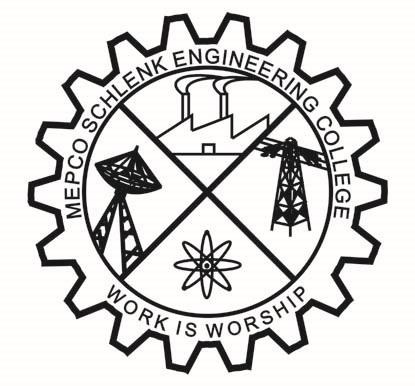
**MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI (AUTONOMOUS)**

# DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

****

**NAME : ………………………………………………………………………………………….. Class & Dept :…………………………………………………………………………………. Roll No. : ……………………………………………………………………………………….. Lab : ……………………………………………………………………………………………….**

LABORATORY RECORD

***Register No*:** .………………………

**CERTIFICATE**

*This is to Certify that it is the Bonafide work done by…………………………………………………… in the – Software Engineering Laboratory at Mepco Schlenk Engineering College, Sivakasi during the year 2024– 2025*

…………………………… ……………………………………

***Faculty in Charge Head of the Department***

*Submitted for the Practical Examination held at Mepco Schlenk Engineering College, Sivakasi on ……………………….*

…………………………. ………………………………

***Internal Examiner External Examiner***

**CONTENTS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Exp No.** | **Name of the Exercise** | **Page No.** | **Date** | | **Marks** | **Faculty Sign** |
| **Exercise** | **Submission** |
| 1 | Software Requirements Specification |  |  |  |  |  |
| 2 | Project Development Model Selection |  |  |  |  |  |
| 3 | High Level Design |  |  |  |  |  |
| 4 | AI Ethics Evaluation |  |  |  |  |  |
| 5 | Use Case Diagram |  |  |  |  |  |
| 6 | Activity Diagram |  |  |  |  |  |
| 7 | Class Diagram |  |  |  |  |  |
| 8 | Interaction Diagram |  |  |  |  |  |
| 9 | State Diagram |  |  |  |  |  |
| 10 | Component and Deployment Diagram |  |  |  |  |  |
| 11 | Low Level Design |  |  |  |  |  |
| 12 | Database Design |  |  |  |  |  |
| 13 | System Testing |  |  |  |  |  |

**MODULE 1**

**Software Requirements Specifications**

**(SRS)**

(Based on IEEE Std 830-1998)

**Nature of the SRS**

The SRS is a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment. The SRS may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both.

The SRS should specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed.

1. The **basic issues** that the SRS writer(s) shall address are the following:
2. Functionality: What is the software supposed to do?
3. External interfaces: How does the software interact with people, the system's hardware, other
4. hardware, and other software?
5. Performance : What is the speed, availability, response time, recovery time of various software
6. functions, etc.?
7. Attributes: What are the portability, correctness, maintainability, security, etc. considerations?
8. Design constraints imposed on an implementation: Are there any required standards in effect,implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

**Characteristics of a good SRS**

An SRS should be

1. Correct;
2. Unambiguous;
3. Complete;
4. Consistent;
5. Ranked for importance and/or stability;
6. Verifiable;
7. Modifiable;
8. Traceable.

**Front Page of SRS**

* Title of Project
* date of starting the document
* revision number
* responsible editor
* date of last update

**Prototype SRS outline**

**Table of Contents**

1. Introduction
   1. Purpose
   2. Scope
   3. Definitions, acronyms, and abbreviations
   4. References
   5. Overview
2. Overall description
   1. Product perspective
   2. Product functions
   3. User characteristics
   4. Constraints
   5. Assumptions and dependencies

3. Specific requirements (According to Template given at the end) Appendix

Index

**Purpose (1.1 of the SRS)**

* 1. What is the goal of the project?
  2. What is the purpose of the specified product?
  3. Who should read this document?

**Scope (1.2 of the SRS)**

1. Identify the software product(s) to be produced by name (e.g., Host DBMS, Report Generator,
2. etc.);
3. Explain what the software product(s) will do;
4. What parts of the domain problem will not be addressed by the specified product?
5. Are there limitations and disclaimers the reader should know of?
6. Describe the application of the software being specified, including relevant benefits, objectives,
7. and goals

**Definitions, acronyms, and abbreviations (1.3 of the SRS)**

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS.

**References (1.4 of the SRS)**

This subsection should provide

1. A complete list of all documents referenced elsewhere in the SRS;
2. Which documents do you have to read in order to understand the specification?
3. Which documents are referred in the text and can be consulted for clarification and detail?
4. Identify each document by title, report number (if applicable), date, and publishing organization.

