

NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY

MILITARY COLLEGE OF SIGNALS



SOFTWARE DESIGN & ARCHITECTURE
(SE-211)

COMPLEX ENGINEERING ACTIVITY

Submitted by:

- 1. AYESHA ISRAR**
- 2. MUHAMMAD AHMAD SULTAN**
- 3. SAMAR QAISER**
- 4. SHANDANA IFTIKHAR**

COURSE:

BESE-28

SECTION:

C

Submitted to: LEC. FAWAD KHAN

Dated: 10-06-2024

CATALOG OF COMPLEX ENGINEERING ACTIVITY

Table of Contents

▪ Title of the Project	3
▪ Abstract, History, and Objective	4
▪ Scope	5
▪ Beyond Current Focus	6
▪ Low-Level Design for Online Design Tool	7
• Data Structures for Online Design Tool.....	7
• Algorithms for Online Design Tool	18
• Database for Online Design Tool	39
▪ Mid-Level Design for Online Design Tool	48
• Graphical User Interfaces (GUI)	48
▪ High-Level Design for Online Design Tool	59
• Architectural Style for Online Design Tool.....	59
• Design Patterns for Online Design Tool	67
▪ Future Enhancements	75
▪ Potential Questions.....	76
▪ Conclusion.....	77
▪ Reference Citations	77

DEPARTMENT OF COMPUTER SOFTWARE ENGINEERING
Military College of Signals
National University of Sciences and Technology

www.mcs.nust.edu.pk



COMPLEX ENGINEERING ACTIVITY

Title of the Project

Takhayyul Crafts: Crafting Creativity Together



Takhayyul Crafts is a collaborative online design tool that empowers users to unleash their creativity collectively. Takhayyul Crafts facilitates efficient project management, collaborative design, and secure data handling for creative professionals.

▪ **Abstract**

The Online Design Tool (ODT) is a sophisticated platform designed to facilitate user interactions, project management, diagram creation, and collaborative work. This document provides a detailed overview of the algorithms underpinning the ODT, highlighting their roles, operations, and significance within the system. The report also explores the history of design tools, the objectives and scope of the ODT, future enhancements, and addresses potential questions regarding the system.

▪ **History of Design Tools**

Design tools have evolved significantly over the years, transitioning from manual drafting techniques to sophisticated digital platforms. Early tools focused on basic drafting capabilities, but as technology advanced, so did the complexity and functionality of design software. Modern design tools now incorporate features such as real-time collaboration, advanced data management, and integration with other software, providing a comprehensive environment for designers and engineers.

▪ **Objective**

The primary objective of the Online Design Tool (ODT) is to provide a user-friendly platform that facilitates the efficient creation, management, and collaboration of design projects. The tool aims to enhance productivity, ensure data security, and support seamless collaboration among users, ultimately improving the overall design process.

▪ Scope

The scope of the ODT encompasses several key areas:

- **User Authentication and Authorization:** Securely manage user access and permissions.
- **Project Management:** Enable users to create, retrieve, and delete projects efficiently.
- **Diagram Management:** Facilitate the creation, retrieval, and deletion of diagrams within projects.
- **Element Management:** Support the addition, retrieval, and deletion of diagram elements.
- **Collaboration Management:** Manage user roles and ensure effective collaboration within projects.
- **Comment Management:** Allow users to add, retrieve, and delete comments on diagram elements.
- **Data Validation:** Ensure the integrity and security of user inputs.
- **Notification Management:** Notify users of significant events and updates.
- **Security:** Protect sensitive data through encryption and two-factor authentication.
- **Backup and Recovery:** Ensure data safety through regular backups and recovery mechanisms.
- **Logging and Monitoring:** Track user activities and monitor system performance.
- **Version Control:** Manage changes to documents and diagrams.

▪ **Beyond Current Focus**

- **Machine Learning Integration**

Machine learning can significantly enhance the ODT by offering intelligent insights and automated processes. The integration of machine learning algorithms can help in providing intelligent suggestions for diagram improvements and error detection, making the design process more efficient and accurate.

- **Intelligent Suggestions**

Machine learning can analyze user design patterns and suggest improvements based on best practices and historical data.

- **Error Detection**

Algorithms can automatically detect potential errors in diagrams and alert users, helping to maintain the quality and accuracy of designs.

- **Advanced Search**

Enhancing search capabilities with natural language processing (NLP) and semantic search can make finding information within the ODT faster and more intuitive.

- **Natural Language Processing (NLP)**

NLP allows the system to understand and process user queries in natural language, making search interactions more user-friendly.

- **Semantic Search**

Implementing semantic search techniques can improve the relevance and accuracy of search results, providing users with more meaningful and contextual information.

LOW LEVEL DESIGN FOR ONLINE DESIGN TOOL (ODT)

Data Structures for Online Design Tool (ODT)

▪ Overview of Online Design Tool (ODT)

The Online Design Tool (ODT) is a web-based application designed to facilitate software design and architecture. It provides users with a platform to create, edit, and collaborate on various design diagrams such as UML diagrams, flowcharts, and wireframes.

▪ Purpose of Data Structures

Data structures are crucial for managing the data flow and storage in an Online Design Tool (ODT). They define how data is organized, stored, and manipulated within the system. Efficient data structures are essential for ensuring optimal performance, scalability, and maintainability of the application.

▪ Core Components

For designing an online design tool (ODT), the core components include Projects, Users, Diagrams, Elements, and Collaborations. Each component's detailed design and implementation are discussed below.

1. PROJECT CLASS:

- **Description:**

The **Project** class represents a software project containing multiple diagrams.

- **Attributes:**

- **project_id:** Unique identifier for the project.
- **project_name:** Name of the project.
- **created_at:** Timestamp when the project was created.
- **updated_at:** Timestamp when the project was last updated.
- **user_id:** Identifier for the user who owns the project.
- **diagrams:** List of diagrams associated with the project.

- **Methods:**

- **add_diagram(diagram):** Adds a new diagram to the project.
- **remove_diagram(diagram_id):** Removes a diagram from the project by its ID.

- **Implementation:**

```
class Project {
    int project_id;
    String project_name;
    Instant created_at;
    Instant updated_at;
    int user_id;
    List<Diagram> diagrams;
    Project(int id, String name, int user) {
        project_id = id;
        project_name = name;
        user_id = user;
        created_at = Instant.now();
        updated_at = created_at;
        diagrams = new ArrayList<>();
    }
    void addDiagram(Diagram diag) {
        diagrams.add(diag);
        updated_at = Instant.now();
    }
}
```



```
void removeDiagram(int diag_id) {  
    diagrams.removeIf(diag -> diag.diagram_id == diag_id);  
    updated_at = Instant.now();    }}
```

- **Detailed Functionality**

The Project class encapsulates essential attributes and functionalities related to a software project. It maintains information such as project ID, name, creation and modification timestamps, owner ID, and a list of diagrams associated with the project. This class enables users to add and remove diagrams from the project, thereby facilitating project management.

- **Data Structures Used**

The Project class utilizes the ArrayList data structure to store the list of diagrams. This allows for dynamic resizing and efficient traversal and manipulation of diagrams within a project.

2. USER CLASS:

- **Description:**

The **User** class represents a user of the ODT.

- **Attributes:**

- **user_id:** Unique identifier for the user.
- **username:** Username of the user.
- **email:** Email address of the user.
- **password:** Password for the user's account (hashed for security).
- **created_at:** Timestamp when the user account was created.
- **projects:** List of projects owned by the user.

- **Methods:**

- **create_project(project):** Adds a new project to the user's list of projects.

- **Implementation:**

```
class User {  
    int user_id;  
    String username;  
    String email;  
    String password;  
    Instant created_at;  
    List<Project> projects;  
    User(int id, String uname, String mail, String pass) {  
        user_id = id;  
        username = uname;  
        email = mail;  
        password = pass;  
        created_at = Instant.now();  
        projects = new ArrayList<>();  
    }  
    void createProject(Project proj) {  
        projects.add(proj);  
    }  
}
```

- **Detailed Functionality**

The User class stores user-specific details like user ID, username, email, password (hashed for security), creation timestamp, and a list of projects owned by the user. Users can create new projects using the createProject method, thereby establishing ownership and initiating project creation.

- **Data Structures Used**

The User class utilizes the ArrayList data structure to store the list of projects. This allows for efficient management of multiple projects owned by a user.

3. 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();  
        updated_at = created_at;  
        elements = new ArrayList<>();  
    }  
}
```

```

void addElement(Element elem) {
    elements.add(elem);
    updated_at = Instant.now();
}

void removeElement(int elem_id) {
    elements.removeIf(elem -> elem.element_id == elem_id);
    updated_at = Instant.now();
}
}

```

- **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.

- **Data Structures Used**

The Diagram class uses the `ArrayList` data structure to store the list of elements. This allows for efficient management and manipulation of elements within a diagram.

4. 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:**

- **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.

- **Data Structures Used**

The Element class uses a Map data structure to store element properties. This allows for efficient key-value pair management of element properties.

5. COLLABORATION CLASS:

- **Description:**

The **Collaboration** class represents a collaboration entry between users on a project.

- **Attributes:**

- **collaboration_id:** Unique identifier for the collaboration entry.
- **project_id:** Identifier for the project being collaborated on.
- **user_id:** Identifier for the user involved in the collaboration.
- **role:** Role of the user in the project (e.g., editor, viewer).
- **created_at:** Timestamp when the collaboration was created.

- **Implementation:**

```
import java.time.Instant;

class Collaboration {
    int collaboration_id;
    int project_id;
    int user_id;
    String role;
    Instant created_at;

    Collaboration(int collaboration_id, int project_id, int user_id, String role, Instant created_at) {
        this.collaboration_id = collaboration_id;
        this.project_id = project_id;
        this.user_id = user_id;
        this.role = role;
    }
}
```

```
        this.created_at = created_at;
    }
}
```

- **Detailed Functionality**

The Collaboration class models collaborative interactions between users on a project. It includes attributes such as collaboration ID, project ID being collaborated on, user ID of the collaborator, role of the user in the project (e.g., editor, viewer), and creation timestamp. This class facilitates collaboration management within the ODT platform.

- **Data Structures Used**

The Collaboration class uses basic data types and Instant for timestamps to ensure efficient handling and storage of collaboration details.

6. ODT DATABASE CLASS

- **Description:**

The **ODTDatabase** class manages all projects and users in the ODT.

- **Attributes:**

- **projects:** Dictionary storing projects by project ID.
- **users:** Dictionary storing users by user ID.

- **Methods:**

- **add_project(project):** Adds a new project to the database.
- **add_user(user):** Adds a new user to the database.
- **get_project(project_id):** Retrieves a project by its ID.
- **get_user(user_id):** Retrieves a user by their ID.

- **Implementation:**

```
import java.util.*;

class ODTDatabase {

    Map<Integer, Project> projects; // Dictionary to store projects by project_id

    Map<Integer, User> users;      // Dictionary to store users by user_id

    ODTDatabase() {

        projects = new HashMap<>();

        users = new HashMap<>();

    }

    void addProject(Project project) {

        projects.put(project.project_id, project);  }

    void addUser(User user) {

        users.put(user.user_id, user);  }

    Project getProject(int project_id) {

        return projects.getDefault(project_id, null);

    }

    User getUser(int user_id) {

        return users.getDefault(user_id, null);}}

}
```

- **Detailed Functionality**

The ODTDatabase class serves as a central repository for managing projects and users within the ODT application. It maintains dictionaries to store projects and users by their respective IDs. The class provides methods to add new projects and users to the database, as well as retrieve project and user information by their IDs.

- **Data Structures Used**

The ODTDatabase class uses HashMap data structures to store projects and users. This allows for fast retrieval and lookup operations based on unique identifiers.

▪ **Future Enhancements for Data Structures**

• **Enhanced Collaboration Features**

Introduce real-time collaboration capabilities, such as concurrent editing and live chat functionality, to further enhance user collaboration.

• **Advanced Diagramming Tools**

Integrate advanced diagramming tools and templates to support a broader range of diagram types and customization options.

• **Enhanced Security Measures**

Implement additional security measures, such as two-factor authentication and data encryption, to further safeguard user data and privacy.

• **Performance Optimization**

Continuously optimize data structures and algorithms to improve application performance, scalability, and responsiveness.

• **Integration with External Tools**

Provide seamless integration with external software development tools and platforms, such as version control systems and project management tools, to enhance productivity and workflow integration.

By incorporating these enhancements, the ODT application can evolve into a robust and feature-rich platform for software design and collaboration.

Algorithms for Online Design Tool (ODT)

▪ Purpose and Overview of Algorithms in ODT

- **Purpose:**

Algorithms are critical components in the Online Design Tool (ODT) for efficiently managing user interactions, project data, diagram structures, and collaborative functionalities. They ensure the system performs its tasks accurately, securely, and efficiently.

- **Overview:**

This section covers the core algorithms implemented in the ODT, detailing their roles, how they operate, and their significance in the overall architecture. These algorithms handle tasks such as user authentication, project management, diagram manipulation, and collaborative editing.

▪ User Authentication and Authorization Algorithms

- **Password Hashing**

Purpose: To securely store user passwords in the database.

Algorithm:

- **Input:** Plaintext password.
- **Process:**
 - Generate a salt value.
 - Combine the salt with the plaintext password.
 - Apply a cryptographic hash function (e.g., SHA-256) to the combined value.
- **Output:** Store the resulting hash and salt in the database.

Implementation:

```
String hashPassword(String password) {  
    String salt = generateSalt();  
    return hashFunction(password + salt) + ":" + salt;  
}
```

- **Login Authentication**

Purpose: To verify user credentials during login.

Algorithm:

- **Input:** Entered email and password.
- **Process:**
 - Retrieve the stored hash and salt for the given email.
 - Combine the entered password with the stored salt.
 - Hash the combined value.
 - Compare the computed hash with the stored hash.
- **Output:** Boolean indicating success or failure.

Implementation:

```
boolean authenticate(String email, String enteredPassword) {  
    String storedHash = getStoredHashByEmail(email);  
    String storedSalt = getStoredSaltByEmail(email);  
    String computedHash = hashFunction(enteredPassword + storedSalt);  
    return storedHash.equals(computedHash);  
}
```

- **Permission Checking**

Purpose: To ensure users have the correct permissions to access or modify a project.

Algorithm:

- **Input:** User ID, Project ID, Required Role.

- **Process:**
Retrieve the user's role for the project from the Permissions table.
Check if the retrieved role matches or exceeds the required role.
- **Output:** Boolean indicating permission status.

Implementation:

```
boolean hasPermission(int userId, int projectId, String requiredRole) {  
    String userRole = getUserRoleForProject(userId, projectId);  
    return roleHierarchy(userRole) >= roleHierarchy(requiredRole);  
}
```

▪ Project Management Algorithms

• Project Creation

Purpose: To create a new project for a user.

Algorithm:

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

Implementation:

```
int createProject(String name, String description, int userId) {  
    int projectId = generateUniqueId();  
    Instant now = Instant.now();  
    insertIntoProjectsTable(projectId, name, description, userId, now, now);  
    return projectId;  
}
```

- **Project Retrieval**

Purpose: To fetch details of a specific project.

Algorithm:

- **Input:** Project ID.
- **Process:**
Query the Projects table for the given project ID.
- **Output:** Project details.

Implementation:

```
Project getProject(int projectId) {  
    return queryProjectsTableById(projectId);  
}
```

- **Project Deletion**

Purpose: To remove a project and its associated data.

Algorithm:

- **Input:** Project ID.
- **Process:**
Delete all diagrams associated with the project.
Delete the project record from the Projects table.
- **Output:** Boolean indicating success or failure.

Implementation:

```
boolean deleteProject(int projectId) {  
    deleteDiagramsByProjectId(projectId);  
    return deleteFromProjectsTable(projectId);  
}
```

▪ Diagram Management Algorithms

• 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;  
}
```

• Diagram Retrieval

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:

```
Diagram getDiagram(int diagramId) {  
    return queryDiagramsTableById(diagramId);  
}
```

- **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);  
}
```

- **Element Management Algorithms**

- **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.
- Insert a new element record into the DiagramElements table.

- **Output:** Element ID of the newly added element.

Implementation:

```
int addElement(String type, Map<String, String> properties, int diagramId) {  
    int elementId = generateUniquelId();  
    Instant now = Instant.now();  
    insertIntoDiagramElementsTable(elementId, diagramId, type, properties, now, now);  
    return elementId;  
}
```

- **Element Retrieval**

Purpose: To fetch details of a specific element.

Algorithm:

- **Input:** Element ID.
- **Process:**
Query the DiagramElements table for the given element ID.
- **Output:** Element details.

Implementation:

```
Element getElement(int elementId) {  
    return queryDiagramElementsTableById(elementId);  
}
```

- **Element Deletion**

Purpose: To remove an element from a diagram.

Algorithm:

- **Input:** Element ID.
- **Process:**

Delete the element record from the DiagramElements table.

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

Implementation:

```
boolean deleteElement(int elementId) {  
    return deleteFromDiagramElementsTable(elementId);  
}
```

▪ Collaboration Management Algorithms

- **Assigning Roles**

Purpose: To assign a user a specific role in a project.

Algorithm:

- **Input:** User ID, Project ID, Role.
- **Process:**
 - Generate a unique collaboration ID.
 - Set the current timestamp as the creation time.
 - Insert a new collaboration record into the Permissions table.
- **Output:** Collaboration ID of the newly created entry.

Implementation:

```
int assignRole(int userId, int projectId, String role) {  
    int collaborationId = generateUniqueId();  
    Instant now = Instant.now();  
    insertIntoPermissionsTable(collaborationId, projectId, userId, role, now);  
    return collaborationId;  
}
```

- **Role Verification**

Purpose: To verify a user's role in a project.

Algorithm:

- **Input:** User ID, Project ID.
- **Process:**
Query the Permissions table for the user's role in the given project.
- **Output: Role of the user.**

Implementation:

```
String getRole(int userId, int projectId) {  
    return queryPermissionsTable(userId, projectId);  
}
```

▪ Comment Management Algorithms

• 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 = generateUniquelId();  
    Instant now = Instant.now();  
    insertIntoCommentsTable(commentId, elementId, userId, content, now, now);  
    return commentId; }
```

- **Retrieving Comments**

Purpose: To fetch comments for a specific diagram element.

Algorithm:

- **Input:** Element ID.
- **Process:**
Query the Comments table for all comments associated with the element ID.
- **Output:** List of comments.

Implementation:

```
List<Comment> getComments(int elementId) {  
    return queryCommentsTableByElementId(elementId);  
}
```

- **Deleting Comments**

Purpose: To remove a comment from a 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);  
}
```

▪ Data Validation Algorithms

• Input Validation

Purpose: To ensure that all user inputs are properly validated to prevent errors and security vulnerabilities.

Algorithm:

- Check for null or empty values.
- Validate data types.
- Ensure values are within acceptable ranges.
- Sanitize inputs to prevent SQL injection.

Implementation:

```
public class InputValidator {  
    public static boolean isValidEmail(String email) {  
        return email != null && email.matches("[A-Za-z0-9+_.-]+@(.+)$");  
    }  
  
    public static boolean isValidUsername(String username) {  
        return username != null && username.length() >= 3 && username.length() <= 20;  
    }  
  
    public static boolean isValidPassword(String password) {  
        return password != null && password.length() >= 8;  
    }  
    public static boolean validateProjectName(String projectName) {  
        return projectName != null && !projectName.trim().isEmpty();  
    }  
}
```

• Data Integrity Checks

Purpose: To ensure the integrity and consistency of data within the database.

Algorithm:

- Ensure foreign key constraints are maintained.
- Ensure unique constraints are enforced.

Implementation:

```
ALTER TABLE Projects
ADD CONSTRAINT fk_user
FOREIGN KEY (owner_id) REFERENCES Users(user_id);

ALTER TABLE Users
ADD CONSTRAINT unique_email
UNIQUE (email);
```

▪ Notification Algorithms

• Event Triggered Notifications

Purpose: To notify users of significant events within the ODT.

Algorithm:

- Detect specific events (e.g., new comment added, project shared).
- Send notifications to relevant users.

Implementation:

```
public class NotificationService {
    public void sendNotification(int userId, String message) {
        // Code to send notification (e.g., email, in-app)
        System.out.println("Notification sent to user " + userId + ": " + message);
    }

    public void notifyNewComment(int elementId, int commenterId) {
```

```

List<Integer> usersToNotify = getUsersToNotify(elementId);
for (int userId : usersToNotify) {
    if (userId != commenterId) {
        sendNotification(userId, "A new comment was added to your diagram element.");
    }
}
}
}

```

- **Batch Notifications**

Purpose: To send batch notifications for periodic updates.

Algorithm:

- Schedule batch notification jobs.
- Collect and send updates to users.

Implementation:

```

import java.util.Timer;
import java.util.TimerTask;

public class BatchNotificationService {
    Timer timer = new Timer();
    public void scheduleBatchNotifications() {
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                sendBatchNotifications();
            }
        }, 0, 86400000); // Schedule for every 24 hours
    }
    public void sendBatchNotifications() {
        List<User> users = getAllUsers();
    }
}

```

```

    for (User user : users) {
        String message = "Daily summary of your project updates.";
        sendNotification(user.userId, message);
    }
}
}

```

▪ Security Algorithms

• Encryption

Purpose: To protect sensitive data stored in the database.

Algorithm:

- Encrypt data before storing it.
- Decrypt data when retrieving it.

Implementation:

```

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class EncryptionUtil {
    private static final String ALGORITHM = "AES";

    public static String encrypt(String data, String key) throws Exception {
        SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), ALGORITHM);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encryptedData = cipher.doFinal(data.getBytes());
        return Base64.getEncoder().encodeToString(encryptedData);
    }
}

```

```

public static String decrypt(String encryptedData, String key) throws Exception {
    SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), ALGORITHM);
    Cipher cipher = Cipher.getInstance(ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, secretKey);
    byte[] decryptedData = cipher.doFinal(Base64.getDecoder().decode(encryptedData));
    return new String(decryptedData);
}
}

```

- **Two-Factor Authentication (2FA)**

Purpose: To enhance user account security.

Algorithm:

- Generate a 2FA token.
- Send token to the user's registered device.
- Verify token upon login.

Implementation:

```

import java.util.Random;

public class TwoFactorAuth {
    private static final int TOKEN_LENGTH = 6;
    public static String generateToken() {
        Random random = new Random();
        StringBuilder token = new StringBuilder(TOKEN_LENGTH);
        for (int i = 0; i < TOKEN_LENGTH; i++) {
            token.append(random.nextInt(10)); // generate a random digit
        }
        return token.toString();
    }
    public static boolean verifyToken(String inputToken, String correctToken) {

```



```
        return inputToken.equals(correctToken);
    }
}
```

▪ Backup and Recovery Algorithms

• Automated Backups

Purpose: To ensure regular backups of the system data.

Algorithm:

- Schedule regular backups.
- Copy data to a backup location.

Implementation:

```
import java.util.Timer;
import java.util.TimerTask;

public class BackupService {
    Timer timer = new Timer();

    public void scheduleBackups() {
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                performBackup();
            }
        }, 0, 604800000); // Schedule for every 7 days
    }

    public void performBackup() {
        // Code to copy data to backup location
        System.out.println("Backup performed successfully.");
    }
}
```

- **Data Recovery**

Purpose: To restore data in the event of a failure.

Algorithm:

- Identify the backup to restore.
- Restore data from the backup location.

Implementation:

```
public class DataRecoveryService {  
    public void recoverData(String backupLocation) {  
        // Code to restore data from backup  
        System.out.println("Data restored from " + backupLocation);  
    }  
}
```

- **Logging and Monitoring Algorithms**

- **Activity Logging**

Purpose: To keep track of user activities for auditing and troubleshooting.

Algorithm:

- Log significant user actions.
- Store logs in a database or file system.

Implementation:

```
public class ActivityLogger {  
    public void logActivity(int userId, String activity) {  
        String logEntry = "User " + userId + " performed: " + activity + " at " + Instant.now();  
        saveLog(logEntry);  
    }  
}
```

```

    }

    private void saveLog(String logEntry) {
        // Code to save log entry (e.g., write to file or database)
        System.out.println(logEntry);
    }
}

```

- **System Monitoring**

Purpose: To monitor the system's performance and health.

Algorithm:

- Collect performance metrics.
- Analyze and report system health.

Implementation:

```

public class SystemMonitor {
    public void collectMetrics() {
        // Code to collect system metrics (e.g., CPU, memory usage)
        System.out.println("Collecting system metrics...");
    }

    public void analyzeHealth() {
        // Code to analyze collected metrics
        System.out.println("System health is good.");
    }
}

```

- **Version Control Algorithms**

- **Document Versioning**

Purpose: To manage changes to documents and diagrams.

Algorithm:

- Track versions of documents.
- Maintain a history of changes.

Implementation:

```
public class VersionControl {  
    private Map<Integer, List<DocumentVersion>> documentVersions = new HashMap<>();  
    public void addNewVersion(int documentId, String content) {  
        DocumentVersion version = new DocumentVersion(content, Instant.now());  
        documentVersions.computeIfAbsent(documentId, k -> new ArrayList<>()).add(version);  
    }  
  
    public List<DocumentVersion> getVersionHistory(int documentId) {  
        return documentVersions.getDefault(documentId, new ArrayList<>());  
    }  
}  
  
class DocumentVersion {  
    String content;  
    Instant timestamp;  
    DocumentVersion(String content, Instant timestamp) {  
        this.content = content;  
        this.timestamp = timestamp;  
    }  
}
```

- **Change Tracking**

Purpose: To track changes made to projects and diagrams.

Algorithm:

- Log changes with timestamps.
- Maintain change history.

Implementation:

```
public class ChangeTracker {  
    private List<ChangeLog> changeLogs = new ArrayList<>();  
  
    public void logChange(int projectId, String changeDescription) {  
        ChangeLog log = new ChangeLog(projectId, changeDescription, Instant.now());  
        changeLogs.add(log);  
    }  
  
    public List<ChangeLog> getChangeHistory(int projectId) {  
        return changeLogs.stream()  
            .filter(log -> log.projectId == projectId)  
            .collect(Collectors.toList());  
    }  
}  
  
class ChangeLog {  
    int projectId;  
    String description;  
}
```

▪ Optimization and Performance Considerations

- **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.

- **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.

▪ **Future Enhancements**

- **Machine Learning Integration:** Implement machine learning algorithms to provide intelligent suggestions for diagram improvements and error detection.
- **Real-time Collaboration:** Develop algorithms to support real-time collaboration features, including live updates and conflict resolution.
- **Advanced Search:** Enhance search capabilities with advanced algorithms for natural language processing and semantic search.
- **Automated Backups:** Implement algorithms for automated backup and recovery to ensure data safety and availability.

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.

Database for Online Design Tool (ODT)

▪ Purpose of Database in ODT

The database will store user information, projects, diagrams, diagram elements, comments, and permissions. It aims to ensure data integrity, consistency, and security while providing efficient access and modifications.

▪ Requirement Analysis

➤ Functional Requirements:

- **User Management:** Registration, authentication, profile management.
- **Project Management:** Create, read, update, and delete projects.
- **Diagram Management:** Create, edit, and delete diagrams within projects.
- **Element Management:** Add, edit, and delete elements in diagrams.
- **Commenting:** Add, edit, and delete comments on diagram elements.
- **Permissions:** Define user roles and access controls.

➤ Non-Functional Requirements:

- **Scalability:** Supports a growing number of users and projects.
- **Performance:** Quick response times for database queries.
- **Security:** Protect user data and ensure only authorized access.
- **Reliability:** Ensure data integrity and consistency.

➤ User Roles and Permissions

- **Admin:** Full access to all functionalities.
- **User:** Access to own projects and collaborative projects.
- **Viewer:** Read-only access to specific projects.

▪ Database Design

Designing a database for an online design tool involves several key entities and their relationships. In our ODT, we have implemented 6 tables in the database:

1. Users Table:

This table stores information about the users of the design tool. This is a foundational entity, as every action within the tool is associated with a user. The associated attributes are:

- **user_id:** A unique identifier for each user, set as the primary key.
- **username:** The display name of the user.
- **email:** The email address of the user, must be unique.
- **password_hash:** A hash of the user's password for secure authentication.
- **created_at:** The time when the user account was created.
- **updated_at:** The time when the user account was last updated.

```
1 CREATE TABLE Users (  
2     user_id INT PRIMARY KEY,  
3     username VARCHAR(50) NOT NULL,  
4     email VARCHAR(100) NOT NULL UNIQUE,  
5     password_hash VARCHAR(255) NOT NULL,  
6     created_at DATETIME,  
7     updated_at DATETIME  
8 );  
9 INSERT INTO Users (user_id, username, email, password_hash, created_at, updated_at) VALUES  
10 ('1', 'samar', 'samarqaiser04@gmail.com', 'pass1sam', '2024-05-01 10:00:00', '2024-05-01 10:00:00'),  
11 ('2', 'shandana', 'shandanaiftikhar20@yahoo.com', 'pass2shan', '2024-05-02 11:00:00', '2024-05-02 11:00:00'),  
12 ('3', 'ayesha', 'ayesha_israr@gmail.com', 'pass3ayesh', '2024-05-03 12:00:00', '2024-05-03 12:00:00'),  
13 ('4', 'ahmed', 'ahmed.sultan@gmail.com', 'pass4ahm', '2024-05-04 13:00:00', '2024-05-04 13:00:00'),  
14 ('5', 'fatima', 'fatimashaheen122@hotmail.com', 'pass5fat', '2024-05-05 14:00:00', '2024-05-05 14:00:00');  
15 SELECT* FROM Users;
```

Results Messages

	user_id	username	email	password_hash	created_at	updated_at
1	1	samar	samarqaiser04@gmail.com	pass1sam	2024-05-01 10:00:00.000	2024-05-01 10:00:00.000
2	2	shandana	shandanaiftikhar20@yahoo.com	pass2shan	2024-05-02 11:00:00.000	2024-05-02 11:00:00.000
3	3	ayesha	ayesha_israr@gmail.com	pass3ayesh	2024-05-03 12:00:00.000	2024-05-03 12:00:00.000
4	4	ahmed	ahmed.sultan@gmail.com	pass4ahm	2024-05-04 13:00:00.000	2024-05-04 13:00:00.000
5	5	fatima	fatimashaheen122@hotmail.com	pass5fat	2024-05-05 14:00:00.000	2024-05-05 14:00:00.000

2. Projects Table:

A project is a container for diagrams. This table contains information about the projects created by users. The associated attributes are:

- **project_id**: A unique identifier for each project, set as the primary key.
- **name**: The name of the project.
- **description**: A detailed description of the project.
- **owner_id**: The user_id of the project's owner, establishing a relationship with the Users table (foreign key).
- **created_at**: The time when the project was created.
- **updated_at**: The time when the project was last updated.

