Software Design Document

Project Vayu

Lab 2, Group 5

Revision: 0.2

11 April, 2020

Oussama Saoudi, Lennon Yu
Christina Korsman, Diego Soriano

SFWRENG 2XB3
Software Engineering Practice and Experience:
Binding Theory to Practice
Department of Computing and Software
McMaster University

Change History

Revision	Date	Changes
V0.1	27 Feb 2020	Added skeleton
V0.2	11 April 2020	Refining details with updated content

Team Members

Name	Student No.	Role
Oussama Saoudi	400172153	Project Lead, Search Alg. Dev.
Lennon Yu	400183521	Doc. Maintainer, Graph Alg. Dev.
Christina Korsman	400192880	Transcriber, FileIO. Dev
Diego Soriano	400172910	Doc. Maintainer, Sort Alg. Dev.

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 SE-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

Contributions

Name	Roles	Contributions	Comments
Oussama Saoudi	Project Lead, Developer	DisasterAreaBuilder, DisasterArea, KdTree, ConvexHull Builder, Quicksort Refinement, Stack, CCFinder Refinement, DisasterAreaBuilder State Machine	Made significant refinements to modules to improve performance
Lennon Yu	Documentation Maintainer, Developer	Graph, CCFinder First Implementation CommandlineController Proofreading SDD MIS	
Christina Korsman	Transcriber, Developer	WeatherTypeEnum, FileOutut, Parse, DisasterType, Meeting Minutes	
Diego Soriano	General Programmer, Developer	ByDamage, ByProximity, Quicksort First Implementation, SRS, SDD	

Summary

Project: Vayu is a system which parses NCDC Storm and Weather Events Database to provide a platform on which users can interact with the weather data. This is facilitated through sorting of data points by casualties or by property damage. In addition, the system computes disaster areas which represent areas affected by various weather and disasters. The areas are in the form of sets of points enclosing regions which have high occurrence of a certain weather type. The disaster areas and sorting allow users such as governments or relief organizations to discern useful information about regions affected by certain weather trends or disasters.

Contents

1 SDD Identification

1.1 Scope

The developed product is a program that can read an input set of data from NCDC Storm Events Database, process and translate it into a usable format. The program will be able to sort the given set of disaster data on different parameters, such as casualty count or property damage costs. From this, a graph will be created and used to write a convex hull representation of the disaster trends. This information displayed in this format would allow for a better understanding and prediction of natural disaster trends around the globe. The end product aims to serve governments/relief organizations with natural disaster data presented in an easier to process format to help predict further trends.

1.2 Purpose

The purpose of this document is to provide a description of the software's design and allow a basis from which the software's goals can be referenced to during it's development process.

1.3 Intended Audience

The intended audience of this software design descriptions document are governments and relief organizations.

1.4 References

[1]Cengproject.cankaya.edu.tr, 2020. [Online]. Available: http://cengproject.cankaya.edu.tr/wp-content/uploads/sites/10/2017/12/SDD-ieee-1016-2009.pdf. [Accessed: 22- Mar- 2020].

1.5 Context

The purpose of this software design description is to outline what the software is intended to accomplish as well as the design details of the project as a whole.

1.6 Design Languages

The Design language that will be use is UML.

2 Design Stakeholders

The stakeholder of the design subject with respective design concerns are the following:

- Governments
  - Disaster Regions
  - Casualties
- Non-profit Organizations
  - Disaster Regions
  - Casualties
- Insurance Companies
  - Property Damage

3 Design Viewpoints

3.1 Context viewpoint

The context viewpoint of this software depicts all the services provided by the program.

3.1.1 Design concerns

The program aims to provide a way to process and present the given input data of natural disasters from the supplied database. Its primary users would be governments and relief organizations, who would analyze trends from the processed data in the use of better natural disaster trend predictions.

3.1.2 Design entities

The external active elements that the system will be working with, is the user and the data set.

3.1.3 Design relationships

The system will receive location data , or filter data from the user. With this input the system will out the severity and disaster from the surround area. if applicable the filter on the type of disaster in that area.

3.1.4 Design Constraints

Users interact with it through command line . In terms of quality, a user review is to be given to users after interaction with the program, and an overall positive rating of at least 75% must be achieved, in accordance with the SRS.

3.2 Composition viewpoint

The composition viewpoint outlines the many components of the software and its subsystems. In-depth explanation of these components and the relations between them will be covered in later sections.

3.2.1 Design concerns

The design of this programs is structured into modules and sub-modules that interact between different packages. The project as a whole is managed by controlling, editing, and monitoring these components, such as KD-tree, Quicksort, and ConvexHull. Each main function of the program is itself divided into smaller packages, such as those for sorting, searching, and graph management. As a whole, the overall program can be subdivided to equally distribute work amongst the team.

3.2.2 Design entities

A full list of design entities is shown in section 3.3.

3.2.3 Design Relationship

The design relationship of the program's modules is further explained in section 3.4.

3.3 Interface viewpoint

The following viewpoint is organized to list the internal and external interfaces of the several modules in the program.

ByCasualties Module

Template Module inherits Comparator<DisasterPoint>

ByCasualties

Uses

None

Syntax

Exported Constants

None

Exported Types

ByCasualties = ?

Exported Access Programs

Routine Name	In	Out	Description
new ByCasualties		ByCasualties	Constructs a new ByCasualties comparator
compare	DisasterPoint, DisasterPoint	int	Compare two DisasterPoints by their casualty numbers, return a negative/positive integer or zero if the first DisasterPoint's casualty is less/greater than or equal to that of the second DisasterPoint.

Semantics

State Variables

None

State Invariant

None

Local Modules

None

Local Functions

None

Design Concerns

ByCasualties allows for comparison between two DisasterPoints relative to the number of casualties stored in each DisasterPoint. This fulfills [FR1.2] which allows for the list of affected areas sorted by casualties.

ByDamage Module

Template Module inherits Comparator<DisasterPoint>

ByDamage

Uses

None

Syntax

Exported Constants

None

Exported Types

ByDamage = ?

Exported Access Programs

Routine Name	In	Out	Description
new ByDamage		ByDamage	Constructs a new ByDamage comparator
compare	DisasterPoint, DisasterPoint	int	Compare two DisasterPoints by their property damage, return a negative/positive integer or zero if the first DisasterPoint's property damage is less/greater than or equal to that of the second DisasterPoint.

