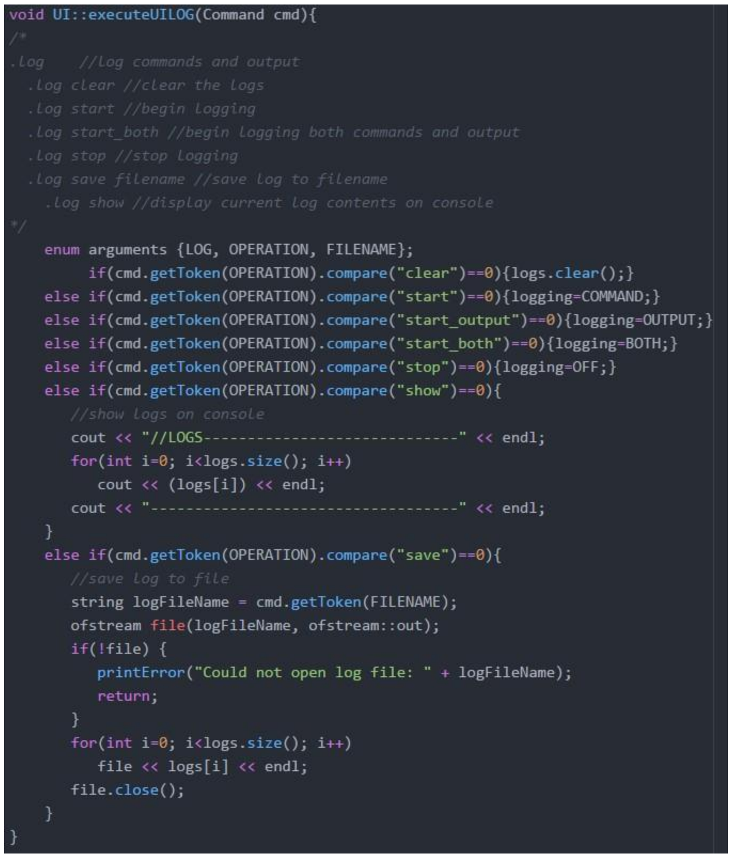
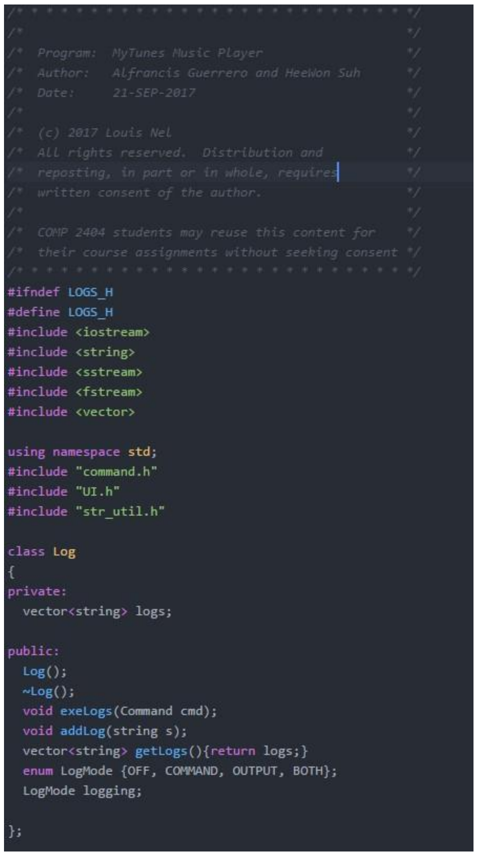
Assignment 3 – Refactoring

