COMP90015: Distributed Systems – Assignment 1

Report by: David Naughton – 320479, dna@student.unimelb.edu.au

1 Introduction

In this project we have been asked to demonstrate our understanding of multi-threaded client server communication and concurrency by building a dictionary application. This application has many requirements detailed in the specification. In brief, it requires multiple clients to be able to read/write to a dictionary managed by a single multi-threaded server and perform the following operations,

1. Query the meaning(s) of a word
2. Add a new word and corresponding meaning(s)
3. Delete an existing word and its meaning(s)
4. Update the meaning(s) for an existing word

The interaction with the client should be through a Graphical User Interface and all errors should be properly managed and communicated. The lowest level of abstraction for client server communication and concurrency is sockets and threads.

2 Components – Server Side

2.1 - Server

The Server class has several key responsibilities.

1. Parses the command line arguments which are the port number for the server to listen on, and the path to the csv which contains the initial dictionary data.
2. Creates the dictionary data structure object which is a Trie (section 2.3) as a static object accessible at package level. This allows all the handler objects to access the same dictionary.
3. Creates a ServerGui (section 2.2) object which provides information about the clients connected to the server.
4. Listens for incoming client connections on the given port number, once a connection is established it creates a new socket and stream objects and passes them to a new Handler object which manages the communication to that client. Then it continues listening for the next client.
5. Finally, once it receives an instruction from the ServerGui to close the server socket and end the server script it first saves the dictionary data structure object (in memory) to disk at the same path as given initially.

The Server also handles a few errors.

1. It checks the correct number of command line arguments are provided. If not, then it sends an error message to the terminal "`Incorrect number of command line arguments`" and exits the program.
2. It checks the port number provided is an integer between 1024 and 65353. If not, then it sends an error message to the terminal "`Invalid port number, must be integer between 1024 and 65353`" and exits the program (n.b. path to csv errors not handled by the Server but by the Trie class).
3. Any IOExceptions that occur while creating/closing server socket, client socket or data streams are picked up and an error message is sent to the terminal "`Server multi-threading error`".

2.2 - ServerGui

This class is a basic graphical user interface for the server. The class extends JFrame and is built using the java Swing library. It performs a few basic functions,

1. It displays the number of active client connections to the server. The number refreshes automatically every second.
2. It has a button labelled "Kill Server" which tells the server to save the dictionary data, end its service and then it closes the server GUI window.

Having this function is useful because it allows you to save the dictionary data before the server closes. However, if the user uses 'CTRL+C' in the terminal to close the server then the dictionary data will not be saved.

2.3 - Trie

A Trie is a search tree data structure that is used to store strings. Strings are input character by character into the tree with each character stored in a single node and the whole string forming a path from the root to the end of the word. The worst-case time complexity of search, insert and delete are all O(n) where n is the length of the word which is very fast. However, the reason I chose to use a Trie is because it allows you to perform word completions very quickly, you first find the last character in the incomplete string and then perform depth first search to find all completions in the rooted subtree. Doing completions in a hash table or other data structure would have been much slower. I have linked the wiki page with more details about Tries in the references [1].

Since the java standard library has no Trie implementation I had to implement my own from scratch. One of the key constraints that you need to set is a limit on what characters you allow in a string, the space consumption of the Trie is extremely sensitive to this limit as each node will store an array of nodes of that size. For this reason, I chose to limit words to the simple lowercase alphabet (a-z), however this does not apply to meaning(s).

Multiple meanings are all stored in the same string. This makes a number of operations easier than storing each meaning in a separate string such as; displaying, creating & editing meanings in single GUI field, storing meanings in a single field in csv and passing them between client and server.

As multiple threads (thread per client) are active at the same time. If multiple threads try to write to the Trie at the same time, errors can occur. To manage this, I have used the 'ReentrantReadWriteLock' (explained in analysis section below). This allows you to create separate read and write locks. The methods which use the write lock are insert and delete. The methods which use the read locks are search, meaning and completions.

The Trie performs the following basic functions,

1. Insert word used for adding words and updating words (updating is equivalent to replacing the meaning(s) of a word)
2. Delete word used for deleting a word
3. Finds all the possible completions of an incomplete word
4. Save a csv to a given path. This encodes commas and line breaks since otherwise they would need to be parsed with an external library or a more complicated parser would need to be written. The basic encoding converts these two characters to ascii characters 254 and 255 respectively which cannot be written directly from the keyboard and are not commonly used.
5. Reads a csv passed as a path to the function. This decodes commas and line breaks from the ascii characters 254 and 255 respectively.

