ISYS3413

Assignment 3: Team-Based

Kasup Wellage | s4074242

Kurt Clado | s4003781

Timothy Nancarrow | s3950562

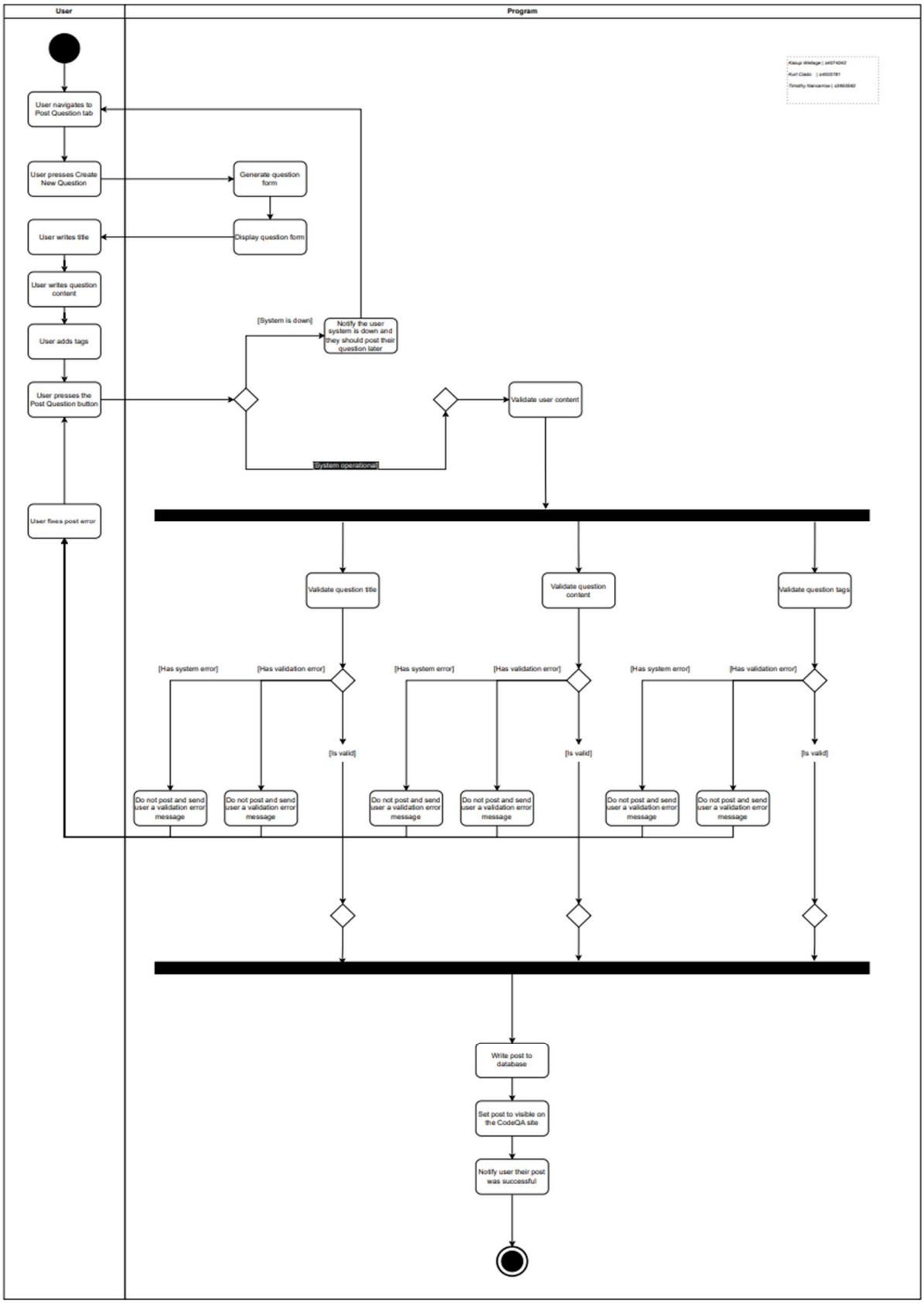

Figure 2: Activity Diagram of Post Question, generated from Draw.io

1.1. Activity Diagram for View Statistics

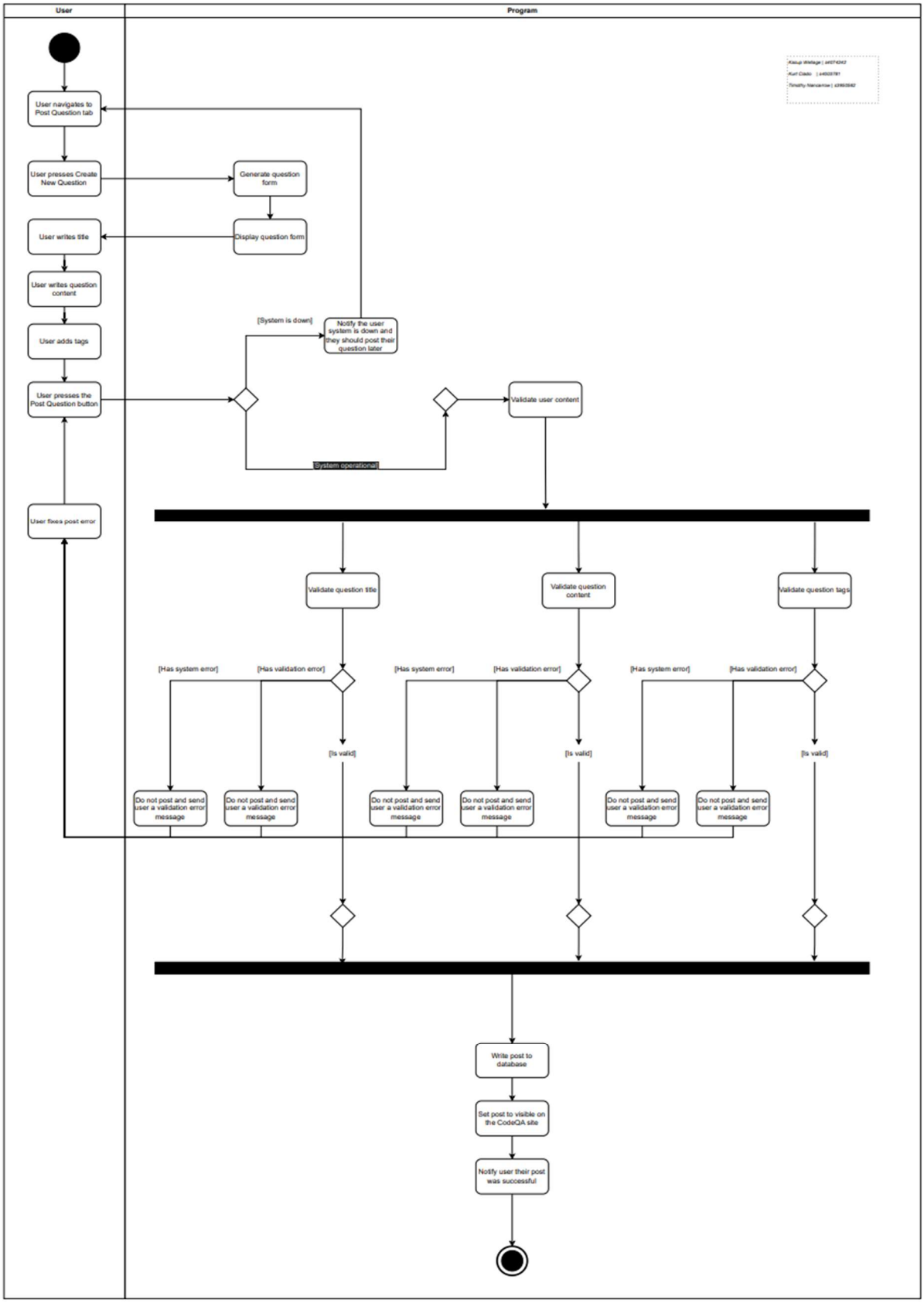

Figure 2: Activity Diagram of Post Question, generated from Draw.io

2. Use Case Descriptions

2.1. Use Case of Register/Sign Up

[Use Case] Register/Sign Up	
[Use Case ID]	CQA-001
[Brief Description]	Aims to allows guest users to create an account on **CodeQA**, enabling them to participate in functions such as asking questions, answering, voting, and commenting.
[Primary Actors]	• Guest
[Secondary Actors]	• System
[Preconditions]	• User mustn't already be registered with CodeQA. • System is operational and accessible for the user.
[Main Flow]	1. The user navigates to the **Sign-Up** page on CodeQA. 2. The user chooses a unique username. 3. The user enters a *valid* email address. 4. The user creates a password that meets security requirements. 5. The user completes additional requirements such as agreeing to terms of service. 6. The user submits the registration form. 7. The system validates the provided information. 8. The system starts a profile for the new user and writes the registration details. 9. The system sends verification email to the user by the provided email address. 10. The user verifies their email address by clicking on the verification link within the sent the email. 11. Upon successful email verification the system logs the user in automatically.
[Postconditions]	• The user completes process and is registered, verified, and logged into the system with a user profile.
