LiveLong

System Requirements Specification

COMPSCI 2XB3, L02, G08
Kenneth Mak - 001318946

March 8, 2020

Contents

1 Domain

LiveLong is an application that provides a list of nursing homes to a client, ranked by their current rating conducted in surveys by the official U.S. government Medicare. The results displayed will change based on the client's location and their filter settings, such as setting the maximum distance or the minimum rating when searching.

The final results will also include provide warnings of any previous case of staff abuse, and any official government warnings of Fire Safety or Health deficiencies given to the nursing home, including the date of assignment and the current status since then.

The stakeholders of this application can then be assumed to be as follows:

1. Members of the general public who are searching for a nursing home.
2. Doctors, nurses, or some other medical professional requiring a nursing home for a patient requiring long-term care.
3. Staff or owners of the nursing home itself to view current ratings
4. Government workers looking into the current deficiencies of a nursing home and its current status.

Each of the stakeholders will in some way be affected by this application. Those searching for an appropriate nursing home within a certain distance will be able to quickly find a list of nursing homes for them to start researching on*, as well as have any noticeable concerns available for them to look at. It will greatly speed up the process of choosing a nursing home and give then choices that are officially backed by Medicare.

Staff or owners of a nursing home will be able to view their current appearance to the general public, and their ranking as compared to other nearby nursing homes. They will be able to see which areas require improvement and attention, and subsequently contact the government if certain deficiencies have been handled.

Finally, official government workers will be able to search for specific Nursing Homes based on their ID or name, and check if the current status deficiencies that the nursing home still has**.

*As an emphasis, this application is intended to provide quick results based on a user's specific filter, and point out any general concerns. Additional research on a is highly recommended.

** Current status of deficiencies is updated monthly on the Medicare official databases.

2 Functional Requirements

2.1 NursingHome ADT Module

Module

NursingHome

Description

NursingHome ADT will store information about a nursing home. It is constructed using the data parsed with DataReader. Any warnings (i.e. Fire Safety or Health deficiencies) will also be stored in this object.

Uses

FireSafetyDeficiency, HealthDeficiency

Syntax

Exported Types

NursingHome = ?

Exported Access Programs

Routine name	In	Out	Exceptions
NursingHome	*seq of String*	NursingHome	NoValidIdentifier
accessors		*relevantType*	
addFSDef	FireSafetyDeficiency		
addHDef	HealthDeficiency		

Semantics

State Variables

id, name, contact information, address, ratings, flags, etc.: $String/\mathbb{Z}/\mathbb{B}$
$fsList$: *seq of* FireSafetyDeficiency
hList: *seq of* HealthDeficiency

State Invariant

None

Access Routine Semantics

NursingHome(*seq of String*):

- transition: constructs a NursingHome from a sequence of Strings that has been split by a comma from the CSV.
- output: *out* :=NursingHome
- exception: No value for NH.id $\rightarrow$ NoValidIdentifier

accessors():

- refers to all relevant accessor methods referring to any variable inside NursingHome (i.e. getID(), getName(), getAbuseFlag(), getFSList(), etc.)

addFSDef(*fsd*):

- transition: adds the FireSafetyDeficiency fsd to this object's fsList

addHDef(*hd*):

- transition: adds the HealthDeficiency hd to this object's hList

2.2 NHGeoInfo ADT Module

Module

NHGeoInfo

Description

NHGeoInfo ADT will store the geographical information about a nursing home. It is constructed using the data parsed with DataReader. It contains the longitude and latitude for a NursingHome object, if that NursingHome object and this have the same ID.

Uses

FireSafetyDeficiency, HealthDeficiency

Syntax

Exported Types

NursingHome = ?

