Yuan Xinran, Stanley - Project Portfolio

1. Introduction

This document serves to highlight my contributions to the project, DeliveryMANS. DeliveryMANS is a command line interface (CLI), desktop application catered towards food delivery administrators to assist them in all their delivery needs. It includes features such as autocomplete, the ability to add, list and delete orders and deliverymen, as well as statistics generation for administrators to manage deliveries effectively. This six week project was done by a group of 5 year 2 NUS School of Computing (SoC) Computer Science students as part of our Software Engineering 2103T module requirements to morph Address Book 3 into a greenfield project with CLI constraints.

Below are the symbols and formatting used in this document.

Symbols

NOTE	Requirements or important things you should take note of.
TIP	Tips to assist you.

Text formatting

`undo`	Commands or user input which can be entered into the application for the User Guide. Components, classes or objects used in the architecture of the application for the Developer Guide.

2. Summary of contributions

This section serves to summarize my contributions to the project, namely feature enhancements, code as well as other contributions.

Autocomplete feature - Major

- What it does: It autocompletes the commands entered by the user.
- Justification: This feature allows the application to be more user-friendly as the administrator would not need to refer to the User Guide or memorize input commands to navigate the application. This enables effective and efficient management of deliveries.
- Highlight: This feature works with user commands with more than 1 input field (see "3.1 Adding an order" below) and is designed for the ease of integration of with future commands. The implementation was also challenging as it required results to be displayed even before the user input was entered using the `Enter` key.

Order Manager feature - Major

- What it does: It allows the user to add, edit, delete, assign, complete and list orders.
- Justification: This feature allows for more effective management of deliveries as delivery administrators may be required to do last-minute edits of orders before delivery, such as when the customer cancels an order or adds more food to be delivered.
- Highlight: The implementation was challenging as it required changes to existing commands as well as in-depth analysis of the `Logic`, `Model`, `Storage` and `Ui` components of the architecture to allow for the ease of integration with future commands and for the application to run smoothly.

Context switching feature - Minor

- What it does: It allows for user targeted commands for different contexts such as deliveryman, customer and restaurant.
- Justification: This feature allows for a greater variety of commands for the administrator to manage different aspects of a delivery, while reducing the confusion caused due to similar command names such as `add` for both restaurant and deliveryman.
- Highlight: The implementation allows for near instantaneous switching of contexts when entered.

Code contributed

- Implementation of context switching (Pull requests #1 #9 #44 #50)
- Implementation of Order Manager (Pull requests #49 #50 #51 #59 #180 #181 #184)
- Implementation of autocomplete feature (Pull requests #98 #111 #188 #200 #205)
- Over 3000 lines of code on RepoSense

Other contributions

- Documentation

    - Updated User Guide with texts and images for explaining universal commands (Pull requests #81 #83 #193 #232)
    - Updated Developer Guide with UML diagrams and texts for explaining implementation of features (Pull requests #77 #85 #87 #217 #234)
- Community
    - Reported bugs and suggestions for other teams (Examples #1 #2 #3)
    - Reviewed PRs with non-trivial review comments (Pull requests #54 #74)
- Project management
    - Managed releases `v1.2` - `v1.4` (3 releases) on GitHub

3. Contributions to the User Guide

Given below are some of my contributions to the User Guide. They showcase my ability to write documentation targeting end-users.