**Overview (1.5 of the SRS)**

This subsection should

1. Describe what the rest of the SRS contains;
2. Explain how the SRS is organized.

**Overall description (Section 2 of the SRS)**

This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements.

This section usually consists of following six subsections:

* 1. **Product perspective:**
     1. If the product is independent and totally self-contained, it should be so stated here.
     2. If the product is a component of a larger system, then describe the relation to the other components. (with a block diagram)
     3. Describe interfaces to other software, hardware and users.
  2. **Product functions :**

Describe the main functions the specified product must perform.Textual or graphical methods can be used to show the different functions and their relationships.

* 1. **User characteristics:**

1. Who will use the system?
2. What training is needed?
   1. **Constraints:**

Describe constraints that apply and why they exist. These include

1. Regulatory policies;
2. Hardware limitations (e.g., signal timing requirements);
3. Interfaces to other applications;
4. Parallel operation;
5. Audit functions;
6. Control functions;
7. Higher-order language requirements;
8. Signal handshake protocols (e.g., XON-XOFF, ACK-NACK);
9. Reliability requirements;
10. Criticality of the application;
11. Safety and security considerations.
    1. **Assumptions and dependencies:**

List each of the factors that affect the requirements stated in the SRS.

These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS.

Eg :

* a specific operating system will be available on the hardware
* certain version of the browser will be available.
  1. **Apportioning of requirements:**

This subsection of the SRS should identify requirements that may be delayed until future versions of the system.

Here you can specify a lower ambition level for the system that you will build if you run out of time.

**Specific requirements (Section 3 of the SRS) :**

This section of the SRS should contain all of the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements.

These requirements should include at a minimum a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input or in support of an output.Either functional or an object oriented method can be used. It contains the following subsections :

* 1. **External interfaces:**

This section is applicable if more detail than the description in 2.1 is needed. It describes required user interface. This is not the design of the user interface but you can find good use of a prototype screen-shot.

* 1. **Functional Requirements :**

Functional requirements should define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as ‘shall’ statements starting with 'The system shall’.

These include

* + 1. Validity checks on the inputs
    2. Exact sequence of operations
    3. Responses to abnormal situations, including
       1. Overflow
       2. Communication facilities
       3. Error handling and recovery
    4. Effect of parameters
    5. Relationship of outputs to inputs, including
       1. Input/output sequences
       2. Formulas for input to output conversion

It may be appropriate to partition the functional requirements into subfunctions or subprocesses. This might be a good place-holder for the use-case descriptions.

* 1. **Performance requirements:**

This subsection should specify both the static and the dynamic numerical requirements placed on the software or on human interaction with the software as a whole. Static numerical requirements may include the following:

* + 1. The number of terminals to be supported;
    2. The number of simultaneous users to be supported;
    3. Amount and type of information to be handled

Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

* 1. **Logical database requirements:**

This should specify the logical requirements for any information that is to be placed into a database. This may include the following:

* + 1. Types of information used by various functions;
    2. Frequency of use;
    3. Accessing capabilities;
    4. Data entities and their relationships;
    5. Integrity constraints;
    6. Data retention requirements.
  1. **Design constraints:**

In principle, the SRS should not deal with design, but there might be requirements that are known that will constrain the designers' work, for instance, use of a certain standard.

For example, this could specify the requirement for software to trace processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standards. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

* 1. **Software system attributes** :

It is important that required attributes be specified so that their achievement can be objectively verified. System attributes or quality factors which have requirements, for example are:

* Reliability
* Availability
* Security - Utilize certain cryptographical techniques, Keep specific log or history data sets,

Assign certain functions to different modules, Restrict communications between some areas of the program, Check data integrity for critical variables.

* Maintainability
* Portability - Percentage of components with host-dependent code, Percentage of code that is host dependent, Use of a proven portable language, Use of a particular compiler language subset. Use of a particular operating system.
  1. **Organizing the specific requirements:**

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organizing these in a manner optimal for understanding.

Sample requirements are :

* **System mode** - For Eg. a control system may have different sets of functions depending on its mode: training, normal, or emergency
* **User class** - For example, an elevator control system presents different capabilities to passengers, maintenance workers, and fire fighters.
* **Objects** - For example, in a patient monitoring system, objects include patients, sensors, nurses, rooms, physicians, medicines, etc.
* **Feature** - For example, in a telephone system, features include local call, call forwarding, and conference call. Each feature is generally described in a sequence of stimulus-response pairs.
* **Stimulus** - For example, the functions of an automatic aircraft landing system may be organized

into sections for loss of power, wind shear, sudden change in roll, vertical velocity excessive, etc.

