CMPE 275 Section 1, Spring 2019
Lab 1 - Aspect Oriented Programming

Last updated: 02/27/2019

In this lab, you use AOP to enhance a simple secret management service, where one can create and read secretes, and share/unshare them with other users as well. The secret service is defined through its interface SecretService. Its implementation, however, lacks validation and access control, which we want to use AOP to implement as crosscutting concerns. We also want to use AOP to intelligently handle network exceptions by retrying for a certain number of times, and provides stats of interest for the secret service. Please note this is an individual assignment.

Service Interface

The SecretService interface is defined here:

```
public interface SecretService {
   /**
    * Creates a secret in the service. A new Secret object is created, identified
    * by randomly generated UUID, with the current user as the owner of the secret.
    *
    * @param userId    the ID of the current user
    * @param secretConent the content of the secret to be created. No duplication
    *                 check is performed; i.e., one can create different secret
    *                 objects with the same content.
    * @throws IOException                if there is a network failure
    * @throws IllegalArgumentException if the userId is null, or the secretContent
    *                                  is more than 100 characters
    * @return returns the ID for the newly created secret object
    */
   UUID createSecret(String userId, String secretContent) throws IOException,
IllegalArgumentException;

   /**
    * Reads a secret by its ID. A user can read a secrete that he has created or
    * has been shared with.
    *
    * @param userId    the ID of the current user
```

```
     * @param secretId the ID of the secret being requested
     * @throws IOException              if there is a network failure
     * @throws IllegalArgumentException if any argument is null
     * @throws NotAuthorizedException   if the given user neither has created or is
     *                                  currently shared with the given secret. If
     *                                  there does not exist a secret with the given
     *                                  UUID, this exception is thrown too.
     * @return the requested secret object
     */
    Secret readSecret(String userId, UUID secretId)
            throws IOException, IllegalArgumentException, NotAuthorizedException;

    /**
     * Share a secret with another user. A user can share a secrete that he has
     * created or has been shared with.
     *
     * @param userId    the ID of the current user
     * @param secretId          the ID of the secret being shared
     * @param targetUserId the ID of the user to share the secret with
     * @throws IOException              if there is a network failure
     * @throws IllegalArgumentException if any argument is null
     * @throws NotAuthorizedException   if the user with userId neither has created
     *                                  or is currently shared with the given
     *                                  secret. If there does not exist a secret
     *                                  with the given UUID, this exception is
     *                                  thrown too.
     */
    void shareSecret(String userId, UUID secretId, String targetUserId)
            throws IOException, IllegalArgumentException, NotAuthorizedException;

    /**
     * Unshare the current user's secret with another user. A user can ONLY unshare
     * a secret that he has created. Unsharing a message one has created with
     * himself is allowed but silently ignored, as one always has access to the
     * messages he has created.
     *
     * @param userId    the ID of the current user
     * @param secretId          the ID of the secret being unshared
     * @param targetUserId the ID of the user to unshare the secret with
     * @throws IOException              if there is a network failure
     * @throws IllegalArgumentException if any argument is null
     * @throws NotAuthorizedException   if the user with userId has not created the
     *                                  given secret. If there does not exist a
```

```
     *                          secret with the given UUID, this exception
     *                          is thrown too.
     */
    void unshareSecret(String userId, UUID secretId, String targetUserId)
                throws IOException, IllegalArgumentException, NotAuthorizedException;
}
```

Network Failure Retry

Since network failure happens relatively frequently, you are asked to add the feature to automatically retry for up to two times for a network failure (indicated by an *IOException*) through *RetryAspect.java*. Please note the two retries are in addition to the original failed invocation. If on the last retry, we still get an IOException, the call on *SecretService* fails with *IOException* thrown.

Parameter Validation

The existing implementation of *SecretService* does not do proper validation of the arguments. Please provide proper implementation for *ValidationAspect.java* such that *IllegalArgumentException* is thrown based on what's described in the interface documentation.

Access Control

The existing implementation of *SecretService* does not enforce access control and you are required to implement it in *AccessControllAspect*.java, such that *NotAuthorizedException* is thrown as documented in the interface.

Please note that our access control assumes that authentication is already taken care of elsewhere, i.e., it's outside the scope of the project to make sure only Alice can call *readSecret* with *userId* as "Alice".

