Software Specifications Document
ReLocate
version 3.2

COMP SCI 2XB3
Computer Science Practice and Experience: Binding Theory to Practice

Group 04
Madeeha Khan, Tasnim Noshin, Umme Salma Gadriwala,
Jenny Feng Chen, Patrick Laskowski

Department of Computer Science
McMaster University

April 11, 2017

Revisions

Contributions

Group Member	Role(s)	Contributions	Comments
Madeeha Khan	Team Leader, Tester, Scrum Team	Assigned tasks, followed up on progress	
		Organized, revised, and submitted project documents	
		Testing of final prototype	
Umme Salma Gardiwala	Scrum Master, Programmer, Log Admin	Created and maintained data structures	
		Organized modules	
		Logged and submitted project logs	
Tasnim Noshin	Programmer, Log Admin, GUI lead, Tester	Created GUI	
		Linked GUI to backend	
		Created and submitted meeting minutes	
Jenny Feng Chen	Programmer, Tester, Documenter	Wrote sorting algorithm	
		Implemented searching functionality	
		Debugged first prototype	
Patrick Laskowski	Programming lead, Testing lead	Wrote parser to extract data	
		Wrote graphing algorithm	

Executive Summary

Canada has a growing problem with involuntary part time worker and new immigrants unable to find work. Relocate aims to solve these problems by using open datasets to provide information on where somebody should move to find a job. Relocate gives users the option to search for a specific job, and optionally filter by province and a minimum median income for any cities Relocate recommends them to move to. Once Relocate is given these parameters, it will combine a dataset of projected job outlooks along with data regarding income statistics in Canadian cities to recommend to the user a city to move to that gives them the best chance of finding employment capable of covering their costs of living. For each city it recommends, Relocate provides the citys name, province, outlook ranking, a reason for the assigned ranking, and the citys average median income. Relocate is also capable of providing information on cities that are closely related to these cities based on outlook and income so that the user is capable of making a well informed decision on where to relocate to. These results are all saved to a text file on the users computer so that they may reference them again at any time without having to run the program again.

Module Decomposition

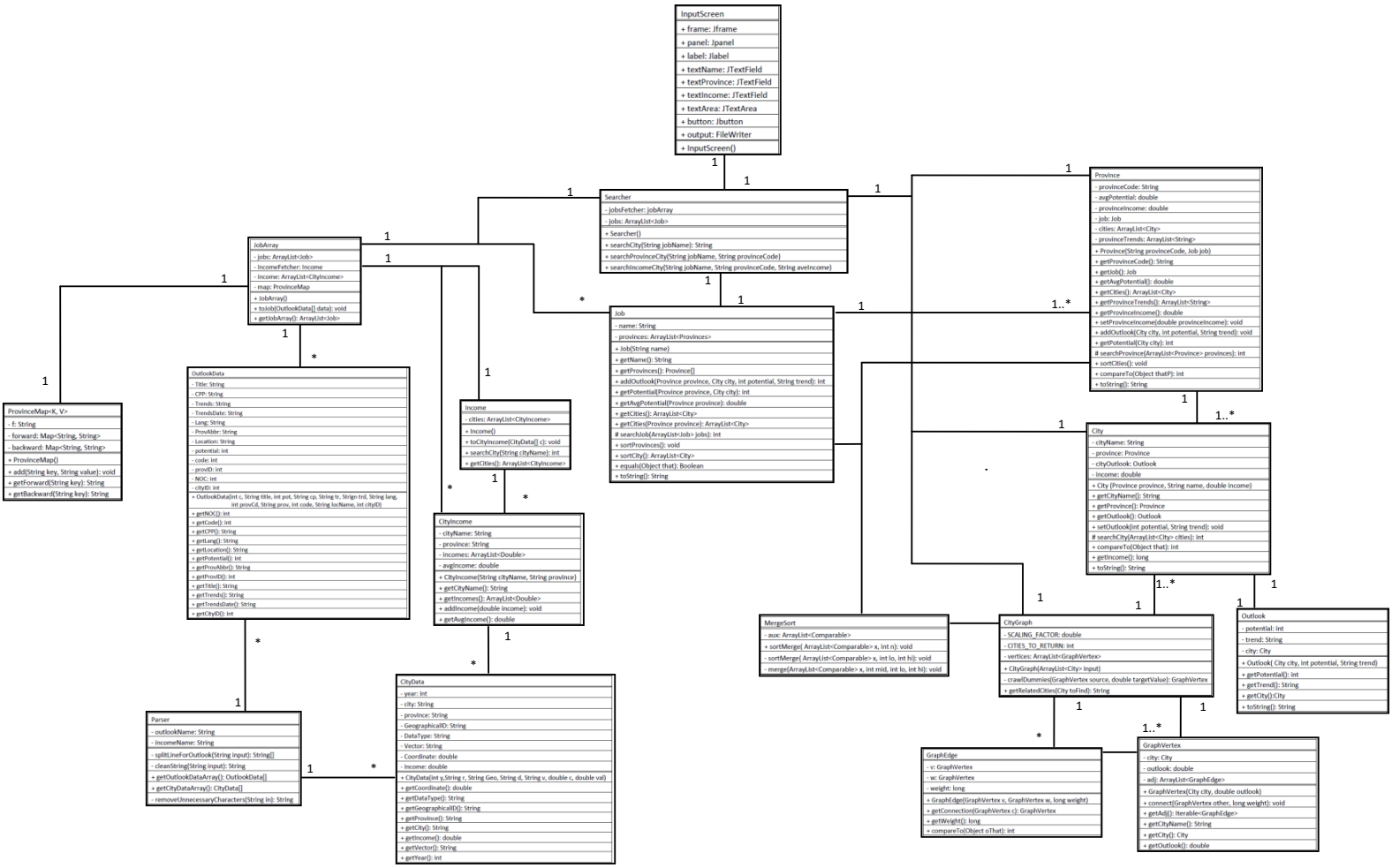
The aim of the module decomposition was to organize information in a logical way such that it is easily accessible and that the overall structure allows more information to be easily added into the program. We considered the principles of seperation of concerns and information hiding in our design so that the security of the data is protected and information can be accessed in a minimalistic fashion.

