SOFTWARE DESIGN AND ARCHITECTURE (SE-211)

ONLINE DESIGN TOOL

COMPLEX ENGINEERING ACTIVITY

Course:
BESE 29 B

Prepared by :

- ZOIBA TABASSUM (478562)
- JANNAT SHAHEER (472248)
- LARAIB ZAHRA (486986)
- M. TANVEER HAIDER (486873)
- MUHAMMAD TALHA (486869)
- HASSAN IFTIKHAR (486855)

Submitted To
LEC . FAWAD KHAN

Vixel

Where Ideas Meet Precision

VIXEL is a robust Collaborative CASE tool that enhances project design, modeling, and management. It streamlines software development and system design, promoting efficient teamwork, structured workflows, and secure data handling for developers and engineers.

TABLE OF CONTENTS

INTRODUCTION

OVERVIEW

Overview

The CASE (Computer-Aided Software Engineering) tool is a software application that aids in the development and maintenance of software systems. It provides a suite of tools that assist developers throughout the entire software development lifecycle, from initial requirements gathering to system testing and maintenance. This tool aims to improve the efficiency, consistency, and quality of software development by automating and facilitating various processes involved in system design, coding, and maintenance.

Introduction

In modern software development, leveraging powerful tools for system design and development is crucial for ensuring efficiency and accuracy. An online design tool that uses the Model-View-Controller (MVC) architecture supports developers by separating the application's concerns, ensuring better organization and scalability. This tool is designed with five major modules: Requirement Analyzer, Diagramming Engine, Code Generator, Testing, and Maintenance, each providing unique functionalities to optimize the software development workflow.

What is the CASE tool?

The CASE tool is a specialized software designed to automate the tasks and processes involved in the development of software systems. It enables designers, developers, and testers to interact with the system through an intuitive interface. The online design tool that implements the MVC architecture is built to help teams better manage the software development lifecycle through clear separation of concerns. The five major modules allow for thorough analysis, accurate diagramming, automated code generation, efficient testing, and effective maintenance of the software system.

Purpose

The primary purpose of the CASE tool with MVC architecture is to improve software development by:

1. Enhancing collaboration between development teams through shared access to design tools.
   Streamlining the software design and development process.
2. Reducing human error through automated processes such as code generation and testing.
3. Ensuring that the system is easily maintainable and extensible by adhering to a structured design approach (MVC).
4. Providing a user-friendly platform to manage and track requirements and changes during the development lifecycle.

Vision & Strategy

The vision for this online design tool is to empower software developers to efficiently design, develop, test, and maintain systems using a modular and scalable toolset. The strategy is to focus on building a robust MVC architecture that allows for clear separation of concerns, enabling developers to work more efficiently, improve the maintainability of their code, and create high-quality software. Each module, from requirement analysis to maintenance, will be integrated to ensure seamless communication and synchronization across the development stages.

Scope

The scope of the CASE tool spans the entire software development lifecycle:

- **Requirement Analyzer:** Allows for gathering and managing requirements from stakeholders, ensuring clear, accurate, and traceable requirements.
- **Diagramming Engine:** Provides visual modeling capabilities, enabling the creation of system architectures, UML diagrams, and flowcharts.
- **Code Generator:** Automatically generates code from diagrams, ensuring consistency and speeding up the development process.
- **Testing:** Supports automated unit and integration testing to ensure code quality, correctness, and performance.
- **Maintenance:** Provides tools for managing changes, debugging, and improving the system over time, ensuring the system remains flexible and scalable as requirements evolve.

HIGH LEVEL DESIGN

ARCHITECTURE STYLE:
MODEL-VIEW-CONTROLLER

1. Purpose

The High-Level Design (HLD) provides a structured overview of the system architecture for the Online Design Tool. It defines the main components, their responsibilities, and how they interact within the MVC framework. This section helps ensure a shared understanding of the system's structure among stakeholders and sets the foundation for mid-level and low-level design.

2. System Overview

The Online Design Tool is a web-based application that facilitates software design and development through an interactive, modular interface based on the Model-View-Controller (MVC) architecture. It guides users through the software engineering lifecycle, from requirement analysis to code generation and maintenance. The tool allows users to input and analyze requirements, create software design diagrams (e.g., UML), automatically generate source code in their chosen programming languages, and test or refine designs using built-in modules. This seamless integration makes the platform ideal for students, developers, and software teams.

- **Core Functional Areas:**
  - Requirement Analyzer
  - Diagramming Tool
  - Code Generator
  - Testing Module
  - Maintenance Module

3. Architecture Overview

- **Architectural Pattern:**

MODEL-VIEW-CONTROLLER (MVC)

- **Description:**

  - **Model:** Manages application data and business rules.
  - **View:** User interface and visualization (diagrams, forms).
  - **Controller:** Handles user input and mediates between View and Model.

- **Rationale for Selecting MVC**

    - **Separation of Concerns**

MVC separates the business logic (Model), user interface (View), and input control (Controller), making the system easier to manage and scale.

    - **Modular Design**

Each component (Model, View, Controller) can be developed, tested, and maintained independently, promoting flexibility and reducing complexity.

    - **Ease of Maintenance**

Changes in the user interface or business logic can be made without affecting other components, simplifying updates and debugging.

    - **Reusability**

The Model can be reused across different views (e.g., for generating different types of diagrams), and Views can be modified independently of the business logic.

    - **Scalability**

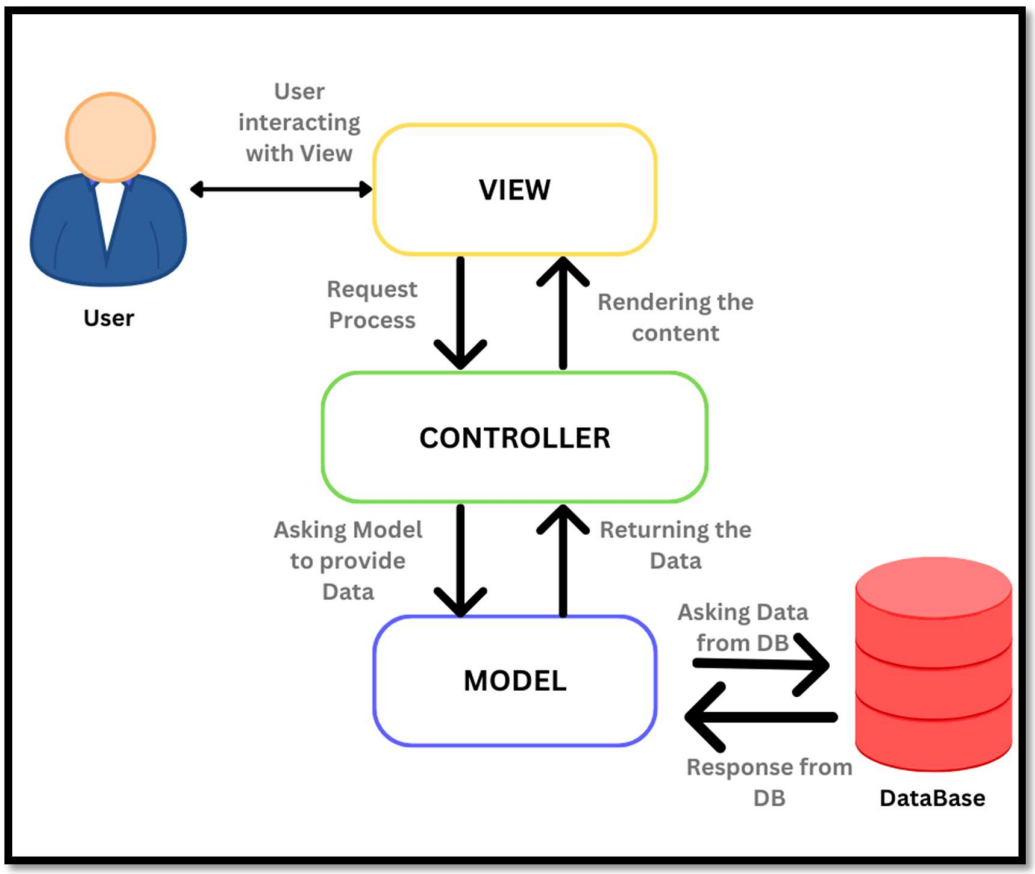
MVC allows for better scalability, as new features and modules can be added without disrupting the entire system's architecture.

    - **Improved Testing**

MVC facilitates unit testing, as business logic (Model) and user interface (View) can be tested separately.

    - **Clearer User Interaction Flow**

The Controller acts as an intermediary to manage user input, providing better control over the flow of user interactions.



4. High-Level Module Descriptions

The Online Design Tool consists of core modules that support various stages of software development and interact seamlessly to ensure a smooth workflow.

- **Requirement Analyzer**

The Requirement Analyzer module serves as the entry point of the system, where users input functional and non-functional requirements. It parses and validates the provided requirements to ensure they are clear, complete, and logically structured. Once validated, the module transforms these requirements into a structured format that can be consumed by other modules. The processed data is then forwarded to the Diagramming Tool to support visual modeling and to the Code Generator for generating the initial code structure based on the user's intent.

- **Diagramming Tool**

The Diagramming Tool provides a visual interface for designing system architecture using standard modeling techniques such as UML diagrams. It allows users to create, edit, and organize diagrams that represent the system's components and their interactions. This module interacts closely with the data models defined by the Requirement Analyzer and synchronizes with the Code Generator to ensure that the visual representations accurately drive code generation. It forms the central component of the design workflow, bridging the gap between requirement gathering and implementation.

- **Code Generator**

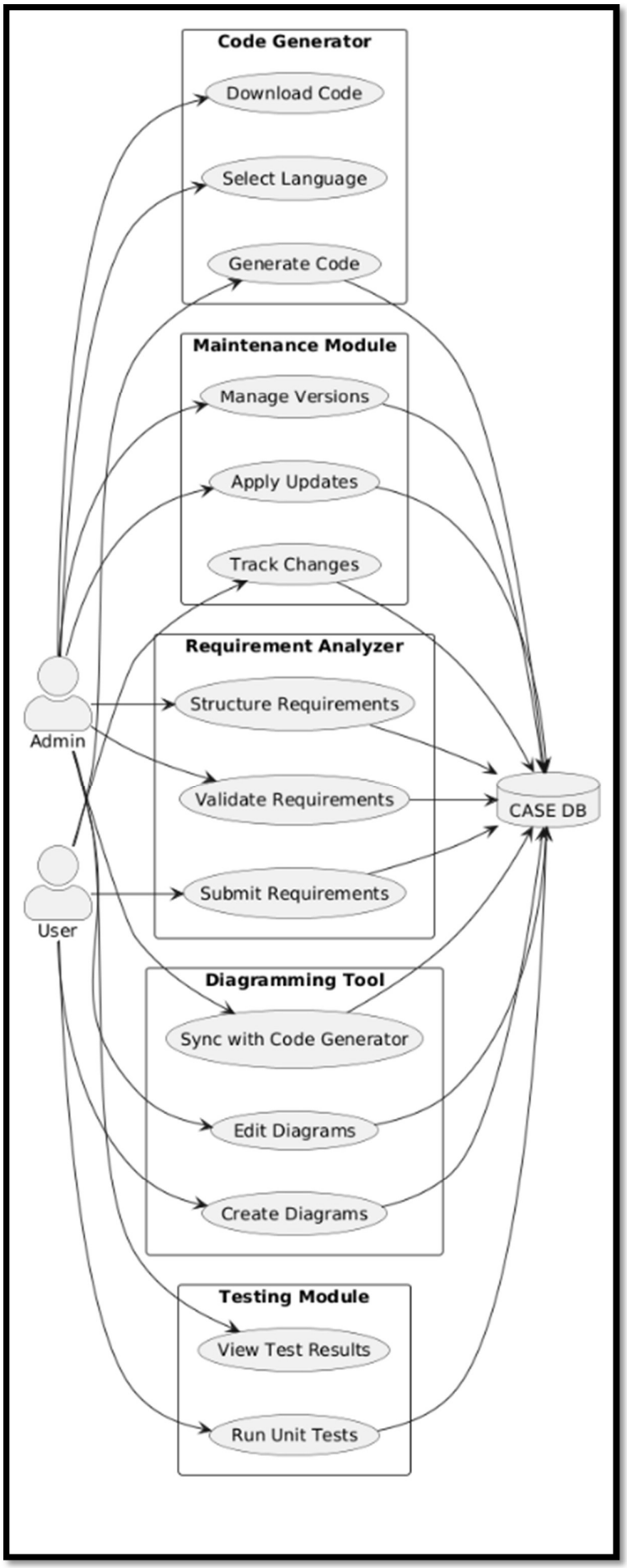
The Code Generator automates the process of converting structured requirements and visual diagrams into source code. It supports multiple programming languages through predefined templates and generates code skeletons that align with industry standards. By interpreting data from both the Requirement Analyzer and the Diagramming Tool, the Code Generator streamlines the transition from design to implementation, reducing manual coding effort and minimizing errors.

- **Testing Module**

The Testing Module allows users to perform basic testing operations on the generated code. It supports unit and functional testing, enabling users to validate whether the generated output meets the expected behavior. Test results and feedback are presented to the user in a simplified format, helping them identify logical or structural issues early in the development cycle. This module works in conjunction with the Code Generator to ensure quality and reliability.

- **Maintenance Module**

The Maintenance Module facilitates post-development activities such as version control integration, code refactoring, and updates to existing designs. It allows users to track changes, manage versions of their diagrams and code, and implement improvements based on testing results or updated requirements. This module ensures long-term maintainability of projects and supports continuous enhancement of the design artifacts and generated code.


Code Generator
Download Code
Select Language
Generate Code
Maintenance Module
Manage Versions
Apply Updates
Track Changes
Requirement Analyzer
Structure Requirements
Validate Requirements
Submit Requirements
Admin
User
CASE DB
Diagramming Tool
Sync with Code Generator
Edit Diagrams
Create Diagrams
Testing Module
View Test Results
Run Unit Tests

5. Data Flow Overview

The system's data flow enables a smooth transition from project requirements to code generation, testing, and maintenance. Each module builds on the output of the previous one, ensuring efficient development with continuous feedback for accuracy and adaptability.

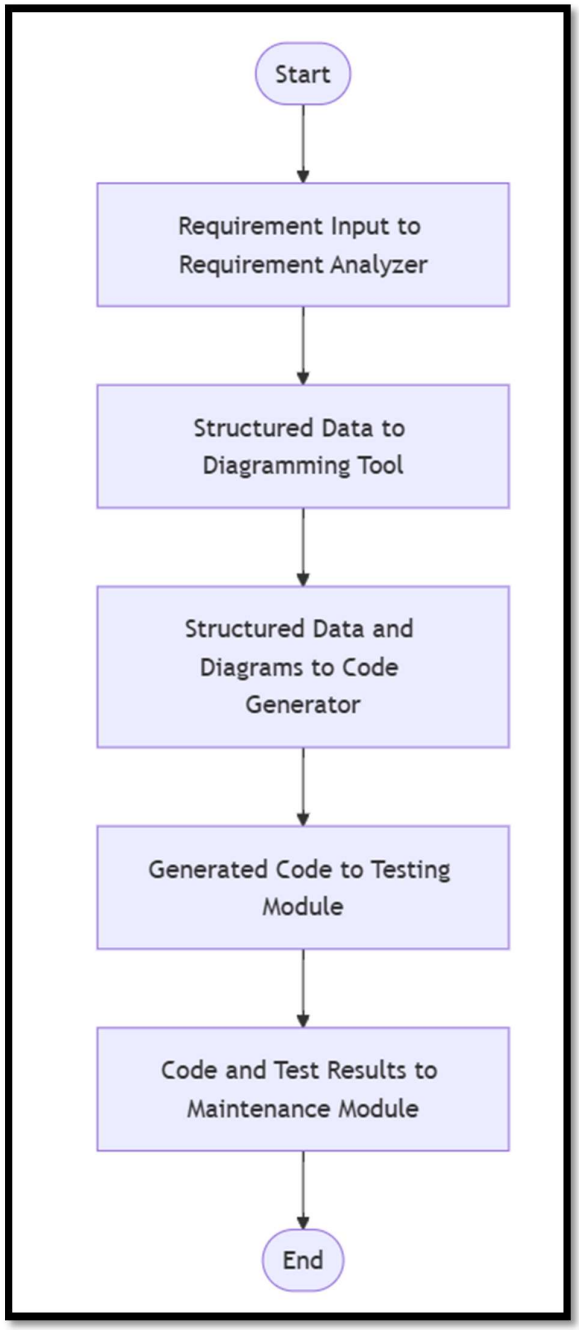
1. **Requirement Input to Requirement Analyzer**:
   Users provide project requirements. The Requirement Analyzer structures and organizes this input to make it usable for the next steps.

2. **Structured Data to Diagramming Tool**:
   The structured requirements are passed to a Diagramming Tool, which creates system models such as UML diagrams to visualize the architecture.

3. **Structured Data & Diagrams to Code Generator**:
   Both the structured data and diagrams are sent to the Code Generator. It uses templates to produce initial source code for the system.

4. **Generated Code to Testing Module**:
   The generated code undergoes unit and functional testing. This step ensures the code behaves as expected and helps catch issues early.

5. **Code & Test Results to Maintenance Module**:
   The final code and test feedback are handed off to the Maintenance Module, which supports refactoring, versioning, and ongoing updates to the project.



6. Deployment Architecture

The online design tool is built on a **Model-View-Controller (MVC)** architecture and is designed for smooth operation in a web environment. It enables end-to-end project automation — from requirements input to code generation and maintenance — through an accessible browser-based interface.

Client (View Layer)

- The **browser-based front end** serves as the user interface for the entire system.

- Users can:
    - Enter project requirements
    - Visualize generated diagrams
    - View and interact with code and test results

- Built using modern JavaScript frameworks (e.g., React, Vue) to provide a responsive and dynamic user experience.
- Communicates with the backend via **API calls** for real-time updates and data processing.

Server (Controller Layer)

- Acts as the **intermediary** between the client and backend services.
- Routes incoming requests to appropriate modules, including:
    - **Requirement Analyzer**: Parses and structures user input
    - **Diagramming Engine**: Generates UML or system diagrams
    - **Code Generator**: Produces source code from structured input
    - **Testing Module**: Executes automated tests on generated code
    - **Maintenance Module**: Handles refactoring, updates, and version control

- Implements application logic, request handling, session management, and security features.
- Exposes RESTful APIs for modular, scalable interactions.

Backend (Model Layer)

- Handles core data operations and system logic.
- Manages:
    - Structured requirement data
    - Diagram configurations
    - Code generation templates
    - Test results and reports

- Interacts with the database for persistent storage and retrieval of project data.

Database

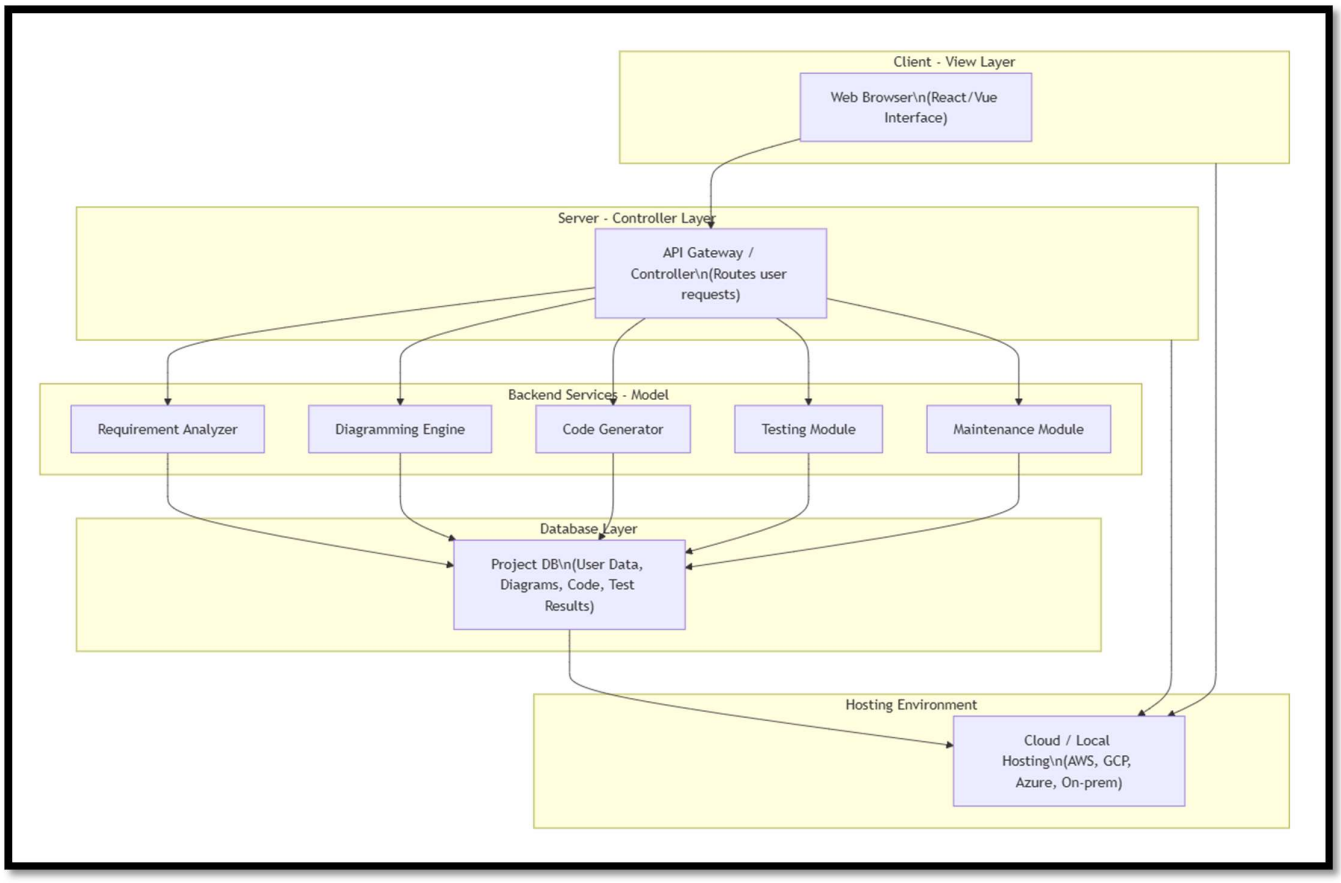
- A central data repository that stores:
    - User accounts and session data
    - Project requirements and analysis output
    - Generated diagrams and source code files
    - Testing outcomes and maintenance metadata

- Can be implemented using SQL (e.g., PostgreSQL) or NoSQL (e.g., MongoDB) depending on scalability and structure needs.

Hosting & Deployment

- Designed for cloud-first deployment, offering flexibility, scalability, and remote access.
    - Compatible with AWS, Google Cloud, Azure, or any standard cloud infrastructure.

- Can also be deployed locally or on-premise in secure environments.
- Uses Docker for containerization and Kubernetes (optional) for orchestration.
- CI/CD pipelines ensure smooth updates and automated testing during deployment.



