CONTENT MANAGEMENT SYSTEM

Software Construction And Development

BSSE 5TH SEMESTER

REPORT (Assignment 03)

Course Instructor: Sir FAROOQ IQBAL

Group Members:

1. **Syeda Anshrah Gillani (1337-2021) - (Group Leader)**
2. **Umema Mujeeb (2396-2021)**
3. **Maheen Ali (1589-2021)**
4. **Areej Asif (2276-2021)**

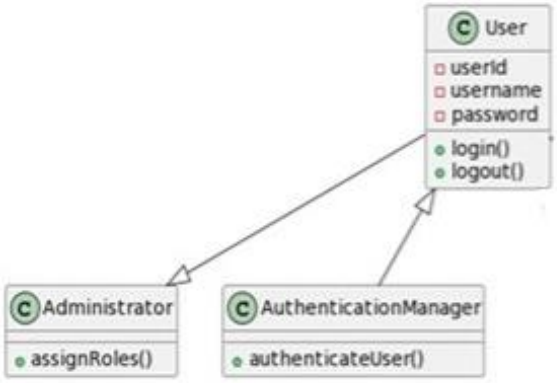
15/1/2024

Internal Routine Designs to Implementation:

- ***Chosen Programming Language:*** **" Python "**
- ***Reason:***

Python's **readability, versatility, extensive standard library, web development frameworks, community support, database connectivity, and security features** collectively make it a well-suited language for the successful implementation of internal routines .

(1) UserManagement Subsystem Class



Implementation:

1. User Class Routines:

```
import sqlite3
import uuid  # For generating unique IDs
from datetime import datetime  # For timestamp

class User:
    def createUser(self):

        user_details = self.gatherUserDetails()
        self.validateUserDetails(user_details)
        user_details['UserID'] = self.generateUniqueUserID()
        return self.createUserObject(user_details)

    def gatherUserDetails(self):
        # Implementation to gather user details from the user interface.
        user_details = {}
        user_details['UserName'] = input("Enter UserName: ")
        user_details['Email'] = input("Enter Email: ")
        return user_details
```

```
    def validateUserDetails(self, user_details):
        # Implementation to validate user details.
        # For simplicity, let's assume both UserName and Email are required.
        if not user_details.get('UserName') or not user_details.get('Email'):
            raise ValidationErrors("UserName and Email are required.")

    def generateUniqueUserID(self):
        # Implementation to generate a unique user ID.
        return str(uuid.uuid4())

    def saveUserDetailsToDatabase(self, user_details):
        # Implementation to save user details in the database.
        try:
            conn = sqlite3.connect("user_database.db")
            cursor = conn.cursor()
```

```
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS Users (
                    UserID TEXT PRIMARY KEY,
                    UserName TEXT,
                    Email TEXT,
                    CreatedAt TEXT
                )
            """)

            cursor.execute("""
                INSERT INTO Users (UserID, UserName, Email, CreatedAt)
                VALUES (?, ?, ?, ?)
            """, (
                user_details['UserID'],
                user_details['UserName'],
                user_details['Email'],
                datetime.now().isoformat(),
            ))

            conn.commit()

        except Exception as e:
            raise DatabaseErrors(f"Error saving user details to the database: {str(e)}")

        finally:
            conn.close()

    def createUserObject(self, user_details):
        # Implementation to create a user object with the given details.
        return {
            'UserID': user_details['UserID'],
            'UserName': user_details['UserName'],
            'Email': user_details['Email'],
            'CreatedAt': datetime.now().isoformat(),
        }
```

```
    def gatherUpdatedUserDetails(self):
        # Implementation to gather updated user details from the user interface.
        updated_user_details = {}
        updated_user_details['UserName'] = input("Enter updated UserName: ")
        updated_user_details['Email'] = input("Enter updated Email: ")
        return updated_user_details

    def validateUpdatedUserDetails(self, updated_user_details):
        # Implementation to validate updated user details.
        # For simplicity, let's assume both UserName and Email are required.
        if not updated_user_details.get('UserName') or not updated_user_details.get('Email'):
            raise ValidationErrors("Updated UserName and Email are required.")

    def updateUserDetailsInDatabase(self, user, updated_user_details):
        # Implementation to update user details in the database.
        try:
            conn = sqlite3.connect("user_database.db")
            cursor = conn.cursor()

            cursor.execute("""
                UPDATE Users
                SET UserName=?, Email=?
                WHERE UserID=?
            """, (
                updated_user_details['UserName'],
                updated_user_details['Email'],
                user['UserID'],
            ))

            conn.commit()


           except Exception as e:
               raise DatabaseErrors(f"Error updating user details in the database: {str(e)}")

           finally:
               conn.close()

   # Custom exception classes
   class ValidationErrors(Exception):
       pass
```

