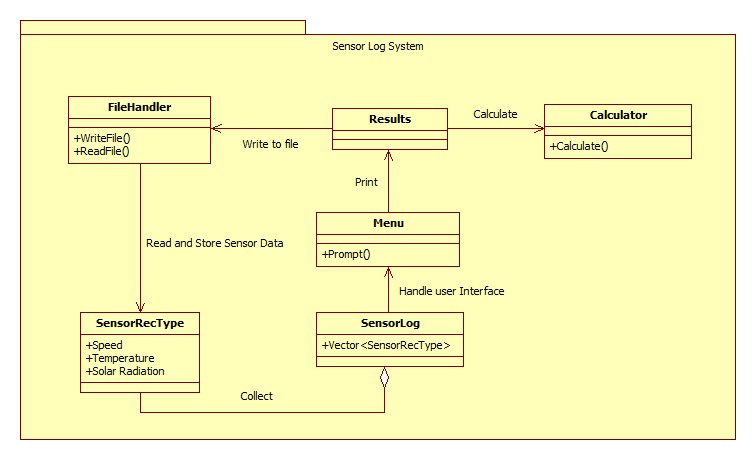
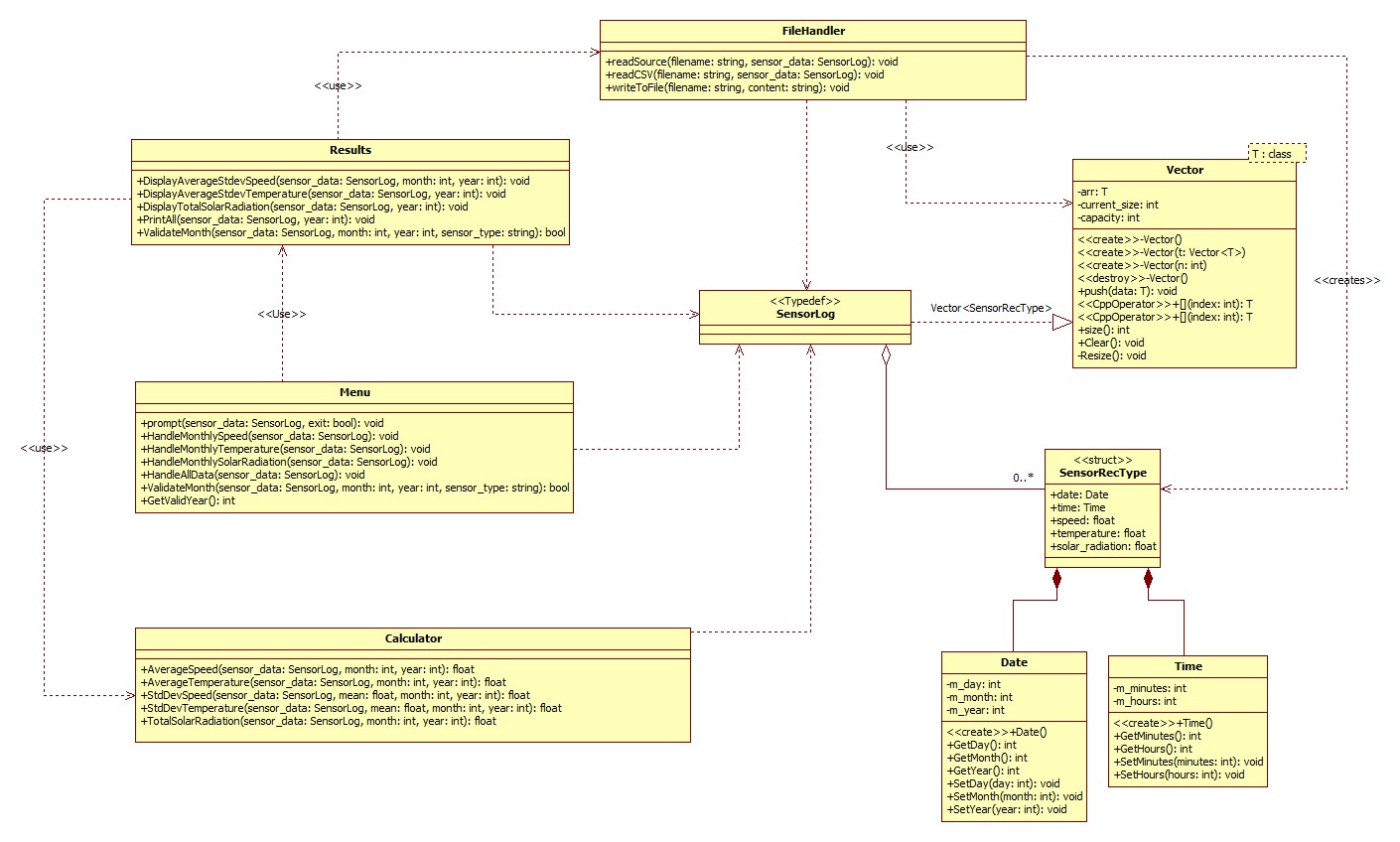
**UML**