The following is a brief description of our classes, followed by a UML Classes Diagram to illustrate the relationship between our modules.

- City: This class represents a City, with fields identifying the name and the outlook.
- CityData: This class is an object represent each record of data in income.csv.
- CityGraph: This class will be used to create the actual undirected graph of the City.
- CityIncome: This class represents a city with income different from City object.
- GraphEdge: This class is used as an edge to connect two vertices.
- GraphVertex :This class will be used as the vertices of the graph.
- Income: This class transfers from a data type to a new data type by reorganizing the data and it holds an arrayList of the data.
- InputScreen: This class obtains inputs from a GUI and displays output in the console, as well as create a text file with the output data.
- Job: This class represents a Job, with fields for the name, and the provinces in which the job was available.
- JobArray: This class takes OutlookData objects and organizes them into the Job-Province-City-Outlook hierarchy for improved organization and sorting.
- MergeSort: This class sorts arraylists of Comparable objects using the merge sort algorithm.
- Outlook: This class represents an Outlook, with fields containing the potential and trend.
- OutlookData: This class is an object represent each record of data in outlook.csv.
- Parser: This class reads in data and creates CityData and OutlookData objects for each record in the data set.
- Province: This class represents a Province, with fields identifying the name, average potential, cities in the province and the job for which this Province object is an instance of.

- ProvinceMap: This class is a hash map that relates the province IDs (ON, BC, etc.) to the province names (Ontario, British Columbia, etc.).
- Searcher: This class takes in input information from the InputScreen class and searches through the Job-Province-City-Outlook hierarchy for results.

Figure 1: UML class diagram with relations. Figure is only visible from 250%+ zoom



Public Methods Description

City.java

This class represents a city.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
City	Province, String, Double	City	
getCityName		String	
getProvince		Province	
getOutlook		Outlook	
setOutlook	int, String		
searchCity	ArrayList<City>	int	
toString		String	
compareTo	Object	int	

Semantics

Access Routine Semantics

City($province, name, income$)

- This is a constructor that takes a province object, a city name and a median income of the city as inputs and assigns them to each of the field in city.

getCityName()

- This method returns the city name.

getProvince()

- This method returns the province object.

getOutlook()

- This method returns the outlook of the city.

setOutlook()

- This method set the outlook of the city.

searchCity(*cities*)

- This method search the city in the given list of cities. For each city in the given list, if it exists, returns the corresponding index else returns -1.
  Returns the index of the city in the list if found else returns -1.

toString()

- Creates a string concatinating cityName, provinceCode, cityOutlook and income.
  Returns the representation of the city as string.

compareTo(*thatC*)

- If the values of outlook of the two objects are equal return 0. If the value of outlook of this object is greater than the outlookk of the given object return 1, else return -1.

