

**RIPHAH INTERNATIONAL UNIVERSITY, GULBERG
GREEN CAMPUS**



**Digital Workspace and Collaboration
Platform (DWCP)**

Submitted to: Ms. Kausar Nasreen

Submitted by: Group 8

Ayesha Mukhtar Asad 4587

Hajra Rizwan-45504

Mahnoor Asif-44957

Aleeha Akhlaq-46174

Azka Humayon-47631

Contents

Artifact 1	5
Use Case Diagram	5
1.1 Use Case Diagram:	6
1.2 Key Components:	6
1.3 Main Concept:	6
1.3.1 Include:	6
1.3.2 Extend:	6
1.4 Association:	6
1.5 Actor:	6
1.6 Digital Workspace and Collaboration Platform (DWCP):	7
Artifact 2	8
Fully Dressed Format	8
2 Fully Addressed Format of Use Cases:	9
2.1 UC-001: User Authentication and Role-Based Access Control	10
2.2 UC-002: Project Creation and Task Assignment	11
2.3 UC-003: Task Progress Tracking	12
2.4 UC-004: Discussion Board for Collaboration	13
2.5 UC-005: Reports and Analytics	14
Artifact 3	15
Domain Modeling	15
3.1 Domain Modeling	16
3.1.1 Key Components of Domain Modeling:	16
3.2 Domain Modeling Diagram of DWCP	17
3.3 Diagram Explanation	17
3.3.1. User Class:	18
3.3.2. Role Class:	18
3.3.3. Admin Class:	18
3.3.4. ProjectManager Class:	18
3.3.5. Team Member Class:	19
3.3.6. Project Class:	19
3.3.7. Task Class:	19
3.3.8. Workspace Class:	19
3.3.9. Discussion Class:	20

3.3.10. Notification Class:.....	20
3.3.11. Report Class:	20
3.3.12. ProductivityReport Class:.....	20
3.3.13. ResourceUtilizationReport Class:	21
3.3.14. Relationships Overview:.....	21
Artifact 4.....	22
Class Diagram	22
4.1 Class Diagram	23
4.2 Class Diagram of DWCP.....	25
4.4 Diagram Explanation Core Classes and Interfaces	26
Specialized User Roles	26
Entities	27
Relationships	28
Artifact 5.....	30
Activity Diagram.....	30
5.1 Activity Diagram.....	31
5.2 Fundamental Properties of Activity Diagrams	31
5.3 Basic Elements of an Activity Diagram	31
5.4 Diagram	33
5.5 Diagram Explanation.....	34
1. User Authentication:	34
2. Admin Role:.....	34
3. Project Manager Role:.....	34
4. Team Member Role:	35
5. Notifications:.....	35
6. Reports:.....	35
Artifact 6.....	36
Sequence Diagram	36
6.1 Sequence Diagram	37
6.2 Key Elements of a Sequence Diagram:	37
6.2 Diagram explanation	40
1. User Login:	40
2. Project Creation and Task Management:	40
3. Project Progress Tracking:	40
4. Discussion Updates:	41
5. Workspace Management:	41

6. Notifications:	41
7. Report Generation:	41
Artifact 6	42
State Transition Diagram.....	42
6.1 State Transition Diagram.....	43
6.2 Explanation of the Diagram.....	44
Artifact7	46
MVC Framework.....	46
7.1 MVC Framework:	47
7.2 Application in DWCP:.....	47
7.2.1 View (User Interface Layer):	47
7.2.2 Controller:	48
7.2.3 Model:	49
7.2.4 Main Application:	50
Artifact 8	52
GRASP Pattern	52
8.1 GRASP Patterns	53
Creator	53
8.2 Application in DWCP:.....	53
8.3 Application in DWCP:.....	56
8.4 Low Coupling	59
8.5 High Cohesion	60
References:	79

Artifact 1

Use Case Diagram

1.1 Use Case Diagram:

In a use case diagram, the focus is on how users interact with the system. It visually represents how different types of users engage with various features of the system and how they use them. The system includes multiple features, and the use case diagram shows which features each type of user can access and how they interact with them.

1.2 Key Components:

- Actor
- Use Case
- Association

1.3 Main Concept:

- Include
- Extend

1.3.1 Include:

Definition: Include means that one use case is always included in the behavior of another use case. It represents shared functionality that is needed every time the main use case runs.

1.3.2 Extend:

Definition: Extend indicates that one use case can optionally extend or add to the behavior of another use case, typically triggered under specific conditions.

1.4 Association:

This is a simple line between actor and use case it represents simply which actor is interacting with which use case.

1.5 Actor:

Any person any organization or anything which is interacting with your system or part of the system is called actor.

1.6 Digital Workspace and Collaboration Platform (DWCP):



Artifact 2

Fully Dressed Format

2 Fully Addressed Format of Use Cases:

A **fully addressed use case** is a detailed and structured document that outlines all aspects of a system or application's behavior, covering the main functionality as well as any alternatives, exceptions, and preconditions. It serves as a blueprint for developers, testers, and other stakeholders to understand how the system should work from the user's perspective.

This format ensures that every relevant aspect of the system's behavior is considered and documented, helping to avoid misunderstandings and miscommunications during development. It typically includes the following key components:

1. **Use Case ID:** A unique identifier for the use case.
2. **Name:** A brief, descriptive name for the use case.
3. **Authors:** The team or individual responsible for creating the use case.
4. **Priority:** The importance of the use case (e.g., high, medium, low).
5. **Criticality:** How essential the use case is to the system's functioning.
6. **Source:** The document or person that provided the requirements for the use case.
7. **Responsible:** The team or individual responsible for the implementation of the use case.
8. **Description:** A detailed explanation of the use case, summarizing what it accomplishes.
9. **Trigger Event:** The action or event that causes the use case to be initiated.
10. **Actors:** The users or system components involved in the use case (e.g., Admin, Project Manager, etc.).
11. **Pre-condition:** The conditions that must be true before the use case can start.
12. **Post condition:** The conditions that will be true once the use case has been completed.
13. **Result:** The expected outcome of the use case, which can be measurable or observable.
14. **Main Scenario:** The typical sequence of steps or actions that occur to complete the use case.
15. **Alternative Scenario:** The possible alternative actions or paths the use case can take (e.g., error handling or edge cases).

2.1 UC-001: User Authentication and Role-Based Access Control

Field	Details
UseCaseID	UC-001
Name	User Authentication and Role-Based Access Control
Authors	Mahnoor Asif
Priority	High
Criticality	Essential
Source	Requirements Document
Responsible	Development Team
Description	Enables secure login and role-specific access for Admin, Project Managers, and Team Members.
Trigger Event	User accesses the login page and attempts to authenticate.
Actors	Admin, Project Manager, Team Member
Precondition	The user must have an active account in the system.
Post condition	The user gains access to role-specific functionalities.
Result	Secure access to the system with appropriate permissions.
Main Scenario	1. User navigates to the login page.
	2. User enters valid credentials.
	3. System authenticates the user and identifies their role.
	4. Role-specific dashboards are displayed.
Alternative Scenario	1. User enters invalid credentials.
	2. System denies access and prompts for retry or password reset.

2.2 UC-002: Project Creation and Task Assignment

Field	Details
UseCaseID	UC-002
Name	Project Creation and Task Assignment
Authors	Ayesha Asad
Priority	High
Criticality	Essential
Source	Requirements Document
Responsible	Development Team
Description	Facilitate the creation of projects, division into tasks, and assignment to team members.
Trigger Event	Project Manager initiates a new project.
Actors	Project Manager
Precondition	The Project Manager must be logged in with appropriate permissions.
Post condition	A new project and its associated tasks are created and assigned.
Result	Efficient task distribution with clear ownership.
Main Scenario	1. Project Manager selects 'Create Project'.
	2. System prompts for project details.
	3. Tasks are added and assigned.
	4. Notifications are sent.
Alternative Scenario	1. Project Manager enters incomplete details.
	2. System prompts for required fields.

2.3 UC-003: Task Progress Tracking

Field	Details
UseCaseID	UC-003
Name	Task Progress Tracking
Authors	Hajra Rizwan
Priority	High
Criticality	Essential
Source	Requirements Document
Responsible	Development Team
Description	Allow Team Members to update task statuses and Project Managers to track overall progress.
Trigger Event	Team Member updates their task or Project Manager reviews the progress.
Actors	Project Manager, Team Member
Precondition	Tasks must be assigned and visible to Team Members.
Post condition	Updated task status and a clear view of progress for Project Managers.
Result	Transparent task management and progress visibility.
Main Scenario	1. Team Member logs in.
	2. Task status is updated.
	3. System reflects changes in the dashboard.
	4. Project Manager monitors updates.
Alternative Scenario	1. Connectivity issues.
	2. System queues the update for later.

2.4 UC-004: Discussion Board for Collaboration

Field	Details
UseCaseID	UC-004
Name	Discussion Board for Collaboration
Authors	Aleeha Akhlaq
Priority	Medium
Criticality	Important
Source	Requirements Document
Responsible	Development Team
Description	Facilitate communication between Team Members and Project Managers via project-specific boards.
TriggerEvent	A user posts a message in the discussion board.
Actors	Project Manager, Team Member
Precondition	The user must be part of the project.
Postcondition	Message is visible to all project participants.
Result	Improved team collaboration and reduced miscommunication.
MainScenario	1. User accesses discussion board.
	2. Posts a message.
	3. Other members receive notifications.
	4. Clarifications are provided.
AlternativeScenario	1. Irrelevant message posted.
	2. Message flagged or deleted by the Project Manager.

