Yuan Xinran, Stanley - Project Portfolio

1. Introduction

This document serves to highlight my contributions to the project, DeliveryMANS.

intro to team, project, reqs (desktop app CLI), list main features 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 requirements.

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.
`Logic`	Components, classes or objects used in the architecture of the application.

2. Summary of contributions

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

2.1. Feature....

- What it does:
- Justification:
- Highlight:

2.2. Code contributed

- Implementation of context switching for user targeted commands (Pull requests #1 #9 #44 #50)
- Implementation of Order Manager

- Implementation of autocomplete feature (Pull requests #98 #111 #188 #200 #205)

2.2.1. 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
- 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`

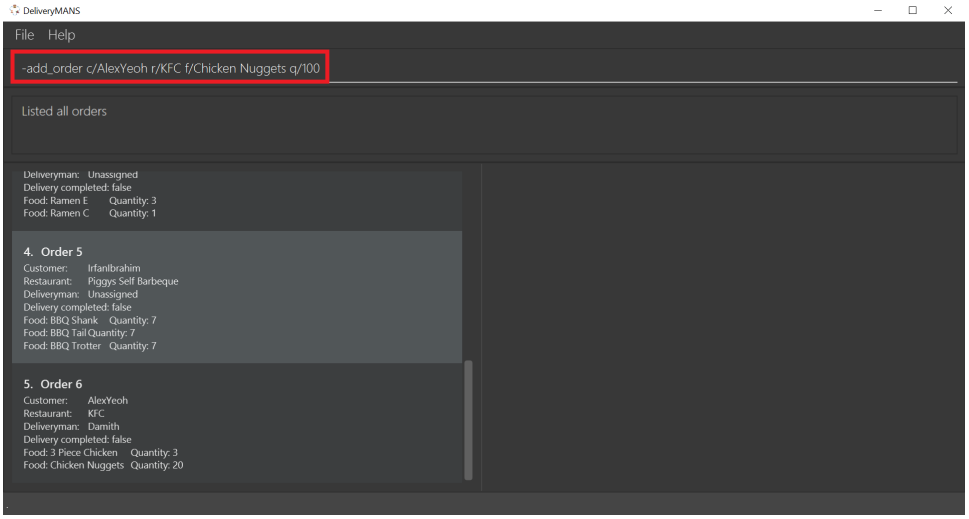
This command allows you to add a new order to the database to be processed. The deliveryman to deliver the order will be allocated automatically based on the internal algorithms.

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

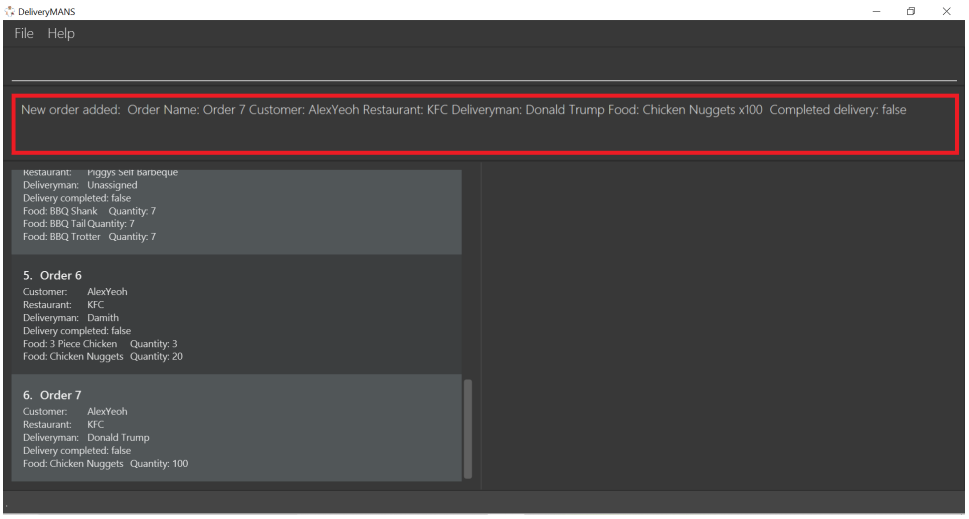
Example: `-add_order c/AlexYeoh r/KFC f/Chicken Nuggets q/100`