7. Technology Stack

The online design tool is built using a modern, scalable technology stack structured across multiple layers:

Layer	Technology Used
Frontend	React, Html, CSS, JS
Backend	Node.js, Express, Java
Database	MySQL, MongoDB
Others	Rest Apls, Docker, Git

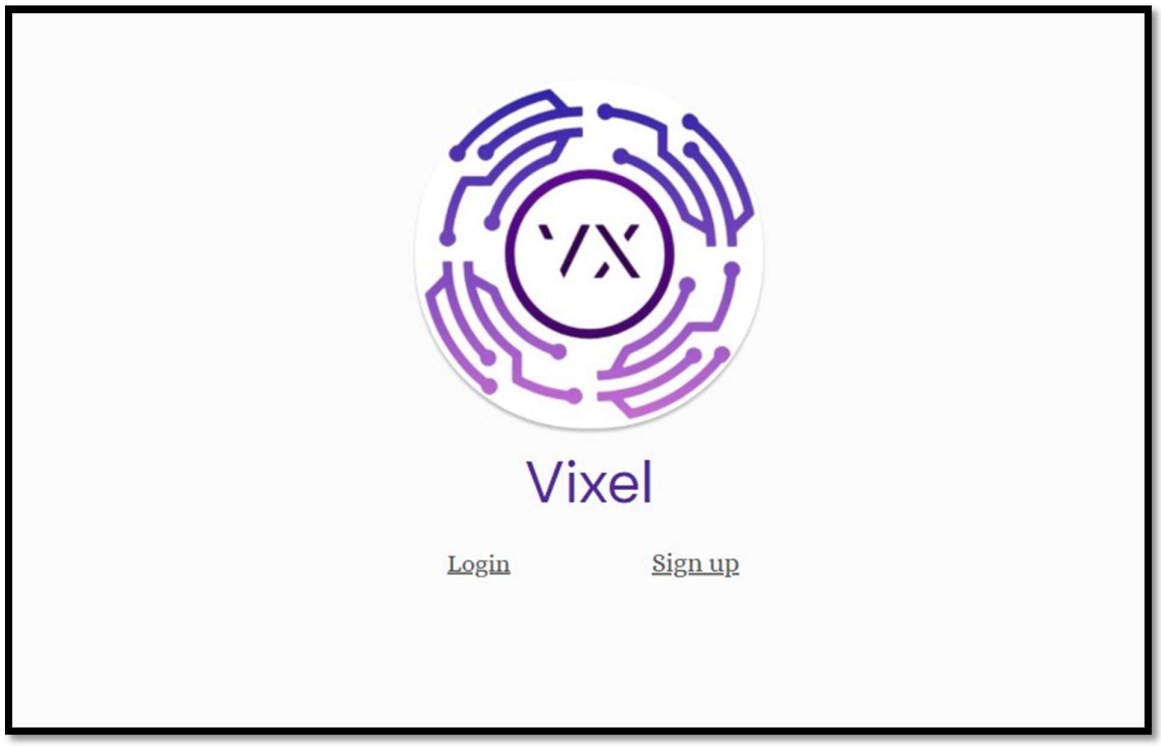
8. Assumptions & Limitations

- The tool requires a stable internet connection for full functionality.
- Initially optimized for desktop browsers; mobile support may come in later phases.
- The MVP version may have a limit on concurrent users due to infrastructure constraints.
- Offline usage is not supported in the current design.
- Integration with third-party tools is limited in early versions and will be expanded over time

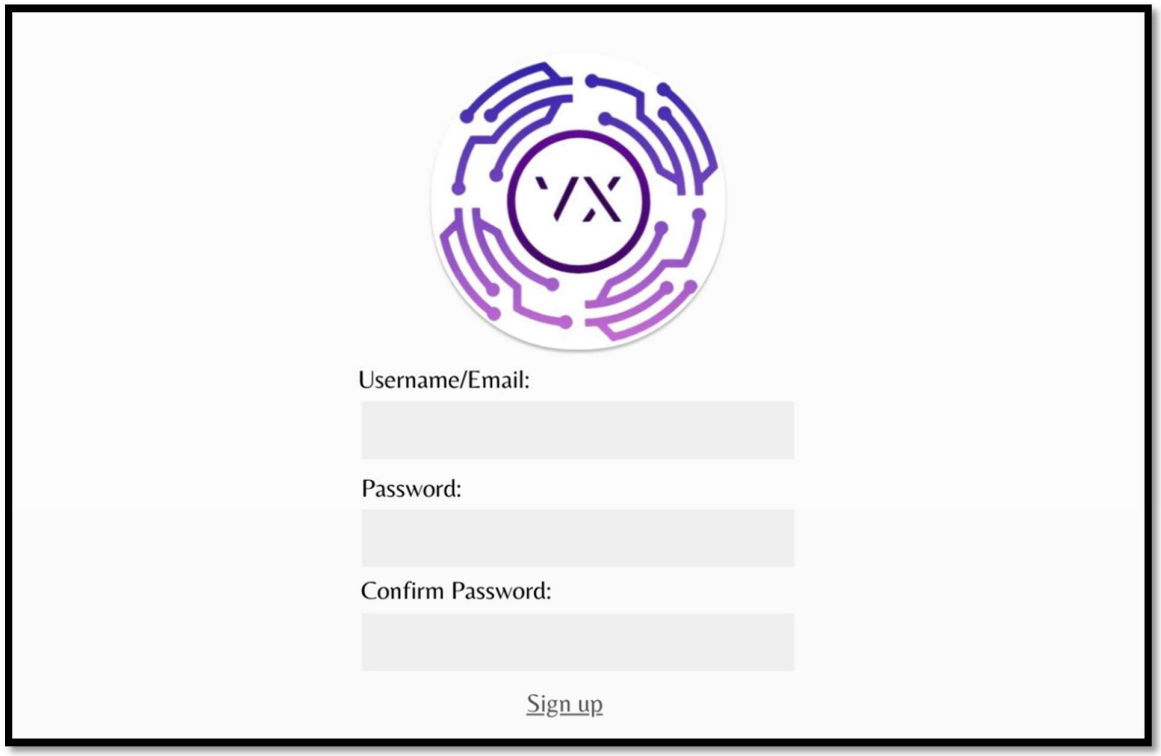
MID LEVEL DESIGN

GRAPHICAL USER INTERFACE

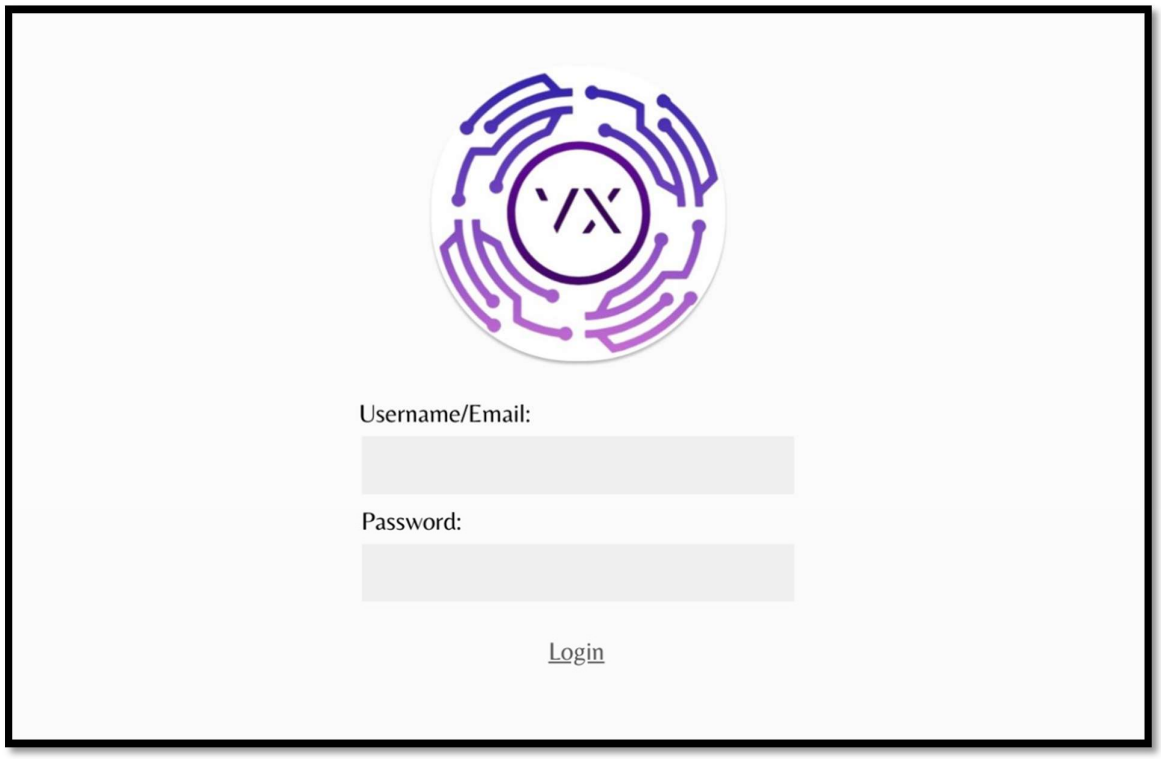
Start Up:



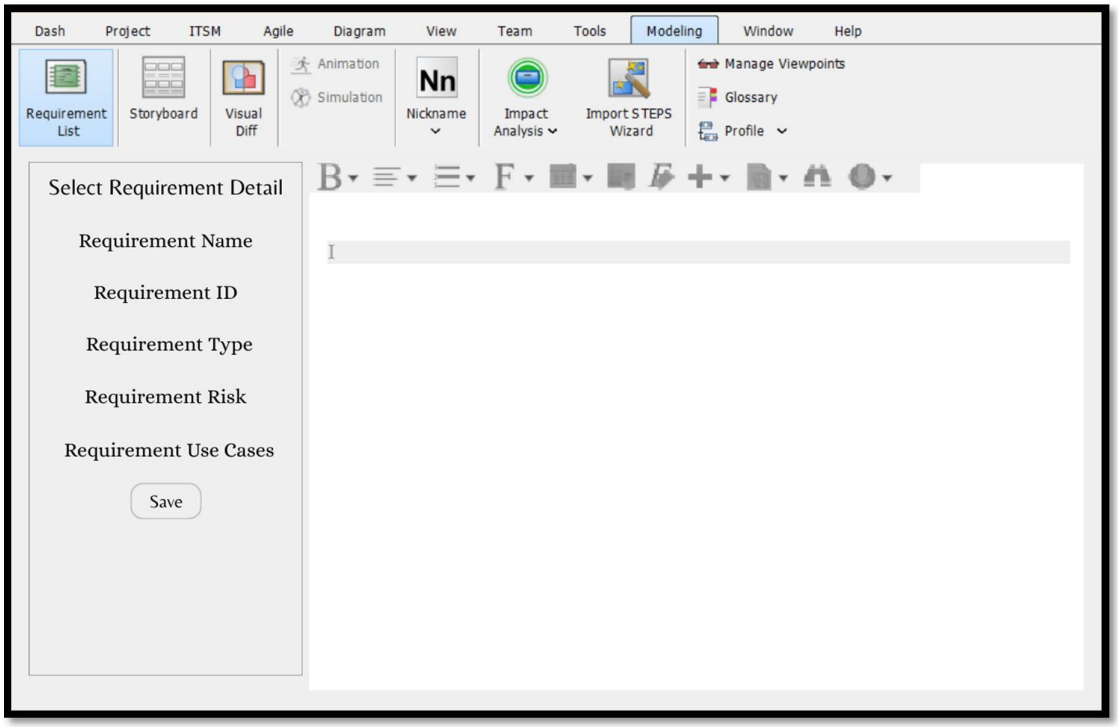
Sign Up:



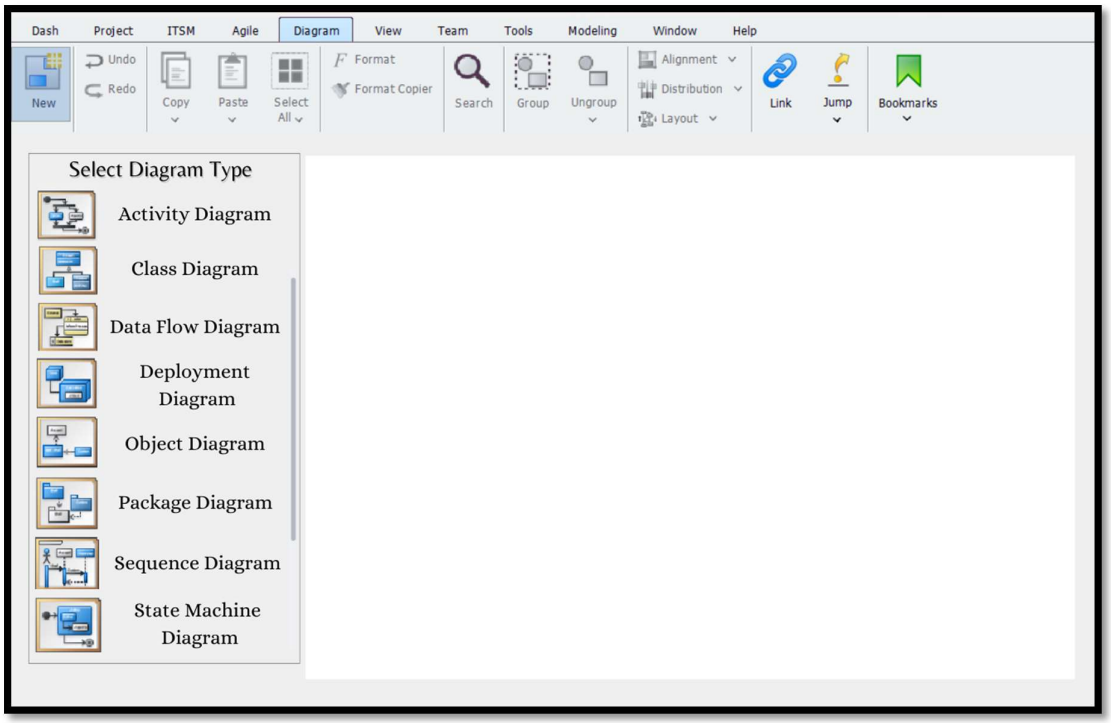
Login:



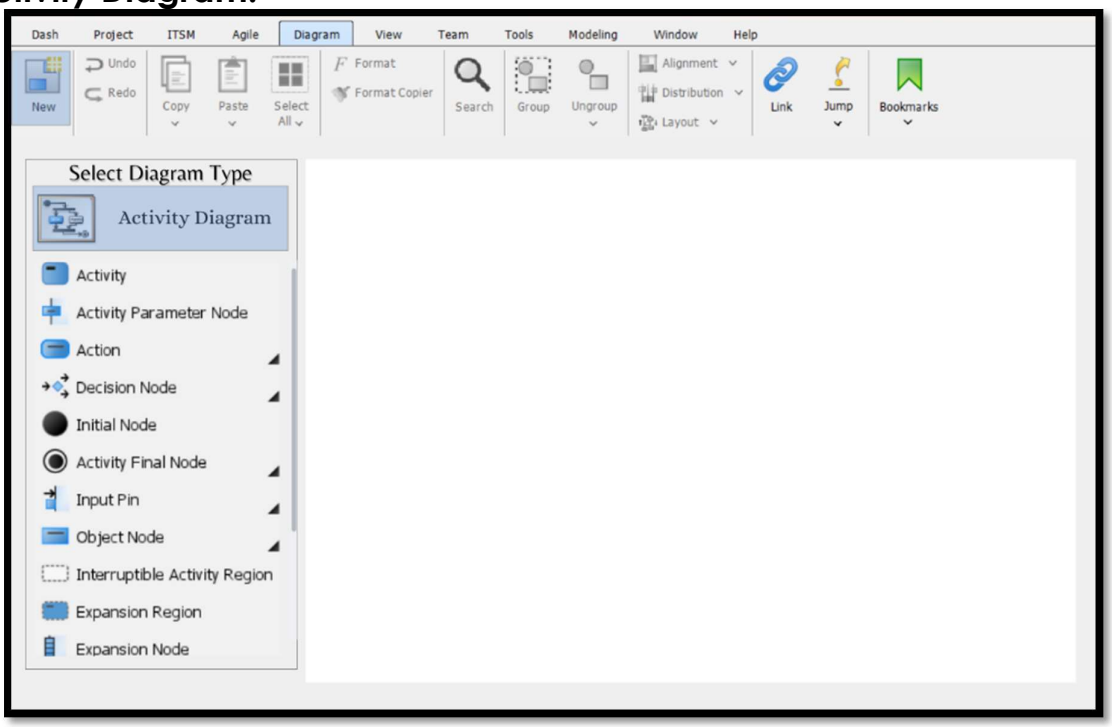
Requirements:



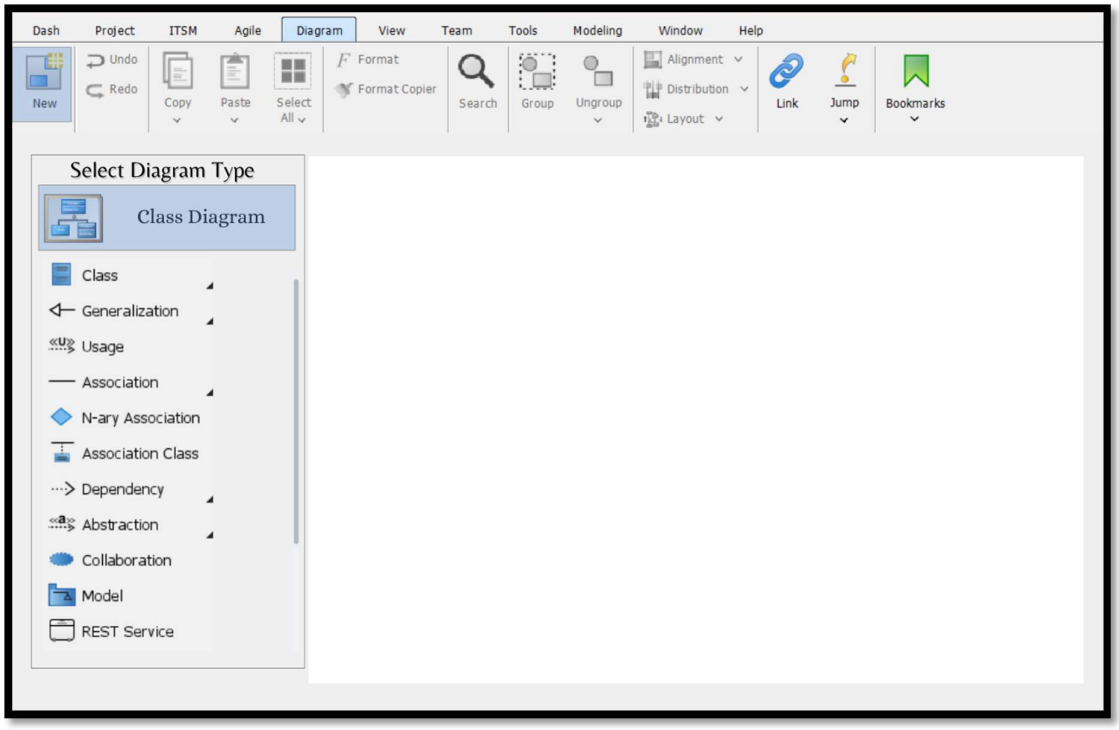
Diagramming Tool:



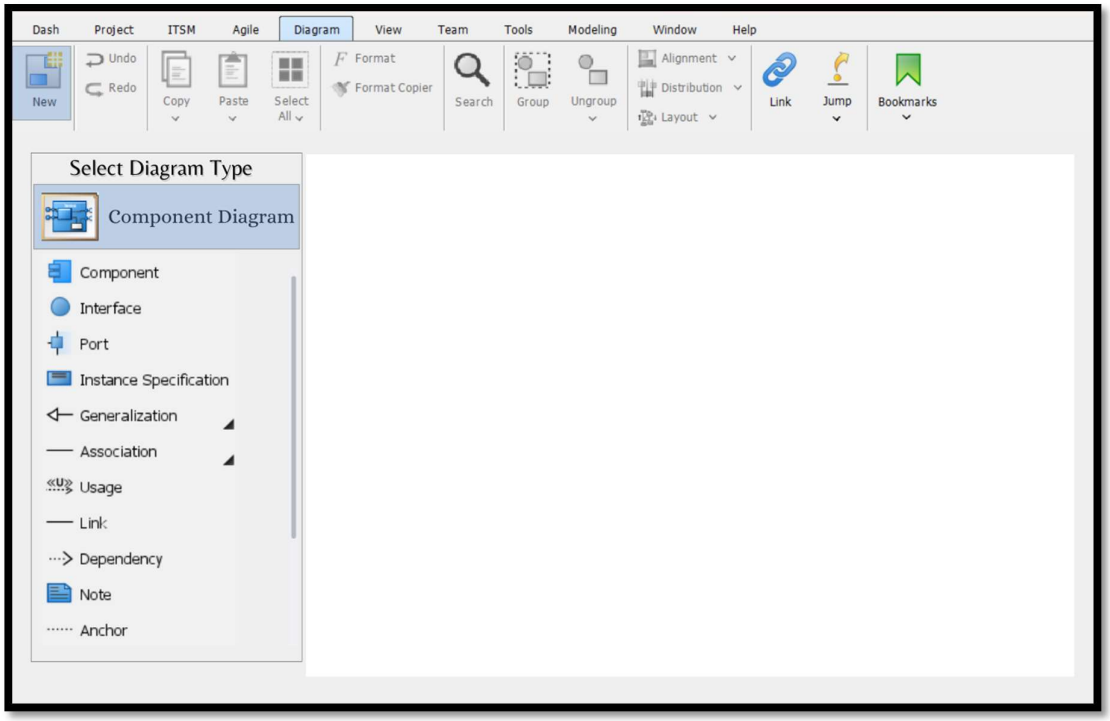
Activity Diagram:



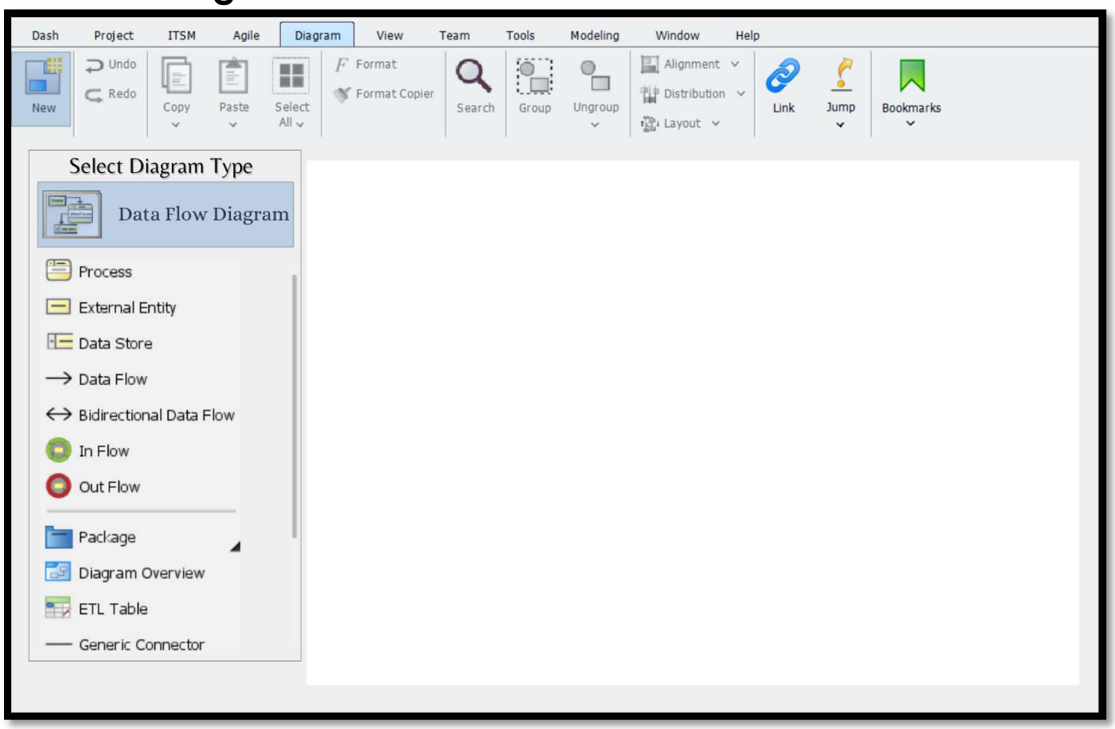
Class Diagram:



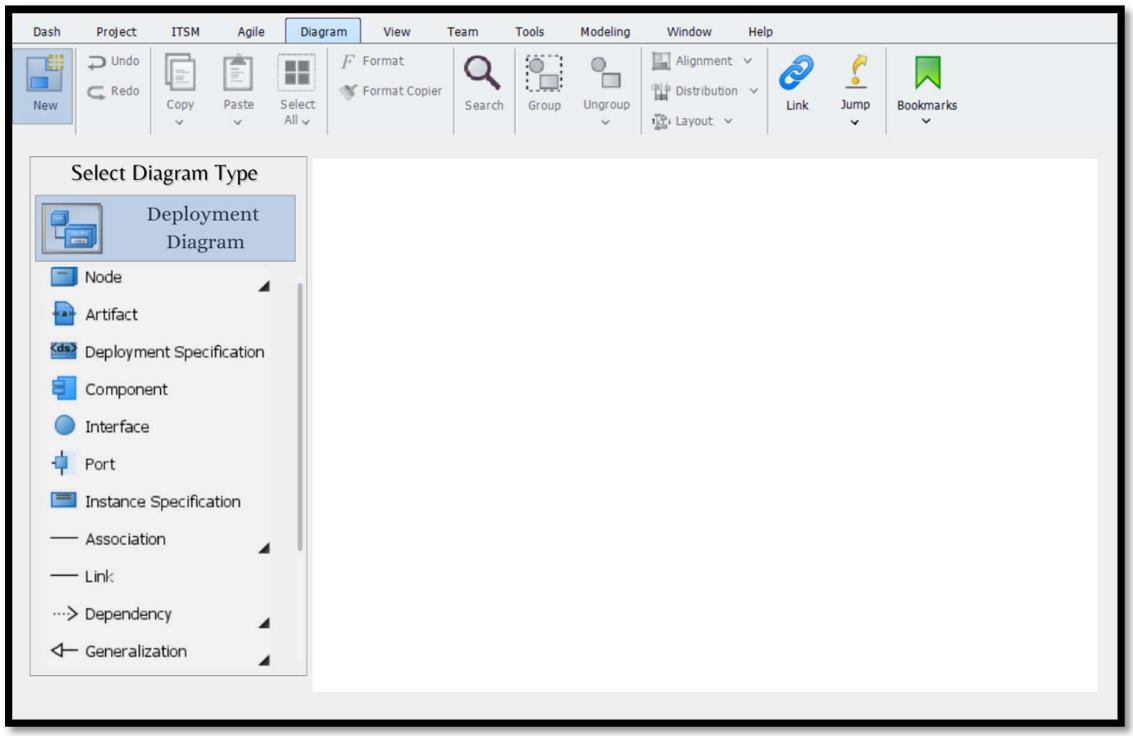
Component Diagram:



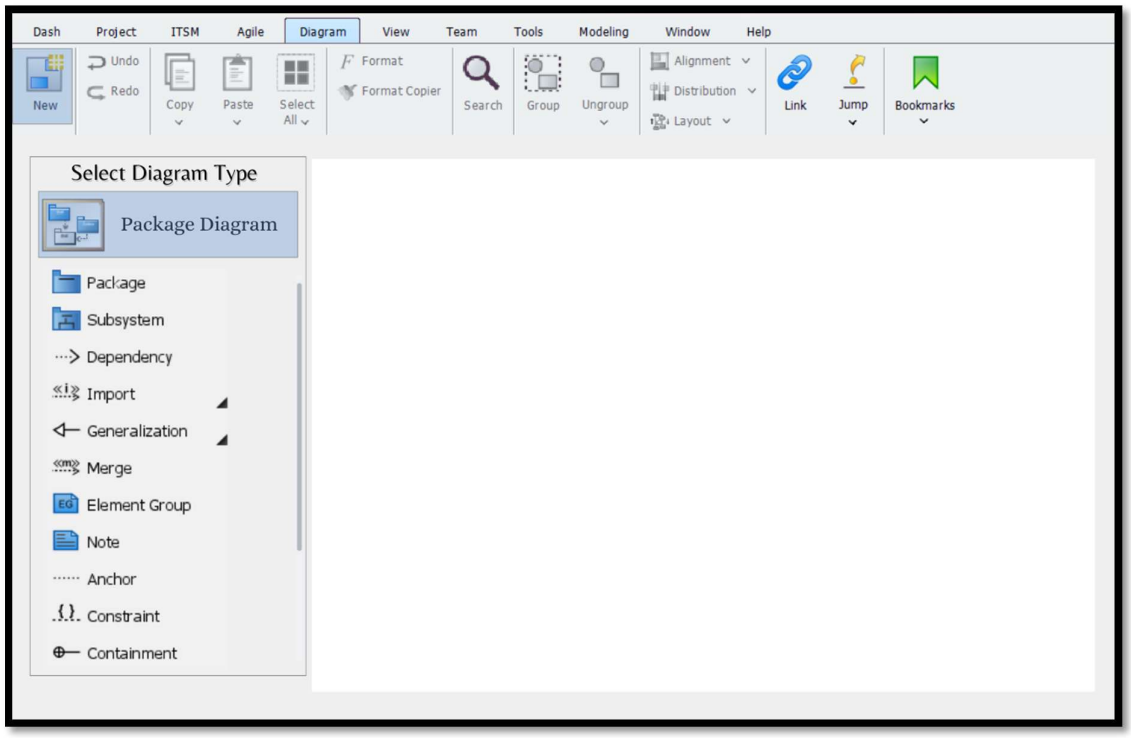
Data Flow Diagram:



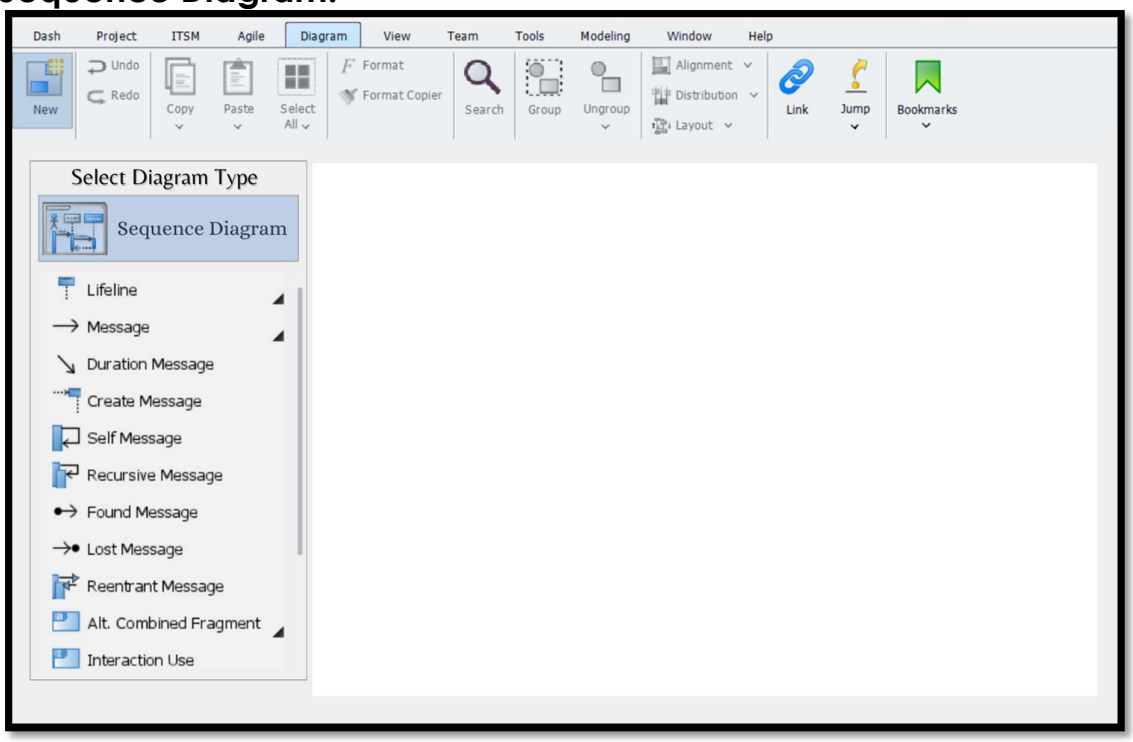
Deployment Diagram:



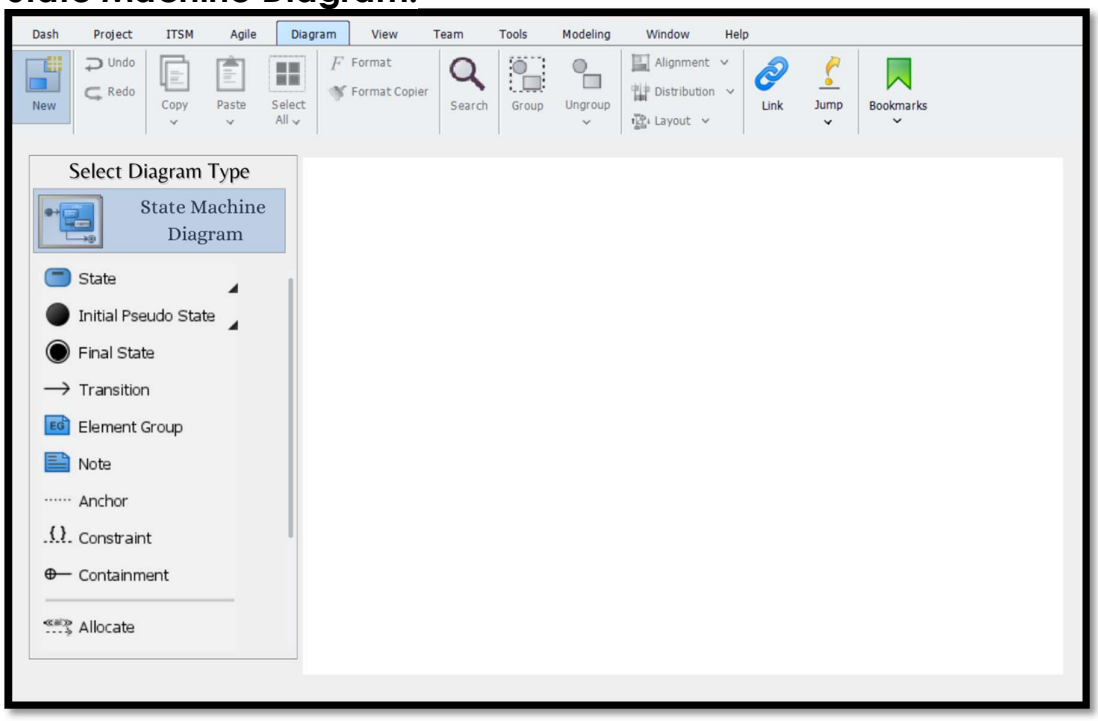
Package Diagram:



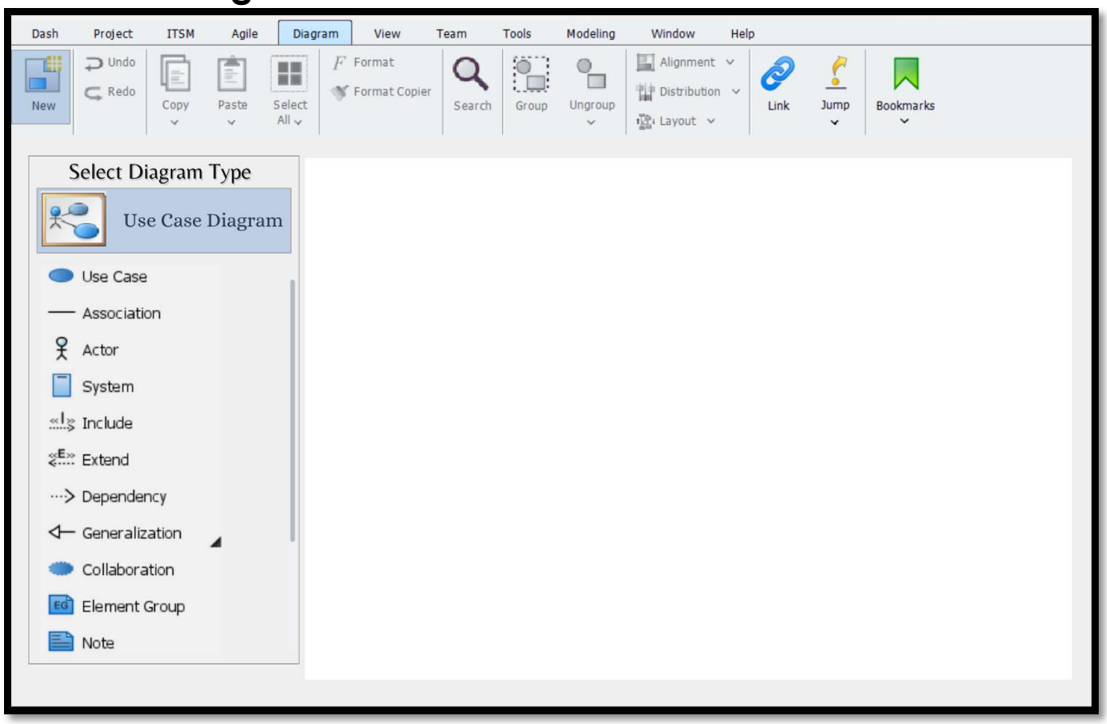
Sequence Diagram:



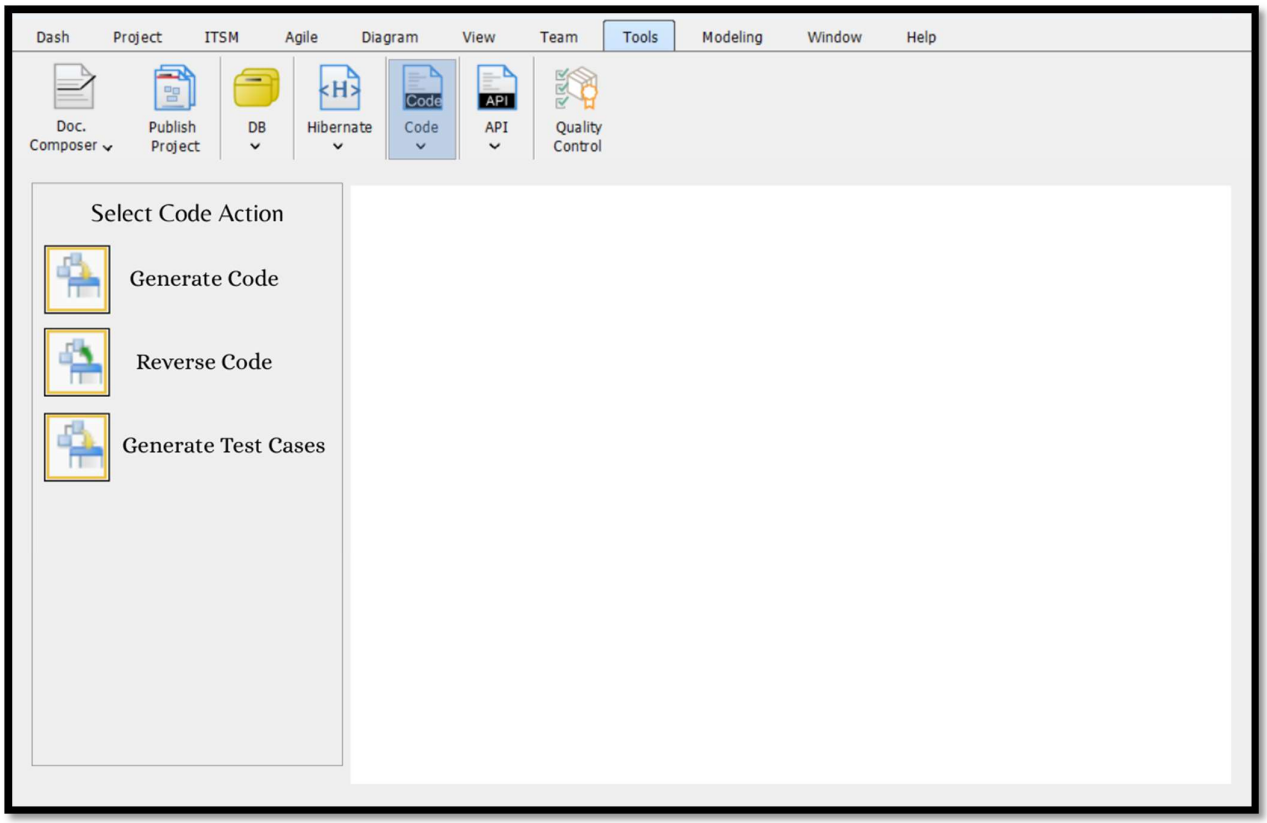
State Machine Diagram:



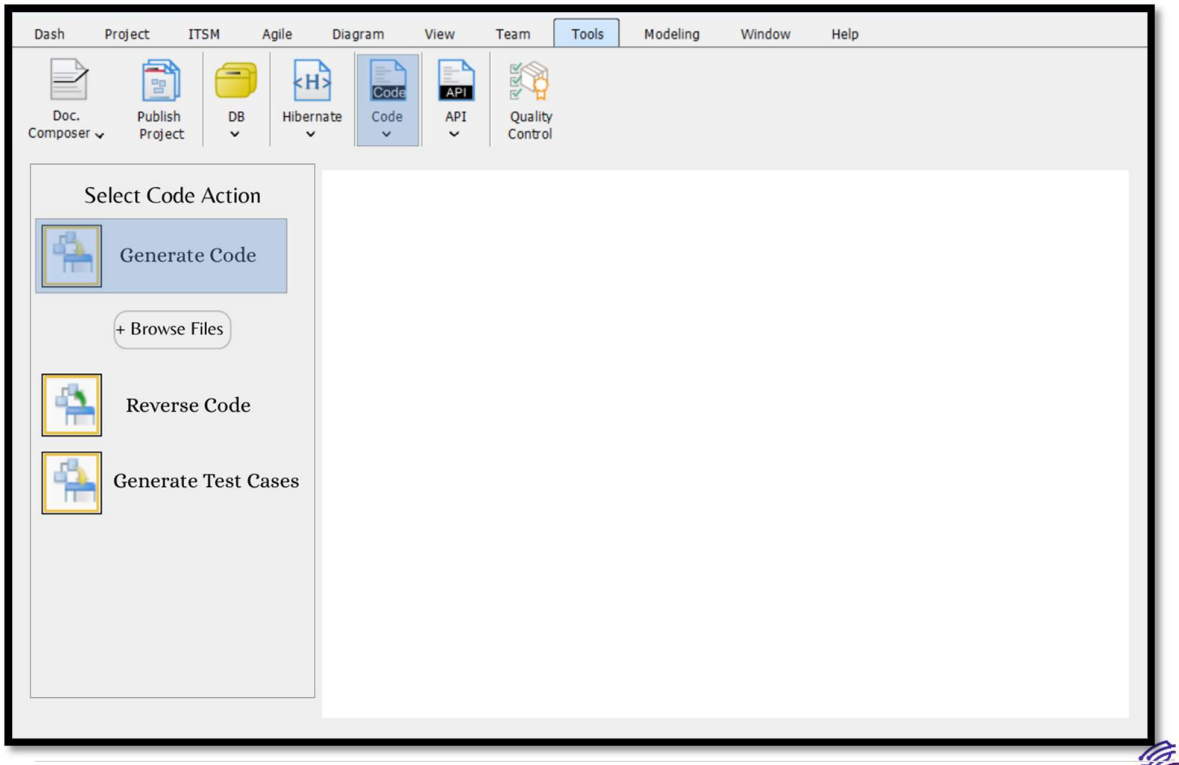
Use Case Diagram:

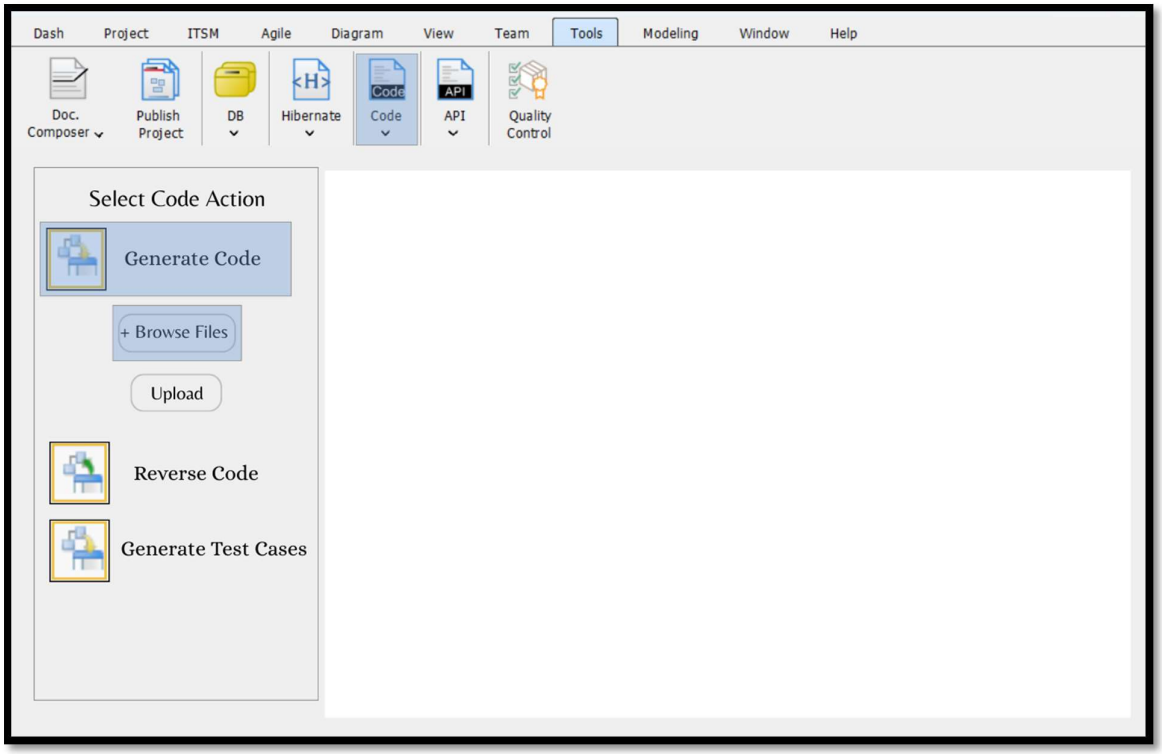


Code Generator:

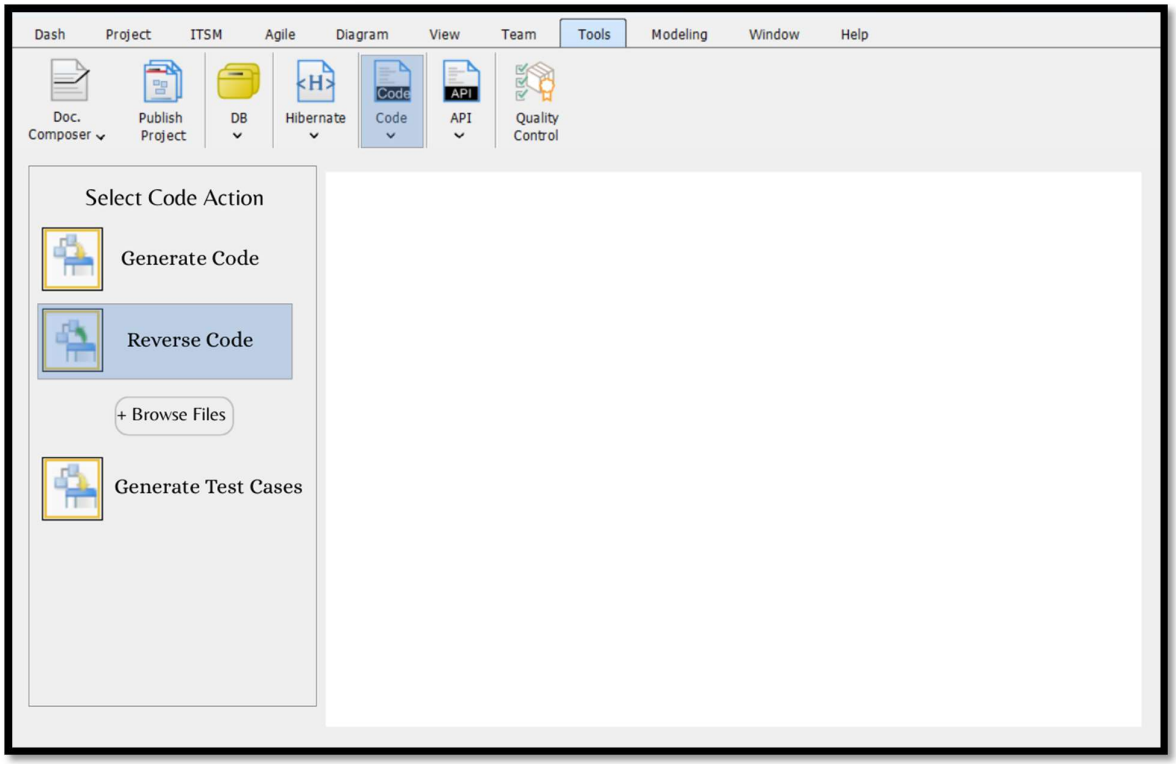


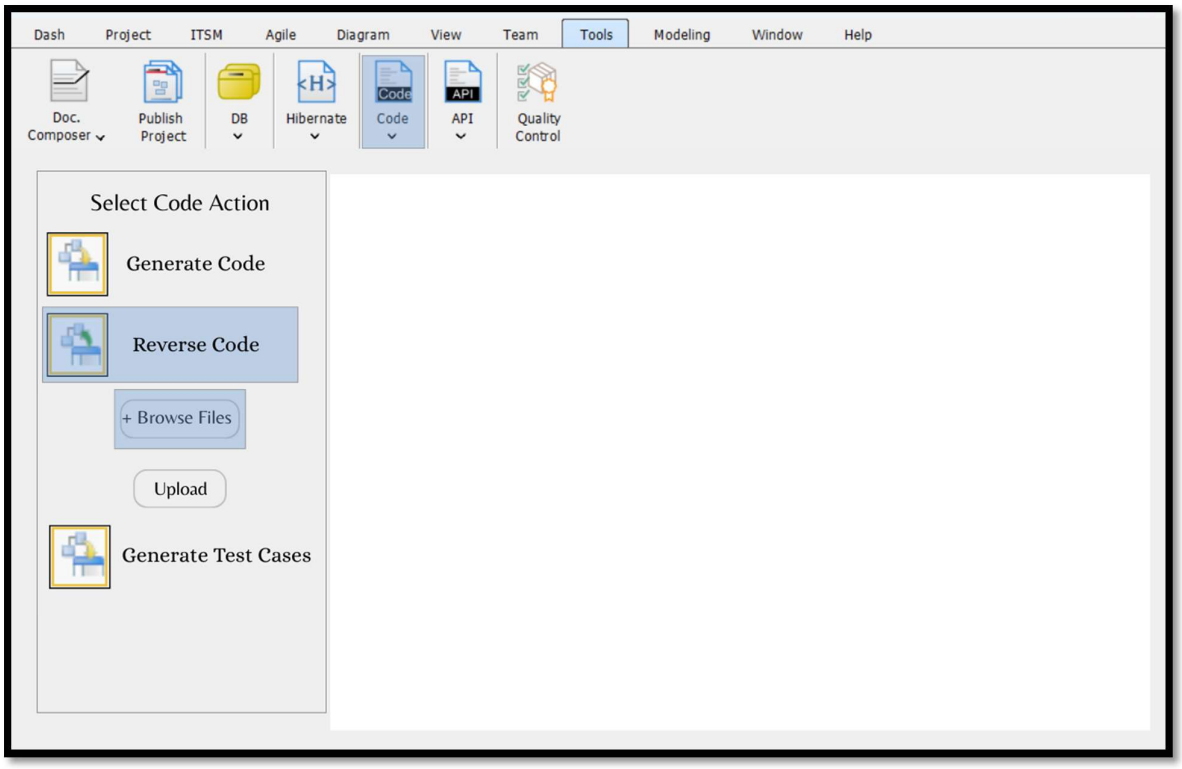
Code generation:



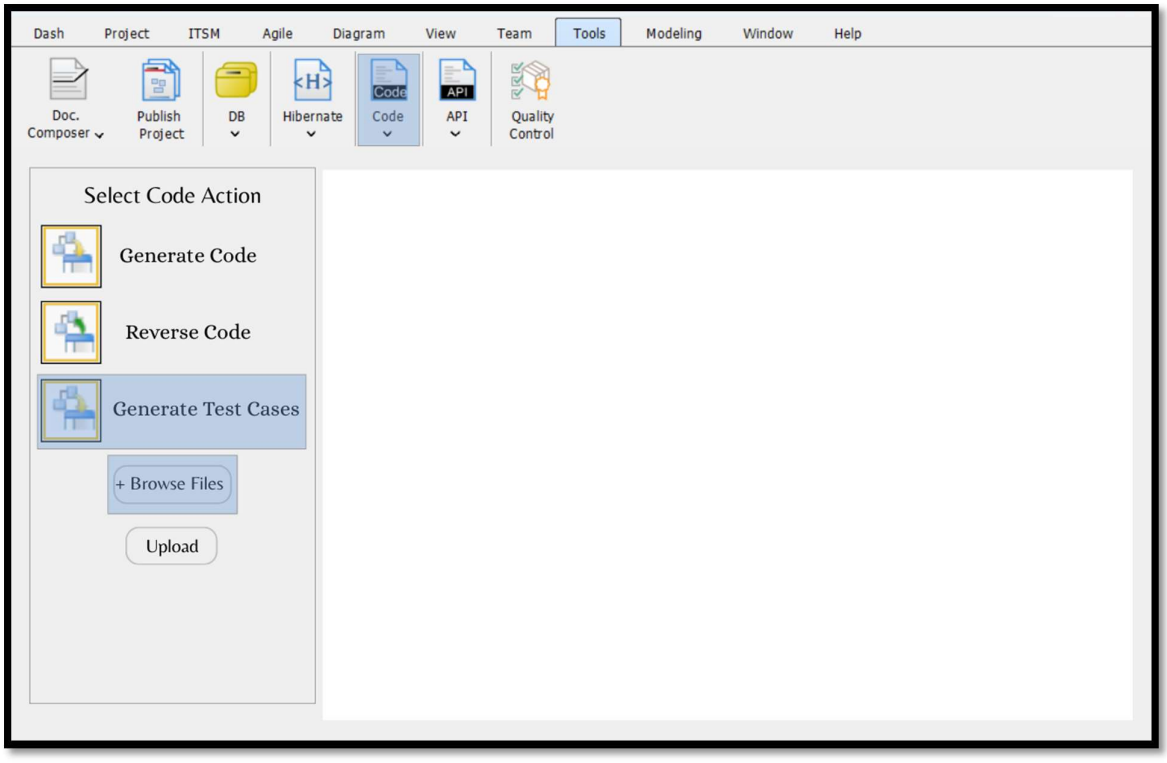


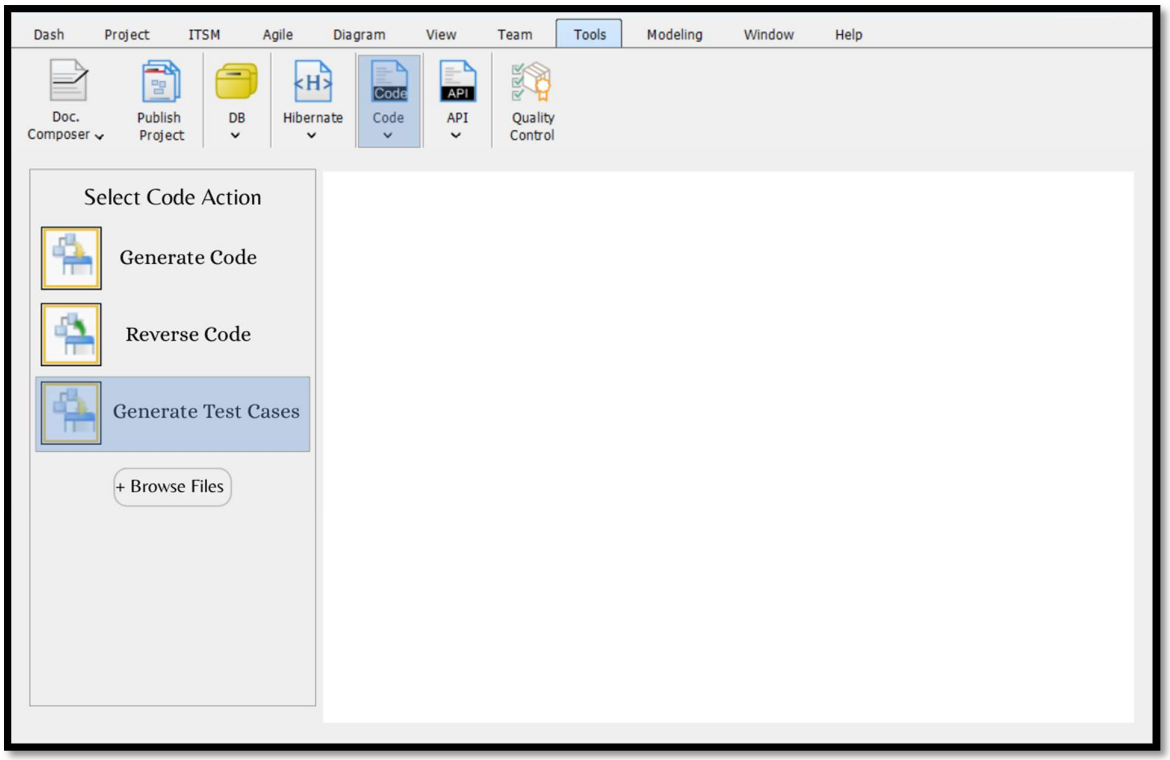
Reverse Generation:

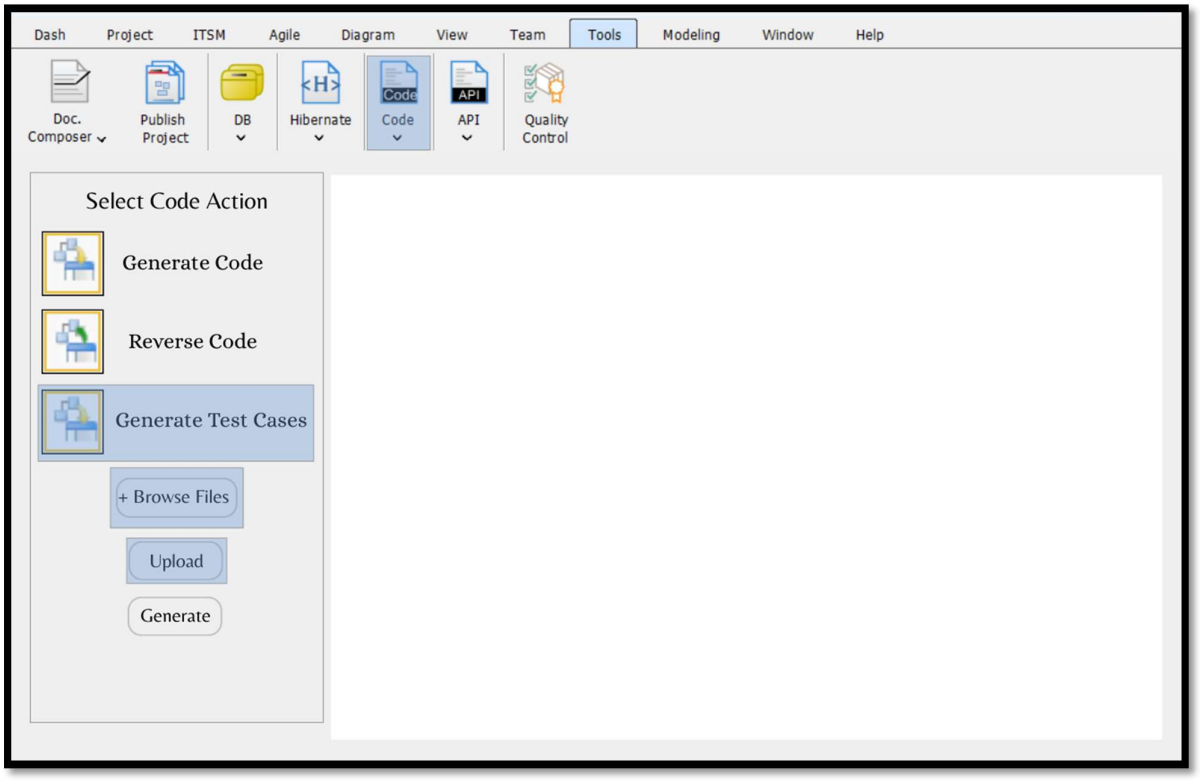



Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Doc. Composer
Publish Project
DB
Hibernate
Code
API
Quality Control
Select Code Action
Generate Code
Reverse Code
+ Browse Files
Upload
Generate Test Cases

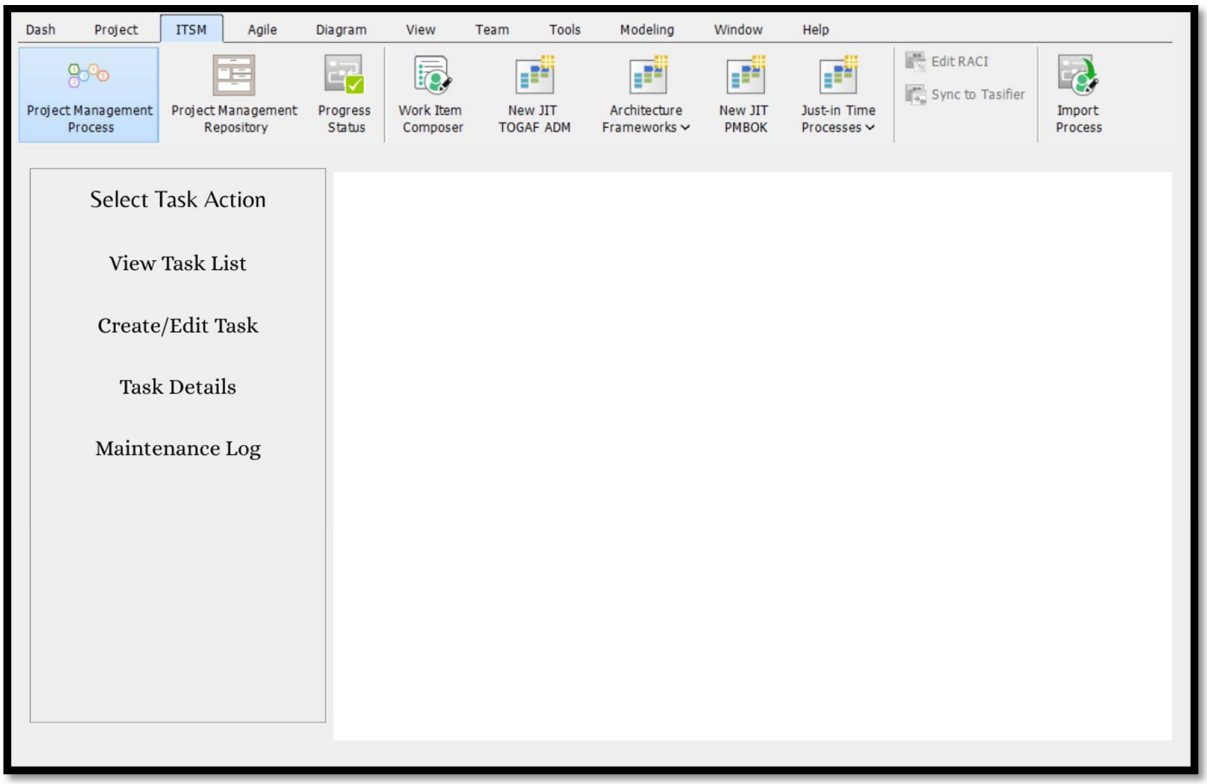
Testing:


Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Doc. Composer
Publish Project
DB
Hibernate
Code
API
Quality Control
Select Code Action
Generate Code
Reverse Code
Generate Test Cases
+ Browse Files
Upload

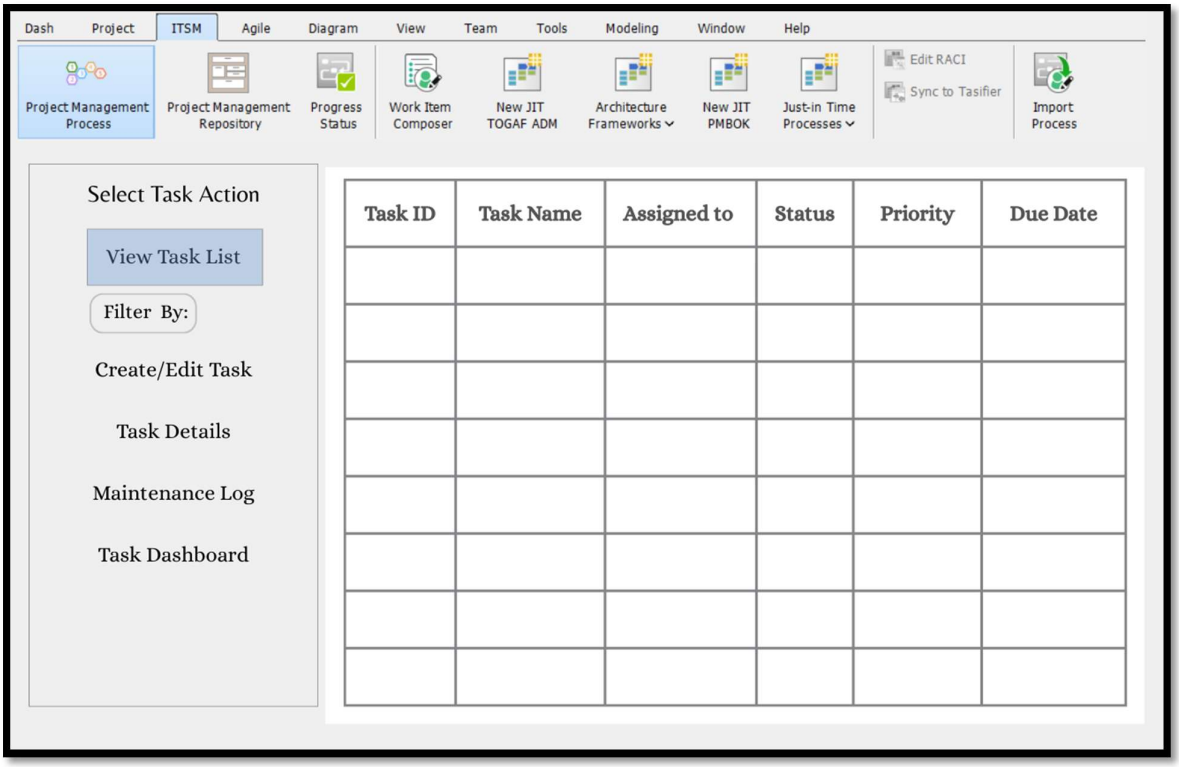

Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Doc. Composer
Publish Project
DB
Hibernate
Code
API
Quality Control
Select Code Action
Generate Code
Reverse Code
Generate Test Cases
+ Browse Files

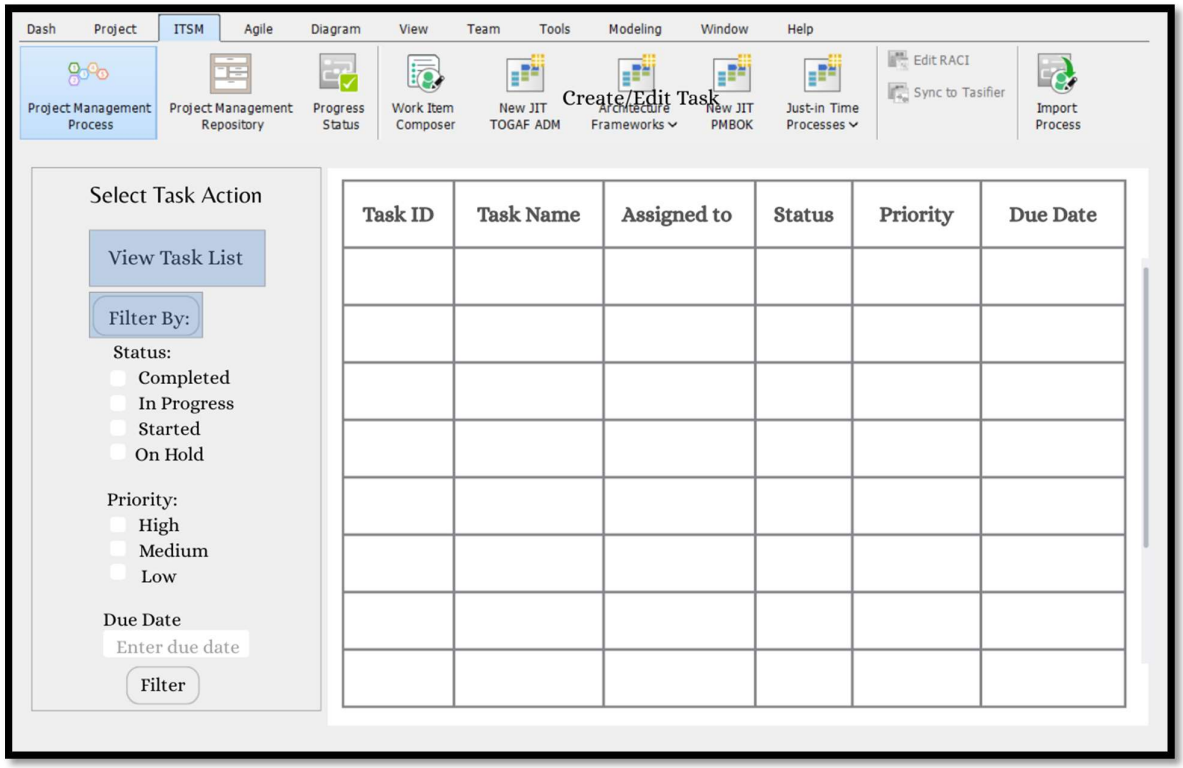

Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Doc. Composer
Publish Project
DB
Hibernate
Code
API
Quality Control
Select Code Action
Generate Code
Reverse Code
Generate Test Cases
+ Browse Files
Upload
Generate

Maintenance:

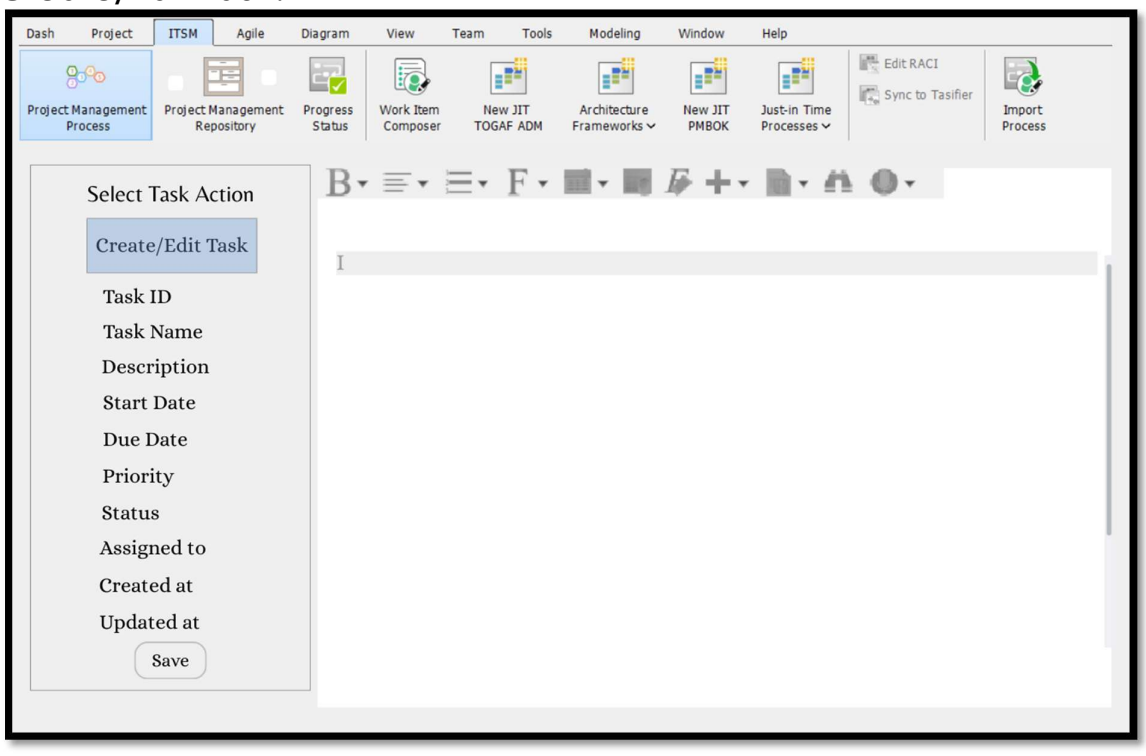


View Task List:

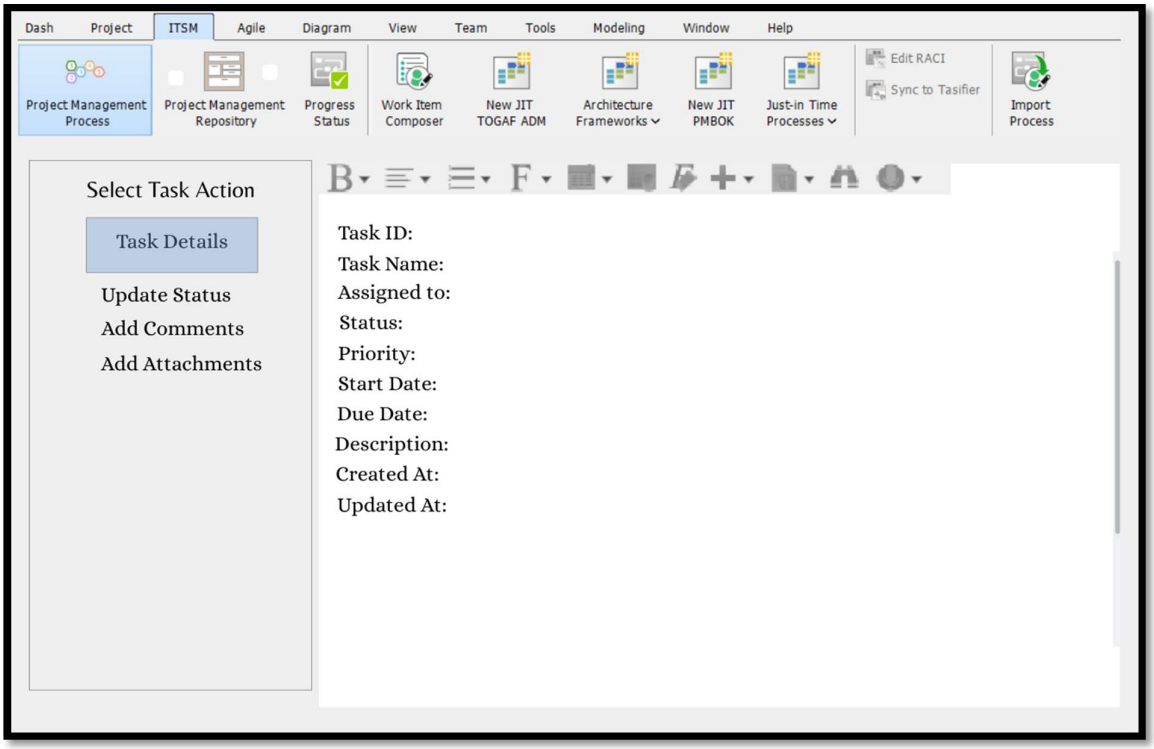


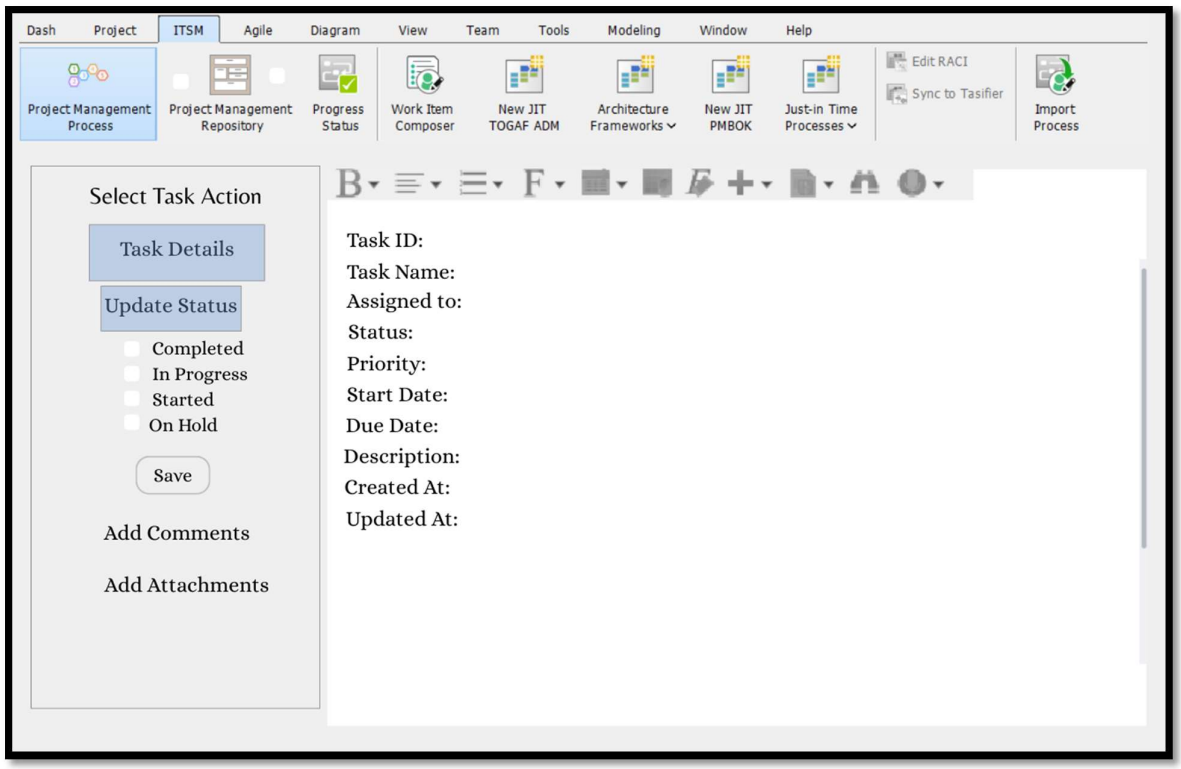


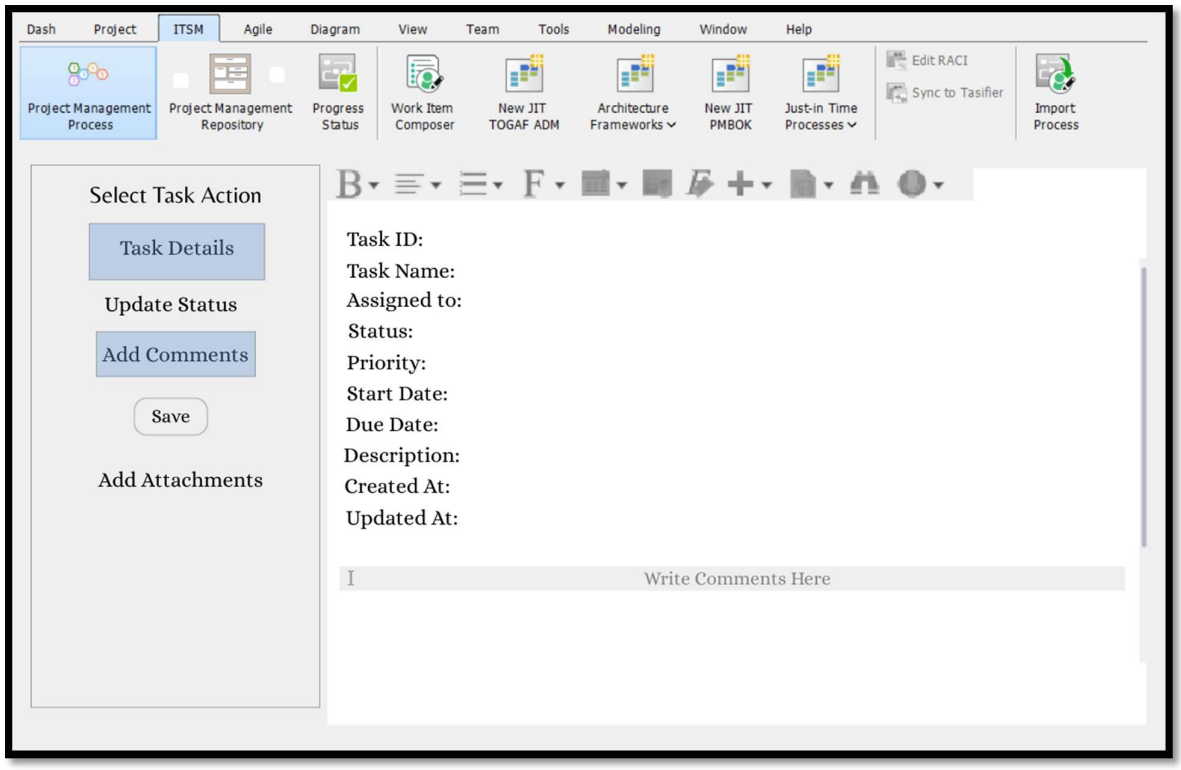
Create/Edit Task:

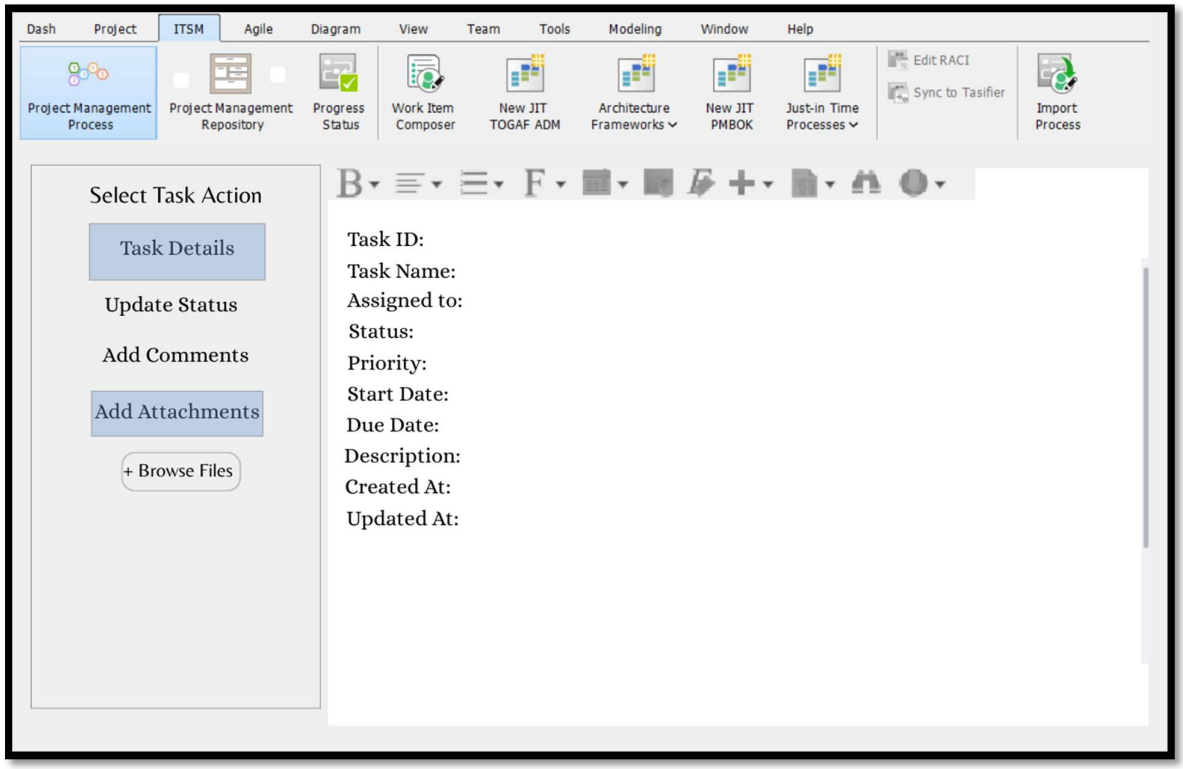


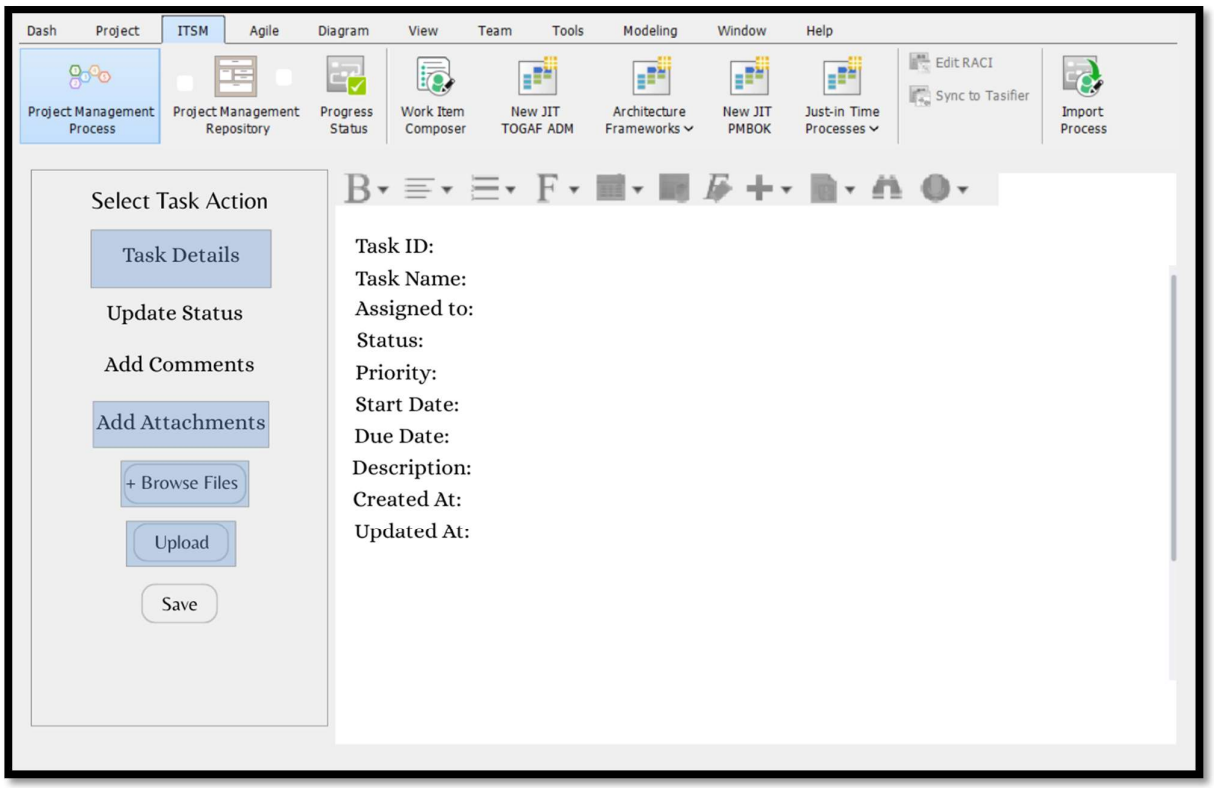
Task Details:



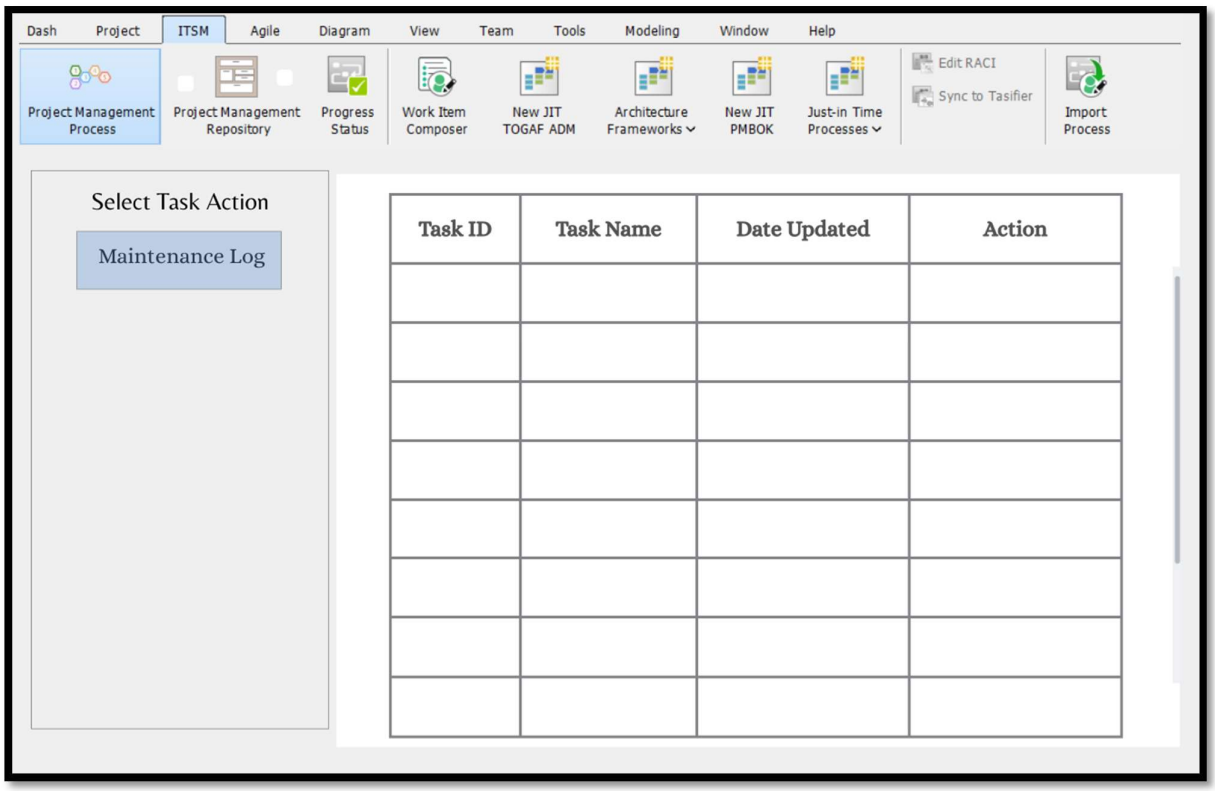



Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Project Management Process
Project Management Repository
Progress Status
Work Item Composer
New JIT TOGAF ADM
Architecture Frameworks
New JIT PMBOK
Just-in Time Processes
Edit RACI
Sync to Tasifier
Import Process
Select Task Action
Task Details
Update Status
Add Comments
Save
Add Attachments
Task ID:
Task Name:
Assigned to:
Status:
Priority:
Start Date:
Due Date:
Description:
Created At:
Updated At:
Write Comments Here


Dash
Project
ITSM
Agile
Diagram
View
Team
Tools
Modeling
Window
Help
Project Management Process
Project Management Repository
Progress Status
Work Item Composer
New JIT TOGAF ADM
Architecture Frameworks
New JIT PMBOK
Just-in Time Processes
Edit RACI
Sync to Tasifier
Import Process
Select Task Action
Task Details
Update Status
Add Comments
Add Attachments
+ Browse Files
Task ID:
Task Name:
Assigned to:
Status:
Priority:
Start Date:
Due Date:
Description:
Created At:
Updated At:



Maintenance Log:



LOW LEVEL DESIGN

DATA STRUCTURE ALGORITHMS

Overview

The Low-Level Design (LLD) defines the internal structure of the Online Design Tool, focusing on class diagrams, data structures, algorithms, and database schemas. It specifies how each module—such as the Requirement Analyzer, Diagramming Tool, and Code Generator—is implemented using design patterns like Singleton and Factory to ensure modularity and maintainability. LLD also outlines how data is stored, accessed, and processed efficiently, providing developers with a clear blueprint for implementation.

◆ Requirement Analyzer (Class Design)

1. Class: StakeholderInputInterface

Description:

The Stakeholder Input Interface class manages the initial requirement input process from stakeholders.

Attributes:

- **Input Fields:** List of required input field names.
- **Input Data:** Map storing user-provided values for each input field.

Methods:

- **Collect Input(data):** Populates input Data from stakeholder-provided values.
- **Validate Input():** Ensures no field is left empty.
- **Submit Input():** Returns a new Requirement object if validation passes.

```
public class Stakeholder Input Interface {
   private List<String> input Fields;
   private Map<String, String> input Data;

   public Stakeholder Input Interface() {
      input Fields = Arrays.as List("title", "description", "priority", "type");
      input Data = new Hash Map<>();
   }
            public void collect Input(Map<String, String> data) {
      for (String field : input Fields) {
         input Data put(field, data.getOrDefault(field, ""));
      }      }
   public boolean validateInput() {
```


```
        return inputFields.stream().allMatch(f -> inputData.get(f) != null &&
!inputData.get(f).isEmpty()); }

    public Requirement submitInput() {
        if (validateInput()) {
            return new Requirement(inputData.get("title"), inputData.get("description"));
        }
        return null;
    }
}
```

Detailed Functionality:

The StakeholderInputInterface class allows stakeholders to input requirement details through a structured form. It collects values for fields like title, description, priority, and type, storing them in a map. The validateInput() method ensures all required fields are filled. Once validated, submitInput() generates a Requirement object to be used in the system.

2. Class: Requirement Categorization and Tagging

Description:

The Requirement class allows for logical grouping and filtering by assigning categories and tags to individual requirements.

Attributes:

- **category:** Classification of the requirement (e.g., functional, non-functional).
- **tags:** List of keywords associated with the requirement.

Methods:

- **setCategory(category):** Assigns a category to the requirement.
- **addTag(tag):** Adds a new tag if it doesn't already exist.
- **getTags():** Returns all assigned tags.

```
public class Requirement {
    private String id;
    private String description;
    private String category;
    private List<String> tags;
    public Requirement(String id, String description) {
        this.id = id;
        this.description = description;
```


```
        this.tags = new ArrayList<>();
    }
    public void setCategory(String category) {
        this.category = category;
    }

    public void addTag(String tag) {
        if (!tags.contains(tag)) {
            tags.add(tag);
        }
    }

    public List<String> getTags() {
        return tags;
    }

    public String getDescription() {
        return description;
    }

    public String getId() {
        return id;
    }
}
```

Detailed Functionality:

The Requirement class enables categorization and tagging of individual requirements for better organization and filtering. Categories such as functional or non-functional can be assigned, and tags can be added to support searching and grouping. Tags are stored in a list and updated only if they are unique. This helps in managing large sets of requirements efficiently.

3. Class: Requirement Validation and Consistency Checking

Description:

The RequirementValidator class validates requirement content and checks for conflicts or duplications between multiple requirements.

Attributes: ***None*** *(utility class)*.

Methods:

- **isValid(req):** Ensures the requirement's description is sufficiently detailed.
- **isConsistent(req1, req2):** Returns false if two requirements have duplicate descriptions.

```
public class RequirementValidator {

   public boolean isValid(Requirement req) {
      return req != null && req.getDescription() != null && req.getDescription().length() > 10;
   }
   public boolean isConsistent(Requirement req1, Requirement req2) {
      return !req1.getDescription().equalsIgnoreCase(req2.getDescription());}
}
```

Detailed Functionality:

The RequirementValidator class checks the integrity and uniqueness of requirement descriptions. The isValid() method ensures that a requirement is not empty and contains meaningful content. The isConsistent() method checks for duplication or conflict by comparing descriptions of two requirements. This avoids redundancy and ensures clarity in specifications.

4. Class: Requirement Traceability Matrix (RTM)

Description:

The Traceability Matrix class creates a mapping between requirements and related system artifacts, such as design components or test cases.

Attributes:

- **matrix:** A map linking requirement IDs to artifact identifiers.

Methods:

- **addMapping(reqId, artifactId):** Adds a traceability link.
- **getMapping(reqId):** Returns the linked artifact ID for a requirement.
- **getFullMatrix():** Returns the entire traceability structure.

```
public class TraceabilityMatrix {
   private Map<String, String> matrix;

   public TraceabilityMatrix() {
      matrix = new HashMap<>();
   }

   public void addMapping(String reqId, String artifactId) {
      matrix.put(reqId, artifactId);
   }
```


```
    public String getMapping(String reqId) {
        return matrix.getOrDefault(reqId, "Not linked");
    }

    public Map<String, String> getFullMatrix() {
        return matrix;
    }
}
```

Detailed Functionality:

The Traceability Matrix class manages links between requirements and their related system artifacts like design elements or test cases. By maintaining a mapping of requirement IDs to associated artifacts, it supports tracking and validation throughout the development lifecycle. The matrix ensures that every requirement is accounted for and can be traced forward and backward.

5. Class: Collaboration and Version Control

Description:

The RequirementVersionControl class enables collaborative editing and tracks all changes made to a requirement by versioning and maintaining a change log.

Attributes:

- **version:** Current version number of the requirement.
- **changeLog:** List of changes with version annotations.

Methods:

- **incrementVersion():** Increases the version number.
- **addChange(changeNote):** Adds a change entry.
- **getChangeLog():** Retrieves all changes.

```
public class RequirementVersionControl {
    private int version;
    private List<String> changeLog;

    public RequirementVersionControl() {
        version = 1;
        changeLog = new ArrayList<>();
    }

    public void incrementVersion() {
        version++;
    }

    public void addChange(String changeNote) {
        changeLog.add("v" + version + ": " + changeNote);
    }

    public List<String> getChangeLog() {
        return changeLog;
    }

public int getVersion() {
        return version;
    }
}
```

Detailed Functionality:

The RequirementVersionControl class tracks updates made to a requirement and maintains a version history. Each time a change is made, the version is incremented and a change log is updated with the new note. This supports collaborative environments by preserving historical edits and enabling rollback or review of previous versions when needed.

6. Class: Export/Import and Integration Capabilities

Description:

The RequirementIO class handles input/output operations for storing and loading requirements in JSON/CSV formats, and supports integration with external tools.

Attributes: ***None*** *(utility class)*.

Methods:

- **exportToJson(reqList):** Serializes requirements to JSON.
- **importFromJson(json):** Loads requirements from a JSON string.
- **exportToCSV(reqList, filePath):** Writes requirements to a CSV file.

```
import com.google.gson.Gson;
import java.io.*;
import java.util.*;

public class RequirementIO {

    public String exportToJson(List<Requirement> reqList) {
        Gson gson = new Gson();
        return gson.toJson(reqList);
    }

    public List<Requirement> importFromJson(String json) {
        Gson gson = new Gson();
        Requirement[] array = gson.fromJson(json, Requirement[].class);
        return Arrays.asList(array);
    }

    public void exportToCSV(List<Requirement> reqList, String



 filePath) throws IOException {
        FileWriter writer = new FileWriter(filePath);
        for (Requirement req : reqList) {
            writer.write(req.getId() + "," + req.getDescription() + "\n");
        }
        writer.close();
    }
}
```

Detailed Functionality:

The RequirementIO class handles the export and import of requirements to and from JSON and CSV formats. It allows integration with external tools and formats for sharing or storing requirements. Using JSON serialization, the class converts requirement objects into string representations and supports reading them back, enabling portability and data exchange.

◆ Requirement Analyzer (Algorithm)

1.Stakeholder Input Interface:

Purpose: To collect and validate requirement input from stakeholders.

Algorithm

- **Input**: Stakeholder input data (title, description, priority, type)
- **Process**:
  - Initialize input fields list.
  - Collect input into a map.
  - Check that all fields are filled.
  - If valid, create a new Requirement object.
- **Output**: Boolean indicating success or failure.

Implementation

```
boolean submitInput(Map<String, String> data) {
   collectInput(data);
   if (validateInput()) {
      Requirement req = new Requirement(data.get("title"), data.get("description"));
      return true;
   }
   return false;
}
```

2. Requirement Categorization and Tagging:

Purpose: To assign a category and tags to a requirement for better filtering and grouping.

Algorithm

- **Input**: Requirement object, category string, tag list
- **Process**:
  - Assign category to the requirement.
  - For each tag in the list, add if not already present.
- **Output**: Boolean indicating tagging success.

Implementation

```
boolean categorizeAndTag(Requirement req, String category, List<String> tags) {
   req.setCategory(category);
   for (String tag : tags) {
      req.addTag(tag);
   }
   return true;
}
```

3. Requirement Validation and Consistency Checking:

Purpose: To ensure requirements are clear and not duplicated.

Algorithm

- **Input**: Two requirement objects
- **Process**:
  - Check if descriptions are valid (non-empty and >10 chars).
  - Compare descriptions for duplication.
- **Output**: Boolean indicating validity and consistency.

Implementation

```
boolean validateAndCheck(Requirement req1, Requirement req2) {
   if (isValid(req1) && isValid(req2)) {
      return isConsistent(req1, req2);
   }
   return false;
}
```

4. Requirement Traceability Matrix (RTM):

Purpose: To map each requirement to related system artifacts.

Algorithm

- **Input**: Requirement ID, Artifact ID
- **Process**:
  - Add mapping to the traceability matrix.
  - If ID exists, update the link.
- **Output**: Boolean indicating mapping success.

Implementation

```
boolean addToRTM(String reqId, String artifactId) {
   matrix.put(reqId, artifactId);
   return matrix.containsKey(reqId);
}
```

5. Collaboration and Version Control:

Purpose: To track requirement edits and maintain version history.

Algorithm

- **Input**: Change description string
- **Process**:
    - Increment version number.
    - Append change description to the log with version tag.
- **Output**: Boolean indicating update success.

Implementation

```
boolean updateRequirement(String changeNote) {
   incrementVersion();
   addChange(changeNote);
   return true;
}
```

6. Export/Import and Integration Capabilities:

Purpose: To export requirements in JSON/CSV and import them for integration.

Algorithm

- **Input**: List of requirements, or JSON string
- **Process**:
    - If exporting, convert list to JSON/CSV format.
    - If importing, parse input and recreate requirement objects.
- **Output**: Boolean indicating success of import/export.