2.5 UC-005: Reports and Analytics

Field	Details
UseCaseID	UC-005
Name	Reports and Analytics
Authors	Azka Humayoun
Priority	Medium
Criticality	Important
Source	Requirements Document
Responsible	Development Team
Description	Generate productivity and resource utilization reports for Admins and Project Managers.
TriggerEvent	Admin or Project Manager selects the "Generate Report" option.
Actors	Admin, Project Manager
Precondition	Data on team productivity and task progress must exist in the system.
Postcondition	A detailed report is generated and displayed or exported.
Result	Insights into team performance and resource allocation.
MainScenario	1. User accesses reports section.
	2. Selects report type.
	3. System generates report.
	4. User views or exports it.
AlternativeScenario	1. Insufficient data.
	2. System suggests generating another report.

Artifact 3

Domain Modeling

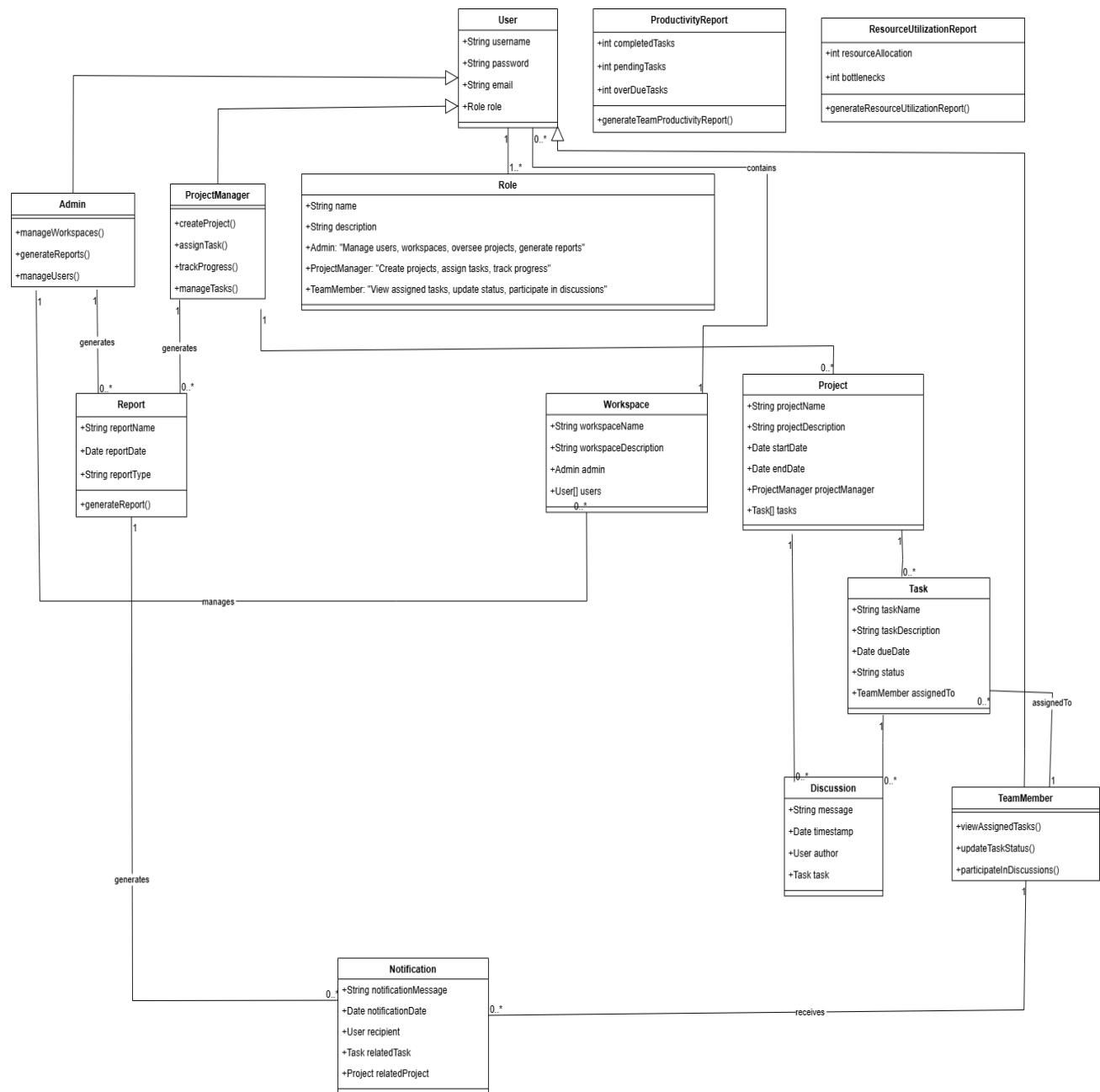
3.1 Domain Modeling

Domain Modeling refers to the process of creating a conceptual model of the structure and behavior of a specific domain or problem space. It involves identifying and defining the key elements, concepts, relationships, and behaviors that exist within the system you're working with, often represented in a diagrammatic form like a **class diagram** or **entity-relationship diagram (ERD)**.

3.1.1 Key Components of Domain Modeling:

1. **Entities:** These are the main objects or concepts in the domain. They represent real-world or abstract things that have significance in the system. For example, in an e-commerce system, entities might be Customer, Order, and Product.
2. **Attributes:** These are the properties or characteristics of an entity. For example, the Customer entity might have attributes like name, email, and address.
3. **Relationships:** These represent how entities are related to one another. Relationships could be one-to-one, one-to-many, or many-to-many. For example, a Customer can place multiple Orders (one-to-many relationship).
4. **Associations:** This is how entities interact with each other. For example, an Order might reference a Product, indicating which products were included in the order.
5. **Behaviors/Methods:** These are the operations that an entity can perform, or actions that can be taken on entities. For example, an Order entity might have methods like addItem() or calculateTotal().

3.2 Domain Modeling Diagram of DWCP



3.3 Diagram Explanation

Domain Model Class Diagram. It represents the conceptual model of the system by showing the key entities (classes), their attributes, methods, and relationships. This type of diagram is used in software design to illustrate the structure of the system and the business logic it aims to represent.

In domain modeling:

3.3.1. User Class:

- **Attributes:**
 - username: Stores the username of the user.
 - password: Stores the user password.
 - email: Stores the user's email address.
- **Methods:**
 - Constructor (+init): Initializes user details.
 - +generateReport(): Allows the user to generate reports.
- **Relationship:** Inherits from the Role class and can have multiple roles (Admin, ProjectManager, TeamMember).

3.3.2. Role Class:

- **Attributes:**
 - name: Name of the role (Admin, ProjectManager, TeamMember).
 - description: A brief description of the role.
- **Role Types:**
 - **Admin:** Manages users, workspaces, and projects, and can generate reports.
 - **ProjectManager:** Creates projects, assigns tasks, and tracks progress.
 - **TeamMember:** Views assigned tasks, updates status, and participates in discussions.

3.3.3. Admin Class:

- **Methods:**
 - +manageWorkspaces(): Manages creation and updates of workspaces.
 - +generateReports(): Generates different types of reports.
 - +manageUsers(): Adds or removes users from the system.

3.3.4. ProjectManager Class:

- **Methods:**
 - +createProject(): Allows the creation of a new project.
 - +assignTask(): Assigns tasks to team members.
 - +trackProgress(): Monitors the progress of the project.

- +manageTasks(): Manages tasks within the project.

3.3.5. Team Member Class:

- **Methods:**

- +viewAssignedTasks(): Views tasks assigned to the team member.
- +updateTaskStatus(): Updates the status of the assigned tasks.
- +participateInDiscussions(): Engages in project-related discussions.

3.3.6. Project Class:

- **Attributes:**

- projectName: Name of the project.
- projectDescription: Description of the project.
- startDate: Project start date.
- endDate: Project end date.

- **Relationships:**

- Managed by a ProjectManager.
- Contains multiple Task objects.

3.3.7. Task Class:

- **Attributes:**

- taskName: Name of the task.
- taskDescription: Description of the task.
- dueDate: Due date for the task.
- status: Status of the task (e.g., pending, completed).

- **Relationships:**

- Assigned to a TeamMember.
- Associated with a Project.

3.3.8. Workspace Class:

- **Attributes:**

- workspaceName: Name of the workspace.
- workspaceDescription: Description of the workspace.

- **Relationships:**
 - Managed by an Admin.
 - Contains multiple users (User).

3.3.9. Discussion Class:

- **Attributes:**
 - message: Content of the discussion message.
 - timestamp: Date and time of the discussion message.
- **Relationships:**
 - Linked to a specific Task.
 - Authored by a User.

3.3.10. Notification Class:

- **Attributes:**
 - notificationMessage: Message content of the notification.
 - notificationDate: Date of the notification.
- **Relationships:**
 - Received by a User.
 - Can be related to a specific Task or Project.

3.3.11. Report Class:

- **Attributes:**
 - reportName: Name of the report.
 - reportDate: Date when the report was generated.
 - reportType: Type of the report (e.g., productivity, progress).
- **Methods:**
 - +generateReport(): Method to generate the report.

3.3.12. ProductivityReport Class:

- **Attributes:**
 - completedTasks: Number of completed tasks.
 - pendingTasks: Number of pending tasks.
 - overDueTasks: Number of overdue tasks.