Semantics

State Variables

None

State Invariant

None

Local Modules

None

Local Functions

None

Design Concerns

This cover the functional requirements [FR 1.1] by comparing the damage of the given area around a point.

Disaster Point Module

Template Module

DisasterPoint

Uses

WeatherTypeEnum

Syntax

Exported Constants

None

Exported Types

DisasterPoint = ?

Exported Access Programs

Routine Name	**In**	**Out**	**Description**
new DisasterPoint	$\mathbb{N}$	DisasterPoint	Constructs a new DisasterPoint with a specified id
new DisasterPoint	$\mathbb{N}, \mathbb{R}, \mathbb{R}$	DisasterPoint	Constructs a new DisasterPoint with a specified id, latitude and longitude
setLat	$\mathbb{R}$		Sets the DisasterPoint's latitude
setLon	$\mathbb{R}$		Sets the DisasterPoint's longitude
setType	WeatherTypeEnum		Sets the DisasterPoint's disaster type
setCas	$\mathbb{N}$		Sets the DisasterPoint's causalities
setYear	$\mathbb{N}$		Sets the DisasterPoint's year of occurrence
setPropertyDam	$\mathbb{N}$		Sets the DisasterPoint's property damage
getId		$\mathbb{N}$	Gets the DisasterPoint's ID
getLat		$\mathbb{R}$	Gets the DisasterPoint's latitude
getLon		$\mathbb{R}$	Gets the DisasterPoint's longitude
getWeatherType		WeatherTypeEnum	Gets the DisasterPoint's disaster type
getCasualties		$\mathbb{N}$	Gets the DisasterPoint's casualties
getPropertyDamage		$\mathbb{N}$	Gets the DisasterPoint's property damage
getYear		$\mathbb{N}$	Gets the DisasterPoint's year of occurrence
hashCode		$\mathbb{Z}$	Returns the hashcode of this DisasterPoint
equals	DisasterPoint	$\mathbb{B}$	Checks if two DisasterPoint are equal by ID
toString		String	Output the contents of DisasterPoint to a String

Semantics

State Variables

$latitude : \mathbb{R}$
$longitude : \mathbb{R}$
$disasterType :$ WeatherTypeEnum
$year : \mathbb{N}$
$casualties : \mathbb{N}$
$propertyDamage : \mathbb{N}$
$id : \mathbb{N}$

State Invariant

None

Local Modules

None

Local Functions

None

Design Concerns

The DisasterPoint module is the most basic data storage entity that is used by all modules.

KdTree Module

Template Module

KdTree

Uses

DisasterPoint, Stack<T>

Syntax

Exported Constants

None

Exported Types

KdTree = ?

Exported Access Programs

Routine Name	In	Out	Description
new KdTree		KdTree	Contructs a new kd-tree
insert	DisasterPoint		Inserts a DisasterPoint into the kd-tree
nearestPoint	DisasterPoint	DisasterPoint	Finds the Nearest point in the KDTree to the given query point and return that point
closePoints	DisasterPoint, $\mathbb{R}$	Iterable<DisasterPoint>	Finds all points that are within a certain radius rad from the given query point

Semantics

State Variables

root : KdTree.Node

State Invariant

None

Local Modules

KdTree.Node DisasterPoint object representing node in the kd binary tree with representative point in 2d space, an orientation, showing if its dividing line is horizontal or vertical. DisasterPoint also has a left or right node, which could also represent up or down if its line is horizontal. RectA inside node represents the Rectangle it sits in and divides.

KdTree.RectA Rectangular Area data type which represents area in 2D space with minimum and maximum x and y values.

Local Functions

Node insert(Node node, DisasterPoint p, boolean orientation, double xmin, double ymin, double xmax, double ymax):

- Inserts the DisasterPoint p at the kdTree by analyzing the node and orientation to find placement of the point. xmin, ymin, xmax, and ymax represent the bounds of the area being checked by the kdtree for the position to place the point. When the new node is made with its associated point, it will occupy a RectA area with bounds xmin, ymin, xmax, and ymax.

int size(Node n):

- Returns the size of the node's subtree if it is not null, else returns 0

int compareX(DisasterPoint p1, DisasterPoint p2):

- Compares the x value between two DisasterPoints p1 and p2. Returns 1 if p1 has greater x value, returns -1 if p2 has greater value, and 0 otherwise.

int compareY(DisasterPoint p1, DisasterPoint p2):

- Compares the y value between two DisasterPoints p1 and p2. Returns 1 if p1 has greater y value, returns -1 if p2 has greater value, and 0 otherwise.

DisasterPoint nearestPoint(Node node, DisasterPoint query, DisasterPoint nearestPoint):

- Finds the nearest point in the kdTree to the query DisasterPoint.

void range(Node node, RectA rect, Stack<DisasterPoint> stack):

- Finds all DisasterPoints existing in the RectA rectangle passed to the method

Design Concerns

It assists in creating connections to the graph, which facilitates the creation of disaster regions for functional requirements [FR2.1] and [FR5.3]

Quicksort Module

Module

Quicksort

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine Name	**In**	**Out**	**Description**
sort	List<T> , Comparator<T>	ArrayList<T>	Copies and sorts list of type T, and returns the copy. Uses comparator c to compare values in the list.
sort	List<T> , $\mathbb{N}, \mathbb{N}$, Comparator<T>	ArrayList<T>	Copies and sorts list of type T with a given starting and ending index, and returns the copy. Uses comparator c to compare values in the list.
quicksort	ArrayList<T>, $\mathbb{N}, \mathbb{N}$,Comparator<T>		Performs quicksort by partitioning the list and recursively sorting the sub lists. Modifies the given list directly.

Semantics

State Variables

State Invariant

None

Local Modules

None

Local Functions

int partitionRandom(ArrayList<T> list, int lo, int hi, Comparator<T> c):

- Switches the first value in a sublist with a random value in that sublist to improve sorting performance

int partition(List<T> list,int lo, int hi, Comparator<T> c):

- Partitions values in a list into values less than the first item in the list and elements greater than the the first element

void exch(List<T> list, int i1, int i2):

- Exchanges two values in a list at indexes i1 and i2

Design Concerns

It fulfills [FR 1.1] which enables sorting all disaster points either by property damage or casualties.