Exported Access Programs

Routine name	In	Out	Exceptions
NHGeoInfo	*seq of String*	NHGeoInfo	NoValidIdentifier
accessors		*relevantType*	
updateDistance	$\mathbb{Z}$, $\mathbb{Z}$		

Semantics

State Variables

id: *String*
lat: $\mathbb{Z}$
lon: $\mathbb{Z}$
dist: $\mathbb{Z}$

State Invariant

$id := null$

Access Routine Semantics

NHGeoInfo(*seq of String*):

- transition: constructs a NHGeoInfo from a sequence of Strings that has been split by a comma from the CSV.
- output: *out* := NHGeoInfo
- exception: No value for NH.id → NoValidIdentifier

accessors():

- refers to all relevant accessor methods referring to any variable inside NHGeoInfo (i.e. getID(), getLat(), getLon(), etc.)

updateDistance(*latitude, longitude*):

- transition: *dist* := distance(*latitude*, *longitude*)

Local Function

distance($lat2$, $lon2$) - Start connection to GoogleAPI maps.
distance($lat2$, $lon2$) ≡ Calculate distance between (lat, lon) with $lat2, lon2$

2.3 Generic Deficiency Module

Generic Template Module

Deficiency

Uses

None

Syntax

Exported Types

Deficiency = ?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Deficiency	*seq of* String	Deficiency	NoValidIdentifier
accessors		$relevantType$	
isActive		$\mathbb{Z}$	

Semantics

State Variables

id, deficiency date, category, description, status: *String*

State Invariant

$id := null$

Assumptions

Superclass of FireSafetyDeficiency and HealthDeficiency. Abstract, should ideally not be implemented on its own.

Access Routine Semantics

Deficiency(*seq of String*):

- transition: constructs a Deficiency from a sequence of Strings that has been split by a comma from the CSV. Requires a value to be found for *id* and *name* in order to add this to the correct NursingHome.
- output: *out* := Deficiency
- exception: No value for *id* $\rightarrow$ NoValidIdentifier

accessors():

- refers to all relevant accessor methods referring to any variable inside NHGeoInfo (i.e. getID(), getStatus(), etc.)

isActive():

- output: *out* := (*status* $\neq$ *Waiver has been granted* $\wedge$ *status* $\neq$ *Deficient, provider has date of correction*)

2.4 FireSafetyDeficiency ADT Module

ADT Module

FireSafetyDeficiency extends Deficiency

2.5 HealthDeficiency ADT Module

ADT Module

HealthDeficiency extends Deficiency

2.6 DataReader Module

Module

DataReader

Description

This module is used for parsing csv files and returning the data to an appropriate location. The data that this module is intended to go through is the **Provider Information** dataset, the **Provider Geo Information** dataset, the **Fire Safety Deficiencies** dataset, and the **Health Deficiencies** dataset.

Uses

NursingHome, FireSafetyDeficiency, HealthDeficiency

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
readProviderInfo	s: *String*	*seq of* NursingHome	IOException
readProviderGeoInfo	s: *String*	*seq of* NHGeoInfo	IOException
readFSDeficiency	s: *String*	*seq of* FireSafetyDeficiency	IOException
readHDeficiency	s: *String*	*seq of* HealthDeficiency	IOException

Semantics

State Variables

None

State Invariant

None

Access Routine Semantics

readProviderInfo(s):

- Given a string s, find the file s and attempt to parse through all rows. Construct a NursingHome object for each row, and place in a sequence. Upon completion, return the sequence. Sequence will be ordered later when inserting into binary tree.
- output: *out* := *seq of* NursingHome
- exception: Unable to find file → IOException

readProviderGeoInfo(s):

- Given a string s, find the file s and attempt to parse through all rows. Construct a NHGeoInfo object for each row, and place in a sequence. Upon completion, return the sequence. Sequence will be ordered later when inserting into binary tree.
- output: *out* := *seq of* NHGeoInfo
- exception: Unable to find file → IOException

readFSDeficiency(s):

