Habitat

Version # 3.0

April 12, 2017

Group 03

Members: Kareem Khaled, Mohid Makhdoomi,
Abdul Moiz, Jakub Pawlikowski, Tyler Phillips
CompSci 2XB3 L02
Department of Computing and Software
McMaster University

Table of Contents

Revision Page

Version 1.0 - Completed April 10, 2017: All the sections for the modules were completed
Version 2.0 - Completed April 11, 2017: The UML diagram was created and linked in the document
Version 3.0 - Completed April 12, 2017: The final sections were added and the document was thoroughly edited for any format or spelling errors

Kareem Khaled	400032153	Project Secretary, Developer
Mohid Makhdoomi	400021935	Lead Developer
Abdul Moiz	400017302	Project Coordinator, Developer
Jakub Pawlikowski	400011899	Head of Research, Developer
Tyler Phillips	400017512	Developer, Design Analyst

“By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.”

Contributions Page

Name	Roles	Contributions	Comments
Tyler	Developer, Design Analyst	• Created the Crime Rate module which gets and parses data for a list of cities from the crime rate data set • Created graphical representation and layout of GUI • **Powerpoint presentation :** Databases Used and References • **Requirements Spec:** Other Requirements, Priority of Functions • **Design Specification:** Crime Rate, Merge, BinarySearchST, Queue CityT, DataGetter, Rank,UI, UML Diagram, Graph UML State diagram	
Abdul	Coordinator, Developer	• Created Income module which gets and parses data for a list of cities from the income data set • Created the ModifiedMSD module which allows for an array of string arrays to be lexicographically sorted • Created the Graph module to incorporate a graphing algorithm and additional functionality to the program • **Powerpoint presentation:** Objectives and Scope slide Verification and Validation slide • **Requirements Spec:** The Domain • **Design Specification:** Executive Summary, Modules writeup, Income, ModifiedMSD, Graph, Internal review and evaluation	
Mohid	Lead Developer	• Created the DataGetter module which handles and merges all dataset modules • Created the CityT module which is an ADT for representing cities and their respective data • Created the UI module which provides an interface for the user to input cities and preferences for Climate, Crime, Income, Population and integrates Graph module • Created the Rank module which ranks the quality of life in cities from best to worst • Created prototype application for presentation • Created the Test class for debugging purposes • Provided feedback on team member's modules for optimizing integration with other components • Added alg4.jar to the project • Re-integrated updated modules from team member's to the project implementation throughout the course of the project • Put together the final implementation of the project • **Powerpoint presentation :** Functional requirements slide Presented Prototype • **Requirements Spec:** Functional Requirements • **Design Specification:** Cleaned up/formatted Contribution page Edited some classes in the Modules sections	
Jakub	Developer, Head of Research	• Created Population module which gets and parses data for a list of cities from the population data set • Created functional aspect of GUI and connected it to backend of project • Added required library to project file (commons-lang3-3.5.jar) • **Powerpoint presentation :** Design Specifications slide Algorithms slide • **Requirements Spec:** Quality Control, Likely changes • **Design Specification:** Population,GUI	

Kareem	Secretary, Developer	• Created Climate module which gets and parses data for a list of cities from the climate data set • Created and submitted all deliverables required after meetings • Recorded all details during meetings and submitted meeting minute documents, text files, and images to document our progress • Updated project log and task list after every meeting • **Powerpoint presentation :** Non-functional Requirements slide • **Requirements Spec:** Non-functional requirements • **Design Specification:** Climate, managed contributions page, revision page, UML state diagram for Climate	

Executive Summary

Habitat is an application developed to give the user an accurate and quantitative measure of quality of life in a location. The main benefactors of such an application would be current citizens of Canada, immigrants and also real estate companies. This solution is produced by an algorithm that takes values for a location from various databases and calculates the quality of life in that city. This value can then be compared to other cities and thus, the optimal location to live can be selected by the user. The inception of Habitat was done to enable people to be able to measure how pleasant living in an area would be, beyond just their direct household or neighborhood. Furthermore, it would ensure that people who are immigrating to Canada reside in an area with the most opportunities and pleasant living experience.

Modules

The main rationale for the decomposition of the classes was to assign a single task to each module and ensure that changes could easily be made to the implementation of one module without destabilizing the performance of the program as a whole. Additionally, by modularizing the implementation the workload could be equally split up amongst the various group members.
Another key motivation behind the decomposition was to divide the program into core modules and supplementary modules called helpers. By doing so, a very clear understanding of uses relationships and dependencies can be easily identified. Simple and easy verification and validation of the entire project as a whole was also a goal our team had in mind when decomposing the program. By dividing the modules first into cores and then helpers it allowed us to focus more of our resources in assuring the core methods are error free and robust, while the helper methods, which perform simple tasks, are quickly dealt with and out of the way.

Income: This class' sole purpose is to take an array of city names and return the income for each city, the value that was used is the pre - tax income. This is due to the fact that income in general is commonly measured using its pre - tax value. To return this value the class searches through each line of the dataset and using a set of checks returns the appropriate value.

- Interface for the **Income** module:
    - get():
        - This method returns the integer list of values that are associated with the inputted list of cities
        - The indexes of the array match up directly with the indexes of the inputted array of strings of cities
    - Trace to requirements:
        - One of the requirements of the project was to use a set of variables to determine the quality of life in a given area, this module achieves this task for the variable of income in an area. Furthermore, the module also meets the input and output requirements due to the fact that the input can be an array of city names and the output is an array of the values for those cities. Non-functional requirements we had set were performance and robustness. By presorting the array and using a binary search the run time can be minimized thus ensuring performance. Additionally, by using the Integer data type if a city is not present it can easily be translated throughout the program by returning null.