CityData.java

This class represents a city with corresponding data from dataset of income.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
CityData	int, String, String, String, String, double, double	CityData	
getCoordinate		double	
getDataType		String	
getGeographicalID		String	
getProvince		String	
getCity		String	
getIncome		double	
getVector		String	
getYear		int	

Semantics

Access Routine Semantics

CityData(y, r, Geo, d, v, c, val)

- This constructor takes a year, city name, province name, geographical ID, data type, vector, coordinates and income to assign each field of the CityData object.

getCoordinate()

- Returns the coordinate value of the data.

getDataType()

- Returns the type of the data as string.

getGeographicalID()

- Returns the Geographical ID as string.

getProvince()

- Returns the province name where the data was collected in.

getCity()

- Returns the city name where the data was collected in.

getIncome()

- Returns the income of the data.

getVector()

- Returns the vector representation as string.

getYear()

- Returns an integer holding the year the data was collected in.

CityGraph.java

This class is used to create the actual undirected graph of the city.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
CityGraph	ArrayList<City>	CityGraph	
getRelatedCities	City	String	
getRelatedCitiesTest	City	ArrayList<City>	

Semantics

Access Routine Semantics

CityGraph(*input*)

- This constructor takes an arrayList of type City to make the graph.
  Add all cities into the graph in the proper place by creating a new vertex to represent a city in the arraylist and add it to the graph. Then link each vertex to the correct vertex with income as the weight.

getRelatedCities(*toFind*)

- This method takes a City object and returns a few cities related to the given City object.
  Scan for the given city as vertex in the graph if not null, and if there are adjacent vertices for the given city hold the vertex else return null. If a vertex is hold, put all the edges into an arrayList excluding the original city. Sort the arrayList using merge sort by edges weight. Returns the sorted arrayList of cities as String.

getRelatedCitiesTest(*toFind*)

- This is a reduced version of the above function that is used for testing, and does essentailly the same, except it returns the ArrayList instead of its String represenation

CityIncome.java

This class represents a city with income from the income dataset.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
CityIncome	String, String	CityIncome	
getCityName		String	
getIncome		ArrayList<Souble>	
addIncome	double		
getAvgIncome		double	

Semantics

Access Routine Semantics

CityIncome(*cityName*, *province*)

- This constructor takes a city name and the province name which to assign for the city and province. It initializes an empty arraylist of type double for income and sets the average income as 0.

getCityName()

- Returns the city name.

getIncome()

- Returns an arrayList of income of type double.

addIncome(*income*)

- Takes an income and adds to the arrayList of income of the city and updates the average income of the city.

getAvgIncome()

- Returns the average income of the city.

GraphEdge.java

This class is used as an edge to connect two vertices.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
GraphEdge	GraphVertex, GraphVertex, double		
getConnection	GraphVertex	GraphVertex	
getWeight		double	
compareTo	Object	int	

Semantics

Access Routine Semantics

GraphEdge($v, w, weight$)

- This method takes two GraphVertex object and a double for the weight to constructs a GraphEdge by assigning them the corresponding value to the corresponding fields.

getConnection(c)

- It takes a GraphVertex object and checks whether the object is equal to either of the GraphVertex in the filed if not, returns null else return the the one that is equal.

getWeight()

- Return the weight of this edge as double.

compareTo($that$)

- If the weights of the two objects are equal return 0. If the weight of this object is greater than the weight of the given object return 1, else return -1.

GraphVertex.java

This class represents a vertex as a city, with its outlook and its adjacent vertices

Syntax

Access Programs

Routine name	Input	Output	Exceptions
GraphVertex	City, double	GraphVertex	
connect	GraphVertex, double		
getAdj		Iterable<GraphEdge>	
getCityName		String	
getCity		City	
getOutlook		double	

Semantics

Access Routine Semantics

GraphVertex($city, outlook$)

- This constructs a vertex by using a City object and the outlook of the City object. Creates an empty arrayList of adjacent edges.

getConnect($other, d$)

- It takes a GraphVertex object and the weight of the edge between this vertex and that vertex, then creates a GraphEdge object. Finally adds to the adjacency edges.

getAdj()

- Return the list of iterable adjacency edges.

getCityName()

- Returns the name of the city.

getCity()

- Returns the City object itself.

getOutlook()

- Returns the noutlook of the city.

