[SECURITY LOG MANAGER 2.0] PROPOSAL

<Avani Nippani, Sanchit Razdan>

CSC316: Data Structures for Computer Scientists

<avnippan@ncsu.edu; srazdan@ncsu.edu >

North Carolina State University

Department of Computer Science

<8th April 2018>

BLACK BOX TEST PLAN

The contents of activity.txt:

```
USERNAME, TIMESTAMP, ACTION, RESOURCE
jtking, 1/18/2018 1:22:21PM, view, prescription information
mbbrown, 1/18/2018 1:23:47PM, create, immunization order
ssoulcrusher, 1/18/2018 1:22:01PM, delete, prescription information
jdschmidt, 1/18/2018 1:24:21PM, view, prescription information
jtking, 1/18/2018 12:58:14PM, delete, demographics information
```

Test ID	Description	Expected Results	Actual Results
Test1: test Generate Operational Profile at Boundary (BVA) Author: Sanchit Razdan; Avani Nippani	Preconditions: The user has specified the input file with the user activity log information. Activitylog.txt is used as the input file. The user chooses to make an operational profile to view the most frequently accessed resources The software prompts the user to enter a begin and end time to use when generating the profile for all user activities between that time period. The user inputs begin time as 1/17/2018 00:00:00AM and end time as 1/18/2018 12:58:14PM.	The program would output the following operational profile: OperationalProfile[delete demographics information: frequency: 1, percentage: 100.0%]	UI does not work because of null Pointer Exception in main method.
Test2: File does not exist (DT) Author: Sanchit Razdan, Avani Nippani	Preconditions: The user has specified the input file with the user activity log information. Activitylog.txt is used as the input file. The user specifies the input file, something.txt that does not exist.	A message stating "File not found" pops up and the software re-prompts the user to specify a new file.	UI does not work because of null Pointer Exception in main method.
Test 3: Generate Operational	Preconditions: The user has specified the input file with the user activity log	The program would output the following operational profile:	UI does not work because of null Pointer Exception in

Profile (ECP) **Author: Sanchit Razdan; Avani Nippani**	information. Activitylog.txt is used as the input file. The user chooses to make an operational profile to view the most frequently accessed resources. The software prompts the user to enter a begin and end time to use when generating the profile for all user activities between that time period. The user inputs begin time as 1/17/2018 00:00:00AM and end time as 1/20/2018 12:00:00PM.	OperationalProfile[view prescription information: frequency: 2, percentage: 40.0% create immunization order: frequency: 1, percentage: 20.0% delete demographics information: frequency: 1, percentage: 20.0% delete prescription information: frequency: 1, percentage: 20.0%]	main method.
Test 4: Produce User Output (ECP) **Author: Sanchit Razdan; Avani Nippani**	Preconditions: The user has specified the input file with the user activity log information. Activitylog.txt is used as the input file. The user chooses to produce a user report for a specified user. The software prompts the user to enter a username. The user inputs username to be jtking.	The program will output the following report: Activity Report for jtking[1/18/2018 12:58:14PM - delete demographics information 1/18/2018 1:22:21PM - view prescription information]	UI does not work because of null Pointer Exception in main method.
Test 5: testInvalidFile (DT) **Author: Sanchit Razdan, Avani Nippani**	Preconditions: The program has started. Invalid.txt is the same as activitylog.txt but the time stamps are where the user names are supposed to be. The user enters invalid.txt when prompted to enter an input file	A message stating "Invalid file" pops up and the user is re prompted	UI does not work because of null Pointer Exception in main method.