- Implementation of the **Income** module:
    - Variables:
        - cities: an array of strings that is used as the input by the user. The elements of the array are the cities the user would like information about.
        - tempCities: an arraylist of arrays of strings. The purpose of having this variable is to hold each line of the dataset that is needed in its split form. This is to be able to select a specific line from the dataset and then be able to select a certain part of that line.
        - allCities: a two dimensional array. This array contains arrays of strings. Essentially it is identical to *tempCities*, however its importance lies in the easy and familiar manipulation of arrays over arraylists and thus was used heavily in the radix sort and binary search.
        - values: this array of integer values hold the extracted data from the datasets for the cities specified by the user, the indexes of this array line up directly with the indexes of the inputted array.
    - Methods:
        - Income(String[] input): The constructor initializes all the state variables to their primitive values, the constructor also runs the binarySearch local method to extract the values for income for each city inputted by the user. This is done by simply iterating through the cities array using a for loop.
        - sortedArray(): This method goes through each line of the dataset and takes the relevant line, specifically the line pertaining to the income of both genders of all age groups in a city and adds it to an unsorted arrayList of string arrays (tempCities). A line is determined to be relevant or not through a series of if statements ensuring it contains only the information required. The allCities variable is set to the value of tempCities after it is converted to a two dimensional array object. The array allCities is then sorted using a modified radix sort.
        - binarySearch(): This method takes a target string and searches through a sorted two dimensional array of arrays of strings. The search is done by checking the middle point of the array allCities. If the target string is lexicographically smaller than the middle point only the left half of the array is now considered and the process is repeated. This is done until the current middle point is actually the target string. When the target string is found, being the city, the value for the income of that city is added to values array.

ModifiedMSD: This class takes an array of string arrays and sorts them using the radix sort algorithm. The code is referenced from the algorithms textbook and lectures in 2C03.

- Interface for the **ModifiedMSD** module:
    - sort(String[][] a):
        - This method initializes the necessary local variables (the integer *n* and the array *aux*) that are necessary to perform the sort. The method then calls the local method also called sort which recursively sorts the input array *a* using the left most digit first.
    - Trace to requirements:
        - One of the requirements that was set for our program was efficiency and performance. By including a radix sort and presorting all the data, searching for values for specific cities becomes significantly quicker. Additionally, by including algorithms such as this

class the input size can be fairly large and a difference in performance will not be noticeable.

- Implementation of the **ModifiedMSD** module:
    - Variables:
        - R: This variable represents the size of the alphabet being used in the sort however is kept at 256 for the standard ASCII alphabet for the sake of the problem
        - CUTOFF: This variable represents the minimum size of a subarray that will simply be sorted using insertion sort as for arrays this size, the runtime is negligible.
        - a: The array that is to be sorted, is not created as a state variable, however is maintained throughout the whole class by being constantly passed. All mutations to the array are maintained however as the actual array is passed.
    - Methods:
        - chartAt(String s, int d):This method returns the ASCII integer value of a selected character *d* from a string *s*. The selection is done by choosing the character at the index *d* in the string and returning its integer ASCII equivalent.
        - sort(String[][] a, int lo, int hi, int d, String[][] aux): This method performs the main radix sort recursively, The procedure is done by first sorting the entire array using the most significant digit (left most character), then the array is divided into sub arrays and the second most significant digit is sorted one sub array at a time. The bounds for the subarrays are
        - insertion(String[][] a, int lo, int hi, int d): For sub arrays that are smaller than a selected cutoff a simple insertion sort is used to sort the array. The insertion sort is performed in the standard manner by iterating through each element of the array and transferring it to a sorted portion. Additionally, each element is compared to every other element in the unsorted section.
        - exch(String[][] a, int i, int j): This method simply swaps two given indexes in the array *a*. The swap is done by creating a temporary variable and holding one element there while the swap is taking place.
        - less(String v, String w, int d): This method checks which if a given string is lexicographically lower than another given string.

Graph: This class represents and constructs a graph of the cities in Ontario was designed to work for all the cities in Canada however due to the sparse clusters of cities in other provinces within our datasets the scope for just this section was limited to Ontario. The code is inspired and sourced from the algorithms 4 textbook and the 2C03 lectures.

- Interface for the **Graph** module:
    - init():
        - This method must be called at the start of the program, it pulls all the data from the dataset containing cities and their positions in Ontario. it stores that data in an array list and the names in the reference list array list. It then also initializes the bags for the indexed adjacency lists for each vertex. Finally it calls the buildGraph method to actually create the edges between the vertices in the graph.
    - adj(String s):
        - This method returns the adjacent cities to a given city represented by its name. This is done by going through that selected city's adjacency list and for each value referencing to the city name in the reference list. These city names are then added to a temporary

iterable list which is then finally returned. To access the correct adjacency list the string's name is converted to its corresponding integer value by using the reference list.

- Trace to requirements:
  - A major requirement of this project was to involve a graphing algorithm which is what this class does. It is to be noted that this algorithm was designed to work for all the cities in Canada however due to the sparse clusters of cities in other provinces within our datasets the scope for just this section was limited to Ontario. This decision was made to show the usefulness and functionality of including this module however only in situations where it applies. Some requirements we set out was for our program to be able to compare the user's best option to any nearby cities that may possible result in a higher quality of life. This class efficiently and reliably performs that task. By including checks such as cities not being present in all datasets or cities being isolated this class is reliable and correct. Two non-functional requirements we set out for our program.

- Implementation of the **Graph** module:
  - Variables:
    - V: This variable represents the number of vertices in the graph. It is helpful for initializing the other variables and also for iterating through the graph.
    - E: This variable represents the minimum size of a subarray that will simply be sorted using insertion sort as for arrays this size, the runtime is negligible.
    - information: This arraylist of string arrays holds the lines of the dataset so they can easily be accessed by simply using an index in the arraylist.
    - referenceList: This variable is an arraylist of strings that holds the names of the cities. Having this variable is critical because it ensures that although the graph algorithm functions using just integers and indexes, at any given time a city being represented by an integer can be referenced.
  - Methods:
    - buildGraph(): This method takes each vertex in the graph and connects it to its surrounding cities. This is done by checking the proximity of a city to all the other cities. Since the edges are undirected and two ways once a city has been checked it does not need to be ever checked again. Hence why the inner loop starts at i + 1
    - checkProximity(int a, int b): * This method takes two integer values and find the city names that correspond with them from the referenceList. It then checks whether the cities fall within a 0.75 difference in longitude and latitude of each other using a series of if statements.
    - addEdge(int v, int w): A edge is created between two vertices by inserting each vertex in the other's adjacency list.