Income.java

This class holds an arrayList of type CityIncome.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
Income		Income	
toCityIncome	CityData[]		
searchCity	String	int	
getCities		ArrayList<CityIncome>	

Semantics

Access Routine Semantics

Income()

- Initializes an empty arrayList and creates a CityData array from the parser of CityData. Constructs the arrayList of CityIncome by converting the CityData array to arrayList of City-Income. Thorws IOException when getting data from the Parser.

toCityIncome(c)

- For every city in *c*, it creates a new CityIncome object with that city's City and Province fields. if the city did not already exist in the array c, it adds the income to the city, and adds the city to the cities
  otherwise, it adds the income to the already existing city.
  If there is no city, and only a province, it creates a City and adds it at the end of the ArrayList

searchCity(*cityName*)

- Takes a city name and search for the city in the arrayList of the CityIncome. Returns -2 if input is null, returns the index where the city name is in the arraylist if found, else return -1 where city name is valid but not found in the arraylist.

getCities()

- Return the arrayList of type CityIncome

InputScreen.java

This class represents the GUI, where the user can enter the information they wish to find through the job search.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
InputScreen			IOException

Semantics

Access Routine Semantics

InputScreen()

- Constructor, which creates the JFrame with all the labels, panels, and buttons in it, and also keeps track of the actions (entering texts and pushing the button). The constructor is also where the search is performed and the results are written into the output file for the user.
  The search is performed by calling the methods from the Searcher class, depending on what the input is from the user:
  - If just a job name is given, the searchCity method is called on the job name given
  - If a job name and a province are given, the searchProvinceCity method is called on them

- If all three are specified, the searchIncomeCity method is called on them
- If an invalid combination (no job name, only a job name and an income), a not found message is displayed

There is an IOException thrown if there is an issue with the output file for whatever reason.

Job.java

This is the class representing a job.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
Job	String	Job	
getName		String	
getProvinces		Provinces[]	
addOutlook	Province, City, int, String		
getPotential	Province, City	int	
getAvgPotential	Province	double	
getCities		ArrayList<City>	
getCities	Province	ArrayList<City>	
searchJob	ArrayList<Job>	int	
sortProvinces			
sortCity		ArrayList<City>	
equals	Object	boolean	
toString		String	

Semantics

Access Routine Semantics

Job()

- The constructor, which takes a string (the name of the job), and creates an ArrayList of type Province to hold all the provinces the job is found in.

getName()

- returns the String, name of the job

getProvinces()

- returns an array of Provinces

addOutlook(*province*, *city*, *potential*, *trend*)

- the spot of the province is found in this.provinces (through a for-loop in the Provinces class) an outlook is added (called from the Province class) in that spot for that province if it exists in the arrayList
  if the province is not found in the list of provinces for that job (index = -1), it is added to the end of the list and an outlook is given to it (called from the Province class)

getPotential(*province*, *city*)

- the spot of the specified province is searched in this.provinces (from the Province class, in a for-loop) if the province is in the list, it finds the province in the list and gets the potential for the city in that province
  if the province is not in the list of provinces (index = -1), 0 is returned (since it means undetermined)

getAvgPotential(*province*)

- gets the average potential for that province if the province is in this.provinces otherwise, returns 0

getCities()

- returns all the cities that the job is in

getCities(*province*)

- returns all the cities the job is in, in a given province

searchJob(*jobs*)

- find the index of a job in the ArrayList of jobs by looping through the list and checking if the name of the job is in the list

sortProvinces()

- sorts the provinces the job is in according to the avgPotential with Mergesort

sortCity()

- sort the cities the job is in according to their potentials with Mergesort

equals(*that*)

- checks if 2 jobs are equal by first checking if they are of the same Class, then uses the built-in euqals method to check their equality

toString()

- returns a String representation of the Job object, which includes the name and all the provinces it is in

JobArray.java

This class creates the array of Jobs that will later be searched to match a job inputted by the user and find its information.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
JobArray		JobArray	IOException
toJob	OutlookData[]		
getJobArray		ArrayList<Job>	

Semantics

Access Routine Semantics

JobArray()

- The constructor initiliazes the instance variables, and also grabs an array of OutlookData[] from the Parser, which has scraped the csv file. It then takes this arrya and turns every entry in it into an instance of the Job class with the toJob method.

toJob(*data*)

- this method takes the OutlookData[] array from the constructor and for every entry *od* in the array, creates a new job with *od*'s title, and checks to see if the job already exists in the array If it does not, it gets the name of the province(s) that the job exists in from the ProvinceMap (the key is the abbreviation of *od*'s province from the csv file), then it finds the income of the cities that the job is in, as well as the avergae income for the province, and adds cities and outlook information for the job. The job is then added to the end of the array.
  If the job already existed in the array, it finds that job's position in the array, then adds all of the same information as above.

getJobArray()

- returns the job array

MergeSort.java

This class sorts an arrayList where extends the comparable type.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
sortMerge	ArrayList<Comparable>, int		

Semantics

Access Routine Semantics

sortMerge(x, n)

- This method takes an arrayList of Comparable type and the size of arrayList. It constructs an arrayList of the same size with each of the element initializing to null. Then merge sort the list.