```
16 CREATE TABLE Projects (  
17     project_id INT PRIMARY KEY,  
18     name VARCHAR(100) NOT NULL,  
19     description TEXT,  
20     owner_id INTEGER NOT NULL,  
21     created_at DATETIME,  
22     updated_at DATETIME,  
23     FOREIGN KEY (owner_id) REFERENCES Users(user_id)  
24 );  
25 INSERT INTO Projects (project_id, name, description, owner_id, created_at, updated_at) VALUES  
26 ('1', 'Project Alpha', 'A project about Alpha.', '1', '2024-05-01 10:30:00', '2024-05-01 10:30:00'),  
27 ('2', 'Project Beta', 'A project about Beta.', '2', '2024-05-02 11:30:00', '2024-05-02 11:30:00'),  
28 ('3', 'Project Charlie', 'A project about Charlie.', '3', '2024-05-03 12:30:00', '2024-05-03 12:30:00'),  
29 ('4', 'Project Gamma', 'A project about Gamma.', '4', '2024-05-04 13:30:00', '2024-05-04 13:30:00'),  
30 ('5', 'Project Delta', 'A project about Delta.', '5', '2024-05-05 14:30:00', '2024-05-05 14:30:00');  
31 SELECT* FROM Projects;  
32
```

Results Messages

	project_id	name	description	owner_id	created_at	updated_at
1	1	Project Alpha	A project about Alpha.	1	2024-05-01 10:30:00.000	2024-05-01 10:30:00.000
2	2	Project Beta	A project about Beta.	2	2024-05-02 11:30:00.000	2024-05-02 11:30:00.000
3	3	Project Charlie	A project about Charlie.	3	2024-05-03 12:30:00.000	2024-05-03 12:30:00.000
4	4	Project Gamma	A project about Gamma.	4	2024-05-04 13:30:00.000	2024-05-04 13:30:00.000
5	5	Project Delta	A project about Delta.	5	2024-05-05 14:30:00.000	2024-05-05 14:30:00.000

3. 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

4. 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.

```

47 CREATE TABLE DiagramElements (
48     element_id INT PRIMARY KEY,
49     diagram_id INTEGER NOT NULL,
50     type VARCHAR(50) NOT NULL,
51     properties TEXT,
52     position_x FLOAT,
53     position_y FLOAT,
54     created_at DATETIME,
55     updated_at DATETIME,
56     FOREIGN KEY (diagram_id) REFERENCES Diagrams(diagram_id)
57 );
58 INSERT INTO DiagramElements (element_id, diagram_id, type, properties, position_x, position_y, created_at, updated_at) VALUES
59 ('1', '1', 'Class', '{"name": "Class1"}', '100', '100', '2024-05-01 11:00:00', '2024-05-01 11:00:00'),
60 ('2', '1', 'Class', '{"name": "Class2"}', '200', '200', '2024-05-01 11:10:00', '2024-05-01 11:10:00'),
61 ('3', '2', 'Interface', '{"name": "Interface1"}', '150', '150', '2024-05-02 12:00:00', '2024-05-02 12:00:00'),
62 ('4', '3', 'Entity', '{"name": "Entity1"}', '250', '250', '2024-05-03 13:00:00', '2024-05-03 13:00:00'),
63 ('5', '4', 'Relationship', '{"name": "Relationship1"}', '300', '300', '2024-05-04 14:00:00', '2024-05-04 14:00:00');
64 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

5. 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

6. Permissions Table:

This table manages user access and roles for each project, ensuring that users have appropriate access rights. The associated attributes are:

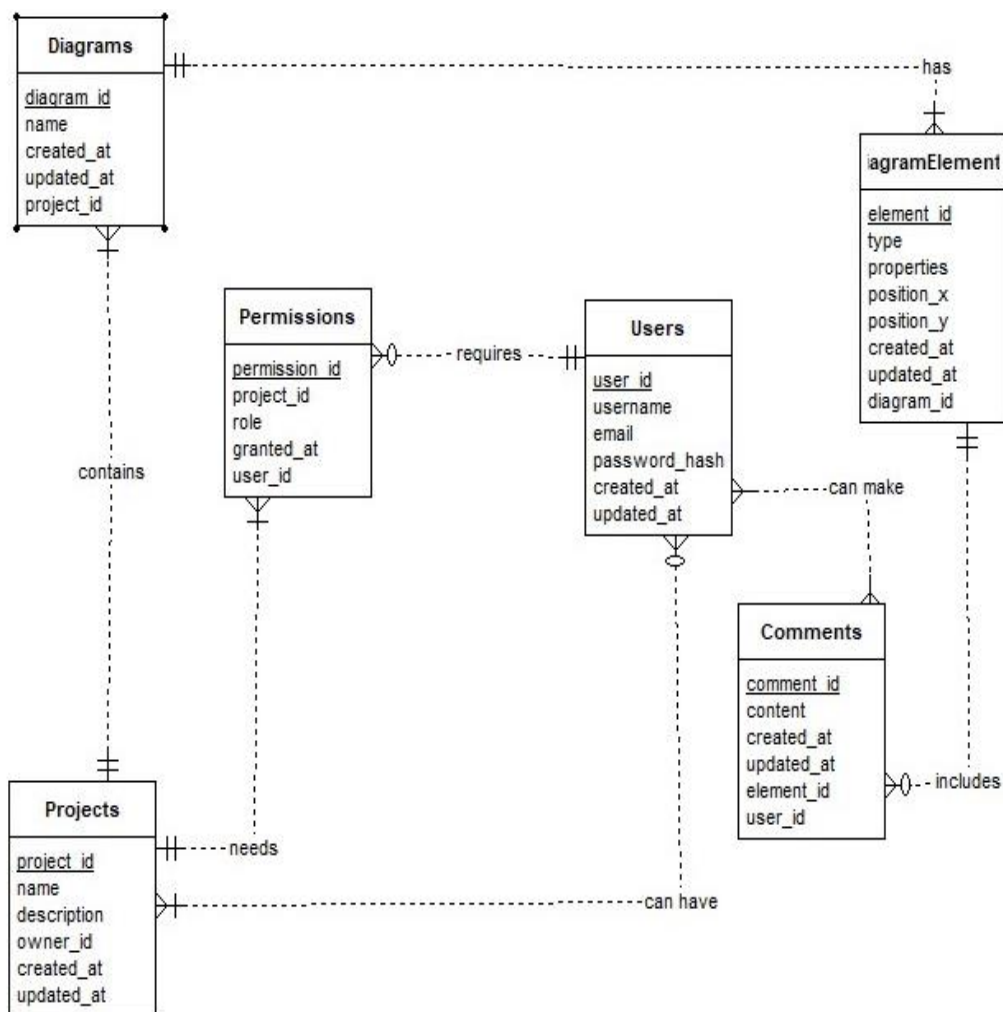
- **permission_id**: A unique identifier for each permission entry, set as the primary key.
- **project_id**: The project_id of the project to which the permission applies, establishing a relationship with the Projects table (foreign key).
- **user_id**: The user_id of the user who has the permission, creating a relationship with the Users table (foreign key).
- **role**: The role assigned to the user (e.g., owner, editor, viewer).
- **granted_at**: The timestamp when the permission was granted.

```
82 CREATE TABLE Permissions (  
83     permission_id INT PRIMARY KEY,  
84     project_id INTEGER NOT NULL,  
85     user_id INTEGER NOT NULL,  
86     role VARCHAR(20) NOT NULL,  
87     granted_at DATETIME,  
88     FOREIGN KEY (project_id) REFERENCES Projects(project_id),  
89     FOREIGN KEY (user_id) REFERENCES Users(user_id)  
90 );  
91 INSERT INTO Permissions (permission_id, project_id, user_id, role, granted_at) VALUES  
92 ('1', '1', '1', 'owner', '2024-05-01 10:30:00'),  
93 ('2', '2', '2', 'partner', '2024-05-02 11:30:00'),  
94 ('3', '3', '3', 'owner', '2024-05-03 12:30:00'),  
95 ('4', '4', '4', 'owner', '2024-05-04 13:30:00'),  
96 ('5', '5', '5', 'partner', '2024-05-05 14:30:00');  
97 SELECT* FROM Permissions;
```

Results Messages

	permission_id	project_id	user_id	role	granted_at
1	1	1	1	owner	2024-05-01 10:30:00.000
2	2	2	2	partner	2024-05-02 11:30:00.000
3	3	3	3	owner	2024-05-03 12:30:00.000
4	4	4	4	owner	2024-05-04 13:30:00.000
5	5	5	5	partner	2024-05-05 14:30:00.000

▪ Entity Relationship Diagram (ERD)



▪ Relationships Between Entities

• Users to Projects: Many-to-Many relationship

Many users can own multiple projects as one project may be developed by a group of people. Similarly, a user can own multiple projects. This relationship is captured by the `owner_id` in the `Projects` table referring to the `user_id` in the `Users` table.

• Projects to Diagrams: One-to-Many relationship

Each project can contain multiple diagrams. This relationship is represented by the `project_id` in the `Diagrams` table referring to the `project_id` in the `Projects` table.

• Diagrams to DiagramElements: One-to-Many relationship

Each diagram can contain multiple elements. This relationship is indicated by the `diagram_id` in the `DiagramElements` table referring to the `diagram_id` in the `Diagrams` table.

- **DiagramElements to Comments: One-to-Many relationship**

Each diagram element can have multiple comments. This relationship is denoted by the element_id in the Comments table referring to the element_id in the DiagramElements table.

- **Projects to Permissions: One-to-Many relationship**

Each project can have multiple permissions assigned to different users. This relationship is captured by the project_id in the Permissions table referring to the project_id in the Projects table.

- **Users to Permissions: One-to-Many relationship**

A user can have multiple permissions across different projects. This relationship is indicated by the user_id in the Permissions table referring to the user_id in the Users table.

- **Logical Schema**

Diagrams Table

diagram_id (Primary Key)	name	project_id (Foreign Key)	Created_at	Updated_at
-----------------------------	------	-----------------------------	------------	------------

Projects Table

Project_id (Primary Key)	name	description	Owner_id (Foreign Key user_id)	Created_at	Updated at
-----------------------------	------	-------------	--------------------------------------	------------	------------

Permissions Table

Permission_id (Primary Key)	Project_id (Foreign Key)	role	Granted_at	User_id (Foreign Key)
--------------------------------	-----------------------------	------	------------	--------------------------

Users Table

User_id (Primary Key)	username	email	Password_hash	Created_at	Updated_at
--------------------------	----------	-------	---------------	------------	------------

Comments Table

Comment_id (Primary Key)	content	Created_at	Updated_at	Element_id (Foreign Key)	User_id (Foreign Key)
-----------------------------	---------	------------	------------	-----------------------------	--------------------------

DiagramElement Table

Element_id (Primary Key)	type	properties	Position_x	Position_y	Created_at	Updated_at	Diagram_id (Foreign Key)
-----------------------------	------	------------	------------	------------	------------	------------	-----------------------------

▪ Reasoning behind the Database Design Choices

- **Normalization:** The database is designed to be normalized to avoid data redundancy and ensure data integrity. Each piece of information is stored in its appropriate table, reducing the risk of inconsistent data.
- **Flexibility:** The use of text for the properties field in the DiagramElements table allows for flexible and varied attributes for different types of diagram elements, accommodating various design needs without requiring schema changes.
- **Security:** Storing hashed passwords in the Users table enhances security. Unique constraints on the email field ensure that each user has a unique identifier for authentication.
- **Date Time:** The inclusion of created_at and updated_at fields in all tables helps in tracking the history and updates of records, which is essential for maintaining the integrity of the data.
- **Collaboration:** The Comments and Permissions tables facilitate collaboration among users by allowing them to comment on diagram elements and manage access rights to projects, respectively.

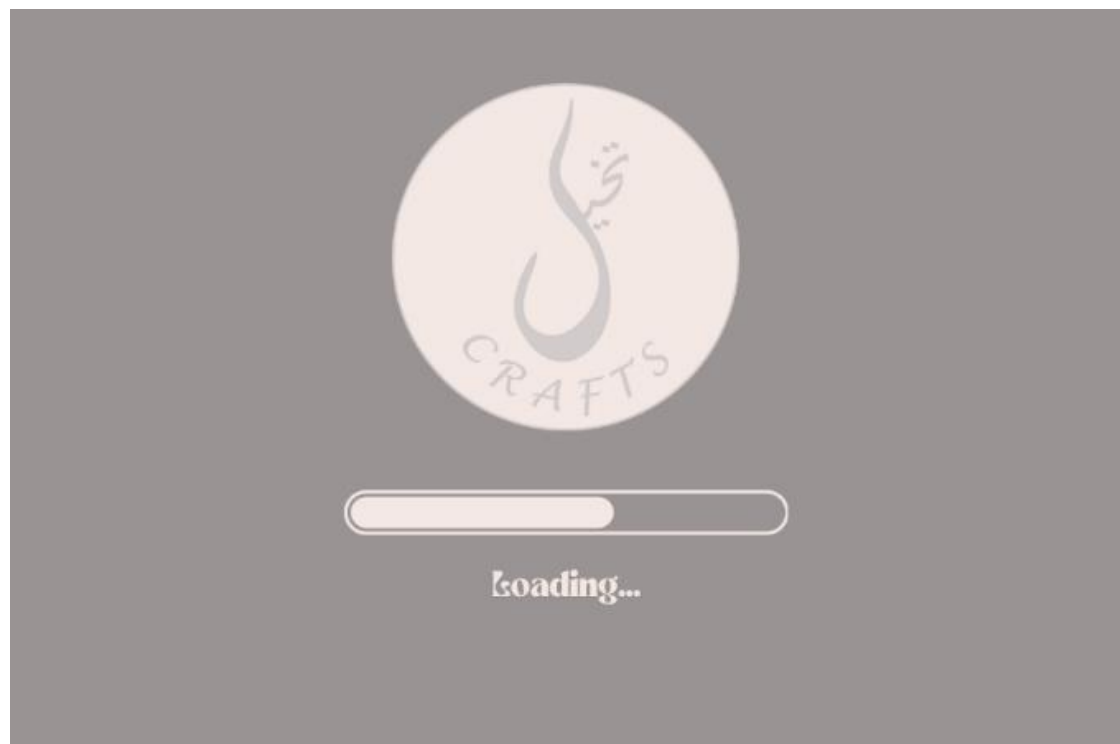
MID LEVEL DESIGN FOR ONLINE DESIGN TOOL (ODT)