Start of extract

3.1. Adding an order: `-add_order`

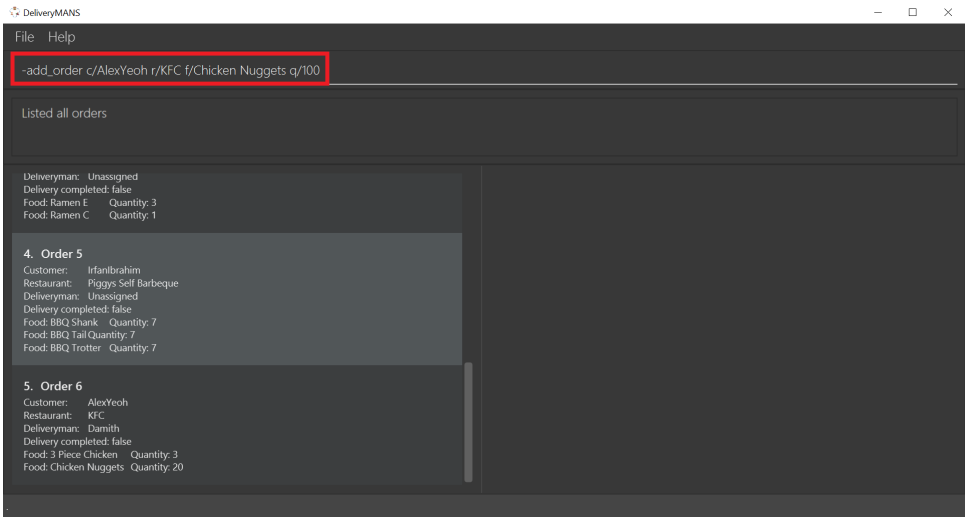
This command allows you to add a new order to the database to be processed. The deliveryman to deliver the order will be allocated automatically based on the internal algorithms.

Format: `-add_order c/CUSTOMER r/RESTAURANT f/FOOD… q/QUANTITY…`

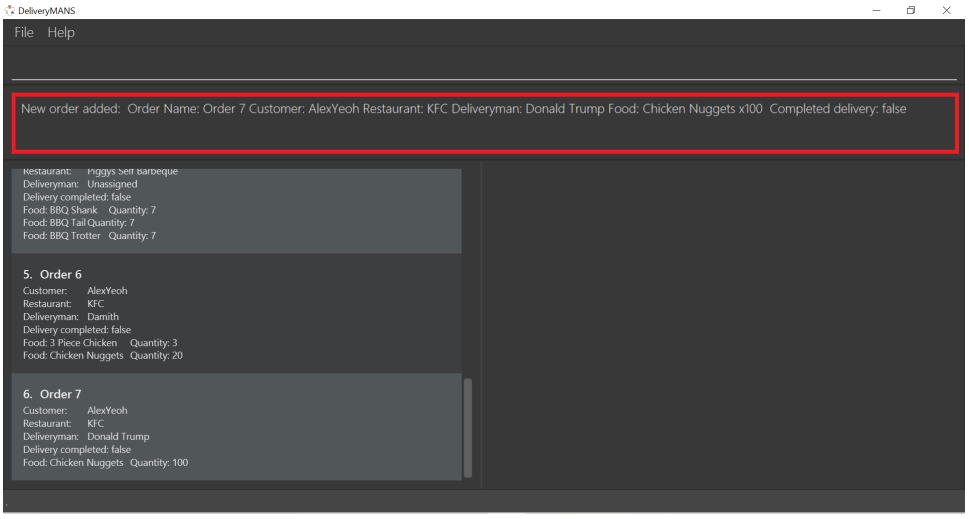
Example: `-add_order c/AlexYeoh r/KFC f/Chicken Nuggets q/100`

