Activity Diagram**
    - Represent workflows or business processes.
    - Include decisions, loops, and parallel actions.

Summary Template for Detailed System Design

1. Introduction

- Objective
- Scope
- Assumptions

2. System Architecture Design

- Architectural Overview
- Component Diagram

3. Detailed Module Descriptions

- Module Overview
- Class Diagrams

4. Database Design

- ER Diagram

5. System Flow Design

- Sequence Diagrams
- Activity Diagrams

6. User Interface Design (Optional)

- Wireframes/Mockups
- Navigation Flow

7. Interaction Between Components

- Collaboration Diagrams

8. Non-Functional Requirements

- Performance, Security, Scalability, Reliability

9. System Interfaces

- External and Internal Interfaces

10. Design Patterns

- Patterns Used
- Justification

11. Detailed Algorithm Descriptions

12. Hardware and Software Requirements

13. Appendices

- Glossary
- References