- **Graphical User Interfaces (GUI)**

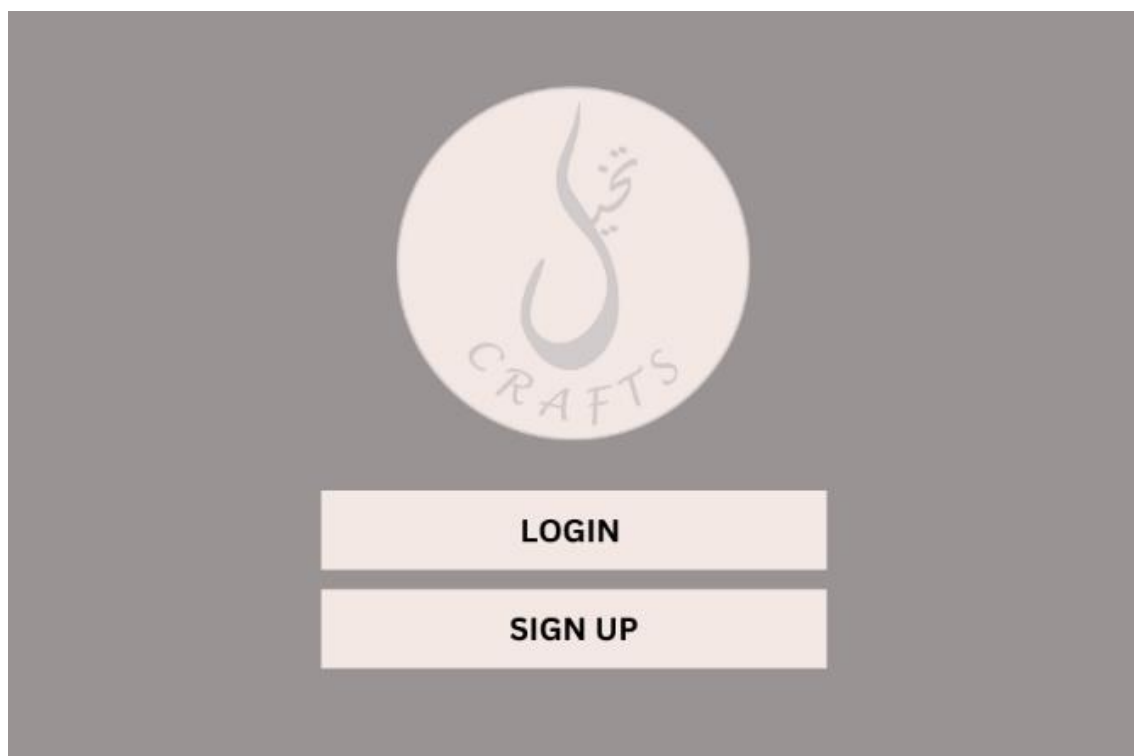
- **LOGO**



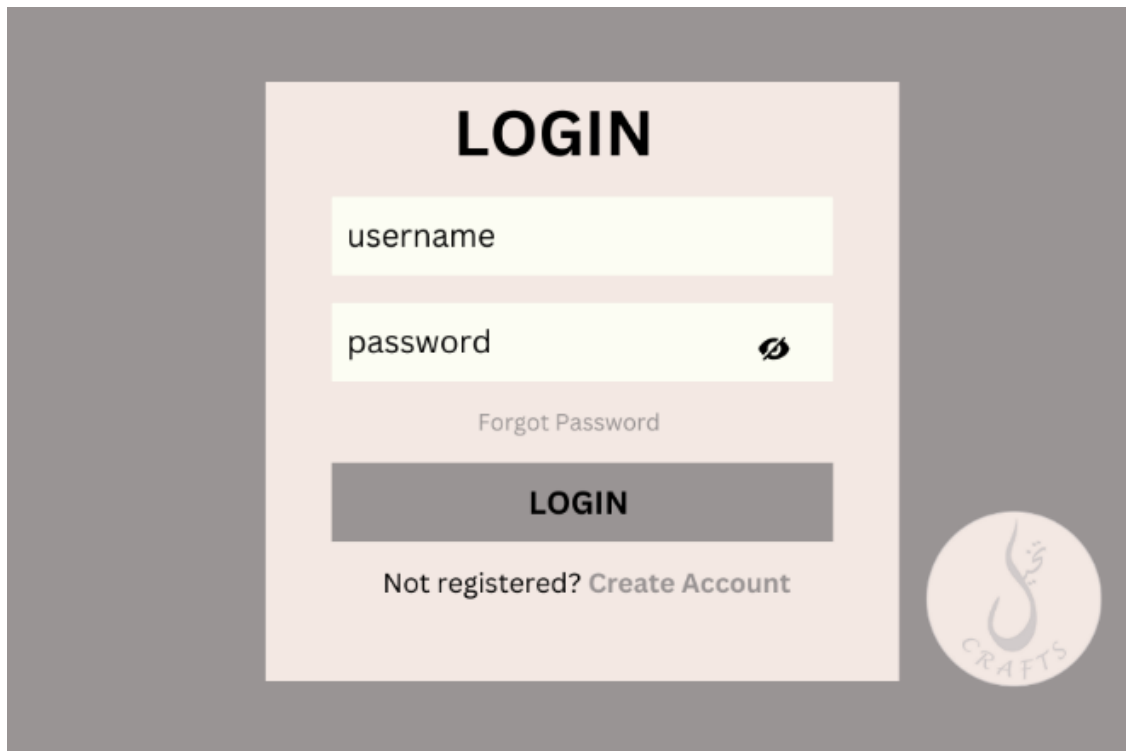
- **SPLASH**



- **LOGIN/SIGNUP**




▪ LOGIN



A login form UI mockup. The form is centered on a light gray background. It has a title "LOGIN" in bold black text. Below the title are two input fields: "username" and "password". The "password" field has a toggle icon (an eye with a slash) to its right. Below the "password" field is a link "Forgot Password". Below that is a dark gray button with the text "LOGIN" in white. Below the button is a link "Not registered? Create Account". To the right of the form is a circular logo with a stylized "S" and the word "CRAFTS" below it.

LOGIN


username

password 

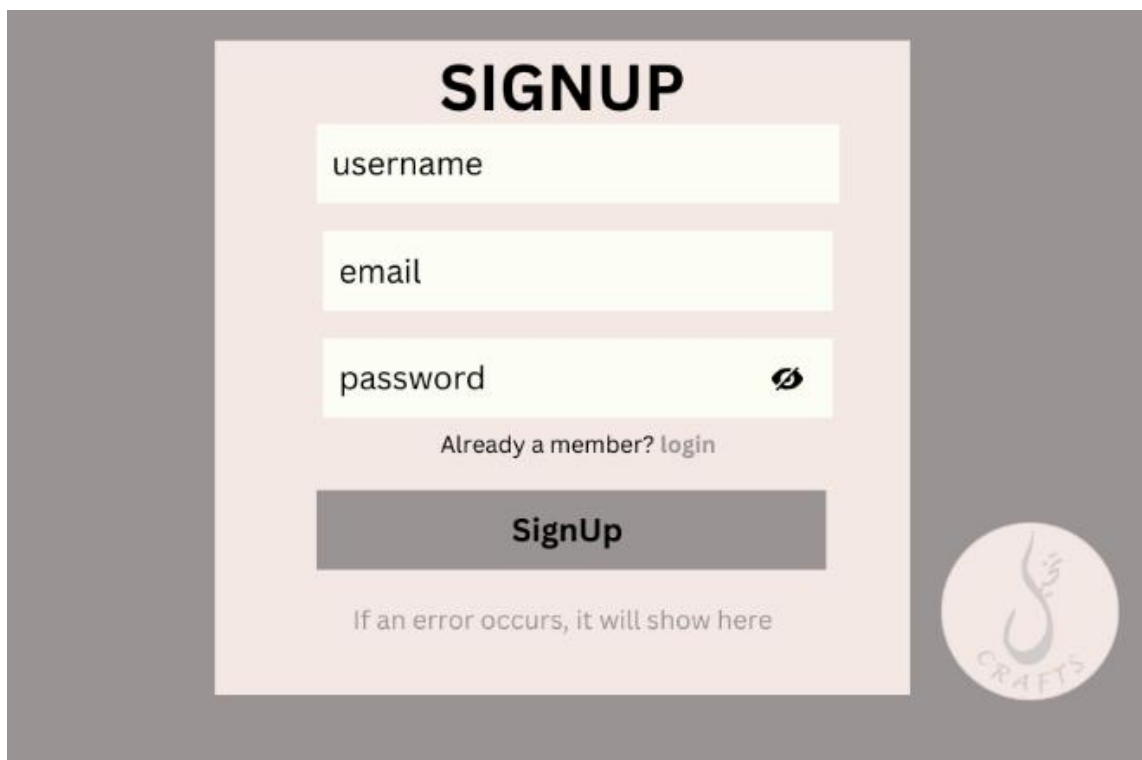
[Forgot Password](#)

LOGIN

[Not registered? Create Account](#)



▪ SIGNUP




A signup form UI mockup. The form is centered on a light gray background. It has a title "SIGNUP" in bold black text. Below the title are three input fields: "username", "email", and "password". The "password" field has a toggle icon (an eye with a slash) to its right. Below the "password" field is a link "Already a member? login". Below that is a dark gray button with the text "SignUp" in white. Below the button is a placeholder text "If an error occurs, it will show here". To the right of the form is a circular logo with a stylized "S" and the word "CRAFTS" below it.

SIGNUP

username


email

password 

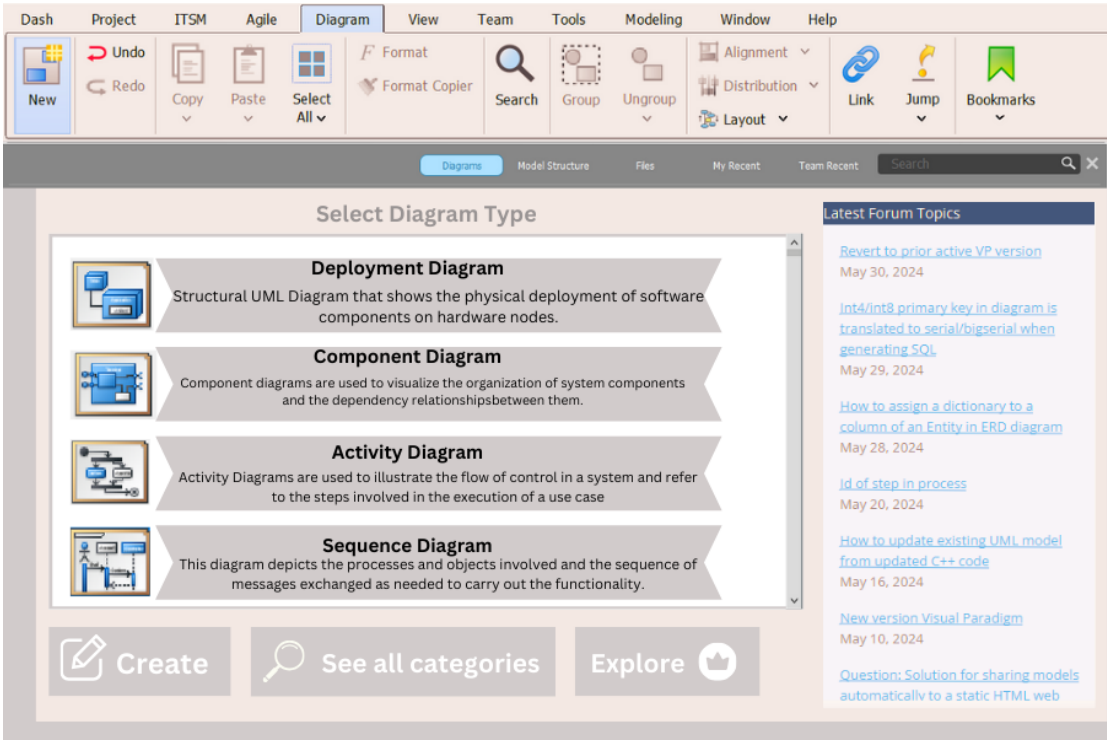
[Already a member? login](#)