- **Methods:**

- +generateTeamProductivityReport(): Generates a report on team productivity.

3.3.13. ResourceUtilizationReport Class:

- **Attributes:**

- resourceAllocation: Information about resource usage.
- bottlenecks: Highlights any bottlenecks in resource allocation.

- **Methods:**

- +generateResourceUtilizationReport(): Generates a report on resource utilization.

3.3.14. Relationships Overview:

- User inherits from Role.
- Project contains multiple Task objects.
- Task is assigned to a TeamMember.
- Workspace is managed by Admin and contains multiple users.
- Report can be generated by Admin or ProjectManager.
- Notification is related to a Task or Project and is received by a User.
- Discussion is linked to a Task and created by a User.:

This diagram provides a comprehensive view of the different classes involved in the project management system, their attributes, methods, and interactions. It effectively models user roles, project and task management, workspace administration, notifications, discussions, and reporting functionalities.

Artifact 4

Class Diagram

4.1 Class Diagram

A **class diagram** is a type of **structural diagram** in UML (Unified Modeling Language) that shows the system's **classes**, their **attributes**, **methods**, and the relationships between the classes. It is used to represent the static structure of a system.

Here are the key components in a class diagram:

1. **Class:**

- Represents a blueprint for objects. It defines the properties and behaviors (attributes and methods) common to all objects of that class.
- **Notation:** A rectangle divided into three parts:
 - **Top part:** Class name
 - **Middle part:** Attributes (variables)
 - **Bottom part:** Methods (functions)

2. **Attributes:**

- Characteristics or properties of a class. They describe the state or data held by the class.
- **Notation:** Typically shown with the format visibility name: type (e.g., + age: int).
- **Visibility:**
 - + (Public): Accessible from outside the class.
 - - (Private): Accessible only within the class.
 - # (Protected): Accessible within the class and its subclasses.

3. **Methods (Operations):**

- Functions or behaviors that a class can perform.
- **Notation:** Shown as visibility name(parameters): returnType (e.g., + getAge(): int).

4. **Relationships:**

- Defines how classes are related to each other.
- Types of relationships:
 - **Association:** A basic relationship between two classes (e.g., a student **enrolls** in a course). Represented by a line connecting classes.
 - **Multiplicity:** Specifies how many instances of a class can be associated with another class (e.g., one-to-many, many-to-many).
 - **Aggregation:** A special form of association, representing a "whole-part" relationship where the part can exist independently of the whole (e.g., a **department** contains **employees**).

- **Composition:** A stronger form of aggregation, where the part cannot exist without the whole (e.g., a **house** contains **rooms**).
- **Inheritance (Generalization):** Represents an "is-a" relationship between a superclass and a subclass (e.g., a **dog** is a type of **animal**). Shown with a solid line and a hollow triangle pointing to the superclass.
- **Realization:** A relationship between an interface and the class that implements it. Shown with a dashed line and a hollow triangle.
- **Dependency:** Represents a weaker relationship where one class depends on another (e.g., one class uses another). Shown with a dashed line and an arrow.

5. **Interface:**

- A collection of abstract methods (no implementation) that a class must implement. It defines a contract.
- **Notation:** Shown as a rectangle with the word <<interface>> on top.

6. **Visibility:**

- Indicates the access level of the class members (attributes and methods).
 - + (Public): Visible to everyone.
 - - (Private): Visible only inside the class.
 - # (Protected): Visible to the class and its subclasses.
 - ~ (Package): Visible within the same package.

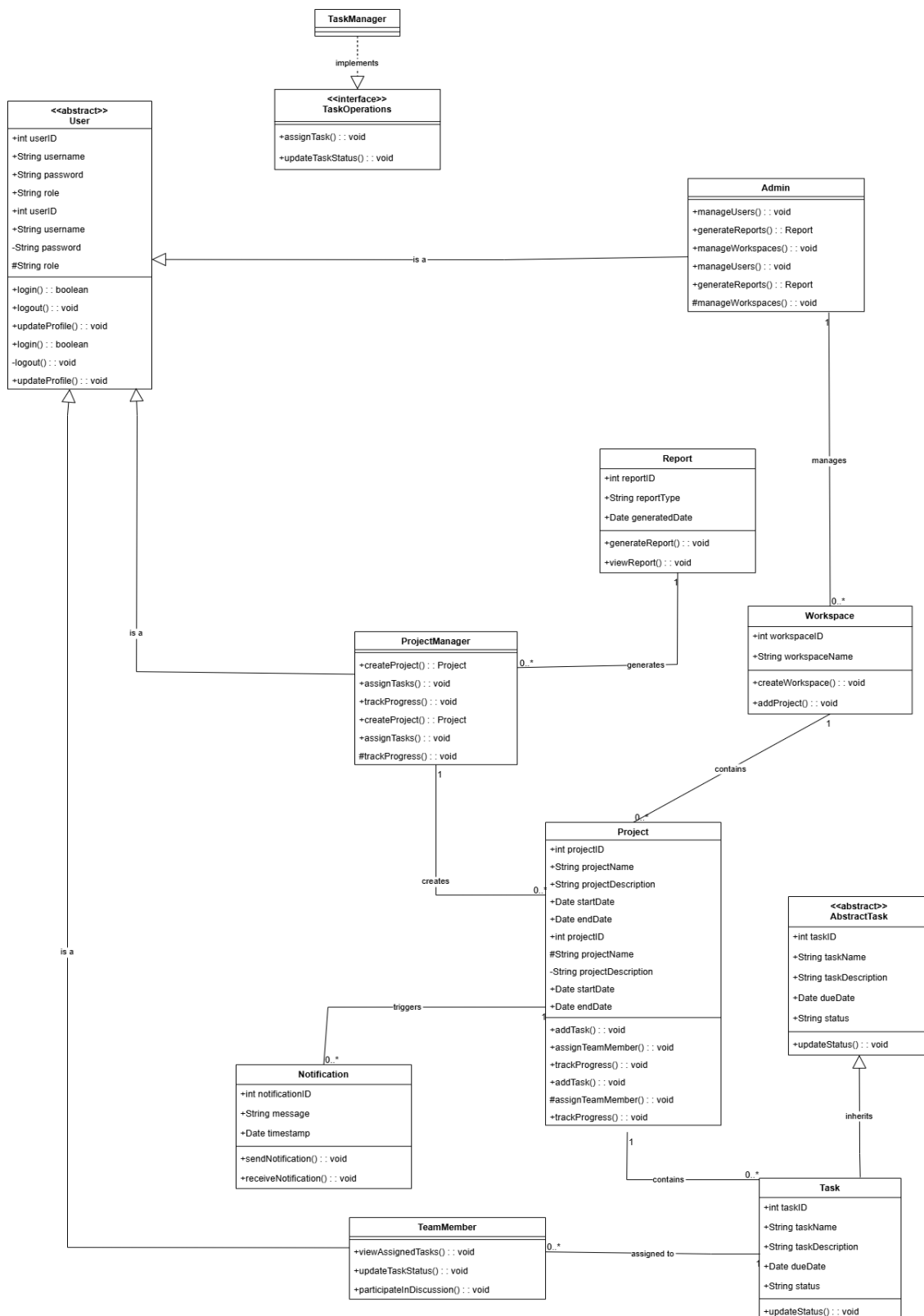
7. **Abstract Class:**

- A class that cannot be instantiated directly. It is meant to be a base class for other classes.
- **Notation:** Italicized class name or with the keyword <<abstract>>.

8. **Notes:**

- Used to add explanations or comments to clarify certain parts of the diagram.
- **Notation:** A rectangle with a folded corner, containing text.

4.2 Class Diagram of DWCP



4.4 Diagram Explanation Core Classes and Interfaces

1. User (Abstract Class)

- This is an abstract class representing a general user in the system.
- Attributes:
 - userID: Unique identifier for the user.
 - username: Username of the user.
 - password: User's password.
 - role: Role assigned to the user.
- Methods:
 - login(): Checks if the login is successful, returning a boolean.
 - logout(): Logs the user out.
 - updateProfile(): Allows the user to update their profile information.
- The User class is extended by the Admin, ProjectManager, and TeamMember classes.

2. TaskOperations (Interface)

- An interface implemented by TaskManager.
- Methods:
 - assignTask(): Allows a task to be assigned.
 - updateTaskStatus(): Enables the updating of a task's status.

3. TaskManager

- Implements the TaskOperations interface, handling task-related operations.

Specialized User Roles

1. Admin (Subclass of User)

- This class represents administrative users with management capabilities.
- Additional Methods:
 - manageUsers(): Manages the users in the system.
 - generateReport(): Generates a report.
 - manageWorkspace(): Manages workspaces within the system.
 - Other functionalities related to managing users, reports, and workspaces.

2. ProjectManager (Subclass of User)

- Represents a user responsible for managing projects.
- Additional Methods:
 - createProject(): Creates a new project.

- assignTask(): Assigns a task within a project.
- trackProgress(): Tracks the progress of tasks or projects.

3. TeamMember (Subclass of User)

- Represents users who are assigned tasks within a project.
- Additional Methods:
 - viewAssignedTasks(): Allows the team member to view tasks assigned to them.
 - updateTaskStatus(): Updates the status of assigned tasks.
 - participateInDiscussion(): Enables participation in project-related discussions.

