CSCI 3901: Project: External Documentation

Waleed R. Alhindi (B00919848)

Program Components:

- PowerService.java: This class serves as the core of the program. It aggregates data about postal codes and distribution hubs provided by the database interface class (Database.java). Using that data, it provides the reporting methods such as fixOrder, mostDamagedPostalCodes, and rateOfRestoration. Furthermore, this class accepts data that needs to be added into the database (i.e., new hubs, new postal codes, updating hub repair estimates, etc.). Before passing that data to the database class, it performs the necessary validation and formatting.

- Database.java: This class constitutes a database interface class that acts an intermediary between PowerService.java and the MySQL database. Thus, it is meant to isolate any database connectivity and SQL operations from the rest of the program.

- DamagedPostalCodes.java: A class to encapsulate the information of individual postal codes. In other words, each postal code in the database (or ones that have been added during execution) are stored in an object of this class.

- HubImpact.java: A class that encapsulates the information of individual distribution hubs. In other words, each distribution hub in the database (or hubs that have been added during execution) are stored in an object of this class.

- Point.java: A class that stores the coordinates of a distribution hub as (x, y) coordinates. That is to say, each HubImpact object contains an object of this class that represents its (x, y) coordinate locations. Note that the constructor of this class is: Point(int x, int y).

- RepairPlanGrid.java: A class used during PowerService's repairPlan method that stores a 2d array locational grid that represents where startHub, intermediate hubs, and endHub are located relative to startHub (i.e., startHub is at point [0,0] in this grid). Using these coordinates, this class calculates the best possible valid repair plan path from the set of all possible path combinations (which are calculated using the PowerSets and HubPathCombinations classes).

- PowerSets.java: A class used during PowerService's repairPlan method that calculates all possible subsets (power-set) of the intermediate hubs between startHub and endHub. For example, if the intermediate hubs are [A1, B2,], then this class is used to calculate all the subsets of these intermediate hubs, which would be [], [A1], [B2], and [A1, B2].

- HubPathCombinations.java: A class used during PowerService's repairPlan method that calculates all possible paths (permutations) between intermediate hubs for each of the subsets extracted by the PowerSets class. For example, using the power-set example

above, the possible path combinations between the intermediate hubs would be [], [A1], [B2], [A1, B2], and [B2, A1].

- BaseTables.sql: An SQL file that contains the statements used to create the tables utilized by this program, which are: PostalCodes, DistributionHubs, PostalHubRelation, and RepairLog (see "Database Design" section for details about these tables).

- credentials.prop: A properties file to store a user's MySQL username, password, and the database they intend to use (i.e., csci3901). The structure of the properties file is as follows:

  Username= <your MySQL username>
  Password= <you MySQL password>
  Database- <database schema you want to utilize>

Assumptions:

The project instructions granted two important assumptions that affected how this program was implemented: overlapping postal codes will not be provided; and postal codes shall not be deleted. Thus, this program does not validate whether postal codes being added overlap with an existing one nor does it provide any methods that allow for the deletion of postal codes. However, although this program does not check whether postal codes overlap, it does check whether they are identical (see design decision 3, 4, and 5 for more details).
Additionally, this implementation assumes that the user can access the Dalhousie database server (i.e., they have a valid CSID username, password, and database). Furthermore, this implementation assumes that a varchar(100) MySQL datatype is sufficient for postalCode, hubIdentifier, and emp_id data supplied by the user.

Design Decisions:

During the implementation of this program, numerous design decision needed to be made; especially when encountering cases that were ambiguous. As such, the following are the design decisions made during the implementation of this program:

1. All error states are handled by throwing the appropriate exception. An appropriate exception in this case refers to a specific type of exception that is supplied with an error message conveying as much information regarding the error as possible. For example, calling hubDamage with a hubIdentifier of null will throw an IllegalArgumentException, rather than just a generic exception, supplied with the message "HubIdentifier is null! \nSource: hubDamage \nDetails: e.getMessage()".

2. There are two exceptions to design decision (1) where error states are handled by return values rather than exceptions: addPostalCode and addDistributionHub. These two methods are the only methods that return a boolean value. As such, it would make sense

to return true or false rather than throwing an exception. Personally, I would have designed these method to not return boolean values, and instead throw exceptions since error messages could be provided to convey what went wrong. However, given the PowerService interface specified in the project instructions, these two method will return false when encountering an error state.

3. A postal code String passed to addPostalCode that corresponds to an existing postal code will be considered invalid. Note that this does not mean that they overlap, rather it means they are the same postal code. For example, adding “B3J” when postal code “B3J” already exists is invalid, while adding “B3J1H6” in the same scenario is valid. The latter is valid since the method does not take into account overlapping postal codes since the project instructions grant the assumption that they will not be provided.

4. A postal code String passed to addPostalCode must meet the Canadian postal code format, which consist of a letter then alternates between numbers and letters. For example, “B3J” will be accepted as a valid postal code, but not “3BJ” since it neither starts with a letter nor alternates between letters and numbers.

5. All postal code and hub identifier Strings passed throughout the program will be converted to uppercase. For example, passing “b3j” will be converted to “B3J”. Furthermore, all postal codes and distribution hubs passed will have their spaces removed. For example, “b 3 j ” will be converted to “B3J”. As such, all postal codes are normalized to a standard format to ensure data consistency. Consequently, this means that adding the postal code “b 3 J” when “B3J” already exists will be invalid. Similarly, passing “A e 3” as a hubIdentifier to hubDamage when “AE3” exists will update hub “AE3”.

6. A distribution hub can exist when it services no postal codes and vice versa. That is to say, adding a distribution hub where Set<String> servicedAreas is empty is valid, and that adding a postalCode that is not serviced by any existing hubs is also valid.

7. A distribution hub can service a postal code that has not been added. However, if that postal code is added later on, then the relation between the two is update. For example, postal code “B3J” is added, then hub “A1” is added with its servicedAreas including “B3J” and “A2H”. This is valid and only the relation between hub “A1” and postal code “B3J” is established. However, if the postal code “A2H” is added later on, then a connection between hub “A1” and “A2H” is then established.

8. A distribution hub cannot share locations. In other words, no two distribution hubs can have the same x *and* y coordinate values as part of their Point location. As such, adding a distribution hub with a location that an existing hub already resides in will be considered an error state.

9. A hub's repairEstimate that is passed to hubDamage cannot be less than or equal to zero. This is because a negative repairEstimate does not make any sense and a repairEstimate of zero would suggest that there is no damage to be reported.

10. Similarly to design decision (9), passing a negative repairTime to hubRepair is considered as an error state. However, passing a repairTime of zero is not invalid. This is because a zero repairTime might indicate that an employee was simply surveying a hub or that a hub has been misidentified as being damaged.

11. Calling hubRepair on a hub that is already in service is invalid. This is because it makes no sense repairing a hub that is not damaged. Rather, if it is damaged and has not been reported yet, then the employee should call the hubDamage method before calling hubRepair.

12. If a repairTime greater than or equal to a hub's repairEstimate is passed alongside an inService value of false, then the hub's repairEstimate will not be changed. This is because a hub's repairEstimate cannot be negative and because there is no way of readjusting its repairEstimate in a way that accurately reflects the misestimation in its repairEstimate. For example, if a hub's repairEstimate is 10 and hubRepair is called with a repairTime of 15 and an inService of false. Then that hub's repairEstimate will remain 10 since it cannot be -5 and because there isn't enough information to readjust the repairEstimate to reflect how much it has been misestimated (i.e., how much the initial repairEstimate should have been).

13. Building upon design decision (12), if an inService of true is passed to hubRepair, then that hub's repairEstimate and impact are reverted to 0 regardless of the values of repairTime and repairEstimate.

14. If the number of people out of service is not a whole number, then it will be rounded up. For example, if the number of people out of service is 10.5, then calling peopleOutOfService will return 11. Such cases arise when a postal code is serviced by multiple hubs. For example, a postal code with a population of 10 that is serviced by 3 hubs, where 2 are damaged, will effectively have 6.667 people without power. Thus, it will be rounded up to 7.