We noticed that the functions for read, help, and log were all enclosed in the UI class. In our previous assignments we thought it was more effective to have our own classes for read, help, and log. So we have decided to refactor the log function into its own class. We are not changing the overall functionality of the software but we are effectively improving readability, reusability, and maintainability. Our class for logging generally makes the code for logging more modular for the application.



```
void UI::executeUILOG(Command cmd){
/*
.log    //log commands and output
  .log clear //clear the logs
  .log start //begin logging
  .log start_both //begin logging both commands and output
  .log stop //stop logging
  .log save filename //save log to filename
    .log show //display current log contents on console
*/
    enum arguments {LOG, OPERATION, FILENAME};
         if(cmd.getToken(OPERATION).compare("clear")==0){logs.clear();}
    else if(cmd.getToken(OPERATION).compare("start")==0){logging=COMMAND;}
    else if(cmd.getToken(OPERATION).compare("start_output")==0){logging=OUTPUT;}
    else if(cmd.getToken(OPERATION).compare("start_both")==0){logging=BOTH;}
    else if(cmd.getToken(OPERATION).compare("stop")==0){logging=OFF;}
    else if(cmd.getToken(OPERATION).compare("show")==0){
       //show logs on console
       cout << "//LOGS----------------------------" << endl;
       for(int i=0; i<logs.size(); i++)
          cout << (logs[i]) << endl;
       cout << "----------------------------------" << endl;
    }
    else if(cmd.getToken(OPERATION).compare("save")==0){
       //save log to file
       string logFileName = cmd.getToken(FILENAME);
       ofstream file(logFileName, ofstream::out);
       if(!file) {
          printError("Could not open log file: " + logFileName);
          return;
       }
       for(int i=0; i<logs.size(); i++)
          file << logs[i] << endl;
       file.close();
    }
}
```



```
/* * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                    */
/*  Program:  MyTunes Music Player                    */
/*  Author:   Alfrancis Guerrero and HeeWon Suh       */
/*  Date:     21-SEP-2017                             */
/*                                                    */
/*  (c) 2017 Louis Nel                                */
/*  All rights reserved.  Distribution and            */
/*  reposting, in part or in whole, requires          */
/*  written consent of the author.                    */
/*                                                    */
/*  COMP 2404 students may reuse this content for     */
/*  their course assignments without seeking consent */
/* * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifndef LOGS_H
#define LOGS_H
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>

using namespace std;
#include "command.h"
#include "UI.h"
#include "str_util.h"

class Log
{
private:
  vector<string> logs;

public:
  Log();
  ~Log();
  void exeLogs(Command cmd);
  void addLog(string s);
  vector<string> getLogs(){return logs;}
  enum LogMode {OFF, COMMAND, OUTPUT, BOTH};
  LogMode logging;

};
```

Figure 1: Before

Figure 2: After

Before refactoring, it is hard to find where can you change or modify how the read commands work without knowing too much of the application itself. If you are a new developer you want to be able to find what certain functions you are looking for without going through another generic class.

For example UI.h we founded it to be a very abstract class if you were to give a definition of what it does. It would execute multiple functions such as executing all the regular and developer commands. However if a developer wanted to modify how the logging functionality works, they would have to go through the entire UI class just to change the functionality of logging. This can make the UI class have a lot of extra libraries that only pertain to logging making it messier and not **maintainable** depending on the scale.

This refactoring method makes the logging functionality reusable for other developers to reuse without even looking through other classes. They will be able to just focus on implementing logging

For example if the logging functionality needed to be organized by songs alphabetically, or users by ascending order in song size, you would want this to be done inside the logging class. These functions can be practical for data analysis purposes where organization and efficiency is crucial for large scale user programs. (ie. Spotify and their large user base)