Entities

1. Report

- Represents reports generated within the system.
- Attributes:
 - reportID: Unique identifier for the report.
 - reportType: Type of the report.
 - generatedDate: Date when the report was generated.
- Methods:
 - generateReport(): Generates a report.
 - viewReport(): Allows viewing of the report.

2. Workspace

- Contains and manages projects.
- Attributes:
 - workspaceID: Unique identifier for the workspace.
 - workspaceName: Name of the workspace.
- Methods:
 - createWorkspace(): Creates a new workspace.
 - addProject(): Adds a project to the workspace.

3. Project

- Represents a project within a workspace.
- Attributes:
 - projectID: Unique identifier for the project.
 - projectName: Name of the project.

- projectDescription: Description of the project.
 - startDate and endDate: Date range for the project.
- Methods:
 - addTask(): Adds a task to the project.
 - assignTeamMember(): Assigns a team member to the project.
 - trackProgress(): Tracks progress within the project.
- 4. **AbstractTask (Abstract Class)**
 - Represents a task within the system and serves as a base class for Task.
 - Attributes:
 - taskID: Unique identifier for the task.
 - taskName: Name of the task.
 - taskDescription: Description of the task.
 - dueDate: Due date for the task.
 - status: Current status of the task.
 - Methods:
 - updateStatus(): Updates the status of the task.
- 5. **Task (Subclass of AbstractTask)**
 - Represents specific tasks within a project, inheriting properties and methods from AbstractTask.
- 6. **Notification**
 - Represents notifications sent to users regarding task or project updates.
 - Attributes:
 - notificationID: Unique identifier for the notification.
 - message: Content of the notification.
 - timestamp: Date and time the notification was created.
 - Methods:
 - sendNotification(): Sends a notification.
 - receiveNotification(): Receives a notification.

Relationships

- **Generalization (Inheritance)**
 - Admin, ProjectManager, and TeamMember inherit from User.
 - Task inherits from AbstractTask.

- **Associations**

- TaskManager implements TaskOperations.
- Admin manages Report and Workspace.
- ProjectManager creates and manages Project, which contains tasks and assigns team members.
- Workspace contains multiple Project instances.
- Project triggers Notification and contains Task objects.
- TeamMember is assigned to Task within a Project.

This structure ensures a comprehensive task management system, where different user roles have specific responsibilities and interact with each other through well-defined relationships.

Artifact 5

Activity Diagram

5.1 Activity Diagram

An Activity Diagram is an actual kind of diagram within the Unified Modeling Language - UML, representing the visual form of workflow related to a process, system, or operation. This diagram expresses the sequence of activities and decision points and control flow or data throughout the system, which makes their inner flow logic clearer and the processes followed better easier to understand.

5.2 Fundamental Properties of Activity Diagrams

- The **flow of actions** represents the sequence of tasks or activities intended to be carried out.
- **Decision Points:** Includes decision nodes to show choices in the workflow (e.g., Yes/No conditions).
- **Parallelism:** Can represent many activities occurring concurrently.
- It centers on entities, or actors, that carry out certain actions.
- Visual simplicity uses **symbols and connectors** such that the illustration of process is clear and direct.

5.3 Basic Elements of an Activity Diagram

Starting Node (Start Node):

- Indicates where the action starts.
- Represented as a solid black circle.

Activity or Action Nodes:

- Represent specific tasks or operations to be carried out.
- Represented as rounded rectangles.

Control Flows:

Links between activity nodes are shown by arrows indicating the sequence of activities.

Decision Nodes:

- Illustrate the decision-making points in the flow: where different paths are chosen based

on certain conditions.

- Presented as a diamond formation.

Cluster nodes

- To merge a few input streams to a single output stream.
- Also written as a diamond shape

Fork and Join Nodes:

- Fork: Divides one flow into several parallel flows.
- Join: Combines several parallel flows into one.
- Seen as horizontal or vertical bars.

Terminal Node (Concluding Point):

Indicates that the activity is over. Represented as a circle containing a solid black dot within it.

Swim lanes:

Segment the diagram into distinct lanes to illustrate various roles, departments, or actors engaged in particular activities.

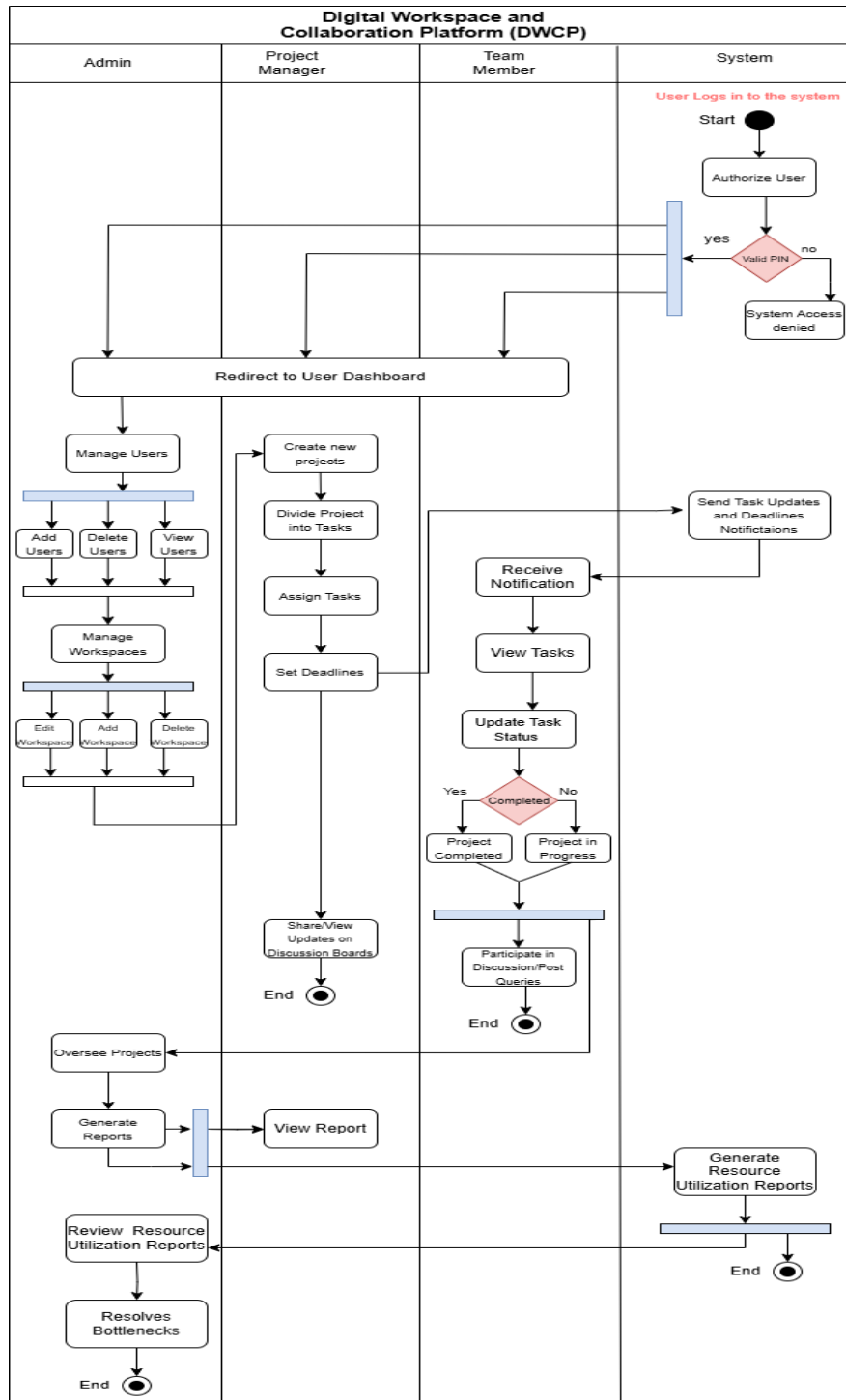
Entities or Information Streams:

It demonstrates generation or use of data in the process.

In "Digital Workspace and Collaboration Platform":

- Swim lanes distinguish tasks among different roles: Admin, Project Manager, Team Member, or System.
- Action Nodes, like Assign Task, Manage Users, or Update Task Status, all indicate different operations.
- Decision Nodes like Task Completed? show conditional branching.
- Final Node indicates where the process ends for a given workflow.

5.4 Diagram:



5.5 Diagram Explanation

1. User Authentication:

- **System role:**
 - The process starts with the user attempting to log in. The system verifies user credentials to authenticate the user.
 - Based on authentication, the user is redirected to their respective dashboard (Admin, Project Manager, or Team Member).

2. Admin Role:

- **Managing Users:**
 - Admin can add, update, or delete users, granting them specific roles within the platform.
- **Workspace Management:**
 - Admin oversees the creation and management of workspaces to ensure that projects and tasks are allocated to the appropriate teams.
- **Project and Report Overview:**
 - Admin has access to project performance and team productivity analytics.
 - They can generate reports for resource utilization and bottlenecks.

3. Project Manager Role:

- **Project Creation and Task Assignment:**
 - Project managers create new projects and divide them into tasks.
 - Tasks are assigned to team members along with deadlines.
- **Progress Monitoring:**
 - They track the status of tasks (e.g., Completed or In Progress) and receive updates via notifications.

- **Generating Reports:**
 - Managers can generate reports on task completion and productivity to assess team performance.