Connected Component Finder Module

Template Module

CCFinder

Uses

Graph

Syntax

Exported Constants

None

Exported Types

CCFinder = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new CCFinder	Graph	CCFinder	Constructs a new Connected Component Finder
connected	$\mathbb{N}, \mathbb{N}$	$\mathbb{B}$	Checks if two vertices are in the same component (i.e. a path exists between two vertices)
componentCount		$\mathbb{N}$	Gets the number of existing connected components. One may assume that all component id's are less than this value.
getComponentById	$\mathbb{N}$	HashSet<Integer>	Gets a connected component by its id
toString		String	String representation of CCFinder

Semantics

State Variables

componentCount: $\mathbb{N}$
marked: sequence of $\mathbb{B}$
id: sequence of $\mathbb{N}$
components: ArrayList<HashSet<Integer> >

State Invariant

None

Local Modules

None

Local Functions

dfs(Graph G, int s):

- Performs depth first search in the graph G starting from source s. Adds all the explored points to a connected component

validPoint(int v):

- Checks if the passed integer vertex is in the graph. If it isn't throws IllegalArgumentException.

Design Concerns

Required by DisasterAreaBuilder module to construct proper convex hulls [FR 2.1].

Graph Module

Template Module

Graph

Uses

None

Syntax

Exported Constants

None

Exported Types

Graph = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new Graph	$\mathbb{N}$	Graph	Constructs a new graph
addEdge	$\mathbb{N}, \mathbb{N}$		Connect two vertices in the graph with a new undirected edge.
adj	$\mathbb{N}$	Iterable<Integer>	Gets the adjacent vertices of a specific vertex
V		$\mathbb{N}$	Gets the number of vertices in this graph
E		$\mathbb{N}$	Gets the number of edges in this graph
degree	$\mathbb{N}$	$\mathbb{N}$	Gets the number of edges of a specific vertex
toString		String	Gets the string representation of this graph

Semantics

State Variables

$V : \mathbb{N}$
$E : \mathbb{N}$
adj : ArrayList<Integer>[]

State Invariant

None

Local Modules

None

Local Functions

void validPoint(int v):

- Checks if the passed integer vertex is in the graph. If it isn't throws IllegalArgumentException.

Consideration

This module is required by the ConvexHullBuilder and KdTree modules in the completion of their functional requirements [FR 2.1]

Weather Type Enum Module

Template Module

WeatherTypeEnum

Uses

None

Syntax

Exported Constants

None

Exported Types

DisasterTypes = {Astronomical Low Tide, Avalanche, Blizzard, Coastal Flood, Cold/Wind Chill, Debris Flow, Dense Fog, Dense Smoke, Drought, Dust Devil, Dust Storm, Excessive Heat, Extreme Cold/Wind Chill, Flash Flood, Flood,Frost/Freeze, Funnel Cloud, Freezing Fog, Hail, Heat, Heavy Rain, Heavy Snow, High Surf, High Wind, Hurricane(Typhoon),Ice Storm, Lake-Effect Snow, Lake shore Flood,Lighting, Marine Hail, Marine High Wind, Marine Strong Wind, Marine Thunder Wind, Rip Current, Seiche, Sleet, Storm Surge/Tide, Strong Wind, Thunderstorm Wind, Tornado, Tropical Depression, Tropical Storm Tsunami, Volcanic Ash, Waterspout , Wildfire, Winter Storm, Winter Weather }

Exported Access Programs

Routine name	In	Out	Description
new WeatherTypeEnum	String	WeatherTypeEnum	Constructs WeatherType-Enum from associated text.
getText		String	Returns the associated name of the WeatherType-Enum
fromString	String	WeatherTypeEnum	Returns the WeatherType-Enum associated with the given text if it exists, else return null.

Semantics

State Variables

None

State Invariant

None

Local Modules

None

Local Functions

None

Considerations

The types enumerates over all the disaster types in the input data file. It is required by all modules to select a particular disaster type in real life.

Disaster Area Module

Template Module

DisasterArea

Uses

DisasterPoint, WeatherTypeEnum

Syntax

Exported Constants

None

Exported Types

DisasterArea = ?

Exported Access Programs

Routine name	In	Out	Description
new DisasterArea	ArrayList<DisasterPoint>, ArrayList<DisasterPoint>, WeatherTypeEnum	DisasterArea	Constructor for DisasterArea which takes in the area's convex hull, all the DisasterPoints contained within the convex hull, and the shared weather type of all the nodes
getType		WeatherTypeEnum	Getter for the WeatherTypeEnum of the DisasterArea
getHull		ArrayList<DisasterPoint>	Gets the convex hull of the DisasterPoints contained in the DisasterArea
getAllNodes		ArrayList<DisasterPoint>	Gets all the nodes contained in the DisasterArea

Semantics

State Variables

$convexHull$: ArrayList<DisasterPoint>
$allNodes$: ArrayList<DisasterPoint>
$disasterType$: WeatherTypeEnum

State Invariant

None

Local Modules

None

Local Functions

None

Consideration

This module is used to cover the functional requirements [FR2.1]

Disaster Area Builder Module

Template Module

DisasterArea

Uses

DisasterPoint, WeatherTypeEnum, DisasterArea, Graph, CCFinder, KdTree, Convex-HullBuiler

Syntax

Exported Constants

None

Exported Types

DisasterAreaBuilder = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new DisasterAreaBuilder	Hashtable<Integer,Integer>, ArrayList<DisasterPoint>, $\mathbb{R}, \mathbb{Z}$	DisasterAreaBuilder	Builds all the disaster areas on the map. The radius rad is for search of nearest nodes when connecting close DisasterPoints. The minimum size of a disaster area can be specified using the method parameter thresh
getAreas	WeatherTypeEnum	ArrayList<DisasterArea>	Returns list of all the DisasterAreas which have a the passed weather type type.

Semantics

State Variables

kdtList: Hashtable<WeatherTypeEnum, KdTree>
pointList: ArrayList<DisasterPoint>

areaList: HashMap<WeatherTypeEnum,ArrayList<DisasterArea> >

State Invariant

None

Local Modules

None

Local Functions

ArrayList<DisasterPoint> convertToPoint(Iterable<Integer> iterable):

- Converts an iterable of integers into a list of DisasterPoints, using the integers as indices in the arrays and returns the list.