15. mostDamagedPostalCodes, fixOrder, underservedPostalByArea, and underservedPostalByPopulation will handle ties at the limit by adding them to the returned list in a manner that is similar to assignment 3. In other words, any entities passed the limit that tie with the entity at the limit will be added to the returned list.

16. Passing a limit of less than 1 to mostDamagedPostalCodes, fixOrder, underservedPostalByArea, or underservedPostalByPopulation is considered an error state and will throw an exception. This is because a negative or zero limit would also return an empty list. Although, returning an empty list and letting the methods complete "vacuously" is a possible approach to handling such an error state, throwing an exception with a dedicated message conveys more information to the user and is consistent with

how error states are being handled throughout the program (with the exceptions mentioned in design decision (2)).

17. mostDamagedPostalCodes does not take into account postal codes that are not serviced by any hubs. This is because although these postal codes may not have power, they are not “damaged” as they have no repair time estimate to when they would get their power back. Thus, such postal codes are omitted when identifying the most damaged postal codes.

18. The increment floating point value passed to rateOfServiceRestoration must be greater than 0 and less than or equal to 1. Any values outside this range are considered invalid. This is because an increment of 0 would produce and infinite list and a negative increment is logically impossible as it would be asking for a negative population. On the other hand, an increment of over 1 would be invalid because it is asking for more than the population we have information about.

19. Supplying rateOfServiceRestoration with an increment value of less than 0.001 (0.1%) is considered invalid since it would constitute a very fine degree of granularity which would result in a very large list, and would result in the inverse amount of granularity on the number of hours in that returned list.

20. rateOfServiceRestoration will not take into account any postal codes that are not serviced by any hubs. This is because doing so would lead to an infinite list since the people in service will never reach 100% even if all the damaged hubs were repaired.

21. If rateOfServiceRestoration is called when no hubs that service at least one postal code are out of service, then a list of zeroes (denoting zero hours of repair) is returned. The size of that list depends on the increment value passed. For example, if an increment of 0.5 is supplied in such a case, then the list [0, 0, 0] is returned denoting 0%, 50%, and 100%, respectively.

22. If rateOfServiceRestoration is called when no hubs exist, then it will throw an exception. This is because throwing an exception with a dedicated message conveys more information to the user than simply return a list of zeroes.

23. underservedPostalsByPopulation and underservedPostalsByArea will treat all postal codes that are not being serviced by any hubs as equally underserved. Furthermore, such postal codes are considered more underserved than postal codes that are being serviced by at least one hub.

24. The program calculates a hub’s impact by dividing the total amount of people effected by a hub’s outage by its repairEstimate.

25. In repairPlan, if two hubs within range are tied for the highest impact (i.e., tied for being endHub), then their distance from startHub will be the tiebreaker, where the tied hub that is furthest from startHub will become endHub. This is because doing so will provide the

widest range between startHub and endHub which, in many cases, maximizes the number of intermediate hubs between them. However, if the two tied hubs are also the same distance from startHub, then repairPlan will just assign the first one it identifies as endHub.

26. repairPlan calculates the distance between hubs using the Pythagorean theorem [distance = $\sqrt{a^2 + b^2}$] and uses the point slop formula [y = m(x) + b] to identify the diagonal line between startHub and endHub. Particularly, the slope of the diagonal (m) is used to identify whether an intermediate hub is below, on, or above the diagonal. This information is used as the basis of identifying whether a repair plan path crosses the diagonal more than once.

27. If repairPlan is called with a maxDistance of 0, then it will return a list containing only the startHub specified. This is because no two hubs can have the same (x,y) coordinate point location. Thus, a maxDistance of 0 means that there will be no hubs in range of startHub.

28. If repairPlan is called with a maxTime of 0, then it will return a list containing only the startHub and endHub since these two will be repaired regardless of their repair time estimates.

29. If repairPlan is called with a negative maxTime or maxDistance, then it will throw an exception since neither time nor distance can be naturally be negative.

30. Calling repairPlan with a startHub that does not exist (has not been added) will be considered an error state and throw an exception since the repair plan's start point would not have been identified.

31. Calling repairPlan with a startHub that is already in service is considered an error state for the same reason as calling hubRepair on an in service hub is invalid, and will throw an IllegalArgumentException.

32. repairPlan takes into account damaged hubs that have an impact of 0 (i.e., they are damaged but do not service any postal codes or service postal codes that have populations of 0). This is because the resultant repair plan would include the maximum amount of possible hub repairs.

33. The database schema (as provided by the BaseTables.sql file) does not include a dedicated employee table. Thus, this program does not verify whether employee ids supplied in hubRepair correspond to an existing employee. Rather, the RepairLog table stores an employee's id as part of logging the repair. This decision was made due to the fact the program has no interface to add employees, meaning employees would need to be added manually through MySQL commands rather than through the program. However, the program and database schema were designed such that tables, relationship constraints, and method can easily be extended to include a dedicated employee table and employee id verification functionality.

Data Structures:

Since the program relies heavily on a MySQL database to store and organize data and their relationships, it only utilizes three major data structures:

- A <String, DamagedPostalCodes> map in the PowerService class that aggregates all existing and newly added postal codes. The String key is a DamagedPostalCode's postal code and its value is the corresponding DamagedPostalCode object. This data structure reduces the amount of SQL operation needed since it is populated with the database's existing postal codes during the PowerService constructor.

- A <String, HubImpact> map in the PowerService class that aggregates all existing and newly added distribution hubs. The String key is a hub's identifier and its value is the correspond HubImpact object. Similarly, to the above data structure, this map is populated with the database's existing hub data during the PowerService constructor to reduce SQL fetch operations.

- A Set of Strings in each HubImpact object that stores the postal codes it services. Unlike the other two data structures, this set is not meant to reduce SQL operations. Rather, it is meant to facilitate the fetching of information concerning each of a hub's serviced areas. For example, when a hub is restored, this set will indicate which specific postal codes need to be updated in the database (and in the DamagedPostalCodes objects) to reflect this restoration.

Additionally, during repairPlan, an object of the RepairPlanGrid class is instantiated, which contains the following minor (i.e., temporary and local to the repairPlan method) data structures:

- A 2d String array in the RepairPlanGrid object serving as a locational grid that represents where startHub, intermediate hubs, and endHub are located relative to startHub. For example, a startHub at (2,2), endhub at (4,4), and intermediate hub at (3,3) would be stored in a 2x2 grid with startHub at [0][0], endHub at [2][2], and intermediate hub at [1][1].

- A <String, int[]> map in the RepairPlanGrid object that stores the relative coordinates of startHub, intermediate hubs, and endHub that were set onto the 2d array grid. For instance, building upon the example provided above, this map would have the following entries: (startHub id, [0,0]), (intermediate hub id, [1,1]), and (endHub id, [2,2]). This map is used to quickly retrieve the location grid coordinates of a hub as opposed to iterating through the 2d array itself. Thus, this map can be thought of as an index of sort.