4. Team Member Role:

- **Task Management:**
 - Team members view their assigned tasks, receive notifications for task updates, and can update the status of their tasks.
- **Communication:**
 - They can participate in discussions via a project-specific discussion board.

5. Notifications:

- The system notifies relevant users (Team Members and Project Managers) about task updates, deadlines, and project changes.

6. Reports:

- **Admins and Project Managers:**
 - Generate various reports, such as productivity analysis and resource utilization trends.
- **Improvement Opportunities:**
 - Insights help identify areas where project efficiency can be improved.

Artifact 6

Sequence Diagram

6.1 Sequence Diagram

A **Sequence Diagram** is a type of **interaction diagram** in UML (Unified Modeling Language) that shows how objects or components in a system interact over time. It is used to model the flow of messages between different parts of a system in a particular sequence.

6.2 Key Elements of a Sequence Diagram:

1. Actors:

- **Actors** represent external entities (users, systems, or other components) that interact with the system being modeled. Actors are usually placed at the top of the diagram.

2. Objects/Instances:

- These are the objects or instances that participate in the interactions. Objects are placed below the actors and can be shown as rectangles with their name at the top (typically with a colon separating the object's name and class).

3. Lifelines:

- **Lifelines** represent the presence of an object over time and are shown as vertical dashed lines that extend from each object. They indicate how long an object exists during the sequence.

4. Messages:

- **Messages** are arrows representing the interaction between objects. The direction of the arrow shows the direction of the message, and the label on the arrow indicates the operation or function being called. Messages can be:
 - **Synchronous:** Represented by a solid arrowhead, indicating the sender waits for a response.
 - **Asynchronous:** Represented by a stick arrowhead, indicating the sender does not wait for a response.

5. Activation Bars:

- **Activation bars** are tall rectangles that appear along the lifeline. They represent when an object is actively performing an action in response to a message.

6. **Return Messages:**

- **Return messages** are dashed arrows showing the response sent back to the caller. They indicate the result of the action.

7. **Frames:**

- **Frames** are rectangular boxes that can be used to group a part of the sequence to show conditions (alt), loops (loop), or options (opt). Frames are used to represent combined fragments.

8. **Conditions and Loops:**

- **Conditions:** The fragment (like **alt** or **opt**) can show different branches of execution based on conditions.
- **Loops:** The **loop** fragment can represent repeated behavior (e.g., a function called multiple times).

6.2 Diagram explanation

The diagram is a sequence diagram, illustrating the interactions between different roles (Admin, Project Manager, Team Member, System, Workspace, Notification, and Report) in a project management system. Here's a detailed explanation of the diagram:

1. User Login:

- **Admin Login:**
 - Admin sends a login request.
 - If successful, the system redirects to the Admin Dashboard.
 - If unsuccessful, a login error message is displayed.
- **Project Manager Login:**
 - Project Manager sends a login request.
 - If successful, the system redirects to the Project Manager Dashboard.
 - If unsuccessful, a login error message is displayed.
- **Team Member Login:**
 - Team Member sends a login request.
 - If successful, the system redirects to the Team Member Dashboard.
 - If unsuccessful, a login error message is displayed.

2. Project Creation and Task Management:

- Admin/Project Manager creates a new project.
- System shows the project creation form and processes the project details.
- After successful project creation, tasks are initialized.
- Task creation involves showing a task form, submitting task details, and then confirming the creation.
- Once the task is created, it is assigned to a Team Member.
- Team Member updates the task status.
- The system tracks task completion and updates the task status successfully.

3. Project Progress Tracking:

- Project progress is monitored based on task completion.
- If all tasks are completed, the project is marked as complete.

- If some tasks are pending, the project is marked as "in progress".

4. Discussion Updates:

- Team Members can post updates in discussion threads.
- Replies can be made to existing updates, and responses are logged.

5. Workspace Management:

- New workspaces can be created by Admin or Project Manager.
- Users can be added or removed from the workspace.
- The system confirms successful user addition or removal.

6. Notifications:

- The system sends notifications for various events:
 - Task deadline reminders.
 - Task status updates.
 - Project scope change notifications.
 - Project status updates.

7. Report Generation:

- Admin or Project Manager can request different reports:
 - **Team Productivity Report:** Shows productivity metrics if the report is available, otherwise displays an error.
 - **Project Progress Report:** Shows the overall progress of the project.
 - **Resource Utilization Report:** Shows insights if available, otherwise indicates no insights.

This diagram captures the typical flow of a project management system, focusing on task assignment, project tracking, workspace management, notifications, and reporting. Each actor's interaction with the system is shown in a linear flow, depicting the sequence of actions and responses.

Artifact 6

State Transition

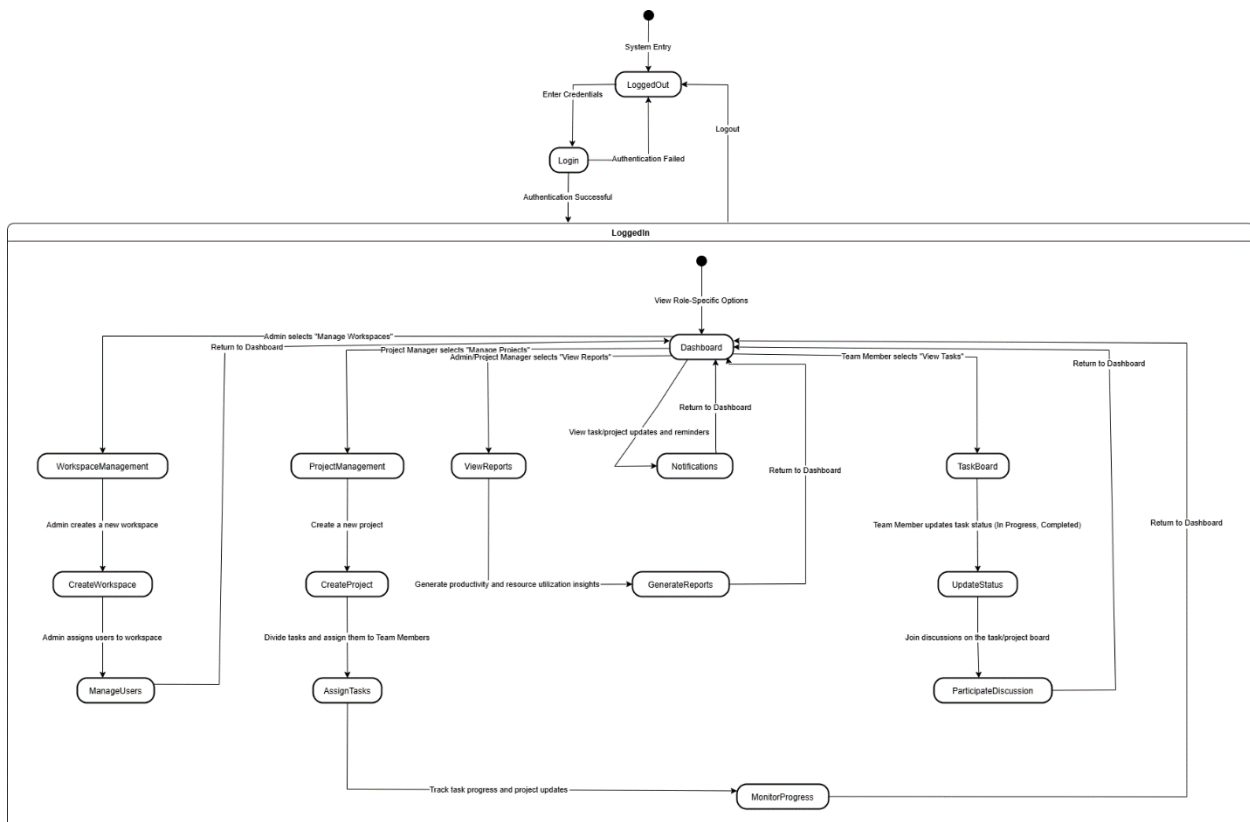
Diagram

6.1 State Transition Diagram

A **state transition diagram** shows how a system or component transitions between various states in response to events or actions. Each "state" represents a specific condition or phase, and the transitions show the movement between these states based on triggers (like user actions).

For example:

1. A user starts in the "LoggedOut" state.
2. When they enter their credentials, the system moves to a "Login" state.
3. If authentication is successful, the system transitions to "LoggedIn."
4. If authentication fails, they stay in the "LoggedOut" state.



6.2 Explanation of the Diagram

Here's a breakdown of the **state transition diagram** I created for the **Digital Workspace and Collaboration Platform (DWCP)**.

Initial State: LoggedOut

- The user starts in the LoggedOut state (the system is idle and waiting for input).
- The user can:
 - **Enter credentials:** This triggers the system to move to the Login state for authentication.

State: Login

- In the Login state, the system checks the credentials.
- Two possible transitions:
 - If **authentication is successful**, the user enters the LoggedIn state.
 - If **authentication fails**, the user returns to LoggedOut and can retry logging in.

State: LoggedIn

Once the user is authenticated, they see a **Dashboard** with options based on their role:

1. Admin Options:

- **Workspace Management:**
 - Admin can select "Manage Workspaces."
 - Inside this workflow:
 - Admin can **create a new workspace** (CreateWorkspace state).
 - Admin can **assign users** to the workspace (ManageUsers state).
 - Once finished, the Admin returns to the **Dashboard**.
- **View Reports:**
 - Admin can generate productivity or resource reports (GenerateReports state).
 - Returns to the **Dashboard** after viewing reports.