Consideration

This module is used to cover the functional requirements [FR2.1]

File Parser Module

Template Module

Parser

Uses

WeatherTypeEnum, DisasterPoint

Syntax

Exported Constants

Exported Types

Parser = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new Parser		Parser	Initializes parsing of the input files and creation of DisasterPoints
detailsParser	String		Parses given file with name fileName, and creates DisasterPoints from data. Adds all the created DisasterPoints to a the nodeList
getData			Returns the ArrayList of DisasterPoints created and stored by the parser
getTable			Getter for the hash table constructed by parser to map id of Disasterpoint to its index

Semantics

State Variables

lookup: Hashtable<Integer,Integer>
nodelist: ArrayList<DisasterPoint>

State Invariant

None

Local Modules

None

Local Functions

void parse():

- Sets the base directory for all the files to be parsed

void getFiles(File dir):

- Gets all the files in the given directory dir and runs the detailsParser on files that are not directories. If a directory is contained in dir, it recursively searches by called getFiles on that directory.

void detailsParser(String fileName):

- Parses given file with name fileName, and creates DisasterPoints from data. Adds all the created DisasterPoints to a the nodeList

int sumCasualties(String injuryD, String injuryI, String deathD, String deathI):

- Takes 4 strings which are each representing integers for injuries and deaths, converts them to integer, and returns the sum of all the integers.

int damageParse(String pDamage, String cDamage):

- Takes string representation of property damage and crop damage and converts to Double, then returns their sum

Consideration

This module converts data entries in ASCII character based CSV files to corresponding programming entities (i.e. DisasterPoint module instances).

File Output Module

Template Module

FileOutput

Uses

DisasterPoint, DisasterArea

Syntax

Exported Types

FileOutput = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new FileOutput			Initializes the FileOutput and creates the line separator which separates the DisasterPoints when being printed
writeData	String, ArrayList<DisasterPoint>		Writes data to a file with the same name as the String. Every line on that file is a new DisasterPoint, and in each line, separated by commas, are the attributes of that DisasterPoint.
writeAreas	String, ArrayList<DisasterArea>		Writes Data to a file with the same name as the String. Every line on that file is a new DisasterArea, and in each line, separated by commas, are the attributes of the WeatherTypeEnum.

Semantics

State Variables

sep: String
numOfFields: $\mathbb{N}$
sizeOfField: $\mathbb{N}$

State Invariant

None

Local Modules

None

Local Functions

None

Design Concerns

This module takes the outputs from the Quicksort module and inputs it in a textfile for the users to use. This module fulfills the functional requirements [FR3.1] and [FR3.2] by outputting the information into a text file with CSV (Comma Separated Value) in the file.

Convex Hull Builder Module

Module

ConvexHullBuilder

Uses

DisasterPoint, Quicksort, Stack<T>

Syntax

Exported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Description**
convexHull	ArrayList<DisasterPoint>	ArrayList<DisasterPoint>	Computes the convex hull of the list of points provided. ie; it computes the smallest set of points when if connected as a polygon contain all the points in the input set DisasterPoints

Semantics

State Variables

None

Local Modules

ConvexHullBuilder.PointComparator A Comparator for comparing points based off angle to a given anchor DisasterPoint. Used in performing Graham Scan algorithm.

Local Functions

DisasterPoint nextToTop(Stack<DisasterPoint> pointStack):

- Returns the second to the top item in the pointStack. The item has a type DisasterPoint.

void exch(ArrayList<DisasterPoint> disasterPoints, int p1, int p2):

- Exchanges Points in the two array positions array[p1] and array[p2]

int orientation(DisasterPoint p1, DisasterPoint p2, DisasterPoint p3):

- Checks if points p1, p2, and p3 have a clockwise, collinear, or counterclockwise orientation. If they are in a counter clock-wise orientation, returns 2, clock-wise returns 1 and collinear returns 0.

Considerations

This module fulfills functional requirement [FR 2.1].

Stack Module

Generic Template Module inherits Iterable<T>

Stack<T>

Uses

None

Syntax

Exported Constants

None

Exported Types

Stack<T> = ?

Exported Access Programs

Routine name	In	Out	Description
new Stack		Stack	Constructs a new empty stack
iterator		Iterator<T>	Returns iterator for all values in the stack
isEmpty		$\mathbb{B}$	Returns true if stack is empty and false otherwise.
size		$\mathbb{N}$	returns number of values in the stack
push	T		Pushes item to the top of the stack
pop		T	Deletes the item on the top of the stack and returns it
top		T	Returns the top item in the stack without popping it

Semantics

State Variables

first: Node
numberOfValues: $\mathbb{N}$

State Invariant

None

Local Modules

Stack.ListIterator Implements an iterator for Items, which allows for proper iteration through the stack structure.

Stack.Node Node class which stores generic item and next node.

Local Functions

None

Consideration

This module is required by the ConvexHullBuilder and KdTree modules in the completion of their functional requirements.

Considerations

The Stack<T> module is used by several other modules to provide the most rudimentary algorithmic functionalities.

CommandlineController Module

Template Module

CommandlineController

Uses

Parser, FileOutput, DisasterAreaBuilder, WeatherTypeEnum, Quicksort

Syntax

Exported Constants

None

Exported Types

CommandlineController = ?

Exported Access Programs

Routine name	**In**	**Out**	**Description**
new CommandlineController		CommandlineController	
start			Starts the user interface

Semantics

State Variables

parser: Parser
DAbuilder: DisasterAreaBuilder
outputter: FileOutput

State Invariant

None

Local Modules

None

Local Functions

void performNextCommand():

- Takes user input and performs command based on the input

void printHelp(String[] args):

- Prints help information, depending on the user input prints information on the sort command or the areas command

void printHelpSort():

- Prints the help for the sort function

printHelpAreas():

- Prints the help for the areas function

void printNoMatch():

- Prints if no match to the command input by user is found

sort(String[] args):

- Initializes the sort command, with the input being the user's arguments for the type of sorting and output file

void areas(String[] args):

- Initializes the areas command, with the input being the user's arguments for the type of sorting and output file

WeatherTypeEnum enumOfString(String rpr):

- Takes string rpr and returns the corresponding WeatherTypeEnum which is associated with that string

void exit(int exitCode):

- Exits from the program with the given exitCode

void close():

- Closes the input scanner used in the command line8

Consideration

This module helps with fulfilling functional requirement [FR 4.2].