SignUp

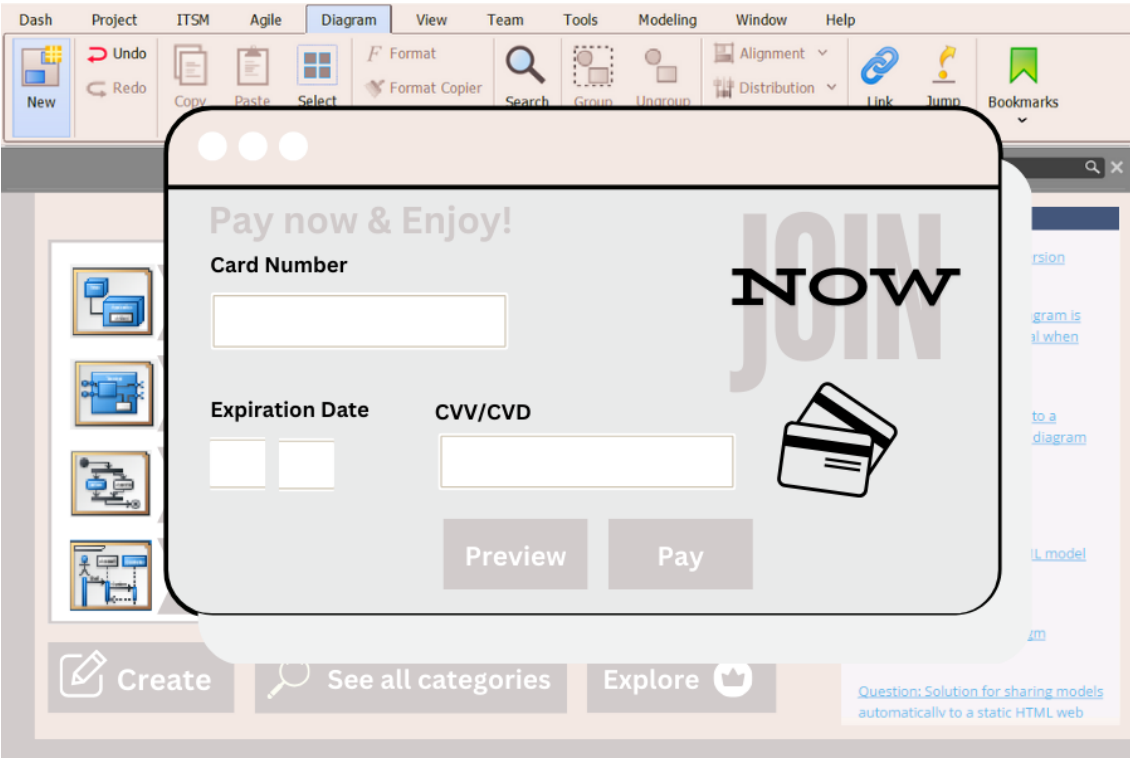
If an error occurs, it will show here



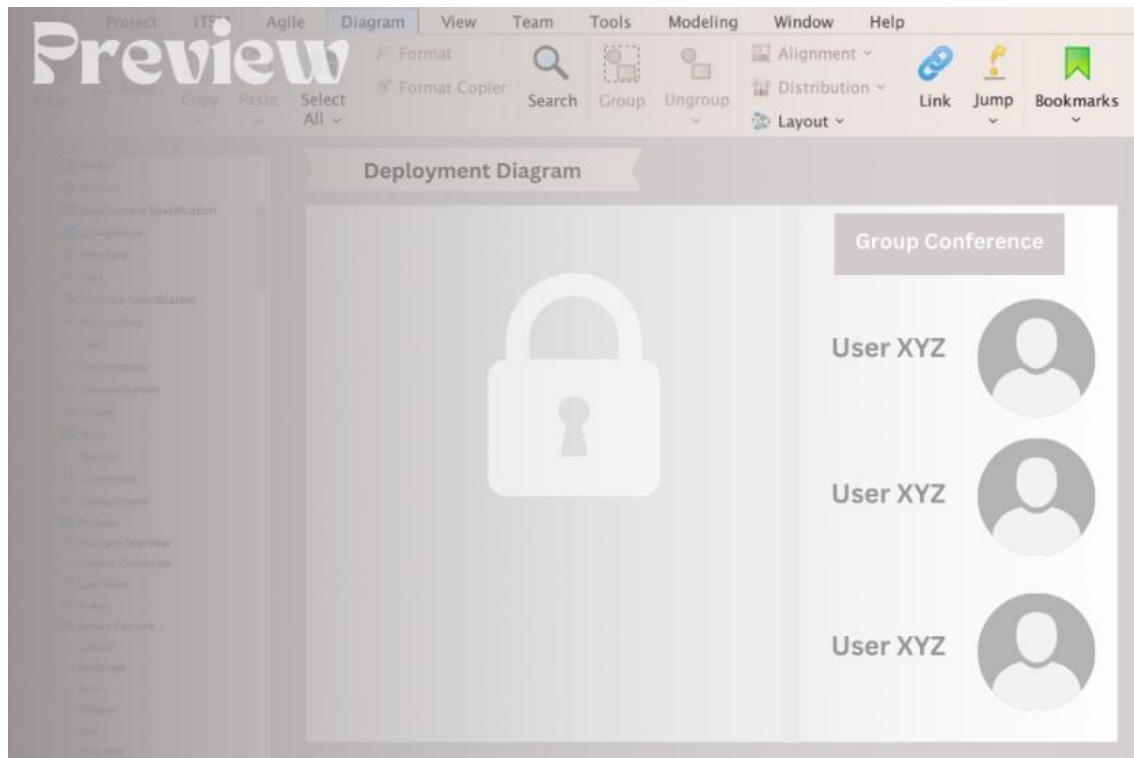
■ DASHBOARD



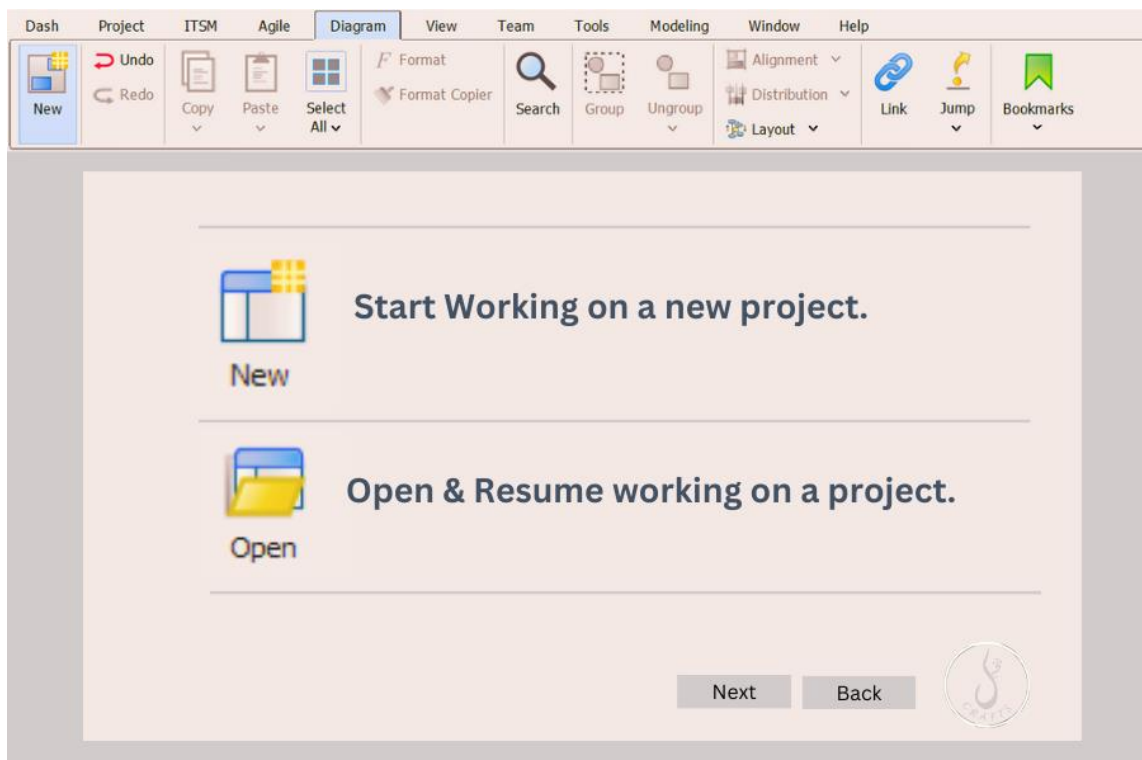
■ BILLING



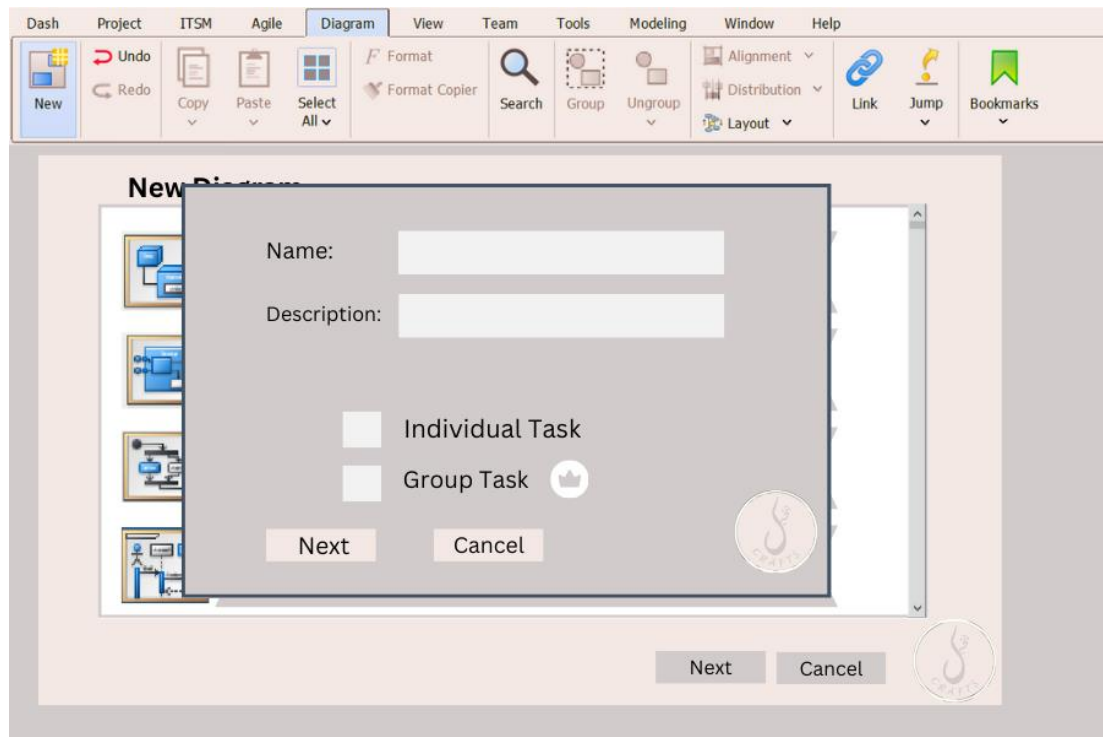
■ PREVIEW



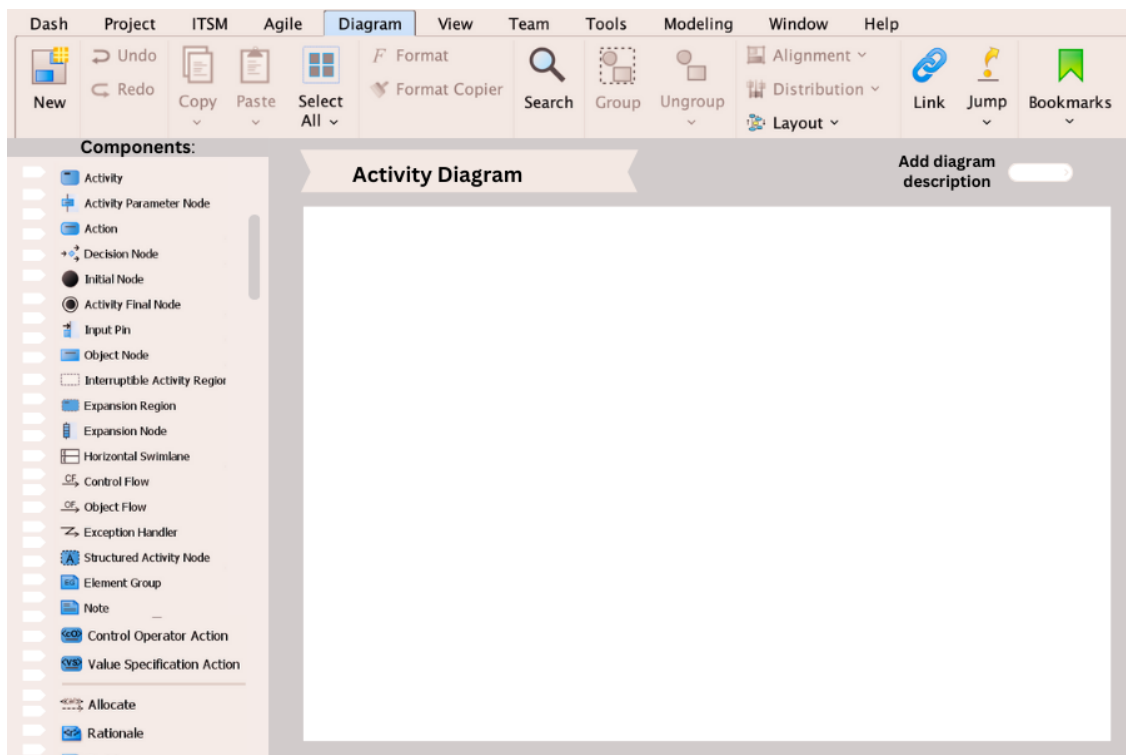
■ NEW/OPEN PROJECT



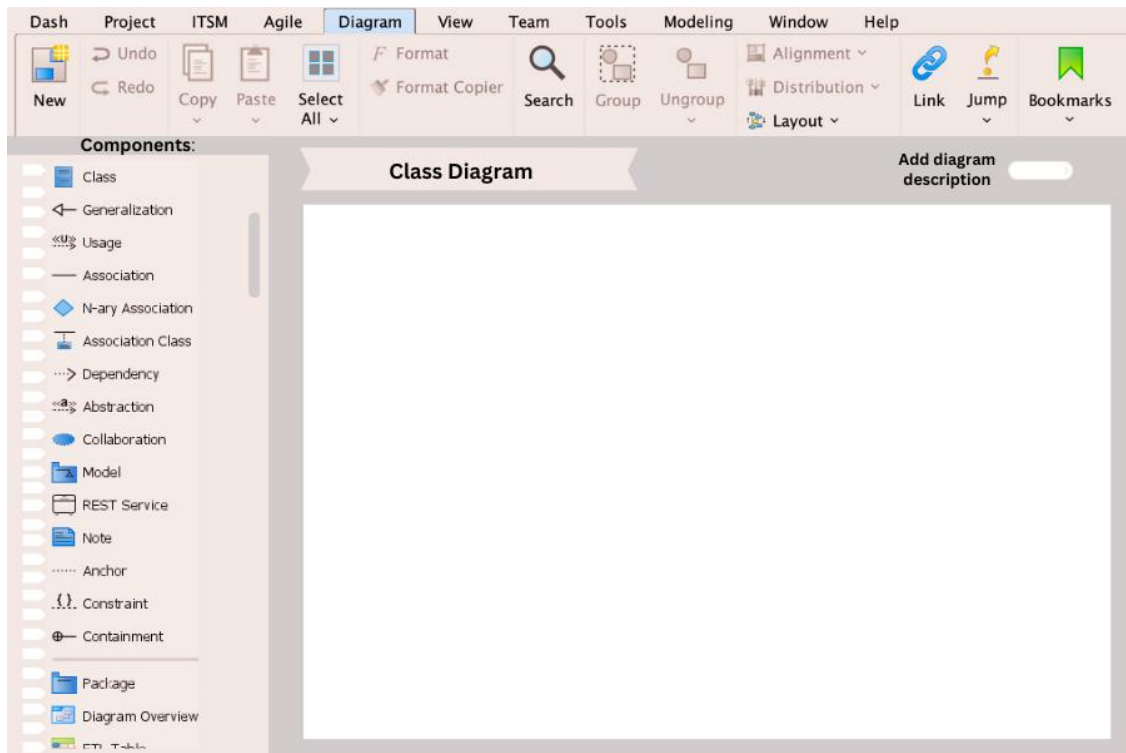
■ PROJECT INFO



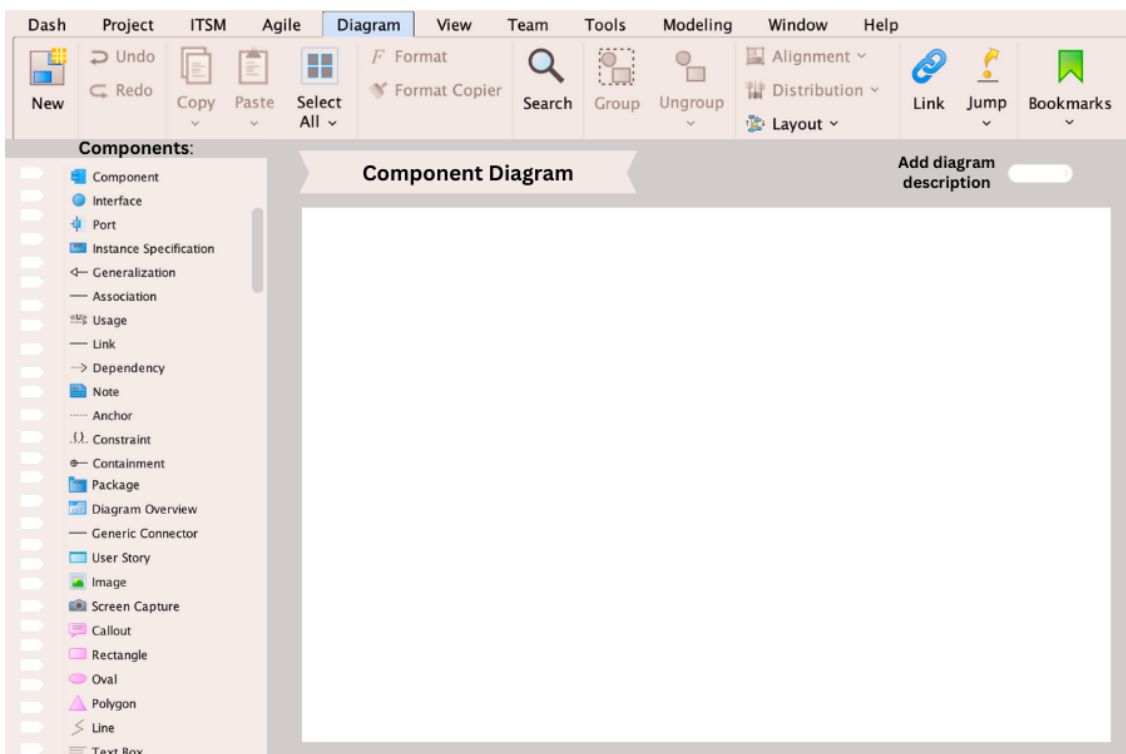
■ ACTIVITY DIAGRAM



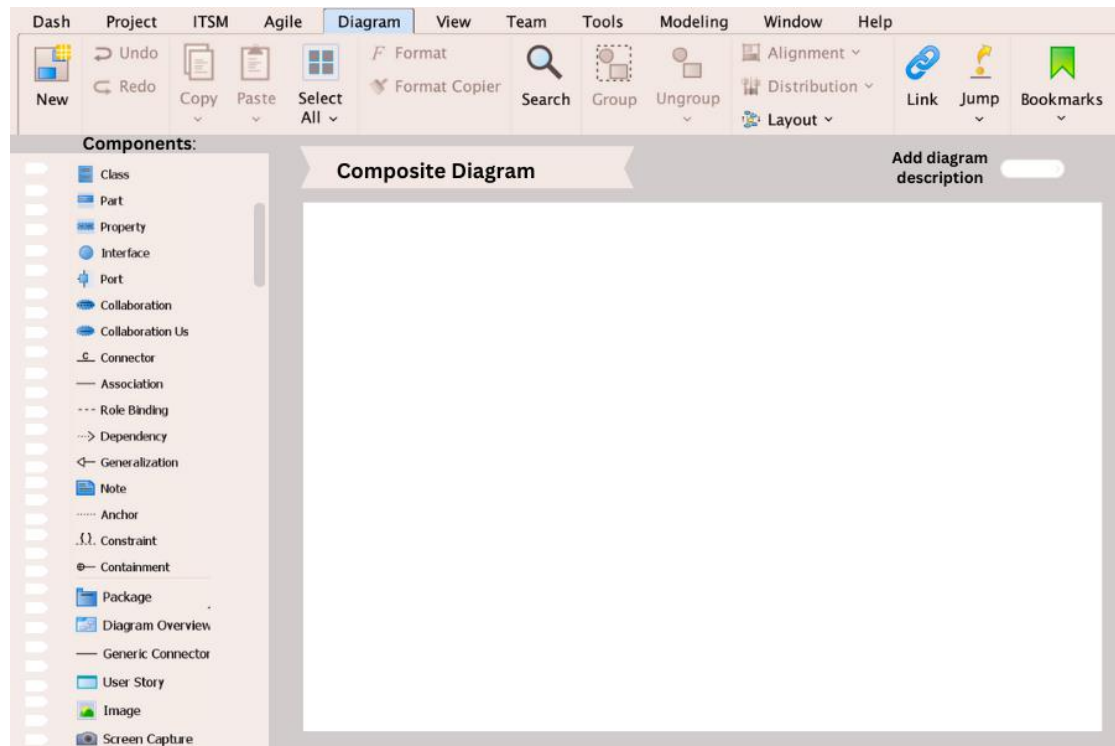
■ CLASS DIAGRAM



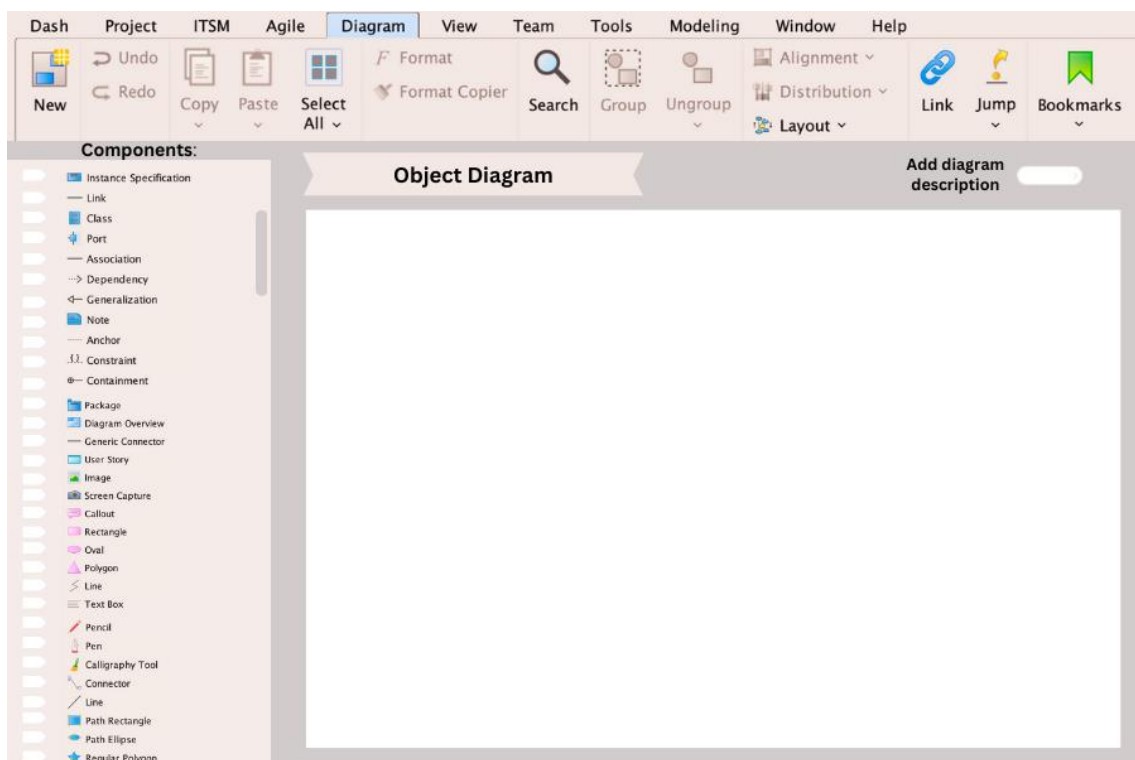
■ COMPONENT DIAGRAM



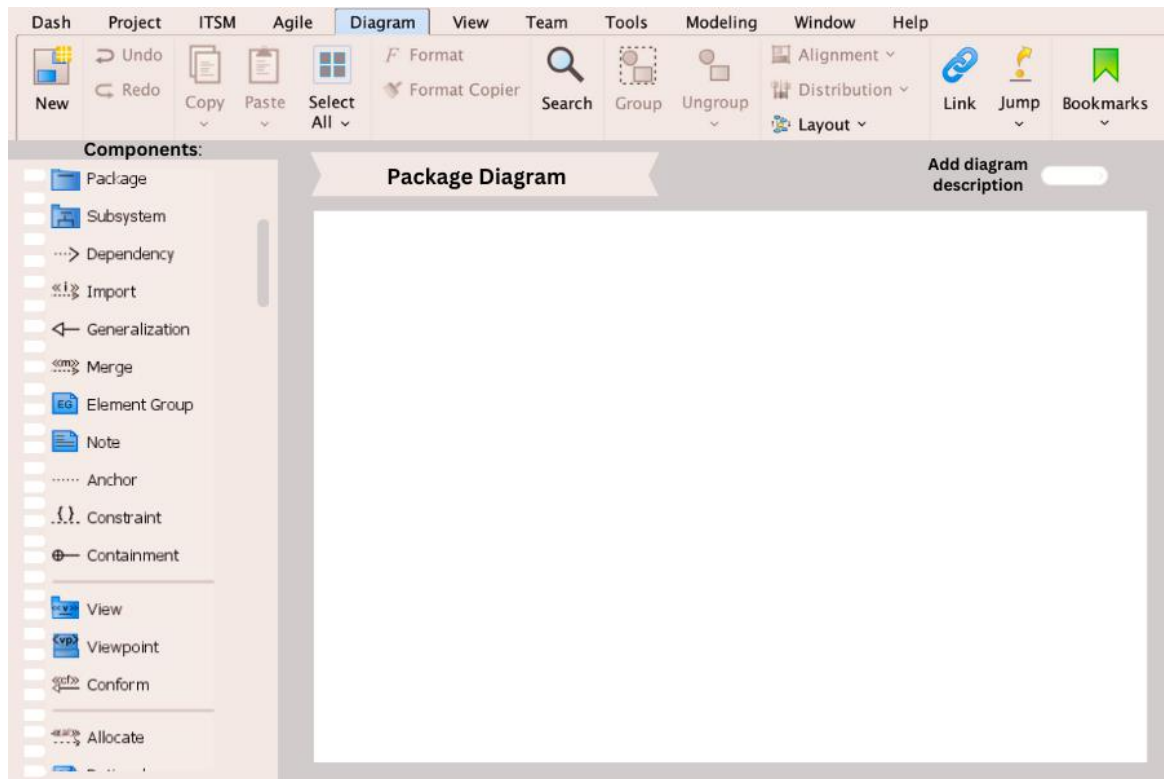
■ COMPOSITE DIAGRAM



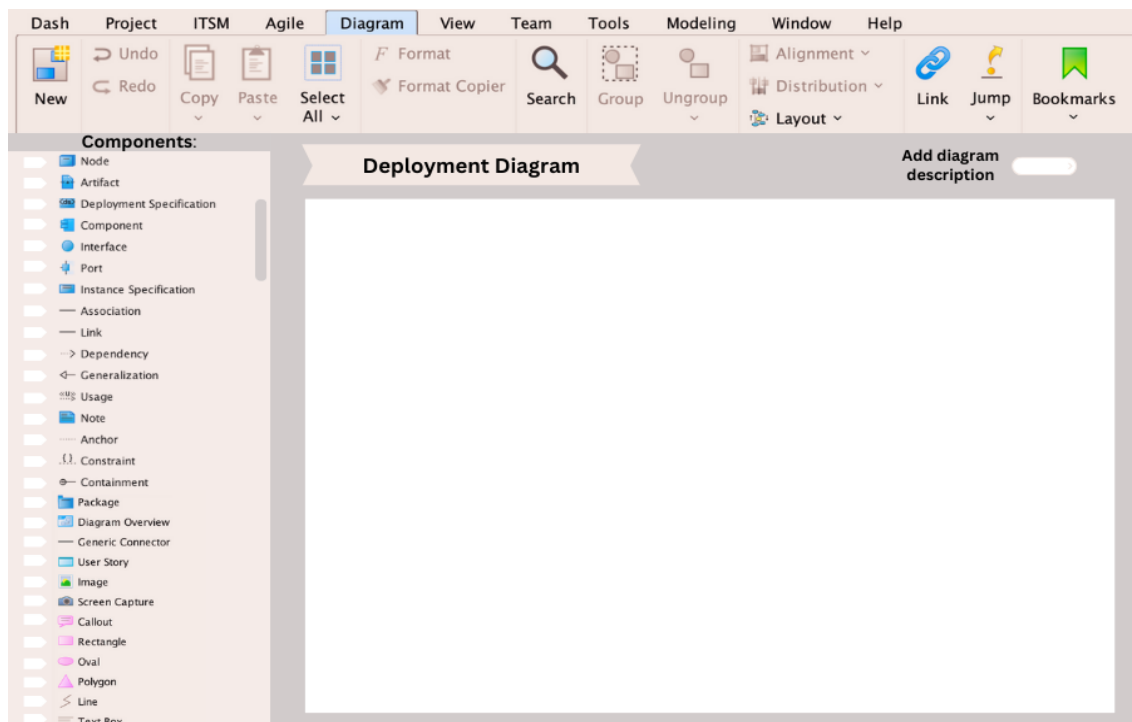
■ OBJECT DIAGRAM



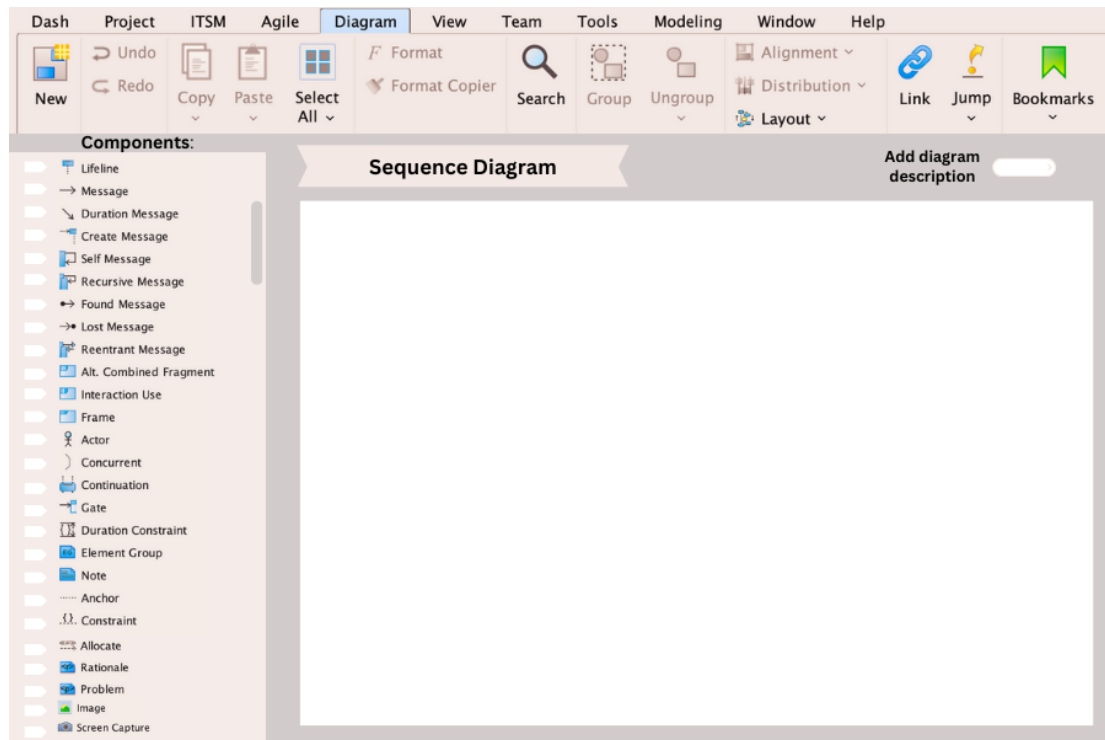
■ PACKAGE DIAGRAM



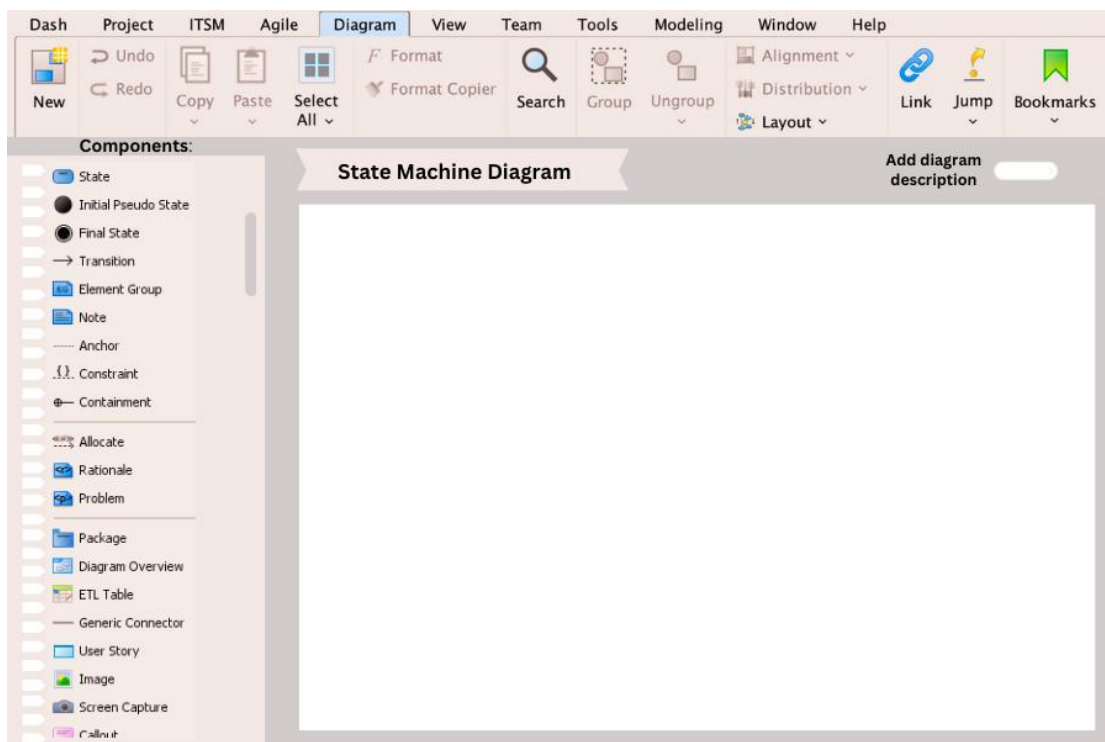
■ DEPLOYMENT DIAGRAM



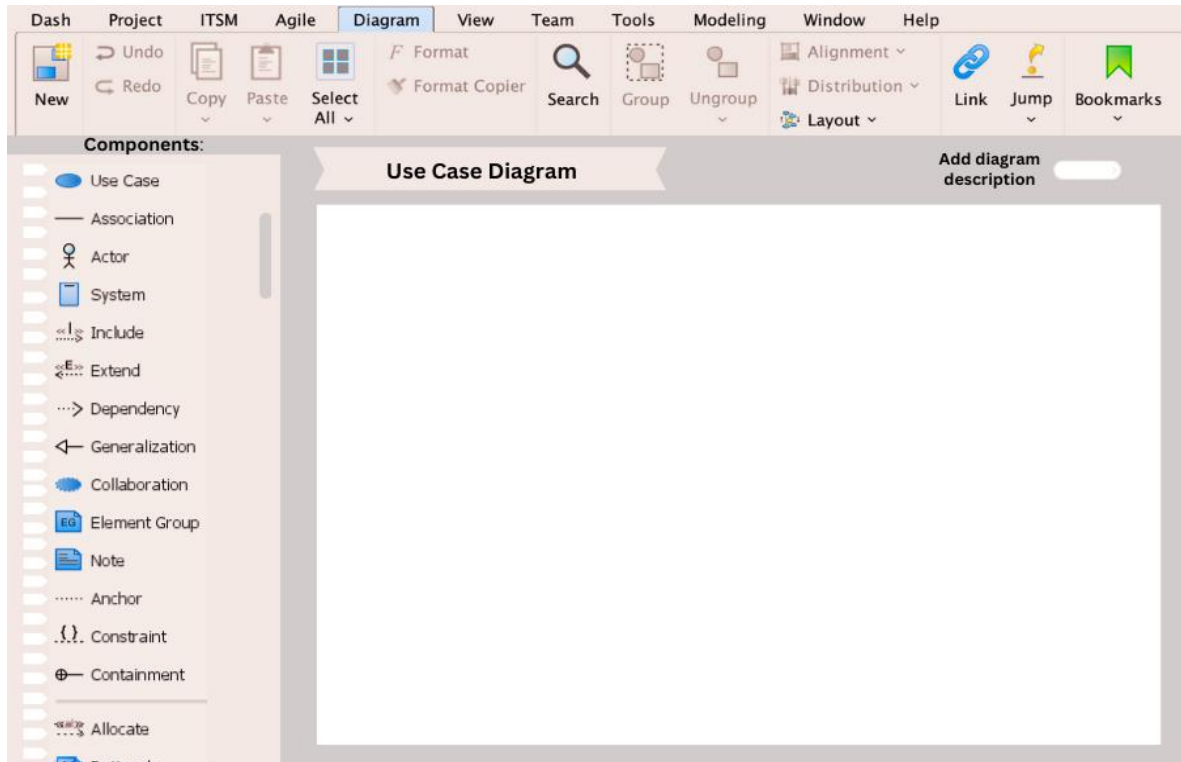
■ SEQUENCE DIAGRAM



■ STATE MACHINE DIAGRAM



■ USE CASE DIAGRAM



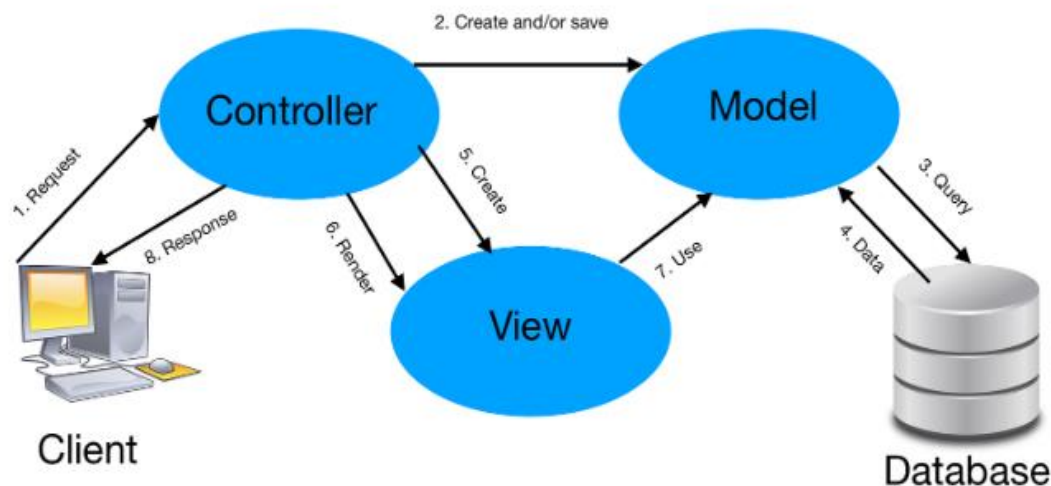
HIGH LEVEL DESIGN FOR ONLINE DESIGN TOOL (ODT)

Architectural Style for Online Design Tool (ODT)

- Selected Architectural Style: **Model-View-Controller (MVC)**

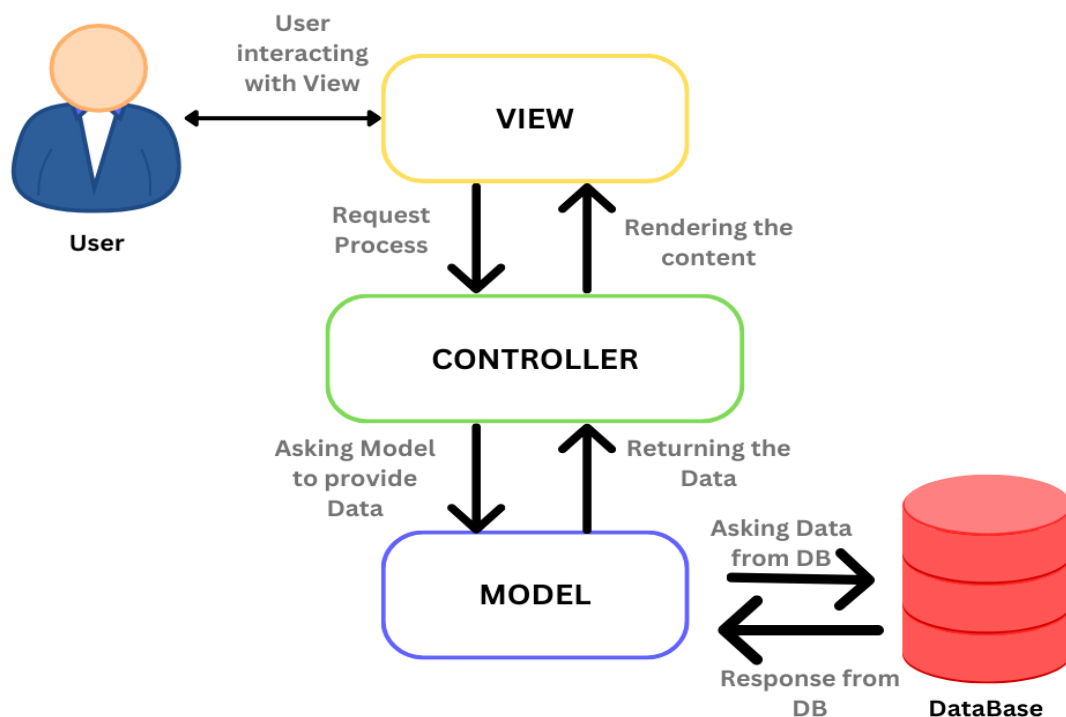
- **The Rationale for Selecting MVC Architectural Style**

The Model-View-Controller (MVC) architectural pattern is a widely used design pattern for interactive applications. It separates the application into three interconnected components: the Model, the View, and the Controller. This separation facilitates the development, testing, and maintenance of applications by dividing responsibilities and encouraging a clean separation of concerns. Here's why MVC is well-suited for the Online Design Tool (ODT):



1. Separation of Concerns:

- **Model:** Represents the application's data and business logic. For the ODT, this includes data structures for diagrams, shapes, templates, and user profiles.
- **View:** Manages the presentation layer. For the ODT, this involves rendering the user interface, including the diagram canvas, toolbars, and menus.
- **Controller:** Handles the input logic and updates the Model and View accordingly. For the ODT, this includes user interactions like creating shapes, editing diagrams, saving files, and so on.



2. Modularity and Maintainability:

- **Modular Design:** Each component (Model, View, Controller) can be developed, tested, and maintained independently, leading to better-organized code.