**High Level: **

**Low Level**

****

**Data Dictionary**

**Vector.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **Vector()** | Initialize an empty vector with default capacity, and current\_size and allocates initial memory. | Constructor | public |
| **Vector(const Vector <T> t)** | Copy constructor, creates a deep copy of the vector. | Copy  Constructor | public |
| **Vector(int n)** | Creates a vector with an initial capacity defined by the int parameter. | Constructor | public |
| **~Vector()** | Destructor, destroy all memory used by the vector | Destructor | public |
| **const T operator[int index] const** | Provides const/read-only access to element specified by the integer index parameter | Template class <T> | public |
| **T operator[int index]** | Provides access to element specified by the integer index parameter and allows for modification. | Template class <T> | public |
| **push(T data)** | Add an element at the end of the Vector and calls a Resize routine if number of elements exceed capacity. | void | public |
| **Resize()** | Double the capacity of the vector when called. | void | private |
| **Clear()** | Removes all elements from the vector and resets the vector to it’s initial empty state. | void | public |
| **arr** | Array that holds the elements | Template class <T> | private |
| **current\_size** | Number of elements currently | Int | private |
| **capacity** | Total capacity of allocated memory | Int | private |

**Vector Rationale**

The **arr** private member allows the class to have an array to store elements into. The **current\_size** and **capacity**, allows the class to track if it is time to resize the array. **Clear()** lets users reset the vector, this lets us reuse the Vector object without creating a new instance of it. **Resize()** is what makes our Vector a dynamic array, when it is called, it will resize the max capacity to allow for more data to be stored. The **push(T data)** allows for insertion of data into the dynamic array, and it also checks if **Resize()** should be called. The **operator[]** allows access to elements inside our Vector, The **const operator[] () const** is the same, but only provides read-only access which allows us to pass a const version of the vector into a parameter, this is helpful to prevent accidental modifications. **~Vector** is destructor that can prevent memory leaks. **Vector(int n)** is useful if you know how much data is going to be inside the Vector, this will reduce the number of times the vector needs to resize, and theoretically improve performance. **Vector(const Vector<T> &other** is a copy constructor the deep copies, which prevents issues when one vector is modified while the reference exists. **Vector()** is the default constructor that initializes the capacity to 1 to minimize memory usage for empty vectors.

**SensorRecType.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **date** | Date of the sensor reading | Date | public |
| **time** | Time of the sensor reading | Time | public |
| **speed** | Wind Speed in m/s | float | public |
| **temperature** | Temperature in degrees Celsius | float | public |
| **solar\_radiation** | Solar Radiation measurement in W/m^2 | float | public |

**SensorRecType Rationale**

This vector struct provides the storage for individual sensor readings. It contains **date** and **time** for the Date and time the sensor was recorded. It also contains the Speed, Temperature , Solar Radiation readings which is a requirement for this program.

**Date.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **Date()** | Constructor that initializes Date with default values | int | public |
| **GetDay()** | Returns m\_day value | int | public |
| **GetMonth()** | Returns m\_month value | int | public |
| **GetYear()** | Returns m\_year value | int | public |
| **SetDay(int day)** | Sets m\_day value | void | public |
| **SetMonth(int month)** | Sets m\_month value | void | public |
| **SetYear(int year)** | Sets m\_year value | void | public |
| **m\_day** | Contains the day value | int | private |
| **m\_month** | Contains the month value | int | private |
| **m\_year** | Contains the year value | int | private |

**Date Rationale**

Is a class that contains Setters and Getters to store or retrieve Day, Month, Year values of the recorded sensor readings.