Database Design:

The database used to facilitate this program's operations consists of four tables (see **Figure 1)**:

- PostalCodes:
    - id: varchar(100) not null [primary key]
    - population: int
    - area: int

- DistributionHubs:
    - Id: varchar(100) not null [primary key]
    - x (i.e., hub location's x coordinate): int
    - y (i.e., hub location's y coordinate): int
    - repairTime: float
    - inService: Boolean

- PostalHubRelation (*bridge table to facilitate many-to-many relationship between PostalCodes and DistributionHubs tables*):
    - postalId: varchar(100) not null
    - hubId: varchar(100) not null [foreign key referencing DistributionHubs(id)]
    - primary key (postalId, hubId)

- RepairLog:
    - Id: int auto-increment not null [primary surrogate key]
    - Emp_id: varchar(100) not null
    - hubId varchar(100) not null [foreign key referecing DistributionHubs(id)]
    - repairTime: float
    - hubRestored (i.e., whether this repair restored the hub): boolean

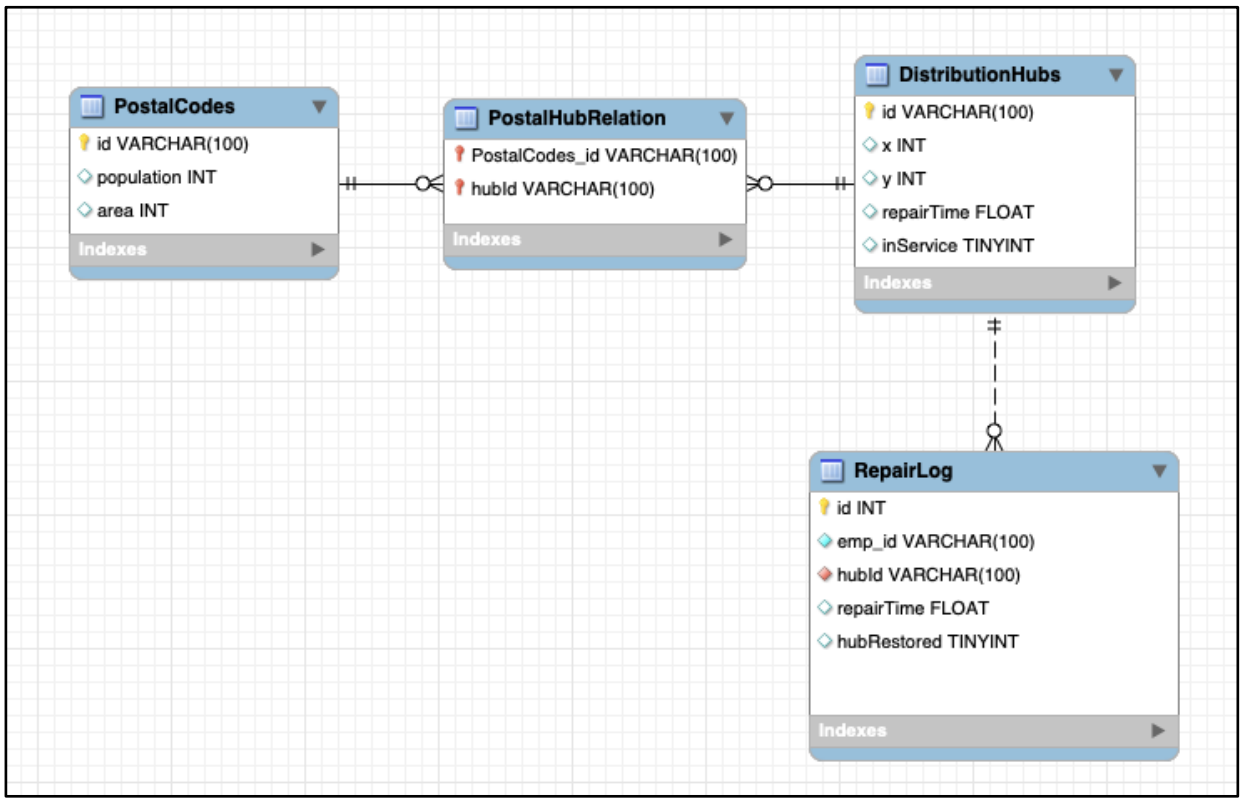

Figure 1: ERD diagram of database's tables

Key Algorithms:

Adding a Postal Code

This algorithm is rather simple. It first checks if the arguments are valid, then checks if the postal code is formatted correctly and that it does not already exist in the PowerService object's postalCodes map. Note that this map remains up to date with the database's data since it is populated with the database's existing data when the object is instantiated and is updated alongside any database operations. If the arguments supplied pass the validation checks, then the method simply created a new DamagedPostalCode object which it passes to the database interface object. The database interface object takes care of inserting it into the MySQL database. Then, we pass the object to another database interface method that updates the bridge table that relates postal codes and hubs. After which, the database interface object calculates the new postal code's repairEstimate, which the method then uses to set the new DamagedPostalCode object's repairEstimate value. Finally, if the insertion and updates were successful, the PowerService object's postalCodes map is updated to reflect the insertion.

Perform all necessary validations (i.e., postalCode is not null, population is not negative, etc.)

Return false if any of the validations failed

If the postalCode already exists as a key in the PowerService object's postalCodes map

Return false

Create a new DamagedPostalCodes object using the arguments supplied

Try

Pass the new DamagedPostalCodes object to the interface database object's "addPostalCodeToDB" method

Catch any SQLExceptions

Return false

Try

For each entry in the distributionHubs map

If that HubImpact object's servicedAreas contains the new postal code

Pass the new postal code and this hub to the database interface object's "updatePostalHubRelation" method

Use the database interface object to calculate the postal code's repairEstimate (this algorithm is detailed below)

Catch any SQLExceptions

Return false

Add the new DamagedPostalCode object to the postalCodes map

Return true

Calculating a Postal Code's Repair Estimate

This algorithm is part of the Database class and is used throughout various PowerService methods. It cross references the PostalHubRelation and DistributionHubs tables to calculate the total repairEstimate of a postal code.

Note this algorithm is supplied with a postalCode String as an argument

Create a totalRepairTime variable and set it to 0

Try

Connecting to the database

Selecting all the columns from PostalHubRelation joined with DistributionHubs on hubId where PostalHubRelation's postalId field equals this postalCode

For each of the rows returned

If the row's "inService" field is false

Increment the totalRepairTime by the row's "repairTime" field value

Catch any SQLExceptions

Throw the exception back to the calling method (since this method is called as part of PowerService methods, which handle the exceptions)

Return the totalRepairTime

Adding a Distribution Hub

Similarly to the algorithm for adding postal codes, this algorithm is rather simple since it basically just validates the arguments passed then passes them to the database interface object to insert and update the relevant tables.

Perform all necessary validations (i.e., hubIdentifier is not null, servicedAreas is not null, etc.)

    Return false if any of them failed

If the hubIdentifier already exists as a key in the distributionHubs map

    Return false

For each entry in the distributionHubs map

    If this HubImpact's Point object's coordinates equal the coordinates of the hub being added

    Return false

Create a new HubImpact object using the arguments supplied

Try

    Pass the new HubImpact object to the database interface object's "addDistributionHubToDB" method

Catch any SQLExceptions

    Return false

Try

    For each postalCode in the servicedAreas set argument

        Pass this postalCode and the newly added hub to the database interface object's "updatePostalHubRelation" method

Catch any SQLExceptions

    Return false

Add the new HubImpact object to the distributionHubs map

Return true

Updating a Hub's Damage

This algorithm updates a hub's repairTime value by passing the relavent arguments to the database interface object, which then updates the DistributionHubs table to reflect the damage being reported. After which, each of the postal codes being serviced by the hub have their repairEstimate values modified to reflect the damage being reported. Finally, the hub's impact and populationEffected values are recalculated to reflect the damage being reported.

Perform validation checks (i.e., hubIdentifier is null, repairEstimate is negative, etc.)

Throw an IllegalArguementException if any of them failed

If the hubIdentifer does *not* exist as a key in the distributionHubs map

Throw an IllegalArgumentException

Get the HubImpact object from the distributionHubs map that corresponds to the hubIdentifier

Increment that HubImpact object's repairTime value by the repairEstimate argument's value

Set that HubImpact object's inService value to false

Try

Pass the hubIdentifier and repairEstimate to the database interface object's "updateHubDamage" method

Catch any SQLExceptions

Throw that SQLException

Try

For each of Strings in this HubImpact's servicedAreas set

Use the database interface object to calculate that postalCode's updated repairEstimate

Set this new repairEstimate as the corresponding DamagedPostalCode object's repairEstimate value

Calculate the number of people affected by this hub outage by passing the hubIdentifier and its corresponding servicedAreas set to the database interface object's "calculatePopulationEffect" method

Set the calculated value as this HubImpact's populationEffected value

Set this HubImpact's impact value as (populationEffected / repairTime)

Catch any SQLExceptions

Throw that SQLException

Repairing a Hub

This algorithm creates a new row in the RepairLog table to log this hub repair. It then updates the DistributionHubs table to reflect the repair and updates the corresponding HubImpact objects as well. It then updates each serviced postal code's DamagedPostalCode object and updates the PostalCodes table to reflect changes in repairTime due to this repair.