ALGORITHM DESIGN

```
Algorithm getLogForUser (E, user)
        Input E, an unsorted list of log entries
                the user for which to get logs
        Output L, a sorted list of logs
int hashcode <- user.hashcode()
int index <- compress(hashcode)
HashTable ht <- new HashTable
for int i <- 0 to E.size() - 1 do
        int hash <- E.get(i).getUsername().hashcode()
        int ind <- compress(hash)
        ht[ind].add(E.get(i))
List l <- ht[index]
MergeSort(l)
return l


Algorithm generateOperationalProfile (E, start, end)
        Input E, an unsorted list of log entries
                the start time for the profile
                the end time for the profile
        Output S, a string representation of the Operational profile

List final <- new List
TimeStamp s <- new TimeStamp(start)
TimeStamp e <- new TimeStamp(end)
int hashstart <- s.hashcode()
int hashend <- e.hashcode()
for int i <- 0 to E.size() - 1 do
        int hashcodestart <- E.get(i).getStart().hashcode()
        int hashcodeend <- E.get(i).getEnd().hashcode()
        if hashcodestart >= hashstart && hashcodeend <= hashend then
                final.add(E.get(i))
List op <- new List
op.get(0).setAction(final.get(0).getAction())
op.get(0).setResource(final.get(0).getAction())
op.get(0).setFrequency(1)
for int i <- 1 to final.size() - 1 do
        for int j <- 0 to op.size() - 1 do
                if op.get(j).getAction().equals(final.get(i).getAction()) &&
op.get(j).getResource().equals(final.get(i).getResource()) then
                        op.get(j).setFrequency(op.get(j).getFrequency()++)
                else
```

```
                              OperationalProfile op1 = new OperationalProfile(final.get(i).getAction(),
final.get(i).getResource(), 1)
                              op.insert(op1)

int total <- 0
for int <- 0 to op.size() - 1 do
      total += op.get(i).getFrequency()

MergeSort(op)
StringBuffer sb <- new StringBuffer
for int i <- 0 to op.size() - 1 do
      sb.append(op.get(i).toString())
      int percentage <- ( op.get(i).getFrequency() / total ) * 100
      sb.append(percentage + “%\n”)

return sb.toString()


Algorithm compress(hashcode)
      Input hashcode,the generated hashcode that is to be compressed.
      Output i, the index where the hashcode is to be put in the hash table array.

int m <- 117
double a <- ( 1 + sqrt(5)) / 2
double phi <- 1/a
double result <- hashcode*phi - ( (int) hashcode * phi )

return ((int) result * m)
```

DATA STRUCTURES

We are using hash table ADT for our algorithms. We are using hash tables because it is easy to hold a lot of information and to be able to add and scan through values.

We chose a hash table because the only operations we need for this are accessing values and adding values to the end of a list. To access values in a hash table is just O(1) and to insert values is O(1) as well.

An alternative data structure for this algorithm could be the linked list implementation. Rather than adding the input list in a hash table, we could add it into a linked list. We did not choose a linked list because to access values in a linked list can be O(n) in the worst case and to add values in a linked list is O(n) as well compared to O(1) in the hash table.

	Hash Table	Linked List
Insert	O(1)	O(n)
Look Up	O(1)	O(n)

If we were to input a linked list it would increase our runtime by a factor of O(n).

Algorithm Analysis

The first method estimates a T(n) of T(n) = 3n + n*log(n) + 5 and O(n) = n*log(n).

```
int hashcode <- user.hashcode()                                O(1)
int index <- compress(hashcode)                                O(1)
HashTable ht <- new HashTable                                  O(1)
for int i <- 0 to E.size() - 1 do
    int hash <- E.get(i).getUsername().hashcode()              O(1)
    int ind <- compress(ind)                                   O(1)      O(n)
    ht[ind].add(E.get(i))                                      O(1)
List l <- ht[index]                                            O(1)
MergeSort(l)                                                   O(n*log(n))
return l                                                       O(1)
```

The above method analysis shows that the T(n) = n*log(n) + 3n + 5.
The Big-Oh of the method is O(n) = n*log(n).

The second method estimates a T(n) of T(n) = n^2 + 3n + n*log(n) + 5 and O(n) = n^2.

```
List final <- new List                                         O(1)
TimeStamp s <- new TimeStamp(start)                            O(1)
TimeStamp e <- new TimeStamp(end)                              O(1)
int hashstart <- start.hashcode()                              O(1)
int hashend <-end. hashcode()
for int i <- 0 to E.size() - 1 do
    int hashcodestart <- E.get(i).getStart().hashcode()        O(1)
    int hashcodeend <- E.get(i).getEnd().hashcode()            O(1)   O(n)
    if hashcodestart >= hashstart && hashcodeend <= hashend then
        final.add(E.get(i))                                    O(1)
List op <- new List                                            O(1)
op.get(0).setAction(final.get(0).getAction())                  O(1)
op.get(0).setResource(final.get(0).getAction())                O(1)
op.get(0).setFrequency(1)                                      O(1)
for int i <- 1 to final.size() - 1 do
    for int j <- 0 to op.size() - 1 do
```

```
                    if op.get(j).getAction().equals(final.get(i).getAction()) &&
op.get(j).getResource().equals(final.get(i).getResource()) then
                              op.get(j).setFrequency(op.get(j).getFrequency()++)              O(1)     O(n^2)
                    else
                    OperationalProfile op1 = new OperationalProfile(final.get(i).getAction(),
final.get(i).getResource(), 1)
                              op.insert(op1)                                                  O(1)
int total <- 0                                                                                O(1)
for int <- 0 to op.size() - 1 do
          total += op.get(i).getFrequency()                                                   O(1)   O(n)

MergeSort(op)                                                                                 O(nlogn)
StringBuffer sb <- new StringBuffer                                                           O(1)
for int i <- 0 to op.size() - 1 do
          sb.append(op.get(i).toString())                                                     O(1)
          int percentage <- ( op.get(i).getFrequency() / total ) * 100                        O(1)       O(n)
          sb.append(percentage + "%\n")                                                       O(1)

return sb.toString()                                                                          O(1)
```