The Trie handles the following errors,

1. File not found exception when trying to read csv from given path, sends error message "`Dictionary load failed, file not found`" to the terminal.
2. IOException when trying to read csv from given path, sends error message "`Error reading dictionary into Trie`" to the terminal.
3. File not found exception when trying to save csv to given path, sends error message "`Dictionary save failed, file not found`" to the terminal.

2.4 - Handler

When the server accepts a new client connection it hands the socket and streams to a new handler instance which is executed as a new Thread. The Handler class implements the runnable class which is an alternative to inheriting from the Thread class. The only function we need to override for runnable interface is the run() function. The server calls start() function which runs the run() function inside the Handler straight away.

The rules I have developed for client server communication are as follows,

1. Handler object waits for client request.
2. Client request is sent as a comma separated String e.g. "a,apple,a fruit that you eat"
3. The command comes before the first comma and can be either 'a' for add, 'd' for delete, 'u' for update, 'w' for word completions, 'm' for meaning(s) and 'close' to close the handler.
4. If the String has at least one comma then the second field is the word to process and
5. If the String has at least two commas then anything after the second comma is the meaning(s). N.b. words cannot contain commas (this is enforced by the client GUI).

The Handler performs the following basic functions,

1. It interprets the request according to the rules above.
2. It parses the String to separate the different elements and then calls the relevant function from Trie class e.g. t.delete(word)
3. It sends a reply message to the Client (except for 'close'). Generally, this is a Boolean which tells the Client if the operation was successful or not.
4. In the case of completions, the Handler needs to send back a list of words to the client. It converts the list to a String and then sends via ObjectOutputStream.

The Handler handles the following errors,

1. IOException when reading from the Input Stream, sends "`Handler read error`" to the terminal.
2. IOException when replying to Output Stream, sends "`Error responding to client request`" to the terminal.

<u>**3 Components – Client Side**</u>

3.1 - Client

The Client class have the following responsibilities,

1. Parses the command line arguments which are the server port number and host name to connect to.
2. Connects to the server and create input and output streams to communicate through.

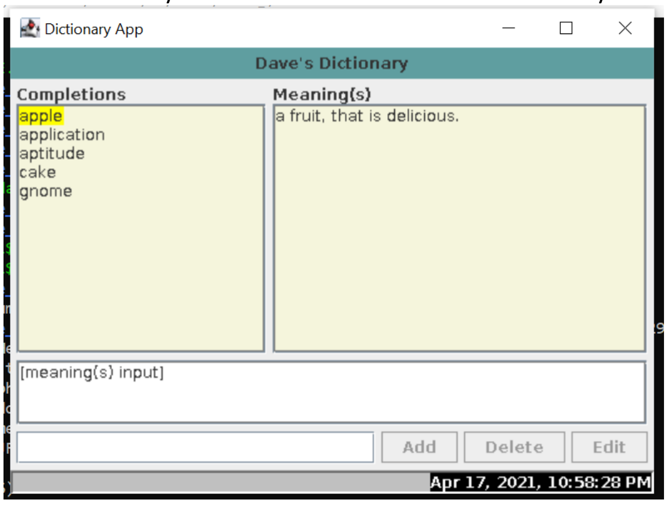
3. Creates the GUI which serves as the client interface for the application.
4. Has functions ‘add’, ‘delete’, ‘update’, ‘completeWords’ and ‘getMeanings’ which send requests to the Handler using the rules defined in (2.4). These receive input from the user through the GUI.
5. Closes the client connection and streams and tells the Handler to close when instructed through the GUI.