Perform validation checks (i.e., hubIdentifier is null, repairTime is negative, etc.)

Throw an IllegalArguementException if any of them failed

If the distributionHubs map does not contain hubIdentifier as a key

Throw an IllegalArguementException

If this hub's inService value is true

Throw an IllegalArgumentException

Try

Pass the relevant argument to the database interface object's "updateRepairLog" method that will insert a row into the RepairLog table to reflect this repair

Catch any SQLExceptions

Throw that SQLException

If the inService argument is true

Set this hub's repairEstimate to 0

Set this hub's impact to 0

Set this hub's populationEffected to 0

Set this hub's inService value to 0

Try

Update the DistributionHubs table to reflect this restoration by passing the relevant argument to the database interface object's "applyHubRepairToDB" method

Catch any SQLExceptions

Throw that SQLException

Try

Update each postal code being serviced by this hub's repairTime in the PostalCodes table by passing the relevant argument to the database interface object's "calculatePostalRepairTime" method

Catch any SQLExceptions

Throw that SQLException

Else

If repairTime is less than this hub's repairEstimate

Set this hub's repairEstimate to (repairEstimate – repairTime)

Try

Update the DistributionHubs table to reflect this restoration by passing the relevant argument to the database interface object's "applyHubRepairToDB" method

Catch any SQLExceptions

Throw that SQLException

Try

Update each postal code being serviced by this hub's repairTime in the PostalCodes table by passing the relevant argument to the database interface object's "calculatePostalRepairTime" method

Catch any SQLExceptions

Throw that SQLException

Calculating the Total Number of People Out of Service

This algorithm calculates the total number of people out of service by passing each postal code in the postalCodes map to the database interface object, which cross references the PostalHubRelations and DistributionHubs tables to find out how many people are out of service in a postal code.

Create a totalPeopleOutOfService variable and set it to 0

Try

    For each postal code in the postalCodes map

        Calculate the number of people out of service in this postal code by passing the postal code to the database interface object's "postalPopulationOutOfService" method (this algorithm is detailed below)

        Increment totalPeopleOutOfService by the calculated value

Catch any SQLExceptions

    Throw that SQLException

Round up totalPeopleOutOfService (i.e., if the value is 10.5, then it is rounded to 11)

Return totalPeopleOutOfService

Calculating the Number of People Out of Service in a Postal Code

This algorithm is part of the Database class and is used in PowerService's peopleOutOfService method. It calculates the number of people out of service in a particular postal code by joining the PostalHubRelation and DistributionHubs tables, then multiplying the postal code's population by the fraction of (damaged hubs/total hubs) that service it.

Note this algorithm is supplied with a postalCode String as an argument

Create a totalHubs variable and set it to 0

Create a damagedHubs variable and set it to 0

Try

    Connect to the database

    Select all fields from PostalHubRelation join DistributionHubs on hubId where postalId equals the postalCode argument

    For each of the rows returned

        Increment totalHubs by 1

        If the row's "inService" field is false

            Increment damagedHubs by 1

Catch any SQLExceptions

    Throw the exception back to the calling method (since this method is called as part of PowerService methods, which handle the exceptions)

If totalHubs is 0

    Return 0

Calculate the fraction of downed hubs as (damagedHubs / totalHubs)

Return that calculated fraction

Identifying the Most Damaged Postal Codes

Since the postalCodes map stores each postal code's corresponding DamagedPostalCodes objects, this algorithm simply adds any postal codes with a repair time that is greater than 1 to a temporary map, where the key is the postal code String and its value is that postal code's repair time. Then, that map is sorted by its values in descending order. Now, the algorithm simply adds entries of the map to the list of most damaged postal codes until it reaches the limit or until it has added all the damaged postal codes (in the case where limit is greater than the number of damaged postal codes). Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3).

If limit is less than 1

    Throw an IllegalArgumentException

Create a postalRepairTimes <String, Float> map to store each postal code's repair time

For each DamagedPostalCode in the postalCodes map

    If that postal code's repairTime > 0

        Add that postal code and its repair time as a key-value pair into the postalRepairTimes map

Sort the postalRepairTimes by its values in descending order by passing it to a sortByValue helper method

Create a list of DamagedPostalCodes called mostDamagedPostalCodes

Create a counter variable and set it to 0

Create a valueAtLimit variable and set it to -1

For each entry in the postalRepairTimes map

    If counter is less than limit – 1

        Add this postal code to the mostDamagedPostalCodes list

        Increment counter by 1

    Else if counter equals limit – 1

        Add this postal code to the mostDamagedPostalCodes list

        Increment counter by 1

        Set valueAtLimit to this postal code's repair time

    Else

        If this postal code's repair time equals valueAtLimit

            Add this postal code to the mostDamagedPostalCodes list

        Else

            Break the loop (stop adding postal codes to mostDamagedPostalCodes)

Return mostDamagedPostalCodes list

Identifying the Most Impactful Hubs (fixOrder)

Similarly to the previous algorithm, since each hub's HubImpact object is stored in the distributionHubs map, this algorithm adds each damaged hub to a temporary map that is then sorted by its value in descending order. Then, the algorithm adds entries of that map to the fixOrder list until the limit is reached or all damaged hubs are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3).

```
If limit is less than 1
        Throw an IllegalArguementException
Create a <String, Float> map called hubImpacts to store each hub's impact
For each hub in the distributionHubs map
        If this hub's inService value is false
                Add this hub's identifier and impact as key-pair values into the hubImpacts map
Sort the hubImpacts map by its values in descending order by passing it to a helper method
Create a new list called fixOrder
Create a counter variable and set it to 0
Create a valueAtLimit variable and set it to -1
For each hub in the hubImpacts map
        If counter is less than limit – 1
                Add the HubImpact object that corresponds to this hub to the fixOrder list
                Increment counter by 1
        Else if counter equals limit – 1
                Add the HubImpact object that corresponds to this hub to the fixOrder list
                Set valueAtLimit to this hub's impact value
                Increment counter by 1
        Else
                If valueAtLimit equals this hub's impact value
                        Add the HubImpact object that corresponds to this hub to the fixOrder list
                Else
                        Break the loop (stop adding hubs to the fixOrder list)
Return fixOrder
```

Identifying the Postal Codes that Are Underserved Based on Population

This algorithm uses the database interface class to find how many hubs each postal code is serviced by, then divides the postal code's population by that number. It then adds that value (capitaPerHub) to a temporary <String postalCode, Float capitaPerHub> map which is sorted by its values in descending order. Then, the postal codes in that map are added to the underservedByPopulation list until it reaches the limit or all postal codes are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3). Also note that this algorithm will treat all postal codes that are not served by any hubs as equally underserved (see design decision (18) for more detail).

If limit is less than 1

    Throw an IllegalArguementException

Create a postalPopHubs <String, Float> map to store each postal code's capitaPerHub

Create a list of strings called noServicePostals to store postal codes that are not served by any hubs

For each DamagedPostalCode in the postalCodes map

    Try

        Get the number of hubs that service that postal code by passing the postal code String to the database interface object's "getNumberOfPostalHubs" method

        If the number of hubs that service this postal code is 0

            Add this postal code to the noServicePostals list

            Continue to next postal code

        Divide this postal code's population by the number of hubs servicing it

        Add the postal code and the value calculated (capitaPerHub) to the postalPopHubs map

    Catch any SQLExceptions

        Throw that SQLException

Sort the postalPopHubs map by its values in descending order by passing it to a helper method

Create a counter variable and set it to 0

Create a valueAtLimit variable and set it to -1

Create a underservedPostals list

For each postal code in the noServicePostals list

    Add that postal code to underservedPostals list

    Increment counter by 1

For each entry in the postalPopHubs map

    If counter is less than limit – 1

        Add this postal code to the underservedPostals list

        Increment counter by 1

    Else if counter equals limit – 1

Add this postal code to the underservedPostals list

Increment counter by 1

Set valueAtLimit to this postal code's capita per hub value

Else

If this postal code's capita per hub value equals valueAtLimit

Add this postal code to the underservedPostals list

Else

Break the loop (stop adding postal codes to the list)

Return underservedPostals

Identifying the Postal Codes that Are Underserved Based on Area

This algorithm uses the database interface class to find how many hubs each postal code is serviced by, then divides the postal code's area by that number. It then adds that value (meterPerHub) to a temporary <String postalCode, Float meterPerHub> map which is sorted by its values in descending order. Then, the postal codes in that map are added to the underservedByPopulation list until it reaches the limit or all postal codes are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3). Also note that this algorithm will treat all postal codes that are not served by any hubs as equally underserved (see design decision (18) for more detail).

