CSCI 3901: Project: Milestone 2

Waleed R. Alhindi (B00919848)

Blackbox Test Cases:

Input Validation

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

- postalCode = null → returns false
- postalCode is empty String → returns false
- postalCode does not follow postal code format (i.e., starting with a letter, then interchanges between numbers and letters) → returns false
- postalCode already exists (is a duplicate) → returns false
- postalCode overlaps with existing postal code (i.e., adding "B3J" when "B3J1H4" already added) → 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 is string containing 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 → returns false
- servicedAreas = null → returns false
- servicedAreas is empty → returns false
- servicedAreas contain duplicate or overlapping postalCodes → returns false

  Note: although a set cannot contain duplicates, the set can contain, for example, "B 3 J" and "B3J", which this method will treat as duplicates
- servicedAreas contains one or more postal codes that do not exist (i.e., a postal code that has not been added using addPostalCode) → returns false

void hubDamage(String hubIdentifier, float repairEstimate):

- hubIdentifier = null → throws IllegalArgumentException
- hubIdentifier does not correspond to an existing hub added through addDistributionHub → throws IllegalArgumentException

  Note: this also covers the cases where hubIdentifier is an empty string since it will always not correspond with an existing hub since addDistributionHub throws an exception when trying to add a hub with an empty string identifier.
- repairEstimate < 0 → throws IllegalArgumentException

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

- hubIdentifier = null → throws IllegalArgumentException
- hubIdentifier does not correspond with an existing hub → throws IllegalArgumentException

  Note: this also covers the cases where hubIdentifier is an empty string since it will always not correspond with an existing hub since addDistributionHub throws an exception when trying to add a hub with an empty string identifier.
- 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

List<HubImpact> fixOrder(int limit):

- limit < 0 → throws IllegalArgumentException

List<Integer> rateOfServiceRestoration(float increment):

- increment = 0 → throws IllegalArgumentException
- increment < 0 → throws IllegalArgumentException
- increment > 1.0 → throws IllegalArgumentException (this is under the assumption that increment must be between 0 and 1.0 since it is a percentage, such as 0.05 which correlates to 5%)

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

- startingHub = null → throws IllegalArgumentException
- startingHub does not correspond to an existing hub → throws IllegalArgumentException

  Note: this also covers the cases where hubIdentifier is an empty string since it will always not correspond with an existing hub since addDistributionHub throws an exception when trying to add a hub with an empty string identifier.
- maxDistance < 0 throws IllegalArgumentException
- maxTime < 0 throws IllegalArgumentException

List<String> underservedPostalByPopulation(int limit):

- limit < 0 throws IllegalArgumentException

List<String> underservedPostalByArea(int limit):

- 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
- 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)

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

- servicedAreas contains only one postalCode → adds distribution hub and returns true

void hubDamage(String hubIdentifier, float repairEstimate):

- repairEstimate = 0 → does nothing but does not throw an exception

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

- repairTime = 0 → updates hub's status (maybe a hub was incorrectly labeled as out of service, so this method is called with repairTime = 0 and inService = true to reflect the correction)
- employeeId is String containing a single character → updates hub's status

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- limit = 0 → returns empty list (alternatively this could throw an exception to indicate to the user what went wrong, but I will implement this to return an empty list for the time being)

List<HubImpact> fixOrder(int limit):

- limit = 0 → returns empty list (alternatively this could throw an exception to indicate to the user what went wrong, but I will implement this to return an empty list for the time being)

List<Integer> rateOfServiceRestoration(float increment):

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

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

- maxDistance = 0 → returns a list containing only the startHub's HubImpact
- maxTime = 0 → returns a list containing only the startHub and endHub's HubImpacts

List<String> underservedPostalByPopulation(int limit):

- limit = 0 → returns empty list (alternatively this could throw an exception to indicate to the user what went wrong, but I will implement this to return an empty list for the time being)

List<String> underservedPostalByArea(int limit):

- limit = 0 → returns empty list (alternatively this could throw an exception to indicate to the user what went wrong, but I will implement this to return an empty list for the time being)

Control Flow

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

- *Covered in data flow*

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

- *Covered in data flow*

void hubDamage(String hubIdentifier, float repairEstimate):

- hubIdentifier corresponds to an existing hub and repairEstimate >= 0 → updates the hub's status and updates all the hub's serviced postal codes to reflect its status (i.e., update corresponding DamagedPostalCodes' repairEstimates)

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

- hubIdentifier corresponds to an existing hub that is damaged → updates that hub's status and updates all the hub's serviced postal codes to reflect status
    - inService = true → updates corresponding postal codes to reflect the how much of their populations have regained power
    - inService = false → updates corresponding postal codes' repair time estimates

int peopleOutOfService():

- Call when no hubs are damaged → returns 0
    - Call after all hubs have been restored using hubRepair → returns 0
    - Call before any hub damages have been reported using hubDamage → returns 0
- *Some covered in data flow*