* **Response** - Some systems can be best organized by describing all the functions in support of the generation of a response. For example, the functions of a personnel system may be organized into sections corresponding to all functions associated with generating paychecks, all functions associated with generating a current list of employees, etc.
* **Functional hierarchy** - When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized by either common inputs, common outputs, or common internal data access.
* **Data flow diagrams and data dictionaries** can be used to show the relationships between and among the functions and data.
  1. **Additional comments:**

Any additional requirements may be put in a separate section at the end of the SRS.

**4 Supporting information:**

The supporting information makes the SRS easier to use. It includes the following:

1. Table of contents;
2. Index;
3. Appendixes.

**Appendixes :**

The appendixes are not always considered part of the actual SRS and are not always necessary. They may include

* 1. Sample input/output formats, descriptions of cost analysis studies, or results of user surveys;
  2. Supporting or background information that can help the readers of the SRS;
  3. A description of the problems to be solved by the software;
  4. Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When appendixes are included, the SRS should explicitly state whether or not the appendixes are to be considered part of the requirements.

**Template of SRS Section 3 organized by functions :**

* + 1. Specific requirements
       1. Functional requirements
          1. External interfaces
    2. User interfaces
       1. Hardware interface
       2. Software interfaces
       3. Communications interfaces
    3. Functional requirement
       1. Functional requirement 1
    4. .n Functional requirement n
    5. Performance
  1. Design constraints
  2. Software system attributes
  3. Other requirements

**PROJECT DEVELOPMENT MODEL SELECTION**

The Software Development Life Cycle (SDLC) is a structured process used to develop high- quality software efficiently and systematically. It defines a sequence of phases that guide software engineers through planning, developing, testing, deploying, and maintaining a software product.

|  |  |
| --- | --- |
| **Phase** | **Description** |
| **1. Requirement Analysis** | Identify user needs, gather data, and define functional/non-functional requirements. |
| **2. System Design** | Translate SRS into architecture and detailed design — defining data flow, interfaces, and components. |
| **3. Implementation** | Developers write the code according to design specifications. |
| **4. Testing** | Verify that the software works as expected — covers unit, integration, system, and acceptance tests. |
| **5. Deployment** | Software is installed or delivered for use in a live environment. |
| **6. Maintenance** | Post-deployment updates, bug fixes, enhancements, and performance optimization. |

Different SDLC models have evolved to suit various project needs, sizes, and complexities. The following are the major SDLC models adopted in software engineering practice

### Classical Waterfall Model

The Classical Waterfall Model is the earliest and most traditional approach to software development. It follows a sequential and linear flow, where each phase must be completed before the next begins. The typical stages include requirement analysis, system design, implementation, testing, deployment, and maintenance. This model emphasizes comprehensive documentation and strict phase discipline. It is simple to understand and manage but does not easily accommodate changes once development begins. Hence, it is best suited for small projects with well-defined, stable requirements.

### Iterative Waterfall Model

The Iterative Waterfall Model is a refinement of the classical approach. It maintains the sequential nature of the original model but allows feedback loops between phases. This means that errors or modifications discovered during later stages can lead to revisiting earlier phases for correction. The iterative refinement ensures higher accuracy and better risk control compared to the rigid waterfall process. This model is suitable for medium-sized projects where limited flexibility is required to accommodate evolving requirements.

### Prototyping Model

The Prototyping Model focuses on building an early working prototype of the software to understand and refine user requirements. A prototype is a preliminary version that demonstrates basic functionality and interface design. Users interact with the prototype and provide feedback, which helps developers improve the system before full-scale implementation. This model is ideal when requirements are not clearly defined in the initial stages. It enhances user involvement, reduces misunderstandings, and results in software that closely matches user expectations.

### Evolutionary Model

The Evolutionary Model emphasizes the gradual development of software through continuous refinement. Instead of delivering the entire product at once, the system evolves over several iterations. Each iteration adds new features or improvements based on user feedback and technological changes. This approach ensures that a working system is available at every stage and that customer feedback drives progress. The model is particularly effective for long-term and complex projects where requirements evolve over time.