Implementation

```
boolean exportToJson(List<Requirement> reqList) {
   String json = new Gson().toJson(reqList);
   return json != null && !json.isEmpty();
}
boolean importFromJson(String json) {
   Requirement[] reqs = new Gson().fromJson(json, Requirement[].class);
   return reqs.length > 0;
}
```

◆ Requirement Analyzer (Database)

1. Stakeholder Table:

- **stakeholder_id:** Unique ID for each stakeholder (Primary Key).
- **name:** Full name of the stakeholder.
- **role:** Role in the project (e.g., Developer, Analyst).
- **requirement_id:** Unique ID for each requirement.
- **title:** Short title of the requirement.
- **description:** Detailed description of the requirement.
- **category:** Category of the requirement.
- **tags:** Comma-separated tags for filtering.
- **submitted_by:** Foreign key referencing the stakeholder who submitted the requirement.
- **created_at:** Timestamp of when the requirement was submitted.

```sql
-- Create the Stakeholders Table
CREATE TABLE Stakeholders (
    stakeholder_id VARCHAR(5),
    name VARCHAR(255) NOT NULL,
    role VARCHAR(100),
    email VARCHAR(255)
);

INSERT INTO Stakeholders (stakeholder_id, name, role, email) VALUES
('S001', 'Alice Johnson', 'Product Owner', 'alice.johnson@example.com'),
('S002', 'Bob Smith', 'Developer', 'bob.smith@example.com'),
('S003', 'Carol Lee', 'QA Engineer', 'carol.lee@example.com');
-- Create the Requirements Table
CREATE TABLE Requirements (
    requirement_id VARCHAR(5),
    title VARCHAR(255) NOT NULL,
    description TEXT,
    category VARCHAR(100),       -- e.g., Functional, Non-functional
    tags VARCHAR(255),           -- e.g., "UI,Performance,Security"
    submitted_by VARCHAR(5),     -- foreign key referencing stakeholder_id
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
INSERT INTO Requirements (requirement_id, title, description, category, tags, submitted_by) VALUES
('R001', 'User Login', 'System should allow users to log in using email and password.', 'Functional', 'UI,Security', 'S001'),
('R002', 'Page Load Time', 'Main dashboard should load within 2 seconds.', 'Non-functional', 'Performance', 'S002'),
('R003', 'Password Encryption', 'Passwords must be stored using SHA-256 encryption.', 'Non-functional', 'Security', 'S001'),
('R004', 'Drag-and-Drop Support', 'Users should be able to drag and drop elements on canvas.', 'Functional', 'UI,UX', 'S002'),
('R005', 'Bug Reporting Tool', 'Enable users to report bugs directly from the interface.', 'Functional', 'UI,Feedback', 'S003');
```

```sql
-- Select joined data with stakeholder names
SELECT
    r.requirement_id,
    r.title,
    r.description,
    r.category,
    r.tags,
    s.name AS submitted_by,
    r.created_at
FROM
    Requirements r
JOIN
    Stakeholders s ON r.submitted_by = s.stakeholder_id;
```

requireme…	title	description	category	tags	submitted_by	created_at
R001	User Login	System should all…	Functional	UI,Security	Alice Johnson	2025-04-12 20:15:27
R002	Page Load Time	Main dashboard s…	Non-functional	Performance	Bob Smith	2025-04-12 20:15:27
R003	Password Encryption	Passwords must b…	Non-functional	Security	Alice Johnson	2025-04-12 20:15:27
R004	Drag-and-Drop Support	Users should be a…	Functional	UI,UX	Bob Smith	2025-04-12 20:15:27
R005	Bug Reporting Tool	Enable users to re…	Functional	UI,Feedback	Carol Lee	2025-04-12 20:15:27

2. Collaboration and Version Control

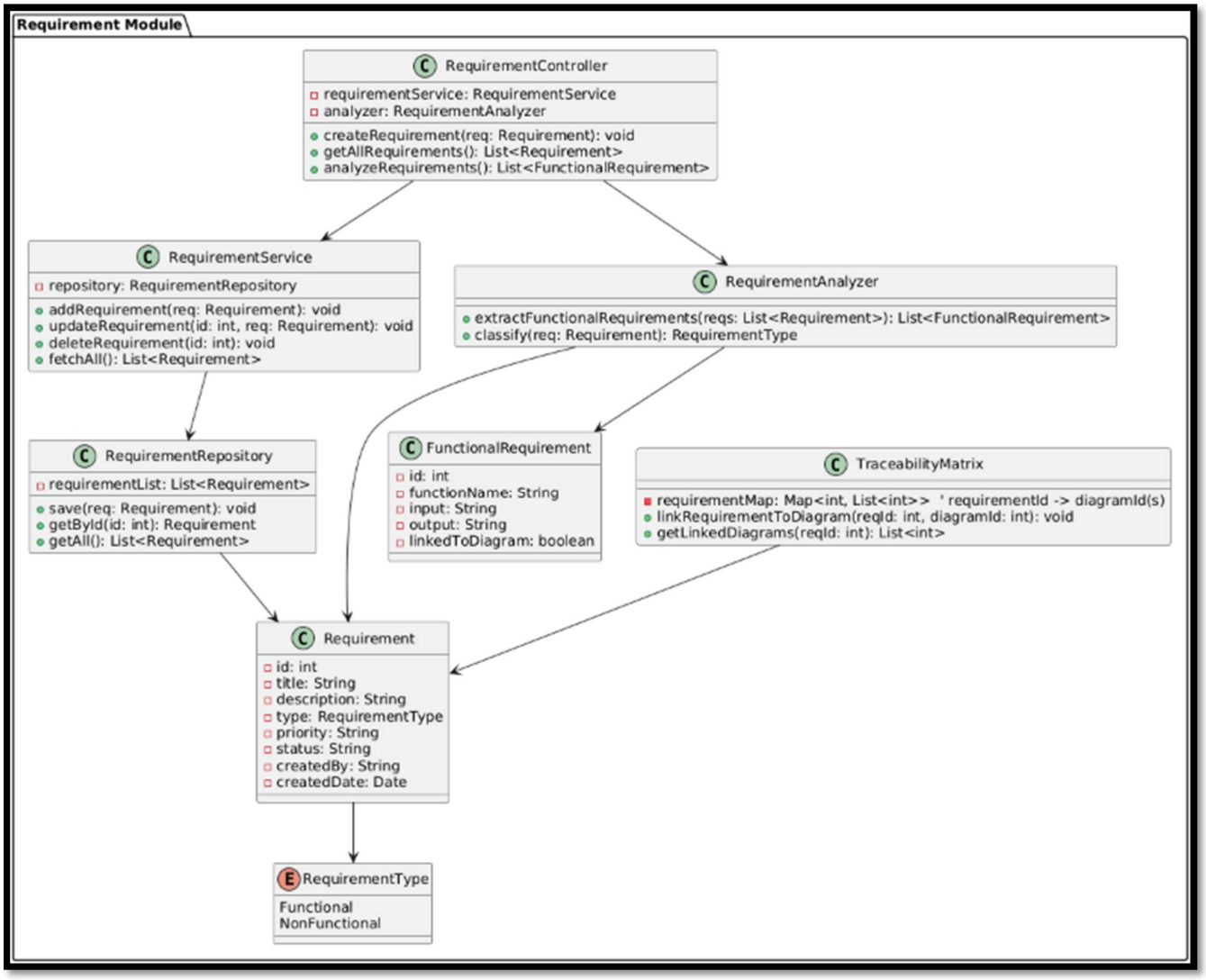
- **version_id:** Unique ID for this version record.
- **requirement_id:** Requirement this version belongs to (FK from Requirements).
- **version_number:** Version number.
- **title:** Title at this version.
- **description**: Description content at this version.
- **modified_by:** Stakeholder who made the modification (FK from Stakeholders).
- **modified_at**: When this version was saved.
- **collaborator_id:** Unique ID for this collaboration record.
- **requirement_id:** Linked requirement (FK from Requirements).
- **stakeholder_id:** Collaborating stakeholder (FK from Stakeholders).
- **role:** Role in collaboration (e.g., Editor, Reviewer).
- **assigned_at**: When the stakeholder was assigned as a collaborator.

```sql
CREATE TABLE RequirementVersions (
    version_id VARCHAR(5) PRIMARY KEY,
    requirement_id VARCHAR(5) NOT NULL,
    version_number INT NOT NULL,
    title VARCHAR(255),
    description TEXT,
    modified_by VARCHAR(5), -- stakeholder_id
    modified_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (requirement_id) REFERENCES Requirements(requirement_id),
    FOREIGN KEY (modified_by) REFERENCES Stakeholders(stakeholder_id)
);

-- Insert sample data into RequirementVersions
INSERT INTO RequirementVersions (version_id, requirement_id, version_number, title, description, modified_by) VALUES
('V001', 'R001', 1, 'User Login', 'Initial version of login feature', 'S001'),
('V002', 'R001', 2, 'User Login Updated', 'Added 2FA description', 'S003'),
('V003', 'R002', 1, 'Page Load Time', 'Initial requirement for performance', 'S002');

-- Table 2: Collaborators
CREATE TABLE Collaborators (
    collaborator_id VARCHAR(5) PRIMARY KEY,
    requirement_id VARCHAR(5) NOT NULL,
    stakeholder_id VARCHAR(5) NOT NULL,
    role VARCHAR(100), -- e.g., Editor, Reviewer
    assigned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (requirement_id) REFERENCES Requirements(requirement_id),
    FOREIGN KEY (stakeholder_id) REFERENCES Stakeholders(stakeholder_id)
);
```

```sql
-- Insert sample data into Collaborators
INSERT INTO Collaborators (collaborator_id, requirement_id, stakeholder_id, role) VALUES
('C001', 'R001', 'S002', 'Editor'),
('C002', 'R001', 'S003', 'Reviewer'),
('C003', 'R002', 'S001', 'Reviewer');
SELECT
    c.collaborator_id,
    s.name AS stakeholder_name,
    r.title AS requirement_title,
    c.role,
    c.assigned_at
FROM
    Collaborators c
JOIN Stakeholders s ON c.stakeholder_id = s.stakeholder_id
JOIN Requirements r ON c.requirement_id = r.requirement_id;
```

collaborator_id	requirement_title	role	assigned_at
C001	User Login	Editor	2025-04-12 20:22:59
C002	User Login	Reviewer	2025-04-12 20:22:59
C003	Page Load Time	Reviewer	2025-04-12 20:22:59

◆ Requirement Analyzer (Diagram)



◆ Diagramming Tool (Class Design)

1. Diagram Class

Description:

The Diagram class represents a diagram within a project.

Attributes:

- **diagram_id:** Unique identifier for the diagram.
- **diagram_name:** Name of the diagram.
- **project_id:** Identifier for the project to which the diagram belongs.
- **created_at:** Timestamp when the diagram was created.
- **updated_at:** Timestamp when the diagram was last updated.
- **elements:** List of elements within the diagram.

Methods:

- **add_element(element):** Adds a new element to the diagram.
- **remove_element(element_id):** Removes an element from the diagram by its ID

Implementation

```
class Diagram {
    int diagram_id;
    String diagram_name;
    int project_id;
    Instant created_at;
    Instant updated_at;
    List<Element> elements;
  Diagram(int id, String name, int proj_id) {
            diagram_id = id;
            diagram_name = name;
```


```
        project_id = proj_id;
        created_at = Instant.now();
        elements = new ArrayList<>();   }
class Diagram {
   int diagram_id;
   int project_id;
   Instant created_at;
   Instant updated_at;
   List<Element> elements;
  Diagram(int id, String name, int proj_id) {
        diagram_id = id;
        diagram_name = name;
        project_id = proj_id;
        created_at = Instant.now();
        updated_at = created_at;
        elements = new ArrayList<>();   }
```

Detailed Functionality

The Diagram class stores information such as diagram ID, name, project ID to which it belongs, creation and modification timestamps, and a list of elements contained within the diagram. Users can add and remove elements from the diagram using the addElement and removeElement methods, respectively.

2. Element Class

Description:

The Element class represents an individual element within a diagram.

Attributes:

- **element_id:** Unique identifier for the element.
- **diagram_id:** Identifier for the diagram to which the element belongs.
- **element_type:** Type of the element (e.g., class, interface, component).
- **element_properties:** Properties of the element stored as a JSON object.
- **created_at**: Timestamp when the element was created.
- **updated_at:** Timestamp when the element was last updated.

Methods:

- **element_properties:** Properties of the element stored as a JSON object.
- **update_properties(new_properties):** Updates the properties of the element.

Implementation

```
import java.time.Instant;

import java.util.Map;

class Element {

   int element_id;

   int diagram_id;

   String element_type;

   Map<String, String> element_properties;

   Instant created_at;

   Instant updated_at;

   Element(int element_id, int diagram_id, String element_type, Map<String,


 String> element_properties, Instant created_at, Instant updated_at) {

         this.element_id = element_id;

         this.diagram_id = diagram_id;

         this.element_type = element_type;

         this.element_properties = element_properties;

         this.created_at = created_at;
```


```
        this.updated_at = updated_at;
    }
    void updateProperties(Map<String, String> newProperties) {
        this.element_properties = newProperties;
        this.updated_at = Instant.now();
    }
}
```

Detailed Functionality

The Element class maintains attributes like element ID, diagram ID to which it belongs, element type, element properties stored as a JSON object, and creation and modification timestamps. The class allows for updating element properties via the updateProperties method.

◆ Diagramming Tool (Algorithms)

1. Diagram Creation

Purpose: To create a new diagram within a project.

Algorithm

- **Input:** Diagram name, project ID.
- **Process:**
  Generate a unique diagram ID.Set the current timestamp as the creation time.
  Insert a new diagram record into the Diagrams table.
- **Output:** Diagram ID of the newly created diagram.

Implementation

```
int createDiagram(String name, int projectId) {
    int diagramId = generateUniqueId();
```


```
    Instant now = Instant.now();

    insertIntoDiagramsTable(diagramId, name, projectId, now, now);

    return diagramId;     }
```

2. Diagram Retrieval Algorithms

Purpose: To fetch details of a specific diagram.

Algorithm

- **Input:** Diagram ID.
- **Process:**
  Query the Diagrams table for the given diagram ID.
- **Output:** Diagram details.

Implementation

```
iDiagram getDiagram(int diagramId) {

    return queryDiagramsTableById(diagramId);     }
```

3. Diagram Deletion

Purpose: To remove a diagram and its associated elements.

Algorithm

- **Input:** Diagram ID.
- **Process:**
  Delete all elements associated with the diagram.
  Delete the diagram record from the Diagrams table.
- **Output:** Boolean indicating success or failure.

Implementation

```
boolean deleteDiagram(int diagramId) {

    deleteElementsByDiagramId(diagramId);

    return deleteFromDiagramsTable(diagramId);

}
```


3. Element Management Algorithm

a. Element Addition

Purpose: To add a new element to a diagram.

Algorithm

- **Input:**Element type, properties, diagram ID.
- **Process:**
  Generate a unique element ID.
  Set the current timestamp as the creation time.
- **Output:** Boolean indicating success or failure.

Implementation

```
boolean deleteDiagram(int diagramId) {

   deleteElementsByDiagramId(diagramId);

   return deleteFromDiagramsTable(diagramId);    }
```

b. Element Retrieval

Purpose: To add a new element to a diagram.

Algorithm

- **Input:**Element type, properties, diagram ID.
- **Process:**
  Generate a unique element ID.
  Set the current timestamp as the creation time.
- **Output:** Boolean indicating success or failure.

Implementation

```
boolean deleteDiagram(int diagramId) {

   deleteElementsByDiagramId(diagramId);

   return deleteFromDiagramsTable(diagramId);

}
```

c. Element Deletion

Purpose: To add a new element to a diagram.

Algorithm

- **Input:** Element type, properties, diagram ID.
- **Process:**
  Generate a unique element ID.
  Set the current timestamp as the creation time.
- **Output:** Boolean indicating success or failure.

Implementation

```
boolean deleteDiagram(int diagramId) {

   deleteElementsByDiagramId(diagramId);

   return deleteFromDiagramsTable(diagramId);

}
```

4. Comment Management Algorithms

b. Adding comments

Purpose: To add a new comment to a diagram element.

Algorithm

- **Input:** Element ID, User ID, Content.
- **Process:**
  Generate a unique comment ID.
  Set the current timestamp as the creation time.
  Insert a new comment record into the Comments table.
- **Output:** Comment ID of the newly added comment.

Implementation

```
int addComment(int elementId, int userId, String content) {

   int commentId = generateUniqueId();

   Instant now = Instant.now();

   insertIntoCommentsTable(commentId, elementId, userId, content, now, now);

   return commentId;   }
```

c. Retrieving comments

Purpose: To fetch comments for a specific diagram element.

Algorithm

- **Input:** Element ID
- **Process:**
  To fetch comments for a specific diagram element.
- **Output:** List of comments..

Implementation

```
List<Comment> getComments(int elementId) {

   return queryCommentsTableByElementId(elementId);

}
```

d. Deleting Comments

Purpose: To remove a comment from the diagram element.

Algorithm:

- **Input:** Comment ID
- **Process:**
  Delete the comment record from the Comments table.
- **Output:** Boolean indicating success or failure.

Implementation:

```
boolean deleteComment(int commentId) {

   return deleteFromCommentsTable(commentId);

}
```


◆ Diagramming Tool (Database)

1. Diagrams Table:

This table holds the diagrams that are part of a project. Each project can have multiple diagrams. The associated attributes are:

- **diagram_id:** A unique identifier for each diagram, set as the primary key.
- **name:** The name of the diagram.
- **project_id:** The project_id of the project to which the diagram belongs, creating a relationship with the Projects table (foreign key).
- **created_at:** The time when the diagram was created.
- **updated_at:** The time when the diagram was last updated.

```
32  CREATE TABLE Diagrams (
33      diagram_id INT PRIMARY KEY,
34      name VARCHAR(100) NOT NULL,
35      project_id INTEGER NOT NULL,
36      created_at DATETIME,
37      updated_at DATETIME,
38      FOREIGN KEY (project_id) REFERENCES Projects(project_id)
39  );
40  INSERT INTO Diagrams (diagram_id, name, project_id, created_at, updated_at) VALUES
41  ('1', 'Diagram A', '1', '2024-05-01 10:45:00', '2024-05-01 10:45:00'),
42  ('2', 'Diagram B', '2', '2024-05-02 11:45:00', '2024-05-02 11:45:00'),
43  ('3', 'Diagram C', '3', '2024-05-03 12:45:00', '2024-05-03 12:45:00'),
44  ('4', 'Diagram D', '4', '2024-05-04 13:45:00', '2024-05-04 13:45:00'),
45  ('5', 'Diagram E', '5', '2024-05-05 14:45:00', '2024-05-05 14:45:00');
46  SELECT* FROM Diagrams;
47
```

Results | Messages

	diagram_id	name	project_id	created_at	updated_at
1	1	Diagram A	1	2024-05-01 10:45:00.000	2024-05-01 10:45:00.000
2	2	Diagram B	2	2024-05-02 11:45:00.000	2024-05-02 11:45:00.000
3	3	Diagram C	3	2024-05-03 12:45:00.000	2024-05-03 12:45:00.000
4	4	Diagram D	4	2024-05-04 13:45:00.000	2024-05-04 13:45:00.000
5	5	Diagram E	5	2024-05-05 14:45:00.000	2024-05-05 14:45:00.000

2. DiagramElements Table:

This table contains elements within a diagram. These can be various types of elements like classes, interfaces, entities, or relationships. The associated attributes are:

- **element_id:** A unique identifier for each diagram element, set as the primary key.
- **diagram_id:** The diagram_id of the diagram to which the element belongs, establishing a relationship with the Diagrams table (foreign key).
- **type:** The type of the element (e.g., class, interface, entity, relationship).
- **properties:** A text field to store element-specific properties, allowing for flexibility in the properties stored.
- **position_x:** The x-coordinate of the element's position in the diagram.
- **position_y:** The y-coordinate of the element's position in the diagram.
- **created_at:** The time when the element was created.
- **updated_at:** The time when the element was last updated.

```sql
CREATE TABLE DiagramElements (
    element_id INT PRIMARY KEY,
    diagram_id INTEGER NOT NULL,
    type VARCHAR(50) NOT NULL,
    properties TEXT,
    position_x FLOAT,
    position_y FLOAT,
    created_at DATETIME,
    updated_at DATETIME,
    FOREIGN KEY (diagram_id) REFERENCES Diagrams(diagram_id)
);
INSERT INTO DiagramElements (element_id, diagram_id, type, properties, position_x, position_y, created_at, updated_at) VALUES
('1', '1', 'Class', '{"name": "Class1"}', '100', '100', '2024-05-01 11:00:00', '2024-05-01 11:00:00'),
('2', '1', 'Class', '{"name": "Class2"}', '200', '200', '2024-05-01 11:10:00', '2024-05-01 11:10:00'),
('3', '2', 'Interface', '{"name": "Interface1"}', '150', '150', '2024-05-02 12:00:00', '2024-05-02 12:00:00'),
('4', '3', 'Entity', '{"name": "Entity1"}', '250', '250', '2024-05-03 13:00:00', '2024-05-03 13:00:00'),
('5', '4', 'Relationship', '{"name": "Relationship1"}', '300', '300', '2024-05-04 14:00:00', '2024-05-04 14:00:00');
SELECT* FROM DiagramElements;
```

Results Messages

	element_id	diagram_id	type	properties	position_x	position_y	created_at	updated_at
1	1	1	Class	{"name": "Class1"}	100	100	2024-05-01 11:00:00.000	2024-05-01 11:00:00.000
2	2	1	Class	{"name": "Class2"}	200	200	2024-05-01 11:10:00.000	2024-05-01 11:10:00.000
3	3	2	Interface	{"name": "Interface1"}	150	150	2024-05-02 12:00:00.000	2024-05-02 12:00:00.000
4	4	3	Entity	{"name": "Entity1"}	250	250	2024-05-03 13:00:00.000	2024-05-03 13:00:00.000
5	5	4	Relationship	{"name": "Relationship1"}	300	300	2024-05-04 14:00:00.000	2024-05-04 14:00:00.000

3. Comments Table:

This table holds comments made on diagram elements, facilitating collaboration among users. The associated attributes are:

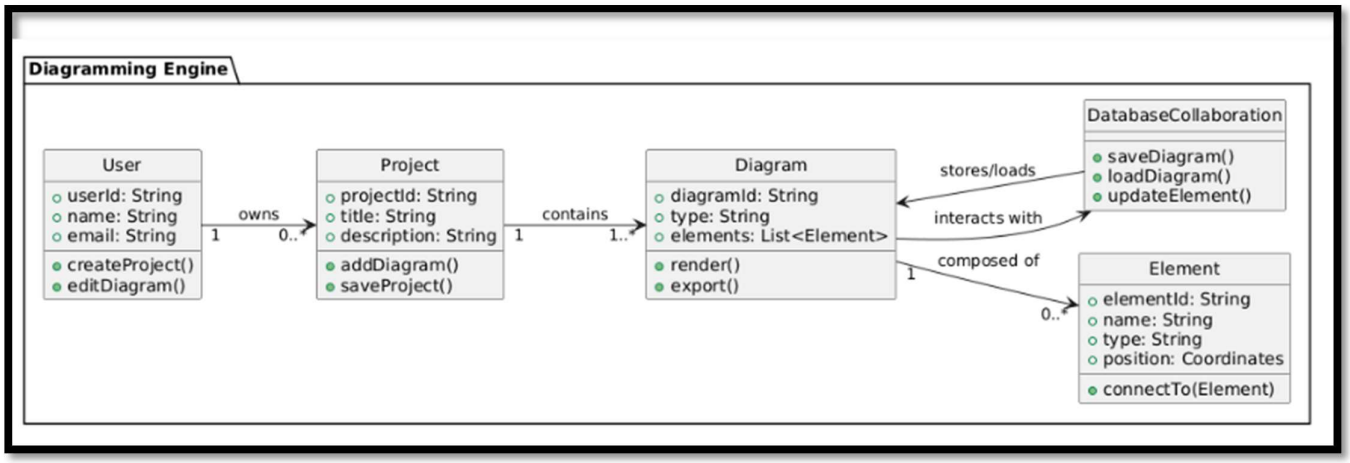
- **comment_id:** A unique identifier for each comment, set as the primary key.
- **element_id:** The element_id of the diagram element to which the comment belongs, creating a relationship with the DiagramElements table (foreign key).
- **user_id:** The user_id of the user who commented, establishing a relationship with the Users table (foreign key).
- **content:** The text content of the comment.
- **created_at:** The timestamp when the comment was created.
- **updated_at:** The timestamp when the comment was last updated.

```
65  CREATE TABLE Comments (
66      comment_id INT PRIMARY KEY,
67      element_id INTEGER NOT NULL,
68      user_id INTEGER NOT NULL,
69      content TEXT NOT NULL,
70      created_at DATETIME,
71      updated_at DATETIME,
72      FOREIGN KEY (element_id) REFERENCES DiagramElements(element_id),
73      FOREIGN KEY (user_id) REFERENCES Users(user_id)
74  );
75  INSERT INTO Comments (comment_id, element_id, user_id, content, created_at, updated_at) VALUES
76  ('1', '1', '1', 'This is a comment on Class1 by John.', '2024-05-01 12:00:00', '2024-05-01 12:00:00'),
77  ('2', '2', '2', 'This is a comment on Class2 by Jane.', '2024-05-02 13:00:00', '2024-05-02 13:00:00'),
78  ('3', '3', '3', 'This is a comment on Interface1 by Alice.', '2024-05-03 14:00:00', '2024-05-03 14:00:00'),
79  ('4', '4', '4', 'This is a comment on Entity1 by Bob.', '2024-05-04 15:00:00', '2024-05-04 15:00:00'),
80  ('5', '5', '5', 'This is a comment on Relationship1 by Carol.', '2024-05-05 16:00:00', '2024-05-05 16:00:00');
81  SELECT* FROM Comments;
82
```

Results Messages

	comment_id	element_id	user_id	content	created_at	updated_at
1	1	1	1	This is a comment on Class1 by John.	2024-05-01 12:00:00.000	2024-05-01 12:00:00.000
2	2	2	2	This is a comment on Class2 by Jane.	2024-05-02 13:00:00.000	2024-05-02 13:00:00.000
3	3	3	3	This is a comment on Interface1 by Alice.	2024-05-03 14:00:00.000	2024-05-03 14:00:00.000
4	4	4	4	This is a comment on Entity1 by Bob.	2024-05-04 15:00:00.000	2024-05-04 15:00:00.000
5	5	5	5	This is a comment on Relationship1 by Carol.	2024-05-05 16:00:00.000	2024-05-05 16:00:00.000

- **Diagramming Tool (Diagram)**



- **Code Generator (Class Design)**

1. Code Generator Controller Class

Description:

This controller class manages interactions between user inputs (selected diagram/model), the generation logic, and the output to the user interface.

Attributes:

- **diagramModel:** holds the parsed UML diagram.
- **selectedLanguage:** programming language selected by the user.
- **generator:** instance of specific language generator class (e.g., JavaCodeGenerator).

Methods:

- **generateCode()**: Triggers code generation for selected language.
- **setLanguage(language: String)**: Sets target language.
- **setModel(model: DiagramModel):** Sets the input diagram for processing.

Implementation:

```
Public class CodeGeneratorController {
   private DiagramModel diagramModel;
   private String selectedLanguage;
   private CodeGenerator generator;

   public void setLanguage(String language) {
      this.selectedLanguage = language;
      if(language.equals("Java")) generator = new JavaCodeGenerator();
else if(language.equals("Python")) generator = new
PythonCodeGenerator();            }
   public void setModel(DiagramModel model) {
      this.diagramModel = model;        }
   public String generateCode() {
      return generator.generate(diagramModel);    }
```

2.Class: JavaCodeGenerator

Description:

Concrete class that converts UML diagrams into Java code.

Attributes:

- **outputCode:** Stores the generated code.