[Alternative Flows]	1. If the email address is already assigned with another user or account, the system prompts user requesting for a different email. 2. If the username has already been created by another, the system prompts the user to enter a different username. 3. If the user fails to verify via email within a timeframe, the system suspense the account.

2.2. Use Case for Ask Questions

[Use Case] Post Question	
[Use Case ID]	CQA-002
[Brief Description]	Allows a verified user to post questions to CodeQA, allowing users to participate with the main function of the platform.
[Primary Actors]	• Registered User
[Secondary Actors]	• System
[Preconditions]	• User must be logged into their account. • System must be operational and accessible.
[Main Flow]	1. The user navigates to the **Post Question** section on CodeQA. 2. The user clicks on the **Create New Question** button. 3. The user is presented with a form to enter the question title and content. 4. The user enters the question title and the content. 5. The user selects tags from a list to categorise the question. 6. The user reviews the question. 7. The user submits the question by clicking **Post Question** button. 8. The system validates the question and ensures it adherence to guidelines. 9. The system writes the question to a database and makes it visible on CodeQA. 10. The system notifies the user that the question has been posted.
[Postconditions]	• The question is posted and viewable publicly. • The user receives confirmation that the posting was successful.
[Alternative Flows]	1. If the form is submitted incomplete, the system displays an error message, prompting the user to complete in the missing fields. 2. If the tags selected do not meet the standards on T&Cs, the system asks the user to select other tags. 3. If the system is not operational when the user tries to submit the question, the user receives a prompt asking them to try later.