### Spiral Model

The Spiral Model, proposed by Barry Boehm, combines iterative development with systematic risk management. The development process is represented as a spiral, where each loop corresponds to one phase such as planning, risk analysis, engineering, and evaluation. At each iteration, risks are identified and mitigated before proceeding further. The spiral model provides flexibility and focuses heavily on early detection and resolution of potential problems. It is suitable for large, high-risk projects where cost, time, and performance are critical.

### Win–Win Spiral Model

The Win–Win Spiral Model is an enhanced version of the spiral model that emphasizes stakeholder collaboration and negotiation. In each iteration, all stakeholders — customers, users, and developers — negotiate objectives and constraints to achieve a “win–win” outcome for everyone involved. This ensures that conflicting requirements are resolved early and all parties are satisfied before moving to the next phase. The model improves requirement clarity, reduces conflicts, and is ideal for complex projects involving multiple stakeholder groups.

### Incremental Model

The Incremental Model develops software in small, manageable increments, each delivering a working module of the final system. Each increment goes through all SDLC phases — analysis, design, coding, and testing — and adds new functionality to the existing system. This approach allows partial delivery of usable software early in the process while maintaining continuous improvement. It reduces risks, enhances flexibility, and is particularly suitable for projects requiring staged delivery or where requirements are prioritized.

### Rapid Application Development (RAD) Model

The RAD Model emphasizes quick development and delivery through iterative prototyping and intensive user involvement. It makes use of CASE (Computer-Aided Software Engineering) tools, reusable components, and automated code generation techniques. The focus is on producing functional components rapidly, which are later integrated into a complete system. RAD shortens development time and enables faster feedback from users. It is best suited for time-constrained projects where requirements are well-understood and users are available for frequent interaction.

### Agile Model

The Agile Model is a modern, iterative approach that values flexibility, collaboration, and customer feedback. Development is divided into short cycles called sprints, each lasting two to four weeks. Every sprint results in a working product increment that can be reviewed and refined based on feedback. Agile promotes adaptive planning, continuous improvement, and face-to-face communication. Frameworks like Scrum, Kanban, and Extreme Programming (XP) are based on Agile principles. This model is ideal for dynamic environments such as AI, mobile, and web applications where requirements change rapidly.

**TITLE:** SMART TRAFFIC AUTOMATION

**SELECTED MODEL:** Iterative (Agile-like) Model

**JUSTIFICATION:**  
The Iterative SDLC approach is chosen for the Smart Traffic Automation project to deliver a working, demonstrable system quickly and then refine it through short, feedback-driven iterations. Each iteration (sprint) targets a small, valuable slice of functionality — for example: sprint 1 implements authentication, seeded graph loading and a basic booking flow; sprint 2 adds occupancy-aware Dijkstra routing and persisted bookings; sprint 3 integrates the Python ML predictor (with a heuristic fallback) and admin controls; subsequent sprints add the interactive graph selector, booking history, CSV export, and real-time updates. This phased approach enables early validation of core assumptions (that bookings, occupancy weighting, and speed recommendations can reduce simulated congestion) and lets stakeholders — instructors, test users, or demonstrators — provide actionable feedback after each demo.

Iterations are time-boxed and feature-scoped to keep momentum and avoid scope creep. Cross-functional work is organized so frontend (React), backend (Node/Express), data (MongoDB) and ML (Python) tasks advance together; APIs and data contracts are stabilized early so components can be developed and tested in parallel. The architecture is intentionally modular (separate controllers, services, and ML invoker) so individual pieces can be replaced or scaled without breaking existing functionality.

Each increment includes a full validation cycle: unit tests for algorithmic modules (Dijkstra, graph utilities), integration tests for API flows (auth, bookings, roads), and manual/user-acceptance tests for the UI and graph interactions. Continuous integration pipelines run automated tests on merges to prevent regressions. Iterative releases also enable targeted performance tuning (graph rendering, Dijkstra latency, predictor response) based on measured metrics from the demo environment, and risk is reduced by isolating changes and providing quick rollback points.

Operational concerns — JWT secrets, password hashing, predictor fallback, and DB seeding — are built into the process so each increment is runnable and demonstrable in the lab environment. Stakeholders see steady, measurable progress via milestone demos that showcase completed functionality and planned next steps. Overall, the iterative model balances speed, quality, and adaptability: it delivers a minimum viable product early, supports experimentation with ML models and weighting strategies, and provides a clear path to a robust, extendable Smart Traffic Automation-system.