The Client handles the following errors,

1. It checks the correct number of command line arguments are provided. If not, then it sends an error message to the terminal “`Incorrect number of command line arguments`” and exits the program.
2. It checks the port number provided is an integer between 1024 and 65353. If not, then it sends an error message to the terminal “`Invalid port number, must be integer between 1024 and 65353`” and exits the program.
3. If the client cannot connect to the server at the given address and port then a ‘ConnectException’ will be thrown and an error message “`Error connecting to server at the address and port provided`” will be sent to the terminal.
4. If the client cannot communicate with the server (read or write) then an IOException will be thrown and an error message will be sent to the terminal “`Lost connection to server`”
5. If the client is trying to read the ObjectInputStream and cast the input to a String but a ‘ClassNotFound’ error is thrown then an error message “`Error reading word completions, class not found`” is sent to the terminal.
6. If the client attempts to close connection but the output message fails then an error message “`Error trying to close client connection`” is sent to the terminal.
7. If the client attempts to close the input and output streams and an IOException occurs then an error message “`Error trying to close streams`” is sent to the terminal. If the streams don’t exist then the null pointer exception is caught but no action required as streams already closed.
8. The client receives Boolean responses to its requests for add, update and delete. If the responses are false then error messages are sent to the GUI to let the client know the operation failed. For e.g. if the client requests to delete a word and the handler return false, then the GUI status bar will print “`failed - word not found`”
9. For the other requests getMeaning() and completeWords() which return Strings, if the Strings return are empty then again error messages are sent to the GUI to inform the client. E.g. if the client tries to get completions for a word and the handler sends back and empty string then the GUI status bar prints “`failed - word not found`”.

3.2 - GUI

The GUI class extends JFrame and is built using the java Swing library. The interface is designed to be user friendly and provide an efficient way for users to interact with the dictionary.



Initially the two input boxes contain text wrapped in ‘[ ]’ to let the user know what input they expect. Once the user clicks into them the initial text disappears and they can begin typing.

The dictionary is implemented using query by default. As you type into the input box after each keystroke completions will appear in the completions text area. The completions will be ordered alphabetically and the first one will be highlighted in yellow. The highlighted word will be queried automatically, and its meaning(s) will appear in the Meaning(s) text area.

The ‘Add’, ‘Delete’ and ‘Edit’ buttons perform the operations as expected however they will be disabled unless an operation appears safe. For e.g. if you typed in ‘cake’ the ‘Add’ button would be greyed out it wouldn’t be possible to add a word that already exists. The ‘Edit’ button however would not be greyed out and to update the meaning(s) you would need to type the meaning in the larger input box above and then hit the button.

The word input box prevents validates the user input as you type and any characters outside the range [a-z] are not allowed.

The status bar is at the bottom of the GUI and provides lots of different messages to the user describing the success or failure of various events. Success messages appear in blue text and failure messages in red. It also contains the date and time which may be useful for the user to track their time spent on the app.

To exit the application the user can click on the 'x' in the top right corner of the window.

The GUI handles the following errors,

1. Sends error message from Client object to the status bar.
2. Tries to find the position of the completion to highlight, throws bad location exception if fails and sends error message "`highlighting error, could not locate`" to the status bar.
3. The user is not prevented from updating or adding a word when the meaning input box has not been initialised or is the empty string.
4. The user is prevented from pressing any button when the word string is empty, deleting a word which does not exist or adding a word which does. This is done by disabling buttons. However, this is based on the current list of completions which is only updated when the user makes a change to the word input box. The state of the words may change in the dictionary and yet the action will be prevented until the completions update.

3.3 - DocFilter

The DocFilter class inherits from the DocumentFilter class. This code has been adapted from [2]. It allows the validation of text as the user types. It handles the following error,

1. Prevents the user from typing words which contain characters outside the range [a-z]. If they try nothing will appear in the input box and the message "`Invalid input, please use [a-z]`" will appear on the status bar.