2.3. Use Case of Register/Sign Up

[Use Case] Delete Posts	
[Use Case ID]	CQA-003
[Brief Description]	Allows moderators to remove posts that are inappropriate, irrelevant, or violate the T&Cs, ensuring the quality and safety.
[Primary Actors]	• Moderator
[Secondary Actors]	• System
[Preconditions]	• Moderator must be logged into account. • Moderator has identified a post that violates the guidelines.
[Main Flow]	1. The moderator navigates to the post in question within the platform interface. 2. The moderator reviews the post and its compliance with the T&Cs. 3. The moderator clicks the **Delete** button linked with the post. 4. The system prompts the moderator to confirm the deletion to avoid unintended deletions. 5. The moderator confirms the deletion. 6. The system removes the post from public view. 7. The system logs the deletion action; post details and the moderator for record keeping. 8. The system sends a notification to the user to whom created the deleted post, advising of the deletion and any reason. 9. The moderator is redirected to the previous page and receives confirmation that the post has been successfully deleted.
[Postconditions]	• The post is no longer visible or accessible on the platform. • The original poster is informed of the deletion and the reason (if available).
[Alternative Flows]	1. If the moderator accidentally clicks the **Delete** button, they can cancel the action during the confirmation step. 2. If due to a system fault the deletion fails, the moderator is prompted with an error message to retry the deletion or report the issue for further support. 3. If a moderator without sufficient privileges attempts to delete a post, the system denies the action and logs the attempt for security monitoring.

