Spring 2025
CMPE 223 Homework_1
Section_3

Tolga Kurtuluş Çapın

Burak Efe Taşkın 10300188292
Arda Eray Başparmak 1010520243

22.03.2025

Introduction

In this assignment the objective was writing a dynamic song lyrics auditing system which has multiple steps and operations for user to access desired output. This assignment is written in java with using the techniques that we have learned so far from the java 1 and java 2 classes. OOP is necessary for this and also, we were tasked to build our own LinkedList with an own Node class, Stack and Queue classes as well.

Approach & Methodology

We approached the problem separating it to 3 smaller problems. First problem was; defining and finding then writing the LinkledList, Stack, Queue and Node classes for continue the code. While writing we decided to use the least memory used way which was such in Stack we used the technique, as we have learned in class, increasing the array size by multiplying 2 and when the array is filled by its quarter size decrementing the array's size by dividing 2.

After writing our own implementations of this java library classes, we decided to carry on with writing the correlated classes of Lyrics as: SongLyric which uses LinkedList class, StepQueue using Queue class, SongLyricStack which uses Stack class. By this methodology we can revert the mistakes that we made while writing the Node, LinkedList, Stack and Queue classes. Also, this method helped us to approach the problem in a different way as taking SongLyricAuditingSystem class by its own, which was easier to debug because of the simplicity and understanding how the code works of the previous classes.

The last problem was the SongLyricAuditingSystem and Testing class. In SongLyricAuditingSystem class we were asked to handle the user input to change the desired operation in the code. Which seemed easy at first glimpse however because of the multiple classes that needed to implement we had a problem as not switching to the desired operation in switch case method. So that we used If-Else statement which may decrease the efficiency of the code but while writing to code we cannot solve this minor problem.

Problem Statement and Code Design

The Song Lyrics Auditing System manages the entire lifecycle of song lyrics by moving them through three main phases: Drafting, Auditing, and Final Recording. Each lyric is represented as a **SongLyric** object, which includes a unique ID, a title, the current phase, a status flag indicating whether it's considered crucial, and a record of any feedback or edits. Once the lyric reaches the final stage of review, it's either approved or rejected based on the input provided by one or more assigned auditors.

In practice, users can create and enqueue new lyrics for review, where they first undergo a Drafting step for an initial check, then move on to the Auditing stage for detailed feedback, and finally proceed to the Final Recording stage for acceptance or rejection. Throughout this process, each lyric can have multiple auditors, and their collective decisions are tracked in a list called **auditorStatuses**, which determines the lyric's ultimate outcome. If a lyric is flagged as crucial, it's moved to a high-priority stack to ensure it receives prompt attention. Additionally, the system keeps a running history of all actions—referred to as **auditingHistory**—so users can easily see whether a lyric is awaiting review, approved, rejected, or fully finalized, and also trace how it got their, step by step.

Implementation and Functionality

1- SongLyric Class:
Attributes: lyricID, name, currentStep, status, auditors, auditorsStatuses, auditingHistory, formalAuditingFlag

Methods:
+SongLyric(int lyricID, String name, boolean formalAuditingFlag)
+addAuditor(String auditor): Adds an auditor.

+allAuditorsApproved(): Checks if all auditors are approved.

+addAuditing(String auditor, String auditing): Adds feedback and updates the status variable.

+updateStatus(String status): Updates the lyric's status.

+Getters: Gets the ID, Name, CurrentStep, Status and AuditingHistory.

+updateCurrentStep(String step): Updates the current step in the process.

+getLatestAuditing() : Returns the latest auditing feedback.

2- StepQueue Class:
Attribute:
Queue<SongLyric> queue : Defines a queue for SongLyric class.

Methods:

+ StepQueue(): Initializes an empty queue.

+enqueue(SongLyric songLyric): Adds a lyric to the queue.

+dequeue(): Removes and returns the next lyric.

+isEmpty(): Checks if the queue is empty.

+viewQueue(): Prints all lyrics in the queue.

3- SongLyricStack Class:
Attribute: Stack<SongLyric> stack: Defines a stack for SongLyrics class.

Methods:
+SongLyricStack(): Initializes an empty stack.

+push(SongLyric songLyric): Adds a lyric to the stack.

+pop(): Removes and returns the top lyric.

+isEmpty(): Checks if the stack is empty.

+viewStack(): Prints all lyrics in the stack.

4- SongLyricAuditingSystem Class:
Attributes: songLyrics, draftingQueue, auditingQueue, finalRecordingQueue, crucialSongLyrics.

Methods:
+ SongLyricAuditingSystem(): Initializes the system with empty queues and a stack.

