Problem 1

This is a group homework problem. Your group will be same as the one that you are enrolled in

for the team project in this course.

Find one open-source project that is closest to your team project. Reverse engineer the design

and a few architectural/design patterns from that project.

https://github.com/vincentlaucsb/csv-parser/tree/master

Describe the above reverse engineered details in the following format. If applicable, provide

pointers to project’s directories/files/classes in your descriptions and/or diagrams.

An interesting aspect of the project is that you can calculate the statistics of large files.

1. Purpose (1 point)

a. Briefly explain the chosen open-source project and its purpose.  
  
Vince’s CSV Parser is a project in C++ for parsing CSV (Comma Separated Values) file. Although the name of the project we chose is “Vince’s CSV Parser,” this project is open source and contributed-to by many people, not just “Vince.” This project is designed to be intuitive and simple to use for most users and it has an extensive test suite.

To be more specific, the project is inspired by Python’s csv module. Therefore, Vince wanted to create a csv library that was simple and intuitive just like the Python module. Since this project was written in C++ and our parsing module is such a huge component of our project, we decided to analyze this open-source parsing module.

b. Provide the link to the open-source project repository.  
https://github.com/vincentlaucsb/csv-parser/tree/master

2. Design Outline (2 points)

a. Identify its components and describe the purpose of each component.

There are interesting components associated with reading and writing to a CSV file

* Reading an arbitrary file (The File can be incredibly large)
  + The big purpose of the project that is being developed is to easily read and parse csv files with C++. The typical problems with parsing big files are essential reasons that this library should be used: for example, speed is a big requirement that solutions implemented in languages like Python may not be able to fulfill.
  + Therefore, creating a simple, very usable library that help read these files with an efficient implementation is quintessential
* Converting to JSON
  + Serialize the individual rows of the file into JSON objects
    - The keys of the returned JSON object are column names
  + This approach serves multiple purposes, the foremost of which is to allow for parsed data/rows in general to be in JSON
* Parsing In Memory Strings
  + You can create a string in the program itself, and it will be able to parse information for it
  + Or, you can have strings as rows and have the parser read it row by row

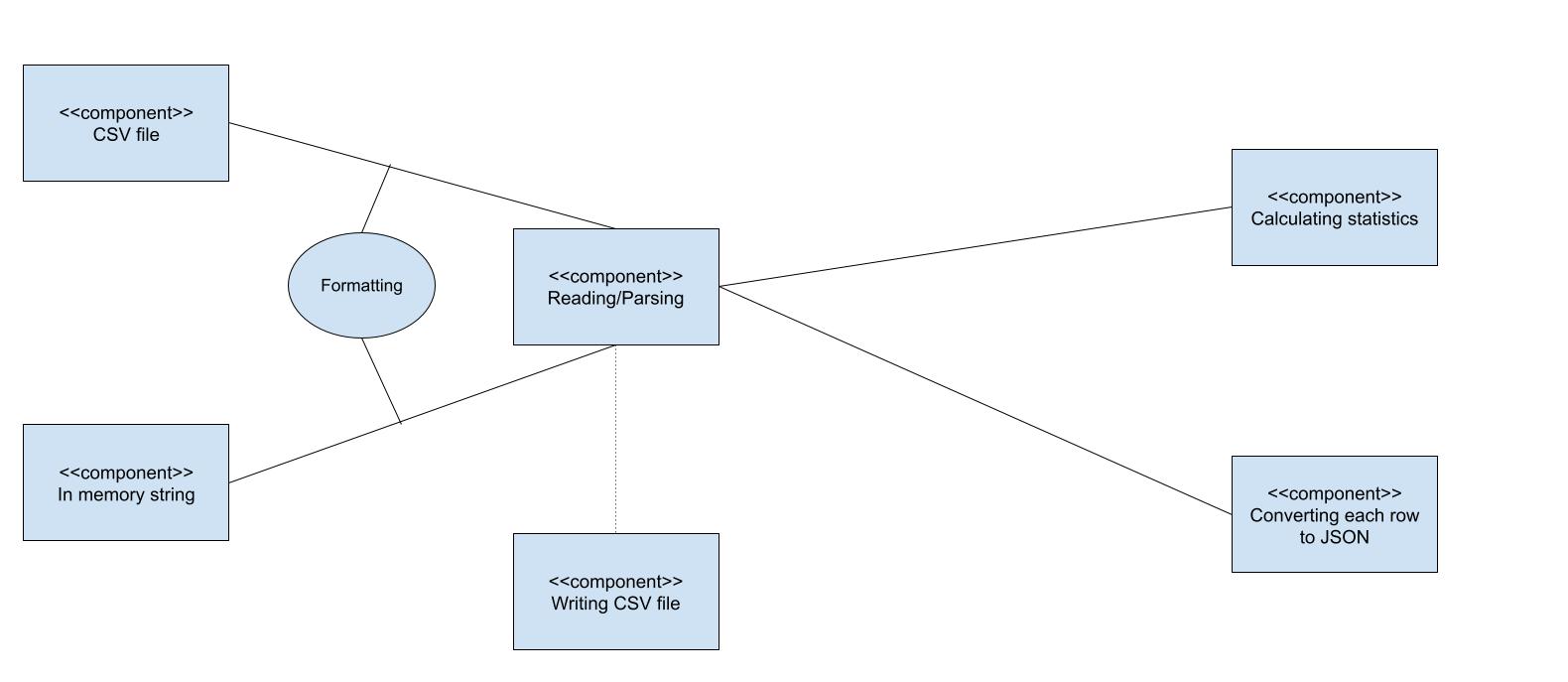
* Specify the Format of the CSV
  + For example if you CSV file has columns or has a header file on the 3rd row, you can let the program know, and it will account for it
    - In fact, it makes it easier to parse
  + There are many specifications you can make such as telling the parser to trim whitespace, handling rows of different lengths, etc.
* Displays the Statistics of a File
  + The CSV files may contain data in which it will be useful to display the statistics of the files
  + Such statistics would be variance, count, min, max, data types, etc.
  + These statistics will be useful for astonishingly large files
* Writing CSV FIles
  + On top of parsing, you will be able to write CSV files with your own formats and your own data, cementing this library as a more-complete way to and a solid-choice for interacting with the CSV file format within C++.