2. Administrator Class Routines:

```
import sqlite3
from datetime import datetime

class Administrator:

    def configureSystem(self):
        admin = self.authenticateAdministrator()
        system_config_details = self.gatherSystemConfigDetails()
        self.validateSystemConfigDetails(system_config_details)
        try:
            # Step 4: Update system configuration
            self.updateSystemConfig(admin, system_config_details)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
```

```
    def gatherSystemConfigDetails(self):
        # Implementation to gather system configuration details.
        system_config_details = {}
        system_config_details['Setting1'] = input("Enter Setting1: ")
        system_config_details['Setting2'] = input("Enter Setting2: ")
        return system_config_details

    def validateSystemConfigDetails(self, system_config_details):
        # Implementation to validate system configuration details.
        # For simplicity, let's assume both Setting1 and Setting2 are required.
        if not system_config_details.get('Setting1') or not system_config_details.get('Setting2'):
            raise ValidationErrors("Setting1 and Setting2 are required.")
```

```
def updateSystemConfig(self, admin, system_config_details):
    # Implementation to update system configuration.
    try:
        conn = sqlite3.connect("system_config_database.db")
        cursor = conn.cursor()

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS SystemConfig (
                AdminID TEXT PRIMARY KEY,
                Setting1 TEXT,
                Setting2 TEXT,
                UpdatedAt TEXT
            )
        """)

        cursor.execute("""
            INSERT INTO SystemConfig (AdminID, Setting1, Setting2, UpdatedAt)
            VALUES (?, ?, ?, ?)
            ON CONFLICT(AdminID) DO UPDATE SET
                Setting1=excluded.Setting1,
                Setting2=excluded.Setting2,
                UpdatedAt=excluded.UpdatedAt
        """, (
            admin['AdminID'],
            system_config_details['Setting1'],
            system_config_details['Setting2'],
            datetime.now().isoformat(),
        ))

        conn.commit()

    except Exception as e:
        raise DatabaseErrors(f"Error updating system configuration: {str(e)}")

    finally:
        conn.close()
```

```
def manageUserRoles(self):
    # Step 1: Authenticate the administrator
    admin = self.authenticateAdministrator()

    # Step 2: Select a user
    selected_user = self.selectUser()

    try:
        # Step 3: Manage roles for the selected user
        self.manageRoles(selected_user)
    except RoleManagementErrors as e:
        # Handle role management errors
        raise e

def selectUser(self):
    # Implementation to select a user.
    # For simplicity, let's assume we have a list of users and the admin selects from it.
    users = [{'UserID': '1', 'UserName': 'User1'}, {'UserID': '2', 'UserName': 'User2'}]
    print("Select a user:")
    for idx, user in enumerate(users, start=1):
        print(f"{idx}. {user['UserName']} ({user['UserID']})")

    selected_index = int(input("Enter the number of the user: ")) - 1
    return users[selected_index]

def manageRoles(self, user):
    # Implementation to manage roles for the selected user.
    # For simplicity, let's print the user's current roles and allow the admin to modify them.
    print(f"Current roles for {user['UserName']} ({user['UserID']}): Role1, Role2")
    new_roles = input("Enter new roles (comma-separated): ").split(', ')

    # Here, you might update the user's roles in the database.
```

```
def authenticateAdministrator(self):
    # Step 1: Gather administrator credentials
    credentials = self.gatherAdminCredentials()

    # Step 2: Validate administrator credentials
    self.validateAdminCredentials(credentials)

    # Step 3: Authenticate the administrator
    authenticated_admin = self.authenticateAdmin(credentials)

    return authenticated_admin

def gatherAdminCredentials(self):
    # Implementation to gather administrator credentials.
    admin_credentials = {}
    admin_credentials['AdminID'] = input("Enter AdminID: ")
    admin_credentials['Password'] = input("Enter Password: ")
    return admin_credentials

def validateAdminCredentials(self, credentials):
    # Implementation to validate administrator credentials.
    # For simplicity, let's assume both AdminID and Password are required.
    if not credentials.get('AdminID') or not credentials.get('Password'):
        raise ValidationErrors("AdminID and Password are required.")
```