Methods:

- **generate(model: DiagramModel):** Parses classes, attributes, methods from model and outputs Java code.

Implementation:

```
public class JavaCodeGenerator implements CodeGenerator {
   private String outputCode;
   public String generate(DiagramModel model) {
      StringBuilder sb = new StringBuilder();
      for(UMLClass umlClass : model.getClasses()) {
         sb.append("public class ").append(umlClass.getName()).append(" {\n");
         for(UMLAttribute attr : umlClass.getAttributes()) {
            sb.append("    ").append(attr.getVisibility())
              .append(" ").append(attr.getType())
append(" ").append(attr.getName()).append(";\n");
         }
         for(UMLMethod method : umlClass.getMethods()) {
            sb.append("    ").append(method.getVisibility())
              .append(" ").append(method.getReturnType())
              .append(" ").append(method.getName()).append("() {}\n");
         }
         sb.append("}\n\n");
  }
      outputCode = sb.toString();
      return outputCode;
   }
}
```

Detailed Functionality

The CodeGeneratorController acts as the mediator between the diagramming model and the code output. Upon selecting a language and providing a UML model, the controller instantiates the appropriate generator class such as JavaCodeGenerator. The generator class then parses the UML model, iterates through all the components like classes, attributes, and methods, and constructs the syntax in the selected language.

◆ Code Generator (Algorithm)

1. Generate Java Code Algorithm

Input: UML Diagram Model

Process:

- Parse each class in the model.
- For each class, extract name, attributes, and methods.
- Format each part into Java class syntax.
- Accumulate and return as string.

Output: Java source code string

Implementation

```
Algorithm GenerateJavaCode(model)
Begin
   Initialize output as empty string
   For each UMLClass in model.classes:
      Append "public class <ClassName> {"
      For each attribute in UMLClass:
         Append "    <visibility> <type> <name>;"
      For each method in UMLClass:
         Append "    <visibility> <returnType> <name>() {}"
      Append "}"
   Return output
End
```


◆ Code Generator (Database)

1. CodeTemplates Table:

This table stores reusable code templates for different programming languages, which can be used by the generator to produce boilerplate code (e.g., class structure, method structure).

- **template_id:** Unique identifier for each template, set as the primary key.
- **language:** Programming language the template is associated with (e.g., Java, Python).
- **template_type:** Type of code (e.g., class, method, attribute).
- **template_body:** The actual template content using placeholders.
- **created_at:** Timestamp of creation.
- **updated_at:** Timestamp of last update.

```sql
CREATE TABLE CodeTemplates (
    template_id INT PRIMARY KEY AUTO_INCREMENT,
    language VARCHAR(50) NOT NULL,
    template_type VARCHAR(50) NOT NULL,
    template_body TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

INSERT INTO CodeTemplates (language, template_type, template_body)
VALUES
('Java', 'class', 'public class {{class_name}} {\n{{attributes}}\n{{methods}}\n}'),
('Java', 'attribute', '{{visibility}} {{type}} {{name}};'),
('Java', 'method', '{{visibility}} {{return_type}} {{name}}() {}');

desc CodeTemplates;
```

```
+-------------+----------+---------------+-----------------------------------------------------------------------+---------------------+---------------------+
| template_id | language | template_type | template_body                                                         | created_at          | updated_at          |
+-------------+----------+---------------+-----------------------------------------------------------------------+---------------------+---------------------+
           1 | Java     | class         | public class {{class_name}} {
{{attributes}}
{{methods}}
} | 2025-04-12 17:08:23 | 2025-04-12 17:08:23 |
           2 | Java     | attribute     | {{visibility}} {{type}} {{name}};                                     | 2025-04-12 17:08:23 | 2025-04-12 17:08:23 |
           3 | Java     | method        | {{visibility}} {{return_type}} {{name}}() {}                          | 2025-04-12 17:08:23 | 2025-04-12 17:08:23 |
+-------------+----------+---------------+-----------------------------------------------------------------------+---------------------+---------------------+
```


2. GeneratedCodeLogs Table:

This table stores generated code for each diagram along with metadata for versioning and auditing purposes.

- **log_id:** Unique identifier for each log entry (primary key).
- **diagram_id**: Foreign key referencing the diagrams table to link the log with a diagram.
- **language:** Programming language in which the code was generated.
- **generated_code**: Text field storing the actual generated source code.
- **generated_by:** User who initiated the generation.

generated_at: Timestamp of code generation.

```
CREATE TABLE GeneratedCodeLogs (
    log_id INT PRIMARY KEY AUTO_INCREMENT,
    diagram_id INT NOT NULL,
    language VARCHAR(50) NOT NULL,
    generated_code TEXT NOT NULL,
    generated_by VARCHAR(100),
    generated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (diagram_id) REFERENCES diagrams(diagram_id) ON DELETE CASCADE
);
INSERT INTO GeneratedCodeLogs (diagram_id, language, generated_code, generated_by)
VALUES
(1, 'Java', 'public class User {\n    private String username;\n    private String password;\n\n    public boolean login() {}\n
select* from GeneratedCodeLogs
```

3. SupportedLanguages Table:

This table stores the programming languages supported by the code generator.

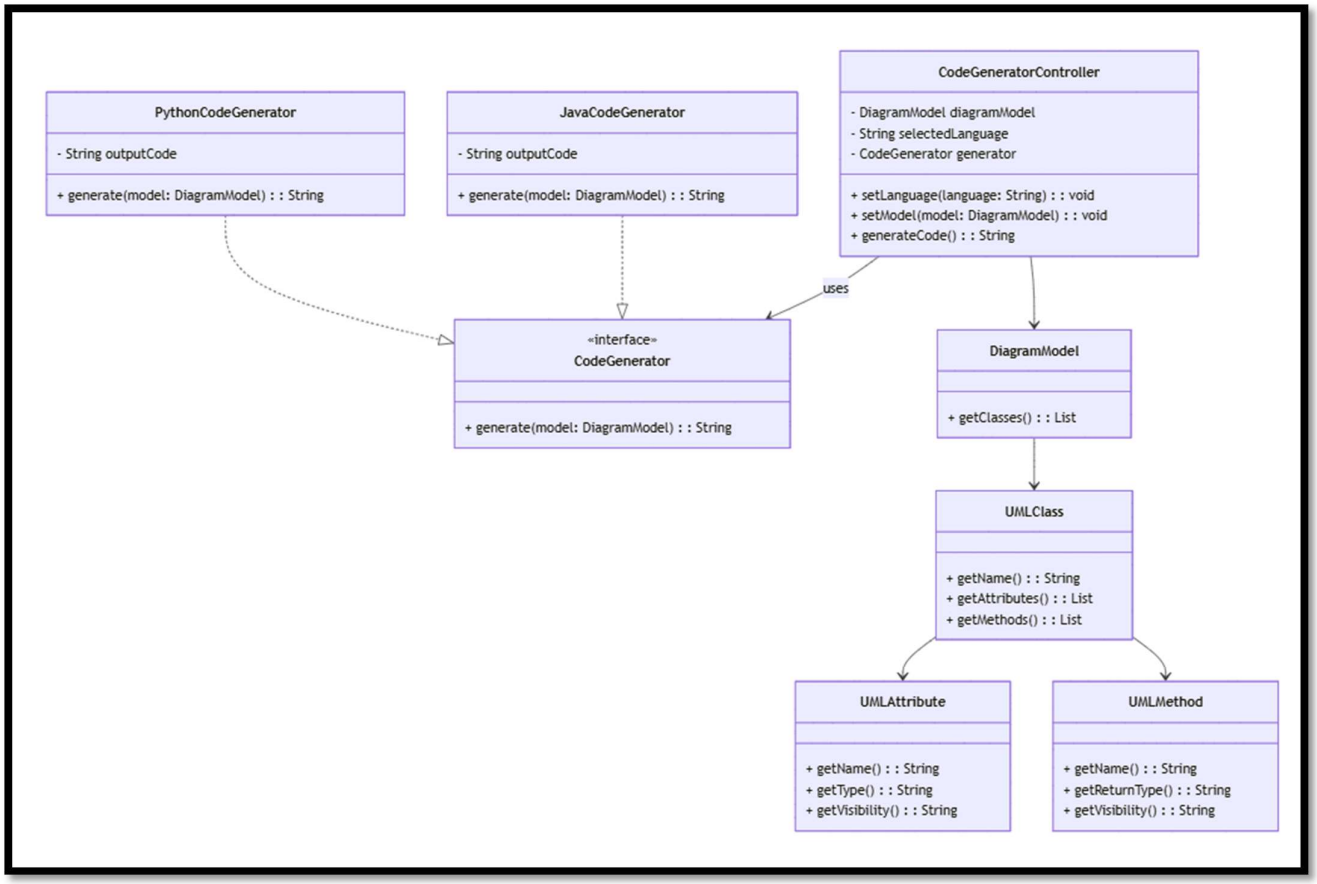
- **language_id:** Unique ID (primary key).
- **language_name:** Name of the language (e.g., Java, Python).
- **file_extension:** Standard file extension for the language (e.g., .java, .py).

is_active: Boolean to indicate if language is currently supported.

```
CREATE TABLE SupportedLanguages (
    language_id INT PRIMARY KEY AUTO_INCREMENT,
    language_name VARCHAR(50) UNIQUE NOT NULL,
    file_extension VARCHAR(10) NOT NULL,
    is_active BOOLEAN DEFAULT TRUE
);
INSERT INTO SupportedLanguages (language_name, file_extension)
VALUES
('Java', '.java'),
('Python', '.py');
 select * from SupportedLanguages;
```


language_id	language_name	file_extension	is_active
1	Java	.java	1
2	Python	.py	1

- **Code Generator (Diagram)**



◆ Testing (Class Design)

1. Testing Type Class

Description:

The **TestingType** enumeration defines various testing categories used to classify test cases. It ensures consistency in specifying the type of testing performed.

Attributes:

- **type_id**: Unique identifier for the testing type.
- **type_name**: Name of the testing type (e.g., Unit Testing, Integration Testing).
- **description:** Description of what the testing type entails.

Methods:

- **get_type_id():** Returns the ID of the testing type.
- **set_type_id(type_id):** Sets the ID of the testing type.
- **get_type_name():** Returns the name of the testing type.
- **set_type_name(type_name):** Sets the name of the testing type.
- **get_description():** Returns the description of the testing type.
- **set_description(description):** Updates the description of the testing type.

Implementation:

```
public enum TestingType {

   UNIT_TESTING,

   INTEGRATION_TESTING,

   SYSTEM_TESTING,

   REGRESSION_TESTING,

   SMOKE_TESTING  }
```


Detailed Functionality:

This class ensures standardization by categorizing tests. It distinguishes between Unit, Integration, System, etc., and links each type to respective modules. This supports consistent QA strategies and aligns test definitions with architectural layers.

2. Test Plan Class

Description:

The **TestPlan** class defines the structure for planning testing activities within a software project. It encapsulates key information including scope, responsibilities, and coverage.

Attributes:

- **plan_id:** Unique identifier for the test plan.
- **plan_name:** Name of the test plan.
- **objectives:** Goals or purpose of the test plan.
- **testing_type**: Type of testing associated with the plan (as a TestingTypes object).

Methods:

- **create_plan():** Creates a new test plan entry.
- **update_plan(objectives):** Updates the objectives of the test plan.
- **get_plan_id():** Returns the test plan ID.
- **set_plan_id(plan_id):** Sets the test plan ID.
- **get_plan_name():** Returns the test plan name.
- **set_plan_name(plan_name):** Sets the test plan name.
- **get_objectives():** Returns the objectives of the plan.
- **set_objectives(objectives):** Updates the test plan objectives.
- **get_testing_type():** Returns the associated testing type.
- **set_testing_type(testing_type):** Updates the associated testing type.

Implementation:

```
class TestPlan {

   String testPlanId, title, scope, objectives, createdBy;

   LocalDate createdDate;

   void createTestPlan(String id, String title, String scope, String obj, String user) {

      this.testPlanId = id;

      this.title = title;



      this.scope = scope;

      this.objectives = obj;

      this.createdBy = user;

      this.createdDate = LocalDate.now();     }

   void updateTestPlan(String scope, String obj) {

      this.scope = scope;

      this.objectives = obj;}

   String getTestPlanDetails() {

      return title + " | " + scope + " | " + objectives;}}
```

Detailed Functionality:

The Test Plan class captures the planning phase of software testing. It stores meta-information such as scope, objectives, and user responsibility. It enables structured documentation of test strategy, with options to update or review it dynamically. The algorithm initializes values and organizes them into a defined format.

3. Test Case Class

Description:

The **TestCase** class defines a test that can be executed to validate a specific functionality or unit of the software. It contains data for test input, expected result, and actual outcome.

Attributes:

- **test_case_id:** Unique identifier for the test case.
- **description:** Summary of the test case scenario.
- **input:** Input data provided for the test.
- **expected_output:** The expected outcome/result.
- **actual_output:** The real outcome from running the test.
- **status:** Boolean indicating test result (true = pass, false = fail).

Methods:

- **run_test(actual_output):** Compares expected and actual output and updates status.
- **get_result_summary():** Returns whether the test passed or failed.
- **get_test_case_id():** Returns the test case ID.
- **get_input():** Returns the input data.
- **get_expected_output():** Returns the expected output.
- **get_actual_output():** Returns the actual output.
- **is_status():** Returns the status (pass/fail).

Implementation:

```
class TestCase {

   String testCaseId, description, inputData, expectedOutput, actualOutput;

   String status;

   void runTestCase(String input) { this.actualOutput = executeLogic(input);

      evaluateResult();            }            void evaluateResult() {

      this.status = expectedOutput.equals(actualOutput) ? "PASS" : "FAIL";    }

   String executeLogic(String input) {

      return input.toUpperCase(); // Simulated logic    }            }
```


Detailed Functionality:

The Test Case class supports the execution of unit-level test validations. Each case is defined with input and expected output. The logic is executed and evaluated, setting the test case's status accordingly. This supports automation, validation, and error detection.

4. Bug Report Class

Description:

The **BugReport** class manages the logging of defects encountered during testing. It includes information on severity, priority, and current status of a bug.

Attributes

- **bug_id:** Unique identifier for the bug.
- **description:** Short description of the bug.
- **severity:** Severity of the bug (e.g., Low, Medium, High).
- **status:** Current status of the bug (e.g., Open, Closed).
- **reported_by:** Name of the person who reported the bug.

Methods

- **update_status(status):** Updates the bug's status.
- **get_bug_id():** Returns the bug ID.
- **get_description():** Returns the bug description.
- **get_severity():** Returns the severity of the bug.
- **get_status():** Returns the current status of the bug.
- **get_reported_by():** Returns the name of the reporter.

Implementation:

```
class BugReport {

    String bugId;

    String description;

    String severity;

String priority;
```


```
  String status;

   String assignedTo;

   BugReport(String bugId, String description, String severity, String priority, String status, String
assignedTo) {



this.bugId = bugId;

      this.description = description;

      this.severity = severity;

      this.priority = priority;

      this.status = status;

      this.assignedTo = assignedTo;

  }

  void displayBug() {

     System.out.println("Bug ID: " + bugId);

      System.out.println("Description: " + description);

      System.out.println("Severity: " + severity);

      System.out.println("Status: " + status);

      System.out.println("Assigned To: " + assignedTo);

  }
```

Detailed Functionality:

This class manages defect reporting. Upon test failure, the bug details are logged with classification (severity and priority). Developers and testers use this class to track resolution progress. Its lifecycle spans from Open to Closed, with updates via status changes.

5. Test Scenario Class

Description:

The **TestScenario** class outlines a broader testing context or goal, typically composed of several individual test cases. It provides a structured description and flow.

Attributes:

- **scenario_id**: Unique identifier for the test scenario.
- **title**: Title of the test scenario.
- **steps**: Steps involved in executing the scenario.
- **expected_result**: The expected result after scenario execution.

Methods:

- **describe_scenario**(): Displays full scenario details.
- **get_scenario_id():** Returns the scenario ID.
- **get_title():** Returns the title of the scenario.
- **get_steps():** Returns the steps involved.
- **get_expected_result():** Returns the expected result.

Implementation:

```
class TestScenario {

   String scenarioId;

   String objective;String steps;

 String expectedResult;

   TestScenario(String scenarioId, String objective, String steps, String expectedResult) {

      this.scenarioId = scenarioId;

      this.objective = objective;

      this.steps = steps;

      this.expectedResult = expectedResult;   }
```


```
    void displayScenario() {
        System.out.println("Scenario ID: " + scenarioId);
            System.out.println("Objective: " + objective);
        System.out.println("Steps: " + steps);
        System.out.println("Expected Result: " + expectedResult);
    }
}
```

Detailed Functionality:

Test Scenario helps group and organize test cases under a meaningful test flow or use-case. It improves test organization and traceability.
Adding/removing test cases within a scenario allows dynamic management of grouped tests, simplifying QA planning.

◆ Testing (Algorithm)

1. Algorithm for Testing Type Class

Purpose: To add a new type of software testing.

Algorithm:

- **Input**: Type name, description
- **Process**:
  Generate a unique type ID
  Store the type name and description
- **Output**: Boolean indicating success or failure

Implementation

```
boolean addTestingType(String typeName, String description) {

   int typeId = generateUniqueTypeId();

   return saveToTestingTypesTable(typeId, typeName, description);

}
```

2. **Algorithm for Test Plan Class**

Purpose: To create a new test plan for a software module.

Algorithm

- **Input**: Plan name, objectives, testing type
- **Process**:
  Generate a unique plan ID
  Link the plan to a testing type
  Store objectives
- **Output**: Boolean indicating success or failure

Implementation

```
boolean createTestPlan(String planName, String objectives, TestingTypes testingType) {

   int planId = generateUniquePlanId();

   return saveToTestPlansTable(planId, planName, objectives, testingType);}
```

3. **Algorithm for Test Case Class**

- **Purpose**: To execute a test case and compare the actual vs expected output.
- **Algorithm**
  - **Input**: Input data, expected output, actual output
  - **Process**:
    Compare expected output with actual output
    Set test status to PASS if equal, else FAIL
  - **Output**: Boolean result (true = PASS, false = FAIL)
- **Implementation**

```
boolean runTestCase(String input, String expectedOutput, String actualOutput) {

   return expectedOutput.equals(actualOutput); }
```


4. Algorithm for Bug Report Class

Purpose: To log and report a software bug.

Algorithm

- **Input**: Bug description, severity, reported by
- **Process**:
  Generate a unique bug ID
  Set bug status as "Open"
  Store bug info in report table
- **Output**: Boolean indicating success or failure

Implementation

```
boolean reportBug(String description, String severity, String reportedBy) {

    int bugId = generateUniqueBugId();

    String status = "Open";

    return saveToBugReportsTable(bugId, description, severity, status, reportedBy);

}
```

5. Algorithm for Test Scenario Class

Purpose: To define a test scenario with its steps and expected outcome.

Algorithm

- **Input**: Title, steps, expected result
- **Process**:
  Generate a unique scenario ID
  Store title, steps, and expected result
- **Output**: Boolean indicating success or failure

Implementation

```
boolean addTestScenario(String title, String steps, String expectedResult) {

    int scenarioId = generateUniqueScenarioId();

 return saveToTestScenariosTable(scenarioId, title, steps, expectedResult); }
```


◆ Testing (Database)

1. Bug_Report Table:

- **bug_id:** Unique identifier for each bug report (Primary Key).
- **description:** Detailed description of the bug.
- **severity:** Severity level of the bug (e.g., Minor, Major, Critical).
- **priority:** Priority level (e.g., Low, Medium, High).
- **steps_to_reproduce:** Instructions or steps to reproduce the bug.
- **expected_result:** The expected outcome if the bug were not present.
- **actual_result:** The actual outcome when the bug occurs.
- **status:** Current status of the bug (e.g., Open, In Progress, Fixed, Closed).
- **assigned_to:** Name or identifier of the person/team the bug is assigned to.

```
CREATE TABLE bug_reports (
    bug_id INT PRIMARY KEY AUTO_INCREMENT,
    description TEXT NOT NULL,
    severity VARCHAR(50),
    priority VARCHAR(50),
    steps_to_reproduce TEXT,
    expected_result TEXT,
    actual_result TEXT,
    status VARCHAR(50), -- Open, In Progress, Fixed, Closed
    assigned_to VARCHAR(100),
    reported_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
INSERT INTO bug_reports (description, severity, priority, steps_to_reproduce, expected_result, actual_result, status, assigned_to)
VALUES ('Canvas error', 'Major', 'High', 'Open tool in browser', 'Canvas should load', 'Canvas invisible', 'Open', 'Frontend Dev');
select * from bug_reports
```

Output:

bug_id	description	severity	priority	steps_to_reproduce	expected_result	actual_result	status	assigned_to	reported_at
1	Canvas error	Major	High	Open tool in browser	Canvas should load	Canvas invisible	Open	Frontend Dev	2025-04-11 07:05:46

2. Test_Plans Table:

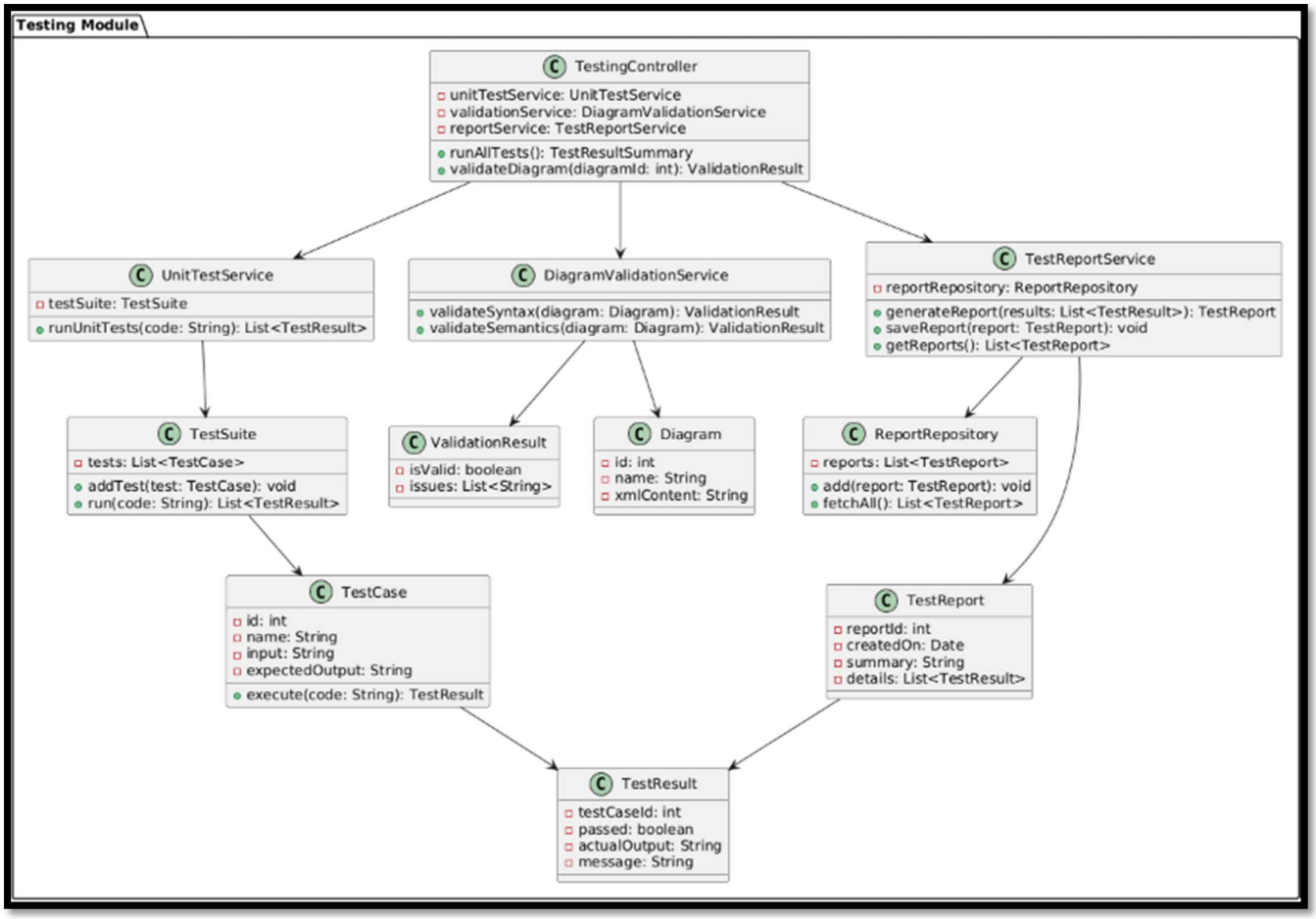
- **plan_id**: Unique identifier for each test plan (Primary Key).
- **project_name:** Name of the project for which the test plan is created.
- **scope:** Defines the objectives and boundaries of the test plan (what will be tested).
- **features_tested:** List of features that have been tested.
- **features_untested:** List of features that have not been tested yet.
- **prepared_by:** Name or role of the person who prepared the test plan.
- **created_at:** Timestamp indicating when the test plan was created (defaults to current time).

```
CREATE TABLE test_plans (
    plan_id INT PRIMARY KEY AUTO_INCREMENT,
    project_name VARCHAR(100) NOT NULL,
    scope TEXT,
    features_tested TEXT,
    features_untested TEXT,
    prepared_by VARCHAR(100),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
INSERT INTO test_plans (project_name, scope, features_tested, features_untested, prepared_by)
VALUES ('Online Design Tool', 'Validate core functionality', 'Canvas, Export, Save', 'Team Collab', 'QA Lead');

select * from test_plans
```

Output:

plan_id	project_name	scope	features_tested	features_untested	prepared_by	created_at
1	Online Design Tool	Validate core functionality	Canvas, Export, Save	Team Collab	QA Lead	2025-04-11 06:55:22

◆ Testing (Diagram)



◆ Maintenance (Class Design)

1. Class: Project Manager

Description:

Handles CRUD operations for projects and diagram elements.