Outlook.java

This class gives the outlook for a City, which is the combination of its potential and its trend.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
Outlook	City, int, String	Outlook	
getPotential		int	
getTrend		String	
getCity		City	
toString		String	

Semantics

Access Routine Semantics

Outlook($city, potential, trend$)

- this constructor initilizaes the instance variables to the given arguments to create a new instance of the Outlook class

getPotential()

- returns the int *potential*

getTrend()

- returns the String *trend*

getCity()

- returns the City object *city*

toString()

- returns a string representation of each Outlook object, with its potential, and trend.

OutlookData.java

This is the info scraped from the outlooks.csv file. Each instance of his class represents a line from that file.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
OutlookData	int, String, int, String, String, String, String, int, String, int, String, int	OutlookData	
getNOC		int	
getCode		int	
getCPP		String	
getLang		String	
getLocation		String	
getPotential		int	
getProvAbbr		String	
getProvID		int	
getTitle		String	
getTrends		String	
getTrendsDate		String	
getCityID		int	

Semantics

Access Routine Semantics

OutlookData $(c, title, pot, cp, tr, trd, lang, provCd, prov, code, locName, cityID)$

- The constructor creates a new instance of OutlookData by taking the given arguments and assigning them to instance variables.
  If the locName is the name of a province, it is nulled out, since we want that information for cities only.

getNoc()

- returns the int NOC

getCode()

- returns the int *code*

getCPP()

- returns the String CPP

getLang()

- returns the String *lang*

getLocation()

- returns the String *location*

getPotential()

- returns the int *potential*

getProvAbbr()

- return the String *provAbbr*

getProvID()

- returns the int *provID*

getTitle()

- returns the String *title*

getTrends()

- returns the String *trends*

getTrendsDate()

- returns th String *trendsDate*

getCityID()

- returns int *cityID*

Parser.java

This class takes the outlooks.csv and income.csv files and scrapes them, and creates arrays out of their information.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
getOutlookDataArray		OutlookData[]	IOException
getCityDataArray		CityData[]	IOException

Semantics

Access Routine Semantics

getOutlookDataArray()

- This method scrapes the outlooks.csv file and for each entry creates a new object of class OutlookData, then stores them in an array to be returned.
  It reads each line of the csv file (skipping the first line, since it has the headings), and intializes an array to the size of number of lines in the file. Then, it takes each row in the file and splits them with a private method to create the fields of an OutlookData object, and creates new objects with those fields.

getCity DataArray()

- This method scrapes the incomes.csv file and for each entry creates a new object of class CityData, then stores them in an array to be returned.
  First, an array is created to the size of number of lines in the file. Then, each row in the file is split and its components are used as arguments to create new CityData objects, which are then stored in the array.
  this method deals with blank lines and fills in '0' for them, as well as inappropriate types (if the file provides a String instead of an int, it is given an error value of int -1).

Province.java

This class represents a province, and contains all the information available about provinces from the combination of both datasets

Syntax

Access Programs

Routine name	Input	Output	Exceptions
Province	String, Job	Province	
getProvinceCode		String	
getJob		Job	
getAvgPotential		double	
getCities		ArrayList<City>	
getProvinceTrends		ArrayList<String>	
getProvinceName		double	
getProvinceIncome		double	
addOutlook	City, int, String		
getPotential	City	int	
SearchProvince	ArrayList<Province>	int	
sortCities			
compareTo	Object	int	
toString		String	
setProvinceIncome		double	

Semantics

Access Routine Semantics

Province($provinceCode, job$)

- This constructor initializes all the instance variables to create a new Province object

getProvinceCode()

- returns String $provinceCode$

getJob()

- returns Job object job

getAvgPotential()

- returns double $avgPotential$

getCities()

- returns ArrayList<City> $cities$

getProvinceTrends()

- returns ArrayList<String> *provinceTrends*

getProvinceName()

- returns double *provinceName*

getProvinceIncome()

- returns double *provinceIncome*

addOutlook(*city*, *potential*, *trend*)

- takes a *city*, *potential*, *trend* tuple and
  if the *city* name is null, it adds the trend to the end of the *provinceTrends*.
  Otherwise, it finds *city* in *cities* and if the city does not already exist, it creates an outlook for the city with the *trend* and *potential*, and adds it to *cities*.
  If the city already exists, it gets the city at the index that matches *city* in *cities*, gives it the *potential* and *trend*, and puts it in *cities* at the same index.

getPotential(*city*)

- This method finds *city* in *cities* and returns 0 if *city* is not there, returns the potential for *city* if it is.

searchProvince(provinces)

- For every province in *provinces*, it tried to match *provinceCode* to the entry in the ArrayList, and returns the index of it if it is found, and -1 if it is not.

sortCities()

- This method uses MergeSort to sort the *cities* by their potentials.

compareTo(*that*)

- Compares two Provinces byt their potentials, returns:
  - 2 if *this* and *that* are not provinces for the same job.
  - 1 if *this*'s potential is larger
  - 0 if they have equal potentials
  - -1 if *that*'s potential is larger

toString()

- Returns a String representation of the Province, with all its cities, provinceCode, average Potential, and provincial trends.

setProvinceIncome(*provinceIncome*)

- Sets the income to the specified value.