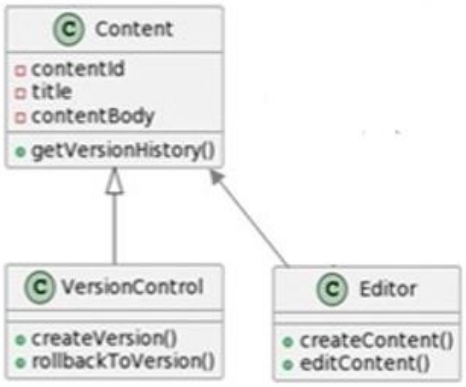
```
    def authenticateAdmin(self, credentials):
        # Implementation to authenticate the administrator.
        # For simplicity, let's assume we have a predefined admin in the system.
        if credentials['AdminID'] == 'admin' and credentials['Password'] == 'adminpassword':
            return {'AdminID': 'admin', 'AuthenticatedAt': datetime.now().isoformat()}
        else:
            raise AuthenticationErrors("Invalid administrator credentials.")


# Custom exception classes
class ValidationErrors(Exception):
    pass

class AuthenticationErrors(Exception):
    pass

class DatabaseErrors(Exception):
    pass

class RoleManagementErrors(Exception):
    pass
```

3. AuthenticationManager Class Routines:

```
import hashlib  # For hashing the password

class AuthenticationManager:
    def authenticate(self, user):
        credentials = self.gatherUserCredentials()
        self.validateUserCredentials(credentials)
        authenticated_user = self.authenticateUser(credentials, user)
        return authenticated_user

    def gatherUserCredentials(self):
        # Implementation to gather user credentials from the user interface.
        credentials = {}
        credentials['Username'] = input("Enter username: ")
        credentials['Password'] = input("Enter password: ")
        return credentials

    def validateUserCredentials(self, credentials):
        # Step 1: Check if the username and password meet validation criteria
        if not self.isValidUsername(credentials['Username']) or not self.isValidPassword(credentials['Password']):
            raise ValidationErrors("Invalid username or password.")

    def authenticateUser(self, credentials, user):
        # Step 1: Check if the provided username and password match the user's credentials
        if credentials['Username'] == user['UserName'] and self.verifyPassword(credentials['Password'], user['Passwo
            return user
        else:
            raise AuthenticationErrors("Invalid username or password.")

    def isValidUsername(self, username):
        # Implementation to check if the username is valid.
        # In this example, any non-empty string is considered a valid username.
        return bool(username)

    def isValidPassword(self, password):
        # Implementation to check if the password is valid.
        # In this example, any non-empty string is considered a valid password.
        return bool(password)
```

```
    def verifyPassword(self, input_password, stored_password):
        # Implementation to securely verify the password.
        # In a real-world scenario, use a secure password hashing library (e.g., bcrypt).

        # Hash the input password using the same hashing algorithm used for storing passwords
        hashed_input_password = self.hashPassword(input_password)

        # Compare the hashed input password with the stored password
        return hashed_input_password == stored_password

    def hashPassword(self, password):
        # Hash the password using a secure hashing algorithm (e.g., SHA-256)
        # In a real-world scenario, use a secure password hashing library (e.g., bcrypt).
        hashed_password = hashlib.sha256(password.encode()).hexdigest()
        return hashed_password

# Custom exception classes
class ValidationErrors(Exception):
    pass

class AuthenticationErrors(Exception):
    pass
```

(2) Content Creation and Editing Subsystem Class Diagram



Implementation:

1. Content Class Routines:

```
class Content:
    def createContent(self, creator, title):
        self.checkContentCreationPermission(creator)
        content_object = self.createNewContentObject(creator, title)
        self.applyVersionControl(content_object)
        try:
            # Step 4: Save content details in the database
            self.saveContentDetailsToDatabase(content_object)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        # Step 5: Return the created content object
        return content_object

    def checkContentCreationPermission(self, user):
        # Implementation to check if the user has permission to create content.
        # Placeholder example: Assume all users have permission.
        if not user:
            raise PermissionErrors("User does not have permission to create content.")

    def createNewContentObject(self, creator, title):
        # Implementation to create a new content object.
        # Placeholder example: Store basic details in a dictionary.
        content_object = {
            'Creator': creator,
            'Title': title,
            'ContentBody': '',  # You may have additional properties
        }
        return content_object

    def applyVersionControl(self, content_object):
        # Implementation to apply version control to the content.
        version_control = VersionControl()
        version_control.applyVersionControl(content_object)

    def saveContentDetailsToDatabase(self, content_object):
        # Implementation to save content details in the database.
        # Placeholder example: Print the content details.
        print(f"Saved content to the database: {content_object}")
```

```
def editContent(self, editor, new_title):
    # Step 1: Check user permissions for content editing
    self.checkContentEditingPermission(editor)

    # Step 2: Edit the content with the new title
    edited_content = self.editContentDetails(editor, new_title)

    # Step 3: Notify the author of the changes
    try:
        self.notifyAuthor(edited_content)
    except NotificationErrors as e:
        # Handle notification errors
        raise e
```

```
        try:
            # Step 4: Save the edited content details in the database
            self.saveEditedContentDetailsToDatabase(edited_content)
        except DatabaseErrors as e:
            # Handle database errors
            raise e

        # Step 5: Return the edited content object
        return edited_content

    def checkContentEditingPermission(self, user):
        # Implementation to check if the user has permission to edit content.
        # Placeholder example: Assume all users have permission.
        if not user:
            raise PermissionErrors("User does not have permission to edit content.")

    def editContentDetails(self, editor, new_title):
        # Implementation to edit the content details.
        # Placeholder example: Update the title of the content.
        content_object = {
            'Editor': editor,
            'Title': new_title,
            'ContentBody': '',  # You may have additional properties
        }
        return content_object
```

```
    def notifyAuthor(self, edited_content):
        # Implementation to notify the author of content changes.
        # Placeholder example: Print a notification message.
        print(f"Notifying author about content changes: {edited_content['Title']}")

    def saveEditedContentDetailsToDatabase(self, edited_content):
        # Implementation to save edited content details in the database.
        # Placeholder example: Print the edited content details.
        print(f"Saved edited content to the database: {edited_content}")

class VersionControl:
    def applyVersionControl(self, content_object):
        # Step 1: Track changes to the content
        self.trackChanges(content_object)

        try:
            # Step 2: Save the versioned content
            self.saveVersion(content_object)
        except VersionControlErrors as e:
            # Handle version control errors
            raise e

    def trackChanges(self, content_object):
        # Implementation to track changes to the content.
        # Placeholder example: Print a message about tracking changes.
        print(f"Tracking changes for content: {content_object['Title']}")
```

```
    def saveVersion(self, content_object):
        # Implementation to save the versioned content.
        # Placeholder example: Print a message about saving the version.
        print(f"Saved version for content: {content_object['Title']}")
```

```
# Custom exception classes
class PermissionErrors(Exception):
    pass

class NotificationErrors(Exception):
    pass

class DatabaseErrors(Exception):
    pass

class VersionControlErrors(Exception):
    pass
```

2. Editor Class Routines:

```
from copy import deepcopy

class Editor:
    def __init__(self, database):
        self.database = database
    def editContent(self, content, new_title):
        self.checkContentEditingPermission()
        edited_content = self.editContentDetails(content, new_title)
        try:
            self.notifyAuthor(content, edited_content)
        except NotificationErrors as e:
            # Handle notification errors
            raise e
        try:
            # Step 4: Save the edited content details in the database
            self.saveEditedContentDetailsToDatabase(edited_content)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        # Step 5: Return the edited content object
        return edited_content

    def checkContentEditingPermission(self):
        user_has_permission = True
        if not user_has_permission:
            raise PermissionErrors("User does not have permission to edit content.")

    def editContentDetails(self, content, new_title):
        # Use deepcopy to create a new instance, ensuring the original content remains unchanged
        edited_content = deepcopy(content)
        edited_content.title = new_title
        return edited_content
```

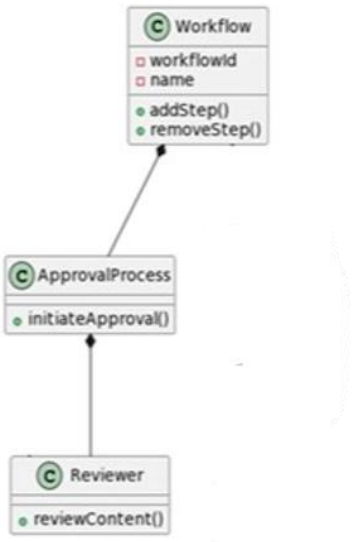
```
    def notifyAuthor(self, old_content, edited_content):
         # Step 1: Check if the author has opted-in for notifications
        if self.isAuthorSubscribed(old_content.creator):
            # Step 2: Send a notification to the author about the content changes
            try:
                self.sendNotification(old_content.creator, edited_content)
            except NotificationErrors as e:
                # Handle notification errors
                raise e

    def sendNotification(self, author, edited_content):
        print(f"Notifying author {author.username} about content changes: {edited_content.title}")

    def saveEditedContentDetailsToDatabase(self, edited_content):
        # In a real-world scenario, you would interact with the database to save the edited content.
        # For simplicity, we print the details here.
        print(f"Saved edited content to the database: {edited_content}")

    def reviewContent(self, content):
        # Step 1: Check user permissions for content review
        self.checkContentReviewPermission()
        # Step 2: Perform the content review
        review_outcome = self.performContentReview(content)
        # Step 3: Notify the author of the review outcome
        try:
            self.notifyAuthor(content, review_outcome)
        except NotificationErrors as e:
            # Handle notification errors
            raise e
    def checkContentReviewPermission(self):
        # Example: Assuming all editors have permission to review
        user_has_permission = True
        if not user_has_permission:
            raise PermissionErrors("User does not have permission to review content.")


    def performContentReview(self, content):
        # Example: Assuming a positive review outcome
        return "Positive"

# Custom exception classes remain the same.
class PermissionErrors(Exception):
    pass

class NotificationErrors(Exception):
    pass

class DatabaseErrors(Exception):
    pass
```