2.4. Use Case for View Statistics

[Use Case] View Statistics	
[Use Case ID]	CQA-004
[Brief Description]	Allows users to access or view statistical information within **CodeQA**, including their content reach and overall site metrics. Different levels of detail are available based on user roles, i.e., administrators accessing the most thorough data.
[Primary Actors]	• Registered User
[Secondary Actors]	• Administrator • System
[Preconditions]	• User must be logged into their account. • System is operational and data processed is up to date.
[Main Flow]	1. The user navigates to the **Statistics** section by using CodeQAs menu. 2. The user selects the type of statistics to view, content performance or engagement metrics. 3. The system verifies the user's role to decide the level of detail and range available to requesting user. 4. System fetches statistical data from the database. 5. The system displays the statistics in a user-friendly format, such as graphs or summary tables. 6. The user reviews the displayed statistics, which may include total views, likes, comments, most active times, and other relevant metrics. 7. The user may choose to filter or adjust the data range to refine the displayed statistics. 8. If the user is an administrator, additional options for downloading or exporting the data to be available. 9. The user exits the statistics view when finished.
[Postconditions]	• The user gains insights from the viewed statistics, potentially influencing future interactions or content creation on the platform.
[Alternative Flows]	• If a user attempts to access statistics not permitted for their role, the system displays an error message and does not show the restricted data. • If the requested statistics are temporarily unavailable due to system maintenance or updates, the system informs the user of the issue and may suggest when to try again. • If a user requests statistics outside the usual reporting range, the system may need additional time to compile these, or it might instruct the user on how to make a special request.