Population: The purpose of this module is to return the population density for each of the user's cities. It accepts an array of strings with the city names and returns an array list of integer values with the population density. The population density is used because it gives an excellent estimate of how many people can be expected in a given area. This class reads from a population dataset and uses mergeSort and binary searches.

- Interface for the **Population** module:
  - dataReader():
    - This method returns an int array list of values associated with the user's inputted cities.Each value corresponds to the population density for each city.
    - The indexes of the array match up directly with the indexes of the inputted array of strings of cities

- Trace to requirements:
    - One of our project's requirements was to find the population density for a given area so that it's quality of life can be determined. This is what this module accomplishes by returning the population density for each city. It also meets our requirements regarding input and output, as it accepts an array of city names and returns an array of int values.

- Implementation of the **Population** module:
    - Variables:
        - cities: is an array list of strings that contains the input from the user. Each element in the list is a name of one of the cities the user wants the program to search through.
        - sorted: is the array list of string values. This contains the datasets information with each line held as a string.It has been sorted using MergeSort in alphabetical order by the city name.
        - Pops: is an array list of integer. Each value in this list holds the population density for each city in the cities arrayList. It is returned to DataGetter
    - Methods:
        - citySorting:this function's purpose is to Reads csv file of data (city names and their populations) creates an array of data (lines of the csv) (City (prov.), pop) sorts the array using mergeSort.
        - dataReader:This function will accept an array list of city names and an array list of the dataset statistics. It will then perform a binary search to find the population density of the cities in the array list. It returns a string array list of the populations
        - Main:. The main method is crucial for this module. It calls the other functions to create an arrayList of population densities for each of the user's cities.

Crime Rate: This class' function is to take an array of strings with city names and provinces and return the crime rate for each city. I used the crime severity index as values. This is because it is an excellent representation of the crime level in the given city. This class finds and returns the list of array values by using a merge sort followed by a binary search.

- Interface for the **Crime Rate** module:
    - get_CrimeRate():
        - This method returns a double array list of values associated with the user's inputted cities.
        - The indexes of the array match up directly with the indexes of the inputted array of strings of cities
    - Trace to requirements:
        - In our project, one of the requirements was to find the crime rate for a given area so that it's quality of life can be determined. This is what this module accomplishes by returning the crime severity index for each city. It also meets our requirements regarding input and output, as it accepts an array of city names and returns an array of double values.

- Implementation of the **Crime Rate** module:
    - Variables:

        - data: is an array list of strings. It is used to store the dataset's information. Each string in the list contains a line in the csv file/dataset.
        - listCit: is a string list that contains the input from the user. Each element in the list is in the format of "city, province" for the names of the cities the user wants the program to search through.
        - Cr: is the array list of double values. This list is the module's final output. Each value is the crime severity index for a city that the user inputted. The indices of this array math listCit perfectly, to avoid any confusion.
    - Methods:
        - getData:this function's purpose is to acquire the dataset's values. It accesses the dataset through a csv file and reads each line, converts each line to a string and adds the string to an array list called data. It calls dataClean(), sending the array list as a parameter. dataClean()'s output is sent to sorted() as a parameter. A sorted array list of strings is returned to getData() and is sent to the user.
        - Sorted:This function's main purpose is to sort the dataset's data in alphabetical order by city name. It contains a parameter named temp which is the array list of strings, with each string holding a line from the dataset. The function uses a Merge class and sorts the data. The sorted array list of strings is returned to the user.
        - dataClean:This function's purpose is to remove unnecessary lines from the data arraylist. It contains a parameter named temp which is the array list of strings, with each string holding a line from the dataset. Only the lines that contain data on crime severity index from the year 2014 are kept in the array list. The cleaned array list is returned.
        - cleanList: This function's purpose is to go through the inputted list of strings and find special cased cities that have complicated names in the csv file. In the csv file some cities are merged and share one line of data, such as Oakville in Burlington. This function takes a parameter called listCit which is a string list that contains the input from the user and replaces names of cities with the corresponding name found in the csv file. Without this function, the special cased cites such as Burlington and Oakville would cause major errors.
        - searcher: This function searches the dataset for the crime severity index of the cities that have been inputted by the user. It contains a parameter named data which is the array list of strings, with each string holding a line from the dataset. This dataset has been filtered through by dataclean(). It takes another parameter called listCit which is a string list that contains the input from the user. This has been filtered usign cleanList(). Binary Search is then used to search for each city name in listCit in data and appends the value associated with it to a string array list called cr. cr is returned.
        - get_CrimeRate:This function is responsible for returning a double array list called cr that contains the crime severity index for each inputted city.This function takes a parameter called listCit which is a string list that contains the input from the user. It calls getData() and sends its output, as well as listCit to searcher() which returns a string array list with the crime severity index. This is then converted to double values and is returned to the user.

Merge: This class' function is to sort an array list containing data from the datasets of crime rate and population. It will sort the data in alphabetical order by city name.

- Interface for the **Merge** module:
  - sort():
    - This method sorts a comparable list alphabetically
  - Trace to requirements:
    - In our project, one of the requirements was to implement a sorting algorithm. Merge accomplishes this by sorting our dataset.

