60-254 Assignment 5

Circle of Trust requires that in the list of users in a circle, once the owner of the circle leaves, the first member added to the circle by the owner (or in other words, the oldest user in the circle) becomes the new owner. As such, a **queue** is the data structure I will be using to represent a circle group, as it provides an easy way to manage the owner of the circle thanks to a FIFO structure (whoever is at the end of the queue is the owner). Posts in each circle will be held in a **queue** as well, as only the 10 most recent posts will be available for users to see at any given time, hence older posts will be removed first.

Assumptions:

1. No two users, circles, or messages can have the same user ID
2. If a user leaves the circle, their messages are not deleted if any of their messages are currently in the 10 most recent messages queue
3. A getCircle(circleID) method exists to find a specific circle using its ID
4. A getUser(userID) method exists to find a specific user using its ID
5. The following classes have already been made with the following attributes and methods:
   1. Circle
      1. Attributes:
         1. individual circle ID
         2. The current owner’s user ID
         3. A queue to store pointers to each user that is a member of the circle
         4. A queue to store messages
         5. Both queues have enqueue(), dequeue(), getLength(), and peek() methods
         6. The user queue also has a contains(userID) method to check if it points to a user with the given userID
      2. Methods
         1. getUserQueue(): Returns the user queue
         2. setUserQueue(userQueue): Sets the user queue
         3. getMessageQueue(): Returns the message queue
         4. getOwnerID(): Returns the user ID of the current owner
         5. setOwnerID(userID): Sets the user ID of the owner of the circle
         6. delete(): Delete the circle
   2. User
      1. Attributes:
         1. The user’s ID
         2. A dynamically resizing list (the structure would depend on the language, so for example, if it was java, I would use an ArrayList),containing pointers to each circle the user is a part of
      2. Methods
         1. getUserID(): Returns the user’s ID
         2. addCircle(circleID): Add a circle to the user’s circle list
         3. removeCircle(circleID): Removes a circle from the user’s circle list
      3. A method to remove a pointer to a circle to the circle list
   3. Message
      1. Attributes:
         1. The message’s ID
         2. A string containing the message’s contents

**1.** /\* Objective: Add a new post to the circle

Assumptions: userID is the ID of the user posting the message, msg is a Message object, and circle is a Circle object

Output: Return void \*/

**procedure** posting(userID, msg, circle)

**if** (circle.getUserQueue().contains(userID)

circle.getMessageQueue().enqueue(msg)

**if** circle.messageQueue.length() > 10

circle.getMessageQueue().dequeue()

**end** **if**

**end** **if**

**end** **procedure**

**2.** /\* Objective: Retrieve the 10 most recent messages from a

specific circle

Assumptions: circleID is the ID of the specific circle, userID is used to check if the user is in the specific circle

Output: Return an array containing the messages in most recent to least recent order if successful, null otherwise \*/

**procedure** getMessages(userID, circleID)

Circle temp = getCircle(circleID)

UserQueue tempUQueue = temp.getUserQueue()

MessageQueue tempMQueue = temp.getMessageQueue()

length = tempMQueue.length()

//create an array to hold the messages in the message queue

MessageArray tempArray = new MessageArray(length)

**if** tempUQueue.contains(userID)

**for** i = length to 1 inclusive **do**

//dequeue is assumed to return whatever it dequeues

tempArray[i] = tempMQueue.dequeue()

tempMQueue.enqueue(tempArray[i])

**end for**

**return** tempArray

**end** **if**

**return** null

**end** **procedure**

3. /\* Objective: Add a user to a circle that they do not own

Assumptions: circleID is the ID of the circle the user wants to join, userID is the ID of the joining user

Output: Return void \*/

**procedure** joinCircle(userID, circleID)

Circle temp = getCircle(circleID)

**if** circle.getUserQueue().contains(userID) == false

**if** temp.getUserQueue().length() <= 10

user = getUser(userID)

user.addCircle(circleID)

temp.getUserQueue().enqueue(user)

**end** **if**

**end** **if**

**end** **procedure**

4. /\* Objective: Remove a user from a circle

Assumptions: circleID is the ID of the circle the user wants to leave, userID is the ID of the leaving user

Output: Return void \*/

**procedure** leaveCircle(userID, circleID)

Circle temp = getCircle(circleID)

UserQueue tempUQueue = temp.getUserQueue()

UserQueue tempUQueue2 = new UserQueue();

length = tempUQueue.getLength()

**for** i = 1 to length inclusive **do**

//dequeue is assumed to return whatever it dequeues

User tempUser = tempUQueue.dequeue()

**if** tempUser.getUserID() != userID

tempUQueue2.enqueue(tempUser)

**else**

tempUser.removeCircle(circleID)

**if** userID == tempUser.getID() **and** length > 1

**if** i == 1

temp.setOwnerID(tempUQueue.peek())

**end** **if**

**end** **if**

**end** **if**

**end for**

temp.setUserQueue(tempUQueue2)

**if** (temp.getUserQueue().getLength() <= 0)

temp.delete()

**end** **if**

**end** **procedure**