4 Analysis and Conclusion

Below I have analysed the key design decisions in my application,

1. I chose the thread per connection method because it is my first experience with multi-threading. I did not feel like I was qualified to leap ahead to the challenge of worker pool implementation. The thread per connection makes more sense to me than thread per request as it takes a few seconds to create a thread and if a client wanted to make many requests in quick succession then there would be considerable time waiting for threads to close and reopen. The thread per connection means a client only has to wait for the initial thread to start and then they can submit as many requests as they like without delay. The disadvantage of thread per connection is that with many clients connected to the server there can be a very large number of threads running on the server and this might cause issues. Also, the worker pool architecture initialises threads on server start up and so when a client connects it is a bit faster because the thread they use is already initialised.
2. I use a Trie for the dictionary data structure. This was critical in being able to perform fast operations; insert, delete and edit all run in O(n) time where n is the length of the word. Since all clients read and write on the same data structure the operations need to be synchronised. If the operations are fast then this works a lot better because operations spend less time in the queue. The completions operation is more time intensive that the others however is very optimised on the Trie data structure and runs super quick relative to other data structures. The only disadvantage of using the Trie is that it uses a lot of space as each node in the tree holds an array of nodes the length of the allowed character set. I mitigated this by using a limited character set [a-z].
3. I create my own message protocol for communication between client and handler. The client sends a comma delimited string to the handler. The first field tells the handler which request to process and from there it knows what the following fields represent e.g. word and then meaning(s). Since this application managed a small number of different requests it was possible to build a quite simple protocol. It was easy to implement and efficient. However, if the application was larger and required a lot more commands it might require a more formal message protocol as it would get hard to manage the complexity with the current approach.
4. One of the key things that separates my GUI is the completions. I noticed that if the dictionary starts off with few words and the user has no visibility over what is in it, then it is hard for them to query, delete or update words. So, I added a text area that shows them completions, and since the word input box starts off empty the completions are the whole dictionary. Closely related to this, if the app shows word completions and the user wants to query a word, then the word the user wants to query will naturally be somewhere in the list of completions. Therefore, it makes sense to query by default so that the user may find the definition as soon as possible. This creates a nice user experience. The disadvantage of this however is the cost in how much resources it takes to get completions every time the word input box changes. This works well for a small dictionary (< about 10k words) however for a dictionary larger than this it would be quite slow.
5. Since many threads may be running on the server at the same time and each thread has access to the same Trie data structure, I needed to synchronise the threads. I chose to utilise the 'ReentrantReadWriteLock'. The advantage of this over 'synchronised' methods or blocks is that it more flexible, we can choose where to lock and unlock within or across methods. Also, the synchronised block does not support fairness, any thread can acquire the lock once released. Locks

can implement fairness to make sure the longest waiting thread gets access to the lock. There are separate locks for reading and writing, multiple threads can acquire the read lock if no thread has acquired the write lock or has requested for it. The write lock can be acquired if no threads are reading or writing. The re-entrant property allows threads to enter the lock on a resource more than once. This structure for managing synchronisation of threads is robust and allows reliable and fast read write access to the data structure. A possible disadvantage is that a malicious client could try to continually perform write operations and hold the lock which would deny other threads the ability to read or write.

6. TCP was used for transport level communication. This was my first experience writing client server code so I took the advice from lectures which was to use TCP. This protocol comes with a lot of advantages over UDP such as flow control, connection-oriented communication and reliability. The disadvantage is that a connectionless service is faster because there are no re-transmission delays.

Excellence

The following are the excellence components of my project (implementation and report) which are described and discussed throughout the report.