## HIGH LEVEL DESIGN

The High Level Design is a crucial phase in the Software Development Life Cycle (SDLC) that translates the Software Requirements Specification into a structured system architecture. It defines what the system will look like as a whole and provides an overall view of how different modules or components interact to meet the functional requirements.

HLD serves as a blueprint for system design, helping developers, testers, and project stakeholders understand the system’s structure before actual coding begins. It focuses on modularity, data flow, and inter-component communication rather than detailed logic.

A typical HLD document includes the following elements:

1. System Architecture Diagram – A block diagram showing major modules, external systems, and their interactions.
2. Module Description – Explanation of each module, its purpose, inputs, and outputs.
3. Database Design Overview – High-level description of key entities, relationships, and data sources.
4. Interface Design – Description of user interfaces, external APIs, and communication interfaces.
5. Technology Stack – Hardware, software, and frameworks to be used in development.
6. Assumptions and Constraints – Conditions considered while designing the architecture

### How to Build an HLD

To prepare an effective High Level Design, follow these steps:

1. Review the SRS Document

Carefully analyze functional and non-functional requirements to understand what the system must achieve.

1. Identify Major Modules

Divide the system into logical modules such as *User Management, Data Processing, Reporting, AI Model Module,* etc.

1. Define Module Interactions

Determine how these modules will communicate — through APIs, shared data, or message queues.

1. Draw the System Architecture Diagram

Use standard symbols (rectangles for modules, arrows for data flow) to represent module relationships and external components.

1. Design Data Flow Diagrams (Level 0 or Level 1)

Illustrate how information moves between input, processing, and output stages.

1. Outline Database and External Interfaces

Specify major database entities and how the system connects with external tools, cloud services, or user devices.

1. Select Technology Stack

Choose appropriate languages, frameworks, cloud platforms, and tools that align with performance and scalability needs.

1. Validate with Stakeholders

Review the design with project members or guide to ensure it satisfies the requirements and constraints defined in the SRS.

The High Level Design provides a macroscopic view of the software system and ensures that the architecture is well-structured, efficient, and aligned with business goals. It acts as a foundation document guiding developers during implementation and testers during verification. A well-prepared HLD reduces project risks, improves clarity, and ensures the success of subsequent design and development phases.

## AI ETHICS EVALUATION

As Artificial Intelligence (AI) becomes an integral part of modern software systems, AI Ethics Evaluation ensures that AI models and systems are designed and deployed responsibly, fairly, and transparently. Ethical AI practices are essential to maintain user trust, prevent harm, and promote accountability in intelligent software that performs autonomous or data-driven decision- making.AI ethics focuses on ensuring that technology benefits humanity while avoiding discrimination, bias, or privacy violations. It forms the foundation for responsible innovation and sustainable AI deployment. Ethical evaluation also aligns AI systems with legal, societal, and environmental responsibilities to ensure long-term acceptance and reliability.

### Key Principles of AI Ethics Fairness:

AI systems should treat all individuals and groups equitably, ensuring that outcomes are free from bias related to gender, race, age, economic background, or other personal characteristics.

### Transparency:

The internal working of AI algorithms, data handling methods, and decision logic must be explainable and understandable to both users and regulators. This ensures that AI systems remain accountable and interpretable.

### Accountability:

Developers, organizations, and stakeholders must take full responsibility for the results, actions, and decisions made by AI systems. Ethical governance and clear responsibility frameworks are crucial for maintaining accountability.

### Privacy and Data Protection:

AI systems should handle personal and sensitive information securely. Data must be collected, stored, and processed in compliance with privacy regulations such as the General Data Protection Regulation (GDPR), ensuring users’ consent and confidentiality.

### Safety and Reliability:

AI applications must be tested thoroughly to ensure consistent, safe, and predictable performance. Built-in safeguards should prevent unintended actions or harm to users or the environment.

### Human-Centric Design:

AI should be developed to assist and enhance human decision-making, not replace or manipulate it. Systems must be designed to complement human intelligence and uphold user autonomy and dignity.