- Given a string s, find the file s and attempt to parse through all rows. Construct a FireSafetyDeficiency object for each row, and place in a sequence. Upon completion, return the sequence. Deficiency sequence will later be parsed and added to its relevant Nursing Home.
- output: *out* := *seq of* FireSafetyDeficiency
- exception: Unable to find file → IOException

readHDeficiency(s):

- Given a string s, find the file s and attempt to parse through all rows. Construct a HealthDeficiency object for each row, and place in a sequence. Upon completion, return the sequence. Deficiency sequence will later be parsed and added to its relevant Nursing Home.
- output: *out* := *seq of* HealthDeficiency
- exception: Unable to find file → IOException

2.7 DataWriter Module

Module

DataWriter

Description

This module is used for writing the objects **NursingHome**, **NHGeoInfo**, **FireSafetyDeficiency** and **HealthSafetyDeficiency** into csv files. This is done to create files that contain only the necessary data, therefore reducing the size of the storage and decreasing load times when initializing the app.

Uses

NursingHome, FireSafetyDeficiency, HealthDeficiency

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
saveProviderInfo	*seq of* NursingHome	CSV File	
saveProviderGeoInfo	*seq of* NHGeoInfo	CSV File	
saveFSDeficiency	*seq of* FireSafetyDeficiency	CSV File	
saveHDeficiency	*seq of* HealthDeficiency	CSV File	

Semantics

Access Routine Semantics

saveProviderInfo(s):

- Given a sequence of NursingHome s, for each NursingHome object, write all of the object's variables into a CSV file, delimited by the ',' sign.
- output: *out* := CSV File

- exception: None

saveProviderGeoInfo(s):

- Given a sequence of NHGeoInfo s, for each NHGeoInfo object, write all of the object's variables into a CSV file, delimited by the ',' sign.
- output: out := CSV File
- exception: None

saveFSDeficiency(s):

- Given a sequence of FireSafetyDeficiency s, for each FireSafetyDeficiency object, write all of the object's variables into a CSV file, delimited by the ',' sign.
- output: out := CSV File
- exception: None

saveHDeficiency(s):

- Given a sequence of HealthDeficiency s, for each HealthDeficiency object, write all of the object's variables into a CSV file, delimited by the ',' sign.
- output: out := CSV File
- exception: None

2.8 Database Module

Static Module

Database is a class used to store all NursingHome and NHGeoInfo objects. NursingHome objects are stored in a Hash Symbol Table, and NHGeoInfo are kept inside a List object.

Uses

NursingHome, NHGeoInfo, FireSafetyDeficiency, HealthDeficiency, DataReader

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
Database			FailedInitialize
ChangeLocation	*String*		
search	seq of $\mathbb{Z}$ and $\mathbb{B}$	seq of NursingHome	
accessors		*RelevantType*	

Semantics

State Variables

$stNH$:= *SequentialChainingHashST* of NursingHome
$nhGeo$:= $seqof$ NHGeoInfo $lat := \mathbb{Z}$ $lon := \mathbb{Z}$

Access Routine Semantics

Database():

- transition: Initializes the Database by reading through the relevant CSV files with DataReader. Construct sequence of NursingHome, NHGeoInfo, FireSafetyDeficiency and HealthSafetyDeficiency objects. Insert NursingHome objects into $stNH$. For each Deficiency object found, insert into the NursingHome in $stNH$ with the same ID. Insert NHGeoInfo objects into $nhGeo$, then do updateDistances() and sort($nhGeo$)

- exception: Any errors (I.e. IOException, memory, etc.) $\rightarrow$ FailedInitialize

ChangeLocation(s):

- transition: lat, lon :=geoCode(s), and call updateDistances()
- exception: None

search(s):

- output: out := seq of NursingHome, with each NursingHome fulfilling the search criteria in s
- exception: None

accessors():

- refers to all relevant accessor methods referring to any the lat and lon variable inside Database (i.e. getLat(), getLon(), etc.)