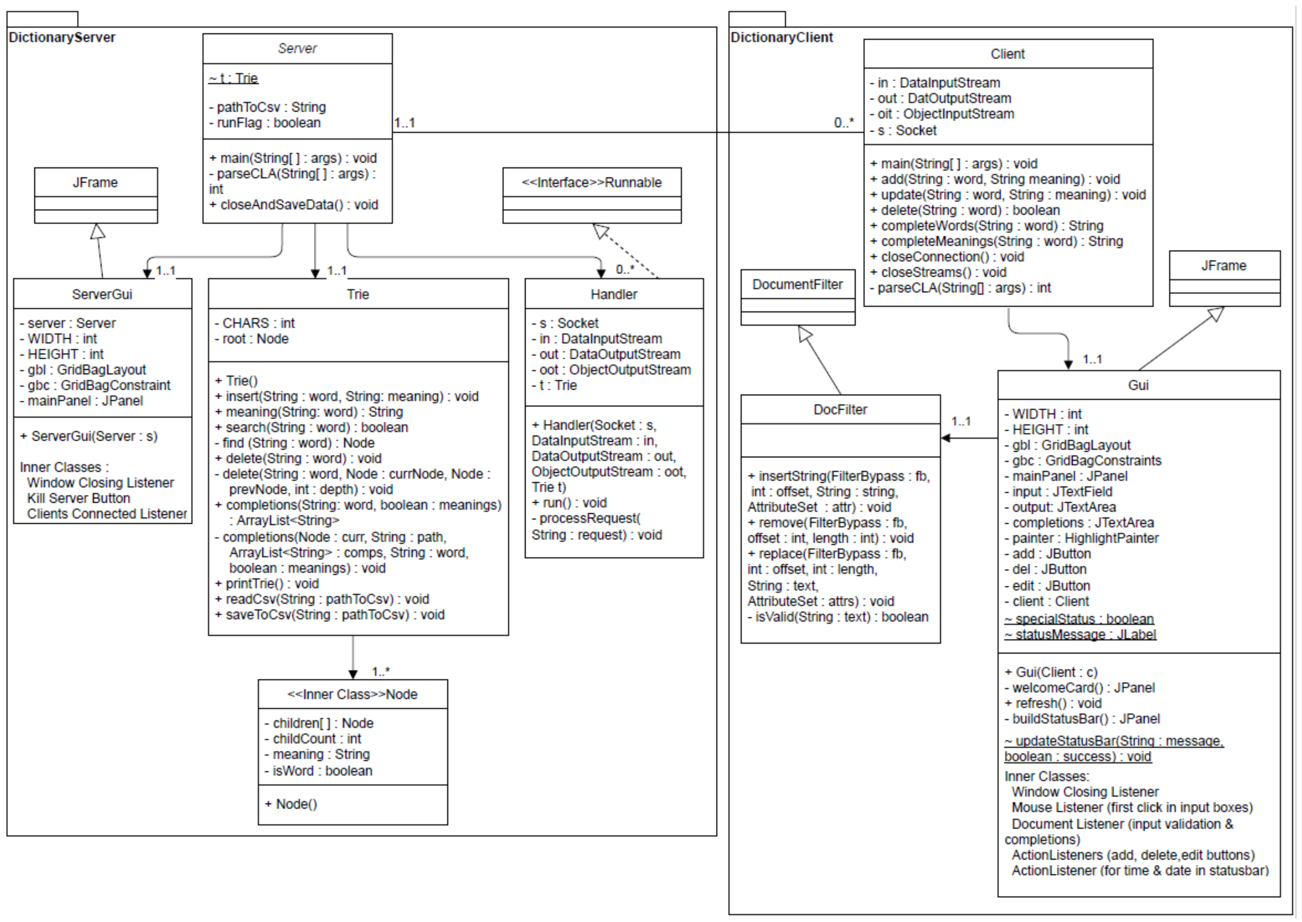
1. GUI prevents several errors by automatically enabling / disabling add, delete, edit buttons.
2. GUI has a status bar which provides success and failure messages to the user.
3. Trie data structure used to store the dictionary, implemented from scratch. It provides fast search, insert, delete and completions.
4. Report contains image of the GUI and detailed UML and Sequence diagrams
5. Report contains in depth analysis above (section 4) which discussed the advantages and disadvantages of key design decisions.

Creativity

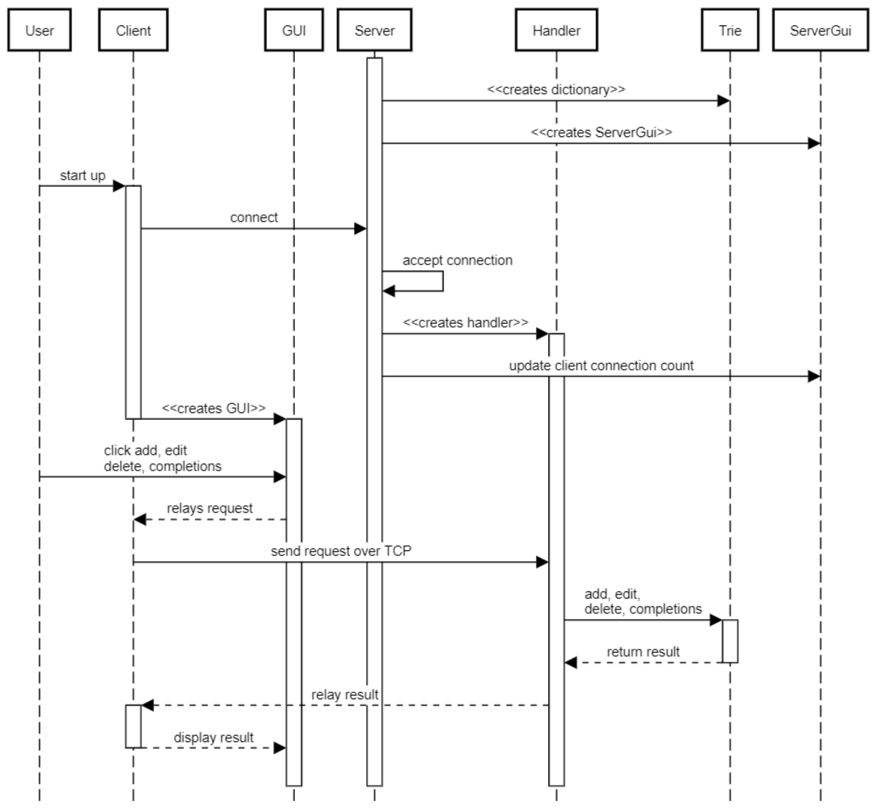
The following are the creativity components of my project which are described and discussed throughout the report.