b. Describe the interactions between components.

* First, we will have either a csv file or a in memory string (using std::string) and read and parse them
* In terms of reading and parsing, we will be able to specify the format in which we parse/read
* After parsing and reading
* We will be able to calculate and display the statistics of a file such as the means, variances, maxes, etc.
* We can also convert each row into JSON objects
* Additionally, we will be able to write our own csv files with our own data from the program
  + In fact, with our own csv file, we can parse it again if we choose to

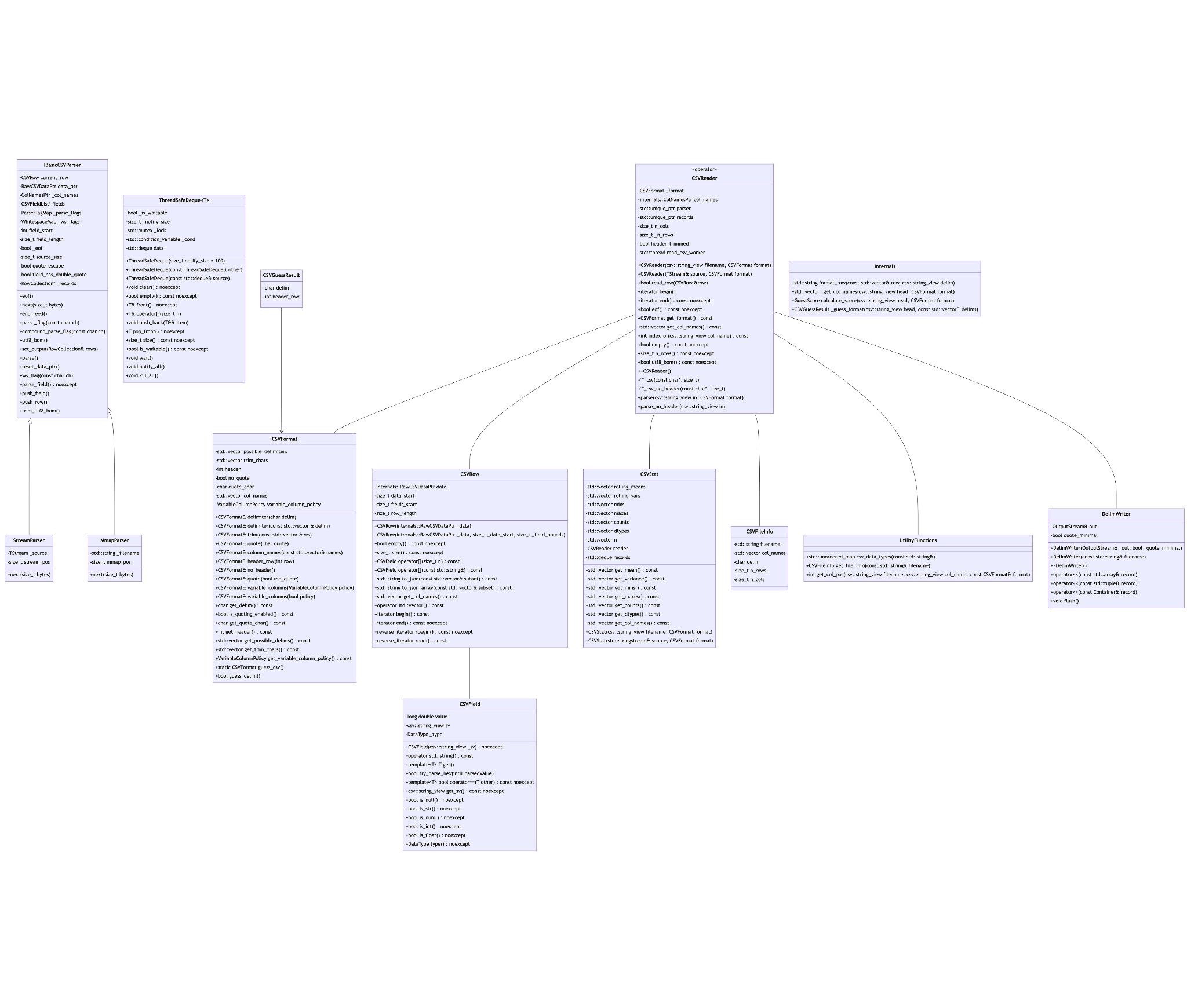
c. Include at least one UML diagram that clearly shows high-level structure of the

system.



3. Design Details (7 points)

a. Include class level design (i.e., class diagrams) and be as detailed as you can.



b. Describe the classes and interactions between the classes.

**IBasicCSVParser**: This class provides an interface for basic functionality for parsing CSV data. It includes methods for navigating through the CSV data, parsing individual fields and handling quotes and whitespace.