Local Functions

updateDistances():
updateDistances() $\equiv (\forall g : nhGeo | g.\text{calculateDistance}(lat, lon))$
geocode(s): $String \to \mathbb{Z}, \mathbb{Z}$ geocode(s): $\equiv$ Return the latitude and longitude of the given address s
sort(seq): $seqof$ NHGeoInfo sort(seq): $\equiv$ Sort the sequence seq from lowest distance to highest distance with QuickSort

2.9 Menu Module

Description

The module in which interactions between the client and the application is carried out. Provides visual feedback towards the client. Currently designed to be a basic command line program that will print out results based on the user's commands. Possible future implementations include providing UI in the forms of interactable menus with a connection to GoogleMaps API.

Uses

Database

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
Menu			FailedInitialize
readLine	*String*		UnexpectedCommand
display		*String*	

Semantics

State Variables

command := *String*
db :=Database
sLoc := *String* of user's location
S := *seq of* NursingHome
state := *enum state to show correct menu*

Access Routine Semantics

Menu():

- transition: Initialize Database. Initialize *state* to main menu screen.
- exception: Failed initialize → FailedInitialize

readLine(s):

- transition: Read line s from CLI value. $state := someotherstate$
  Switch state of interface based on input (I.e. Main Menu → Show results). Appropriate commands can extract information from database when needed.

- exception: Unhandled command given → UnexpectedCommand

display():

- transition: Given *state* and S, display visual feedback to client. (I.e. main menu, selection options, list of nursing homes).

- exception: None

Local Functions

initializeMenu() - Ideally, a visual UI with buttons and search bar instead of a CLI is created.
connectToMapAPI() - Start connection to GoogleAPI maps.
displayMap(*seq of* NH) - display google map with NH addresses and your location to GoogleMaps API.

3 Non-Functional Requirements

3.1 Reliability

LiveLong should currently only fail at providing services if it fails to find the CSV file to pull information from. This can result because of a moved file, misnamed file, or a corrupted CSV file. Additionally, the dataset needs to be downloaded from the official government site at Medicare to be updated. As such, an option to directly link to the dataset online should be looked into to provide reliable data. Alternatively, the application can write the stored data in the Database to a internal csv to better store data and conserve space, while also promising that there is always a dataset that the application can initialize with.

3.2 Accuracy of Results

Each search result should return results such that all nursing home that fulfills the criteria is included (up to a certain number, ranked based on distance or rating, decision pending), and that there are no nursing homes that violate the criteria is shown. Additionally, any warnings or deficiencies must be clear for the client to view (i.e. abuse warnings, health deficiencies).

Results should be constant each time, such that a repeated search after another initialization of the application will yield the same results.

3.3 Performance

As *LiveLong* is advertised as a quick application to return results for a client, it is important that the efficiency of the search algorithm can quickly find accurate results pertaining to the user location and the search filter. Additionally, the initialization of the database should be relative quick, no more than a minute in the worst case. This includes the reading of the CSV files, the parsing and creation of appropriate data types, and the insertion into the database. The *get*() command is used multiple times when appending deficiencies to a Nursing Home, which can drastically increase the initialization times.

3.4 Human-Computer Interface Issues

Currently the interface uses a Command-Line-Interface, which is not accessible for the general public, nor is it easy to use. Ideally, the application should make the interface straightforward to use.

There are plans to eventually implement an interactable menu and use a visual map API to show the results. An interactable menu would making setting a search filter far more easy for a client to use, rather than manually typing out the filter in the command line.

3.5 Scalability

As this program is based off of a growing dataset that is being updated monthly, it is vital to ensure that the application can accept new changes or additions in the dataset, as well as being able to still run efficiently. Steps must be taken to ensure that new information can be added fluidly into the application (I.e. overwriting the previous dataset file, changed or reordered column names, etc.).