3. Class Diagram

3.1. Skeleton Code for Sign Up

```
4.  // Class that shows the process for a user to sign up on CodeQA
5.  public class UserSignUp {
6.      public static void main(String[] args) {
7.          UserSignUp signupProcess = new UserSignUp();
8.          signupProcess.startSignUp();
9.      }
10.
11.     // sign-up process by creating a user object.
12.     public void startSignUp() {
13.         User user = new User();
14.
15.         // [1] user submits username
16.         user.setUsername(submitUsername());
17.
18.         // [2] check if username is unique
19.         while (!isUsernameUnique(user.getUsername())) {
20.             user.setUsername(promptUniqueUsername());
21.         }
22.
23.         // [3] user submits email
24.         user.setEmail(submitEmail());
25.
26.         // [4] check if email is valid
27.         while (!isEmailValid(user.getEmail())) {
28.             user.setEmail(promptValidEmail());
29.         }
30.
31.         // [5] user submits password
32.         user.setPassword(submitPassword());
33.
34.         // [6] check if password is secure
35.         while (!isSecure(user.getPassword())) {
36.             user.setPassword(promptForSecurePassword());
37.         }
38.
39.         // [7] present T&Cs
40.         presentTandCs();
41.
42.         // [8] user agrees to terms and services
43.         if (agreeToTerms()) {
44.             // [9] user submits Sign-Up form
45.             submitSignUpForm(user);
46.
47.             // [10] create new user and store information
```

```
48.             createUser(user);
49.
50.             // [11] send verification email
51.             sendVerificationEmail(user);
52.
53.             // [12] user verifies their email
54.             if (verifyEmail(user)) {
55.                 // [13] create user account
56.                 CreateUserAccount(user);
57.             } else {
58.                 // User did not press the verification link
59.                 suspendUserAccount(user);
60.             }}}
61.
62.     // skeleton for the user submitting a username.
63.     // this method returns the username.
64.     private String submitUsername() {
65.         return ""; // user input
66.     }
67.
68.     // checks if the username is unique.
69.     // this method returns true if the username is unique, false if not.
70.     private boolean isUsernameUnique(String username) {
71.         // Placeholder for actual check
72.         return !username.equals("existingUsername");
73.     }
74.
75.     // prompts the user to enter a unique username.
76.     // returns a new unique username.
77.     private String promptUniqueUsername() {
78.         return "newUniqueUsername"; // Placeholder for user input
79.     }
80.
81.     // skeleton for the user submitting an email.
82.     // returns a email.
83.     private String submitEmail() {
84.         return "user@example.com"; // Placeholder for user input
85.     }
86.
87.     // checks if the email is valid.
88.     // returns true if the email is valid, false if not.
89.     private boolean isEmailValid(String email) {
90.         // Placeholder for actual check
91.         return email.contains("@");
92.     }
93.
94.     // prompts the user to enter a valid email.
95.     // returns a valid email.
```

```
96.     private String promptValidEmail() {
97.         return ""; // user input
98.     }
99.
100.    // skeleton for the user submitting a password.
101.    // returns a password.
102.    private String submitPassword() {
103.        return ""; // Placeholder for user input
104.    }
105.
106.    // checks if the password is secure.
107.    // returns true if the password is secure, false otherwise.
108.    private boolean isSecure(String password) {
109.        // actual check placeholder
110.        return password.length() > 6;
111.    }
112.
113.    // prompts the user to enter a secure password.
114.    // returns a secure password.
115.    private String promptForSecurePassword() {
116.        return "securePassword"; // Placeholder for user input
117.    }
118.
119.    // skeleton for presenting the T&Cs to the user.
120.    private void presentTandCs() {
121.        // Placeholder
122.    }
123.
124.    // skeleton for the user agreeing to the terms of service.
125.    // returns true for agree/confirmed.
126.    private boolean agreeToTerms() {
127.        return true;
128.    }
129.
130.    // skeleton for submitting the sign-up form.
131.    private void submitSignUpForm(User user) {
132.        // Placeholder for form submission
133.    }
134.
135.    // skeleton for creating a new user and storing their information.
136.    private void createUser(User user) {
137.        // Placeholder for creating user
138.    }
139.
140.    // skeleton for sending a verification email to the user.
141.    private void sendVerificationEmail(User user) {
142.        // Placeholder for sending email
143.    }
```

```
144.
145.     // skeleton for user verifying their email.
146.     // returns true to indicate successful verification.
147.     private boolean verifyEmail(User user) {
148.         return true; // Placeholder for actual verification
149.     }
150.
151.     // skeleton for creating the user account.
152.     private void CreateUserAccount(User user) {
153.         // Placeholder
154.     }
155.
156.     // skeleton for suspending the user's account.
157.     private void suspendUserAccount(User user) {
158.         // suspending account
159.     }
160.
161.     // class of the user's information.
162.     // holds the username, email, and password of the user.
163.     class User {
164.         private String username;
165.         private String email;
166.         private String password;
167.
168.         public String getUsername() {
169.             return username;
170.         }
171.
172.         public void setUsername(String username) {
173.             this.username = username;
174.         }
175.
176.         public String getEmail() {
177.             return email;
178.         }
179.
180.         public void setEmail(String email) {
181.             this.email = email;
182.         }
183.
184.         public String getPassword() {
185.             return password;
186.         }
187.
188.         public void setPassword(String password) {
189.             this.password = password;
190.         }
191.     }}
```