## USE CASE DIAGRAM

### Aim

To create use case model for <your software> system using CASE tools (Argo UML) and

write the use case template for each scenario

### Procedure

1. Go to Create > New Use Case Diagram.
2. To add an actor to the diagram click on the actor icon on the editing pane toolbar.
3. Then click at the location where you wish to place it.
4. To add the use case, do the same steps as it is done for actor.
5. For Include, the including (main) use case should be selected first (button down) and the included (subsidiary) use case second (button release). It is possible to name include relationships using the property tab.
6. For Extend, the extending (subsidiary) use case should be selected first (button down) and the extending (main) use case second (button release).
7. The rectangle tool can be used to draw the boundary box

<Write a small write up about your use case diagram For ex: Figure 3.1 shows the use case

diagram for project ABC with actors like….>

**Symbols Used:**

|  |  |
| --- | --- |
|  | USE CASE |
|  | RELATION |
|  | ACTOR |
| << include>> | KEYWORD |
| << extend>> |

1. **Use Case Templates**

SYSTEM BOUNDARY

* 1. **UC - 001**

|  |  |
| --- | --- |
| **Use Case Name:** | **<Name of the use case>** |
| Actors: | <Actors involved> |
| Description: | <Explanation and purpose of the use case> |
| Pre – conditions: | <Conditions or state before the scenario> |
| Post – conditions: | <Resulting Conditions or state> |
| Main Success Scenario: | <What happens if everything goes right> |
| Alternative Scenario1: | <What happens if something goes wrong> |
| Alternative Scenario2: | <What happens if something goes wrong> |
| … |  |

# ACTIVITY DIAGRAM

### Aim

To create activity diagram for <your software> system using CASE tools

### Procedure

* + 1. Go to Create > New Activity Diagram.
    2. Add an action state to the diagram by selecting and drawing the “Action State” icon.
    3. Add a transition between two action states selected using button motion (from the originating action state to the receiving action state).
    4. Add an initial pseudostate and a final state to the diagram. (There is nothing to stop you adding more than one initial state to a diagram).
    5. Add a junction (decision) pseudostate to the diagram. A well-formed junction should have one incoming transition and two or more outgoing.
    6. We can also add a fork and join pseudostate to the diagram. Fork should have one incoming transition and two or more outgoing. Join should have one outgoing transition and two or more incoming.
    7. Add a callstate to the diagram. A call state is an action state that calls a single operation. Hence, the name of the operation being called is put in the symbol, along with the name of the classifier that hosts the operation in parentheses under it.
    8. Add objectflowstate to the diagram. An objectflowstate is an object that is input to or output from an action.

**Symbols used**

|  |  |
| --- | --- |
|  | INITIAL STATE |
|  | FINAL STATE |
|  | TRANSITION |

|  |  |
| --- | --- |
|  | ACTION |
|  | BRANCHING |
|  | SYNCHRONIZATION BAR (FORK/JOIN) |

# CLASS DIAGRAM

### Aim

To create class diagram for <your software> system using CASE tools

### Procedure

1. Go to Create > New Class Diagram.
2. In the properties pane, you can rename your diagram by modifying the Name field.
3. For class diagram, specify class name, attributes and operations.
4. To add a class to the diagram, select “New class” icon and draw.
5. When the mouse is over a selected class it displays two handles to left and right which may be clicked or dragged to form association relationships.
6. Add new attribute to the currently selected class. The attribute is given the default name newAttr of type int and can be edited by double clicking the button and using the keyboard, or by selecting with button click (after the class has been selected) and using the property tab.
7. The operation is given the default name newOperation with no arguments and returns type void and can be edited by double click and by using the keyboard, or by selecting with click (after the class has been selected) and using the property tab.

**SYMBOLS USED:**

|  |  |
| --- | --- |
|  | GENERALIZATION |
| **Name** | OBJECT |

|  |  |
| --- | --- |
|  | DEPENDENCY |
|  | REALIZATION |
|  | COMPOSITION |
|  | AGGREGATION |

# INTERACTION DIAGRAMS

### Aim

To create interaction diagrams such as system sequence diagrams and collaboration diagrams for

<your software> system using CASE tools

### Procedure Sequence Diagram

1. Go to Create > New Sequence Diagram.
2. Add a classifier role to the diagram by selecting the classifier role icon.
3. Then add a call message between two classifier roles selected using button motion (from the originating classifier role to the receiving classifier role).
4. We can also add a return message between two classifier roles selected using button motion (from the originating classifier role to the receiving classifier role).

### Collaboration Diagram

* 1. Go to Create > New Collaboration Diagram.
  2. Add a classifier role to the diagram which is similar to that of the interaction diagram.
  3. Add an association role between two classifier roles selected using button motion (from the originating classifier role to the receiving classifier role).
  4. Add a message to the selected association role.
  5. We can also add a generalization between two model elements selected using button (from the child to the parent).
  6. Dependency between two model elements selected using button motion (from the dependent model element) can be added if necessary.