2. Project Manager Options:

- **Project Management:**
 - The Project Manager can:
 - **Create new projects** (CreateProject state).
 - **Assign tasks** to Team Members (AssignTasks state).
 - **Monitor progress** of tasks and projects (MonitorProgress state).
 - Returns to the **Dashboard** after task updates.
- **View Reports:**
 - Similar to the Admin, Project Managers can generate reports on progress and productivity.

3. Team Member Options:

- **Task Board:**
 - A Team Member can:
 - **View assigned tasks** (TaskBoard state).
 - **Update task status** (e.g., "In Progress," "Completed") in the UpdateStatus state.
 - **Join discussions** on tasks in the ParticipateDiscussion state.
 - Returns to the **Dashboard** after completing tasks or discussions.

4. Notifications for All Users:

- Any user can check **Notifications** for updates like deadlines or task changes.
- Returns to the **Dashboard** afterward.

Final Transition: Logout

- From any state in the LoggedIn flow, users can choose to **logout**.
- This returns the system to the LoggedOut state.

Artifact7

MVC Framework

7.1 MVC Framework:

MVC (Model-View-Controller) is a design pattern used in software development to separate an application into three interconnected components: Model, View, and Controller. This architecture promotes the separation of concerns, which enhances modularity and maintainability. The Model represents the data and business logic of the application, the View is responsible for displaying the data to the user, and the Controller manages user input and updates the Model and View accordingly. By separating these concerns, MVC allows developers to work on each component independently, improving the flexibility and scalability of the application.

7.2 Application in DWCP:

7.2.1 View (User Interface Layer):

Description: Displays the data to the user, presenting the interface and visualizing the information from the Model. The View displays data to the user. It provides a dashboard and task list views.

```
# views.py

class UserDashboard:
    def show(self, user):
        print(f"Dashboard for {user.username} ({user.role})")
        if user.role == "Admin":
            print("Manage workspaces and users.")
        elif user.role == "Project Manager":
            print("Create projects and assign tasks.")
        elif user.role == "Team Member":
            print("View assigned tasks and update status.")

class TaskListView:
    def show(self, project):
        print(f"\nTasks for Project: {project.project_name}")
        for task in project.tasks:
            print(f"Task: {task.task_name}, Assignee: {task.assignee.username}, Status: {task.status}")
```

7.2.2 Controller:

Description: Manages user input, processes it, and updates both the Model and View, acting as an intermediary between the two. The Controller handles user interactions and updates the Model and View.

```
# controllers.py

from models import User, Project, Task
from views import UserDashboard, TaskListView

class UserController:
    def __init__(self, user):
        self.user = user
        self.dashboard = UserDashboard()

    def show_dashboard(self):
        self.dashboard.show(self.user)

class ProjectController:
    def __init__(self, user):
        self.user = user
        self.projects = []

    def create_project(self, project_name, due_date):
        if self.user.role == "Project Manager":
            new_project = Project(project_name, due_date)
            self.projects.append(new_project)
            print(f"Project '{project_name}' created.")
        else:
            print("Only Project Managers can create projects.")

    def add_task(self, project, task_name, assignee):
        if self.user.role == "Project Manager":
            task = Task(task_name, assignee)
            project.add_task(task)
```



```

        print(f"Task '{task_name}' added to project '{project.project_name}'.")
    else:
        print("Only Project Managers can assign tasks.")

def show_project_tasks(self, project):
    task_list_view = TaskListView()
    task_list_view.show(project)

class TaskController:
    def __init__(self, user):
        self.user = user

    def update_task_status(self, task, status):
        if self.user.role == "Team Member" and task.assignee == self.user:
            task.update_status(status)
            print(f"Task '{task.task_name}' updated to {status}.")
        else:
            print("Only the assigned team member can update the task status.")

```

7.2.3 Model:

Description: Represents the data and business logic of the application, handling operations and rules related to data management. The Model handles the data structure and logic for users, tasks, and projects.

```

# models.py

class User:
    def __init__(self, username, role):
        self.username = username
        self.role = role

class Project:
    def __init__(self, project_name, due_date):
        self.project_name = project_name

```

```

        self.due_date = due_date

        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

class Task:
    def __init__(self, task_name, assignee, status="Not Started"):
        self.task_name = task_name
        self.assignee = assignee
        self.status = status

    def update_status(self, status):
        self.status = status

```

7.2.4 Main Application:

```

# main.py

from models import User, Project, Task
from controllers import UserController, ProjectController, TaskController

# Create users
admin = User("John", "Admin")
pm = User("Alice", "Project Manager")
team_member = User("Bob", "Team Member")

# Create the controllers
admin_controller = UserController(admin)
pm_controller = UserController(pm)
team_member_controller = UserController(team_member)

# Show dashboards
admin_controller.show_dashboard()
pm_controller.show_dashboard()

```

```
team_member_controller.show_dashboard()

# Create a project and add tasks
project_controller = ProjectController(pm)
project_controller.create_project("DWCP Development", "2024-12-01")
project = project_controller.projects[0]

# Add tasks to the project
project_controller.add_task(project, "Design UI", team_member)

# Team Member updates task status
task_controller = TaskController(team_member)
task_controller.update_task_status(project.tasks[0], "In Progress")

# Show project tasks
project_controller.show_project_tasks(project)
```

Artifact 8

GRASP Pattern

8.1 GRASP Patterns

GRASP stands for General Responsibility Assignment Software Patterns and is a methodology to guide software designers in making responsibility assignments in classes and objects in an object-oriented system. By utilization of GRASP patterns, systems are able to develop low coupling, high cohesion, and flexibility, thus creating maintainable and scalable designs.

GRASP patterns are used in the DWCP for user roles, functionality related to creating tasks and projects, or to send notifications, among other functionality. Each of these patterns would be stated in the context of your system, illustrated with examples and simple codes.

Creator

Description: It lets a class contain and directly access the information required to create an object

8.2 Application in DWCP:

1. Admin Creating Users

Description: The Admin class creates new users and assigns roles.

```
class User:
    def __init__(self, username, role=None):
        self.username = username
        self.role = role

class Admin:
    def __init__(self):
        self.users = []

    def create_user(self, username, role):
        user = User(username, role)
        self.users.append(user)
        return user

admin = Admin()
```

```
user1 = admin.create_user("Azka", "Admin")
print(user1.username, user1.role)
```

2. Project Managers Creating Projects

Description: The ProjectManager class creates projects within a workspace.

```
class Project:
    def __init__(self, name):
        self.name = name

class ProjectManager:
    def __init__(self):
        self.projects = []

    def create_project(self, project_name):
        project = Project(project_name)
        self.projects.append(project)
        return project

manager = ProjectManager()
project = manager.create_project("New Website")
print(project.name)
```

3. Task Creation

Description: The Project class creates tasks and assigns them to team members.

```
class Task:
    def __init__(self, name, assigned_user=None):
        self.name = name
```

```

        self.assigned_user = assigned_user

class ProjectWithTasks:
    def __init__(self, name):
        self.name = name
        self.tasks = []

    def create_task(self, task_name, assigned_user=None):
        task = Task(task_name, assigned_user)
        self.tasks.append(task)
        return task

project = ProjectWithTasks("Build API")
task = project.create_task("Design Database", "myApp")
print(task.name, task.assigned_user)

```

4. Discussion Board Posts

Description: The DiscussionBoard class creates and links posts to a project.

```

class Post:
    def __init__(self, user, message):
        self.user = user
        self.message = message

class DiscussionBoardWithPosts:
    def __init__(self, project_name):
        self.project_name = project_name
        self.posts = []

    def create_post(self, user, message):

```

```
post = Post(user, message)
self.posts.append(post)
return post

discussion_board = DiscussionBoardWithPosts("Website Redesign")
post = discussion_board.create_post("Azka", "Let's finalize the homepage.")
print(post.user, post.message)
```

Output

```
⇒ Azka Admin
   New Website
   Design Database myApp
   Azka Let's finalize the homepage.
```

Information Expert

Description: Assigns responsibility to the class that has the required data to fulfill it.

8.3 Application in DWCP:

1. Task Management

Description: The Task class manages task-specific information and provides methods to update it.

```
class Task:
    def __init__(self, title, assignee=None, deadline=None):
        self.title = title
        self.assignee = assignee
        self.deadline = deadline
        self.status = "Pending"
```



```

def update_status(self, status):
    self.status = status

def update_deadline(self, new_deadline):
    self.deadline = new_deadline

def assign_user(self, user):
    self.assignee = user

task = Task("Design Homepage", "Azka", "2024-11-30")
task.update_status("In Progress")
task.update_deadline("2024-12-05")
print(task.title, task.status, task.deadline, task.assignee)

```

2. Project Progress Tracking

Description: The Project class calculates progress based on the status of associated tasks.

```

class Project:
    def __init__(self, name):
        self.name = name
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def calculate_progress(self):
        if not self.tasks:
            return "No tasks yet."
        completed_tasks = sum(1 for task in self.tasks if task.status == "Completed")
        return (completed_tasks / len(self.tasks)) * 100

# Usage Example

```

```

project = Project("New Website")
task1 = Task("Build API")
task2 = Task("Create UI")
project.add_task(task1)
project.add_task(task2)

task1.update_status("Completed")
progress = project.calculate_progress()
print(f"Project '{project.name}' is {progress:.2f}% complete.")