3. VersionControl Class Routines:

```
import hashlib
import datetime

class VersionControl:
    def trackChanges(self, content_object):
        # Step 1: Determine the changes made to the content
        changes = self.determineContentChanges(content_object)
```

```
    try:
        # Step 2: Log the changes for version control
        self.logChanges(content_object, changes)
    except VersionControlErrors as e:
        # Handle version control errors
        raise e
def determineContentChanges(self, content_object):
    content_hash = hashlib.sha256(content_object.encode()).hexdigest()
    return {"content_hash": content_hash}
def logChanges(self, content_object, changes):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_entry = f"{timestamp} - Changes for {content_object}: {changes}"
    print(log_entry)
def saveVersion(self, content_object):
    # Step 1: Create a new version for the content
    new_version = self.createVersion(content_object)
    try:
        # Step 2: Save the versioned content in the database
        self.saveVersionToDatabase(new_version)
    except DatabaseErrors as e:
        # Handle database errors
        raise e
    return new_version
def createVersion(self, content_object):
    version_number = random.randint(1, 1000)
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    return {"content": content_object, "version_number": version_number, "timestamp": timestamp}

def saveVersionToDatabase(self, version):
    print(f"Saving version to database: {version}")
```

(3) Workflow and Approval Subsystem Class Diagram



Implementation:

1. Workflow Class Routines:

```
import datetime
import random

class Workflow:
    def initiateWorkflow(self, content):
        # Step 1: Check if a workflow is already in progress for the content
        existing_workflow = self.checkExistingWorkflow(content)
        if existing_workflow:
            raise WorkflowErrors("Workflow already in progress for the content.")
        # Step 2: Set the initial status and steps for the workflow
        workflow_object = self.setInitialWorkflowDetails(content)
        # Step 3: Notify reviewers about the new workflow
        try:
            self.notifyReviewers(content, workflow_object)
        except NotificationErrors as e:
            # Handle notification errors
            raise e
        try:
            # Step 4: Save workflow details in the database
            self.saveWorkflowDetailsToDatabase(workflow_object)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        # Step 5: Return the initiated workflow object
        return workflow_object
    def checkExistingWorkflow(self, content):
        # Placeholder implementation, replace with actual logic to check if a workflow is in progress
        return None
    def setInitialWorkflowDetails(self, content):
        # Placeholder implementation, replace with actual logic to set initial details
        initial_workflow_object = {
            "content": content,
            "status": "In Progress",
            "steps": ["Review", "Approval"],
            "reviewers": ["Reviewer1", "Reviewer2"],
            "timestamp": datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        }
        return initial_workflow_object

    def notifyReviewers(self, content, workflow_object):
        # Step 1: Check if there are reviewers assigned to the workflow
        if workflow_object.get("reviewers"):
            # Step 2: Send notifications to the assigned reviewers
            try:
                self.sendNotificationsToReviewers(content, workflow_object)
            except NotificationErrors as e:
                # Handle notification errors
                raise e
    def sendNotificationsToReviewers(self, content, workflow_object):
        # Placeholder implementation, replace with actual logic to send notifications
        print(f"Notifying reviewers about the new workflow for {content}")

    def saveWorkflowDetailsToDatabase(self, workflow_object):
        # Placeholder implementation, replace with actual logic to save workflow details to the database
        print(f"Saving workflow details to the database: {workflow_object}")

    def progressWorkflow(self, workflow_object):
        # Step 1: Check if the workflow is in progress
        if workflow_object.get("status") == 'In Progress':
            # Step 2: Move to the next step in the workflow
            self.moveToNextWorkflowStep(workflow_object)
```

```
            # Step 3: Update the workflow status
            self.updateWorkflowStatus(workflow_object)

            # Step 4: Notify reviewers about the progress
            try:
                self.notifyReviewers(workflow_object.get("content"), workflow_object)
            except NotificationErrors as e:
                # Handle notification errors
                raise e


    def moveToNextWorkflowStep(self, workflow_object):
        # Placeholder implementation, replace with actual logic to move to the next step
        current_steps = workflow_object.get("steps")
        if current_steps:
            next_step = "Final Approval" if current_steps[-1] == "Approval" else "Approval"
            current_steps.append(next_step)
            workflow_object["steps"] = current_steps

    def updateWorkflowStatus(self, workflow_object):
        # Placeholder implementation, replace with actual logic to update workflow status
        workflow_object["status"] = "In Progress" if "Approval" in workflow_object.get("steps") else "Completed"


class WorkflowErrors(Exception):
    pass

class NotificationErrors(Exception):
    pass

class DatabaseErrors(Exception):
    pass
```