ProvinceMap.java

This class holds two maps which the key of one map is province name and value is the abbreviation whereas the other maps key is abbreviation and value is province name.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
ProvinceMap			FileNotFoundException
add	String, String		
getForward	String	String	
getBackward	String	String	

Semantics

Access Routine Semantics

ProvinceMap()

- Scan the file containing provinces and abbreviation and store the data into two maps. Each line consists of a province name and an abbreviation, split them by a comma.

add($key, value$)

- add the key and the value to each of the map

getCityAbbr(key)

- This method takes a string of province name and returns the corresponding abbreviation.

getCityName(key)

- This method takes a string of abbreviation and returns the corresponding province name.

Searcher.java

This class is where the searching happens. It has different methods for different kinds of searches, depending on what information the user gives, and also has reduced versions of the methods for testing. It throws an IOException because it has cennections to the Parser, which uses files.

Syntax

Access Programs

Routine name	Input	Output	Exceptions
Searcher		Searcher	IOException
searchCity	String	String	IOException
searchProvinceCity	String, String	String	IOException
searchIncomeCity	String, String, String	String	IOException
searchCityTest	String	int	IOException
searchProvinceCityTest	String, String	ArrayList<City>	IOException
searchIncomeProvinceTest	String, String, String	ArrayList<City>	IOException

Semantics

Access Routine Semantics

searchCity($jobName$)

- This method searches through the job array when the name of a job is given.
  it creates a new JobArray, then finds the index of the entry whose name matches $jobName$. If the job is not there, it will give -1, and the message "Job not found" is returned.
  An ArrayList of type City is created, where the cities that have the job in them are put, then sorted by their potential. Each entry in this list is then turned into its String representation and added to the result string. When all the cities have been added, the String is returned.

searchProvinceCity($jobName, provinceCode$)

- This method performs the same function as above, but only adds the city to the result String if its provinceCode matches the city's. Returns the first 5 things in the ArrayList, in String form.

searchIncomeCity($jobName, provinceCode, aveIncome$)

- This method takes a String $aveIncome$ and parses it to a double, then performs the same function as the method above, except the city is only added to the result String if the income is equal to or above the specified $aveIncome$. The result String is the first 5 entries in the ArrayList in String form.

searchCityTest($jobName$)

- This method was created for testing searchCity($jobname$). It returns the index of the job instead of its String representation.

searchProvinceCityTest($jobname, provinceCode$)

- This method was created for testing searchProvinceCity($jobname, provinceCode$). It returns the sorted ArrayList instead of its String representation.
  It would return an empty ArrayList if the job did not exist.

searchIncomeCity($jobname, provinceCode, aveIncome$)

- This method was created for testing searchIncomeCity($jobname, provinceCode, aveIncome$). It returns the sorted ArrayList instead of its String representation.
  It would return an empty ArrayList if the job did not exist.

TestGraph.java

This class tests the CityGraph and takes its input from a text file testText.txt

Syntax

Access Programs

Routine name	Input	Output	Exceptions
test			IOException

Semantics

Access Routine Semantics

test()

- This method tests the getRelatedCities method of the CityGraph class by running getRelatedCitiesTest() on an searchIncomeCity result from each line in the text file, then checking to see if the potentials of the related cities, if any, were equal to that of the given city.
  If the potentials were not equal, a boolean value was switched to false and so the assertTrue at the end of the method would fail the test.
  To handle jobs that did not have any related cities, the boolean value was switched back to true if the ArrayList returned by getRelatedCitiesTest() had size of 0, since that was the expected result.