Example use case

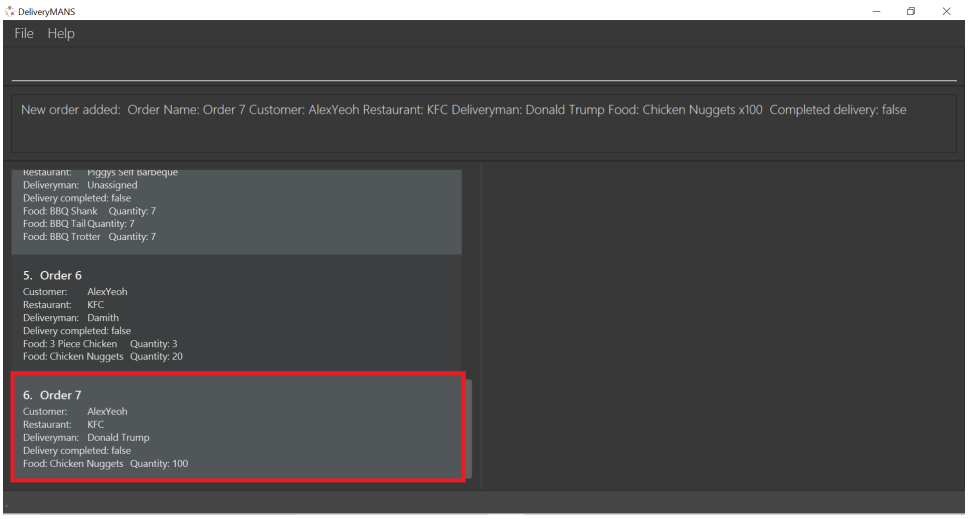
1. Type the command statement from the example above into the program and press **Enter** to execute it.



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.

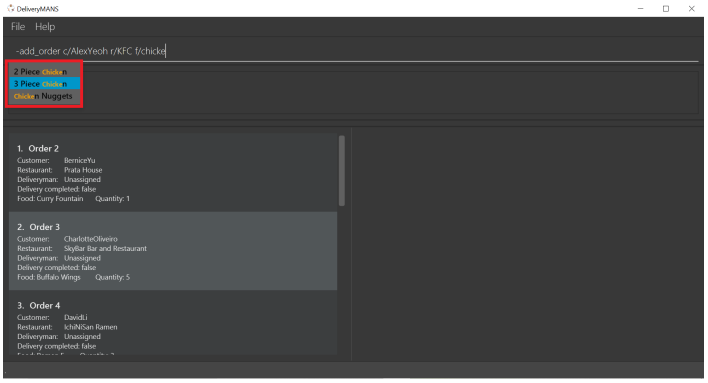


NOTE

- 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:

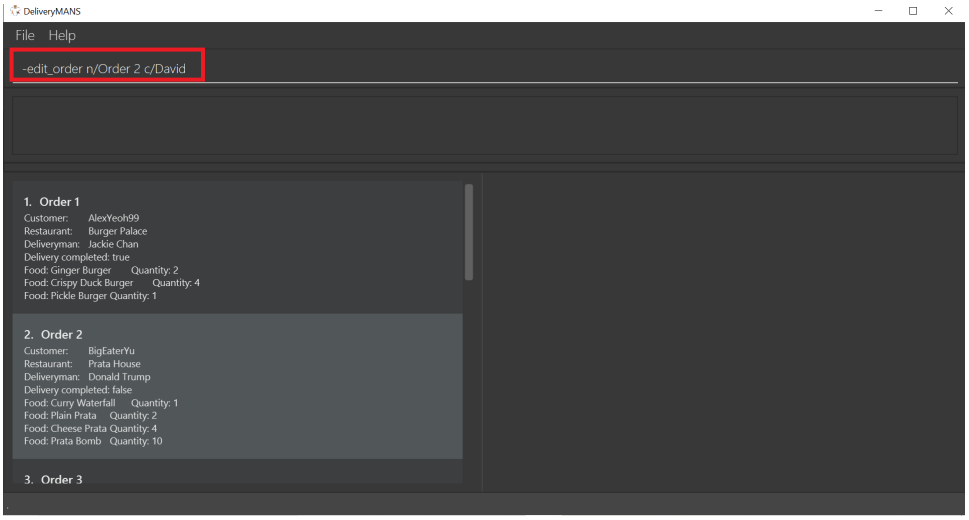
- The customer `c/CUSTOMER` who made the order
- The restaurant `r/RESTAURANT` which the order was made from
- The food `f/FOOD` ordered as well as its quantity `q/QUANTITY`