2. ApprovalProcess Class Routines:

```
import datetime

class ApprovalProcess:
    def approveContent(self, content):
        # Step 1: Check if the content is in the approval process
        self.checkContentInApprovalProcess(content)
        # Step 2: Approve the content
        self.performContentApproval(content)
        # Step 3: Update the approval status
        self.updateApprovalStatus(content)
        # Step 4: Notify the author of the approval
        try:
            self.notifyAuthor(content, "approval")
        except NotificationErrors as e:
            # Handle notification errors
            raise e
    def checkContentInApprovalProcess(self, content):
        # Placeholder implementation, replace with actual logic to check if the content is in the approval process
        if content.approval_status != "Pending":
            raise ApprovalProcessErrors("Content is not in the approval process.")
    def performContentApproval(self, content):
        # Placeholder implementation, replace with actual logic to perform content approval
        content.approval_status = "Approved"
    def updateApprovalStatus(self, content):
        # Placeholder implementation, replace with actual logic to update the approval status
        content.approval_timestamp = datetime.datetime.now()
    def rejectContent(self, content):
        # Step 1: Check if the content is in the approval process
        self.checkContentInApprovalProcess(content)
        # Step 2: Reject the content
        self.performContentRejection(content)
        # Step 3: Update the rejection status
        self.updateRejectionStatus(content)
        # Step 4: Notify the author of the rejection
```

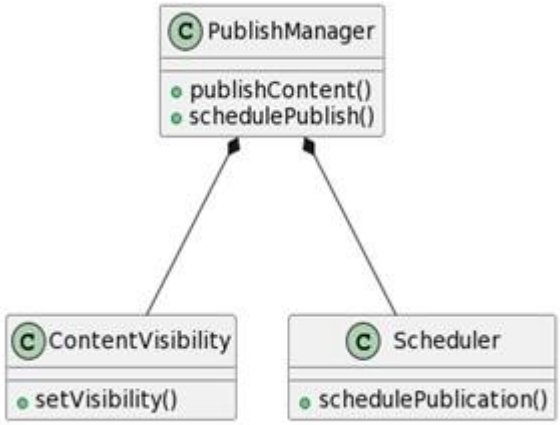
```
        # Step 4: Notify the author of the rejection
        try:
            self.notifyAuthor(content, "rejection")
        except NotificationErrors as e:
            # Handle notification errors
            raise e

    def performContentRejection(self, content):
        content.approval_status = "Rejected"
    def updateRejectionStatus(self, content):
        content.approval_timestamp = datetime.datetime.now()
    def notifyAuthor(self, content, action):
        # Step 1: Check if the author has opted-in for notifications
        if self.isAuthorSubscribed(content.creator):
            # Step 2: Send a notification to the author about the approval or rejection
            try:
                self.sendNotification(content.creator, content, action)
            except NotificationErrors as e:
                # Handle notification errors
                raise e
    def isAuthorSubscribed(self, author):
        # Placeholder implementation, replace with actual logic to check if the author has opted-in for notifications
        return True

    def sendNotification(self, author, content, action):
        # Placeholder implementation, replace with actual logic to send a notification
        print(f"Notification sent to {author} about the {action} of the content.")
```