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- Call when hubs are damaged → returns empty list
    - Call after all hubs have been restored using hubRepair → returns empty list
    - Call before any hub damages have been reported using hubDamage → returns empty list
- limit = number of damaged postal codes → returns list containing all damaged postal codes
- limit > number of damaged postal codes → returns a list of size <count(damaged postal code)> containing all damaged postal codes
- *Some covered in data flow*

List<HubImpact> fixOrder(int limit):

- Call when no hubs are damaged → returns empty list
    - Call after all hubs have been restored using hubRepair → returns empty list
    - Call before any hub damages have been reported using hubDamage → returns empty list
- limit = number of damaged hubs → returns list containing all damaged hubs
- limit > number of damaged hubs → returns a list of size <count(damaged hubs)> containing all damaged hubs
- *Some covered in data flow*

List<Integer> rateOfServiceRestoration(float increment):

- Call when no hubs are damaged → returns list of size <1/increment> where each entry is 0 (i.e., for increment = 0.5 → List = <0, 0>)
    - Call before reporting any hub damages using hubDamage → returns list of size <1/increment> containing 0s
    - Call after all hubs are restored using repairHub → returns list of size <1/increment> containing 0s
- *Some covered in data flow*

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

- Call when startHub is inService (i.e., it is not damaged) → throws an IllegalArgumentException (this is because it would not make sense for a repair plan to start at a running hub)
- Call when no other hubs reside within <maxDistance> of startHub → returns list containing only startHub
- Call when only one damaged hub resides within <maxDistance> of startHub → the other damaged hub is deemed as endHub and the method returns a list containing only the startHub and endHub
- Call when only running hubs reside within <maxDistance> of startHub (i.e., all other hubs in range are up and running) → returns list containing only startHub
- Call when multiple damaged hubs reside within <maxDistance> of startHub → return list of hubs depending on their individual repairTimes, impacts, and the maxTime parameter passed
- Call when mutltiple intermediate hubs can be reached while only crossing the diagonal once and staying monotonic → returns list of startHub, <reachable intermediate hubs>, endHub

- Call when when only one intermediate hub can be reached while only crossing the diagonal once and staying montonic → returns list of startHub, intermediate hub, and endHub (size = 3)
- *Some covered in boundary cases*
- *Some covered in data flow*

List<String> underservedPostalByPopulation(int limit):

- Call with a limit that causes a tie at the limit → add tied postal codes at limit to list (like in assignment 3) [Alternatively, the tie can be broken either alphabetically or by area. This decision will be made after further clarification].
- Limit = number of postal code → returns a list containing all postal codes
- Limit > number of postal codes → returns a list of size <count(postal codes)> containing all postal codes
- *Some covered in data flow*

List<String> underservedPostalByArea(int limit):

- Call with a limit that causes a tie at the limit → add tied postal codes at limit to list (like in assignment 3) [Alternatively, the tie can be broken either alphabetically or by population. This decision will be made after further clarification].
- Limit = number of postal code → returns a list containing all postal codes
- Limit > number of postal codes → returns a list of size <count(postal codes)> containing all postal codes
- *Some covered in data flow*

Data Flow

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

- Add first postal code → adds first postal code and returns true
- Add second postal code (i.e., call method twice) → adds second postal code and returns true

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

- Call before adding postal codes → returns false
- Add first distribution hub after adding postal code(s) → adds hub and returns true

- Add second distribution hub (i.e., call method twice) after adding postal codes → adds second hub and returns true

void hubDamage(String hubIdentifier, float repairEstimate):

- Call before adding any postal codes → throws an exception (type of exception to be determined)
- Call after adding postal codes, but before adding any hubs → throws an exception (type of exception to be determined)

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

- Call before adding any postal codes → throws an exception (type of exception to be determined)
- Call before adding any hubs → throws an exception (type of exception to be determined)
- Call after adding hubs, but before hubDamage → throws an exception (type of exception to be determined)
- Call twice on same hub without any hubDamage calls between them:
    - If first hubRepair makes inService true, and second hubRepair makes inService true → does nothing but does not throw exception
    - If first hubRepair makes inService true, and second hubRepair makes inService false → throws an exception (this is because it would denote that a hub is damaged without supplying a repair time estimate) [type of exception to be determined]
    - If first hubRepair makes inService false, and second hubRepair makes inService true → updates hub's status
    - If first hubRepair makes inService false, and second hubRepair makes inService false → updates hub's status

int peopleOutOfService():

- Call before adding any postal codes → returns 0 (no postal codes added means no populations added, meaning that no one is recorded as being out of service. Alternatively, this can throw an exception. That decision will be determined upon further clarification)
- Call after adding postal codes, but before adding any hub → returns 0 (this assumes that if no hubs were added, and thus no hub damages recorded, that all postal codes have power. Alternatively, this type of call could also throw an exception. That decision will be determined upon further clarification)

List<DamagedPostalCodes> mostDamagedPostalCodes(int limit):

- Call before adding any postal codes → throws an exception (type of exception to be determined)
- Call after adding postal codes, but before adding any hubs → returns an empty list (this assumes that if no hubs were added, and thus no hub damages recorded, that all postal codes have power. Alternatively, this type of call could also throw an exception. That decision will be determined upon further clarification)