- **Ease of Updates:** Changes in the user interface (View) or business logic (Model) can be made with minimal impact on other components, making the system easier to maintain and extend.

3. Reusability:

- **Reusable Components:** The separation into distinct components encourage reusability. For example, the diagram rendering logic (Model) can be reused across different views (e.g., web, mobile).
- **Consistency:** Ensures a consistent user experience across different parts of the application.

4. Parallel Development:

- **Team Collaboration:** Different teams can work on different components simultaneously. Front-end developers can focus on Views, while back-end developers can work on Models and Controllers.
- **Efficient Workflow:** This parallelism speeds up the development process, making it possible to deliver features faster.

5. Testability:

- **Unit Testing:** Each component can be tested independently. Models can be tested for business logic, Views for user interface rendering, and Controllers for application flow and input handling.
- **Reduced Complexity:** The clear separation reduces the complexity of individual components, making them easier to test and debug.

▪ Applying MVC to ODT:

• Key Components for the ODT:

1. Model:

- **Diagram Data Structures:** Represent diagrams, including shapes, connectors, text, and metadata.
- **User Data:** Includes user profiles, preferences, and access rights.

- **Templates and Assets:** Predefined shapes and templates that users can incorporate into their diagrams.
- **Storage Management:** Handles saving, loading, and version control of diagrams.

2. View:

- **User Interface:** The graphical interface for creating, editing, and managing diagrams. This includes:
 1. **Canvas:** Where diagrams are drawn.
 2. **Toolbars and Menus:** Provide tools and options for diagram creation and editing.
 3. **Dialogs and Modals:** For file operations, settings, and user interactions.
- **Responsive Design:** Ensures the tool works well on different devices and screen sizes.

3. Controller:

- **Input Handling:** Manages user interactions such as mouse clicks, keyboard input, and touch events.
- **Command Execution:** Interprets user actions and updates the Model accordingly (e.g., adding a shape, saving a diagram).
- **Data Synchronization:** Ensures consistency between the Model and View, updating the View when the Model changes and vice versa.