Example use case

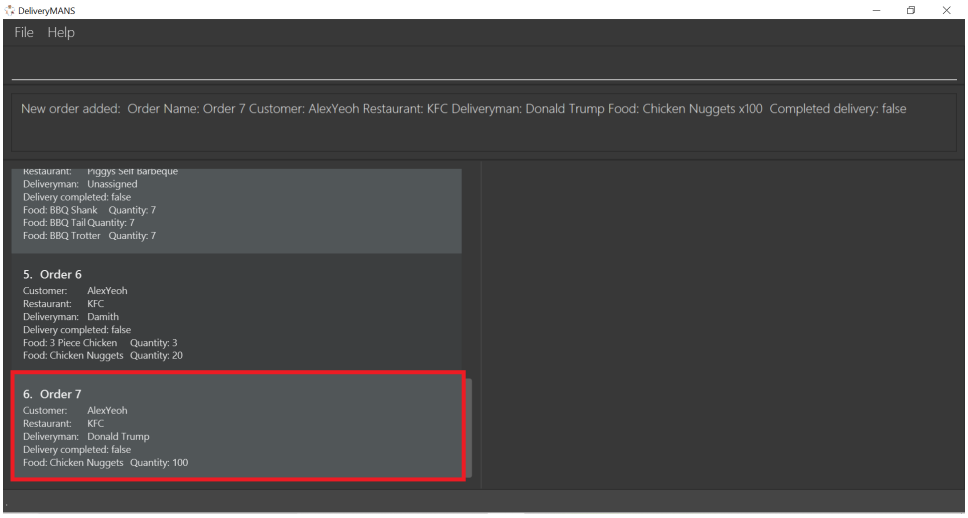
1. Type the command statement from the example above into the program and press **Enter** to execute it.



2. If you are successful, the result box displays the message: `"New order added: Order Name: Order 7 Customer: AlexYeoh Restaurant: KFC Deliveryman: Donald Trump Food: Chicken Nuggets x100 Completed delivery: false".`



3. The order list shows the newly added order.

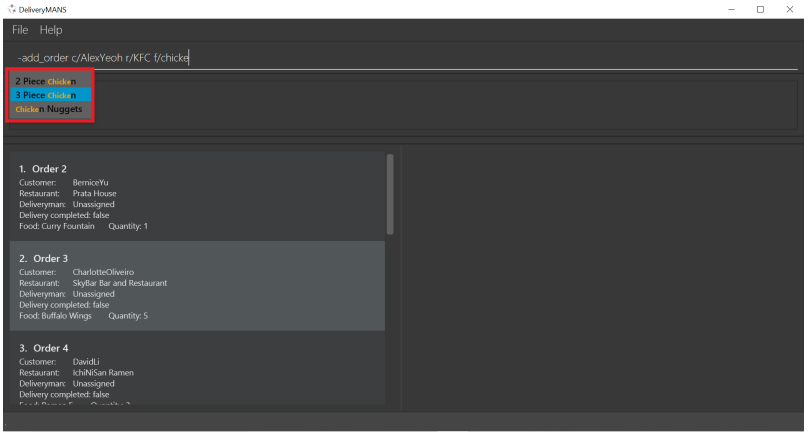


NOTE

- A valid customer `c/CUSTOMER`, restaurant `r/RESTAURANT` and restaurant menu item `f/FOOD` must be provided and exists currently in the database.
- The quantity of food `q/QUANTITY` to be delivered must be provided and be greater than 0.

TIP

- Fill in the restaurant `r/RESTAURANT` before entering the restaurant menu item `f/FOOD` for the autocompletion feature to load the list of that restaurant's menu in a drop down box for you.



3.2. Editing an order: `-edit_order`

This command enables you to edit an order. The order to edit will have to be specified by its order name when you are entering the command.

You can change:

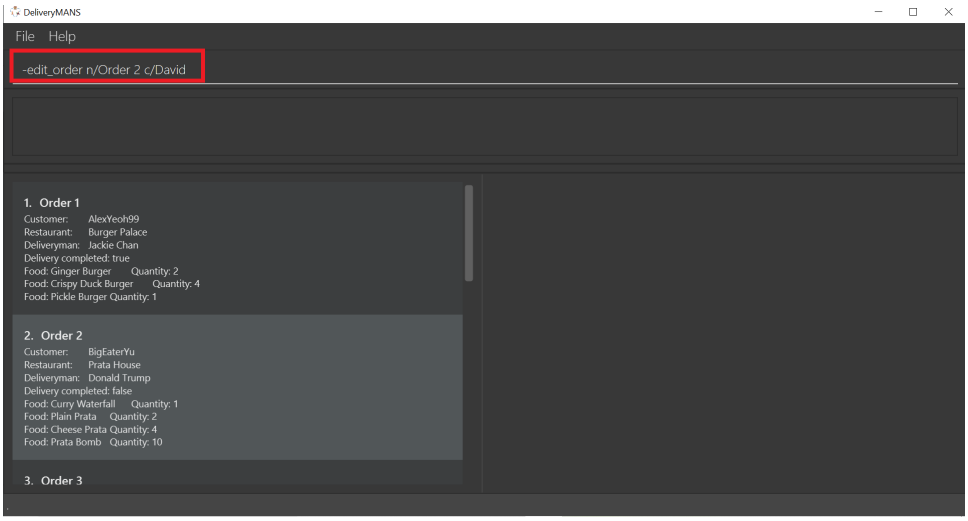
- The customer `c/CUSTOMER` who made the order
- The restaurant `r/RESTAURANT` which the order was made from
- The food `f/FOOD` ordered as well as its quantity `q/QUANTITY`