Secret Stats

You need to provide the stats as defined in *SecretStats.java*. Your implementation of this interface resides in *SecretStatsImpl.java.*

```
public interface SecretStats {

    /**
     * Reset all the four measurements. For purpose of this lab, it also clears up
     * all secret objects ever created and their sharing/unsharing as if the system
     * is starting fresh for any purpose related to the metrics below.
```

```
 */
void resetStatsAndSystem();

/**
 * @return the length of the longest secret by content a user has successfully
 *     created since the beginning or last reset. If no secrets are created,
 *     return 0.
 */
int getLengthOfLongestSecret();

/**
 * If Alice shares a message foo with Bob, the tuple (Alice, foo) is considered
 * a sharing occurrence with Bob. The most trusted user is determined by the
 * maximum total number of unique sharing occurrences, each defined by a tuple
 * of (sharerID, secretID). For each of the sharing, The uniqueness of secrets
 * are defined their UUIDs; i.e., two secrets with the same content but
 * different UUIDs are considered different secrets. Unsharing does NOT affect
 * this stat. If Alice and Bob share the same secret with Carl once each, it's
 * considered as two total sharing occurrences with Carl. If Alice shares the
 * same secret he created with Carl five times and later unshares it, it is
 * still considered one sharing occurrence. Sharing a message with a user
 * himself does NOT count for the purpose of this stat. If there is a tie,
 * return the 1st of such users based on alphabetical order of the user ID. Only
 * successful sharing matters here; if no users have been successfully shared
 * with any secret, return null.
 *
 * @return the ID of the most trusted user.
 */
String getMostTrustedUser();

/**
 * The concept of unique sharing occurrences is defined the same as above. The
 * net sharing balance for a user is the total number of unique sharing
 * occurrences shared with him minus the total number of unique sharing
 * occurrences he shared with others. If Alice and Bob share the same message
 * with Carl three times each, and Carl shares the same message with Doug, Ed,
 * and Fred, Carl's net sharing balance is 2-3 = -1. Again, sharing/unsharing
 * with one himself does not count here.
 *
 * @return the ID of the person with the smallest net sharing balance. If there
 *     is a tie, return the 1st of such users based on alphabetical order of
 *     the user ID. If no users have been successfully shared with any
 *     secret, return null.
```

```
     */
    String getWorstSecretKeeper();

    /**
     * Returns the secret that has been successfully read by the biggest number of
     * different users, OTHER THAN the creator himself. If the same secret is read
     * by the same user more than once successfully, it is still considered as one.
     * If Alice shares a secret with Bob, Bob reads this secret, and later Alice
     * unshares it from Bob, Bob's read still counts because it was successful.
     *
     * @return the content of the secret that has been read by the biggest
     *     number of different users. If no secrets are ever read by users other
     *     than the creators, return null. If there is a tile, return based on
     *     the alphabetic order of the secret content.
     */
    String getBestKnownSecret();

}
```

Project Setup

The setup of the project can be accessed here, including the build file with dependencies, application context, and Java files.

Miscellaneous

You have the option to use whatever storage mechanism you prefer, including maintaining all states in memory. For simplicity, you can also assume the service will be invoked linearly, and you do not need to worry about concurrency issues or multi-threading.

To help achieving the correctness of your implementation, you can consider the following JUnit test cases.

A. test1: Bob cannot read Alice's secret, which has not been shared with Bob
B. test2: Alice shares a secret with Bob, and Bob can read it.
C. test3: Alice shares a secret with Bob, and Bob shares Alice's it with Carl, and Carl can read this secret.
D. test4: Alice shares a secret with Bob, Bob shares it with Carl, Alice unshares it with Carl, and Carl cannot read this secret anymore.
E. testJ: Alice stores the same secrete object twice, and get two different UUIDs.

Submission

Please submit through Canvas, only the FIVE java files

1. AccessControlAspect.java
2. RetryAspect.java
3. StatsAspect.java
4. ValidationAspect.java
5. Secret*StatsImpl.java.*

The code you submit must compile with the given project setup. Your five java files CANNOT include any additional classes or packages, except those under java.util or those already provided in the given build dependencies. If your code does not compile with the TA's code because of extra inclusion or dependency, you automatically lose all of your correctness points.

Due date

Pleaser refer to Canvas.

Grading:

This lab has total points of 8, all based on the correctness of the implementation.