3.4 Dependency viewpoint

3.4.1 Design concerns

The concerns of this viewpoint are the relationships between the components in the program. This allows maintainers to see the overall structure of the program as a series of interconnected modules, which will allow an overall easier process in the location and isolation of modules creating errors or problems.

3.4.2 Design entities, relations, and attributes

A full list of the dependencies of this program can be found in section 3.6.

3.5 Information viewpoint

3.5.1 Design concerns

Concerns of this viewpoint are the persistent data structures. The way that this will be address is that the data will be stored in a Graph. This covers the functional requirements [FR2.1], [FR2.1]

3.5.2 Design entities

Modules

- QuickSort - A module that implements the quick sort algorithm on the graph
- Casualties Comparator - A module compare the casualties of two disasters
- Property Damage Comparator - A module compare the property damage of two disasters
- DisasterPoint - A class that store the data relating to a disaster occurrence.
- Weather Type Enum - A Data type for all type of disasters in the data set
- Parser -Take the file and transfers the data to another class to be used
- Graph - A graph data structure
- Connected Components Finder (Algorithm)- A module that implements algorithm that finds connected components on the Graph
- Convex Hull finder(Algorithm) -A module that implements an algorithm that finds the convex hull on the graph

- Convex Hull - A module that implements the Convex Data structure
- KD-Tree (Data Structure)- A module that implements the KD- Tree Data structure

3.5.3 Design relationships

- Quicksort use the DisasterPoint class as it the object type to be sorted.
- Causalities Comparator will use DisasterPoint. It will use the getter methods to allow the comparison
- Property Damage Comparator will use DisasterPoint. It will use the getter methods to allow the comparison
- Promitiy Comparator will use DisasterPoint. It will use the getter methods to allow the comparison
- DisasterPoint use Enum for the disaster type that it will store for that occurrence
- Filter use Weather Type to apply the filter to be used.
- Graph class use the DisasterPoint class to create a graph data structure
- Parser use DisasterPoints, taking data and separating it into the correct places for the DisasterPoint.
- finder use Graph and Convex Hull
- Convex hull will use graph
- KD -tree will use graph

3.5.4 Design Attributes

DisasterPoint will be persistently use through out this software,as it is the main container for the data.

3.6 Logical viewpoint

3.7 Algorithm viewpoint

The algorithms that were used in this software was Graph, DepthFirstSearch,Quicksort,KdTree