**StreamParser** and **MmapParser**: Concrete implementations of the **IBasicCSVParser** interface, **StreamParser** is designed to parse CSV data from a stream, where **MmapParser** utilizes memory-mapped files for parsing.

**ThreadSafeDeque:** This class represents a thread-safe double-ended queue. It can handle concurrent access, allowing multiple threads to push and pop elements safely. When buffering CSV rows and fields, this class can provide support for multi-threaded processing.

**CSVFormat:** Encapsulates the format specifications for CSV file, such as the delimiter character, quota character, and the present of header row. The class give parsing behavior more customization to match the input data.

**CSVReader:** The class responsible for reading and interpreting CSV data according to the provided **CSVFormat**, It also utilizes the parser class to perform actual parsing. It offers various methods for iteration over the CSV rows and accessing individual fields.

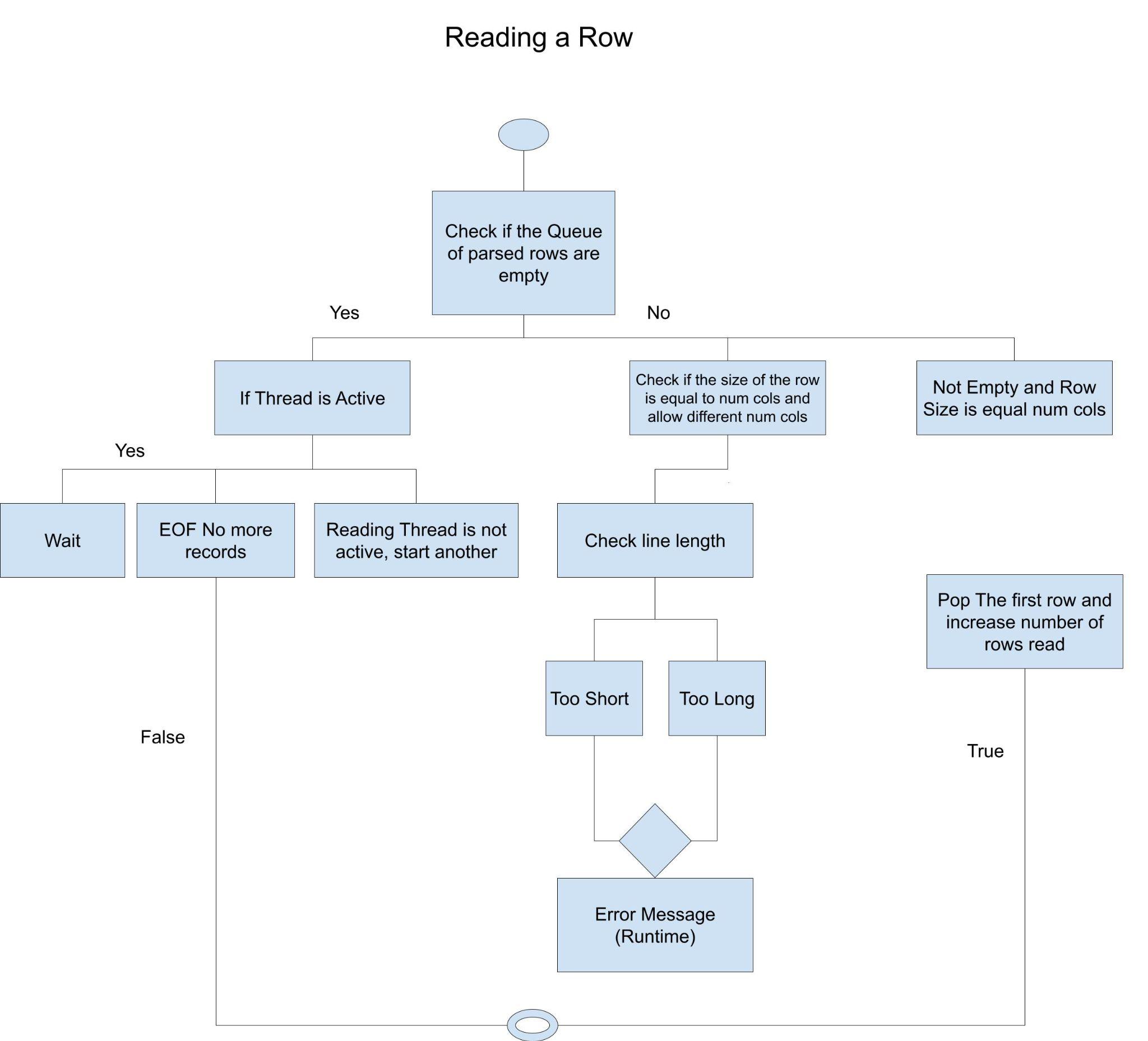
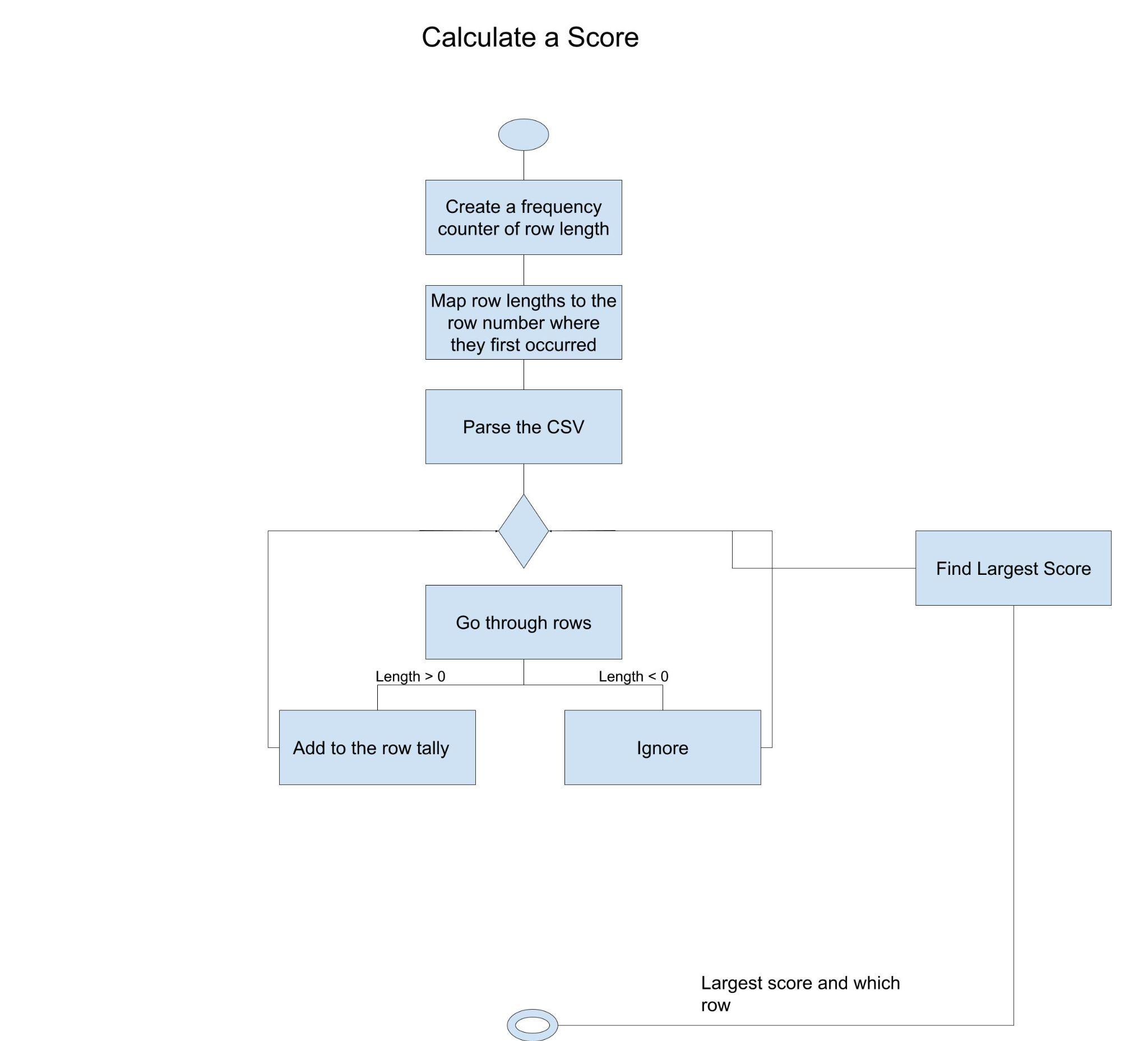
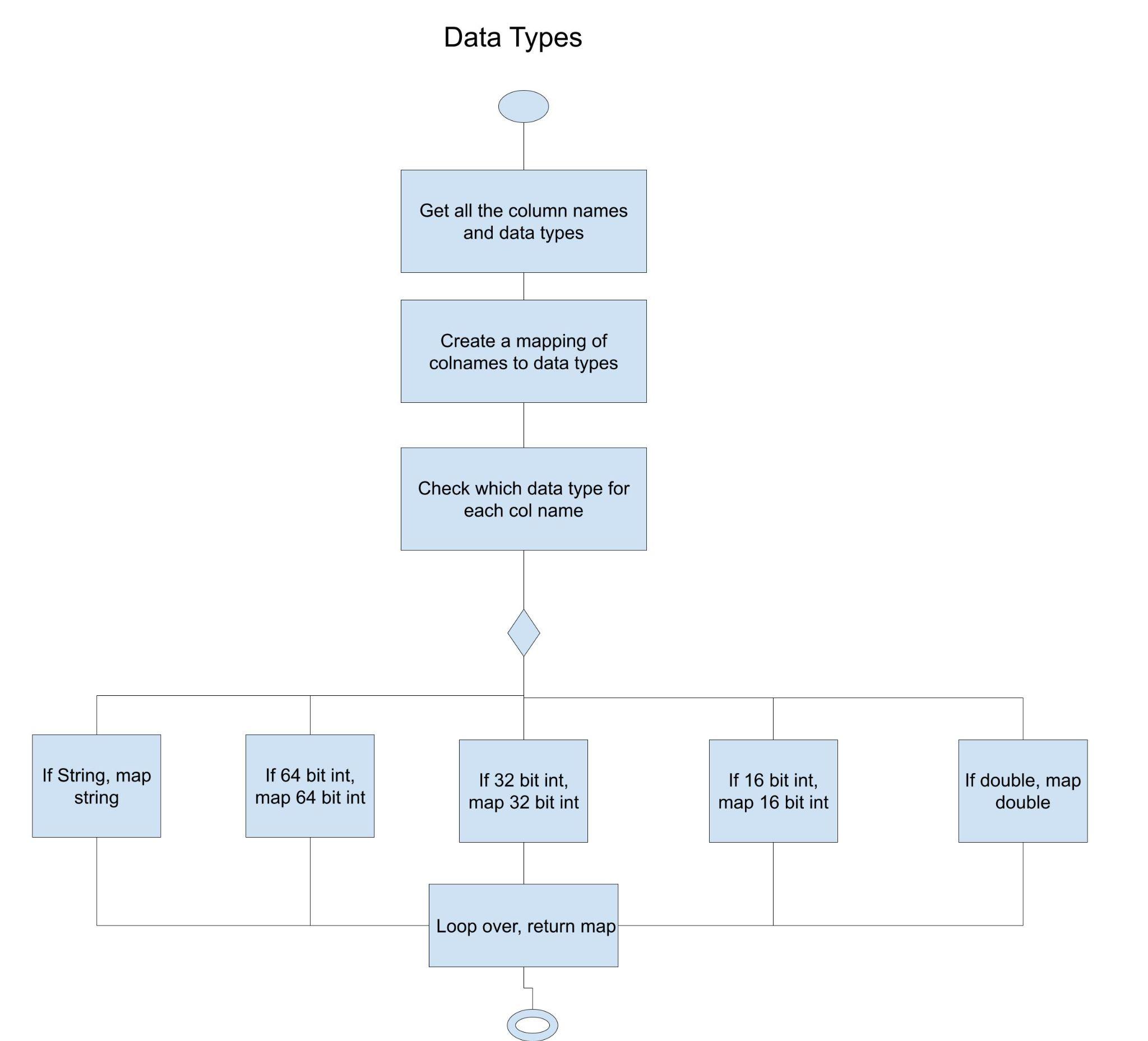
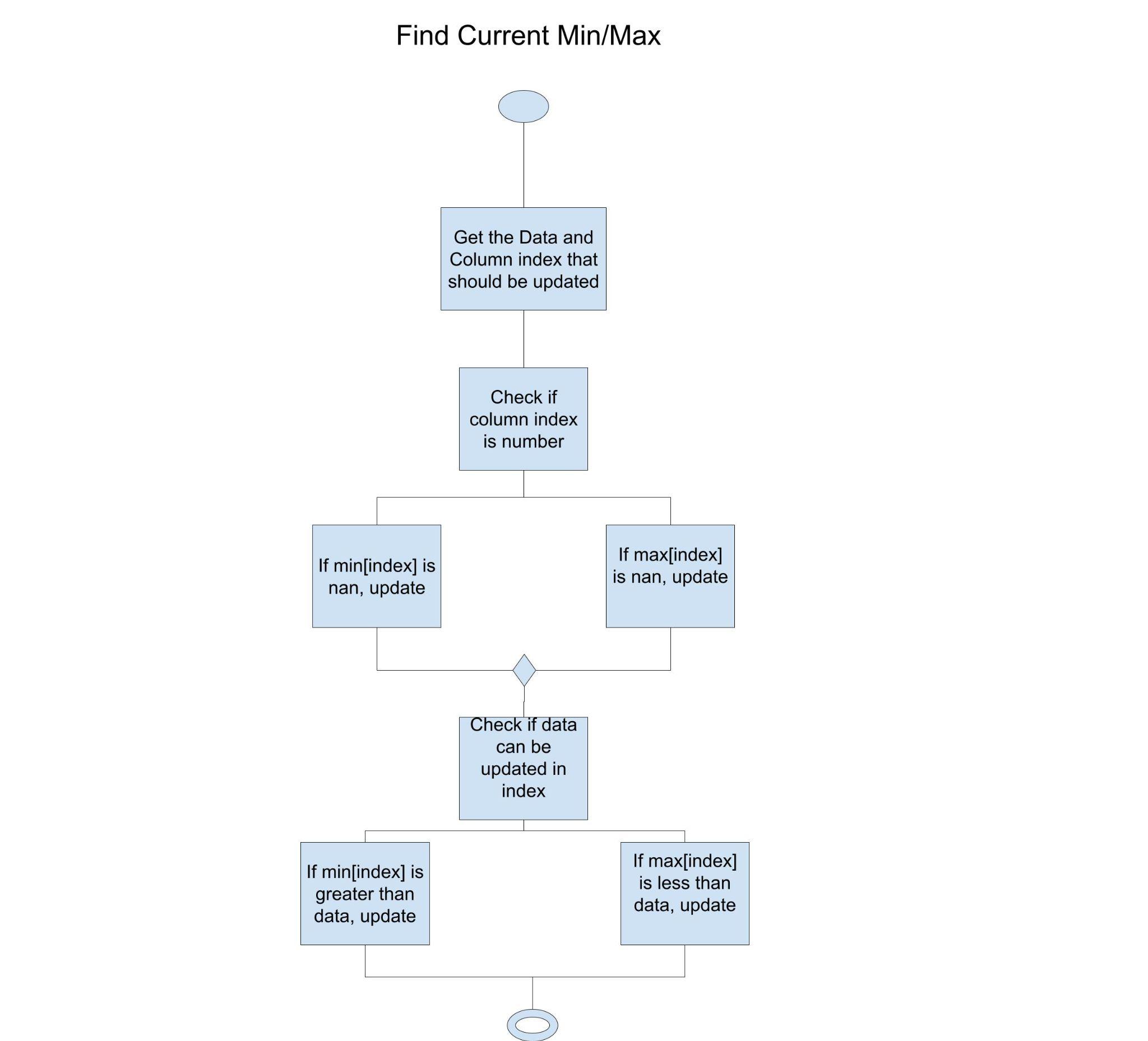
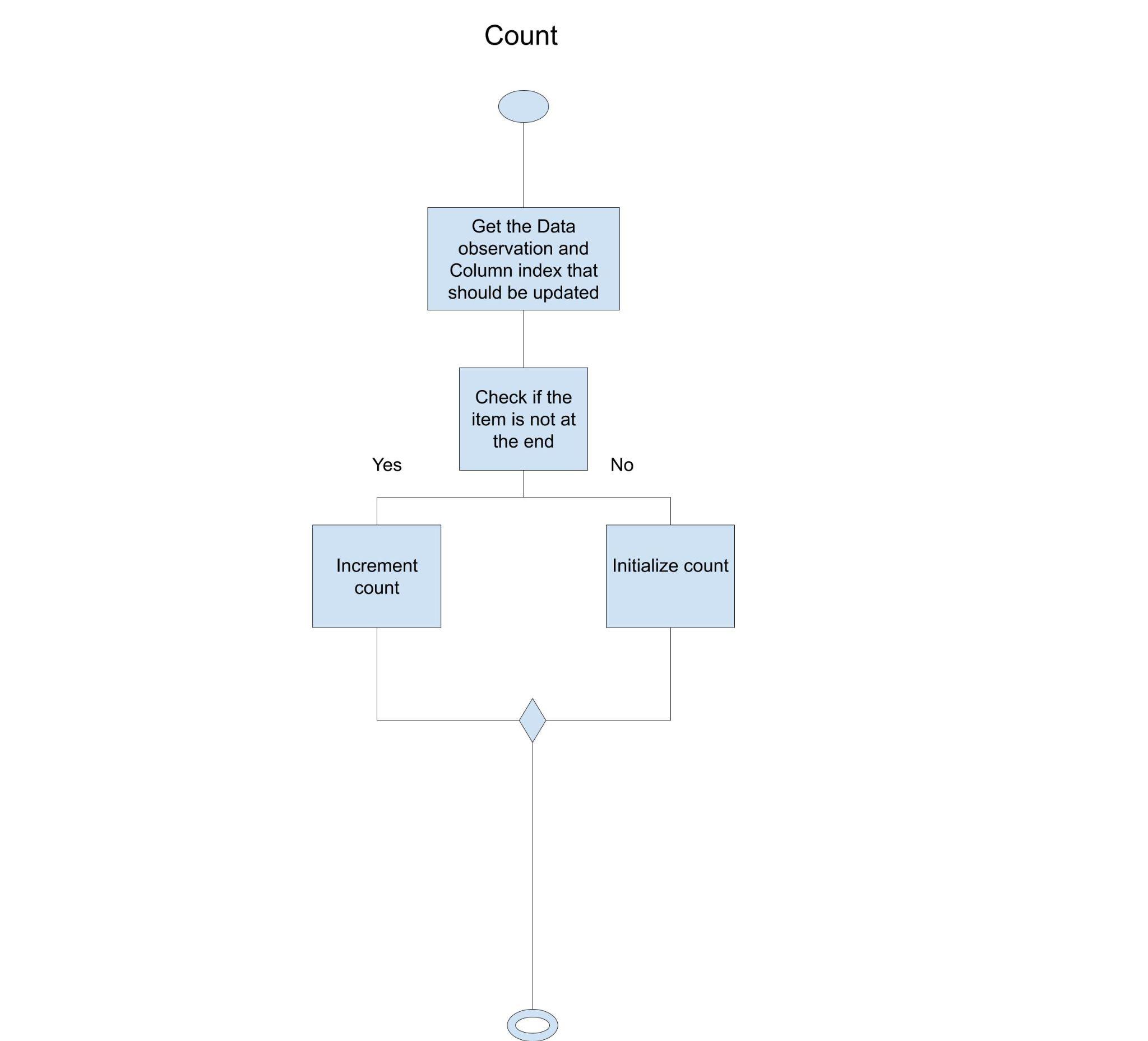
**CSVField and CSVRow:** The class represents a field and a row within a CSV file.