- Implementation of the **Merge** class:
  - Variables:
    - hi: is an integer value that holds the highest index of the list at a given iteration
    - lo:is an integer value that holds the lowest index of the list at a given iteration
    - mid:is an integer value that holds the center index of the list at a given iteration
    - a: is a Comparable[] list of values to be sorted
  - Methods:
    - merge:this function's purpose is to stably merge a[lo .. mid] with a[mid+1 .. hi] using aux[lo .. hi]. It takes Comparable list a as a parameter.
    - sort:This function rearranges the array in ascending order, using the natural order.
    - Less: this function compares two Comparable type values (Comparable v, Comparable w) it returns a boolean based on whether the v is greater than w.
    - IsSorted: this function returns a boolean value based on whether the Comparable type list called 'a' is sorted. It accepts 'a' as a parameter.
    - indexSort: This function returns a permutation that gives the elements in the array in ascending order. It takes Comparable list a as a parameter.
    - show:This function is responsible for printing out the sorted Comparable list a.

BinarySearchST: This class' function is to search an array list containing data from the datasets of crime rate and population. It will search for the values that are associated to the key city names that the user has inputted.

- Interface for the **BinarySearchST** module:
  - **BinarySearchST**():
    - This creates an object for the class
  - Trace to requirements:
    - In our project, one of the requirements was to implement a searching algorithm. BinarySearchST accomplishes this by searching our datasets and finding the desired value.

- Implementation of the **BinarySearchST** class:
  - Variables:
    - keys: is a list of Key Comparable type objects
    - Vals: is a list of Value Comparable type objects
    - n: is the size of the symbol table
  - Methods:
    - Constructor:this function's purpose is to Initializes an empty symbol table with the specified initial capacity. Capacity is an integer parameter.

- Resize:This function resize the underlying arrays vals and keys to its integer parameter called capacity.
- size: this function returns the number of key-value pairs in this symbol table.
- isEmpty: this function returns a boolean value based on if this symbol table is empty .
- contains: This function takes a Key object named key and returns a boolean value based on if the symbol table contains key.
- get:This function is responsible for returning the value associated with the given key in this symbol table.
- rank:This function is responsible for returning the number of keys in this symbol table strictly less than {@code key}.
- put:This function inserts the specified key-value pair into the symbol table, overwriting the value with the new value if the symbol table already contains the specified key. Deletes the specified key (and its associated value) from this symbol table if the specified value is {@code null}.

- delete:This function takes a Key object named key and removes the specified key and associated value from this symbol table (if the key is in the symbol table).
- deleteMin:This function is responsible for removing the smallest key and associated value from this symbol table.
- deleteMax:This function is responsible for removing the largest key and associated value from this symbol table.
- min:This function is responsible for returning the smallest key in this symbol table.
- max:This function is responsible for returning the largest key in this symbol table.
- select:This function is responsible for taking an int k as input and returning the kth smallest key in this symbol table .
- floor:This function is responsible for taking a Key object named key and returning the largest key in this symbol table less than or equal to {@code key}.
- ceiling : This function is responsible for taking a Key object named key and returning the smallest key in this symbol table greater than or equal to {@code key}.
- size:This function is responsible for returning the number of keys in this symbol table in the specified range. It accepts two Key objects called hi and lo.
- Keys:This function is responsible for returning all keys in this symbol table as an {@code Iterable}.To iterate over all of the keys in the symbol table named {@code st}, use the foreach notation: {@code for (Key key : st.keys())}.
- keys: This function is responsible for returning all keys in this symbol table in the given range, as an {@code Iterable}. .It accepts two Key objects called hi and lo.
- check:This function is responsible for checking if isSorted() and rankCheck() is true.
- rankCheck:This function is responsible for checking that rank(select(i)) = i

Queue: This class' purpose is is to create a queue of Item objects using the first in first out principle as it is needed to implement Binary Search.

- Interface for the **Queue** module:

    - **Queue**():
        - This creates an object for the class
    - Trace to requirements:
        - In our project, one of the requirements was to implement a searching algorithm. BinarySearchST.java uses this class to accomplish this.

- Implementation of the **Queue** class:
    - Variables:
        - first : is a Node item that holds the beginning of the queue.
        - last: is a Node item that holds the end of the queue.
        - n: is an integer that holds the size of the queue
    - Methods:
        - Constructor:this function's purpose is to Initialize the first and last variables to null and n to 0.
        - isEmpty:This function returns a boolean based on if the queue is empty
        - size: this function returns the number of key-value pairs in this symbol table.
        - isEmpty: this function returns true if this symbol table is empty .
        - size: This function n, the size of the queue. No parameter
        - peek:This function is responsible for returning the first value in the queue
        - enqueue:This function is responsible for taking an Item object and adding it to the queue.
        - dequeue:This function is responsible for removing the last item in the queue.
        - toString:This function returns the queue as a string.
        - iterator:This function returns an iterator over the items in the queue in proper order.
        - Private Class ListIterator:
            - constructor:This function is responsible for
            - hasNext:This function is responsible for returning a boolean value based on whether the queue has a next value.
            - remove:This function is responsible for throwing an UnsupportedOperationException.
            - next:This function is responsible for returning the next item in the list.

Climate: Newcomers to Canada can experience various climates across Canada. They may have a preference to what they would enjoy, so we take climate into account. ClimateGetter.java takes an array of strings in the form CITYNAME,PROVINCE and returns the mean temperature and precipitation of those cities for every season. It accesses 4 separate databases; one for each seasonal equinox or solstice. This is done to ensure accuracy and correctness, as there aren't any cities in Canada with a temperature that does not fluctuate per season.

- Justification for using linear search: Weather station names do not always start with the city name. Even if we alphabetize the data (which a binary search would require), many matches for a city are far apart. In order to ensure all results are caught, we use a linear search. Also, linear search is O(n) whereas we would need over O(nlogn) for a binary search in this scenario. (to insert into an array, sort, and find cities). For performance and accuracy, we use a linear search.
- Interface for the **Climate** module**:**
    - getClimate(String [] of Cities):

        - This method returns an ArrayList of String[] ArrayLists. It is first split into 4 seasons: Spring, Winter, Fall and Summer. Each season has a String[] array for every city in the same order that the cities are provided by the user.
        - getClimate first calls SetUpArrays(cities) to initialize all the arrays present in the code. setUpSeasons() is then called to find all city matches in the dataset and store them into the correct array by season. Finally, the cities in the array are selected and sorted in the order that the original cities were received in.
    - Trace to requirements
        - Having this module in our application gives the user an additional factor in their quality of life for a particular city, as stated in the requirements specification.
        - Meets the non-Functional requirement of maintainability, as ClimateGetter.java is a separate data access method that can be modified without changing any other modules for any new requirements. Also, data access methods for 4 datasets within the module can be changed by modifying just one method instead of 4. This generality helps with maintainability, as dataset accesses or selecting/sorting cities for all 4 seasons can all be modified together under one function.