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

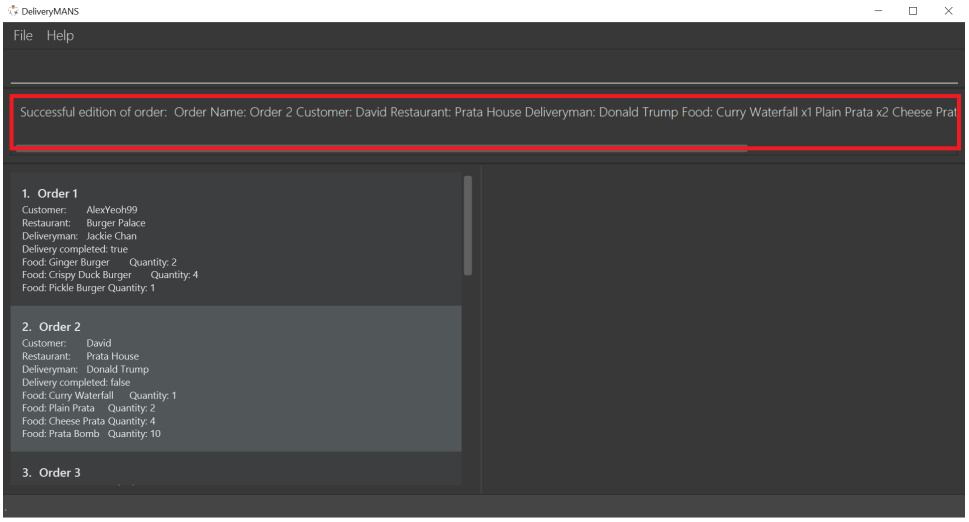
Example: `-edit_order n/Order 2 c/David`

Example use case

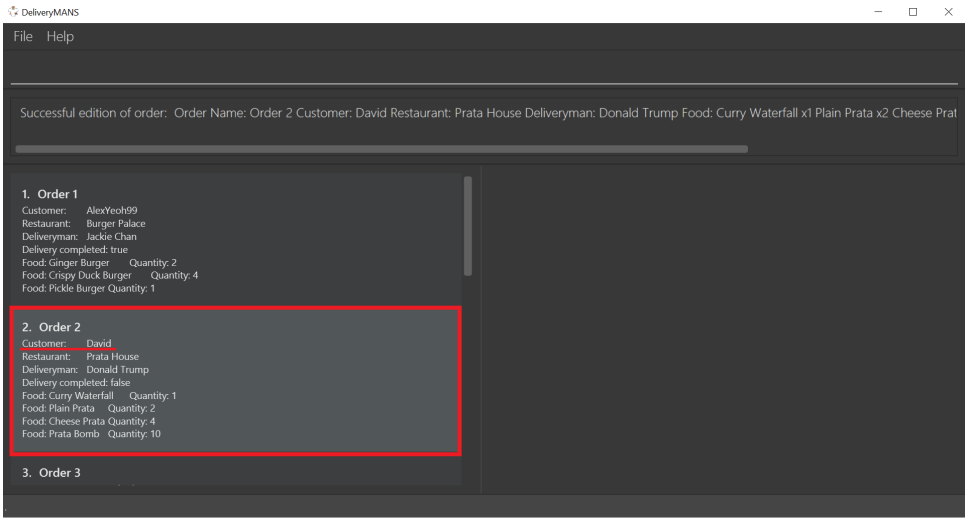
1. Type the command statement from the example above into the program and press **Enter** to execute it.



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.