```

3. User Details Management

Description: The User class stores roles, assigned tasks, and workspace memberships.

```

class User:
    def __init__(self, username):
        self.username = username
        self.role = None
        self.tasks = []
        self.workspaces = []

    def assign_role(self, role):
        self.role = role

    def assign_task(self, task):
        self.tasks.append(task)

    def join_workspace(self, workspace):
        self.workspaces.append(workspace)

user = User("Azka")
user.assign_role("Developer")
user.assign_task("Fix bugs in API")
user.join_workspace("IT Workspace")
print(user.username, user.role, user.tasks, user.workspaces)

```

Output

```
➡ Design Homepage In Progress 2024-12-05 Azka  
Project 'New Website' is 50.00% complete.  
Azka Developer ['Fix bugs in API'] ['IT Workspace']
```

8.4 Low Coupling

Description: Reduce dependency between components to improve modularity, flexibility, and maintainability.

Application in DWCP:

1. Notifications System

Description: The NotificationService handles notifications independently from task or workspace logic.

```
class NotificationService:  
    def send_notification(self, message, recipient):  
        print(f"Notification sent to {recipient}: {message}")  
  
notifier = NotificationService()  
notifier.send_notification("Task deadline is tomorrow!", "Alice")
```

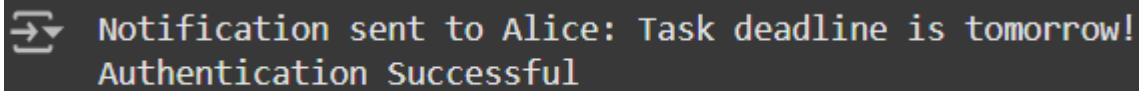
2. Authentication Module

Description: Authentication is managed in a separate class to decouple it from user management.

```
class AuthenticationService:  
    def authenticate(self, username, password):  
        return username == "Azka" and password == "1234"
```

```
auth_service = AuthenticationService()
is_authenticated = auth_service.authenticate("Azka", "1234")
print("Authentication Successful" if is_authenticated else "Authentication Failed")
```

Output



```
⇒ Notification sent to Alice: Task deadline is tomorrow!
Authentication Successful
```

8.5 High Cohesion

Description: This makes sure each class retains a unique and focused responsibility.

Application in DWCP:

1. Task Responsibility

Description: The Task class focuses on task-specific operations, such as updating the status or assigning users.

```
class Task:
    def __init__(self, name, assigned_user=None):
        self.name = name
        self.status = "Pending"
        self.assigned_user = assigned_user

    def update_status(self, status):
        self.status = status

    def assign_user(self, user):
        self.assigned_user = user

# Usage Example
task = Task("Design Wireframe")
```

```
task.assign_user("Azka")
task.update_status("In Progress")
print(task.name, task.status, task.assigned_user)
```

2. Project Management

Description: The Project class manages tasks within a project but delegates notifications or other responsibilities to separate services.

```
class Project:
    def __init__(self, name):
        self.name = name
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def list_tasks(self):
        return [task.name for task in self.tasks]

# Usage Example
project = Project("New Website")
task1 = Task("Create Homepage")
task2 = Task("Build API")
project.add_task(task1)
project.add_task(task2)
print(project.list_tasks())
```

3. User Role Management

Description: The RoleManager class is responsible for validating and assigning roles to users.

```
class RoleManager:
    def __init__(self):
        self.roles = { }

    def assign_role(self, user, role):
        self.roles[user] = role

    def get_role(self, user):
        return self.roles.get(user, "No Role Assigned")

# Usage Example
role_manager = RoleManager()
role_manager.assign_role("Azka", "Admin")
print(role_manager.get_role("Azka"))
```

4. Discussion Board

Description: The DiscussionBoard class manages message threads and user interactions related to a specific project.

```
class DiscussionBoard:
    def __init__(self, project_name):
        self.project_name = project_name
        self.messages = []

    def add_message(self, user, message):
        self.messages.append(f'{user}: {message}')
```

```

def display_messages(self):
    return "\n".join(self.messages)

board = DiscussionBoard("New Website")
board.add_message("Azka", "Let's finalize the homepage design.")
board.add_message("Team Member", "I'll update the API documentation.")
print(board.display_messages())

```

Output

```

⇒ Design Wireframe In Progress Azka
  ['Create Homepage', 'Build API']
  Admin
  Azka: Let's finalize the homepage design.
  Team Member: I'll update the API documentation.

```

Polymorphism

Description: The Polymorphism pattern ensures that behavior is determined at runtime through inheritance or interfaces, enabling flexibility.

Application in DWCP:

1. Role-Based Access Control

Description: Different user roles implement their unique access logic via polymorphism.

```

class User:
    def access_system(self):
        pass

```

```

class Admin(User):
    def access_system(self):
        print("Admin: Full access to manage users, projects, and reports.")

class ProjectManager(User):
    def access_system(self):
        print("Project Manager: Access to create and manage projects.")

class TeamMember(User):
    def access_system(self):
        print("Team Member: Access to view and update assigned tasks.")

# Example Usage
users = [Admin(), ProjectManager(), TeamMember()]
for user in users:
    user.access_system()

```

2. Notification Types

Description: Different notification types implement their unique sending logic using the same structure, ensuring code reusability and scalability.

```

class Notification:
    def send(self):
        pass

class EmailNotification(Notification):
    def send(self):
        print("Sending email notification...")

class SMSNotification(Notification):
    def send(self):
        print("Sending SMS notification...")

```



```
# Example Usage
notifications = [EmailNotification(), SMSNotification()]
for notification in notifications:
    notification.send()
```

Output

```
➡ Admin: Full access to manage users, projects, and reports.
   Project Manager: Access to create and manage projects.
   Team Member: Access to view and update assigned tasks.
   Sending email notification...
   Sending SMS notification...
```

Controller:

Description: Assigns the responsibility of handling system events to a class representing a use case or a specific role in the system.

Application in DWCP:

1. Handling User Login Requests:

Description: Manages authentication events by validating user credentials.

```
class AuthController:
    def handle_login(self, username, password):
        if username == "admin" and password == "password123":
            return "Login successful"
        return "Invalid credentials"

# Example Usage
controller = AuthController()
response = controller.handle_login("admin", "password123")
print(response)
```

Output

A terminal window with a dark background. On the left, there is a green icon of a terminal. To its right, the text "Login successful" is displayed in a light green monospace font.

2. Managing File Uploads:

Description: Handles file upload events and stores information about user uploads

```
class FileController:
    def handle_file_upload(self, user, file):
        print(f"User {user} uploaded {file.name}")
        return f"File {file.name} uploaded successfully."

# Example Usage
```

```
class File:
    def __init__(self, name):
        self.name = name

file_controller = FileController()
file = File("document.pdf")
response = file_controller.handle_file_upload("Alice", file)
print(response)
```

Output

```
➔ User Alice uploaded document.pdf
   File document.pdf uploaded successfully.
```

3. Coordinating Real-time Chat:

Description: Processes and delivers chat messages between users in real time.

```
class ChatController:
    def handle_message(self, sender, receiver, message):
        print(f"Message from {sender} to {receiver}: {message}")
        return f"Message delivered to {receiver}"

# Example Usage
chat_controller = ChatController()
response = chat_controller.handle_message("Alice", "Bob", "Hello, Bob!")
print(response)
```

Output

```
➔ Message from Alice to Bob: Hello, Bob!
   Message delivered to Bob
```

4. Assigning Task to a Team Member:

Description: Assigns tasks to team members and tracks assignments.

```

class TaskController:
    def assign_task(self, task_name, assignee):
        print(f"Task '{task_name}' assigned to {assignee}")
        return f"Task '{task_name}' successfully assigned to {assignee}"

# Example Usage
task_controller = TaskController()
response = task_controller.assign_task("Design Homepage", "Charlie")
print(response)

```

Output

```

➡ Task 'Design Homepage' assigned to Charlie
   Task 'Design Homepage' successfully assigned to Charlie

```

Pure Fabrication

Description: Introduces an artificial class to handle a responsibility that doesn't fit naturally into other classes, improving design and maintainability.

Application in DWCP:

1. Data Encryption Utility:

Description: Handles data encryption tasks to enhance security, independent of core business logic.


```

class EncryptionUtility:
    def encrypt(self, data):
        return ''.join(chr(ord(char) + 3) for char in data)

# Example Usage
encryption_utility = EncryptionUtility()
encrypted_data = encryption_utility.encrypt("sensitiveData")
print(encrypted_data)

```

Output

 vhqvlwlyhGdwd


2. Email Notification Service:

Description: Sends email notifications as a utility class, separated from the main system logic.

```
class EmailService:
    def send_email(self, recipient, subject, body):
        print(f"Sending email to {recipient} with subject '{subject}': {body}")
        return "Email sent successfully"

# Example Usage
email_service = EmailService()
response = email_service.send_email("user@example.com", "Welcome!", "Thank you for joining our platform.")
print(response)
```

Output

 Sending email to user@example.com with subject 'Welcome!': Thank you for joining our platform.
Email sent successfully

3. Logging Service:

Description: Logs system events and errors, decoupled from other system components.

```
class Logger:
    def log(self, message):
        print(f"LOG: {message}")

# Example Usage
logger = Logger()
logger.log("User logged in successfully.")
```

Output

```
LOG: User logged in successfully.
```

4. PDF Report Generator:

Description: Generates PDF reports as a separate utility, making it reusable across different modules.

```
class PDFReportGenerator:
    def generate_report(self, title, content):
        print(f"Generating PDF report titled '{title}' with content: {content}")
        return f"Report '{title}' created"

# Example Usage
pdf_generator = PDFReportGenerator()
response = pdf_generator.generate_report("Monthly Report", "Report content goes here.")
print(response)
```

Output

```
Generating PDF report titled 'Monthly Report' with content: Report content goes here.
Report 'Monthly Report' created
```

Indirection:

Description: Assigns responsibility to an intermediary object to reduce direct coupling between classes, improving flexibility.