- Implementation of **Climate** module**:**
    - Variables (ArrayList<String[]>)
        - allSpring: maintains all city catches for Spring
        - allSummer: maintains all city catches for Summer
        - allFall: maintains all city catches for Fall
        - allWinter: maintains all city catches for Winter
        - sortedSpring: selects and stores the best city for Spring in order of provided cities
        - sortedSummer: selects and stores the best city for Summer in order of provided cities
        - sortedFall: selects and stores the best city for Fall in order of provided cities
        - sortedWinter: selects and stores the best city for Winter in order of provided cities
        - checked: keeps a record on cities that have been stored
        - cities: keeps track of all city names to be found. Split from CP for code understandability
        - CP: keeps track of provinces corresponding to each city
    - Methods:
        - setUpArrays(String[]): initializes every state variable, adds capitalized city and province names to *CP*. Also adds capitalized city names to *cities*.
        - setUpSeasons(): calls setUpSeason(dataset name, *all'season'* array) for every season.
        - sortSeasons(): calls sortSeason(*all'season'*, *sorted'season*') for every season.
        - setUpSeason(dataset name, *all'season'* array) linearly appends all city catches between cities and the dataset to *all'season'* (the specified season).
        - sortSeason(*all'season'*, *sorted'season*'): goes through *all'season'* and appends the most useful data for a city into *sorted'season'.* For each city added to *all'season'*, the city is appended to *checked* to ensure that there is no greater than one catch, and if there isn't a catch, null is appended in order into *sorted'season'*.
        - existsInName(string): works in setUpSeason to ensure entries appended to *all'season'* are valid substrings of our cities in *cities*.

        - notChecked(string): works in sortSeason to ensure only one city from *cities* is appended into *sorted'season'*. Otherwise, a null value is appended in its place.

CityT: This class' purpose is to create an object that contains getter and setter methods for name, climate, crime, income, and population of a city.

- Interface for the **CityT** module:
    - CityT():
        - This module is not used in the interface of the program, rather it is used to create objects used by the DataGetter class.
    - Trace to requirements:
        - Having this module in our application gives the project increased modularity to implement our other factors of quality of life as stated in the requirements specification.

- Implementation of the **CityT** module:
    - Variables:
        - name: This is a string variable that holds the name of the city.
        - climate: This is an array list of strings whose purpose is to hold the climate data of a city.
        - crime: This is a double value that holds the crime rate of a city
        - income: This is a double value that holds the average income of a city
        - population:This is an integer value that holds the population density of a city
    - Methods:
        - Constructor:this function's purpose is to set the class variables to their inputted values. These include: name, climate, crime, income, and population.
        - name:This function's main purpose is to return the string name of the city to the user.
        - get_Climate:This function's main purpose is to return the string array list form the variable climate.
        - get_Crime:This function's main purpose is to return the double value form the variable crime.
        - get_Income:This function's main purpose is to return the double value form the variable income.
        - get_Population:This function's main purpose is to return the integer value population.

DataGetter: This class' purpose is to connect the Population, Climate, Crime Rate, and Income modules and obtain the data for each of the user's cities.

- Interface for the **DataGetter** module:
    - get_allCity():
        - This is a method that returns a list of CityT objects. Each object corresponds to a city the user has entered and contains all the data required for our project.
    - Trace to requirements:
        - Having this module allows us to get the required information from several datasets so that we can return the optimal cities to the user, as stated in the requirements.

- Implementation of the **DataGetter** module:
    - Variables:
        - cityList: This is a string list of the the input cities of the user.
        - climate: This is a nested array list of strings whose purpose is to hold climate data for each city in cityList, which is acquired from the Climate module.
        - crime: This is an array list of double values whose purpose is to hold data(crime severity index) of each city , which is acquired from the Crime Rate module.
        - income: This is an array list of double values whose purpose is to hold data(average income) of each city, which is acquired from the Income module.
        - population:This is an array list of integer values whose purpose is to hold data(population density) of each city , which is acquired from the Population module.
        - allCity:This is an array list of CityT objects, each object contains data(name, climate, crime, population, income) on each city in cityList
    - Methods:
        - Init: This function initializes the abstract object with a list of cities. Using the modules in the dataGetters package, the class is implemented so that the abstract object contains data for the Climate, Crime, Income and Population of the user provided list of cities.
        - get_Climate:This function's main purpose is to return the climate data for all cities in cityList.
        - get_crime:This function's main purpose is to return the crime data for all cities in cityList.
        - get_income:This function's main purpose is to return the average income data for all cities in cityList.
        - get_population:This function's main purpose is to return the population density data for all cities in cityList.
        - get_allCity:This function's main purpose is to return a list of the population,income, crime, and climate data for each city in cityList.
        - get_City: This function will return all the data(income, climate, crime, population) data for an inputted city.
        - provHelper: This function takes a list of provinces, and an integer of the number of cities and returns customized names of the provinces specifically for each dataset.
        - climateHelper: This function accepts cityList(list of user's cities) and returns the climate data using ClimateGetter.
        - crimeHelper: This function accepts cityList(list of user's cities) and separately returns the crime rate data using CrimeGetter.
        - incomeHelper: This function Takes the list of city names and returns the income data using IncomeGetter. The income dataset requires that the value for each city be divided by its population and then multiplied by 1000 to get the accurate value and so this is done after using IncomeGetter to get the data from the income dataset.
        - populationHelper:This function accepts cityList(list of user's cities) and separately returns the population data using PopulationGetter.