+addSongLyric(SongLyric songLyric): Adds a song lyric to the system and drafting queue.

+processNextSongLyricInQueue(String step): Processes the next song lyric in the specified step.

+moveSongLyricToNextStep(SongLyric songLyric Moves a song lyric to the next step based on approval.
+moveSongLyricToStack(SongLyric songLyric): Moves a rejected song lyric to the stack.

+showStack(): Displays the crucial song lyric stack.

+showQueue(String step): Displays the queue for the specified step.

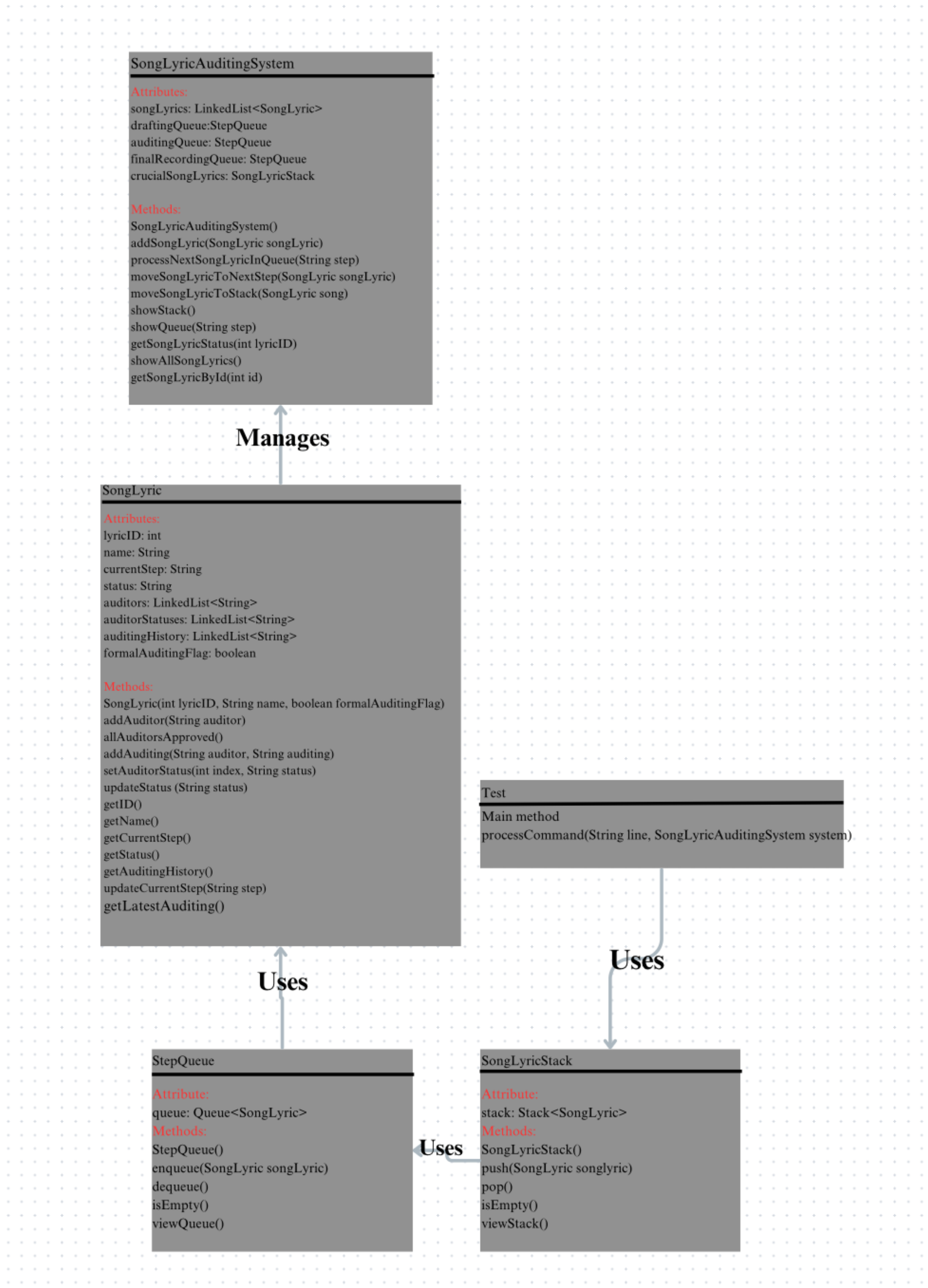
+getSongLyricStatus(int lyricID): Retrieves the status of a song lyric by ID.

+showAllSongLyrics(): Displays all song lyrics in the system.

+ getSongLyricById(int id): Retrieves a song lyric by ID.