Format: `-edit_order n/ORDERNAME [c/CUSTOMER] [r/RESTAURANT] [f/FOOD]… [q/QUANTITY]…`

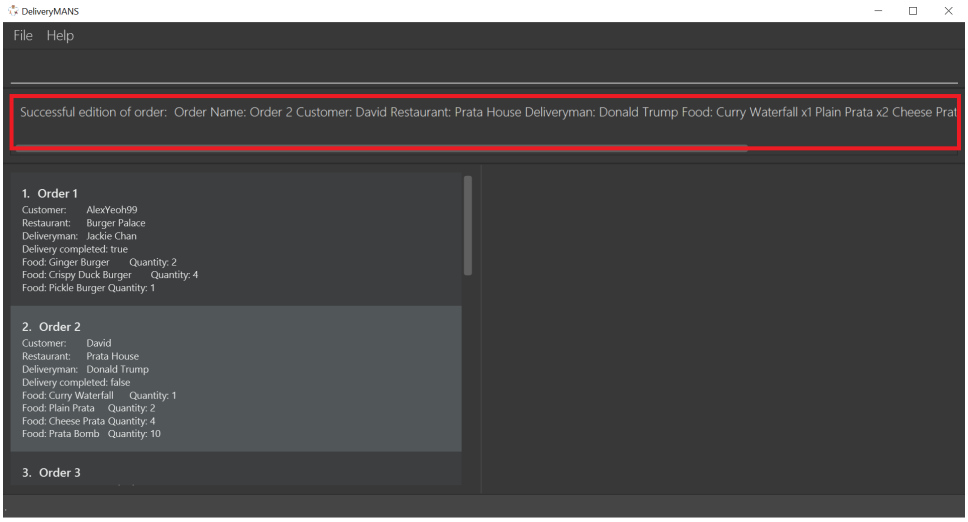
Example: `-edit_order n/Order 2 c/David`

Example use case

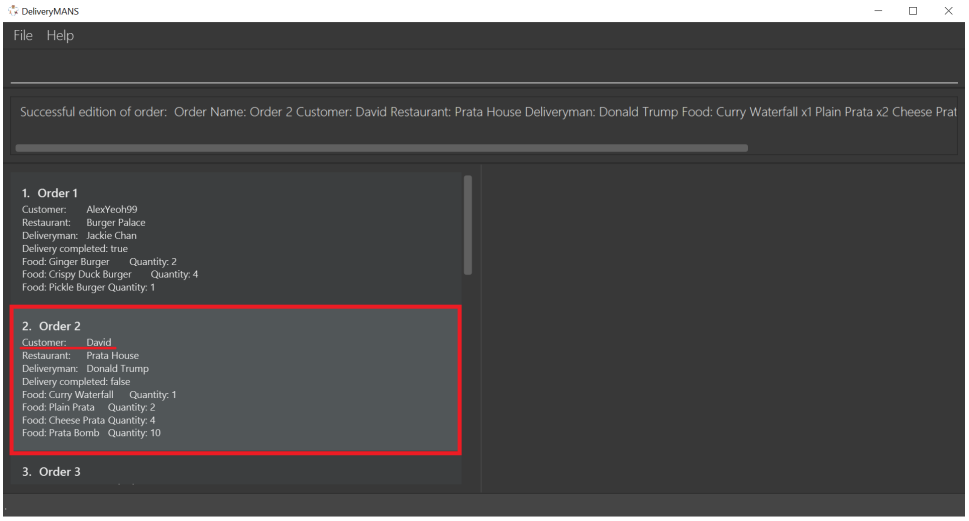
1. Type the command statement from the example above into the program and press **Enter** to execute it.