List<HubImpact> fixOrder(int limit):

- Call before adding any postal codes → returns empty list (no postal codes means that there cannot be any damaged postal codes. Alternatively, this type of call could also throw an exception. That decision will be determined upon further clarification)
- Call after adding postal codes, but before adding any hubs → returns empty list (no hubs added, means no outages are impacting postal codes. Alternatively, this type of call could also throw an exception. That decision will be determined upon further clarification)

List<Integer> rateOfServiceRestoration(float increment):

- Call before adding any postal codes → throws an exception (Since no postal codes added, the sum of populations = 0. And we cannot operate on 0 to indicate percent of population that are getting their power back. Since this is a mathematical problem, it would be better to throw an exception than to return an empty list) [type of exception to be determined]
- Call after adding postal codes, but before adding any hubs → throws an exception for the same reason as the above case [type of exception to be determined]

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

- Call before adding any postal codes → throws an exception (this is because no hubs can be added if no postal codes have been added, thus startHub will never correspond with an existing hub) [type of exception to be determined]
- Call after adding postal codes, but before adding hubs → throws an exception (startHub will never correspond with an existing hub) [type of exception to be determined]
- Call after adding hubs, but before hub damage → returns an empty list (this is because no hubs require repairs, thus there is no repair plan. Alternatively, this type of call could also throw an exception. That decision will be determined upon further clarification)

List<String> underservedPostalByPopulation(int limit):

- Call before adding any postal codes → throws an exception (an exception is better here than returning an empty list, because an empty list can be interpreted as there being no

underservedPostalCodes rather than denoting that the calculations could not be performed) [type of exception to be determined]

- Call after adding postal codes, but before adding hubs → throws an exception for the same reason as the above case [type of exception to be determined]

List<String> underservedPostalByArea(int limit):

- Call before adding any postal codes → throws an exception (an exception is better here than returning an empty list, because an empty list can be interpreted as there being no underservedPostalCodes rather than denoting that the calculations could not be performed) [type of exception to be determined]
- Call after adding postal codes, but before adding hubs → throws an exception for the same reason as the above case [type of exception to be determined]

Plan and Timeline:

My plan is to compose a high-level design of how the program should operate, store data, relate data, and handling error states. This high-level design will be composed around using a database to store, retrieve, and group information; so, interface classes will also be outlined. Furthermore, because DBMSs can handle many of the grouping operations needed throughout this program, such operations will be excluded from the initial design of the program's methods. For example, instead of grouping distribution hubs to their postal codes manually in mostDamagedPostalCodes, the design will grant the assumption that such groupings will be handled by SQL queries.

With that being said, the first draft implementation that I am currently working on uses data structures instead of a database to store, organize, and group data. This is because we are still covering databases in class, and I wanted to try my hand at implementing my initial design to identify areas of concern or things I might not have thought of. However, this initial implementation isolated these data structures to simulate a database interface.

Going forward, I will continue working on my first draft implementation throughout the next week. Next week, I will start over from scratch, this time using a database instead of data structures. This would mean that I would have completed Lab 9 by then, which concerns JDBC, meaning I'd have a better understanding of how to incorporate SQL into Java programs. The reason I will start from scratch is because my initial implementation served more as "scratch paper" that I was just trying out approaches on.

My next iteration of this implementation will start with the implementation of data methods first (i.e., addPostalCode, addDistributionHub, hubRepair, and hubDamage). I believe this part should take about ½ - ¾ of a week to fully implement. Then, I will move on to the following reporting methods (peopleOutOfService, mostDamagedPostalCodes, fixOrder, rateOfServiceRestoration, underservedPostalbyPopulation, and underservedPostalByArea). I estimate these methods should take me about a week (or maybe a day or two) over to fully implement. Note that this does not

include repairPlan as I believe that it will be the most difficult to implement and, as such, should be given a lot of focus and attention. Although I will work on implementing this method while implementing the other control methods, I am anticipating that I will need to dedicate extra time solely to the repairPlan method. As such, my plan accounts for the extra time I will probably need by allotting an extra week to work on repairPlan. This would leave me with about a week before the final deadline (December 15), where I can refactor and finalize my code.

The following are my estimates of how long each method should take to implement (the order in which they are listed is also the order in which I intend to implement them):

addPostalCode and addDistributionHub:

- Estimate: 1 day to implement both

hubDamage and hubRepair:

- Estimate: 1-2 days to implement both

peopleOutOfService, mostDamagedPostalCodes, underservedPostalByPopulation, and underservedPostalByArea:

- Estimate: 2-4 days to implement them all

rateOfServiceRestoration and fixOrder:

- Estimate: 2-3 days to implement both

repairPlan:

- Estimate: Up to a week to implement

  Note: I will be trying to interleave the implementation of this method throughout the time slots above whenever possible since I believe I will need to continuously work on this method to get it confidently working by Dec 7~8