**CSVStat:** The class can provide statistical analysis to the CSV data, like calculating the descriptive statistics like mean, variance, minimum etc.

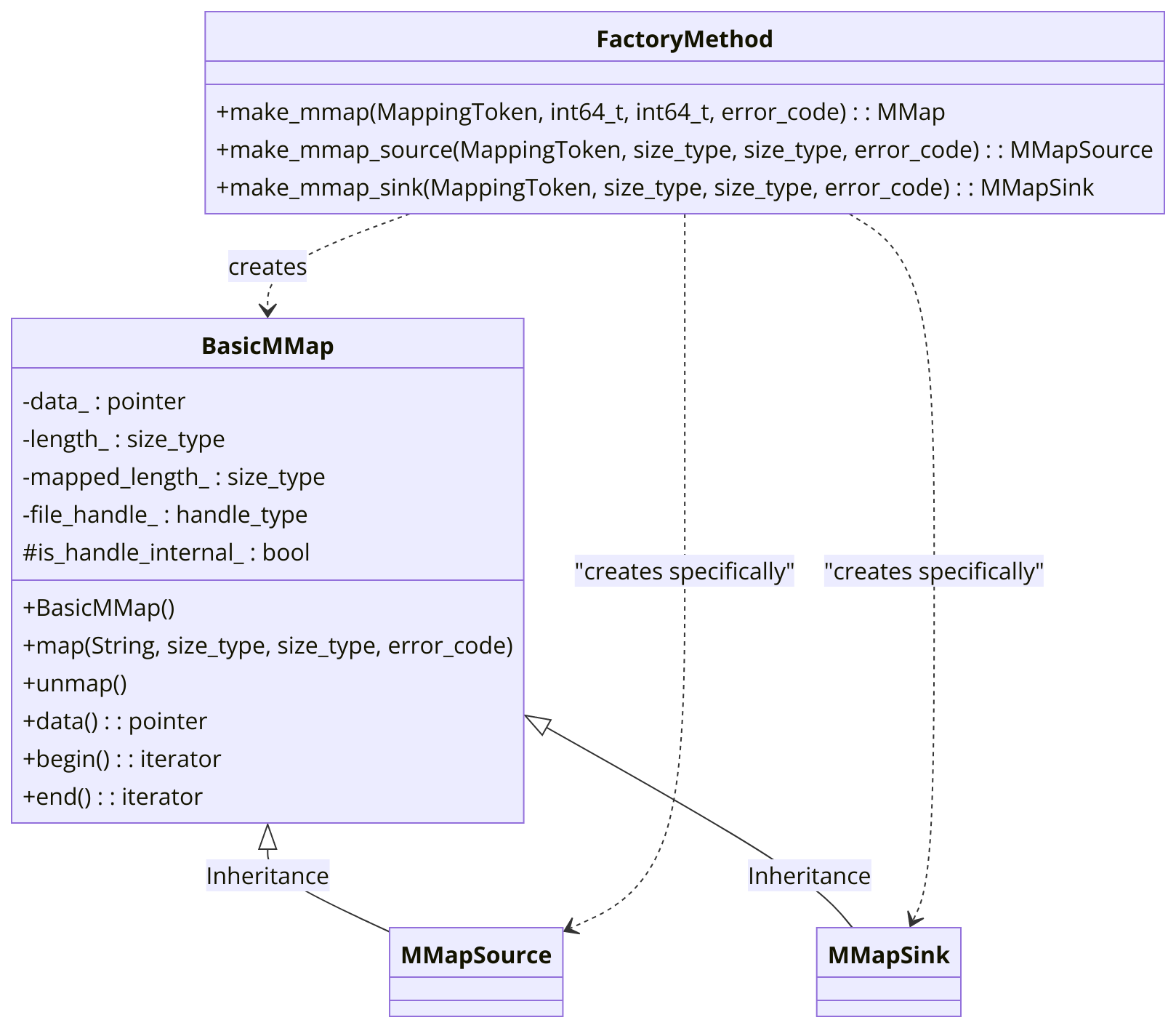
**UtilityFunctions, CSVFileInfo** and **DelimWriter:** **UtilityFunctions** have methods to determine the data type of fields, **CSVFileInfo** gather metadata of a CSV file, **DelimWriter**  help writing CSV data with custom formatting or delimiters.