3.2. Skeleton Code for Post Question

```
// Class that shows the process for a user to post a question on CodeQA

public class UserPostQuestion {

    public static void main(String[] args) {
        UserPostQuestion postProcess = new UserPostQuestion();
        postProcess.start();
    }

    // Starts by creating a question object.
    // initiating the flow of creating and posting a question.
    public void start() {
        User user = new User("exampleUsername");
        Question question = new Question();

        // [1] user creates a question
        // the user sets the title and content for their question.
        question.setTitle(getTitle());
        question.setContent(getContent());

        // [2] system checks if the question is valid
        // to ensure it has a title and content.
        if (checkValidQuestion(question)) {
            // [3] user submits the question
            // If valid, the user submits the question.
            submitQuestion(user, question);

            // [4] system saves the question to the database.
            saveQuestion(question);

            // [5] System confirms the question is posted
            // The system confirms that the question has been posted successfully.
            confirmPosted(question);
        } else {
            // If the question is invalid, prompt the user to edit it
            // If the question is invalid, the user is prompted to make corrections.
            promptEditQuestion();
        }
    }

    // user creates a question title.
    // returns a title for the question.
    private String getTitle() {
        System.out.println("What is the question title?");
        return "";
```

```
49.     }
50.
51.     // user writing question content.
52.     // returns content for the question.
53.     private String getContent() {
54.         System.out.println("Write your question...");
55.         return "";
56.     }
57.
58.     // checks if the question is valid.
59.     // is valid if it is not empty with title and content.
60.     private boolean checkValidQuestion(Question question) {
61.         System.out.println("Checking if the question is valid...");
62.         return question.getTitle() != null &&
    !question.getTitle().isEmpty() &&
63.                 question.getContent() != null &&
    !question.getContent().isEmpty();
64.     }
65.
66.     // user submitting the question.
67.     // prints the username and the question title being submitted.
68.     private void submitQuestion(User user, Question question) {
69.         System.out.println(user.getUsername() + " posting your question
    to the community - " + question.getTitle());
70.     }
71.
72.     // saving the question in the database.
73.     // prints the title of the question being saved.
74.     private void saveQuestion(Question question) {
75.         System.out.println("Saving " + question.getTitle());
76.     }
77.
78.     // confirms the question has been posted.
79.     // prints a confirmation message with the question title.
80.     private void confirmPosted(Question question) {
81.         System.out.println("Your question '" + question.getTitle() + "'
    has been posted!");
82.     }
83.
84.     // prompts the user to edit their question if it is invalid.
85.     // prints a message prompting the user to edit their question.
86.     private void promptEditQuestion() {
87.         System.out.println("The question is not able to be posted.
    Please edit your question and try again.");
88.     }
89.
90.     // class of the user's information.
91.     // holds the username of the user.
```

```
92.     class User {
93.         private String username;
94.
95.         public User(String username) {
96.             this.username = username;
97.         }
98.
99.         public String getUsername() {
100.                return username;
101.            }
102.
103.            public void setUsername(String username) {
104.                this.username = username;
105.            }
106.        }
107.
108.        // class of the question information.
109.        // holds the title and content of the question.
110.        class Question {
111.            private String title;
112.            private String content;
113.
114.            public String getTitle() {
115.                return title;
116.            }
117.
118.            public void setTitle(String title) {
119.                this.title = title;
120.            }
121.
122.            public String getContent() {
123.                return content;
124.            }
125.
126.            public void setContent(String content) {
127.                this.content = content;
128.            }
129.        }
130.    }
```