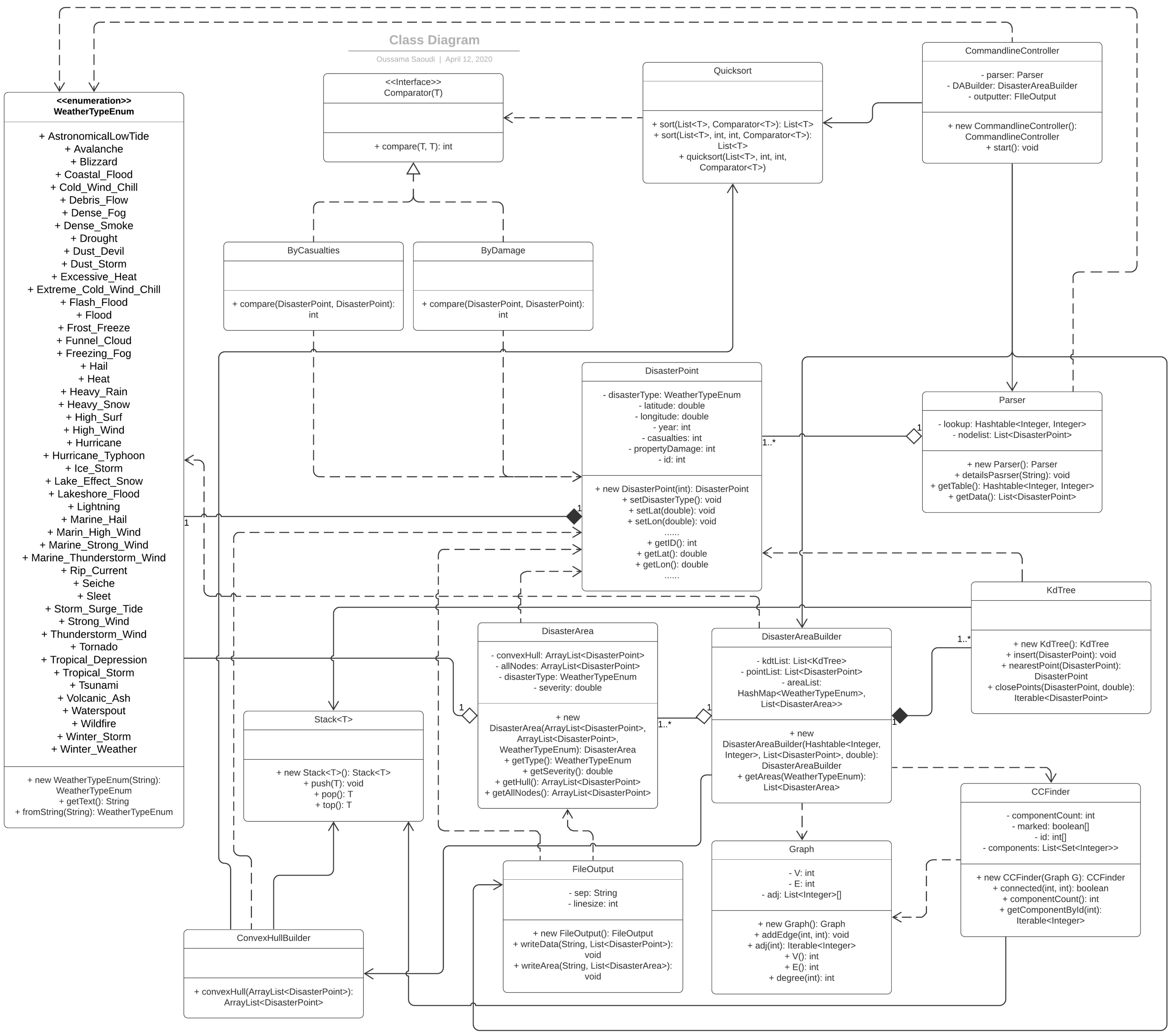
Class Diagram
Oussama Saoudi | April 12, 2020
<<enumeration>>
WeatherTypeEnum
+ AstronomicalLowTide
+ Avalanche
+ Blizzard
+ Coastal_Flood
+ Cold_Wind_Chill
+ Debris_Flow
+ Dense_Fog
+ Dense_Smoke
+ Drought
+ Dust_Devil
+ Dust_Storm
+ Excessive_Heat
+ Extreme_Cold_Wind_Chill
+ Flash_Flood
+ Flood
+ Frost_Freeze
+ Funnel_Cloud
+ Freezing_Fog
+ Hail
+ Heat
+ Heavy_Rain
+ Heavy_Snow
+ High_Surf
+ High_Wind
+ Hurricane
+ Hurricane_Typhoon
+ Ice_Storm
+ Lake_Effect_Snow
+ Lakeshore_Flood
+ Lightning
+ Marine_Hail
+ Marin_High_Wind
+ Marine_Strong_Wind
+ Marine_Thunderstorm_Wind
+ Rip_Current
+ Seiche
+ Sleet
+ Storm_Surge_Tide
+ Strong_Wind
+ Thunderstorm_Wind
+ Tornado
+ Tropical_Depression
+ Tropical_Storm
+ Tsunami
+ Volcanic_Ash
+ Waterspout
+ Wildfire
+ Winter_Storm
+ Winter_Weather
+ new WeatherTypeEnum(String): WeatherTypeEnum
+ getText(): String
+ fromString(String): WeatherTypeEnum
<<Interface>>
Comparator(T)
+ compare(T, T): int
Quicksort
+ sort(List<T>, Comparator<T>): List<T>
+ sort(List<T>, int, int, Comparator<T>): List<T>
+ quicksort(List<T>, int, int, Comparator<T>)
CommandlineController
- parser: Parser
- DABuilder: DisasterAreaBuilder
- outputter: FIleOutput
+ new CommandlineController(): CommandlineController
+ start(): void
ByCasualties
+ compare(DisasterPoint, DisasterPoint): int
ByDamage
+ compare(DisasterPoint, DisasterPoint): int
DisasterPoint
- disasterType: WeatherTypeEnum
- latitude: double
- longitude: double
- year: int
- casualties: int
- propertyDamage: int
- id: int
+ new DisasterPoint(int): DisasterPoint
+ setDisasterType(): void
+ setLat(double): void
+ setLon(double): void
......
+ getID(): int
+ getLat(): double
+ getLon(): double
......
Parser
- lookup: Hashtable<Integer, Integer>
- nodelist: List<DisasterPoint>
+ new Parser(): Parser
+ detailsPasrser(String): void
+ getTable(): Hashtable<Integer, Integer>
+ getData(): List<DisasterPoint>
1..*
1
1
1
KdTree
+ new KdTree(): KdTree
+ insert(DisasterPoint): void
+ nearestPoint(DisasterPoint): DisasterPoint
+ closePoints(DisasterPoint, double): Iterable<DisasterPoint>
DisasterArea
- convexHull: ArrayList<DisasterPoint>
- allNodes: ArrayList<DisasterPoint>
- disasterType: WeatherTypeEnum
- severity: double
+ new DisasterArea(ArrayList<DisasterPoint>, ArrayList<DisasterPoint>, WeatherTypeEnum): DisasterArea
+ getType(): WeatherTypeEnum
+ getSeverity(): double
+ getHull(): ArrayList<DisasterPoint>
+ getAllNodes(): ArrayList<DisasterPoint>
DisasterAreaBuilder
- kdtList: List<KdTree>
- pointList: List<DisasterPoint>
- areaList: HashMap<WeatherTypeEnum>, List<DisasterArea>>
+ new DisasterAreaBuilder(Hashtable<Integer, Integer>, List<DisasterPoint>, double): DisasterAreaBuilder
+ getAreas(WeatherTypeEnum): List<DisasterArea>
1
1..*
1
1
1..*
1..*
1
Stack<T>
+ new Stack<T>(): Stack<T>
+ push(T): void
+ pop(): T
+ top(): T
CCFinder
- componentCount: int
- marked: boolean[]
- id: int[]
- components: List<Set<Integer>>
+ new CCFinder(Graph G): CCFinder
+ connected(int, int): boolean
+ componentCount(): int
+ getComponentById(int): Iterable<Integer>
Graph
- V: int
- E: int
- adj: List<Integer>[]
+ new Graph(): Graph
+ addEdge(int, int): void
+ adj(int): Iterable<Integer>
+ V(): int
+ E(): int
+ degree(int): int
FileOutput
- sep: String
- linesize: int
+ new FileOutput(): FileOutput
+ writeData(String, List<DisasterPoint>): void
+ writeArea(String, List<DisasterArea>): void
ConvexHullBuilder
+ convexHull(ArrayList<DisasterPoint>): ArrayList<DisasterPoint>

Figure 1: UML Class Diagram for Project Vayu

3.7.1 Design concerns

QuickSort Quicksort is a sorting algorithm with a typical performance of $Nlg(N)$ and a worst case performance of $O(N^2)$ for an array of size N. Typically Quicksort is in place, but the implementation used makes a copy of the array and then sorts it, which leads to extra space of $O(N)$

Graph A graph is used in the implementation. A graph is created with nodes size of V and has E edges added to it. Upon initializing the graph takes a time of $O(V)$ to do. Adding an edge is constant time $O(1)$, and doing so E times will result in overall performance in constructing the graph to be $O(V + E)$. The extra space used by the Graph is used to store an adjacency list, which takes a space of $O(V + E)$

Connected Component Finder CCFinder is a module which uses depth first search to is an algorithm used to traverse a graph and assign elements to their respective connected component. Depth First search analyzes each edge and each vertex once, thus having a performance of $O(V + E)$. CCFinder also stores a list of all the connected components, which uses extra space proportional to $O(V)$

KdTree Finding the nearest point only takes $O(lg(N))$ in typical cases, but can reach a performance of $O(N)$ in the worst case. For range, it depending on how dense the graph is, the performance can also be between $O(lg(N))$ and $O(N)$. The space that the KdTree takes is an additional $O(N)$ for representing each node in the tree and a rectangle which represents the space the node is a root for.