If limit is less than 1

Throw an IllegalArguementException

Create a postalAreaHubs <String, Float> map to store each postal code's meterPerHub

Create a list of strings called noServicePostals to store postal codes that are not served by any hubs

For each DamagedPostalCode in the postalCodes map

Try

Get the number of hubs that service that postal code by passing the postal code String to the database interface object's "getNumberOfPostalHubs" method

If the number of hubs that service this postal code is 0

If the number of hubs that service this postal code is 0

Add this postal code to the noServicePostals list

Continue to next postal code

Divide this postal code's area by the number of hubs servicing it

Add the postal code and the value calculated (meterPerHub) to the postalPopHubs map

Catch any SQLExceptions

Throw that SQLException
Sort the postalAreaHubs map by its values in descending order by passing it to a helper method
Create a counter variable and set it to 0
Create a valueAtLimit variable and set it to -1
Create a underservedPostals list
For each postal code in the noServicePostals list
    Add that postal code to underservedPostals list
    Increment counter by 1
For each entry in the postalPopHubs map
    If counter is less than limit – 1
        Add this postal code to the underservedPostals list
        Increment counter by 1
    Else if counter equals limit – 1
        Add this postal code to the underservedPostals list
        Increment counter by 1
        Set valueAtLimit to this postal code's capita per hub value
    Else
        If this postal code's capita per hub value equals valueAtLimit
            Add this postal code to the underservedPostals list
        Else
            Break the loop (stop adding postal codes to the list)
Return underservedPostals

Calculating the Rate of Service Restoration

This algorithm makes use of the fixOrder method and peopleOutOfService method to calculate the percentage of people in service after each hub in the fixOrder list is restored. Using this information, it populates a the rateOfServiceRestoration list with times that reflect the incremental percentage of people that will be in service as hubs are restored.

If increment is less than or equal to 0, or if increment is greater than 1
    Throw an IllegalArgumentException
If increment is less than 0.001
    Throw an IllegalArguementException (granularity issue)
If the distributionHubs map is empty (i.e., no hubs exist)
    Throw an IllegalArguementException
Create a list called rateOfRestoration
Get a list of the most impactful hubs called fixOrder by calling the fixOrder method

Get a variable called peopleOutOfService that stores the number of people out of service by calling the peopleOutOfService method

Get a variable called totalPopulation that stores the total number of people by calling a helper method

Calculate the current percentage of the population in service by subtracting poepleOutOfService from totalPopulation and store that value as percentageOfInServicePop

Divide percentageOfInServicePop by increment and store that value as alreadyHavePowerIncrement

Add a alreadyHavePowerIncrement amount of 0s to the rateOfRestoration list

Create a currIncrement variable and set it to alreadyHavePowerIncrement

Create a repairTimeElapsed variable and set it to 0

For each HubImpact in the fixOrder list

    Incremebt repairTimeElapsed by this HubImpact's repairEstimate value

    Increment percentageOfInServicePop by this HubImpact's populationEffected value

    While percentageOfInServicePop is less than currIncrement

        If currIncrement equals 1

            Add repairTimeElapsed to the rateOfRestoration list

            Break the loop (stop adding to rateOfRestoration)

        Add repairTimeElapsed to the rateOfRestoration list

        Increment currIncrement by increment

Return rateOfRestoration

Identifying an Optimal Repair Plan

This algorithm first identifies the damaged that are in <maxDistance> range of startHub using the formula [distance = $\sqrt{a^2 + b^2}$] where [a = startHub's Y – potentialHub's Y] and [b = startHub's X – potentialHub's X]. Of these in-range hubs, the algorithm identifies which of them is endHub by comparing their impacts (or in the case of tied impacts, comparing their distances). Once the algorithm has identified endHub, it passes startHub, endHub, and the list of in-range hubs to another key algorithm (which is detailed further below) to identify the hubs that reside between startHub and endHub. Then, all possible path combinations between startHub, among the intermediate hubs, to endHub is calculated. This list of all possible path combinations is then fed into another key algorithm (also detailed further below) that identifies which of these path combinations is valid, and which of the valid ones is the best (most impactful).

Perform all necessary input validations (i.e., maxDistance is not negative, startHub not null, etc.)

    If it doesn't pass any of these validations, throw an IllegalArgumentException

Check if startHub exists (has already been added)

    If it doesn't, throw an IllegalArgumentException

Get the startHub's (x,y) coordinates (i.e., its location Point object's attributes)

Create a hubsInRange list

Create endHub variable to store the hub with highest impact found in range

Create a highestImpact variable to track endHub's impact

Iterate through the distributionHubs map

If this hub is in service

Skip this hub (continue to next iteration)

If this hub is startHub

Skip this hub (continue to next iteration)

Get this hub's (x,y) coordinates

Calculate this hub's distance from startHub using the formula [distance = $\sqrt{a^2 + b^2}$]

If that calculated distance is less than or equal to maxDistance

Add this hub to the hubsInRange list

If this hub's impact is greater than the value currently stored in highestImpact

Set endHub to this hub

Set highestImpact to this hub's impact value

Check if hubsInRange is empty

If it is, return a repairPlan list containing only the startHub

//Now that we have identified all hubs in range of startHub and designated one of them as //endHub, we feed that information into a separate algorithm to identify the subset of hubs in //range that reside between startHub and endHub (this key algorithm is explained further below)

Get an intermediateHubs list by passing the startHub, endHub, and hubsInRange list to "findIntermediateHubs" method

If intermediateHubs list is empty

Return repairPlan list containing only startHub and endHub

//Now that we know startHub, endHub, and the hubs that reside between them, we use this //information to calculate all the possible paths from startHub, between the intermediate hubs, to //endHub. In other words, we find all permutations of each subset of the intermediate hubs.

Calculate the power-set (all possible subsets) of the list of intermediate hubs

Create a list of lists called pathCombinations

For each of these subsets

Calculate all possible path combinations between the hubs in this subset (i.e., calculate all permutations of this subset) and add them to the pathCombinations list

//Now, we feed all possible path combinations to a separate algorithm which identifies whether //paths are valid, and calculates which of those valid paths is the most impactful (this key //algorithm is explained further below)

Pass the pathCombinations list to the "findBestRepairPath" method which of these paths is the most impactful and valid

Add startHub to the repairPlan list

Add all the intermediate hubs in the returned bestRepairPlan list to the repairPlan list