Rank: This class' purpose is to creates an abstract object for calculating the quality of life for a list of cities using the user's preferences on crime, climate, income and population.

- Interface for the **Rank** module:
  - getRankedCities():*
    - This function returns the ranked list of cities in order of highest Quality of Life to lowest.
  - Trace to requirements:
    - This module implements an abstract object for calculating the Quality of Life for a list of cities based on preferences on Climate, Crime, Income and Population. This is stated in the requirements.
- Implementation of the **Rank** module:
  - Variables:
    - preferences: This is a integer list of user preferences on importance/value of all 4 factors(Climate (Temperature, Precipitation), Crime, Income, Population) Note: Climate is broken down into two sub preferences, Temperature and Precipitation.
    - cityList: This is a string list of the the input cities of the user.
    - notAvailable: This is a list of boolean's indicating cities that are missing from one or more dataset. The index of this list aligns with the index of cityList.
    - factorMult: This is a list of multiplier values for Quality of Life calculations that are derived from the user provided preferences.
    - avg: This is an array list of list of average values for all factors of all the cities.
    - cityWeight: This is a list of Quality of Life values (weights) for all the cities.
    - weightPerFactor: This is a list of the Quality of Life values for each factor for all the cities.
    - rankedCities: This is a ranked list of cities in order of highest Quality of Life to lowest.
  - Methods:
    - init: This function initializes the abstract object with a list of preferences and a list of cities. Using the DataGetter class, this class is implemented so that the abstract object contains data for which cities have no data, the average values for all 4 factors (Climate, Crime, Income, Population), the Quality of Life of all/each city as well as those cities ranked in order of highest QoL to lowest.
    - finalRank:This function's main purpose is to rank list of cities in order of highest Quality of Life to lowest. It accepts cityList as input.
    - calcQOL: This function Calculates the Quality of Life value (weight) for each city using the percentage deviation of all the factors of each city from the average of all the cities. It accepts a list of cities and their data as an array of CityT instances as input.
    - calcAvg: This function's main purpose is to calculates the average values for all factors using the list of cities while taking into account cities that are missing from datasets. It accepts a list of cities and their data as an array of CityT instances as input.
    - noDataCount: This function calculates the number of cities missing from one or more datasets.
    - climateHelper: This function's main purpose is to calculate the average values for Temperature and Precipitation for a city taking into account any missing values.
    - convertToMultiplier: This function takes a list of preferences for the factors (1 <= value <= 10) and converts them to a list of multiplier values.

        - deviationHelper: This function calculates the percentage deviation of a value from the average.
        - getPreferences: This function returns the list of user preferences on importance/value of all 4 factors (Climate (Temperature, Precipitation), Crime, Income, Population) Note: Climate is broken down into two sub preferences, Temperature and Precipitation.
        - getCityList: This function returns the list of cities provided by the user.
        - getNotAvailable: This function returns the list of boolean's indicating cities that are missing from one or more dataset. The index of this list aligns with the index of cityList.
        - getFactorMult:This function returns the list of multiplier values for Quality of Life calculations that are derived from the user provided preferences.
        - getAvg: This function returns the list of average values for all factors of all the cities.
        - getCityWeight: This function returns the list of Quality of Life values (weights) for all the cities.
        - getWeightPerFactor: This function returns the list of the Quality of Life values for each factor for all the cities.
        - getRankedCities: This function returns the ranked list of cities in order of highest Quality of Life to lowest.

UI: This class' purpose is to create a user interface through its main method so that the user can input cities and preferences on Crime Rate, Population, Income and Climate and so it can be used to display the cities ordered based on the user's choices. This class also incorporates output from the Graph class to suggest better cities in accordance with the user's preferences.

- Interface for the **UI** module:
    - The module's main method is used to create an interface for the user and is used to take input as well as display final output.
    - Trace to requirements:
        - This module receives the list of cities the user is interested in moving to, along with the weight values for(climate ,crime, income, and population). It also displays the top cities based off of the user's choice as stated in the requirements.
- Implementation of the **UI** module:
    - Variables:
        - prefs: This is a integer list of user preferences on importance/value of all 4 factors(Climate (Temperature, Precipitation), Crime, Income, Population) Note: Climate is broken down into two sub preferences, Temperature and Precipitation.
        - cityList: This is a string list of the the input cities of the user.
        - multipliers: This is a list of multiplier values for Quality of Life calculations that are derived from the user provided preferences.
        - cityWeight:This is a list of Quality of Life values (weights) for all the cities.
        - weightPerFactor:This is a list of the Quality of Life values for each factor for all the cities.
        - rankedCities:ranked list of cities in order of highest Quality of Life to lowest.
    - Methods:
        - Main: This function is used to build the user interface. It initializes the class variables, as it takes the factor preferences from the user as well as a list of cities