▪ Interaction Between Model and View:

- **Updating the View:** When the Model changes (e.g., a new shape is added), it notifies the View to update the display.
- **User Actions:** When a user interacts with the View (e.g., clicks a button to add a shape), the View sends these actions to the Controller, which then updates the Model.

▪ **Example Workflow in MVC for ODT:**

1. User Interaction (Controller):

- The user clicks on the canvas to create a new shape.
- The Controller captures the click event and determines the appropriate action.

2. Business Logic (Model):

- The Controller updates the Model to include the new shape.
- The Model processes any business rules, such as ensuring shapes do not overlap or adhere to a specific diagram type's rule.

3. Update UI (View):

- The Controller instructs the View to update the canvas to reflect the new shape.
- The View renders the updated diagram to the user.

4. Feedback Loop:

- Any changes in the Model (e.g., the user saves the diagram) trigger updates in the View through the Controller, ensuring the user sees the most current data.

▪ **Model Components:**

1. User Model

- **Attributes:** user_id, username, email, password_hash, created_at, updated_at
- **Methods:**
 - createUser(username, email, password_hash)
 - getUserById(userId)
 - updateUser(userId, updatedData)
 - deleteUser(userId)
 - authenticateUser(email, password)
 - getAllUsers()

2. Project Model

- **Attributes:** id, name, description, owner_id, created_at, updated_at
- **Methods:**
 - create_project(name, description, owner_id)
 - getProjectById(projectId)
 - updateProject(projectId, updatedData)
 - deleteProject(projectId)
 - getProjectsByUserId(userId)
 - getAllProjects()

3. Diagram Model

- **Attributes:** id, name, project_id, created_at, updated_at
- **Methods:**
 - add_diagram(diagram)
 - remove_diagram(diagram_id)
 - getDiagramById(diagramId)
 - updateDiagram(diagramId, updatedData)
 - getDiagramsByProjectId(projectId)
 - getAllDiagrams()

4. DiagramElement Model

- **Attributes:** id, diagram_id, type, properties, position_x, position_y, created_at, updated_at
- **Methods:**
 - add_element(element)
 - remove_element(element_id)

- `getDiagramElementById(elementId)`
- `update_properties(new_properties)`
- `getDiagramElementsByDiagramId(diagramId)`
- `getAllDiagramElements()`