5- Test Class:
Reads input from a “sample_input.txt” file, calls system functions and prints results.

Structure Chart


SongLyricAuditingSystem
Attributes:
songLyrics: LinkedList<SongLyric>
draftingQueue:StepQueue
auditingQueue: StepQueue
finalRecordingQueue: StepQueue
crucialSongLyrics: SongLyricStack
Methods:
SongLyricAuditingSystem()
addSongLyric(SongLyric songLyric)
processNextSongLyricInQueue(String step)
moveSongLyricToNextStep(SongLyric songLyric)
moveSongLyricToStack(SongLyric song)
showStack()
showQueue(String step)
getSongLyricStatus(int lyricID)
showAllSongLyrics()
getSongLyricById(int id)
Manages
SongLyric
Attributes:
lyricID: int
name: String
currentStep: String
status: String
auditors: LinkedList<String>
auditorStatuses: LinkedList<String>
auditingHistory: LinkedList<String>
formalAuditingFlag: boolean
Methods:
SongLyric(int lyricID, String name, boolean formalAuditingFlag)
addAuditor(String auditor)
allAuditorsApproved()
addAuditing(String auditor, String auditing)
setAuditorStatus(int index, String status)
updateStatus (String status)
getID()
getName()
getCurrentStep()
getStatus()
getAuditingHistory()
updateCurrentStep(String step)
getLatestAuditing()
Test
Main method
processCommand(String line, SongLyricAuditingSystem system)
Uses
Uses
StepQueue
Attribute:
queue: Queue<SongLyric>
Methods:
StepQueue()
enqueue(SongLyric songLyric)
dequeue()
isEmpty()
viewQueue()
Uses
SongLyricStack
Attribute:
stack: Stack<SongLyric>
Methods:
SongLyricStack()
push(SongLyric songlyric)
pop()
isEmpty()
viewStack()

Testing

Testing was performed using a combination of unit and integration tests. Each class LinkedList, Stack, Queue, SongLyric, SongLyricAuditingSystem, etc. was first verified in isolation, then tested collectively through the Test class using commands from “sample_input.txt”.

1. **Unit Testing**

• Verified LinkedList, Stack, and Queue operations independently: add/remove, push/pop, enqueue/dequeue.

• Confirmed that SongLyric updates status correctly after each operation as adding auditors and adding feedback.

2. **Integration Testing**

• Executed the Test class with the sample input to ensure SongLyricAuditingSystem coordinates all steps.

• Checked how crucial or rejected lyrics are moved to the stack and whether statuses are properly updated.

3. **Boundary Testing**

• Attempted adding lyrics with duplicate IDs.

• Processed queues when empty.

• Checked behavior when no auditors are added but feedback is attempted.

Key Observations and Results

• **Drafting to Auditing Transition:**

After the PROCESS_QUEUE Drafting command, SongLyric 9 and SongLyric 10 were moved to the Auditing queue.

• Result: SongLyric 9 progressed to Auditing first, followed by SongLyric 10.

• **Auditor Feedback and Approval:**

• SongLyric 9 received “Approved” feedback from Auditor K.

• SongLyric 10 received “Rejected” feedback from Auditor L.

• Result: SongLyric 9 moved forward to Final Recording, while SongLyric 10 was pushed to the crucial/rejected stack.

• **Final Recording:**

• Upon the second PROCESS_QUEUE Final Recording call, SongLyric 9 was confirmed to be fully approved.

• Result: SongLyric 9 had a final status of "Approved."

• **Stack Operations:**

• SongLyric 10 was pushed to the SongLyricStack due to rejection.

• The SHOW_STACK command displayed SongLyric 10 as expected.

• **SHOW_ALL_SONGLYRICS:**

• Demonstrated that the system retained both SongLyric 9 approved and SongLyric 10 rejected in the final output.

• Result: Verified that historical data was accurate.

Overall, the test confirmed that the system behaved correctly in normal and edge-case scenarios. Any minor inefficiencies e.g: using if-else for multiple operations did not impact correctness.

Conclusion

The trouble points while we were completing this assignment were Coordinating transitions between Drafting, Auditing, and Final Recording queues without losing track of a lyric's status. And also, as this much handling rejected lyrics by pushing them to the stack at the correct time was another problem that we were faced. The most challenging part in this assignment for us were implementing custom data structures (LinkedList, Stack, Queue) from scratch and ensuring they worked seamlessly together as we have mentioned in the introduction. Managing multiple auditors and ensuring feedback logic was correct. We have enjoyed applying queues and stacks in a practical, multi-stage workflow. Learned valuable lessons about organizing code for readability and maintaining consistent state transitions. And most importantly gained confidence in building and debugging custom data structures in Java.