Jacques Thurling Project Code

Written 2024-23-12

```
/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: Used the main CSVReader from the original MerkelMain project given for us to use
 * and updated it using the new WeatherEntry object instead of the OrderBookEntry
*/


#include <string>
#include <vector>

#include "WeatherEntry.h"

// CSVReader class declaration
class CSVReader {
public:
  // Constructor
  CSVReader();

  // Static method to read a CSV file and return a vector of WeatherEntry
  // objects
  static std::vector<WeatherEntry> readCSV(std::string csvFile);

  // Static method to tokenize a CSV line into a vector of strings based on a
  // separator
  static std::vector<std::string> tokenise(std::string csvLine, char separator);

private:

  /**
   * ====================================================
   * Code written by Jacques Thurling
   * ====================================================
  */

  static std::vector<WeatherEntry>

  // Static method to convert a vector of strings to a vector of WeatherEntry
  // objects
  stringsToWE(std::vector<std::string> strings);

  /**
   * ====================================================
   *  End of written code section
```

```
   * ================================================
  */
};


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/


#pragma once

#include <vector>

// The Candlestick class represents a single candlestick in a candlestick chart,
// which is commonly used in financial analysis to represent the price movements
// of a security over a specific period of time, however for this use case it is
// used to determine the movement of weather temperature over time.
class Candlestick {
private:
  // Maps a Y-coordinate from an index. This is a static helper function.
  static double mapYCoordFromIndex(int index);

public:
  // Constructor to initialize a Candlestick object with the given temperatures.
  Candlestick(double openingTemp, double closingTemp, double highestTemp,
              double lowestTemp);

  // Static function to print the candlestick chart from a vector of Candlestick
  // objects.
  static void printCandleStickChart(std::vector<Candlestick> &candlesticks);

  // Member variables representing the opening, closing, highest, and lowest
  // temperatures.
  double openingTemp;
  double closingTemp;
  double highestTemp;
  double lowestTemp;
};
/**
 * ================================================
 *  End of written code section
 * ================================================
```

```
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ==================================================
 * Code written by Jacques Thurling
 * ==================================================
*/

#pragma once

#include "WeatherEntry.h"
#include <vector>

// Class responsible for rendering weather data charts
class ChartRenderer {
private:
  // Maps array index to y-coordinate in the chart
  static double mapYCoordFromIndex(double index);

public:
  // Default constructor
  ChartRenderer();

  // Prints a graphical representation of weather data
  // @param yearly_entries Vector of vectors containing WeatherEntry objects
  static void printGraph(std::vector<std::vector<WeatherEntry>> yearly_entries);

  // Calculates points in linear space between two values
  // @param y1 Starting value
  // @param y2 Ending value
  // @param i Current position between y1 and y2
  static double linearSpace(double y1, double y2, double i);
};
/**
 * ==================================================
 *  End of written code section
 * ==================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: Used the main MerkelMain class from the initial program given
```

```
 * This is then used as the baseline to update the class to allow the
 * use of the new weather statistical functions and print methods
*/

#pragma once

#include "Candlestick.h"
#include "Weather.h"
#include <vector>

#include "CSVReader.h"

/**
 * @class MerkelMain
 * @brief Main class for managing the simulation
 */
class MerkelMain {
public:
  /** Constructor for MerkelMain */
  MerkelMain();
  /** Initializes and starts the simulation */
  void init();

private:
  /** Displays the main menu options */
  void printMenu();
  /** Shows help information */
  void printHelp();


  /**
   * ================================================
   * Code written by Jacques Thurling
   * ================================================
  */

  /** Shows weather-related statistics */
  void printWeatherStats();
  /** Displays market predictions */
  void printPrediction();
  /** Advances simulation to next time period */
  void gotoNextTimeframe();
  /** Gets user input for menu selection
   * @return Selected menu option as integer */
  int getUserOption();
  /** Processes the user's menu selection
   * @param userOption The selected menu option */
  void processUserOption(int userOption);

  /** Displays filtered chart data */
  void printFilteredChart();
```

```
  /** Displays candlestick chart */
  void printCandlesticksChart();

  /** Prints individual candlestick data
   * @param candlestick The candlestick to display */
  void printCandlesticks(Candlestick &candlestick);

  /** Stores the current timestamp */
  std::string currentTime;

  /** Weather data object initialized with weather.csv file */
  Weather weather{"weather.csv"};

  /**
   * ==================================================
   *  End of written code section
   * ==================================================
  */

};


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#pragma once

#include "Candlestick.h"
#include <vector>
class Prediction {
public:
  // Constructor that initializes prediction model with historical candlestick
  // data
  Prediction(std::vector<Candlestick> candlestickData);

  // Shifts data vector back by k positions, used for creating lagged features
  // Returns: vector shifted by k positions with zeros at the start
  std::vector<double> backshift(std::vector<double> data, int k = 1);

  // Calculates the autoregressive (AR) component of the time series
  // Returns: AR component of the time series