192. Code

```
193. // -- User Class Hierarchy --
194. public abstract class User {
195.     private int userID;
196.     protected String username;
197.     private String password;
198.     protected String email;
199. 
200.     public User(int userID, String username, String password, String
    email) {
201.         this.userID = userID;
202.         this.username = username;
203.         this.password = password;
204.         this.email = email;
205.     }
206. 
207.     // Getter for userID
208.     public int getUserID() {
209.         return userID;
210.     }
211. 
212.     // Getter for username
213.     public String getUsername() {
214.         return username;
215.     }
216.     // Getter for email
217.     public String getEmail() {
218.         return email;
219.     }
220. 
221.     public abstract boolean login(String username, String password);
222.     public abstract void logout();
223.     public static boolean register(String username, String password,
    String email) {
224.         return true;
225.     }}
226. 
227. public class Guest extends User {
228.     private String sessionID;
229.     public Guest(int userID, String username, String password, String
    email) {
230.         super(userID, username, password, email);
231.     }
232. 
233.     public void browseContent() {
234.         // Code: Allows a user without an account to browse the site's
    content
```

```
235.     }}
236.
237. public class RegisteredUser extends User {
238.     protected DateTime registrationDate;
239.     public RegisteredUser(int userID, String username, String password,
     String email, DateTime registrationDate) {
240.         super(userID, username, password, email);
241.         this.registrationDate = registrationDate;
242.     }
243.
244.     public void postQuestion(Question question) {
245.         // Code: Posts a user's question
246.     }
247.
248.     public void postAnswer(Answer answer) {
249.         // Code: Posts a user's answer
250.     }}
251.
252. public class Moderator extends RegisteredUser {
253.     protected int moderationLevel;
254.     public Moderator(int userID, String username, String password, String
     email, DateTime registrationDate, int moderationLevel) {
255.         super(userID, username, password, email, registrationDate);
256.         this.moderationLevel = moderationLevel;
257.     }
258.
259.     public void deletePost(int postId) {
260.         // Code here: to delete a post
261.     }}
262.
263. public class Administrator extends Moderator {
264.     protected int adminLevel;
265.
266.     public Administrator(int userID, String username, String password,
     String email, DateTime registrationDate, int moderationLevel, int
     adminLevel) {
267.         super(userID, username, password, email, registrationDate,
     moderationLevel, adminLevel);
268.     }
269.
270.     public void manageUser(int userId) {
271.         // Code here for managing users
272.     }
273.
274.     public Report createReport() {
275.         // Code here to create a report
276.         return new Report();
277.     }}
```

```
// -- CONTENT CLASSES --
public abstract class Content {
    protected int contentId;
    protected boolean approved;

    public Content(int contentId) {
        this.contentId = contentId;
        this.approved = false;
    }

    public void approveContent() {
        this.approved = true;
    }

    public void rejectContent() {
        this.approved = false;
    }}

public class Question extends Content {
    protected String title;
    private List<Answer> answers = new ArrayList<>();

    public Question(int contentId, String title) {
        super(contentId);
        this.title = title;
    }

    public void addAnswer(Answer answer) {
        answers.add(answer);
    }}

public class Answer extends Content {
    private List<Comment> comments = new ArrayList<>();
    public Answer(int contentId) {
        super(contentId);
    }

    public void addComment(Comment comment) {
        comments.add(comment);
    }}

public class Comment extends Content {
    protected String text;
    protected User postedBy;

    public Comment(int contentId, String text, User postedBy) {
        super(contentId);
        this.text = text;
```

```
326.         this.postedBy = postedBy;
327.     }
328.
329.     public void editText(String newText) {
330.         this.text = newText;
331.     }
332. }
333.
334. // -- SUPPORTING CLASSES --
335. public class Notification {
336.     protected int notificationId;
337.     protected String message;
338.
339.     public void sendToUser(int userId) {
340.         // code to send notification to user
341.     }
342.
343.     public void markAsRead() {
344.         // code to change notification to read
345.     }
346. }
347.
348. public class System {
349.     private List<Notification> notifications = new ArrayList<>();
350.
351.     public void sendNotification(Notification notification) {
352.         notifications.add(notification);
353.     }
354.
355.     public void backupData() {
356.         // code for backup data
357.     }
358. }
359.
360. public class Statistics {
361.     protected int totalViews;
362.     protected int totalAnswers;
363.
364.     public void updateViews(int viewIncrement) {
365.         totalViews += viewIncrement;
366.     }
367.
368.     public void updateAnswers(int answerIncrement) {
369.         totalAnswers += answerIncrement;
370.     }
371. }
```