Add endHub to the repairPlan list

Return the repairPlan list

Identifying Intermediate Hubs Between StartHub and EndHub

This algorithm is used as part of the larger algorithm that identifies the best repair plan path. It is supplied with the startHub, identified endHub, the list of hubs in range of startHub, and maxTime. Using startHub's and endHub's relative coordinates, it calculates whether the rectangle between the two is in quadrant 1, 2, 3, or 4 relative to startHub. In other words, it identifies whether the rectangle is to the upper-right, lower-right, upper-left, or lower-left of startHub. Using this information, it iterates through the list of in-range hubs and calculates whether that hub resides in the rectangle between startHub and endHub by comparing its relative coordinates against startHub's and endHub's. Finally, it returns an intermediateHubs list containing the hubs it identifies as residing between startHub and endHub.

```
//this algorithm is provided with startHub, endHub, the list of hubs in range, and maxTime as its
//parameters
Create an intermediateHubs list
Get startHub's (x,y) coordinates
Get endHub's (x,y) coordinates
Iterate through the list of hubs in range
        If this hub is endHub
                Skip it (continue to next iteration)
        If this hub's repair time estimate is greater than maxTime
                Skip it (continue to next iteration)
        Get this hub's (x,y) coordinates
        //Now, we find which quadrant relative to startHub the rectangle resides in (i.e., is the
        //rectangle between startHub and endHub to the upper right, upper left, lower left, or
        //lower right of startHub).
        //Then, we check if this hub resides in that rectangle.
        If startHub's X is greater than endHubs' (i.e., quadrant 2 or 3 [to left of startHub])
                If startHub's Y is greater than endHub's (quadrant 3 [lower-left])
                        If endHub's X <= this hub's X <= startHub's X
                                If endHub's Y <= this hub's Y <= startHub's Y
                                        Add this hub to intermediateHubs list
                Else (quadrant 2 [upper-left])
                        If endHub's X <=this hub's X <= startHub's X
                                If startHub's Y <= this hub's Y <= endHub's Y
                                        Add this hub to intermediateHubs list
```

Else (i.e., quadrant 1 or 4 [to right of startHub])
If startHub's Y is greater than endHub's (quadrant 4 [lower-right])
If endHub's X <= this hub's X <= startHub's X
If endHub's Y <= this hub's Y <= startHub's Y
Add this hub to intermediateHubs list
Else (quadrant 1 [upper-right])
If startHub's X <= this hub's X <= endHub's X
If startHub's Y <= this hub's Y <= endHub's Y
Add this hub to intermediateHubs list
Return intermediateHubs list

Finding the Best Valid Repair Path Among All Possible Paths

This algorithm is used as part of the larger algorithm that identifies the best repair plan path. It is supplied with a list of all possible path combinations between startHub, among intermediate hubs, to endHub as well as endHub itself. The algorithm iterates through each of these possible paths to check whether it is valid (i.e., a path is either x-monotonic or y-monotonic *and* crosses the diagonal no more than once). To check whether a path is x or y monotonic, the algorithm simply check if the displacement between two consecutive hubs in the path is negative on the x or y axis (i.e., check if previous hub's X/Y is less than current hub's X/Y). Furthermore, the algorithm tallies how many times a path cross the diagonal by utilizing the slope of the diagonal between startHub and endHub, which is calculated as [m = (endHubY – startHubY) / (endHubX – startHubX)]. For each hub in a path, it check whether that hub is above, on, or below the diagonal by multiplying that hub's X with the slope (m). Thus, if the hub's Y is greater than X(m), then it is above the diagonal; if it smaller, it is below the diagonal; and if they are equal, then it is on the diagonal. Using this information, the algorithm checks whether the previous hub on the path and the current hub are on different sides of the diagonal (incrementing the crossedDiagonal tally accordingly). Finally, of the valid paths identified, the algorithm returns the one with the highest impact.

//this algorithm is given an allPathCombinations List<List<HubImpact>> and endHub.
//Additionally, since this algorithm is part of the RepairPlanGrid class, it has access to the
//locational grid coordinates (i.e., the relative coordinates of startHub, intermediate hubs, and
//endHub)

Calculate the slope (m) of the diagonal line between startHub and endHub using the formula [m = (endHubY – startHubY) / (endHubX – startHubX)]

Create list of hubs variable called bestPath to store best valid path (initially set to null)

Create bestImpact variable to store impact overall value of bestPath (initially set to -1)

For each path combination (List<HubImpact>) in allPathCombinations

Create xMonotonic boolean and set to true

Create yMonotonic boolean and set to true

Create crossedDiagonal tally variable and set to 0

Create previousHubDiagonal variable to store whether a previous hub was above (1), on (0), or below (-1) the diagonal line. And initially set it to 0 (since startHub is on the diagonal)

Create previousCoordinates variable and set it to [0, 0] (startHub's coordinates)

For each hub in this path combination

Get this hub's relative (x,y) coordinates

//Now, to find whether this hub is below, on, or above the diagonal, we multiply
//its x-coordinate with the slope of the diagonal (m) to find what its y-coordinate
//would be if it were situated on the line. Then, compare its actual y-coordinate
//with that value

Multiply this hub's x-coordinate with the diagonal's slope (m)

If that value is greater than this hub's Y (it is below the diagonal)

If previousHubDiagonal is 1 (was above diagonal)

Increment crossedDiagonal tally

Set previousHubDiagonal to -1

Else if value calculated is smaller than hub's Y (it is above diagonal)

If previousHubDiagonal is -1 (was below diagonal)

Increment crossedDiagonal tally

Set previousHubDiagonal to 1

//Note that when the hub is on the diagonal, we do not change any variables since
//it has not crossed the diagonal, and we need to retain the variables that indicate
//whether the hub before this one was below or above the diagonal.

//Now, we checked whether the path between the previous hub and this hub is x
//and y monotonic.

If previousCoordinates X > this hub's X

Set xMonotonic to false

If previousCoordinates Y > this hub's Y

Set yMonotonic to false

//Now we determine whether this path combination is valid using the xMonotonic,
//yMonotonic, and crossedDiagonal variables

If xMonotonic *or* yMonotonic is true

If crossDiagonal is less than or equal to 1

Calculate the overall impact of this path (add impacts of each hub in path)

If this path's impact is greater than bestImpact

Set bestPath to this path

Set bestimpact to this path's impact

Return bestPath

Test Plan (Test Cases):

Input Validation

boolean addPostalCode(String postalCode, int population, int area):

- postalCode = null → returns false
- postalCode is empty String → returns false
- postalCode string contains only spaces (i.e., " ") → returns false
- postalCode does not follow postal code format (i.e., starting with a letter, then interchanges between numbers and letters) → returns false [see design decision (4)]
- postalCode already exists (is a duplicate) → returns false
- population < 0 → returns false
- area = 0 → returns false
- area < 0 → returns false

boolean addDistributionHub(String hubIdentifier, Point location, Set<String> servicedAreas):

- hubIdentifier = null → returns false
- hubIdentifier is empty string → returns false
- hubIdentifier string contains only spaces (i.e., " ") → returns false
- hubIdentifier is a duplicate of an existing hubIdentifier → returns false
- location = null → returns false
- location overlaps with an existing hub's location (i.e., a hub already exists at the location's [x,y] coordinates) → returns false [see design decision (8)]
- servicedAreas = null → returns false
- servicedAreas contains one or more postal codes that are null → returns false
- servicedAreas contains one or more postal codes that are empty strings → returns false
- servicedAreas contains one or more postal code strings that contain only spaces (i.e., " ") → returns false
- servicedAreas contains one or more postal codes that do not conform to the Canadian postal code format (i.e., letter-number-letter-number…) → returns false

void hubDamage(String hubIdentifier, float repairEstimate):

- hubIdentifier = null → throws IllegalArgumentException

- hubIdentifier is empty String → throws IllegalArgumentException
- hubIdentifier is string of only spaces (i.e., “ ”) → throws IllegalArguementException
- hubIdentifier does not correspond to an existing hub added through addDistributionHub → throws IllegalArgumentException
- repairEstimate < 0 → throws IllegalArgumentException
- repairEstimate = 0 → throws IllegalArgumentException

void hubRepair(String hubIdentifier, String employeeId, float repairTime, boolean inService):

- hubIdentifier = null → throws IllegalArgumentException
- hubIdentifier is empty String → throws IllegalArgumentException
- hubIdentifier is string of only spaces (i.e., “ ”) → throws IllegalArguementException
- hubIdentifier does not correspond with an existing hub → throws IllegalArgumentException
- employeeId = null → throws IllegalArgumentException
- employeeId is empty String → throws IllegalArgumentException
- employeeId is String containing only spaces (i.e., “ ”) → throws IllegalArgumentException
- repairTime < 0 → throws IllegalArgumentException

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- limit < 0 → throws IllegalArgumentException
- limit = 0 → throws IllegalArgumentException

List<HubImpact> fixOrder(int limit):

- limit < 0 → throws IllegalArgumentException
- limit = 0 → throws IllegalArgumentException