```

```
  std::vector<double> arComponent(std::vector<double> data);

  // Calculates the seasonal autoregressive component of the time series
  // Returns: Seasonal AR component of the time series
  std::vector<double> seasonalArComponent(std::vector<double> data);

  // Makes future predictions for specified number of steps
  // Returns: Vector of predicted values
  std::vector<double> predict(int steps = 1);

  // Trains the model parameters using gradient descent
  // epochs: Number of training iterations
  void fit(int epochs = 100);

  // Time series data
  std::vector<double> data;
  // AR coefficient for regular component
  double phi = 0.7;
  // AR coefficient for seasonal component
  double PHI = 0.8;
  // Learning rate for gradient descent optimization
  double learningRate = 0.01;
};
/**
 * ================================================
 *  End of written code section
 * ================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#pragma once

#include "WeatherEntry.h"
#include <string>
#include <variant>
#include <vector>

// Enumeration for weather data filtering options
enum WeatherFilterOptions { yearly, monthly };
```

```
class Weather {
public:
  // Constructor that takes a filename containing weather data
  Weather(std::string filename);

  /** Gets the weather entries for a region based on the filters
   * Returns either a vector of entries or a vector of vector of entries
   * depending on the timeframe selected
   */
  std::variant<std::vector<WeatherEntry>,
               std::vector<std::vector<WeatherEntry>>>
  getWeatherEntries(WeatherEntryType region, std::string timestamp,
                    WeatherFilterOptions timeframe = yearly);

  // Returns the earliest timestamp in the dataset
  std::string getEarliestTime();

  // Advances to the next time period from the current timestamp
  std::string goToNextTimeFrame(std::string currentTime);

  // Static utility functions for temperature calculations
  static double getHighestTemp(std::vector<WeatherEntry> &currentTimeEntries);
  static double getLowestTemp(std::vector<WeatherEntry> &currentTimeEntries);
  static double getClosingTemp(std::vector<WeatherEntry> &currentTimeEntries);
  static double getOpeningTemp(std::vector<WeatherEntry> &previousTimeEntries);

private:
  std::vector<WeatherEntry> entryPoints;
};
/**
 * ================================================
 *  End of written code section
 * ================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ========================================
 * Code written by Jacques Thurling
 * ========================================
*/

#pragma once
```

```
#include <map>
#include <string>

// Enumeration for European country/region codes
enum class WeatherEntryType {
  AT,
  BE,
  BG,
  CH,
  CZ,
  DE,
  DK,
  EE,
  ES,
  FI,
  FR,
  GB,
  GR,
  HR,
  IE,
  IT,
  LT,
  LU,
  LV,
  NL,
  NO,
  PL,
  PT,
  RO,
  SE,
  SI,
  SK
};

// Class representing a single weather data entry
class WeatherEntry {
public:
  // Constructor to initialize a weather entry with temperature, timeframe and
  // region
  WeatherEntry(double temp, std::string timeframe, WeatherEntryType region);

  // Maps numeric index to corresponding region enum
  static WeatherEntryType mapFromTokenToRegion(int index);

  // Maps string input to corresponding region enum
  static WeatherEntryType mapFromInputToRegion(std::string input);

  double temp;
  std::string timeframe;
  WeatherEntryType region;
```

```
  // Static map to convert string representations to region enums
  static const std::map<std::string, WeatherEntryType> weatherRegionMap;
};
/**
 * ================================================
 *  End of written code section
 * ================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: Used the main CSVReader from the original MerkelMain project given for us to use
 * and updated it using the new WeatherEntry object instead of the OrderBookEntry
*/

#include "CSVReader.h"
#include "WeatherEntry.h"
#include <exception>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

// Constructor for CSVReader class
CSVReader::CSVReader() {}

// Method to read a CSV file and return a vector of WeatherEntry objects
std::vector<WeatherEntry> CSVReader::readCSV(std::string csvFilename) {
  // Vector to store WeatherEntry objects
  std::vector<WeatherEntry> entries;
  // Open CSV file
  std::ifstream csvFile{csvFilename};
  // String to store each line of the CSV file
  std::string line;
  // Log Message
  std::cout << "Loading file data" << std::endl;

  // Check if the file is open
  if (csvFile.is_open()) {
    // Read each line of the file
    while (std::getline(csvFile, line)) {
      try {
        // Tokenise the line and convert to WeatherEntry objects
        std::vector<WeatherEntry> woe = stringsToWE(tokenise(line, ','));
        for (const WeatherEntry &entry : woe) {
          // Add each WeatherEntry to the entries vector
          entries.push_back(entry);
        }
      } catch (const std::exception &e) {
```

```
        // Log Message for bad data
        std::cout << "CSVReader::readCSV bad data" << std::endl;
      }
    } // end of while
  }

  std::cout << "CSVReader::readCSV read " << entries.size() << " entries"
            << std::endl;
  // Return the vector of WeatherEntry objects
  return entries;
}

// Method to tokenise a CSV line into a vector of string based on a separator
std::vector<std::string> CSVReader::tokenise(std::string csvLine,
                                             char separator) {
  // Vector to store tokens
  std::vector<std::string> tokens;
  // Variables to store start and end positions of tokens
  signed int start, end;
  // String to store each token
  std::string token;
  // Fine the first non-seporator character
  start = csvLine.find_first_not_of(separator, 0);
  do {
    // Find the next separator char
    end = csvLine.find_first_of(separator, start);
    // Break if end of line or no more
    if (start == csvLine.length() || start == end)
      break;
    // Extract the token
    if (end >= 0)
      token = csvLine.substr(start, end - start);