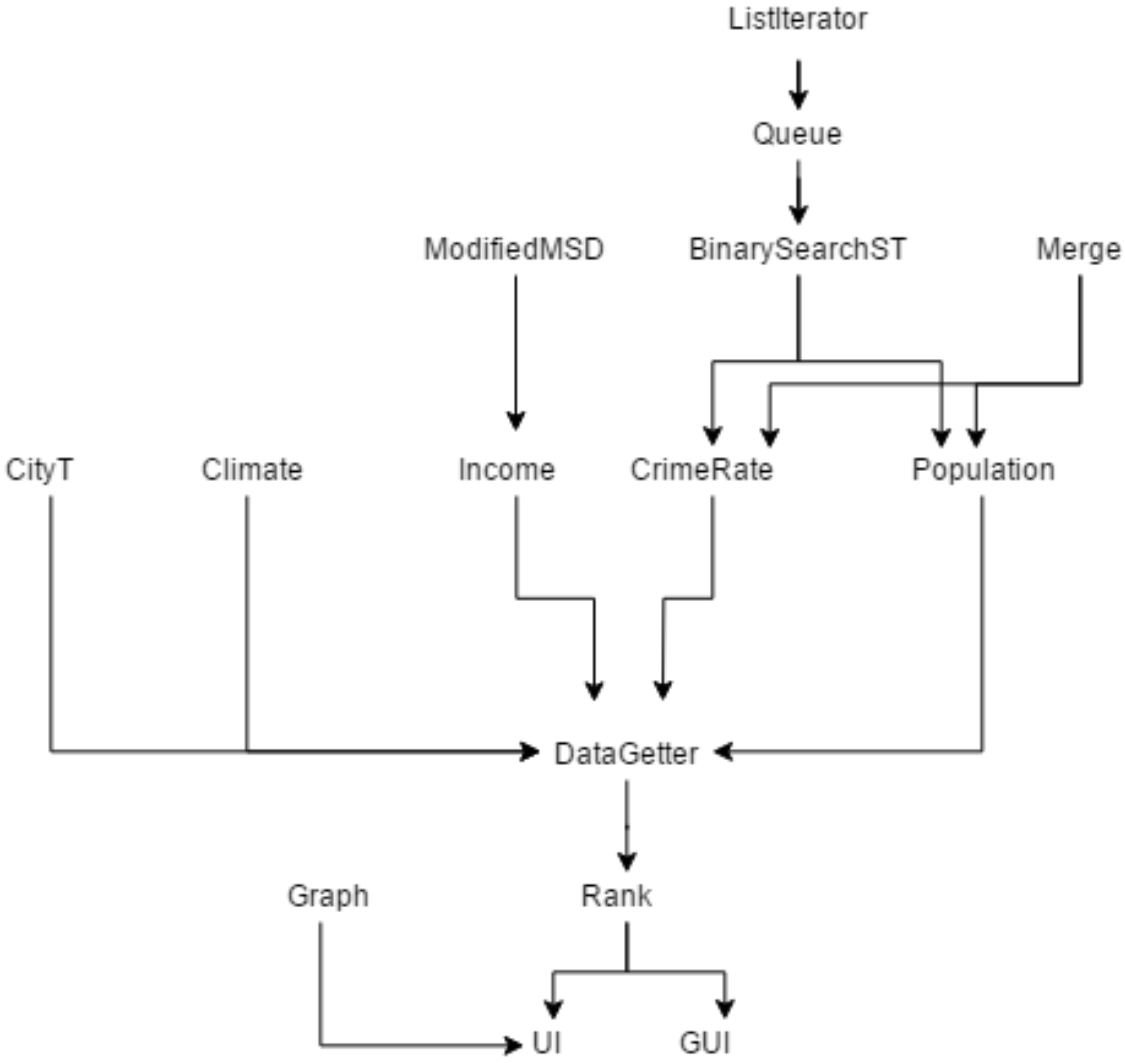
and uses them with the Rank module. It then receives a list of the user's cities ranked by preference for the user and displays the list as output. If the best city of the user inputted cities is in Ontario, it utilizes the Graph module as well as the Rank module again to suggest even better cities.
        - getInputRank:This function's main purpose is to return the input from the user for importance/value of factors (Climate (Temperature, Precipitation), Crime, Income, Population) that affect the Quality of Life of any city. Note: Climate is broken down into two sub preferences, Temperature and Precipitation
        - getInputCities: This function returns the input from the user for a list of cities in Canada
        - cityDisplay: This functions displays the rank, name and quality of life weights for individual cities.

GUI: The purpose of this module is to display the application in an easy to use and interactive manner. Similarly to the UI, the user can input cities as well as weightings for the ranking arguments. The inputted cities will be displayed in ranked order once the "Done" button is clicked.

- Interface for the **GUI** module:
    - runGui():
        - creates a gui interface for the user, it displays JTextFields (text boxes) for up to 10 cities to be entered, as well as areas to enter the weight factor for income, crime rate, population, and climate. It then runs the rest of our project and displays the final output to the user. The final output will be the list of cities the user inputted ordered based on the rankings from best to worst measured quality of life.
    - Trace to requirements:
        - This module obtains the list of cities the user is interested in moving to, along with the weight values for climate ,crime, income, and population. It uses a gui interface to accomplish this, and displays the cities in order of suitability to the user's tastes. This accomplishes one of our main requirements to obtain and display information to the user.
- Implementation of the **GUI** module:
    - Variables:
        - cityInput1: JTextField for user to input city
        - cityInput2: JTextField for user to input city
        - cityInput3: JTextField for user to input city
        - cityInput4: JTextField for user to input city
        - cityInput5: JTextField for user to input city
        - cityInput6: JTextField for user to input city
        - cityInput7: JTextField for user to input city
        - cityInput8: JTextField for user to input city
        - cityInput9: JTextField for user to input city
        - cityInput10: JTextField for user to input city
        - tempInput: JTextField for user to input temperature weighting
        - precipInput: JTextField for user to input precipitation weighting
        - crimeInput: JTextField for user to input crime rate weighting
        - incomeInput: JTextField for user to input income weighting
        - popInput: JTextField for user to input income weighting

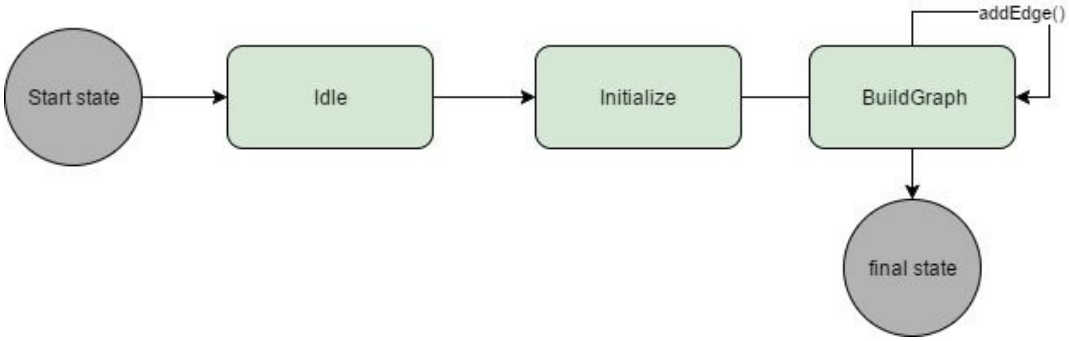
        - inputCities: Array of JTextFields which were used as city inputs
        - inputWeights: Array of JTextFields which were used as weighting inputs
        - btn: JButton which executes the backend of the application
        - Lbl: JLabel which displays the header of the frame
    - Methods:
        - runGui: This function is used to create the frame to be displayed as well as call the backend and display the output of the program. The first frame has information for the user as well as text boxes for input. The “Done” button runs the backend of the application and a new window and frame are opened to display output.