List<Integer> rateOfServiceRestoration(float increment):

- increment = 0 → throws IllegalArgumentException
- increment < 0 → throws IllegalArgumentException
- increment > 1.0 → throws IllegalArgumentException [see design decision (18)]
- increment < 0.001 → throws IllegalArgumentException (granularity issue) [see design decision (19)]

List<HubImpact> repairPlan(String startHub, int maxDistance, float maxTime):

- startHub = null → throws IllegalArgumentException
- startHub is empty String → throws IllegalArgumentException
- startHub is String of empty spaces (i.e., " ") → throws IllegalArgumentException
- startingHub does not correspond to an existing hub → throws IllegalArgumentException
- maxDistance < 0 throws IllegalArgumentException
- maxTime < 0 throws IllegalArgumentException

List<String> underservedPostalByPopulation(int limit):

- limit < 0 throws IllegalArgumentException
- limit = 0 throws IllegalArgumentException

List<String> underservedPostalByArea(int limit):

- limit < 0 throws IllegalArgumentException
- limit = 0 throws IllegalArgumentException

Boundary Cases

boolean addPostalCode(String postalCode, int population, int area):

- postalCode is String containing only one letter (i.e., "B") → adds postal code and returns true
- postalCode has spaces between its characters (i.e., "B 3 J" or "B3J 1H4") → parse out the spaces and add postal code (i.e., adds "B3J1H4"), then returns true [see design decision (5)]
- population = 0 → adds postal code and returns true (this is because some postal codes may not have a permanent population like a rural field with an observation station within it that needs power)
- area = 1 → adds postal code and returns true

boolean addDistributionHub(String hubIdentifier, Point location, Set<String> servicedAreas):

- hubIdentifier is String containing only one character (i.e., "1" or "a") → adds distribution hub and returns true
- servicedAreas contains only one postal code → adds distribution hub and returns true
- servicedAreas is empty set → adds distribution hub and returns true [see design decision (6)]
- servicedAreas contains one or more postal codes that do not exist (have not already been added) → adds distribution hub and returns true [see design decision (7)]

void hubDamage(String hubIdentifier, float repairEstimate):

- repairEstimate = 1 → updates hubs damage

void hubRepair(String hubIdentifier, String employeeId, float repairTime, boolean inService):

- repairTime = 0 → updates hub's status [see design decision (10)]
- employeeId is String containing a single character → updates hub's status

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- limit =1 → returns list of most damaged postal codes (size of list is variable [see design decision (15)])

List<HubImpact> fixOrder(int limit):

- limit = 1 → returns list of most impactful hubs (size of list is variable [see design decision (15)])

List<Integer> rateOfServiceRestoration(float increment):

- increment = 1.0 → returns list of size 2, where first Integer is 0% restoration and second is 100% restoration
- increment = 0.001 → returns list of size 1001

List<HubImpact> repairPlan(String startHub, int maxDistance, float maxTime):

- maxDistance = 0 → returns a list containing only the startHub's HubImpact [see design decision (27)]
- maxTime = 0 → returns a list containing only the startHub and endHub's HubImpacts [see design decision (28)]

List<String> underservedPostalByPopulation(int limit):

- limit = 1 → returns list of underserved postal codes based on population (size of list is variable [see design decision (15)])

List<String> underservedPostalByArea(int limit):

- limit = 1 → returns list of underserved postal codes based on area (size of list is variable [see design decision (15)])

Control Flow

PowerService constructor():

- Properties file (credentials.prop) cannot be accessed → throws IOException
- Database connection cannot be established using username, password, and database values supplied in properties file (credentials.prop) → throws SQLException
- SQL select command on PostalCodes table fails → throws SQLException
- SQL select command on DistributionHubs table fails → throws SQLException
- SQL select command on PostalHubRelation join DistributionHubs tables fails → throws SQLException

boolean addPostalCode(String postalCode, int population, int area):

- SQL insert command on PostalCodes table fails → returns false
- SQL select command on PostalHubRelation join DistributionHubs tables failed → returns false
- *Some covered in data flow*

boolean addDistributionHub(String hubIdentifier, Point location, Set<String> servicedAreas):

- SQL insert command on DistributionHubs table fails → returns false
- SQL select command on PostalHubRelation table fails → returns false
- SQL insert command on PostalHubRelation table fails → returns false
- *Some covered in data flow*

void hubDamage(String hubIdentifier, float repairEstimate):

- Call on a hub that services no postal codes → updates hub's inService and repairEstimate attributes accordingly
- Call on a hub that services one or more postal codes → updates hub's inService, repairEstimate, then calculates and updates hub's populatioEffected (number of people effected by its outage) and impact (populationEffected/repairEstimate). Finally, the method recalculates and updates each of the serviced postal code's updated repair time estimate.
- Call on a hub that was in service → sets hub's inService attribute to false and sets its repairEstimate to the value provided (i.e., 0 + repairEstimate)
- Call on a hub that was out of service → increments its repairEstimate by the value provided (i.e., previousRepairEstimae + repairEstimate)
- SQL update command on DistributionHubs table fails → throws SQLException
- SQL select command on PostalCodes table fails → throws SQLException
- SQL select command on PostalHubRelation table fails → throws SQLException
- SQL select command on PostalHubRelation join Distributionhubs tables fails → throws SQLException
- *Some covered in data flow*

<u>void hubRepair(String hubIdentifier, String employeeId, float repairTime, boolean inService):</u>

- Call on a hub that is already in service → throws IllegalArgumentException [see design decision (11)]
- Call on hub that services no postal codes → updates hub's repair time estimate and in service attributes accordingly
- Call on hub that services one or more postal codes → updates hub's repair time estimate and in service attributes, then recalculates and updates hub's impact. Then, the method recalculates and updates each of the hub's postal codes' repair time estimates.
- inService = true → sets hub's inService to true, its populationEffected to 0, and its impact to 0 (regardless of the repairTime argument). Then, updates each of the hub's postal codes' repair time estimates to reflect this hub is back online.
- inService = false → sets hub's inService to false, recalculates and updates its populationEffected and impact attributes, then updates each of the hub's postal codes' repair time estimates to reflect the change in the hub's remaining repair time estimate.
- repairTime > hub's repair time estimate → does not update hub's repair time estimate [see design decision ()], recalculates and updates hub's populationEffected and impact (just to make sure database and program are in sync), then recalculates and updates each of the hub's postal codes' repair time estimates (just to make sure database and program in sync).

- repairTime = hub's repair time estimate → does not update hub's repair time estimate [see design decision ()], recalculates and updates hub's populationEffected and impact (just to make sure database and program are in sync), then recalculates and updates each of the hub's postal codes' repair time estimates (just to make sure database and program in sync).
- repairTime < hub's repair time estimate → sets hub's repair time estimate to (initial value – repairTime), recalculates and updates hub's populationEffected and impact (just to make sure database and program are in sync), then recalculates and updates each of the hub's postal codes' repair time estimates to reflect hub's new repair time estimate.
- SQL update command on DistributionHubs table fails → throws SQLException
- SQL insert command on RepairLog table fails → throws SQLException
- SQL select command on PostalHubRelation join DistributionHubs tables fails → throws SQLException
- SQL select command on PostalCodes table fails → throws SQLException
- SQL select command on PostalHubRelation table fails → throws SQLException
- *Some covered in data flow*

<u>int peopleOutOfService():</u>

- Call when all postal codes are serviced by at least one hub and:
    - At least one hub servicing a postal is damaged → returns number of people out of service due to hub outages
    - Only hubs that do not service any postals are damaged → returns 0
    - No hubs are damaged → returns 0
- Call when at least one postal code is not serviced by any hub and:
    - No hubs are damaged → returns sum of populations of postal code that are not serviced by any hubs
    - At least one hub that services a postal code is damaged → returns (sum of populations of postal codes that are not serviced by any hubs) + (number of people out of service due to hub outages)
- Call when all hubs are damaged → returns sum of all postal code populations
- Call when all postal codes have a population of 0 → returns 0
- Call when number of people out of service is not a whole number → rounds that number *up* to the nearest digit and returns it (i.e., 10.5 is rounded to 11, then returned) [see design decision (14)]
- SQL select command on PostalHubRelation join DistributionHubs tables fails → throws SQLException