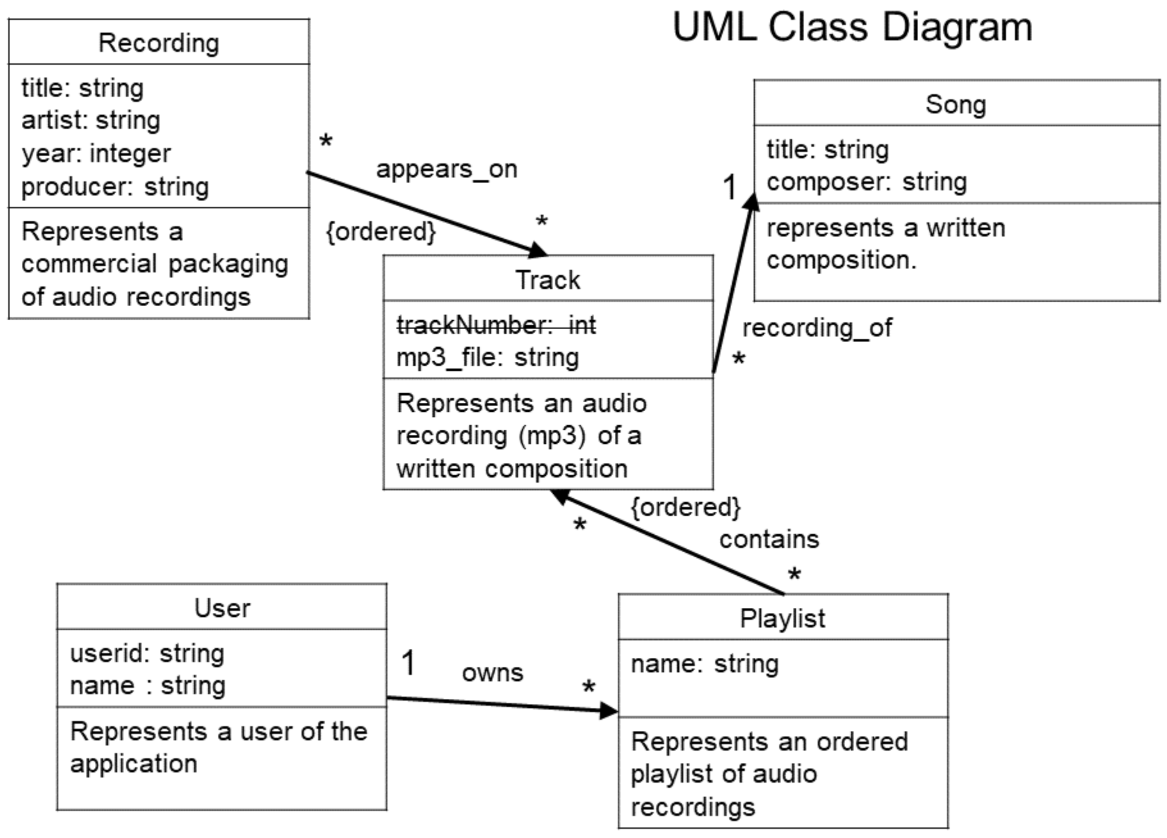


Figure 3: Before

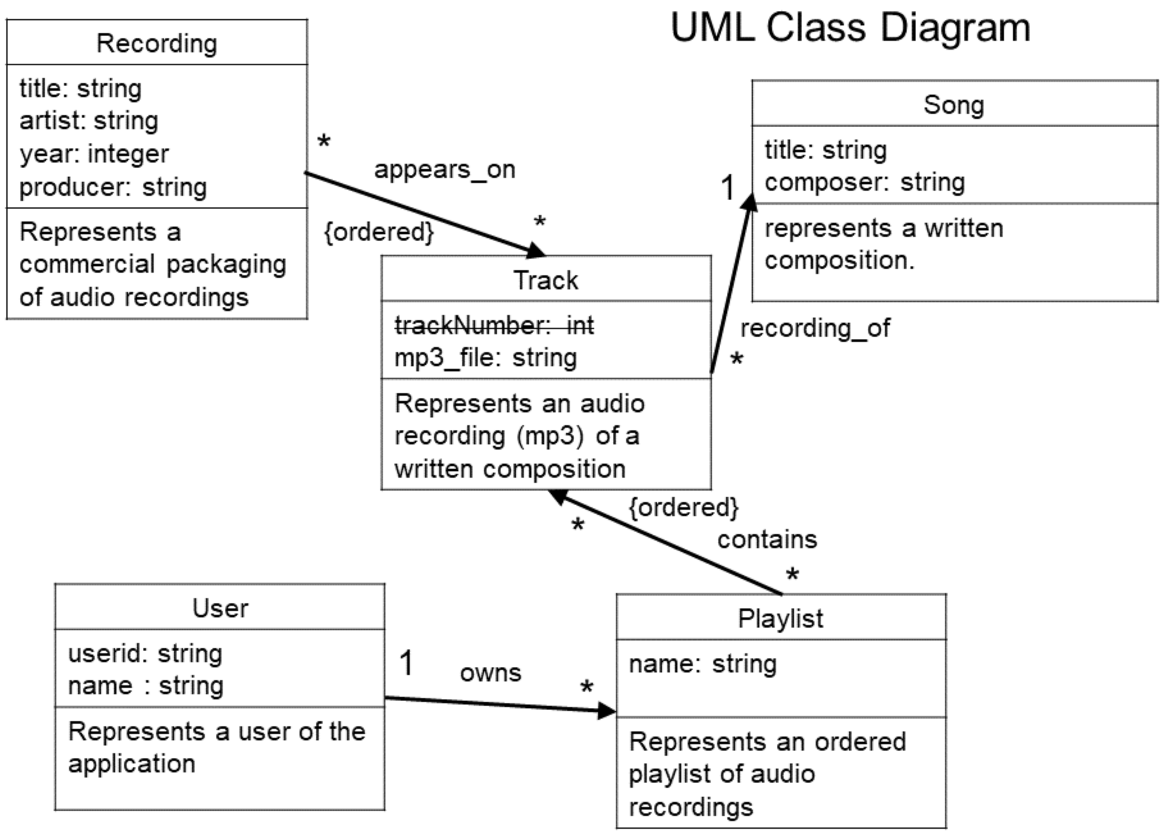


Figure 1: After

UML structure has not been changed, the changes that have been made are from UI model.

In summary, by making the logging functionality more modular and expandable for developers to work in, we improve its maintainability and reusability with simplifying how it's implemented and making the class expandable for further and more complex functions for data analysis if needed.

Readability, reusability, maintainability, scalability, encapsulation, decoupling, fragility, efficiency etc.