    // Extract the last token
    else
      token = csvLine.substr(start, csvLine.length() - start);
    // Add the token to the vector
    tokens.push_back(token);
    // Update the start position
    start = end + 1;
  } while (end > 0);

  // Return the vector of tokens
  return tokens;
}

/**
* ============================================
* Code written by Jacques Thurling
* ============================================
*/
```

```
// Method to convert a vector of strings to a vector of WeatherEntry objects
std::vector<WeatherEntry>
CSVReader::stringsToWE(std::vector<std::string> strings) {
  // Vector to store WeatherEntry objects
  std::vector<WeatherEntry> entries;

  for (int i = 1; i < strings.size(); i++) {
    try {
      // Map token to region
      WeatherEntryType region = WeatherEntry::mapFromTokenToRegion(i);
      // Convert string to double for temperature
      double temperature = std::stod(strings[i]);
      // Get the timeframe
      std::string timeframe = strings[0];

      // Create Weather Entry object
      WeatherEntry entry{temperature, timeframe, region};
      // Add the WeatherEntry object to the vector
      entries.push_back(entry);
    } catch (const std::exception &e) {
      // Skip bat data
      continue;
    }
  }

  // Return the vector of WeatherEntry objects
  return entries;
}
/**
 * ====================================================
 *  End of written code section
 * ====================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ==========================================
 * Code written by Jacques Thurling
 * ==========================================
*/

#include "Candlestick.h"
#include <iostream>

// Constructor initializes a candlestick with opening, closing, highest, and
```

```
// lowest temperatures
Candlestick::Candlestick(double openingTemp, double closingTemp,
                         double highestTemp, double lowestTemp)
    : openingTemp(openingTemp), closingTemp(closingTemp),
      highestTemp(highestTemp), lowestTemp(lowestTemp) {}

// Prints an ASCII art candlestick chart using the provided vector of
// candlesticks
void Candlestick::printCandleStickChart(
    std::vector<Candlestick> &candlesticks) {
  // Chart height in characters
  unsigned int height = 25;
  // Chart width in characters
  unsigned int width = 90;

  // Initial temperature values for y-axis
  int temp = 50;

  std::cout << std::endl;

  // Temperature tolerance for drawing candlesticks
  int tolerance = 3;

  // Iterate through each row of the chart
  for (int i = 0; i < height; i++) {
    // Map row index to temperature values
    temp = mapYCoordFromIndex(i);

    // Iterate though each column
    for (int j = 0; j < width; j++) {
      if (j == 0 && i == height - 1) {
        // Bottom left corner
        std::cout << "    └";
      } else if (j == 0) {
        if ((i % 2) == 0) {
          // Prepare the y-axis
          if (temp < -10) {
            std::cout << temp << " ┤";
          } else if (temp < 10 && temp > -1) {
            std::cout << "  " << temp << " ┤";
          } else if (temp < -1 && temp > -10) {
            std::cout << " " << temp << " ┤";
          } else {
            std::cout << " " << temp << " ┤";
          }
        } else {
          // Y-axis line
          std::cout << "    │";
        }
      } else if (i == height - 1) {
        // Populate the x-axis
```

```
        if (j % 10 == 0) {
          std::cout << "┬";
        } else {
          std::cout << "─";
        }
      } else {
        if (j % 10 == 0) {
          Candlestick candlestick = candlesticks[(j / 10) - 1];

          // Draw candlestick components based on temperature relationships
          if (candlestick.highestTemp == candlestick.lowestTemp) {
            // No variation in temperatures
            std::cout << " ";
          } else if (candlestick.openingTemp < candlestick.closingTemp) {
            // Bullish candlestick (closing > opening)
            if (temp < candlestick.highestTemp &&
                temp > candlestick.closingTemp + tolerance) {
              std::cout << "│";
            } else if (temp < candlestick.closingTemp + tolerance &&
                       temp > candlestick.openingTemp - tolerance) {
              if (candlestick.closingTemp - candlestick.openingTemp < 1) {
                std::cout << "┼";
              } else {
                std::cout << "█";
              }
            } else if (temp < candlestick.openingTemp &&
                       temp > candlestick.lowestTemp) {
              std::cout << "│";
            } else {
              std::cout << " ";
            }
          } else if (candlestick.closingTemp < candlestick.openingTemp) {
            // Bearish candlestick (closing < opening)
            if (temp < candlestick.highestTemp &&
                temp > candlestick.openingTemp) {
              std::cout << "│";
            } else if (temp < candlestick.openingTemp &&
                       temp > candlestick.closingTemp) {
              std::cout << "█";
            } else if (temp < candlestick.closingTemp &&
                       temp > candlestick.lowestTemp) {
              std::cout << "│";
            } else {
              std::cout << " ";
            }
          } else {
            std::cout << " ";
          }

        } else {
          std::cout << " ";
```

```
        }
      }
    }
    std::cout << std::endl;
  }
}

// Maps a y-coordinate index to a temperature value
// This creates the temperature scale on the y-axis
double Candlestick::mapYCoordFromIndex(int index) {
  return 40 - ((80 * index) / 25);
}
/**
 * ================================================
 *  End of written code section
 * ================================================
*/
/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#include "ChartRenderer.h"
#include "Weather.h"
#include "WeatherEntry.h"
#include <cmath>
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
#include <strings.h>
#include <vector>

// Default constructor
ChartRenderer::ChartRenderer() {}

// Prints a line graph visualization of weather data
void ChartRenderer::printGraph(