Symbols used

|  |
| --- |
| <<Name>> |
| OBJECT |

|  |  |  |
| --- | --- | --- |
|  |  | OBJECT LIFELINE |
|  | |
| <<Name>> | | MESSAGE |
|  | | FOCUS OF CONTROL |
|  | | SELF LOOP |

**Symbols used**

|  |  |
| --- | --- |
| <<Name>> | OBJECT |
|  | FLOW OF MESSAGE |
| <<Name>> | MESSAGE WITH SEQUENCE NUMBER |

# STATE DIAGRAM

### Aim

To create state chart diagram for <your software> system using CASE tools

Procedure

1. Go to Create > New State Chart Diagram.
2. Add a simple state to the diagram.
3. Add a transition between two states selected using button 1 motion (from the originating state to the receiving state).
4. Add an initial pseudostate to the diagram.
5. Then add a Call Event as trigger to a transition.
6. Now add a guard condition to a transition if necessary.
7. Add a call action (i.e. the effect) to a transition.

|  |  |
| --- | --- |
|  | Initial State |
|  | Final State |
|  | State |
|  | Transition |
|  | Synchronization and Splitting of Control |

# COMPONENT AND DEPLOYMENT DIAGRAMS

### Aim

To create component and deployment diagrams for <your software> system using CASE tools

### Procedure

1. Go to Create > New Deployment Diagram.
2. Add a node to the diagram. For convenience, when the mouse is over a selected node it displays four handles to left, right, top and bottom which may be dragged to form association relationships.
3. Add a node instance to the diagram. For convenience, when the mouse is over a selected node instance it displays four handles to left, right, top and bottom which may be dragged to form link relationships.
4. Similarly for component diagram use component and component instance which may contain the dependency relationships.
5. The constraint that associations between classes and interfaces must not be navigable from the interface still applies on deployment diagrams.
6. Add an object to the diagram. For convenience, when the mouse is over a selected object it displays four handles to left, right, top and bottom, which may be dragged to form link relationships.
7. Then add a link between two model elements (node instance, component instance or object) selected using button motion

**Symbols used**

COMPONENT

**Symbols used**

INTERFACE

|  |  |
| --- | --- |
|  | Node |
|  | Association |
|  | Component and Nodes |

## LOW LEVEL DESIGN

**Introduction**

The Low Level Design (LLD) phase is the next step after the High Level Design (HLD) and provides a detailed and technical view of the software system. While HLD defines what modules will exist and how they interact, the LLD defines how each module will be implemented — including algorithms, data structures, logic, and interface details. LLD acts as a blueprint for programmers, bridging the gap between system architecture and actual code development. It ensures that every requirement defined in the SRS and every component identified in the HLD has a precise implementation strategy.

**Components of LLD**

1. **Module Description**: Defines internal logic, algorithms, input-output formats, and data validation techniques for each module.
2. **Class and Sequence Diagrams:** Used to represent classes, methods, and object interactions within and between modules.
3. **State Transition Diagrams:** Describe how the system behaves based on events, especially in reactive or real-time components.
4. **Pseudocode or Algorithmic Logic**: Defines the exact step-by-step procedure to achieve the required functionality.
5. **Error Handling and Exception Flow:** Specifies conditions for error detection, recovery, and reporting mechanisms.
6. **Interface Specifications:** Details about how a module interacts with others, including input/output parameters, file structures, or APIs**.**

## DATABASE DESIGN

Database Design is a crucial part of the software design process that ensures efficient storage, retrieval, and management of data.

It defines the logical structure of the database, including the relationships between entities, attributes, and constraints, ensuring data integrity and consistency across the system..

**Steps in Database Design**

1. **Requirement Analysis:** Identify what data needs to be stored and how it will be used by different modules.
2. **Entity-Relationship (ER) Modeling:** Develop an **ER diagram** showing entities, attributes, and relationships (one-to-one, one-to-many, many-to-many).
3. **Normalization:** Apply **1NF, 2NF, and 3NF** (Normal Forms) to eliminate data redundancy and improve data integrity.
4. **Schema Definition:** Define tables, data types, constraints, and keys using SQL or ORM models.
5. **Database Security and Access Control:** Implement authentication, authorization, and encryption policies to safeguard sensitive data.
6. **Testing and Optimization:** Validate queries and optimize database performance for faster data retrieval and minimal latency.