Attributes:

- **project_id:** Unique identifier for the project.
- **name:** Name of the project.
- **owner_id:** Identifier for the project owner.
- **created_at:** Timestamp of project creation.
- **updated_at:** Timestamp of last update.
- **diagramElements:** List of elements within the project.

Methods:

- **createProject (name, owner_id):** Creates a new project.
- **getProject (project_id):** Retrieves a project.
- **updateProject (project_id, updated_data):** Updates project details.
- **deleteProject (project_id):** Deletes the project.
- **addDiagramElement (project_id, element):** Adds a new diagram element.
- **updateDiagramElement (project_id, element_id, new_properties):** Updates an element's properties.
- **deleteDiagramElement (project_id, element_id):** Deletes a diagram element.

Implementation:

```
public Project createProject(String name, String ownerId) {

   String id = UUID.randomUUID().toString();

   Project p = new Project(id, name, ownerId);

   projectStore.put(id, p);

   diagramElements.put(id, new ArrayList<>());

   return p;
```


```
}

public Project getProject(String projectId) {

   return projectStore.get(projectId);}

public void updateProject(String projectId, Map<String, String> updatedData) {

   Project p = projectStore.get(projectId);

   p.setName(updatedData.get("name"));

   p.setUpdatedAt(LocalDateTime.now());

}

public void deleteProject(String projectId) {

   diagramElements.remove(projectId);        }

public void addDiagramElement(String projectId, DiagramElement element)

public void updateDiagramElement(String projectId, String elementId, JSONObject
newProps) {

   for (DiagramElement e : diagramElements.get(projectId)) {

      if (e.getElementId().equals(elementId)) {

         e.update_properties(newProps);   break;       }}}

public void deleteDiagramElement(String projectId, String elementId) {

   diagramElements.get(projectId).removeIf(e -> e.getElementId().equals(elementId));

}
```

Detailed Functionality:

The ProjectManager class is responsible for managing the lifecycle of projects and their associated diagram elements. It allows users to create new projects, retrieve existing project details, update project metadata (like name), and delete projects when no longer needed. Within each project, users can also add, update, or delete diagram elements individually. The class uses HashMaps to efficiently store and retrieve project and element data using unique identifiers, enabling quick lookups and modifications.

2. Class: DiagramElement

Description:

Represents a diagram element in a design (class, interface, etc.)

Attributes:

- **element_id:** Unique identifier for the element.
- **diagram_id:** Identifier for the diagram to which the element belongs.
- **element_type:** Type of the element (e.g., class, interface, component).
- element_propertie**s:** Properties of the element stored as a JSON object.
- **created_at:** Timestamp when the element was created.
- **updated_at:** Timestamp when the element was last updated.

Methods:

- **update_properties(new_properties):** Updates the properties of the element.

Implementation:

```
public DiagramElement(String id, String diagramId, String type, JSONObject props) {

    this.element_id = id;

    this.diagram_id = diagramId;

    this.element_type = type;

    this.element_properties = props;

    this.created_at = LocalDateTime.now();

}

public String getElementId() { return element_id; }

public void update_properties(JSONObject new_properties) {

    this.element_properties = new_properties;

   this.updated_at = LocalDateTime.now();

}
```


Detailed Functionality:

DiagramElement encapsulates the data structure for a visual component (e.g., class, interface, component) in a diagram. Each element is uniquely identified and associated with a diagram via a diagram ID. The element's properties, such as name, attributes, or position, are stored in a JSON object for flexibility. The update_properties method allows real-time modification of these attributes, updating the timestamp to reflect the change. This design makes element manipulation extensible and straightforward during diagram editing.

3. Class: VersionNotifier

Description:

Notifies users about version updates.

Attributes:

- **project_id**: ID of the project.
- **version_number:** Current version number.
- **subscribers:** List of user IDs subscribed to notifications.

Methods:

- notifyUsers(project_id, version): Notifies all subscribers.
- subscribeUser(project_id, user_id): Subscribes a user.

Implementation:

```
public void notifyUsers(String projectId, int version) {

   List<String> users = subscribers.getOrDefault(projectId, new ArrayList<>());

   for (String userId : users) {

      String msg = "User "+userId+"notified:Project"+projectId +" updated to version " +
version;          System.out.println(msg);    }}

public void subscribeUser(String projectId, String userId) {

   subscribers.computeIfAbsent(projectId, k -> new ArrayList<>()).add(userId);

}
```


Detailed Functionality:

VersionNotifier handles communication regarding version updates within a project. It maintains a list of subscribers for each project who should be notified when a new version is saved or deployed. When notifyUsers is called, each subscriber receives a message (via log or notification system) informing them about the latest version. This feature enhances team collaboration and transparency, especially when multiple users are editing the same design project concurrently.

4. Class: CollaboratorManager

Description:

Manages users (collaborators) in a project.

Attributes:

- project_i**d:** Project identifier.
- **collaborators:** List of user IDs in the project.

Methods:

- **addCollaborator(project_id, user_id):** Adds a user to the project.
- **removeCollaborator(project_id, user_id):** Removes a user from the project.

Implementation:

```
public void addCollaborator(String projectId, String userId) {

    collaborators.computeIfAbsent(projectId, k -> new ArrayList<>()).add(userId);

}



public void removeCollaborator(String projectId, String userId) {

    collaborators.getOrDefault(projectId, new ArrayList<>()).remove(userId);

}
```


Detailed Functionality:

CollaboratorManager facilitates collaborative work by maintaining a list of users assigned to each project. It supports functions to add or remove collaborators, allowing dynamic team formation. Collaborators can be granted read or write access depending on future implementation of permissions. The use of HashMaps ensures fast access and updates, ensuring scalability across multiple projects. This feature supports multi-user environments and version-controlled teamwork.

5. Class: AutoSaveManager

Description:

Handles automatic backups of projects.

Attributes:

- **project_id:** Project ID to track.
- **last_backup:** Last saved backup.
- **auto_save_interval:** Time interval for auto-save.

Methods:

- **triggerAutoSave(project_id):** Saves a backup.
- **restoreLastBackup(project_id):** Restores the backup.

Implementation:

```
public void triggerAutoSave(String projectId) {
    ProjectSnapshot snapshot = captureCurrentState(projectId);
    backupStore.put(projectId, snapshot);       }
public ProjectSnapshot restoreLastBackup(String projectId) {
    return backupStore.get(projectId);
}
private ProjectSnapshot captureCurrentState(String projectId) {
    return new ProjectSnapshot(projectId);      }
```


Detailed Functionality:

The AutoSaveManager automatically backs up a project's current state at regular intervals to safeguard against data loss. It creates snapshots using a placeholder captureCurrentState method (which can be expanded to serialize data). These backups are stored in a map and can be restored if needed, offering a fail-safe mechanism. The autoSaveInterval can be configured to adjust how frequently backups occur. This class ensures continuity of work even during unexpected browser crashes or session timeouts.

6. Class: PasswordResetManager

Description:

Handles forgotten password reset.

Attributes:

- **user_id:** ID of the user.
- **reset_token:** Unique token for reset.
- **credentials:** Map of user credentials.

Methods:

- **initiateReset(user_id):** Generates a token.
- **verifyToken(user_id, token):** Verifies a token.
- **updatePassword(user_id, new_password):** Updates user password.

Implementation:

```
public String initiateReset(String userId) {
    String token = UUID.randomUUID().toString();
    resetTokens.put(userId, token);
    return token;
}
public boolean verifyToken(String userId, String token) {
    return resetTokens.getOrDefault(userId, "").equals(token);
}
public void updatePassword(String userId, String newPassword) {
    userCredentials.put(userId, hash(newPassword));
```


```
    resetTokens.remove(userId);
}
private String hash(String input) {
    return Integer.toHexString(input.hashCode()); // Simplified for demo}
```

Detailed Functionality:

PasswordResetManager enables secure recovery of lost or forgotten passwords. When a user requests a reset, a unique token is generated and associated with their ID. The token must be verified before allowing a password update. After successful verification, the new password is hashed and stored securely, and the reset token is removed. This prevents unauthorized access and ensures that only legitimate users can modify their credentials, improving overall system security.

◆ Maintenance (Algorithm)

1. Project Creation

Purpose: To create a new project in the system.

Algorithm

- **Input:** Project name, owner ID
- **Process:**

  Generate a unique project ID.

  Set the current timestamp as creation and update time.

  Create a Project object and store it.

- **Output:** Project object or ID

Implementation:

```
Project createProject(String name, String ownerId) {
    String id = generateUniqueProjectId();
    LocalDateTime now = LocalDateTime.now();
    return storeProject(new Project(id, name, ownerId, now, now));
}
```

2. Element Update

Purpose: To update the properties of a diagram element.

Algorithm

- **Input:** Project ID, Element ID, new properties (JSON)
- **Process:**

  Find the element in the list.

  Call update_properties().

  Update the timestamp.

- **Output:** Boolean indicating success or failure

- **Implementation:**

```
boolean updateElementProperties(String projectId, String elementId, String
newPropsJson) {
    Element e = findElement(projectId, elementId);
    if (e == null) return false;
    e.updateProperties(newPropsJson);    e.updatedAt = LocalDateTime.now();
    return true;
}    return true;       }
Element findElement(String projectId, String elementId) {
    return null; // return the found element or null if not found       }
```


2. Version Notification

Purpose: To notify users when a new version of a project is saved.

Algorithm:

- **Input:** Project ID, version number
- **Process:**

  Fetch list of subscribers.

  Generate and print message for each.

- **Output:** None (log output only)

Implementation:

```
boolean notifyVersion(String versionId, String message) {
   List<User> users = getActiveUsers();
   for (User user : users)
      sendNotification(user, versionId, message);
   return true;}
```

3. Collaborator Addition

Purpose: To add a collaborator to a project.

Algorithm:

- **Input:** Project ID, User ID
- **Process:**

  Get or create list of collaborators.

  Add user ID to list.

- **Output:** Boolean indicating success

Implementation:

```
boolean addCollaborator(String projectId, String collaboratorId) {
    if (!isValidUser(collaboratorId)) return false;
    return addToProjectTeam(projectId, collaboratorId);
}
```

4. **Auto-Save Trigger**

Purpose: To auto-save the current state of a project.

Algorithm:

- **Input:** Project ID
- **Process:**

  Generate project snapshot.

  Store it in backup store.

- **Output:** Boolean indicating success

Implementation:

```
boolean autoSaveDiagram(String diagramId, String stateJson) {
    LocalDateTime now = LocalDateTime.now();
    return saveSnapshot(diagramId, stateJson, now); }
```

5. **Password Reset**

Purpose: To update user password after verifying reset token.

Algorithm:

- **Input:** User ID, token, new password
- **Process:**

  Verify token.

  Hash and store new password.

  Remove token.

- **Output:** Boolean indicating success or failure

Implementation:

```
boolean resetPassword(String userId, String newPassword) {

    if (!userExists(userId)) return false;

    String hashed = hashPassword(newPassword);

    return updatePassword(userId, hashed);

}
```

◆ Maintenance (Database)

1. Projects Table

- **project_id:** Primary key. Unique identifier for each project.
- **name:** Name of the project .
- **owner_id:** Foreign key referencing Users.user_id; identifies who created the project.
- **created_at:** Timestamp when the project was initially created.
- **updated_at:** Timestamp when the project was last modified or saved.
- **element_id:** Primary key. Unique identifier for each diagram element.
- **project_id:** Foreign key referencing Projects.project_id; identifies which project this belongs to.
- **element_type:** Type of element .
- **position:** Coordinates or position metadata for UI rendering.
- **created_at:** Timestamp of element creation.

```sql
CREATE TABLE Projects (
    project_id INT PRIMARY KEY,
    name VARCHAR(255),
    owner_id INT,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
);

CREATE TABLE DiagramElements (
    element_id INT PRIMARY KEY,
    project_id INT,
    element_type VARCHAR(100),
    content TEXT,
    position VARCHAR(100),
    created_at TIMESTAMP,
    FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);

INSERT INTO Projects (project_id, name, owner_id, created_at, updated_at)
VALUES
(1, 'ODT Requirement Design',   1, '2025-04-01 09:00:00', '2025-04-01 09:00:00'),
(2, 'ODT Maintenance Diagrams', 2, '2025-04-02 10:30:00', '2025-04-04 14:00:00');
```

project_id	name	owner_id	created_at	updated_at
1	ODT Requirement Design	1	2025-04-01 09:00:00	2025-04-01 09:00:00
2	ODT Maintenance Diagrams	2	2025-04-02 10:30:00	2025-04-04 14:00:00

2. Version Control and Notification Table:

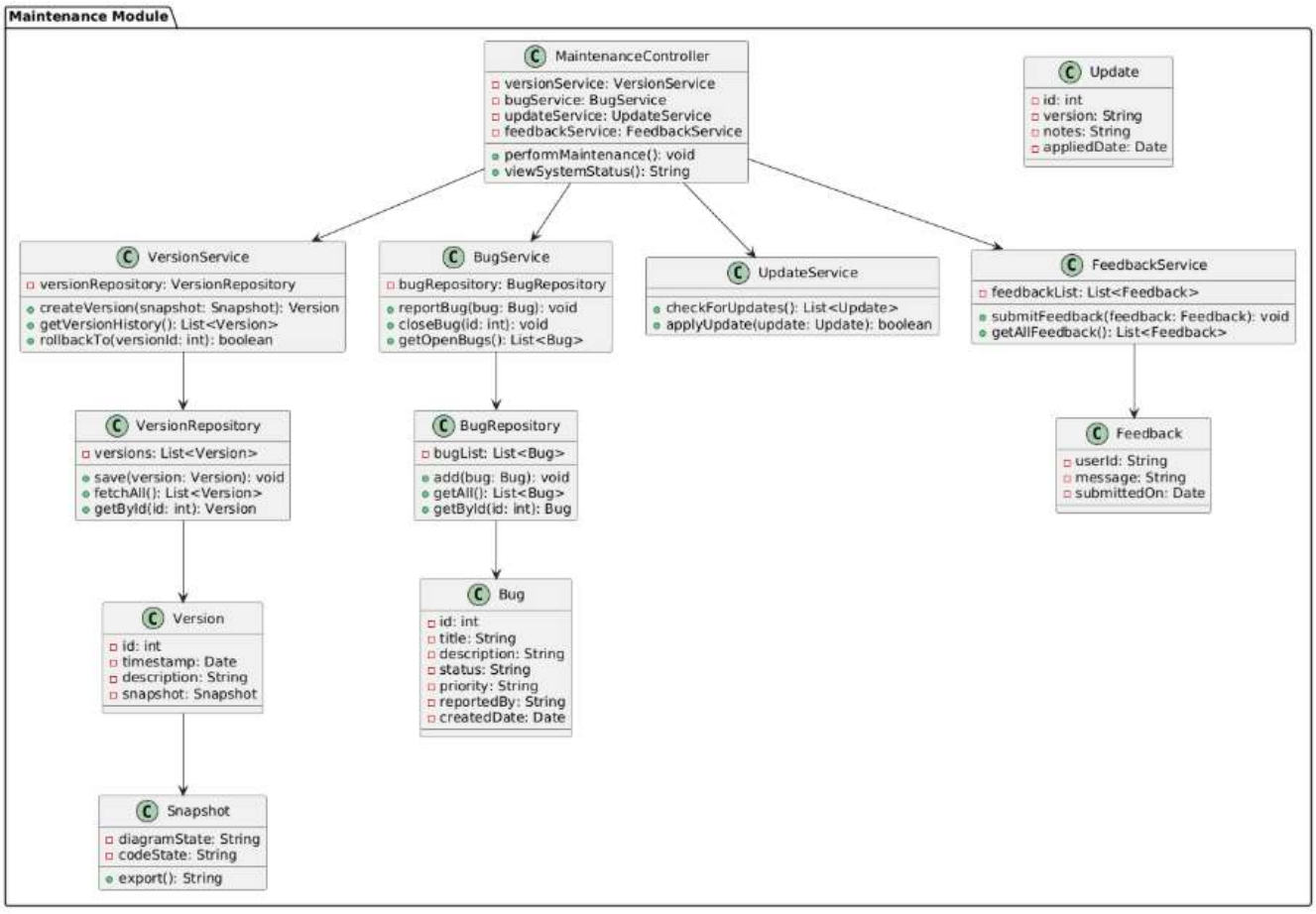
- **version_id:** Primary key. Unique identifier for each version entry.
- **project_id:** Foreign key referencing Projects.project_id; identifies the relevant project.
- **element_id:** Foreign key referencing DiagramElements.element_id; nullable if version applies to whole project.
- **version_number**: Numeric or semantic version.
- **created_at:** Timestamp when the version entry was created.
- **created_by:** Foreign key referencing Users.user_id; indicates who made the change.
- **notification_id:** Primary key. Unique identifier for each notification.
- **user_id:** Foreign key referencing Users.user_id; recipient of the notification.
- **created_at**: Timestamp when the notification was created.

```sql
CREATE TABLE Versions (
    version_id INT PRIMARY KEY,
    project_id INT,
    version_note TEXT,
    created_at TIMESTAMP
);

CREATE TABLE Notifications (
    notification_id INT PRIMARY KEY,
    user_id INT,
    message TEXT,
    is_read BOOLEAN,
    created_at TIMESTAMP
);

INSERT INTO Versions (version_id, project_id, version_note, created_at)
VALUES
(1, 101, 'Initial design of class diagram', '2025-04-01 10:30:00'),
(3, 102, 'Updated use-case relationships', '2025-04-03 09:15:00');

INSERT INTO Notifications (notification_id, user_id, message, is_read, created_at)
VALUES
(1, 201, 'Your project version has been saved successfully.', FALSE, '2025-04-01 10:30:10'),
(3, 201, 'Reminder: Review the latest changes made yesterday.', TRUE, '2025-04-03 08:00:00');
```

version_id	project_id	version_note	created_at
1	101	Initial design of class diagram	2025-04-01 10:30:00
3	102	Updated use-case relationships	2025-04-03 09:15:00

notification_id	user_id	message	is_read	created_at
1	201	Your project version has been saved successfully.	0	2025-04-01 10:30:10
3	201	Reminder: Review the latest changes made yesterday.	1	2025-04-03 08:00:00

◆ Maintenance (Diagram)



◆ OPTIMIZATION AND PERFORMANCE CONSIDERATIONS

- **Batch Processing:** Optimize bulk operations by implementing batch processing techniques to minimize transaction overhead.
- **Concurrency Control:** Ensure data consistency and integrity by employing appropriate concurrency control mechanisms, such as optimistic or pessimistic locking.
- **Caching:** Implement caching strategies for frequently accessed data to reduce database load and improve response times.
- **Indexing:** Use database indexing on commonly queried fields (e.g., user_id, project_id) to enhance query performance.

This documentation covers the essential algorithms used in the Online Design Tool (ODT), providing a comprehensive guide to their purpose, functionality, and implementation. These algorithms are designed to ensure the tool operates efficiently, securely, and effectively, meeting the needs of its users.

DESIGN PATTERNS

◈ DESIGN PATTERNS

This section outlines the key design patterns applied across different modules of the Online Design Tool CASE system. The patterns have been selected based on each module's functionality, complexity, and the need for modular, maintainable architecture. Each pattern is categorized as Creational, Structural, Behavioral, or Architectural.

1. Requirement Analyzer

Singleton pattern

- **Category:** Creational
- **Intent:** Ensure one centralized parser/validator instance.
- **Motivation:** Maintains consistent state across requirement input and validation.
- **Code Snippet:**

```
class RequirementParser {

   private static RequirementParser instance;

   private RequirementParser() {}


   public static RequirementParser getInstance() {

      if (instance == null) instance = new RequirementParser();

      return instance;
   }   public void parse(String text) { /* parsing logic */ }
}
```


Strategy Pattern

- **Category:** Behavioral
- **Intent:** Define a family of parsing algorithms (e.g., plain text, JSON, docx).
- **Motivation:** Allows the system to plug in different formats without modifying the core logic.
- **Code Snippet:**

```
interface ParseStrategy {
    RequirementModel parse(String input);
}
class TextStrategy implements ParseStrategy { /* ... */ }
class JSONStrategy implements ParseStrategy { /* ... */ }
```

2. Diagramming Engine

Factory Pattern

- **Category:** Creational
- **Intent:** Create diagram elements (class, interface, connector) dynamically.
- **Motivation:** Allows easy addition of new element types in the future.
- **Code Snippet:**

```
class ElementFactory {
   static Element create(String type) {
      switch (type) {
         case "Class": return new ClassBox();
         case "Interface": return new InterfaceBox();
         case "Arrow": return new ConnectorArrow();
default: return null;   } } }
```


Observer Pattern

- **Category:** Behavioral
- **Intent:** Keep UI updated when model data changes.
- **Motivation:** Ensures real-time updates to the view without tightly coupling components.
- **Code Snippet:**

```
interface DiagramListener {
   void onDiagramChanged();

}

class DiagramModel {

    List<DiagramListener> listeners = new ArrayList<>();

    void addListener(DiagramListener l) { listeners.add(l); }

    void notifyChange() {

        for (DiagramListener l : listeners)
l.onDiagramChanged();

    }

}
```

3. Code Generator

Template Method Pattern

- **Category:** Behavioral
- **Intent:** Define the structure of code generation with language-specific steps defined by subclasses.
- **Motivation:** Supports consistent code generation for multiple languages.

- **Code Snippet:**

```
iabstract class CodeGenerator {
    public final void generate() {
        addHeaders();
        defineStructure();
        writeCode();
    }
    protected abstract void addHeaders();
    protected abstract void defineStructure();
    protected abstract void writeCode();
}
class JavaCodeGenerator extends CodeGenerator {
    protected void addHeaders() { /* Java-specific headers */ }
    protected void defineStructure() { /* Java structure */ }
    protected void writeCode() { /* Java code */ }
}
```

Factory Pattern

- **Category:** Creational
- **Intent:**Create language-specific components dynamically.
- **Motivation:** Easily support multiple output formats and structures.
- **Code Snippet:**

```
class CodeFactory {
    static CodeGenerator getGenerator(String language) {
        if (language.equals("Java")) return new JavaCodeGenerator();
        if (language.equals("Python")) return new
 PythonCodeGenerator();
        return null;      }
}
```


4. Testing Module

Strategy Pattern

- **Category**: Behavioural
- **Intent:** Define interchangeable testing strategies (unit, functional).
- **Motivation:** Add or switch test types without changing existing logic.
- **Code Snippet:**

```
interface TestStrategy {
   void run();       }

class UnitTest implements TestStrategy {
   public void run() { /* run unit test */ }

}

class FunctionalTest implements TestStrategy {
  public void run() { /* run functional test */ }

}
```

Observer Pattern

- **Category**: Behavioural
- **Intent:**Notify the UI when test results change.
- **Motivation:** Real-time test reporting and alerts..
- **Code Snippet:**

```
interface TestObserver {

   void onTestComplete();

}
class TestManager {
   List<TestObserver> observers = new ArrayList<>();
   void addObserver(TestObserver o) { observers.add(o); }
   void notifyObservers() {
      for (TestObserver o : observers)
    o.onTestComplete();
   }
```


5. Maintenance

Singleton Pattern

- **Category**: Creational
- **Intent:** Centralized change tracking and version management.
- **Motivation:** Ensures a consistent versioning interface across the tool.
- **Code Snippet:**

```
class VersionControl {
    private static VersionControl instance;
    private VersionControl() {}
    public static VersionControl getInstance() {
        if (instance == null)
            instance = new VersionControl();
        return instance;   }
}
```

Command Pattern

- **Category**: Behavioral
- **Intent:** Encapsulate actions as commands to allow undo/redo.
- **Motivation:** Enables user actions (add, delete, update) to be reversible.
- **Code Snippet:**

```
interface Command {
    void execute();
    void undo();     }
class AddElementCommand implements Command {
    Diagram diagram;
    Element element;
    public void execute() { diagram.addElement(element); }
    public void undo() { diagram.removeElement(element); }
}
```


◈ Module Wise Design Pattern Summary

Module	Design Pattern(s)	Category	Purpose / Usage
Requirement Analyzer	• Singleton • Strategy (Optional)	• Creational • Behavioral	• Central parser instance • Switch between input formats (e.g., text, JSON)
Diagramming Tool	• MVC (Model-View-Controller) • Factory • Observer (Optional)	• Architectural • Creational • Behavioral	• Structure UI interaction • Create diagram elements dynamically • Update UI on model change
Code Generator	• Template Method • Factory	• Behavioral • Creational	• Language-agnostic generation flow • Generate code components per language
Testing Module	• Strategy • Observer	• Behavioral	• Swap test types (unit, functional) • Notify UI of test result changes
Maintenance Module	• Singleton • Command	• Creational • Behavioral	• Central version control • Enable undo/redo of design/code changes

Reference Citations

- https://d2u1z1lopyfwlx.cloudfront.net/thumbnails/924ae65a-4f26-5b31-8604-b9e7efb89066/e1552eaa-b7d1-56e6-a296-59dbf0e1db7d.jpg
- https://www.tutorialspoint.com/software_architecture_design/introduction.htm
- https://www.geeksforgeeks.org/software-design-patterns/
- https://www.visual-paradigm.com/
- https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm
- https://d2u1z1lopyfwlx.cloudfront.net/thumbnails/924ae65a-4f26-5b31-8604-b9e7efb89066/e1552eaa-b7d1-56e6-a296-59dbf0e1db7d.jpg
- https://algomaster.io/learn/lld
- https://refactoring.guru/design-patterns