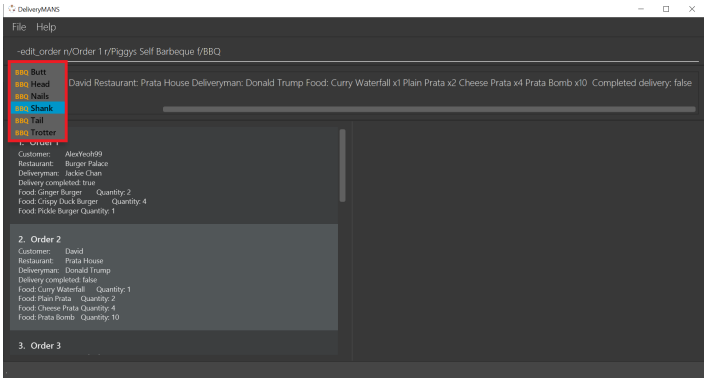


NOTE

- 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

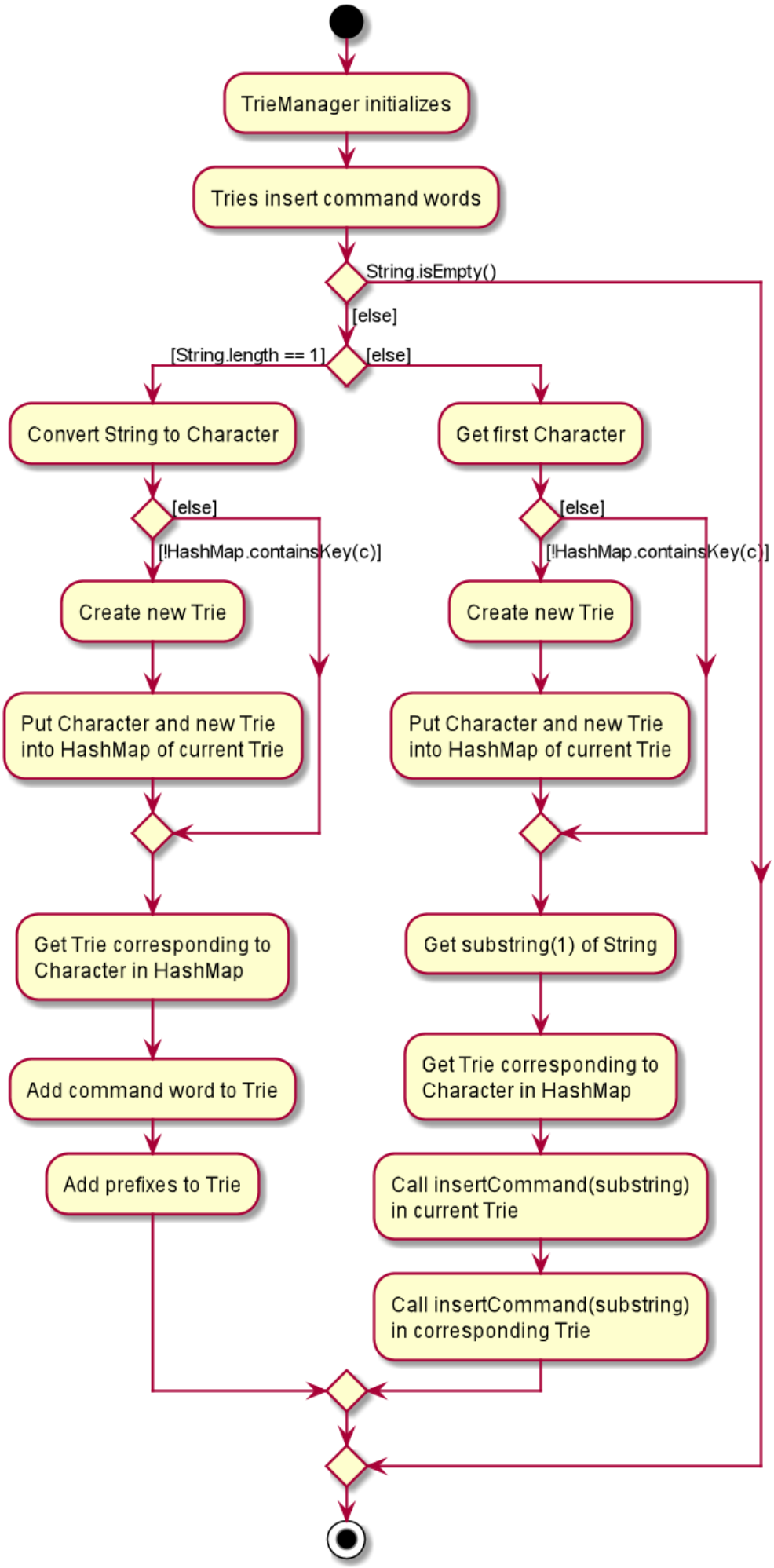
The autocomplete mechanism is facilitated by the `KeyListener` and a `Trie`, a tree-like abstract data type (ADT). The `KeyListener` passes the current input text in the input command box to the `TrieManager` via `LogicManager#getAutoCompleteResults()`. The `TrieManager` calls `Trie#autoCompleteCommandWord()` 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.