c. Include sequence diagrams for different activities and/or activity diagrams and/or

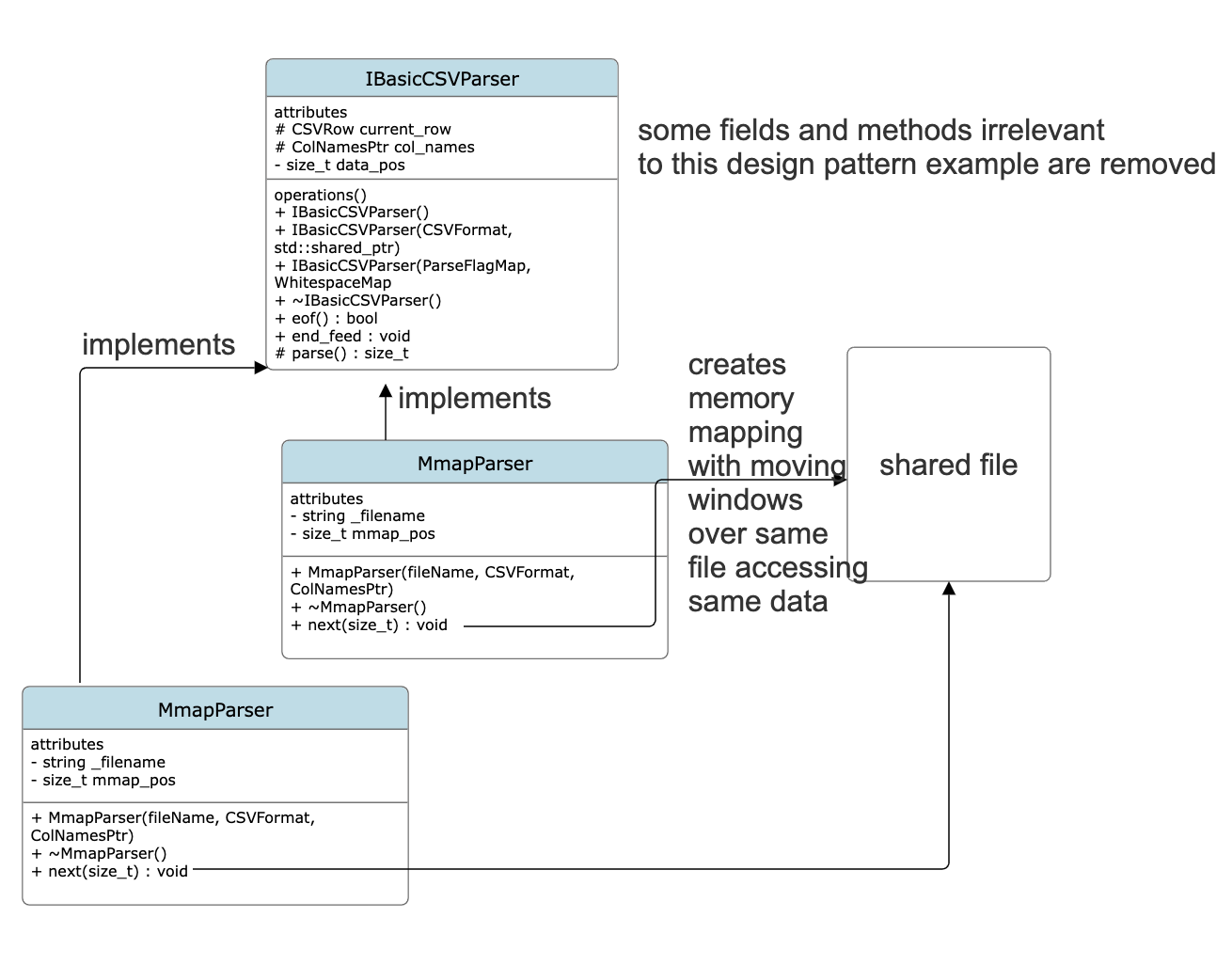
state diagrams (at least five UML diagrams).