    std::vector<std::vector<WeatherEntry>> yearly_entries) {
  std::vector<WeatherEntry> data_to_render;

  // Use more memory for rendering - keep data separate
  for (std::vector<WeatherEntry> entries : yearly_entries) {
    std::string time = entries.begin()->timeframe;
```

```
  // Process and prepare data for rendering
  // Extract mean temperatures for each timeframe
  double mean_temp = Weather::getClosingTemp(entries);

  // Create new entry with processed data
  WeatherEntry new_entry{mean_temp, time, entries.begin()->region};

  data_to_render.push_back(new_entry);
}

// Set chart dimensions
unsigned int height = 25;
unsigned int width = 90;

std::cout << data_to_render.size() << std::endl;

// Debug output - print data size and temperatures
for (int i = 0; i < data_to_render.size(); i++) {
  std::cout << data_to_render[i].temp << std::endl;
}

std::cout << std::endl;

double temp = 50;

// Iterate through each row of the chart
for (int i = 0; i < height; i++) {

  // Calculate temperature for current row
  temp = round(mapYCoordFromIndex((double)i) * 10.0) / 10.0;
  std::cout << std::fixed << std::setprecision(1);

  // Iterate through each column
  for (int j = 0; j < width; j++) {
    if (j == 0 && i == height - 1) {
      std::cout << "      └";
    } else if (j == 0) {
      if ((i % 2) == 0) {
        // Prepare the y-axis
        if (temp < -10) {
          std::cout << temp << " ┤";
        } else if (temp < 10 && temp > -1) {
          std::cout << "  " << temp << " ┤";
        } else if (temp < -1 && temp > -10) {
          std::cout << " " << temp << " ┤";
        } else {
          std::cout << " " << temp << " ┤";
        }
      } else {
        std::cout << "      │";
```

```
        }
      } else if (i == height - 1) {
        // Populate the x-axis
        // Draw x-axis with tick marks
        if (j % 10 == 0) {
          std::cout << "┬";
        } else {
          std::cout << "─";
        }
      } else {
        // Plot data points and connecting lines
        WeatherEntry start = data_to_render[floor(j / 10)];
        WeatherEntry end = data_to_render[floor(j / 10) + 1];

        if (j % 10 == 0) {
          // Draw data points
          if (start.temp > mapYCoordFromIndex(i + 1) && start.temp < temp) {
            std::cout << "*";
          } else {
            std::cout << " ";
          }
        } else {
          // Draw connecting lines between data points
          double linSpace = linearSpace(start.temp, end.temp, j % 10);

          if (linSpace > mapYCoordFromIndex(i + 1) && linSpace < temp) {
            std::cout << "*";
          } else {
            std::cout << " ";
          }
        }
      }
    }
    std::cout << std::endl;
  }
}

// Maps y-coordinate index to temperature value
// Creates the temperature scale on the y-axis
double ChartRenderer::mapYCoordFromIndex(double index) {
  return -0.56 * index + 13.0;
}

// Calculates intermediate points for line drawing between two temperatures
// Uses linear interpolation
double ChartRenderer::linearSpace(double y1, double y2, double i) {
  return y1 + (i / (10 - 1)) * (y2 - y1);
}
/**
 * ================================================
 *  End of written code section
```

```
 * ================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: Used the main MerkelMain class from the initial program given
 * This is then used as the baseline to update the class to allow the
 * use of the new weather statistical functions and print methods
*/

#include "MerkelMain.h"
#include "Candlestick.h"
#include "ChartRenderer.h"
#include "Prediction.h"
#include "Weather.h"
#include "WeatherEntry.h"
#include <exception>
#include <iomanip>
#include <ios>
#include <iostream>
#include <ostream>
#include <string>
#include <vector>

/**
 * MerkelMain.cpp
 * Main application class for a weather analysis and prediction system.
 * Handles user interaction and data visualization for weather patterns across
 * Europe.
 */
MerkelMain::MerkelMain() {}

/**
 * Main program loop that initializes the application and processes user
 * commands
 */
void MerkelMain::init() {
  // Variable to store user's menu selection
  int input;

  // Set the initial time to the earliest available time in weather data
  currentTime = weather.getEarliestTime();

  // Main program loop
  while (true) {
    // Display the menu options to the user
    printMenu();
    // Get the user's selected option
    input = getUserOption();
```

```
    // Process the user's selection
    processUserOption(input);
  }
}

/**
 * Displays the main menu options to the user
 */
void MerkelMain::printMenu() {
  // 1 print help
  std::cout << "1: Print help " << std::endl;
  // 2 print Weather for region
  std::cout << "2: Print Weather data for region" << std::endl;
  // 3 print Candlestick Chart for region and year
  std::cout << "3: Print Candlestick chart for region and year" << std::endl;
  // 4 print Graph Chart for date range and region
  std::cout << "4: Print Graph for date range and region" << std::endl;
  // 5 get prediction and then Print the prediction in a graph
  std::cout << "5: Predict future temperatures" << std::endl;
  // 6 continue
  std::cout << "6: Continue " << std::endl;

  std::cout << "============== " << std::endl;
}

/** Displays help information about the program's purpose */
void MerkelMain::printHelp() {
  std::cout
      << "Help - Analyse weather patterns in different regions of Europe. "
      << std::endl;
}

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

/**
 * Prints weather statistics for a specified region showing open, high, low and
 * closing temperatures Format: region code (e.g. FR for France)
 */
void MerkelMain::printWeatherStats() {
  // Prompt user for region input
  std::cout << "Enter the region (FR): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input into tokens (though only one token expected)
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');
```

```
  // Container for weather entries
  std::vector<WeatherEntry> temp;

  try {
    // Convert input string to region enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    // Start from 1980 as base year