- SQL select command on PostalCodes left join PostalHubRelation tables fails → throws SQLException
- *Some covered in data flow*

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- Call when all postal codes are serviced by at least one hub and:
    - No hubs are damaged → returns empty list
    - At least one hub is damaged → returns ordered list of postal codes based on their repair time estimates
- Call when none of the postal codes are serviced by any hubs and:
    - No hubs are damaged → returns empty list [see design decision (17)]
    - All hubs are damaged → returns empty list [see design decision (17)]
- Call with a limit that creates tie(s) between two or more serviced postal code repair times at the limit → add all tied serviced postal codes passed the limit to the list [see design decision (15)]
- Call with a limit that does not create any ties at the limit → returns list of size <limit>
- SQL select command on PostalHubRelation join DistributionHubs tables fails → throws SQLException
- *Some covered in data flow*

List<HubImpact> fixOrder(int limit):

- Call when no hubs are damaged → returns empty list
- Call with a limit that creates tie(s) between two or more damaged hub impacts at the limit → add all tied damaged hubs passed the limit to the list [see design decision (15)]
- SQL select command on PostalHubRelation join DistributionHubs tables fail → throws SQLException
- *Some covered in data flow*

List<Integer> rateOfServiceRestoration(float increment):

- Call when all postal codes are serviced by at least one hub and:
    - No hubs are damaged → returns a list containing all 0s
    - One or more hubs are damaged → returns a list of repair hours depending on the increment supplied
- Call when no postal codes are serviced by any hubs:

    - No hubs are damaged → returns list containing all 0s
    - One or more hubs are damaged → returns list containing all 0s [see design decision (20)]
- Call when one or more repair hours in the returned list is not a whole number → rounds *up* all such repair hours to nearest digit (i.e., a list with repair hours [0, 3.5, 5] is converted to [0, 4, 5])
- SQL select command on PostalHubRelation join DistributionHubs tables fails → throws SQLException
- *Some covered in data flow*

List<HubImpact> repairPlan(String startHub, int maxDistance, float maxTime):

- Call when startHub is in service → throws an IllegalArugmentException (see design decision [31])
- maxDistance = 0 → returns list containing only startHub
- maxTime = 0 and at least one damaged hub in range of startHub → returns list containing only startHub and endHub
- Call with a maxTime and maxDistance that would result in multiple damaged hubs in range and within the alotted time of startHub where:
    - Only one hub resides between startHub and endHub → returns a list of startHub, one intermediate hub, endHub
    - Multiple hubs reside between startHub and endHub → returns a list of startHub, best path among intermediate hubs, endHub
    - No hubs reside between startHub and endHub → returns list containing only startHub and endHub
- *Some covered in data flow*

List<String> underservedPostalByPopulation(int limit):

- Call when no postal codes are serviced by any hubs → returns list of all postal codes since they are all equally underserved [see design decision (22)]
- Call when there is a mix between serviced and un-serviced postal codes → returns ordered list of postal codes based on hubs per capita, with all un-serviced postal codes at the beginning of the list, regardless of the limit, since they are all tied for being the most underserved postal [see design decision (22)]
- Call when all postal codes are serviced by at least one hub → returns ordered list of postal codes based on hubs per capita
- Call with a limit that causes a tie between postals at the limit → add all tied postal codes passed the limit to the returned list [see design decision (15)]

- SQL select command on PostalHubRelation fails → throw SQLException
- *Some covered in data flow*

List<String> underservedPostalByArea(int limit):

- Call when no postal codes are serviced by any hubs → returns list of all postal codes since they are all equally underserved [see design decision (22)]
- Call when there is a mix between serviced and un-serviced postal codes → returns ordered list of postal codes based on hubs per area, with all un-serviced postal codes at the beginning of the list, regardless of the limit, since they are all tied for being the most underserved postal [see design decision (22)]
- Call when all postal codes are serviced by at least one hub → returns ordered list of postal codes based on hubs per area
- Call with a limit that causes a tie between postals at the limit → add all tied postal codes passed the limit to the returned list [see design decision (15)]
- SQL select command on PostalHubRelation fails → throw SQLException
- *Some covered in data flow*

Data Flow

Note: that the phrases "before adding any postal codes" and "before adding any distribution hubs" refers to these entities not being present in the database (i.e., not just whether a postal code or hub has been added during this execution of the program).

boolean addPostalCode(String postalCode, int population, int area):

- Add a postal code after the hub that is servicing it:

  [Ex. addDistributionHub(hubId, location, {"B3J"}) > addPostalCode("B3J", pop., area)]

  → Adds the postal code, calculates and sets the new postal code's repair time estimate (in case the hub is also damaged), then recalculates and updates the corresponding hub's populationEffected and impact to account for this new postal code (in case the hub is also damaged). Additionally, it updates the PostalHubRelation table to reflect the connection between the hub and postal code.

boolean addDistributionHub(String hubIdentifier, Point location, Set<String> servicedAreas):

- Add a hub after the postal code it services:

  [Ex. addPostalCode("B3J", pop., area) > addDistributionHub(hubId, location, {"B3J"}]

→ Adds the distribution hub and updates the PostalHubRelation table to reflect the connection between the hub and postal code.

void hubDamage(String hubIdentifier, float repairEstimate):

- Call twice on the same hub → updates that hub's repairEstimate, populationEffected, impact, and each of the hub's serviced postal codes' repair time estaimtes twice based on the repairEstimate arguments

void hubRepair(String hubIdentifier, String employeeId, float repairTime, boolean inService):

- Call twice on same hub (that is initially out of service) without any hubDamage calls between them where:
    - First call's inService is true → second call will always throw an exception since the second hubRepair is being called on a hub that is in service [see design decision (11)]
    - First call's inService is false and second call's inService is false → hub will ultimately be out of service with a repairEstimate according to the the repairTime arguments supplied in each call (see control flow for how repairEstimates are updated)
    - First call's inService is false and second call's inService is true → hub will ultimately be in service with a repairEstimate, effectedPopulation, and impact of 0 (see control flow for how a true inService argument is handled)

int peopleOutOfService():

- Call when no postal codes exist → returns 0
- Call after adding postal codes, but before adding distribution hubs → returns sum of all postal code populations (since they are all effectively being serviced by no hubs [see how such postal codes are handled in control flow])

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- Call after adding postal codes, but before adding any hubs → returns empty list [see design decision (17)]
- Call after adding hubs, but before adding any postal codes → returns empty list

List<HubImpact> fixOrder(int limit):

- Call before adding any hubs → returns empty list
- Call after adding hubs, but before adding any postal codes → returns list of all damaged hubs since they all have a tied impact of 0

List<Integer> rateOfServiceRestoration(float increment):

- Call before adding any hubs → throws IllegalArgumentException [see design decision (22)]
- Call after adding hubs, but before adding any postal codes → returns list containing all 0s

List<HubImpact> repairPlan(String startHub, int maxDistance, float maxTime):

- Call after adding postal codes, but before adding any hubs → always throws an IllegalArgumentException (since startHub will never correspond to an existing hub)
- Call after adding hubs, but before calling any hubDamages (i.e., when no hubs are out of service) → always throws an IllegalArguementException (since startHub will always be in service)
- Call after adding hubs and hub damages, but before adding any postal codes → returns best repair plan path from startHub to endHub [see design decision (25) and (32) for details on how 0 impact damaged hubs and tied damaged hubs are handled]

List<String> underservedPostalByPopulation(int limit):

- Call after adding postal codes, but before adding any hubs → returns list of all postal codes since they are all equally underserved [see design decision (22)]
- Call before adding any postal codes → returns empty list

List<String> underservedPostalByArea(int limit):

- Call after adding postal codes, but before adding any hubs → returns list of all postal codes since they are all equally underserved [see design decision (22)]
- Call before adding any postal codes → returns empty list