Uses Relationship Between Modules

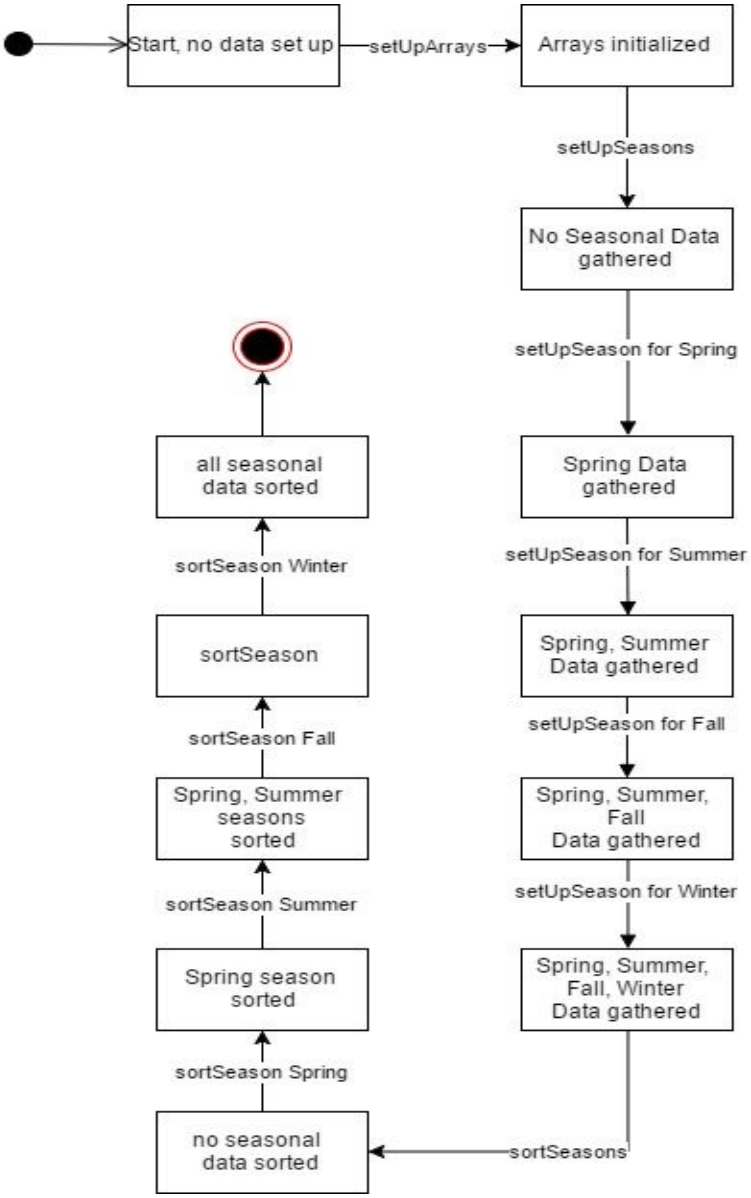


UML state machine diagrams

Graph



Climate



UML Diagram Link:

Gliffy was used to create the uml diagram. A link to the diagram has been provided as opposed to a jpg picture. This is also beneficial as it provides a better view of the diagram,
Thank you for your understanding.
https://www.gliffy.com/go/publish/11938463

Internal Evaluation

Overall, our final product meets most if not all of the requirements set within our requirements specification document. The only major goals we were not able to achieve were: given the user's top choice checking all the other close by cities and seeing if there was a better option. This was not able to be completely implemented to its full scope due to the lack of information in the datasets on less populated provinces. However, we still met the requirement for the most applicable province being Ontario. Also, modules were tested as they were created and after they were joined with the rest of the program. Formal junit testing was not used throughout the program due to time constraints. Another issue we ran into was inconsistency in datasets, to address this issue we had our program omit cities that were not in all datasets altogether. The cities that were omitted are cities that are not deemed CMA (Census Metropolitan Areas) by Statistics Canada. However, this omission does not hinder the usefulness of our program as given the needs of the stakeholders that are stated within the requirements document, the cities that are included already present the best opportunities. Furthermore, our implementation meets all of our non functional requirements so it can still be deemed a met requirement. As for the original domain for the project I feel our implementation very successfully meets all the needs for the stakeholders. Our development team as a stakeholder was able to gain valuable development experience and also create a quality product. The users of our program will also have all their needs met and this was assured through testing. Additionally, all the expectations for our implementation have been met aside from the above stated exceptions. In terms of functional requirements it is evident that our program is exceptional in terms of its implementation. There is a very consistent input and output format throughout the modules, all the modules use the same method for reporting exceptions which are then handled by the main module. Lastly, the main philosophy in the design of our implementation was to keep non functional requirements in mind. By using algorithms such as merge sort, radix sort, binary search, and graphing algorithms, our project is very optimized and has a high performance level. Our team also ensured that by thorough unit testing all potential exceptions would be accounted for and handled in a very clean manner. Thus, ensuring correctness, reliability and robustness.

References

the code and info for Merge, Queue, MSD and BinarySearchST

Standard input and output libraries. (n.d.). Retrieved April 1, 2017, from http://algs4.cs.princeton.edu/code/