Aarjav Kothari aakothar@ucsd.edu PID: A91087718
Shrivatsan Rajagopalan s1rajago@eng.ucsd.edu PID: A53245201
Ruochao Yan ruy007@ucsd.edu PID: A53247716
Alan Yessenbayev ayessenb@ucsd.edu PID: A92411003

SFArchiver Technical Report

Aarjav Kothari aakothar@ucsd.edu PID: A91087718
Shrivatsan Rajagopalan s1rajago@eng.ucsd.edu PID: A53245201
Ruochao Yan ruy007@ucsd.edu PID: A53247716
Alan Yessenbayev ayessenb@ucsd.edu PID: A92411003

Introduction:

We created the SFArchiver program (further referred to as **the Program**) to manage binary files that we refer to as **archives**. We decided to use .arc file extension for our archive binary files. Each archive contains the head with all the metadata about the archive, called the **directory.** The interfacing between the native OS command-line facilities and the Program is implemented through the **parser**.

Archive:

The archive object is responsible for writing to the archive binary file. Because of the problems of the linear approach to managing the archive, we went with the blocks approach. We chose the block size of 1024 bytes (1 kilobyte). The archive uses the directory object to keep track of the location of files inside the directory, as well as, empty blocks that are present in an archive. The archive class also implements the remaining functions that modify the archive. The add method creates an archive if one is not present and can also be used to add files to an existing directory. It calculates the number of blocks needed for the file and checks for empty blocks. Otherwise the file is appended to the end of the archive. The delete method checks for a valid file and deletes it if present. It also invokes the defragmentation method to check for empty spaces and combines them. The list function simply returns all the files in the archive or a specific file present in the archive depending on the user input. Each file has a ‘date added’ property associated with it which is also returned when the list command is called. The find method runs through each file in the archive sequentially while running through each block internally in every file to look for the target string.

Directory:

The directory object reads and writes the head of the archive. It keeps track of the size of the directory in blocks (as a size_t), the list of empty blocks (as a queue), and each file that is

Aarjav Kothari aakothar@ucsd.edu PID: A91087718
Shrivatsan Rajagopalan s1rajago@eng.ucsd.edu PID: A53245201
Ruochao Yan ruy007@ucsd.edu PID: A53247716
Alan Yessenbayev ayessenb@ucsd.edu PID: A92411003

recorded in the directory (as a map of FileName/FileEntry). Each FileEntry object keeps track of the file name, file type, the size in bytes, the vector of blocks were the file is written and the date added. This permits for a simple implementation of the list command. The directory is split into “rows” by the newline character. The directory’s constructor reads the file, and it also creates a file if proper flag was passed to the constructor. Both Directory and FileEntry objects, have operator<< overloaded to permit easy writing to the archive (and also to follow principles of polymorphism and encapsulation). From the compiler’s perspective, the directory is a top part of the .arc file, that is terminated by the “EOF” string. From the coder’s perspective, Directory is a data structure used in management of the archive files. From the designer’s perspective, the directory abstracts the problem of keeping track of files, from writing files to the directory. The size of the directory for a newly created archive is one block. If by any chance, the directory becomes bigger than its current size, we increase the size by one block and move the block that is on the way to the tail of the archive.

Parser:

The parser class has been implemented to act as the primary interface with the user. The command sent by the user is sent to the parser. A parser object has a set of valid commands with which it first identifies if a command is valid command and suitably responding if the command provided by the user is not a valid one. This completely eliminates a bad command in the first stage of operations and does not carry forward to the rest of the process. The parser also detects if the arguments passed to the command are correct and prints a separate message if that is not the the case. If the command and arguments are valid, the parser calls the necessary function on an archive object. Commands which do not modify an archive object like the test and version commands are directly handled by the parser. The parser however does not handle data validity. This is done by the respective functions in the archive class.

Tests:

Numerous tests were carried out to check the working of the code and iron out any bugs. Additionally, tests to characterize the performance of the program were carried out. One particular test case was the deletion of a file stored after a number of large files (Test 1). It was compared with the deletion of the same file stored in the beginning of the archive (Test 2). The average execution times are given below.

Test Case	Average Execution Time (milliseconds)
Test 1	6.988
Test 2	6.893

Aarjav Kothari	aakothar@ucsd.edu	PID: A91087718
Shrivatsan Rajagopalan	s1rajago@eng.ucsd.edu	PID: A53245201
Ruochao Yan	ruy007@ucsd.edu	PID: A53247716
Alan Yessenbayev	ayessenb@ucsd.edu	PID: A92411003

Such a result is obtained because the program does not traverse through the files but rather traverses the blocks as it is aware of the start and end of all files. This means that the difference in execution time is negligible and will vary based on other characteristics like access time.

Conclusion/Possibilities for improvement:

This report details the working of the SFArchiver program. The various modules present in the program and their interactions and working has been presented.

Working together taught us all, that teamwork is the key. We all have a long way to go in terms of expressing our programming ideas using natural language (aka English). Also, it is important to invoke effective design practices when it come to coding in a team. For example, we believe that poor choice of names for the Block class methods caused us to fall back to hard coding block size into the program. We will be looking for these things in our future collaborations.