***Documentation Front Page:***

**All Group Members:** Claire Burkhardt, John Jarvis, Victor Percival

**Responsibilities:**

Claire B. – Audio processing, user interface, driver, group management, Doxygen

John J. – File I/O

Victor P. – User interface, UML diagram

**Description of the Design:**

In accordance with SOLID Design principles, we designed the three areas of concern (File I/O, Audio Processing, User Interface) separately, then unified all their functionality in the Driver.cpp file. The following sections from each group member reflects on their part of the design and challenges experienced:

**Claire B:**

My primary area of design was Audio Processing. The biggest technical requirement of the section was that the processing needed to be able to handle Mono/Stereo and 8/16 bit data. This meant the functions doing the audio processing needed to be able to read several different types of data, making them perfect candidates for templates. I created three templates (one for each of echo, noise gate, and normalizer) that read in either unsigned char (8 bit) or short (16 bit) data types. In doing so, the central challenge became creating code that was robust enough to process the different data types. I achieved this by doing all of my audio processing math with variables (max/min/amplitude/zeroValue) and basing the values of those variables on their data type. For example, rather than hardcoding 128 in as the value of zero for 8-bit and then 0 as the value of 0 for 16-bit, I just did the calculations with the variable zeroVal (which had an assigned value based on the data type of the template). As a result, the templates were able to handle all the different permutations of data.

I ended up handling the design and implementation of Driver.cpp file (and a part of File I/O as a part). The central challenge here was combining the functionality of each area of interest without crossing their modular variables/dependencies/jobs. I managed to achieve this primarily through switch statements (to create a systematic design and walk the use through their options/steps) and by creating half-a dozen previously unforeseen functions in File I/O to specify options for the user to choose form.

**John J:**

Over the course of this project, I have had the opportunity to really flex my knowledge of the C++ language, and implement things I never thought I’d have the chance to do. While I did have many successes, I did run into problems. The SOLID design principles was my first challenge as I had to make my classes concise, while also adhering to the grading guidelines of including templates, inheritance, etc. At first, this was a real problem. I had to come up with ways to make my code work, but be contained into neat classes that only do 1, or two things in general. I decided on having a class that handles general file things (opening, finding sizes, closing, etc.), one that handles the more specific aspects of WAV files (my soundFile class), and finally at the end of my SOLID funnel, I had a class for my 8/16 bit files that handled specific printing/writing. I found the SOLID principles to actually really help me decide on where to make changes to my code, and what to put in said classes, without getting lost as to where my code was. The real problems I had with this program come down to two aspects of a wav file: metadata, and the writing of 8/16 bit files. For metadata, I had massive difficulty even trying to access the data. This was very frustrating to me as I easily accessed the technical data of a WAV file but I never got metadata access to work, and I couldn’t figure it out by the end of the project. My second major problem was actually writing out of 8/16 files. There was a seg fault somewhere in my handling of the 16/8 bit sound files at their respect classes, and I really just didn’t understand what I was doing wrong, or how to fix it. In the future, I plan on attending more office hours to complete my specific issues, because during the course of this semester I was actually very intimidated by going to office hours because I didn’t want to look stupid. Now, towards the end, I really wish I took advantage of the help and overcame my irrational fears of not looking smart. I did have a lot of fun with this project, and this class, and hope to continue my CS degree.

**Victor P:**

For my portion of the project, I worked on the implementation of audio normalization and the program’s user interface. When designing the user interface, one major challenge that I came across was finding a way to implement the selection of the logic to interact with the main function solely through the header. I figured out that I should have an object be created for the Processor class, and that object will be used to call the functions, such as analyze and display audio information. I also split the code for user interaction was kept outside of the main function, and these functions would instead be called *inside* the main function. The logic for this was thus kept outside of the main function in accordance with the prescribed instructions. I attempted to work within all the SOLID principles when designing this portion of the project, specifically the Open/Closed Principle. I created the main structure of the program and it had basic functionality, while also leaving it open for other team members to implement their portions of the project into the user interface. It also followed the Single-Responsibility Principle, in that this portion was sure to only display a menu to the user and allow the user to make a selection which would in turn execute another function. This also follows the Interface Segregation Principle, since it does not use any interfaces. The user interface is designed in the following way: I created a class titled UserInterface. This class has two files, UserInterface.h and UserInterface.cpp. I included functions for setters and getters of the user’s input, and a function that printed to the console a menu selection system, and various submenus for different audio types. In the main.cpp file, I created two functions which included pulling the user’s input and a switch statement for the user’s selection. I created two of these functions, one for the main menu and another for a submenu. While designing the audio normalization, I did not run into any major challenges while creating this part; I only needed to do a little research on how audio normalization worked and viewed example code of a similar type of audio processing. Basically, audio normalization is increasing the amplitude of lower peaks in the audio file in order to increase the overall volume of an audio file. In other words, all parts of the audio file should be about the same volume as the rest; there are not too many differences between the loudest parts and the quietest parts. I attempted to stay within the SOLID principles of object-oriented programming, and some of the best examples of adherence to these principles were found in the principle of the Open/Closed Principle, in that even if small changes to the program were made, these would not affect the overall functionality. I designed this program as a class titled Processor. I intended this program to be a part that would be added to an existing Processor class. It is comprised of the files Processor.h and Processor.cpp. The way that the program works is to find the loudest part of the audio file, and multiply all other parts by a specified multiplier.

**UML Diagram:** Saved as a PDF “UML Diagram.pdf” for clarity (it’s a horizontally oriented document)