Convex Hull The ConvexHullBuilder constructs a convex hull out of a list of points. A convex hull is the smallest set of points such that when a polygon is made from the points it covers all points and edges between them. The algorithm used to construct the convex hull builder is the Graham Scan, which uses the quick sort algorithm for a performance of $O(NlgN))$ for an input set of size N. The ConvexHuLLBuilder creates a new array using the created sorting algorithm, so the extra space used is $O(N)$.

3.7.2 Design elements

Each algorithm has it's own module by the same name.

3.8 Processing Attributes

Quicksort Quick sort works by partitioning a given list with a pivot element, and putting all lesser elements to the left of the element, and all greater elements on the right of the pivot element. Then, it recursively sorts the two subsections on either side of the pivot element. The pivot element is chosen by randomly choosing an element from all the elements in the list. Quicksort can also have the lowest and highest indices defined for the sort to sort only a subsection of the algorithm.

Graph The graph is made by creating an array of $ArrayList < Integer >$, where each of the lists represents the adjacent vertices for a vertex. An integer vertex's adjacency list can be found at its integer position at the array of $ArrayList < Integer >$. Additing an edge adds the vertex to both of the ArrayLists of the vertices. Parallel and self loop edges are allowed in the Graph model used.

Connected Component Finder The connected component finder works by creating an array of booleans called marked, which shows whether each vertex has been visited by the dfs yet. Then, CCFinder goes through each vertex in order of their values, performing dfs at each vertex that is not yet marked. When dfs is run on a source vertex, it is added to a Stack, and then enters a loop. In the loop the stack is popped, the vertex is added to that connected component, and all the popped vertex's adjacent vertices are added to the Stack if they haven't been marked, and are marked. This continues until the stack is empty. Once the dfs returns, the number of connected components is increased. And the loop continues until all vertices are marked. This supports constant time return of connected components.

KdTree A KdTree is a binary tree datastructure representing points in 2d space, allowing for quick insertion of nodes as well as quick retrieval of information such as finding the nearest neighbor and finding all points which reside inside a rectangular area. The KdTree is constructed as follows: All the even valued levels in the binary tree have nodes with a vertical orientation, meaning that all nodes to its left are spatially to its left, and all nodes in the right branch are spatially to its right or on the dividing line. All the odd valued levels in the binary tree have nodes with horizontal orientation, meaning that all nodes to the left subtree are spatially below the point, and all nodes to the right subtree are spatially above or on the dividing line. Additionally every node contains a rectangle which contains all the nodes contained in the node's subtree. Insertion simply works like binary search in that if a node is to the "left" then it goes down to that branch until it hits a null value and returns a reference to a leaf node. In searching for nearest neighbor, the KdTree progressively improves its estimate of the nearest neighbor as it searches for the value in its subtree. If the distance to the node is closer to the dividing line of the node than to the nearest node, the opposite subtree must also be explored. The range function explores the tree looking for points which intersect with a given rectangular range. It does so by recursively checking both sides. The cases are as follows: if the node's left rectangle intersects with the searched rectangle, then it calls range on the node's left child. If the node's right rectangle intersects with the searched rectangle, then it calls range on the node's right child. if the rectangle ever intersects with the node's point, then add the point to a stack to save the points that are within the range.

Convex Hull The convex hull algorithm used is the Graham scan algorithm which works as follows: First find the point with the minimum y value, and break ties by choosing the one with the minimum x value. Set that point as the anchor. Then sort all the points in the input list by comparing their radial angles from the x axis relative to the anchor, with lowest angle to x axis being at the beginning. Then go through and add the points one by one, adding them to the stack. For each point added, check that the orientation of the next-to-top node of the stack, the top node of the stack, and the next point to be added is not clockwise. While it is the case, pop the top element of the stack. Then, add the point to the stack. Once all points have been analyzed the Stack will contain all the points that are supposed to be on the convex hull. If there are fewer than three points, return null because a convex hull must have at minimum 3 points. Else return the stack or transfer the nodes on the stack to an ArrayList (As done in this software implementation).

4 Design Rationale

An object-oriented approach during the design process allowed for efficient division of the workload between group members, and an overall simpler design that allowed for a fairly simple integration of components. The design rationale shall be discussed for the following six subsections:

Sorting The sorting algorithm used is quicksort, which is used to fulfill the requirement FR[1.1]. The algorithm was chosen for its fast linearithmic performance as well as its in-place nature, which makes it much more space efficient than the alternative mergesort. The speed of quicksort was of utmost importance in order to meet the functional requirement FR[4.1]

Disaster Areas To fulfill functional requirement FR[2.1], a method was needed to find proximal points of the same type, aggregate them, and create areas which represent areas affected by a type of natural disaster. To find the proximal points, a KdTree was used because of its typical logarithmic performance of finding proximal points. To aggregate points, it was decided to use a graph to represent points, edges to represent sufficient proximity, and connected components to represent the aggregation of all the proximal points. The connected components contains all the DisasterPoints which are contained in an area. The algorithm used to find the connected components was a simple depth first search variant which marked the connected components of every node. It utilized already made Stack code, and is able to find the connected components in optimal time, which is proportional to the number of vertices and edges. Finally, to fully fulfill FR[2.1] a convex hull algorithm needed to be used to compute the convex hull of the connected component. The algorithm chosen was the Graham Scan for its simplicity of implementation, its use of already implemented code (namely, quick sort), and its linearithmic performance.

Parser Due to the input data, there were several fields describing indirect and direct injury and death as a result of the disaster. It was decided to represent casualties by the sum of all these values and to store that into the DisasterPoint data type. The input data also had many unfilled fields throughout the files. It was decided to retain these fields and set unused values to either null or 0 to retain whatever information the DisasterPoint does have instead of discarding it. However, any line that did not contain a longitude and latitude would not be created into a DisasterPoint and the data was instead discarded. This is because latitude and longitude values are integral to the system working, namely in the DisasterAreaBuilder module. Finally, the decision to only parse StormEvent Details is due to the fact that the file contains all the information needed by the system, and so the additional information files were not parsed or used by the program.