# MODULE 3 SYSTEM TESTING

### Aim

To write test cases and perform system testing for Smart Traffic Routing System

**Test cases**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Scenario / Function** | **When** | **Input Data** | **Expected Result** | **Actual Result** | **Status (Pass/Fail)** |
| TC\_001 | Register new user | User opens /register and submits valid details | [Name: Test UserEmail: new@me.comPassword: Test@123](mailto:new@me.com) | 200 OK from /api/auth/register, JSON with token, role, email, frontend navigates to /book, user exists in DB | System successfully created account and redirected user to booking page | Pass |
| TC\_002 | Login with valid credentials | User submits correct email & password on /login | Email: user@user.com / Admin: admin@admin.comPassword: User@123 / Admin@123 | 200 OK from /api/auth/login, token saved in localStorage, redirect to /book or /admin | User logged in successfully and redirected to correct dashboard | Pass |
| TC\_003 | Login with invalid credentials | User submits wrong password | [Email: user@user.comPassword: wrongpass](mailto:user@user.com) | 401 Unauthorized, error message shown, no token stored, stay on login | Error message displayed correctly and no login occurred | Pass |
| TC\_004 | Booking creation (happy path) | User selects src & dest (graph) and clicks Calculate Route | Source: N1Dest: N7Slot: 2025-10-26T10:00 | POST /api/bookings/book returns booking JSON {path[], recommendedSpeed, slot}; UI shows route & speed; booking in DB | Booking created successfully and displayed on UI | Pass |
| TC\_005 | Booking when source == destination | User selects same node for src & dest and attempts booking | Source/Dest: N3 | Validation message "Source and destination cannot be same"; no POST request | Error message displayed correctly and no API call made | Pass |
| TC\_006 | ML predictor fallback | User makes booking that triggers predictor | Source: N2, Dest: N9, Slot: 2025-10-26T09:00 | Booking returned; recommendedSpeed present (heuristic); ML error logged on server | Booking completed successfully using heuristic | Pass |
| TC\_007 | Admin view: see all bookings | Admin opens admin dashboard | — | Admin UI lists all bookings; CSV download contains those records | Admin dashboard displayed all bookings and CSV exported properly | Pass |
| TC\_008 | User cannot see other users' bookings | Login as userA and view bookings | userA: a@x.com; userB: b@x.com | GET /api/bookings returns only records with userEmail = userA ; admin still sees all | UserA saw only own bookings, admin saw all | Pass |
| TC\_009 | Toggle road status (admin) | Admin clicks Close/Open on a road | Road id: <sample\_id> | POST /api/roads/update/:id returns ok; roads collection shows new status; subsequent route avoids closed road | Road status updated successfully and reflected in UI | Pass |
| TC\_010 | Graph selector interaction | User clicks node (source) and Ctrl+Click node (dest) | Click node N5 then Ctrl+Click N12 | Node visuals update; hidden selects equal N5/N12; events work in Chrome/Firefox/Edge | Graph interaction and highlighting worked correctly | Pass |
| TC\_011 | Path highlighting on booking selection (admin) | Admin clicks "Highlight" on a booking entry | Click Highlight on booking id | Graph shows highlighted links/nodes for path; UI scrolls to booking | Highlighting worked and UI scrolled correctly | Pass |
| TC\_012 | Token expiry & unauthorized access | Client sends request with expired/invalid token | Authorization: Bearer <expired\_token> | 401 Unauthorized; frontend shows login page / error | System redirected to login on invalid token | Pass |
| TC\_013 | API failure handling | User attempts login or booking | Try login while backend off | Frontend displays "Network error" or similar; console shows handled error | Network error handled gracefully without crash | Pass |
| TC\_014 | Basic performance sanity | Load booking page and interact (click nodes) | — | Graph renders within ~2s; interactions responsive (no freeze) | Graph loaded fast and interactions smooth | Pass |
| TC\_015 | Data integrity after booking | Inspect booking document in DB | Booking id from TC\_004 | DB doc has user, userEmail, path, slot, recommendedSpeed, createdAt with expected values | Database record contained all correct details | Pass |

**Result:**

Smart Traffic Automation: A practical, user-first traffic management web app built iteratively with a React frontend, Node.js/Express backend, MongoDB and RESTful APIs to enable easy extension and testing. Users can register/login, visually pick source & destination on an interactive node-graph, and book road-usage time slots; the server computes occupancy-aware optimal routes with Dijkstra and returns ML-backed recommended speeds to smooth flow. Admin features include road status control, booking oversight and CSV export. Extra conveniences — booking history, slot validation, heuristic fallback for the predictor, and real-time updates (WebSocket-ready) — make the system demo-ready and extendable for future analytics, map integration, or production deployment.