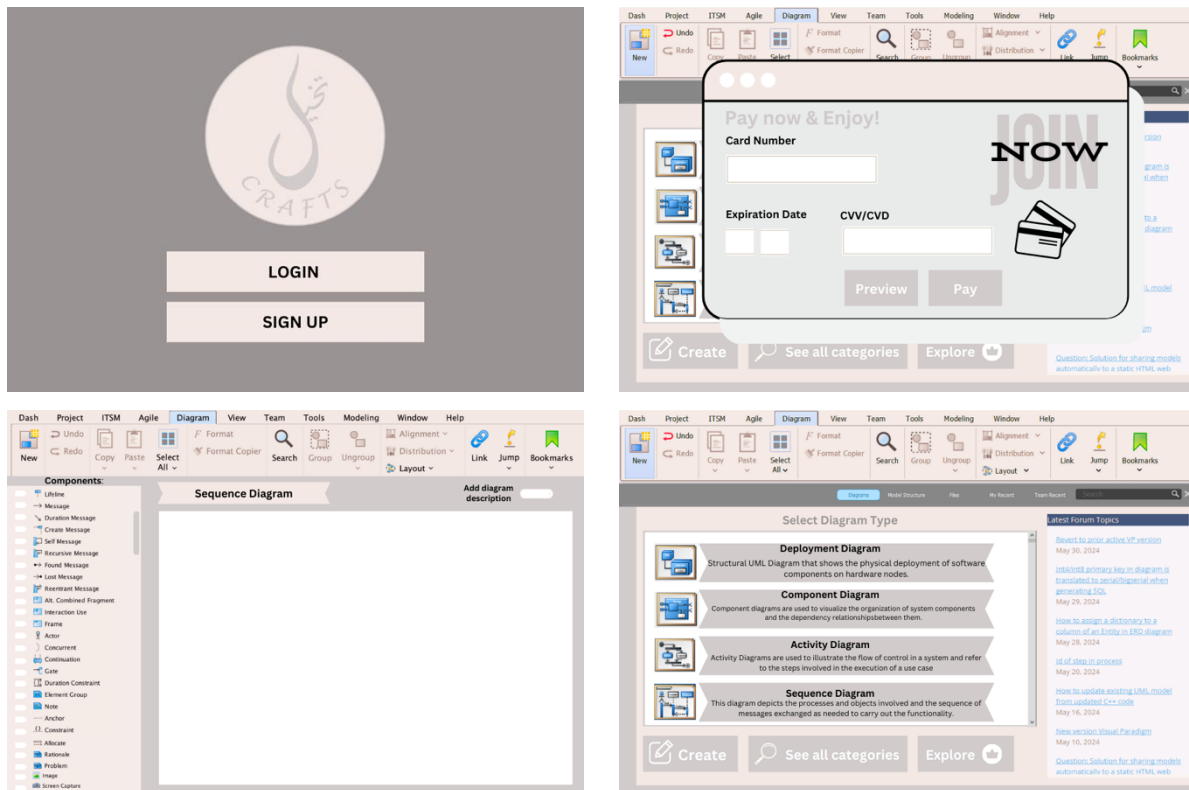
5. Comment Model

- **Attributes:** `id`, `element_id`, `user_id`, `content`, `created_at`, `updated_at`
- **Methods:**
 - `createComment(elementId, userId, content)`
 - `getCommentById(commentId)`
 - `updateComment(commentId, updatedData)`
 - `deleteComment(commentId)`
 - `getCommentsByElementId(elementId)`
 - `getAllComments()`

6. Permission Model

- **Attributes:** `id`, `project_id`, `user_id`, `role`, `granted_at`
- **Methods:**
 - `grantPermission(projectId, userId, role)`
 - `revokePermission(permissionId)`
 - `getPermissionsByProjectId(projectId)`
 - `getPermissionsByUserId(userId)`
 - `getAllPermissions()`

- Some of the Interfaces our ODT VIEW will have been:



Summary

The MVC architectural style is well-suited for the ODT because it provides a clear separation of concerns, facilitating modularity, maintainability, and parallel development. It ensures that the application is organized, making it easier to develop, test, and extend. By leveraging MVC, the ODT can provide a robust, user-friendly, and maintainable platform for creating and managing software analysis and design diagrams.

Design Patterns for Online Design Tool (ODT)

▪ Purpose and Overview of Design Patterns in ODT

• Purpose:

Design patterns are essential in the Online Design Tool (ODT) for ensuring a robust, maintainable, and scalable software architecture. They provide proven solutions to common software design problems, facilitating better communication among developers and enhancing code reusability and flexibility.

• Overview:

This section covers the essential design patterns implemented in ODT, detailing their roles, purposes, and how they contribute to the overall architecture. The patterns are categorized into structural, behavioural, and creational types, each serving a distinct purpose in the system.

▪ Structural Design Pattern: Facade

• Purpose:

The Facade pattern provides a simplified interface to a complex subsystem, making the subsystem easier to use and reducing dependencies between the client and the subsystem.

• Implementation in ODT:

Context:

In ODT, various components like diagram management, project management, and user authentication have complex interactions. The Facade pattern simplifies these interactions for the user interface and client components.

Implementation Details:

I. Facade Class:

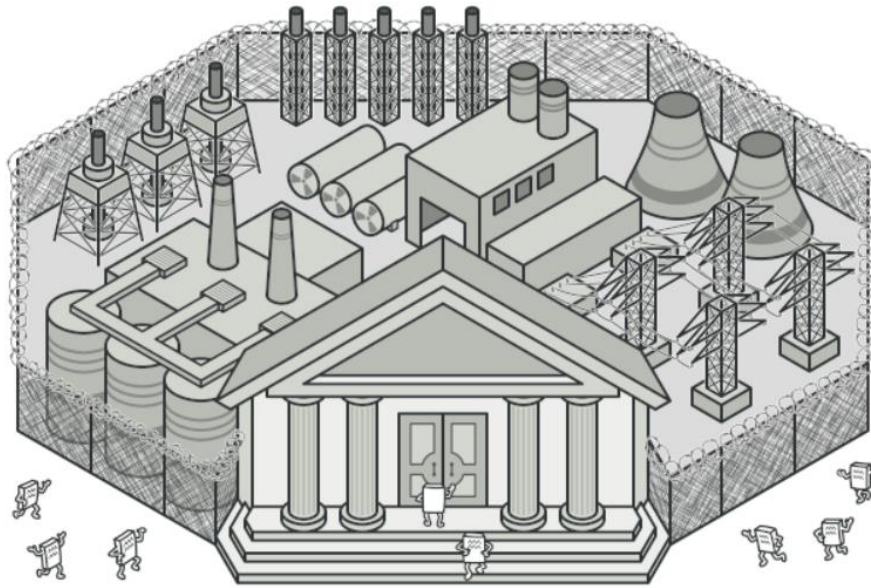
- Acts as a single point of entry for the ODT functionalities.
- Simplifies the interface for managing projects, diagrams, and user sessions.

II. Subsystem Classes:

- Classes like **ProjectManager**, **DiagramManager**, and **UserManager** handle the actual logic.
- The Facade class delegates calls to these subsystem classes.

- **Code:**

```
public class ODTFacade {  
    private ProjectManager projectManager;  
    private DiagramManager diagramManager;  
    private UserManager userManager;  
    public ODTFacade() {  
        this.projectManager = new ProjectManager();  
        this.diagramManager = new DiagramManager();  
        this.userManager = new UserManager();    }  
    public int createProject(String name, String description, int userId) {  
        return projectManager.createProject(name, description, userId);  
    }  
    public int createDiagram(String name, int projectId) {  
        return diagramManager.createDiagram(name, projectId);  
    }  
    public boolean authenticateUser(String email, String password) {  
        return userManager.authenticate(email, password);  
    }  
}
```



- **Benefits:**

- Simplifies client interaction with the system.
- Reduces the dependency between clients and complex subsystems.
- Enhances maintainability and flexibility by decoupling the subsystem from the client.

- **Behavioural Design Pattern: Observer**

- **Purpose:**

The Observer pattern allows objects to be notified of changes in another object. It is useful for implementing distributed event handling systems.

- **Implementation in ODT:**

Context:

ODT needs to update different parts of the system when certain events occur, such as diagram modifications or project updates. The Observer pattern ensures that all relevant components stay synchronized.

Implementation Details:

I. Subject Interface:

Maintains a list of observers and notifies them of any changes.

II. Observer Interface:

Defines an update method to be implemented by concrete observers.

III. Concrete Subject:

Implements the Subject interface and notifies observers of state changes.

IV. Concrete Observers:

Implement the Observer interface and update themselves based on notifications from the subject.

- **Code:**

```
interface Observer {  
    void update();  
}  
  
class Diagram implements Observer {  
    private String diagramState;  
    @Override  
    public void update() {  
        // Update diagram state  
    }  
}  
  
class Project implements Subject {  
    private List<Observer> observers;  
    private String projectState;  
    public Project() {  
        this.observers = new ArrayList<>();  
    }  
}
```

@Override

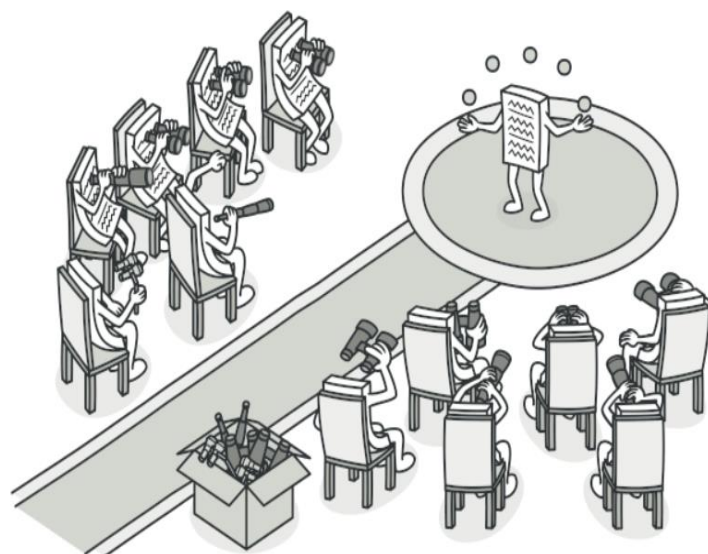
```
public void attach(Observer observer) {  
    observers.add(observer);  
}
```

@Override

```
public void detach(Observer observer) {  
    observers.remove(observer);  
}
```

@Override

```
public void notifyObservers() {  
    for (Observer observer : observers) {  
        observer.update();  
    }  
}  
  
public void changeState(String newState) {  
    this.projectState = newState;  
    notifyObservers();  
}  
}
```



- **Benefits:**

- Decouples the subject and its observers, promoting loose coupling.
- Enhances the flexibility and reusability of the components.
- Facilitates an event-driven architecture.

- **Creational Design Pattern: Builder**

- **Purpose:**

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

- **Implementation in ODT:**

Context:

ODT needs to create complex objects like projects and diagrams with various configurations. The Builder pattern provides a flexible solution for constructing these objects.

Implementation Details:

I. Builder Interface:

Defines the steps required to build the complex object.

II. Concrete Builder:

Implements the Builder interface and provides specific implementations for the construction steps.

III. Director:

Manages the construction process using the Builder interface.

IV. Product:

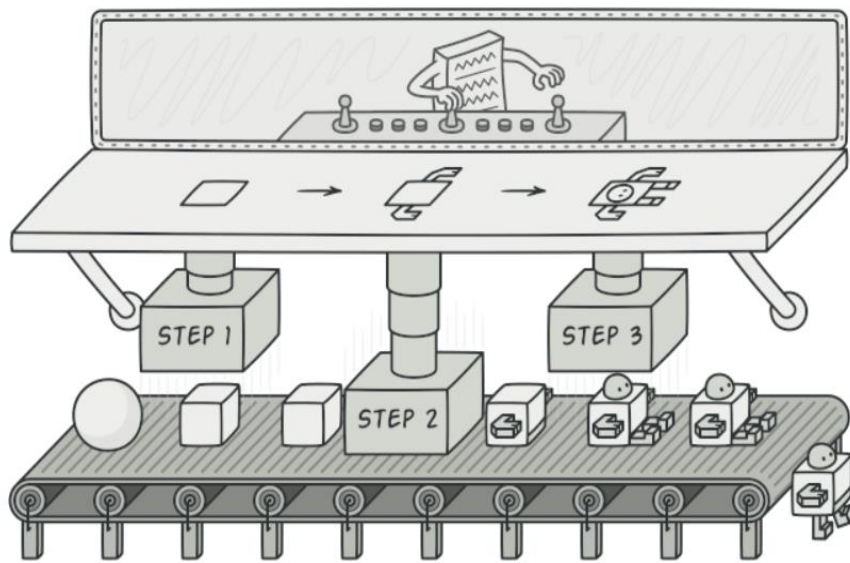
The complex object being built.

- **Code:**

```
class Project {  
    private String name;  
    private String description;  
    private int userId;  
    // Private constructor to force the use of the Builder  
    private Project(ProjectBuilder builder) {  
        this.name = builder.name;  
        this.description = builder.description;  
        this.userId = builder.userId;}  
    public static class ProjectBuilder {  
        private String name;  
        private String description;  
        private int userId;  
        public ProjectBuilder setName(String name) {  
            this.name = name;  
            return this;}  
        public ProjectBuilder setDescription(String description) {  
            this.description = description;  
            return this;}  
        public ProjectBuilder setUserId(int userId) {  
            this.userId = userId;  
            return this; }  
        public Project build() {  
            return new Project(this); }}}
```

// Usage

```
Project project = new Project.ProjectBuilder()  
    .setName("New Project")  
    .setDescription("Project Description")  
    .setUserId(123)  
    .build();
```



- **Benefits:**

- Provides clear separation between the construction and representation of an object.
- Allows for step-by-step construction of complex objects.
- Enhances code readability and maintainability by using method chaining.

- **Summary**

In summary, the design patterns used in ODT provide a structured approach to solving common design problems, improving the system's overall architecture. The Facade pattern simplifies complex subsystems, the Observer pattern supports an event-driven architecture, and the Builder pattern facilitates the construction of complex objects. Each pattern is carefully chosen to address specific needs within the ODT, ensuring a robust, maintainable, and scalable design.

▪ Future Enhancements

- **Improved User Interface**

A user-friendly and intuitive interface is crucial for the ODT's success. Enhancements to the user interface can improve usability and user satisfaction.

- **Customization Options**

Providing users with options to customize their workspace and design environment can make the tool more adaptable to individual needs.

- **Advanced Security Features**

To further protect user data and maintain trust, the ODT can implement more advanced security features.

- **Biometric Authentication**

Adding biometric authentication, such as fingerprint or facial recognition, can provide an extra layer of security.

- **Enhanced Encryption Techniques**

Adopting state-of-the-art encryption methods can ensure that user data remains secure against evolving threats.

- **Enhanced Integration Capabilities**

Integrating the ODT with other popular design and project management tools can streamline workflows and improve productivity.

- **API Integration**

Providing robust APIs can allow seamless integration with third-party tools and platforms, expanding the ODT's functionality.

- **Cross-Platform Compatibility**

Ensuring the ODT works well with other software solutions can enhance its versatility and user adoption.

- **Potential Questions**

How does the ODT ensure data security and privacy?

What mechanisms are in place for data recovery in the event of a failure?

How does the ODT handle user collaboration on projects?

▪ Conclusion

In a nut shell, the Online Design Tool (ODT) is a comprehensive platform designed to streamline the design process through efficient project and diagram management, robust security measures, and seamless collaboration functionalities. By focusing on user-friendly interfaces, advanced security, and enhanced integration capabilities, the ODT meets the evolving needs of designers and engineers. Future enhancements, such as machine learning integration, real-time collaboration, and advanced search capabilities, will further elevate the tool's utility and efficiency. This documentation provides a thorough overview of the algorithms and functionalities that make the ODT a powerful and reliable tool for the modern design landscape.

▪ Reference Citations

- https://www.tutorialspoint.com/software_architecture_design/introduction.htm
- https://www.youtube.com/watch?v=NU_1StN5Tkk
- <https://www.geeksforgeeks.org/software-design-patterns/>
- <https://www.visual-paradigm.com/>
- <https://www.javatpoint.com/data-structure-tutorial>
- https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm

THE END