1. Server GUI which tracks the number of clients connected to the server. Also used to kill the Server and save the changes to the dictionary to disk.
2. GUI performs word completions based on the characters entered in the word input box.
3. GUI automatically queries by default based on the first word in the completions list.
4. Designed my own message protocol between client and server which is simple and robust.

In conclusion, this report and the associated application has demonstrated a solution to the task which not only satisfies the functional requirements but goes beyond in both creativity and excellence. Please see the next page for UML class and sequence diagrams.


DictionaryServer
Server
~ t : Trie
- pathToCsv : String
- runFlag : boolean
+ main(String[ ] : args) : void
- parseCLA(String[ ] : args) : int
+ closeAndSaveData() : void
1..1
0..*
JFrame
<<Interface>>Runnable
1..1
1..1
0..*
ServerGui
- server : Server
- WIDTH : int
- HEIGHT : int
- gbl : GridBagLayout
- gbc : GridBagConstraint
- mainPanel : JPanel
+ ServerGui(Server : s)
Inner Classes :
Window Closing Listener
Kill Server Button
Clients Connected Listener
Trie
- CHARS : int
- root : Node
+ Trie()
+ insert(String : word, String: meaning) : void
+ meaning(String: word) : String
+ search(String : word) : boolean
- find (String : word) : Node
+ delete(String : word) : void
- delete(String : word, Node : currNode, Node : prevNode, int : depth) : void
+ completions(String: word, boolean : meanings) : ArrayList<String>
- completions(Node : curr, String : path, ArrayList<String> : comps, String : word, boolean : meanings) : void
+ printTrie() : void
+ readCsv(String : pathToCsv) : void
+ saveToCsv(String : pathToCsv) : void
Handler
- s : Socket
- in : DataInputStream
- out : DataOutputStream
- oot : ObjectOutputStream
- t : Trie
+ Handler(Socket : s, DataInputStream : in, DataOutputStream : out, ObjectOutputStream : oot, Trie t)
+ run() : void
- processRequest(String : request) : void
1..*
<<Inner Class>>Node
- children[ ] : Node
- childCount : int
- meaning : String
- isWord : boolean
+ Node()
DictionaryClient
Client
- in : DataInputStream
- out : DatOutputStream
- oit : ObjectInputStream
- s : Socket
+ main(String[ ] : args) : void
+ add(String : word, String meaning) : void
+ update(String : word, String : meaning) : void
+ delete(String : word) : boolean
+ completeWords(String : word) : String
+ completeMeanings(String : word) : String
+ closeConnection() : void
+ closeStreams() : void
- parseCLA(String[] : args) : int
DocumentFilter
JFrame
1..1
DocFilter
+ insertString(FilterBypass : fb, int : offset, String : string, AttributeSet : attr) : void
+ remove(FilterBypass : fb, offset : int, length : int) : void
+ replace(FilterBypass : fb, int : offset, int : length, String : text, AttributeSet : attrs) : void
- isValid(String : text) : boolean
1..1
Gui
- WIDTH : int
- HEIGHT : int
- gbl : GridBagLayout
- gbc : GridBagConstraints
- mainPanel : JPanel
- input : JTextField
- output: JTextArea
- completions : JTextArea
- painter : HighlightPainter
- add : JButton
- del : JButton
- edit : JButton
- client : Client
~ specialStatus : boolean
~ statusMessage : JLabel
+ Gui(Client : c)
- welcomeCard() : JPanel
+ refresh() : void
- buildStatusBar() : JPanel
~ updateStatusBar(String : message, boolean : success) : void
Inner Classes:
Window Closing Listener
Mouse Listener (first click in input boxes)
Document Listener (input validation & completions)
ActionListeners (add, delete,edit buttons)
ActionListener (for time & date in statusbar)

High Level Dictionary App Sequence Diagram



References [1] https://en.wikipedia.org/wiki/Trie [2] https://stackoverflow.com/questions/60300755/how-to-remove-content-of-jtextfield-while-documentlistener-is-running