2. If you are successful, the result box displays the message: `"Successful edition of order: Order Name: Order 2 Customer: David Restaurant: Prata House Deliveryman: Donald Trump Food: Curry Waterfall x1 Plain Prata x2 Cheese Prata x4 Prata Bomb x10 Completed delivery: false"`.



3. The order list shows the updated order.

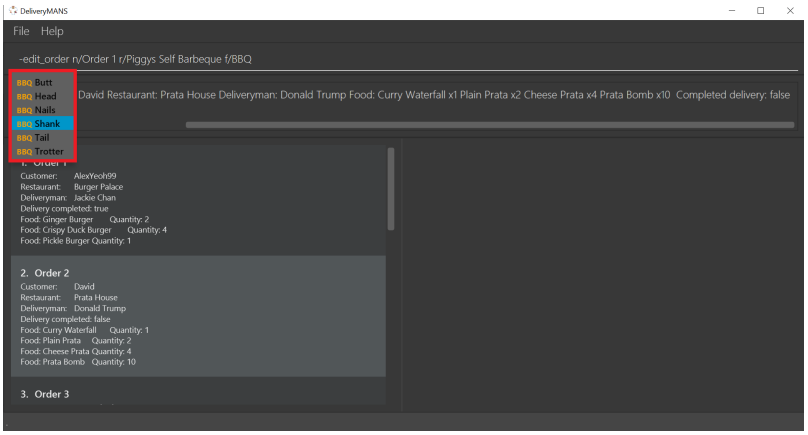


NOTE

- The order name `n/ORDERNAME` must exist in the order list.
- A customer `c/CUSTOMER`, restaurant `r/RESTAURANT` or restaurant menu item `f/FOOD` provided must be valid and exists currently in the database.
- Optional items with '[]' tags may be ommitted e.g. `[r/RESTAURANT]`. However at least 1 tag has to be present for the order to be edited.

TIP

- Fill in the restaurant `r/RESTAURANT` before entering the restaurant menu item `f/FOOD` for the autocompletion feature to load the list of that restaurant's menu in a drop down box for you.



End of extract

My other contributions to the User Guide include: switching contexts, assigning, completing, deleting and listing of orders.

4. Contributions to the Developer Guide

Given below are my contributions to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Start of extract

4.1. Autocomplete commands feature

This is a feature which allows you to view all available commands matching the input keyword or letters, eliminating the need to memorize the commands or leave a browser tab open with the User Guide of this application.