**Time.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **Time()** | Constructor that initializes Date with default values | int | public |
| **GetMinutes()** | Returns m\_minutes value | int | public |
| **GetHours()** | Returns m\_hours value | int | public |
| **SetMinutes(in)** | Returns m\_minutes value | int | public |
| **SetHours()** | Sets m\_hours value | void | public |
| **m\_minutes** | Contains the minute value | int | private |
| **m\_hours** | Contains the hour value | int | private |

**Time Rationale**

Is a class that contains Setters and Getters to store or retrieve hours and minute values of the recorded sensor readings.

**Calculator.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **AverageSpeed (**  **const SensorLog &sensor\_data,**  **const int month,**  **const int year) const;** | Calculates the average speed for the given month and year | float | Public |
| **AverageTemperature (**  **const SensorLog &sensor\_data, const int month,**  **const int year) const** | Calculates the temperature speed for the given month and year | float | Public |
| **StdDevSpeed (**  **const SensorLog &sensor\_data, float mean,**  **const int month,**  **const int year) const** | Calculates the sample standard deviation of speed for the given month and year | float | Public |
| **StdDevTemperature (**  **const SensorLog &sensor\_data, float mean,**  **const int month,**  **const int year) const** | Calculates the sample standard deviation of temperature for the given month and year | float | Public |
| **TotalSolarRadiation(**  **const SensorLog &sensor\_data, const int month,**  **const int year) const** | Calculates the Total Solar Radiation for the given month and year | float | Public |

**Calculator Rationale**

Each function of this class has different statistical operations for the calculation of each sensor reading, Speed, Temperature, Solar Radiation. These statistical calculations is a requirement.

**Menu.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **prompt(**  **const SensorLog &sensor\_data, bool &exit) const** | Displays main menu and handles user input | void | public |
| **HandleMonthlySpeed (const SensorLog &sensor\_data** | Gets and validate user input for specific month and year | void | public |
| **GetValidYear() const** | Gets and validates year input from user | int | public |

**Menu Rationale**

This class is responsible for user interaction. The **prompt()** function is responsible for displaying the Menu prompt, which handles user input and processes it into the function the user wishes to perform. The **HandleMonthlySpeed ()** is afunction is responsible for handling the user input for month, the rationale for creating a function like this is to maintain readable code. **GetValidYear()** is a function similar to **HandleMonthlySpeed()** where it prompts user and validates if the preconditions are correct before returning the year the user entered. My rationale for that function is to allow user to input a year without breaking the program, because if I use an int type to store user input, and they accidentally typed a non-digit character, the program will go into an infinite loop and break.

**Results.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **DisplayAverageStdevSpeed(**  **const SensorLog &sensor\_data, const int month,**  **const int year) const** | Calculates and displays average wind speed and standard deviation for specified month/year | void | public |
| **DisplayAverageStdevTemperature(**  **const SensorLog &sensor\_data,**  **const int year) const** | Calculates and displays average temperature and standard deviation for each month of specified year | void | public |
| **DisplayTotalSolarRadiation(**  **const SensorLog &sensor\_data,**  **const int year) const;** | Calculates and displays total solar radiation for each month of specified year | void | public |
| **PrintAll(**  **const SensorLog &sensor\_data, const int year) const** | Writes wind speed, temperature and solar radiation data to WindTempSolar.csv | void | public |
| **ValidateMonth(**  **const SensorLog &sensor\_data, const int month,**  **const int year,**  **const string sensor\_type) const** | Checks if data exists for specified month and year | bool | public |

**Results Rationale**

Results class is responsible for displaying aggregated results by calling the Calculator class to perform a calculation. Each function Except for ValidateMonth() performs functions that is part of the requirement. ValidateMonth() is a helper function that checks if data exists. Helper functions reduce code duplication and allows for a more readable code.

**FileHandler.h**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function/Member** | **Description** | **Type** | **Protection** |
| **readSource(**  **const string &filename, SensorLog &sensor\_data) const** | leads a source file containing a list of csv filenames | void | public |
| **readCSV(**  **const string &filename, SensorLog &sensor\_data) const** | Reads and parses a CSV file containing sensor measurements | void | public |
| **writeToFile(const string &filename,**  **const string &content) const** | Writes content to a specified file | void | public |

**FileHandler Rationale**

This is responsible for any file handling operations. **readSource()** reads a list of filenames which can be processed into the vector struct by calling **readCSV**, this parses and inserts the sensor data and date into the struct which will then be pushed into a vector. The **writeToFile()** allows the insertion of data to be processed into a external csv file.