TestSearch.java

This class tests the Searcher and takes its input from a text file testText.txt

Syntax

Access Programs

Routine name	Input	Output	Exceptions
jobTest			Exception
provinceTest			Exception
incomeTest			Exception

Semantics

Access Routine Semantics

jobTest()

- This method uses searchCityTest() on the first argument (the job name) of each line of the text file, and checks to see if the index of the job matches the expected outcome (which is also specified in the text file, which makes the tests more easily changed).
  If at any point the index does not match, a boolean value is switched to false, which will cause the assertTrue at the end of the test to fail the test.

provinceTest()

- This method uses searchProvinceCityTest() on the first and second arguments (the job name and province code) of each line in the text file, and checks to see if each city in the result arrayList from searchProvinceCityTest() has the same province code as each other, and as the one specified in the input for the search.
  If at any point the province does not match, a boolean value is switched to false, which will cause the assertTrue at the end of the test to fail the test.
  If the job does not exist, either in that province or at all, the boolean value is switched back to true if the size of the result ArrayList is 0, since that is expected.

incomeTest()

- This method uses searchIncomeCityTest() on all three of the arguments (job name, province code, income) of each line of the text file, and checks to see if the income of any city in the result array is less than the income specified in the text file.
  If at any point the income is less, a boolean value is switched to false, which will cause the assertTrue at the end of the test to fail the test.
  If the job does not exist, either in that province or income range, or at all, the boolean value is switched back to true if the size of the result ArrayList is 0, since that is expected.

Uses Relationship

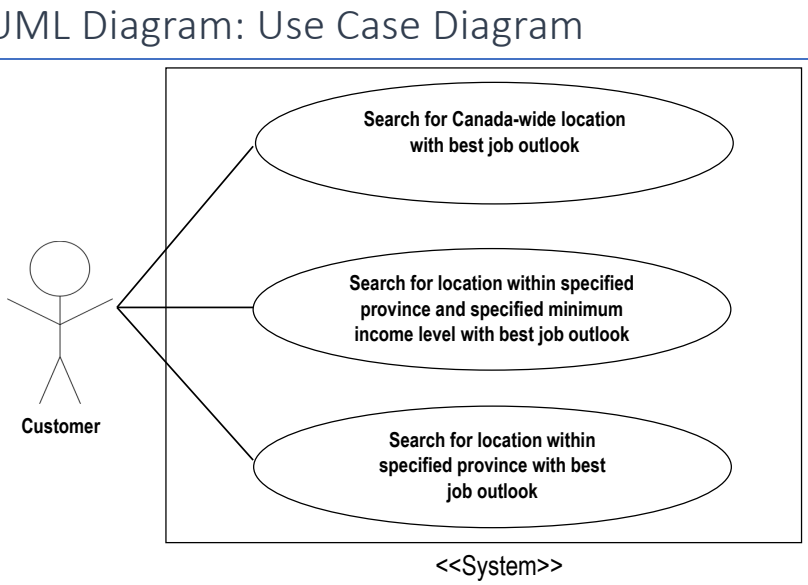


Figure 2: Uses Case Diagram for ReLocate product.

Traceability Matrix

Requirement	**Module(s)**
Upon entering the application, user should be greeted with a GUI asking them to input their search filters, the user must input a job name they are interested in searching for, but optionally they may input a province code (ON,BC, etc.) and a minimum average income for the city.	InputScreen
If the user does not wish to filter by province code or income, they may leave the fields untouched to signify this	InputScreen
Once the user has completed constructing their search query, they will press a button labelled Search to begin the search	InputScreen
The program will read the datasets in and convert them to objects for easier access	Parser, OutlookData, CityData
The program will search according to the users criteria and assign outlooks and income to each city	City, Income, Job, JobArray,Outlook, Province, ProvinceMap, Searcher
The program will rank the cities according to their outlook	MergeSort, Outlook, City
The program will also find cities related to the current city via graphing algorithm	CityGraph, GraphEdge, GraphVertex
When the search is complete, the results of the search will be printed out to the console. These results will consist of the city name, its province, a numerical ranking for the jobs outlook in this city, the reasoning for this outlook (pulled from the dataset), and the average median income of the city	InputScreen

Private Methods Description

City.java

Semantics

State (Instance) Variables

private final String *cityName*
private Province *province*
private Outlook *cityOutlook*
private double *income*

CityData.java

Semantics

State (Instance) Variables

private final int *year*
private final String *city*
private final String *province*
private final String *GeographicalID*
private final String *DataType*
private final String *Vector*
private final double *Coordinate*
private final double *Income*

CityGraph.java

Syntax

Access Programs

Routine name	Input	Output	Exceptions
crawlDummies	GraphVertex, double	GraphVertex	

Semantics

State (Instance) Variables

private final double $SCALING\_FACTOR = 1$
private final int $CITIES\_TO\_RETURN = 2$
private ArrayList<GraphVertex> *vertices*

Access Routine Semantics

crawlDummies(GraphVertex source, double targetValue)

- This method takes a GraphVertex source and a double targetValue as input and crawls up the graph to find the node to link an outlook to. If the source cannot be linked to an outlook, the method looks for a link recursively in vertices with value -1 which are adjacent to the source. Once it finds a match the appropriate GraphVertex is returned.