3. Reviewer Class Routines:

```
import random

class Reviewer:
    def reviewContent(self, content):
        # Step 1: Check if the reviewer has permission to review content
        self.checkReviewPermission(content.creator)
        # Step 2: Review the content
        review_outcome = self.performContentReview(content)
        # Step 3: Provide feedback or comments
        self.provideReviewFeedback(content, review_outcome)
        # Step 4: Notify the outcome of the review
        try:
            self.notifyOutcome(content, review_outcome)
        except NotificationErrors as e:
            # Handle notification errors
            raise e
    def checkReviewPermission(self, author):
        # Placeholder implementation, replace with actual logic to check if the reviewer has permission to review content
        if not self.isAuthorSubscribed(author):
            raise PermissionErrors("Reviewer doesn't have permission to review content.")
    def performContentReview(self, content):
        # Placeholder implementation, replace with actual logic to perform the content review and return the review outcome
        # For simplicity, let's assume the review outcome is either "Approved" or "Rejected" based on random choice.
        return "Approved" if random.choice([True, False]) else "Rejected"
    def provideReviewFeedback(self, content, review_outcome):
        # Placeholder implementation, replace with actual logic to provide feedback or comments based on the review outcome
        feedback_message = f"The content is {review_outcome.lower()} based on the review."
        print(feedback_message)
    def notifyOutcome(self, content, review_outcome):
        # Step 1: Check if there are subscribers interested in review outcomes
        if self.hasSubscribersForReviewOutcomes(content.creator):
            # Step 2: Send notifications to the subscribers
            try:
                self.sendReviewOutcomeNotifications(content.creator, content, review_outcome)
            except NotificationErrors as e:
                # Handle notification errors
                raise e
```

```
    def sendReviewOutcomeNotifications(self, author, content, review_outcome):
        # Placeholder implementation, replace with actual logic to send notifications to the subscribers about the review
        notification_message = f"Content review outcome for {content.title}: {review_outcome}"
        print(f"Notification sent to {author} - {notification_message}")
class PermissionErrors(Exception):
    pass

class ReviewErrors(Exception):
    pass

class NotificationErrors(Exception):
    pass
```

(4) Publishing Subsystem Class Diagram



Implementation:

1. Publish Manager Class Routines:

```
import datetime
import random

class PublishManager:
    def publishContent(self, content):
        # Step 1: Check if the user has permission to publish content
        self.checkContentPublicationPermission(content.creator)
        # Step 2: Publish the content
        published_content = self.performContentPublication(content)
        # Step 3: Notify subscribers about the new publication
        try:
            self.notifySubscribers(content, published_content)
        except NotificationErrors as e:
            # Handle notification errors
            raise e
```

```
        try:
            # Step 4: Save publication details in the database
            self.savePublicationDetailsToDatabase(published_content)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        # Step 5: Return the published content object
        return published_content
    def checkContentPublicationPermission(self, user):
        # Check if the user has permission to publish content.
        if not self.isUserAllowedToPublish(user):
            raise PermissionErrors("User doesn't have permission to publish content.")
    def performContentPublication(self, content):
        # Perform the content publication and return the published content.
        # For simplicity, let's assume the content is now published with a timestamp.
        content.published_at = datetime.datetime.now()
        return content
    def notifySubscribers(self, content, published_content):
        # Step 1: Check if there are subscribers interested in new publications
        if self.hasSubscribersForNewPublications():
            # Step 2: Send notifications to the subscribers
            try:
                self.sendPublicationNotifications(content, published_content)
            except NotificationErrors as e:

    def sendPublicationNotifications(self, content, published_content):
        # Send notifications to the subscribers about the new publication.
        # Placeholder implementation, replace with actual logic to send notifications.
        notification_message = f"New publication: {published_content.title} is now available."
        print(notification_message)
    def savePublicationDetailsToDatabase(self, published_content):.
        print(f"Saving publication details to the database: {published_content.title}")
    def schedulePublication(self, content, publish_date):
        # Step 1: Check if the user has permission to schedule publication
        self.checkPublicationSchedulingPermission(content.creator)
        # Step 2: Schedule the publication for the specified date
        scheduled_publication = self.scheduleContentPublication(content, publish_date)
        try:
            # Step 3: Save the publication schedule details in the database
            self.savePublicationScheduleToDatabase(scheduled_publication)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        return scheduled_publication
    def checkPublicationSchedulingPermission(self, user):
        # Check if the user has permission to schedule publication.
        # Placeholder implementation, replace with actual logic.
        if not self.isUserAllowedToSchedule(user):
            raise PermissionErrors("User doesn't have permission to schedule publication.")
    def scheduleContentPublication(self, content, publish_date):
        content.scheduled_publish_date = publish_date
        return content
    def savePublicationScheduleToDatabase(self, scheduled_publication):
        # Save the publication schedule details in the database.
        # Placeholder implementation, replace with actual logic.
        print(f"Saving publication schedule details to the database: {scheduled_publication.title}")
class PermissionErrors(Exception):
    pass
class NotificationErrors(Exception):
    pass
class DatabaseErrors(Exception):
    pass
```