The above method analysis shows that the T(n) = n*log(n) + 3n + n^2 + 11.
The Big-Oh of the method is O(n) = n^2.

The third method estimates a T(n) of T(n) = 4 and O(n) =O(1).

```
int m <- 117                                                                                  O(1)
double a <- ( 1 + sqrt(5)) / 2                                                                O(1)
double phi <- 1/a                                                                             O(1)
double result <- hashcode*phi - ( (int) hashcode * phi )                                      O(1)

return ((int) result * m)                                                                     O(1)
```

The above method analysis shows that the T(n) = 4.
The Big-Oh of the method is O(n) = 1.

SOFTWARE DESIGN

We will be implementing a HashTable data structure for our program. We are using Model View Controller to make the project.

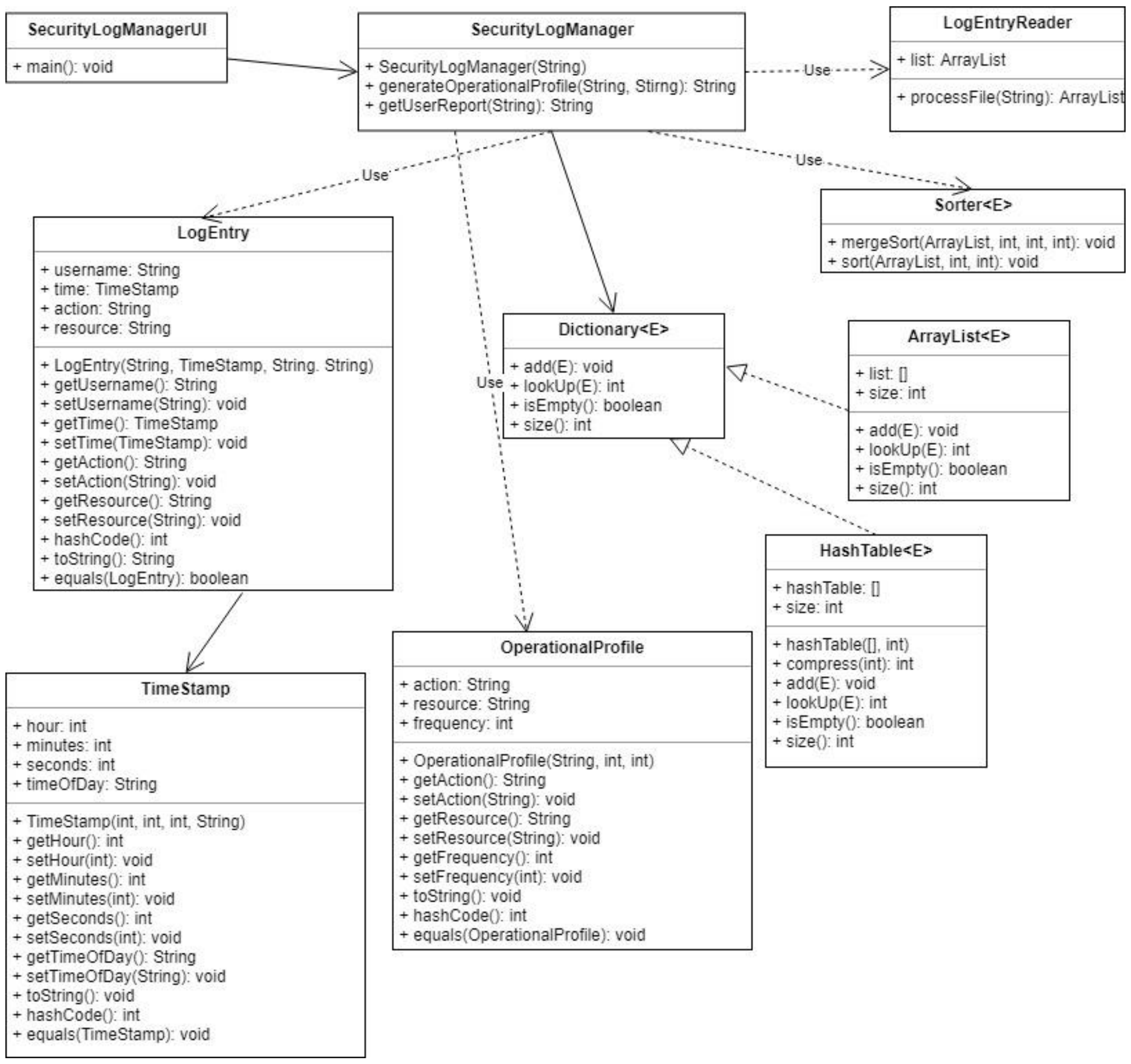
The model of the UML contains the following classes:

1. TimeStamp:
   The timestamp class makes a new timestamp object. It has the hours, minutes, seconds and the time of day attributes.
2. LogEntry:
   The Log Entry class creates a new Log Entry object. It has action, resource, the username and the timestamp as its attributes. This class is used in the second operation.
3. HashTable:
   The hashtable class makes a new hashtable to store hash codes of objects. the hash codes are made using usernames. It makes it easier to access the array and find the list with the same user names. The hashtable is used in the second operation of the program. The instance of a hashtable makes the program much more efficient and faster.
4. ArrayList:
   This class makes a new arraylist object with common arraylist attributes and methods.
5. Sorter:
   The Sorter class contains a Sorter object. It can perform operations to sort a list of objects using the MergeSort implementation.
6. OperationalProfile:
   The OperationalProfile class creates a new OperationalProfile. It includes action, resource and the frequency of the description. It is used for the operational profile operation. It is basically used to make the operational profile instance of each log entry. The operation in the manager can directly use instances of the operational profile to output strings.
7. SecurityLogManager:
   This class is the class where all the operations are being carried out. This class uses all the other classes in printing out the two strings needed from the two operations. It also coordinates with the reader class for the input file and the UI class for the output.

The view of the UML contains the SecurityLogManagerUI class. It interacts with the user and asks for the operations to be done and prints the required strings.

The controller of the UML contains the LogEntryReader class. It contains a static method which reads the input file and makes an arraylist of the log entries.

The UML diagram shows the connections between all the classes of the MVC.


SecurityLogManagerUI
+ main(): void
SecurityLogManager
+ SecurityLogManager(String)
+ generateOperationalProfile(String, Stirng): String
+ getUserReport(String): String
LogEntryReader
+ list: ArrayList
+ processFile(String): ArrayList
Use
Use
Use
Use
Sorter<E>
+ mergeSort(ArrayList, int, int, int): void
+ sort(ArrayList, int, int): void
LogEntry
+ username: String
+ time: TimeStamp
+ action: String
+ resource: String
+ LogEntry(String, TimeStamp, String. String)
+ getUsername(): String
+ setUsername(String): void
+ getTime(): TimeStamp
+ setTime(TimeStamp): void
+ getAction(): String
+ setAction(String): void
+ getResource(): String
+ setResource(String): void
+ hashCode(): int
+ toString(): String
+ equals(LogEntry): boolean
Dictionary<E>
+ add(E): void
+ lookUp(E): int
+ isEmpty(): boolean
+ size(): int
ArrayList<E>
+ list: []
+ size: int
+ add(E): void
+ lookUp(E): int
+ isEmpty(): boolean
+ size(): int
HashTable<E>
+ hashTable: []
+ size: int
+ hashTable([], int)
+ compress(int): int
+ add(E): void
+ lookUp(E): int
+ isEmpty(): boolean
+ size(): int
TimeStamp
+ hour: int
+ minutes: int
+ seconds: int
+ timeOfDay: String
+ TimeStamp(int, int, int, String)
+ getHour(): int
+ setHour(int): void
+ getMinutes(): int
+ setMinutes(int): void
+ getSeconds(): int
+ setSeconds(int): void
+ getTimeOfDay(): String
+ setTimeOfDay(String): void
+ toString(): void
+ hashCode(): int
+ equals(TimeStamp): void
OperationalProfile
+ action: String
+ resource: String
+ frequency: int
+ OperationalProfile(String, int, int)
+ getAction(): String
+ setAction(String): void
+ getResource(): String
+ setResource(String): void
+ getFrequency(): int
+ setFrequency(int): void
+ toString(): void
+ hashCode(): int
+ equals(OperationalProfile): void