TrieManager initializes
Tries insert command words
String.isEmpty()
[else]
[String.length == 1]
[else]
Convert String to Character
Get first Character
[else]
[!HashMap.containsKey(c)]
[else]
[!HashMap.containsKey(c)]
Create new Trie
Create new Trie
Put Character and new Trie into HashMap of current Trie
Put Character and new Trie into HashMap of current Trie
Get Trie corresponding to Character in HashMap
Get substring(1) of String
Add command word to Trie
Get Trie corresponding to Character in HashMap
Add prefixes to Trie
Call insertCommand(substring) in current Trie
Call insertCommand(substring) in corresponding Trie

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.

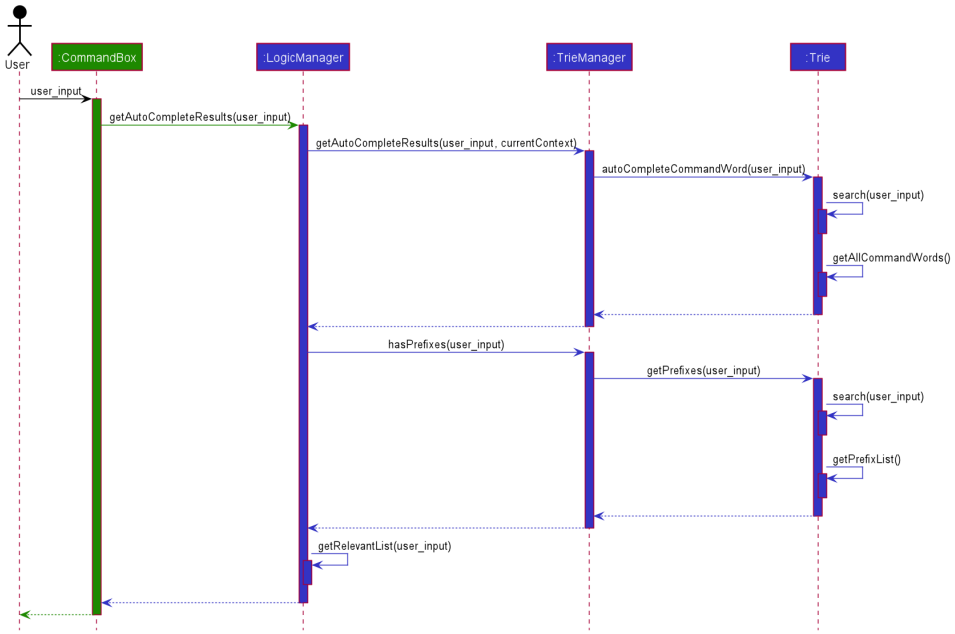
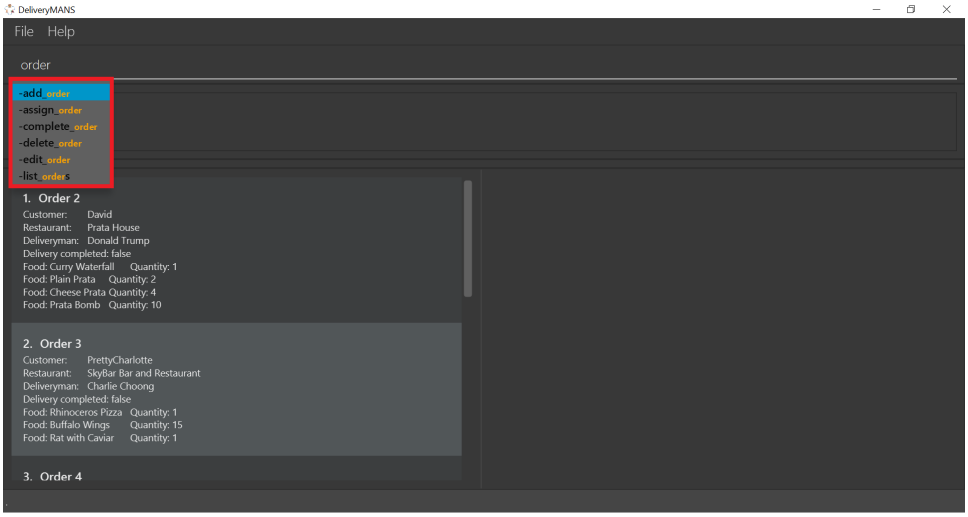


Figure 1. Sequence diagram illustrating the handling of user input via autocomplete

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.