Output In order to make the output user friendly as well, as outlined on FR[4.2], it was decided to prioritize readability of the output file so that values are presented in a table on the output file instead of making the file a CSV file. The output file will always be a text file output by java, thus satisfying FR[3.1]

Commandline Interface The decision to use a command line interface stems primarily from ease of implementation and the short time it takes to build a good one, which allows for development under the time constraint. The commandline interface is designed to provide help to the user through helpful tips on how to use it, which assists in fulfilling FR[4.2]

Overall Design An important decision made with the overall design is to create a list of disaster points and a lookup table and pass references to them the primary controlling methods, namely DisasterAreaBuilder and CommandlineController. The use of both a list and a lookup table is to allow for constant time operation of getting the index representation of a DisasterPoint and getting a DisasterPoint from its index. An example of use of the lookup table is inserting edges into the graph in DisasterAreaBuilder, where the DisasterPoints were known but the index versions of the node were needed to insert edges into the graph. An example of the use of the DisasterPoints list is in passing the reference for sorting in CommandlineController, as well as converting integer points into DisasterPoints such as when the connected components were retrieved from the graph implementation in DisasterAreaBuilder.

State machines

DisasterAreaBuilder

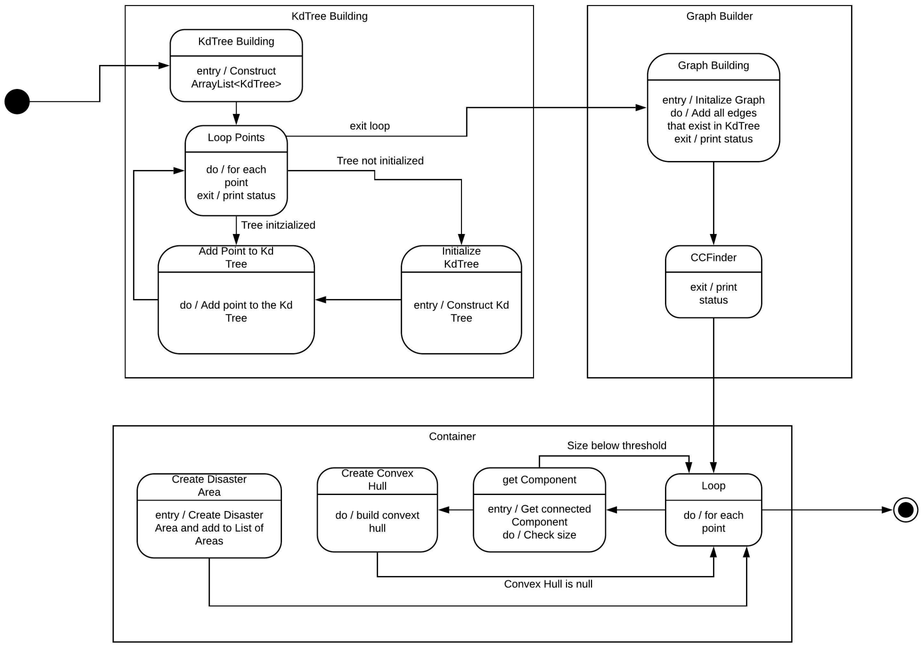

Figure 2: UML State Diagram for DisasterAreaBuilder

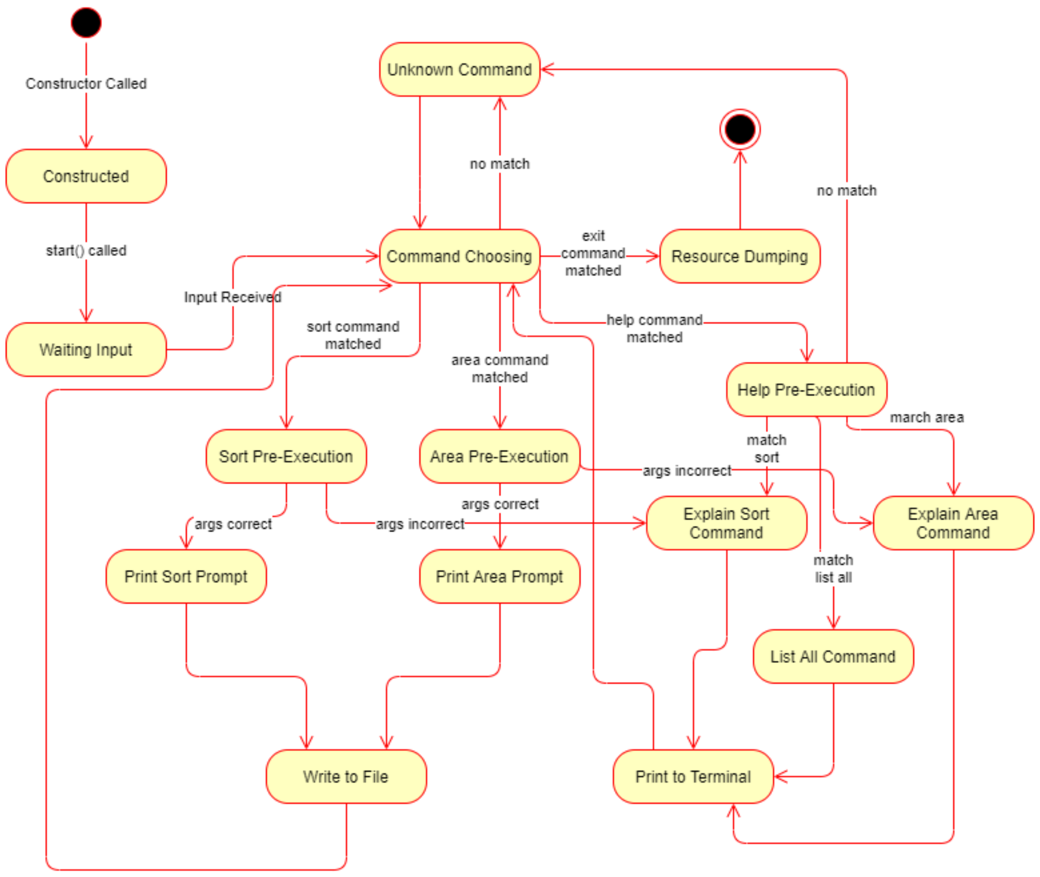

Figure 3: UML State Diagram for CommandlineController