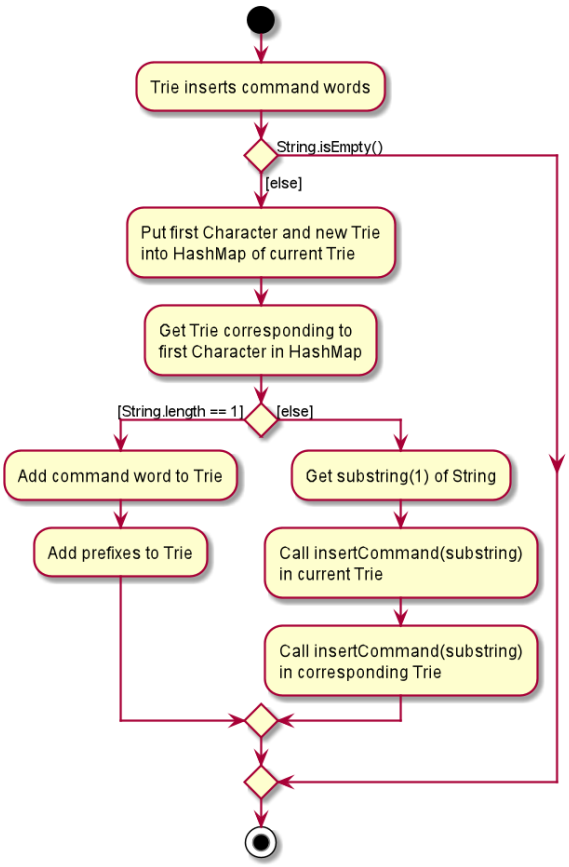
4.1.1. Implementation

The autocomplete mechanism is facilitated by the `KeyListener` and a `Trie`, a tree-like abstract data type (ADT). The `KeyListener` passes the current input text in the input command box to the `TrieManager` via `LogicManager#getAutoCompleteResults()`. The `TrieManager` calls `Trie#autoCompleteCommandWord()` and a sorted list of matching commands is passed back to the `CommandBox` and is displayed on the `Ui` via a dropdown box below the user input command box.

The underlying data structure used is a directed graph with the `Trie` as a node and `HashMap<Character, Trie>` to represent all outgoing edges. The keys in the `HashMap` are `Characters` in the command words while the values are the `Tries` containing the subsequent `Characters` in the command words. Each `Trie` contains a `List<String>` of command words, which is returned when `Trie#autoCompleteCommandWord()` is called.

Given below is an example usage scenario and how the autocomplete mechanism behaves at each step.

Step 1: You launch the application. The `TrieManager` initializes the respective `Tries` with their context-specific command words using `Trie#insertCommand()`. The `Trie` adds each `Character` of the input `String` and new `Tries` into the `HashMap<Character, Trie>`, as well as the command word into the `List<String>`, recursively as illustrated by the activity diagram below.



Step 2: You want to add an order to the database, however are uncertain how to spell the command and type in `order`. The `KeyListener` passes the `String` in the `CommandBox` to the `Trie` via the `LogicManager` and `TrieManager`. The trie searches for relevant commands and pass them as a list back to the `CommandBox` via `Trie#getAutoCompleteCommandWord()`, `Trie#search()` and `Trie#getAllCommandWords()`. The `Ui` displays the relevant results in a dropdown box below the user input command box.