4.2. Order Manager

Order Manager is an address book of Orders and has some useful functions specifically catered towards the ease of management of orders.

Firstly, the automated allocation of deliveryman once new orders are added or completed. When a new order is created on the database, or when an existing order is completed, a deliveryman will be assigned to deliver the new/existing pending orders based on whether he/she is present as well as whether he/she is currently preoccupied with delivering another order. This helps to ease the burden on the user as they would not need to manually allocate deliverymen to the orders. However, the feature to manually allocate is still present if the user wishes to do so.

Secondly, the Order Manager allows for sorting of orders, based on date, customer, restaurant, menu or even deliveryman, depending on what information the user wishes to see to allow for better management.

Additionally it implements the following operations:

- `-add_order` - adds an order to the database.

- `-assign_order` - assigns an available deliveryman to an existing order in the database.
- `-complete_order` - updates the completion status of an existing order in the database.
- `-delete_order` - removes an existing order in the database.
- `-edit_order` - edits an existing order in the database.
- `-list_orders` - lists all existing orders in the database.

These operations are exposed in the `ModelManager` class as `ModelManager#addOrder()`, `ModelManager#getOrder(Name targetOrder)`, `ModelManager#setOrder(Order target, Order editedOrder)`, `ModelManager#deleteOrder(Order order)` and `ModelManager#assignUnassignedOrder()`.

Order manager implements its own `Model`, `Command` and `Parser` for the 'Logic Component', `JsonOrderDatabaseStorage`, `JsonSerializableOrderDatabase`, `JsonAdaptedOrder` and `JsonAdaptedFoodOrder`, along with methods in the `StorageManager` for the `Storage Component` and lastly, `OrderCard` and `OrderListPanel` for displaying on the `Ui Component`.

insert object diagrams here + brief explanation of object diagrams

4.2.1. Implementation

Add command: `-add_order`

The add command adds an order to the `ModelManager` and `UniqueOrderList`. The `UniversalParser` invokes `AddOrderCommandParser#parse()`, which parses the target **customer**, **restaurant**, **food** and **quantity** from a **String** into **Name** and **Integer** objects.

Only valid **customer**, **restaurant**, **food** and **quantity** are allowed. This validation is done through accessing `UniqueCustomerList` and `UniqueRestaurantList` through `ModelManager#getFilteredCustomerList()`, `ModelManager#getFilteredRestaurantList()` and calling their respective `isValidName()` methods. **Food** validity will be checked through retrieving the respective using `Restaurant#getMenu()` and `Menu#isValidName()`.

Duplicated `Order` will be checked for using `ModelManager#hasOrder()` and is then added to the `UniqueOrderList` via `ModelManager#addOrder()`.

Delete command: `-delete_order`

The delete command deletes an `Order` from the `ModelManager` and `UniqueOrderList` by a specified index. The `UniversalParser` invokes `DeleteOrderCommandParser#parse()` and user input is used to get the index of the `Order` to be deleted.

4.2.2. Design Considerations

Below are a few design considerations of the Order manager class.

Aspect: Data structure for modelling, storage and utilization of `Order`.

- **Alternative 1 (current choice):** Make use of existing data structures as references to create new data structures needed for the implementation of an Order Manager.

    - Pros: Straightforward to implement.
    - Cons: Tedious to implement as several regions of the codebase needs to be edited for `Order` to run, display and save successfully.
- **Alternative 2:** Implement data structures from scratch.
    - Pros: Pride and accomplishment of implementing data structures from scratch.
    - Cons: Tedious and time wasting to code the necessary classes.

Alternative 1 was selected, as it is much faster to implement compared to alternative 2, given the short time spam of 6 weeks to complete the project.

End of extract