    int year = 1980;

    // Print header for data table
    std::cout << "Date                    Open       High      Low      Closing"
              << std::endl;
    // Set decimal precision for temperature values
    std::cout << std::fixed << std::setprecision(3);

    // Process data year by year until no more data is available
    do {
      // Get weather entries for current year and region
      temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));

      // Exit loop if no data found for current year
      if (temp.size() == 0) {
        break;
      }

      // Calculate temperature statistics for the year
      double lowestTemp = Weather::getLowestTemp(temp);
      double highestTemp = Weather::getHighestTemp(temp);
      double closingTemp = Weather::getClosingTemp(temp);
      double openingTemp = Weather::getOpeningTemp(temp);

      // Print statistics for current year
      std::cout << temp.begin()->timeframe << "   " << openingTemp << "      "
                << highestTemp << "   " << lowestTemp << "   " << closingTemp
                << std::endl;

      // Move to next year
      year++;
    } while (temp.size() > 0);

  } catch (const std::exception &e) {
    // Handle any errors during data processing
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
  }
}

/**
```

```
 * Generates and displays a candlestick chart for temperature data
 * Format: region,year (e.g. FR,1990)
 */
void MerkelMain::printCandlesticksChart() {
  // Prompt user for input in format: region,year
  std::cout << "Enter the region and year (FR,1990): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input string into tokens using comma as delimiter
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  // Container for monthly temperature entries
  std::vector<std::vector<WeatherEntry>> monthly_entries;

  try {
    // Convert region string to enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    // Retrieve monthly weather data for specified region and year
    monthly_entries = std::get<std::vector<std::vector<WeatherEntry>>>(
        weather.getWeatherEntries(region, tokens[1],
                                  WeatherFilterOptions::monthly));

  } catch (const std::exception &e) {
    // Handle any errors during data retrieval
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
    throw e;
  }

  // Container for candlestick data
  std::vector<Candlestick> candlesticks;

  // Process each month's data into candlestick format
  for (int i = 0; i < monthly_entries.size(); i++) {
    // Calculate temperature metrics for the month
    double lowestTemp = Weather::getLowestTemp(monthly_entries[i]);
    double highestTemp = Weather::getHighestTemp(monthly_entries[i]);
    double closingTemp = monthly_entries[i].end()->temp;
    double openingTemp = monthly_entries[i].begin()->temp;

    // Create candlestick object with temperature data
    Candlestick candlestick{openingTemp, closingTemp, highestTemp, lowestTemp};

    // Add to collection
    candlesticks.push_back(candlestick);
  }

  // Display the candlestick chart
```

```
  Candlestick::printCandleStickChart(candlesticks);
}

/**
 * Makes temperature predictions using historical data and displays forecast
 * Uses a prediction model to estimate next 5 temperature values
 */
void MerkelMain::printPrediction() {
  // Prompt user for region input
  std::cout << "Enter the region (FR): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Parse input into tokens
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  // Initialize containers for weather data and candlestick patterns
  std::vector<WeatherEntry> temp;
  std::vector<Candlestick> data;

  // Start year for historical data collection
  int year = 1980;

  try {
    // Convert input string to region enum
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    std::cout << "Making Predictions" << std::endl;

    // Set decimal precision for temperature output
    std::cout << std::fixed << std::setprecision(3);

    // Collect historical temperature data year by year
    do {
      // Get weather entries for current year and region
      temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));

      // Break if no data available for current year
      if (temp.size() == 0) {
        break;
      }

      // Calculate temperature metrics for the year
      double lowestTemp = Weather::getLowestTemp(temp);
      double highestTemp = Weather::getHighestTemp(temp);
      double closingTemp = Weather::getClosingTemp(temp);
      double openingTemp = Weather::getOpeningTemp(temp);

      // Create candlestick object from temperature data
      Candlestick candle{openingTemp, closingTemp, highestTemp, lowestTemp};
```

```
      // Add to historical data collection
      data.push_back(candle);

      year++;
    } while (temp.size() > 0);

  } catch (const std::exception &e) {
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
  }

  // Create and train prediction model using historical data
  Prediction model{data};
  model.fit();

  // Generate 5-year temperature forecast
  std::vector<double> forecast = model.predict(5);
  std::cout << "Next 5 temps: " << std::endl;

  // Prepare data structure for visualization
  std::vector<std::vector<WeatherEntry>> chart;

  // Convert forecast data to WeatherEntry format
  for (int i = 0; i < forecast.size(); i++) {
    std::vector<WeatherEntry> predictions;

    // Create WeatherEntry for each predicted temperature
    WeatherEntry prediction{forecast[i],
                            std::to_string(year) + "-01-01T00:00:00Z",
                            WeatherEntry::mapFromInputToRegion(tokens[0])};

    predictions.push_back(prediction);
    chart.push_back(predictions);
  }

  // Display forecast as graph
  ChartRenderer::printGraph(chart);
}

/**
 * Advances the current timeframe to the next period
 */
void MerkelMain::gotoNextTimeframe() {
  // Inform user that timeframe is advancing
  std::cout << "Going to next time frame. " << std::endl;
  // Split current timestamp into tokens using '-' as delimiter