2. ContentVisibility Class Routines:s

```
class ContentVisibility:
    def setVisibility(self, content, user):
        # Step 1: Check if the user has permission to set content visibility
        self.checkContentVisibilityPermission(user)
        # Step 2: Set the visibility of content for the user
        visibility_settings = self.setVisibilityForUser(content, user)
        try:
            # Step 3: Save the visibility settings in the database
            self.saveVisibilitySettingsToDatabase(visibility_settings)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
    def checkContentVisibilityPermission(self, user):
        # Check if the user has permission to set content visibility.
        # Placeholder implementation, replace with actual logic.
        if not self.isUserAllowedToSetVisibility(user):
            raise PermissionErrors("User doesn't have permission to set content visibility.")
    def setVisibilityForUser(self, content, user):
        # Set the visibility of content for the user and return visibility settings.
        # Placeholder implementation, replace with actual logic.
        visibility_settings = {
            'content_id': content.id,
            'user_id': user.id,
            'visibility': 'visible'  # Placeholder value, replace with actual visibility setting
        }
        return visibility_settings
    def saveVisibilitySettingsToDatabase(self, visibility_settings):
        # Save the visibility settings in the database.
        # Placeholder implementation, replace with actual logic.
        print(f"Saving visibility settings to the database: {visibility_settings}")
    def applyVisibilitySettings(self):
        # Step 1: Retrieve visibility settings for all content
        visibility_settings = self.retrieveVisibilitySettings()
        # Step 2: Apply the visibility settings
        self.performVisibilitySettingsApplication(visibility_settings)

    def retrieveVisibilitySettings(self):
        # Retrieve visibility settings for all content.
        # Placeholder implementation, replace with actual logic.
        return [
            {'content_id': 1, 'user_id': 1, 'visibility': 'visible'},
            {'content_id': 2, 'user_id': 1, 'visibility': 'hidden'},
            # ... add more visibility settings
        ]
    def performVisibilitySettingsApplication(self, visibility_settings):
        # Apply the visibility settings for all content.
        # Placeholder implementation, replace with actual logic.
        for settings in visibility_settings:
            content_id = settings['content_id']
            user_id = settings['user_id']
            visibility = settings['visibility']
            print(f"Applying visibility '{visibility}' for user {user_id} on content {content_id}")
class PermissionErrors(Exception):
    pass

class DatabaseErrors(Exception):
    pass
```

3. Scheduler Class Routines:

```
class Scheduler:
    def scheduleTask(self, date, task):
        # Step 1: Check if the user has permission to schedule tasks
        self.checkTaskSchedulingPermission()
        # Step 2: Schedule the task for the specified date
        scheduled_task = self.scheduleTaskForDate(date, task)
        try:
            # Step 3: Save the task schedule details in the database
            self.saveTaskScheduleToDatabase(scheduled_task)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        return scheduled_task
    def scheduleTaskForDate(self, date, task):
        # Schedule the task for the specified date and return the scheduled task.
        # Placeholder implementation, replace with actual logic.
        return {'task_id': task.id, 'scheduled_date': date, 'status': 'Scheduled'}
    def saveTaskScheduleToDatabase(self, scheduled_task):
        # Save the task schedule details in the database.
        # Placeholder implementation, replace with actual logic.
        print(f"Saving task schedule details to the database: {scheduled_task}")
    def executeTask(self, task):
        # Step 1: Check if the user has permission to execute tasks
        self.checkTaskExecutionPermission()
        # Step 2: Execute the task
        executed_task = self.performTaskExecution(task)
        # Step 3: Update the task status
        self.updateTaskStatus(executed_task)
        try:
            # Step 4: Save the task execution details in the database
            self.saveTaskExecutionDetailsToDatabase(executed_task)
        except DatabaseErrors as e:
            # Handle database errors
            raise e
        return executed_task
    def performTaskExecution(self, task):
        # Perform the execution of the task and return the executed task.
        # Placeholder implementation, replace with actual logic.
        return {'task_id': task.id, 'status': 'Executed', 'execution_date': datetime.now()}
    def updateTaskStatus(self, executed_task):
        # Update the task status after execution.
        # Placeholder implementation, replace with actual logic.
        executed_task['status'] = 'Completed'
    def saveTaskExecutionDetailsToDatabase(self, executed_task):
        # Save the task execution details in the database.
        # Placeholder implementation, replace with actual logic.
        print(f"Saving task execution details to the database: {executed_task}")
class PermissionErrors(Exception):
    pass
class TaskExecutionErrors(Exception):
    pass
class DatabaseErrors(Exception):
    pass
```

THE END