1. 
   1. This activity diagram reflects how the parser reads a row
   2. The rows are read through a queue, so the programs use a queue
   3. The method returns a boolean value if the row was read correctly or not
   4. The Source file is csv\_reader.cpp and the line numbers are 272 - 310
2. 
   1. Goes over which row has the highest score
   2. Calculates all the scores of the row and finds the max score
   3. The source file is csv\_reader.cpp, and the lines are 41 - 69
3. 
   1. This figures out the data types for each column in the csv file
   2. Maps each column name to a data type and returns it
   3. Goes from line 240 - 267 in source file csv\_stat.cpp
4. 
   1. This activity diagram determines if a column index should be updated with observation
   2. First checks if col is nan
   3. Then checks if the min and max index need to be updated
   4. This is in the csv\_stat.cpp source file from lines 192 - 208
5. 
   1. This activity diagram shows the frequency counter
   2. This is in the source file from csv\_stat.cpp from lines 174 - 190

4. Design Patterns (4 points)

1. Factory Method  


The **make\_mmap**, **make\_mmap\_source**, **make\_mmap\_sink** act as factory methods to encapsulate the creation logic of **basic\_mmap** object and they also share the same base class. The code is located at line 560 of csv.hpp in the single\_include folder.

2. Flyweight Pattern (structural design pattern)

The MmapParser class’s **next()** method calls mmap to map from a CSV row position to real physical memory. You can have multiple MmapParsers per one file, so objects can access the same shared physical memory even though they are separate. Therefore, this behavior exhibits the Flyweight structural design pattern. You can see this happen in **basic\_csv\_parser.cpp** line 232 while the mmap itself is created at line 241 with the code using the mio library to accomplish this.

5. Design Issues (1 point)

a. Identify one functional design issue and one non-functional design issue.

**Functional**:

**Enhancing Stream Flexibility in CSVReader:** Users want to use the parser with more generic streams, so that they could read from compressed files.

**Solution**:

The project has undergone a major redesign to more flexibly accommodate a variety of input stream types. Previously, CSVReader supported reading data from `std::string` and `std::ifstream` primarily by copying the data to an internal `std::string`, which was inefficient. To address this limitation and enhance the functionality of the library, the project introduced the `IBasicCSVParser` interface and included `StreamParser` as a generic class derived from this interface. With the redesign, CSVReader can directly process and parse data from `std::stringstream`, memory-mapped files, and `std::ifstream` without a lot of repetitive code. In addition, CSVReader provides direct support for reading compressed files, due to the desire to more efficiently process CSV files stored in formats such as gzip or bzip2.

**Alternative solution:**

**Direct std::istream Support**: Refactor StreamParser to operate with std::istream& references, eliminating the need for move semantics. This modification allows seamless integration with all types of streams derived from std::istream, including non-movable streams.

**Compression Plug-ins**: Implement a plug-in system where developers can easily add support for new compressed stream types (like gzip or bzip2). Each plug-in would be responsible for decompressing data on-the-fly and presenting it as a standard std::istream to the CSVReader.

**Non-Functional:**

**Performance Optimization of CSV Parser Libraries**

**Solution:**

Performance optimization of the CSV parser library takes into account the raw speed of parsing large files, as well as memory usage efficiency, adaptability to various hardware configurations, and the ability to efficiently handle a wide range of file sizes and formats. The authors have ensured that the library delivers optimal performance in a variety of user scenarios, from fast processing of small files to efficient processing of multi-GB datasets, without compromising speed or resource utilization. The authors have integrated a number of benchmarks and configuration files into the code to identify performance bottlenecks in the software and address them.

**Alternative solution:**

Machine learning algorithms can be utilized to predict the optimal parsing strategy based on file characteristics and system performance metrics. By analyzing snapshots of file size, structure, and current CPU and memory load, the system can make an intelligent choice between single-threaded, multi-threaded, or parallel processing techniques. In addition, the integration of intelligent buffering techniques that adjust to observed throughput and system memory pressure can further improve performance. This ensures that the program library adapts to the specifics of the CSV file, aligning its resource usage with the current operating state of the system.