  std::vector<std::string> tokens = CSVReader::tokenise(currentTime, '-');

  // Update current time to next available timeframe in weather data
```

```
  // tokens[0] contains the current timestamp
  currentTime = weather.goToNextTimeFrame(tokens[0]);
}

/**
 * Gets and validates user input for menu options (1-6)
 * @return int Selected menu option
 */
int MerkelMain::getUserOption() {
  // Initialize user's selection
  int userOption = 0;
  std::string line;

  // Prompt user for input
  std::cout << "Type in 1-6" << std::endl;
  // Get entire line of input from user
  std::getline(std::cin, line);
  try {
    // Convert string input to integer
    userOption = std::stoi(line);
  } catch (const std::exception &e) {
    // If conversion fails, userOption remains 0
    // This handles invalid inputs like letters or special characters
  }

  // Echo user's selection
  std::cout << "You chose: " << userOption << std::endl;
  return userOption;
}

/**
 * Displays temperature data as a graph for a specified date range
 * Format: region,start_year,end_year (e.g. FR,1990,2001)
 */
void MerkelMain::printFilteredChart() {
  // Prompt user for input
  std::cout << "Enter the region, start year and end year e.g. FR,1990,2001: "
            << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input string into tokens using comma as delimiter
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  try {
    // Calculate the year range
    int year_difference = std::stoi(tokens[2]) - std::stoi(tokens[1]);

    // Convert region string to enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);
```

```
    // Validate year range is positive
    if (year_difference < 1) {
      std::cout << "Please choose valid years" << std::endl;
      return;
    }

    // Container for weather data across multiple years
    std::vector<std::vector<WeatherEntry>> weatherDataYearlyEntries;

    // Collect weather data for each year in the range
    for (int i = 0; i <= year_difference; i++) {
      int year = std::stoi(tokens[1]) + i;

      // Get weather entries for current year and region
      std::vector<WeatherEntry> temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));
      weatherDataYearlyEntries.push_back(temp);
    }

    // Render the graph using collected data
    ChartRenderer::printGraph(weatherDataYearlyEntries);

  } catch (const std::exception &e) {
    // Handle any errors during processing
    std::cout << "printFilteredChart - there has been an error" << std::endl;
  }
}

/**
 * Processes the user's menu selection and calls appropriate functions
 * @param userOption Selected menu option (1-6)
 */
void MerkelMain::processUserOption(int userOption) {
  if (userOption == 0) // bad input
  {
    std::cout << "Invalid choice. Choose 1-6" << std::endl;
  }

  // Display help menu and available commands
  if (userOption == 1) {
    printHelp();
  }

  // Show statistical analysis of weather data
  if (userOption == 2) {
    printWeatherStats();
  }

  // Display temperature data as candlestick chart
  if (userOption == 3) {
    printCandlesticksChart();
```

```
  }

  // Show filtered temperature data visualization
  if (userOption == 4) {
    printFilteredChart();
  }

  // Calculate and display weather predictions
  if (userOption == 5) {
    printPrediction();
  }

  // Advance to next time period in the dataset
  if (userOption == 6) {
    gotoNextTimeframe();
  }
}
/**
 * ====================================================
 *  End of written code section
 * ====================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#include "Prediction.h"
#include "Candlestick.h"
#include <vector>

/**
 * Constructor: Initializes prediction model with historical candlestick data
 * @param candlestickData Vector of historical temperature data points
 */
Prediction::Prediction(std::vector<Candlestick> candlestickData) {
  for (const Candlestick& entry : candlestickData) {
    data.push_back(entry.closingTemp);
  }
}

/**
```

```
 * Shifts data backwards by k positions, filling with zeros
 * @param clonedData Input data vector to shift
 * @param k Number of positions to shift
 * @return Vector with shifted data
 */
std::vector<double> Prediction::backshift(std::vector<double> clonedData, int k) {

  std::vector<double> result;

  if (k < clonedData.size()) {
    // Add zeroes to the beginning
    result.insert(result.begin(), k, 0.0);

    // Add data[:-k]
    result.insert(result.end(), clonedData.begin(), clonedData.end() - k);
  } else {
    // If K >= data.size(), return a vector of zeroes
    result.resize(clonedData.size(), 0.0);
  }

  return result;
}

/**
 * Calculates the autoregressive (AR) component of the prediction
 * @param clonedData Input data vector
 * @return Vector containing AR components
 */
std::vector<double> Prediction::arComponent(std::vector<double> clonedData) {
  std::vector<double> shifted = backshift(clonedData);

  std::vector<double> result;

  for (int i = 0; i < clonedData.size(); i++) {
    result.push_back(shifted[i] * phi);
  }

  return result;
}

/**
 * Calculates the seasonal autoregressive component of the prediction
 * @param clonedData Input data vector
 * @return Vector containing seasonal AR components
 */
std::vector<double> Prediction::seasonalArComponent(std::vector<double> clonedData) {
  std::vector<double> shifted = backshift(clonedData, 1);

  std::vector<double> result;

  for (int i = 0; i < clonedData.size(); i++) {
```

```
    result.push_back(shifted[i] * PHI);
  }

  return result;
}

/**
 * Generates future predictions based on historical data
 * @param steps Number of future steps to predict
 * @return Vector of predicted values
 */
std::vector<double> Prediction::predict(int steps){
  std::vector<double> predictions;
  std::vector<double> working_data = data;

  // Loop without index needed
  // Generate predictions for specified number of steps
  for (int _ = 0; _ < steps; _++) {
    double ar = arComponent(working_data).back();
    double seasonalAr = seasonalArComponent(working_data).back();

    double nextValue = ar + seasonalAr;
    predictions.push_back(nextValue);
    working_data.push_back(nextValue);