Additionally, the initialization of the Database and the subsequent search and filter methods should be able to run their commands in a very short amount of time. Ideas such as generating a compressed local CSV file containing only necessary information could speed up initialization times, and applications of more proven search and sort algorithms (I.e. QuickSort for sorting, or hash tables for Storing and Searching quickly) will ensure that even with the growth of the dataset, this application will still run fluidly.

3.6 Other Constraints or Issues

LiveLong should be able to function on multiple devices and operating systems with little change from each platform. As this application is built through Java, the only requirement for the platform is the ability to run Java applications. Thus, there should not be any platform-specific algorithms that would prevent it from running on another.

4 Requirements on Development and Maintenance Process

LiveLong has several important goals it must fulfill when it is used, ranked in order of importance.

1. Accuracy: Results must always return the same results when given the same criteria. Information presented must be accurate based on the dataset provided.

2. Speed & Efficiency: *LiveLong* must be able to run quickly despite the large amount of information it needs to parse during initialization. It must also be able to quickly return results to the client.

3. Usability: *LiveLong* should be simple and straightforward for the client to use. The search filter should not be too confusing to set - this implies the implementation of a interactable UI.

4.1 Quality Control Procedures

LiveLong will undergo several tests with each new implementation or version change.

1. Unit Testing - Each module will have its methods tested based on differing values of input, and whether the results are consistent with the expected behaviour. This will be ideally run with JUnit to test each module independently and to ensure that the smallest components of the application are implemented correctly.

2. Integration Testing - Related modules will be tested together to ensure that the modules are integrated correctly. For example, ensuring the DataReader module correctly creates sequences of NursingHome ADT when parsing, and ensuring that the module correctly read the csv. The sequence can later be given to Database to test for insertion and searching algorithms.

3. System Testing - The entirety of the application undergoes BlackBox testing methods to ensure that the application functions such that it meets the requirements, and that the application delivers the results it requires in an efficient and accurate manner.

Further tests will be done by non-programmers to gain feedback from expected stakeholders, such as feedback of the UI flow and usability, and any concerns or ideas that were not considered by the developers.

4.2 Priority of the Required Functions

1. DataReader: To accurately parse and construct the appropriate objects from a CSV file in a quick and efficient manner

2. DataWriter: To correctly write an object's variables into a CSV file such that the same object can be constructed with said values

3. NursingHome: ADT to store information about a Nursing Home. Immutable, created only from DataReader.

4. NHGeoInfo: ADT to store geographical information about a Nursing Home. Immutable, created only from DataReader.

5. Deficiency, FireSafetyDeficiency, and HealthDeficiency: ADT to store information about deficiencies assigned to a NursingHome from the official government.

6. Database: To correctly store homes in a sorted manner. To correctly search for and return a sequence nursing home fulfilling a search criteria.

7. Menu: To visually display the application in a way such that the UI delivers information to the client effectively, via maps and an interactable interface. Should be easily understandable for the client to start using.

4.3 Likely Changes in the Future

1. Changes to how data is gathered, from a local CSV file to a direct connection to the Medicare site online

2. Saving a 'copy' of the database as a backup CSV file in the project files to ensure that there is a dataset present in initialization, and to reduce load times

3. Changes to the Menu UI and how it should be constructed based on user feedback for UX.

4. Implementing a connection to the GoogleMaps API to provide results in a clearer way

5. Implementation on a different platform (I.e. Windows $\rightarrow$ iPhone)

4.4 Other Possible Requirements

Development of *LiveLong* should follow proper versioning cycles to be able to follow the application's current progress rate, and whether any changes should be made to follow the deadline.

Tests should be done in such a manner that all exceptions, edge cases, and deliberate fails (i.e. scenarios of sending 'DROP_TABLE', or abusing string concatenation, invalid text in the CSV file, corrupted files) is tested. Additionally, a final acceptance test phase should be done by programmers and non-programmers that do not include the developers themselves. This is to get their objective feedback on the current state of the application, and to catch any unlikely scenarios or interactions that the developers did not account for.