**Algorithm**

FUNCTION AverageSpeed(sensor\_data, month, year) – from Calculator.h

SET sum = 0

SET data\_count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month THEN

sum = sum + record.speed

data\_count = data\_count + 1

END IF

END FOR

IF data\_count < 2 THEN

RETURN 0

ELSE

average = sum / data\_count

RETURN ROUND(10 \* average) / 10.0 // Round to 1 decimal place

END IF

END FUNCTION

FUNCTION StdDevSpeed(sensor\_data, mean, month, year) – from Calculator.h

SET sum\_square\_diff = 0

SET data\_count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month THEN

diff = record.speed - mean

sum\_square\_diff = sum\_square\_diff + (diff \* diff)

data\_count = data\_count + 1

END IF

END FOR

IF data\_count < 2 THEN

RETURN 0

ELSE

standard\_deviation = SQRT(sum\_square\_diff / (data\_count - 1))

RETURN ROUND(10 \* standard\_deviation) / 10.0 // Round to 1 decimal place

END IF

END FUNCTION

FUNCTION AverageTemperature(sensor\_data, month, year) – from Calculator.h

SET sum = 0

SET data\_count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month THEN

sum = sum + record.temperature

data\_count = data\_count + 1

END IF

END FOR

IF data\_count < 2 THEN

RETURN 0

ELSE

average = sum / data\_count

RETURN ROUND(10 \* average) / 10.0 // Round to 1 decimal place

END IF

END FUNCTION

FUNCTION StdDevTemperature(sensor\_data, mean, month, year) – from Calculator.h

SET sum\_square\_diff = 0

SET data\_count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month THEN

diff = record.temperature - mean

sum\_square\_diff = sum\_square\_diff + (diff \* diff)

data\_count = data\_count + 1

END IF

END FOR

IF data\_count < 2 THEN

RETURN 0

ELSE

standard\_deviation = SQRT(sum\_square\_diff / (data\_count - 1))

RETURN ROUND(10 \* standard\_deviation) / 10.0 // Round to 1 decimal place

END IF

END FUNCTION

FUNCTION TotalSolarRadiation(sensor\_data, month, year) – from Calculator.h

SET totalRadiation = 0

SET data\_count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month AND record.solar\_radiation >= 100 THEN

// Convert W/m² to kWh/m²: W/m² \* (10minutes / 60min/hour) / 1000W/kW

kWh\_conversion = record.solar\_radiation \* (10.0 / 60.0) / 1000.0

totalRadiation = totalRadiation + kWh\_conversion

data\_count = data\_count + 1

END IF

END FOR

IF data\_count < 2 THEN

RETURN 0

ELSE

RETURN ROUND(10 \* totalRadiation) / 10.0 // Round to 1 decimal place

END IF

END FUNCTION

FUNCTION readCSV(filename, sensor\_data) – from FileHandler.h

SET file\_path = "data/" + filename

OPEN file at file\_path for reading

WHILE not at end of file

READ line from file

// Parse date and time

CREATE string stream from line

DECLARE date and time variables

READ date and time from string stream

// Parse sensor values

DECLARE speed, temperature, solar\_radiation variables

READ speed, temperature, solar\_radiation from string stream

// Create and store sensor record

CREATE new SensorRecType record

SET record.date = date

SET record.time = time

SET record.speed = speed

SET record.temperature = temperature

SET record.solar\_radiation = solar\_radiation

ADD record to sensor\_data

END WHILE

CLOSE file

END FUNCTION

FUNCTION ValidateMonth(sensor\_data, month, year, sensor\_type) – from Results.h

SET count = 0

FOR each record IN sensor\_data

IF record.date.year EQUALS year AND record.date.month EQUALS month then

IF sensor\_type EQUALS "S" AND record.speed > 0 THEN

count = count + 1

ELSE IF sensor\_type EQUALS "T" AND record.temperature > 0 THEN

count = count + 1

ELSE IF sensor\_type EQUALS "SR" AND record.solar\_radiation > 0 THEN

count = count + 1

END IF

END IF

END FOR

IF count >= 2 THEN

RETURN true

ELSE

RETURN false

END IF

END FUNCTION

FUNCTION Resize() – from Vector.h

// Create a new array with double the capacity

CREATE new array temp with size (capacity \* 2)

// Copy all existing elements to the new array

FOR index FROM 0 TO (capacity - 1)

temp[index] = arr[index]

END FOR

// Delete the old array

DELETE arr

// Double the capacity

capacity = capacity \* 2

// Point to the new array

arr = temp

END FUNCTION