  }

  return predictions;
}

/**
 * Trains the model by adjusting parameters to minimize prediction error
 * @param epochs Number of training iterations
 */
void Prediction::fit(int epochs) {
  for (int _ = 0; _ < epochs; _++) {
    std::vector<double> prediction = predict(1);

    double error = prediction[0] - data[data.size() - 1];

    phi -= learningRate * error * data[data.size() - 2];
    PHI -= learningRate * error * data[data.size() - 1];
  }
}
/**
 * ================================================
 *  End of written code section
 * ================================================
*/


/**
```

```
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#include "Weather.h"
#include "CSVReader.h"
#include "WeatherEntry.h"

#include <algorithm>
#include <string>
#include <variant>
#include <vector>

/**
 * Trains the model by adjusting parameters to minimize prediction error
 * @param epochs Number of training iterations
 */
Weather::Weather(std::string filename) {
  entryPoints = CSVReader::readCSV(filename);
}

/**
 * Retrieves weather entries based on specified filters
 * @param region The geographic region to filter by
 * @param timestamp The time period to filter by
 * @param timeframe The granularity of data (yearly/monthly)
 * @return Either a vector of entries or a vector of monthly entry vectors
 */
std::variant<std::vector<WeatherEntry>, std::vector<std::vector<WeatherEntry>>>
Weather::getWeatherEntries(WeatherEntryType region, std::string timestamp,
                           WeatherFilterOptions timeframe) {
  // Filter entries by region and timestamp
  std::vector<WeatherEntry> filtered_entries;

  for (const WeatherEntry &entry : entryPoints) {
    if (entry.region == region &&
        entry.timeframe.find(timestamp) != std::string::npos) {
      filtered_entries.push_back(entry);
    }
  }

  std::vector<std::vector<WeatherEntry>> monthly_entries;

  // Only done when montly time period is selected - get the montly entries for
```

```
  // a specific year Handle monthly timeframe option
  if (timeframe == WeatherFilterOptions::monthly) {
    int month = 1;

    std::vector<WeatherEntry> monthly;

    // Collect entries for each month
    while (month <= 12) {
      // Format month string with leading zero if needed
      std::string month_string;
      if (month < 10) {
        month_string = "0" + std::to_string(month);
      } else {
        month_string = std::to_string(month);
      }

      std::string time = timestamp + "-" + month_string;

      // Filter entries for current month
      for (const WeatherEntry &entry : filtered_entries) {
        if (entry.region == region &&
            entry.timeframe.find(time) != std::string::npos) {
          monthly.push_back(entry);
        }
      }

      monthly_entries.push_back(monthly);

      month++;
    }

    return monthly_entries;
  }

  return filtered_entries;
}

/**
 * Gets the earliest timestamp in the dataset
 * @return String representing the earliest timestamp
 */
std::string Weather::getEarliestTime() {
  return entryPoints.begin()->timeframe;
}

/**
 * Advances to the next time period in the dataset
 * @param currentTime Current timestamp
 * @return Next available timestamp, or earliest if at end of dataset
 */
std::string Weather::goToNextTimeFrame(std::string currentTime) {
```

```
  int currentYear = std::stoi(currentTime);

  currentYear++;

  // Find next entry with incremented year
  auto t =
      std::find_if(entryPoints.begin(), entryPoints.end(),
                   [&currentYear](const WeatherEntry &entry) {
                     return entry.timeframe.find(std::to_string(currentYear)) !=
                            std::string::npos;
                   });

  if (t != entryPoints.end()) {
    return t->timeframe;
  }

  return getEarliestTime();
}

/**
 * Finds the highest temperature in a set of entries
 * @param currentTimeEntries Vector of weather entries to search
 * @return Highest temperature found
 */
double Weather::getHighestTemp(std::vector<WeatherEntry> &currentTimeEntries) {
  double highestPrice = 0.0;

  for (const WeatherEntry &entry : currentTimeEntries) {
    if (entry.temp > highestPrice) {
      highestPrice = entry.temp;
    }
  }

  return highestPrice;
}

/**
 * Calculates average opening temperature for a set of entries
 * @param previousTimeEntries Vector of weather entries
 * @return Average opening temperature
 */
double Weather::getOpeningTemp(std::vector<WeatherEntry> &previousTimeEntries) {
  double openingTemp = 0.0;

  for (const WeatherEntry &entry : previousTimeEntries) {
    openingTemp += entry.temp;
  }

  return openingTemp / previousTimeEntries.size();
}
```

```
/**
 * Calculates average closing temperature for a set of entries
 * @param currentTimeEntries Vector of weather entries
 * @return Average closing temperature
 */
double Weather::getClosingTemp(std::vector<WeatherEntry> &currentTimeEntries) {
  double closingTemp = 0.0;

  for (const WeatherEntry &entry : currentTimeEntries) {
    closingTemp += entry.temp;
  }

  return closingTemp / currentTimeEntries.size();
}

/**
 * Finds the lowest temperature in a set of entries
 * @param currentTimeEntries Vector of weather entries to search
 * @return Lowest temperature found
 */
double Weather::getLowestTemp(std::vector<WeatherEntry> &currentTimeEntries) {
  double lowestTemp = 0.0;

  for (const WeatherEntry &entry : currentTimeEntries) {
    if (entry.temp < lowestTemp) {
      lowestTemp = entry.temp;
    }
  }

  return lowestTemp;
}
/**
 * ================================================
 *  End of written code section
 * ================================================
*/


/**
 * Author: Jacques Thurling
 * Date: 2024-23-12
 * Notes: All code written without assistance
*/

/**
 * ================================================
 * Code written by Jacques Thurling
 * ================================================
*/

#include "WeatherEntry.h"
```

```
#include <algorithm>
#include <cctype>
#include <string>

/**
 * Static map that associates country codes with WeatherEntryType enum values