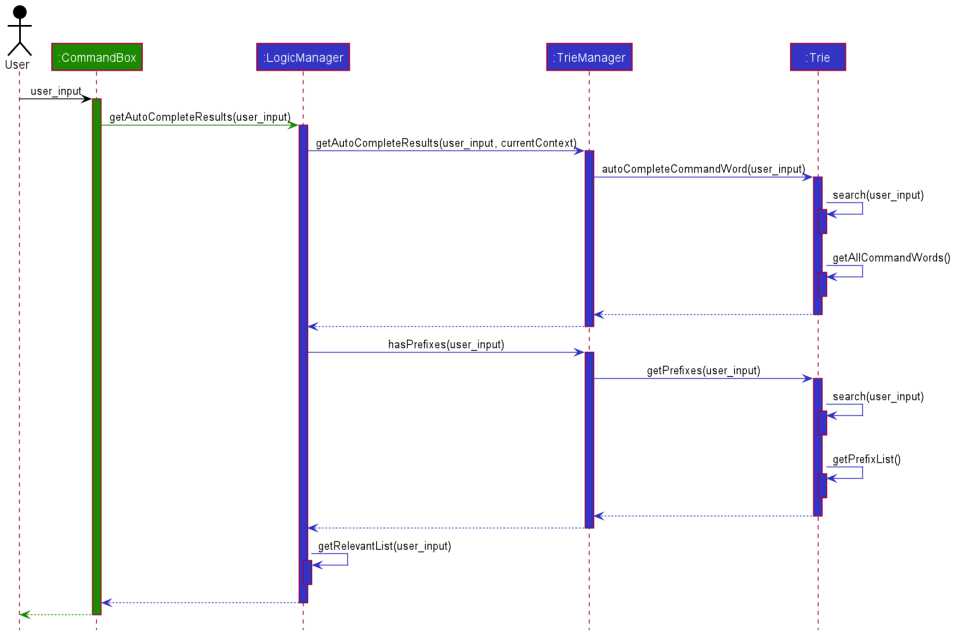
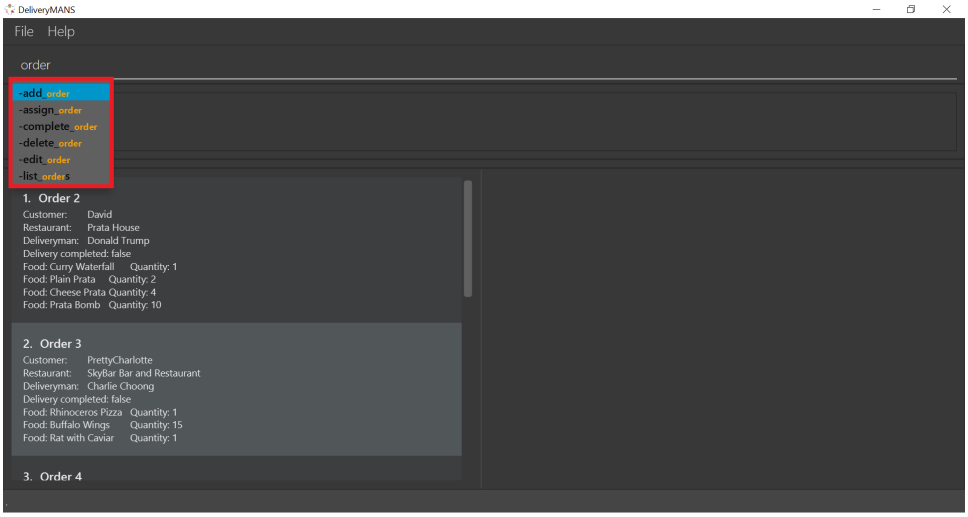


Figure 1. Sequence diagram illustrating the handling of user input via autocomplete

Step 3: You can now complete the command you want by entering the relevant command shown in the dropdown box.



4.1.2. Design Considerations

Below are a few design considerations of the autocomplete commands feature.

Aspect: How autocomplete executes

- **Alternative 1 (current choice):** Use a `KeyListener` to record and handle user inputs in the user input command box before they are entered.

    - Pros: Aesthetically pleasing, allows for on-the-fly display of results.
    - Cons: Laborious to implement, especially in terms of debugging and troubleshooting. It may also break Object-Oriented Programming (OOP) principles if not implemented properly.
- **Alternative 2:** Handle user input only when the command is entered, utilizing the `Parser` to handle user inputs and pass it to the `Trie` to be evaluated.
    - Pros: Adheres to current flow of command executions, will not break any OOP principles.
    - Cons: Tedious for the user, as the user will have to retype the whole command again. Furthermore, it does not look aesthetically pleasing.

Alternative 1 was selected, as it is more user friendly, and leaves a better impression onto users compared to alternative 2.

Aspect: Data structure to support the autocomplete commands feature

- **Alternative 1 (current choice):** Use a `Trie` to store `Characters` of commands as keys.
    - Pros: Efficient and rapid searching, retrieving and displaying of results due to the tree-like ADT.
    - Cons: Tedious to implement, as `Tries` are not currently implemented in Java.
- **Alternative 2:** Use a list to store all current commands.
    - Pros: Easy to implement as lists are already available in Java.
    - Cons: Inefficient and slow searching, because of the need to iterate through the entire list of commands while calling `.substring()` and `.contains()` methods.

Alternative 1 was selected, as it allows for faster searching and listing of relevant commands compared to alternative 2.

End of extract

My other contributions to the Developer Guide include: Order Manager, class and sequence diagrams.