CityIncome.java

Semantics

State (Instance) Variables

private final String *cityName*
private String *province*
private ArrayList<Double> *incomes*
private double *avgIncome*

GraphEdge.java

Semantics

State (Instance) Variables

private GraphVertex v
private GraphVertex w
private double *weight*

GraphVertex.java

Semantics

State (Instance) Variables

private City *city*
private double *outlook*
private ArrayList<GraphEdge> *adj*

Income.java

Semantics

State (Instance) Variables

ArrayList<CityIncome> *cities*

Job.java

Semantics

State (Instance) Variables

private final String *name*
private ArrayList<Province> *provinces*

JobArray.java

Semantics

State (Instance) Variables

ArrayList<Job> *jobs*
Income *incomeFetcher*
ArrayList<CityIncome> *income*
ProvinceMap *map*

MergeSort.java

Syntax

Access Programs

Routine name	Input	Output	Exceptions
sortMerge	ArrayList<Comparable>, int, int		
merge	ArrayList<Comparable>, int, int, int		

Semantics

State (Instance) Variables

private static ArrayList<Comparable> *aux*

Access Routine Semantics

sortMerge(ArrayList<Comparable> x, int lo, int hi)

- This method takes an ArrayList<Comparable> x, an int lo and an int hi as input. If hi is less than equal to lo then it ends the method. Otherwise, recursively splits and sorts the arrays. Then calls the method merge to merge the two halves together.

merge(ArrayList<Comparable> x, int lo, int hi, int mid)

- This method takes an ArrayList<Comparable> x, an int lo, an int hi and an int mid as input. This method uses extra memory to sort the array. First it creates an auxiliary array within the range of lo to hi and copies values in that range from x. Then within that range, it sorts the original array in ascending order.

Outlook.java

Semantics

State (Instance) Variables

private int *potential*
private String *trend*
private City *city*

OutlookData.java

Semantics

State (Instance) Variables

private final String $Title, CPP, Trends, TrendsDate, Lang, ProvAbbr, Location$
private final int $potential, code, provID, NOC, cityID$

Parser.java

Syntax

Access Programs

Routine name	Input	Output	Exceptions
splitLineForOutlook	String	String[ ]	
cleanString	String	String	
removeUnnecessaryCharacters	String	String	

Semantics

State (Instance) Variables

private final String $outlookName$ = "data/outlooks.csv"
private final String $incomeName$ = "data/income.csv"

Access Routine Semantics

splitLineForOutlook(String input)

- This method takes a String input, which is input data from outlook.csv. This data holds all the information in single line. splitLineForOutlook method takes each line and returns an array of Strings with properly split data.

cleanString(String input)

- This method takes a String input as input and returns a String removing all HTML elements.

removeUnnecessaryCharacters(String in)

- This method takes a String in as input and returns a String removing unnecessary leading and tailing spaces and quotes.

Province.java

Semantics

State (Instance) Variables

private final String *provinceCode*
private double *avgPotential*
private double *provinceIncome*
private Job *job*
private ArrayList<City> *cities*
private ArrayList<String> *provinceTrends*

ProvinceMap.java

Semantics

State (Instance) Variables

private final String f = "data/provinces.txt"
private Map<String,String> $forward$ = new Hashtable<String,String>()
private Map<String,String> *backward* = new Hashtable<String,String>()

Searcher.java

Semantics

State (Instance) Variables

JobArray *jobsFetcher*
ArrayList<Job> *jobs*

Review of Design

Elements of the design that were done well:

- There is a clear separation of concerns in the data structures - each object represents one particular piece of information such as a job or a city, and stores the information related to that object. This preserves encapsulation.
- The graph ADT is well designed - it uses the features provided by the object oriented programming paradigm to turn every piece of the graph into its own object and store data relevant to its operations inside these objects.
- Implementing MergeSort in its own class was a good idea, this allowed it to be re-used by several other classes which needed sorting capabilities
- The design is extendable, more features can be added by adding classes which implement the necessary functionality, and then using the searcher class to pass the list of cities being considered to the new class

Elements of the design that were done poorly:

- There were more data structures than necessary - rather than having the parser go from the datasets to a CityData object to finally a City object, it would have been better to make it go directly to a City object.
- A lot of the data structures were made into separate objects when it was not really necessary. For instance, rather than making a citys Outlook an entirely separate entity, it would have been better to directly include the relevant information in City because in reality the outlook is a property of the city
- The flow of data inside the program is confusing, in an ideal situation the Searcher would have handled creating all data structures but instead parts of the data are populated in unusual places. For instance, a citys outlook is calculated and placed inside the City object inside the JobArray class, which does not make much sense and should have been done inside Searcher instead.