 * Contains mappings for European countries (e.g., "AT" -> Austria, "BE" ->
 * Belgium)
 */
const std::map<std::string, WeatherEntryType> WeatherEntry::weatherRegionMap = {
    {"AT", WeatherEntryType::AT}, {"BE", WeatherEntryType::BE},
    {"BG", WeatherEntryType::BG}, {"CH", WeatherEntryType::CH},
    {"CZ", WeatherEntryType::CZ}, {"DE", WeatherEntryType::DE},
    {"DK", WeatherEntryType::DK}, {"EE", WeatherEntryType::EE},
    {"ES", WeatherEntryType::ES}, {"FI", WeatherEntryType::FI},
    {"FR", WeatherEntryType::FR}, {"GB", WeatherEntryType::GB},
    {"GR", WeatherEntryType::GR}, {"HR", WeatherEntryType::HR},
    {"IE", WeatherEntryType::IE}, {"IT", WeatherEntryType::IT},
    {"LT", WeatherEntryType::LT}, {"LU", WeatherEntryType::LU},
    {"LV", WeatherEntryType::LV}, {"NL", WeatherEntryType::NL},
    {"NO", WeatherEntryType::NO}, {"PL", WeatherEntryType::PL},
    {"PT", WeatherEntryType::PT}, {"RO", WeatherEntryType::RO},
    {"SE", WeatherEntryType::SE}, {"SI", WeatherEntryType::SI},
    {"SK", WeatherEntryType::SK}};

/**
 * Constructor: Creates a new WeatherEntry instance
 * @param temp Temperature value
 * @param timeframe Time period of the measurement
 * @param region Geographic region identifier
 */
WeatherEntry::WeatherEntry(double temp, std::string timeframe,
                           WeatherEntryType region)
    : temp(temp), timeframe(timeframe), region(region) {}

/**
 * Maps a numeric index to corresponding WeatherEntryType
 * @param index Numeric identifier for region (1-27)
 * @return Corresponding WeatherEntryType enum value
 */
WeatherEntryType WeatherEntry::mapFromTokenToRegion(int index) {
  switch (index) {
  case 1:
    return WeatherEntryType::AT;
    break;
  case 2:
    return WeatherEntryType::BE;
    break;
  case 3:
    return WeatherEntryType::BG;
    break;
```

```
  case 4:
    return WeatherEntryType::CH;
    break;
  case 5:
    return WeatherEntryType::CZ;
    break;
  case 6:
    return WeatherEntryType::DE;
    break;
  case 7:
    return WeatherEntryType::DK;
    break;
  case 8:
    return WeatherEntryType::EE;
    break;
  case 9:
    return WeatherEntryType::ES;
    break;
  case 10:
    return WeatherEntryType::FI;
    break;
  case 11:
    return WeatherEntryType::FR;
    break;
  case 12:
    return WeatherEntryType::GB;
    break;
  case 13:
    return WeatherEntryType::GR;
    break;
  case 14:
    return WeatherEntryType::HR;
    break;
  case 15:
    return WeatherEntryType::IE;
    break;
  case 16:
    return WeatherEntryType::IT;
    break;
  case 17:
    return WeatherEntryType::LT;
    break;
  case 18:
    return WeatherEntryType::LU;
    break;
  case 19:
    return WeatherEntryType::LV;
    break;
  case 20:
    return WeatherEntryType::NL;
    break;
```

```
  case 21:
    return WeatherEntryType::NO;
    break;
  case 22:
    return WeatherEntryType::PL;
    break;
  case 23:
    return WeatherEntryType::PT;
    break;
  case 24:
    return WeatherEntryType::RO;
    break;
  case 25:
    return WeatherEntryType::SE;
    break;
  case 26:
    return WeatherEntryType::SI;
    break;
  case 27:
    return WeatherEntryType::SK;
    break;
  }
}

/**
 * Converts country code string to corresponding WeatherEntryType
 * @param input Two-letter country code (e.g., "AT" for Austria)
 * @return Corresponding WeatherEntryType enum value
 */
WeatherEntryType WeatherEntry::mapFromInputToRegion(std::string input) {
  // Convert input to uppercase for consistent matching
  std::transform(input.begin(), input.end(), input.begin(), ::toupper);

  // Map country code to region type using weatherRegionMap
  switch (weatherRegionMap.at(input)) {
  case WeatherEntryType::AT:
    return WeatherEntryType::AT;
    break;
  case WeatherEntryType::BE:
    return WeatherEntryType::BE;
    break;
  case WeatherEntryType::BG:
    return WeatherEntryType::BG;
    break;
  case WeatherEntryType::CH:
    return WeatherEntryType::CH;
    break;
  case WeatherEntryType::CZ:
    return WeatherEntryType::CZ;
    break;
  case WeatherEntryType::DE:
```

```
    return WeatherEntryType::DE;
    break;
  case WeatherEntryType::DK:
    return WeatherEntryType::DK;
    break;
  case WeatherEntryType::EE:
    return WeatherEntryType::EE;
    break;
  case WeatherEntryType::ES:
    return WeatherEntryType::ES;
    break;
  case WeatherEntryType::FI:
    return WeatherEntryType::FI;
    break;
  case WeatherEntryType::FR:
    return WeatherEntryType::FR;
    break;
  case WeatherEntryType::GB:
    return WeatherEntryType::GB;
    break;
  case WeatherEntryType::GR:
    return WeatherEntryType::GR;
    break;
  case WeatherEntryType::HR:
    return WeatherEntryType::HR;
    break;
  case WeatherEntryType::IE:
    return WeatherEntryType::IE;
    break;
  case WeatherEntryType::IT:
    return WeatherEntryType::IT;
    break;
  case WeatherEntryType::LT:
    return WeatherEntryType::LT;
    break;
  case WeatherEntryType::LU:
    return WeatherEntryType::LU;
    break;
  case WeatherEntryType::LV:
    return WeatherEntryType::LV;
    break;
  case WeatherEntryType::NL:
    return WeatherEntryType::NL;
    break;
  case WeatherEntryType::NO:
    return WeatherEntryType::NO;
    break;
  case WeatherEntryType::PL:
    return WeatherEntryType::PL;
    break;
  case WeatherEntryType::PT:
```

```
    return WeatherEntryType::PT;
    break;
  case WeatherEntryType::RO:
    return WeatherEntryType::RO;
    break;
  case WeatherEntryType::SE:
    return WeatherEntryType::SE;
    break;
  case WeatherEntryType::SI:
    return WeatherEntryType::SI;
    break;
  case WeatherEntryType::SK:
    return WeatherEntryType::SK;
    break;
  }
}
/**
 * ====================================================
 *  End of written code section
 * ====================================================
*/


/**
 * Author: UoL
 * Date: 2024-23-12
 * Notes: All code taken from the initial program given
*/

#include "MerkelMain.h"

/**
 * Main entry point for the weather analysis application
 * @return 0 on successful execution
 */
int main() {
  // Create instance of the main application class
  MerkelMain app{};
  // Initialize and start the application
  app.init();
}
```