Application in DWCP:

1. Payment Processing via Payment Gateway:

Description: Uses an intermediary class to route payment requests to the appropriate payment gateway without direct dependency.

```
class PaymentGateway:
```

```

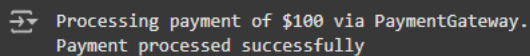
def process_payment(self, amount):
    print(f"Processing payment of ${amount} via PaymentGateway.")
    return "Payment processed successfully"

class PaymentController:
    def handle_payment(self, amount):
        gateway = PaymentGateway()
        return gateway.process_payment(amount)

# Example Usage
payment_controller = PaymentController()
response = payment_controller.handle_payment(100)
print(response)

```

Output



```

Processing payment of $100 via PaymentGateway.
Payment processed successfully

```

2. Database Query Handler:

Description: Routes database queries through an intermediary class to abstract the database interactions.

```

class Database:
    def execute_query(self, query):
        print(f"Executing query: {query}")
        return "Query executed"

class QueryHandler:
    def handle_query(self, query):
        database = Database()
        return database.execute_query(query)

# Example Usage

```

```
query_handler = QueryHandler()
response = query_handler.handle_query("SELECT * FROM users")
print(response)
```

Output

```
➦ Executing query: SELECT * FROM users
Query executed
```

3. Email Sending through SMTP Server

Description: Routes email sending through an intermediary class to abstract SMTP server details.

```
class SMTPServer:
    def send_email(self, recipient, subject, body):
        print(f"Sending email to {recipient} with subject '{subject}' via SMTP.")
        return "Email sent"

class EmailController:
    def send_email(self, recipient, subject, body):
        smtp_server = SMTPServer()
        return smtp_server.send_email(recipient, subject, body)

# Example Usage
email_controller = EmailController()
response = email_controller.send_email("user@example.com", "Welcome!", "Thank you for signing up!")
print(response)
```

Output

```
➦ Sending email to user@example.com with subject 'Welcome!' via SMTP.
Email sent
```

4. User Authentication via Authentication Service:

Description: Uses an intermediary class to handle user authentication, separating the logic from the user interface.


```

class AuthService:
    def authenticate(self, username, password):
        print(f"Authenticating user {username}.")
        return "User authenticated"

class AuthController:
    def handle_authentication(self, username, password):
        auth_service = AuthService()
        return auth_service.authenticate(username, password)

# Example Usage
auth_controller = AuthController()
response = auth_controller.handle_authentication("alice", "password123")
print(response)

```

Output

```

➡ Authenticating user alice.
   User authenticated

```

Protected Variations

Description: Protects elements from changes in other parts of the system by encapsulating variability behind interfaces or abstract classes.

Application in DWCP:

1. User Notification System with Abstract Notifier:

Description: Uses an abstract class to protect the notification system from changes in the notification delivery method (e.g., email or SMS).

```

from abc import ABC, abstractmethod

class Notifier(ABC):
    @abstractmethod
    def send_notification(self, message):

```

```

    pass

class EmailNotifier(Notifier):
    def send_notification(self, message):
        print(f"Sending email: {message}")

class SMSNotifier(Notifier):
    def send_notification(self, message):
        print(f"Sending SMS: {message}")

class NotificationService:
    def __init__(self, notifier: Notifier):
        self.notifier = notifier

    def notify(self, message):
        self.notifier.send_notification(message)

# Example Usage
email_notifier = EmailNotifier()
sms_notifier = SMSNotifier()

email_service = NotificationService(email_notifier)
email_service.notify("Welcome to the platform!")

sms_service = NotificationService(sms_notifier)
sms_service.notify("You have a new message!")

```

Output

```

🔏 Sending email: Welcome to the platform!
    Sending SMS: You have a new message!

```

2. Payment Gateway with Interface for Different Providers:

Description: Protects the payment system from changes by using an interface for different payment providers.

```
from abc import ABC, abstractmethod

class PaymentGateway(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class StripeGateway(PaymentGateway):
    def process_payment(self, amount):
        print(f"Processing payment of ${amount} via Stripe.")
        return "Payment successful"

class PayPalGateway(PaymentGateway):
    def process_payment(self, amount):
        print(f"Processing payment of ${amount} via PayPal.")
        return "Payment successful"

class PaymentProcessor:
    def __init__(self, gateway: PaymentGateway):
        self.gateway = gateway

    def process(self, amount):
        return self.gateway.process_payment(amount)

# Example Usage
stripe_processor = PaymentProcessor(StripeGateway())
print(stripe_processor.process(100))

paypal_processor = PaymentProcessor(PayPalGateway())
print(paypal_processor.process(200))
```

Output

```
Processing payment of $100 via Stripe.  
Payment successful  
Processing payment of $200 via PayPal.  
Payment successful
```

3. Team Communication System with Multiple Channels:

Description: Protects the communication system from changes by using an interface for different communication channels (e.g., Slack, Teams, Email).

```
from abc import ABC, abstractmethod

class CommunicationChannel(ABC):
    @abstractmethod
    def send_message(self, message, user):
        pass

class SlackChannel(CommunicationChannel):
    def send_message(self, message, user):
        print(f"Sending Slack message to {user}: {message}")

class TeamsChannel(CommunicationChannel):
    def send_message(self, message, user):
        print(f"Sending Teams message to {user}: {message}")

class EmailChannel(CommunicationChannel):
    def send_message(self, message, user):
        print(f"Sending Email message to {user}: {message}")

class CommunicationService:
    def __init__(self, channel: CommunicationChannel):
        self.channel = channel

    def send(self, message, user):
        self.channel.send_message(message, user)
```

```
# Example Usage
slack_channel = SlackChannel()
teams_channel = TeamsChannel()
email_channel = EmailChannel()

communication_service = CommunicationService(slack_channel)
communication_service.send("Please review the new document", "Alice")

communication_service = CommunicationService(teams_channel)
communication_service.send("Let's have a call tomorrow", "Bob")

communication_service = CommunicationService(email_channel)
communication_service.send("Your weekly report is ready", "Charlie")
```

Output

```
➦ Sending Slack message to Alice: Please review the new document
Sending Teams message to Bob: Let's have a call tomorrow
Sending Email message to Charlie: Your weekly report is ready
```

4. Document Sharing System with Different Permissions:

Description: Protects the document sharing system from changes by using an interface for different permission levels (e.g., View, Edit, Admin).

```
from abc import ABC, abstractmethod

class Permission(ABC):
    @abstractmethod
    def share_document(self, document, user):
        pass

class ViewerPermission(Permission):
    def share_document(self, document, user):
        print(f"Sharing document '{document}' with {user} for viewing only.")

class EditorPermission(Permission):
```

```

def share_document(self, document, user):
    print(f"Sharing document '{document}' with {user} for editing.")

class AdminPermission(Permission):
    def share_document(self, document, user):
        print(f"Sharing document '{document}' with {user} for full access.")

class DocumentService:
    def __init__(self, permission: Permission):
        self.permission = permission

    def share(self, document, user):
        self.permission.share_document(document, user)

# Example Usage
viewer_permission = ViewerPermission()
editor_permission = EditorPermission()
admin_permission = AdminPermission()

doc_service = DocumentService(viewer_permission)
doc_service.share("Project Plan", "John")

doc_service = DocumentService(editor_permission)
doc_service.share("Design Specs", "Alice")

doc_service = DocumentService(admin_permission)
doc_service.share("Internal Report", "Admin")

```

Output

```

➦ Sharing document 'Project Plan' with John for viewing only.
Sharing document 'Design Specs' with Alice for editing.
Sharing document 'Internal Report' with Admin for full access.

```

References:

- **Auth0.** (n.d.). Authentication and authorization. Retrieved from <https://auth0.com>
- **Asana.** (n.d.). API documentation. Retrieved from <https://developers.asana.com>
- **Trello.** (n.d.). Developer documentation. Retrieved from <https://developer.atlassian.com/cloud/trello>
- **Slack.** (n.d.). API for collaboration. Retrieved from <https://api.slack.com>
- **Microsoft.** (n.d.). Microsoft Graph API documentation. Retrieved from <https://learn.microsoft.com/en-us/graph>
- **Khan Academy.** (n.d.). MVC framework basics. Retrieved from <https://www.khanacademy.org/computing>
- **OODesign.** (n.d.). GRASP design patterns. Retrieved from <https://www.oodesign.com>
- **FastAPI.** (n.d.). Modern web framework for APIs. Retrieved from <https://fastapi.tiangolo.com>
- **Django Software Foundation.** (n.d.). Django: The web framework for perfectionists. Retrieved from <https://www.djangoproject.com>
- **GitHub.** (n.d.). Docs: Webhooks and API integration. Retrieved from <https://docs.github.com>