Zhou Tianyu - Project Portfolio

PROJECT: DeliveryMANS

Overview

Summary of contributions

- **Major enhancement**: Restaurant Manager
  - What it does: Manages all the restaurants in the entire database, including their details, food menu and orders.
  - Justification: Restaurants, together with Customers and Deliverymen, are one of the 3 main stakeholders of a food delivery application. This restaurant manager allows for efficient managing of the huge number of restaurants in the database through the following highlights.
  - Highlights:
    - EditMode for Restaurant. Involves multiple components. UI is changed upon entering od EditMode. Every change made to restaurant (changing of name), changing of menu) would have to update the relevant information in the relevant orders.
    - Auto-tagging of food items as "Popular". Often an important information customers would want to know. Auto-tagging saves administrators the workload of manually tagging each popular food in the huge food database. Statistic from previous completed orders, such as number of orders of each food item, would have to be stored. Upon completion of every new order, statistics are updated, percentage of food item is calculated and tagged as "popular" if requirement is met.
- **Code contributed**: [View on RepoSense]
- **Other contributions**:
  - Project management:
    - Managed releases `v1.1` - `v1.4` (4 releases) on GitHub
    - Managed milestones `v1.2` - `v1.4` (3 releases) on GitHub
  - Enhancements to existing features:
    - Wrote tests for Restaurant-side `Storage` and `Logic` (Pull requests #227)
  - Documentation:
    - Added documentation on Restaurant-side commands in User Guide (Refer to contributions below)
    - Added documentation on EditMode feature for restaurants in Developer Guide (Refer to contributions below)

- Community:
  - Reported bugs and suggestions for other teams in the class (Reported bugs in PED)

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Restaurant commands

Commands in the restaurant context

Adding a restaurant: `add`

This command adds a restaurant to the restaurant database.

Format: `add n/NAME l/LOCATION [t/TAG]`

- `LOCATION` can only be one of the following locations: Jurong, Tuas, Woodlands, Bishan, City, Marina, Changi, Punggol.

Example: `add n/KFC l/Jurong t/FastFood`

Deleting a restaurant: `delete`

This command deletes the restaurant at the specified index in the restaurant list from the restaurant database.

Format: `delete INDEX`

- `INDEX` must be a positive integer from 1 to n, the number of restaurants in the restaurant list.

Example: `delete 1`

NOTE	*Note:* • Deleting a restaurant will delete all its existing orders from the order database as well.

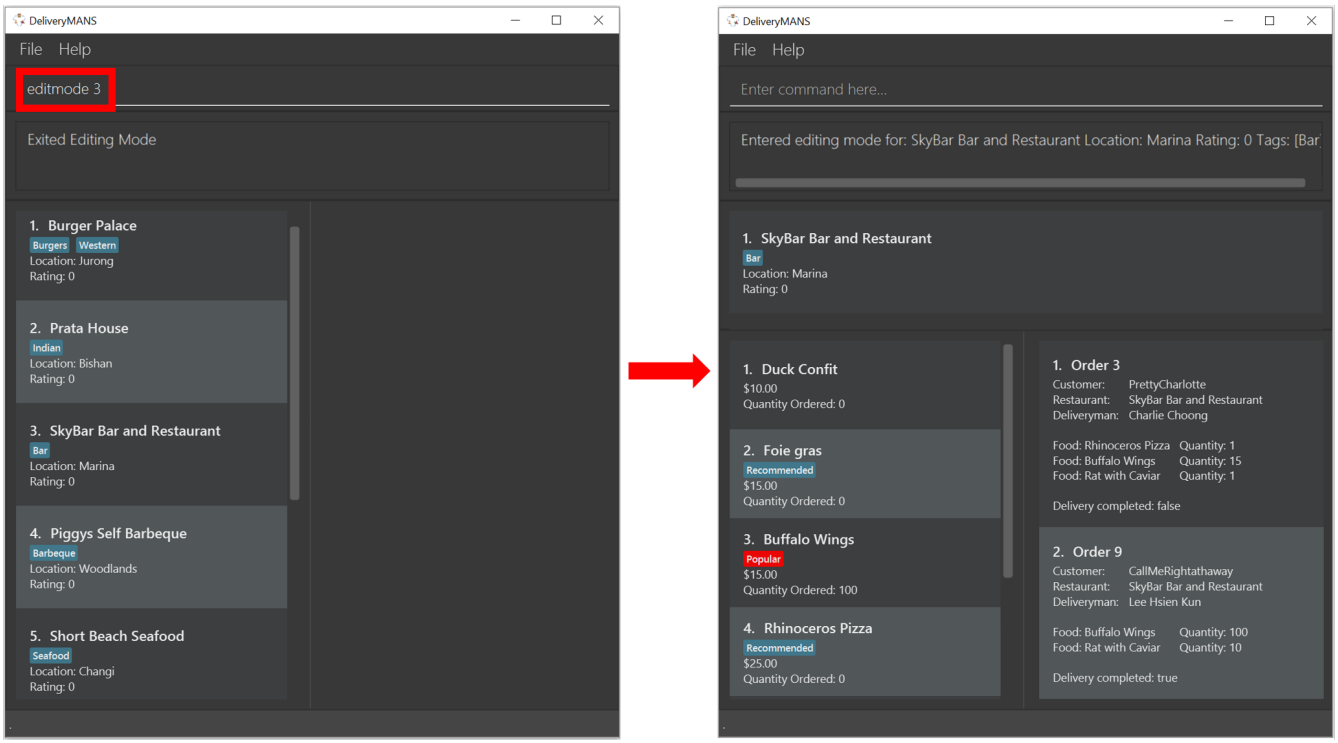
Entering EditMode: `editmode`

This command enters EditMode for the restaurant identified by the specified index in the restaurant list. Displays the restaurant's details, menu and current orders. Unlocks commands for editing details, adding and removing of food items in the menu, and adding of rating.

Format: `editmode INDEX`

- `INDEX` must be a positive integer from 1 to n, the number of restaurants in the restaurant list.

Example: `editmode 3`



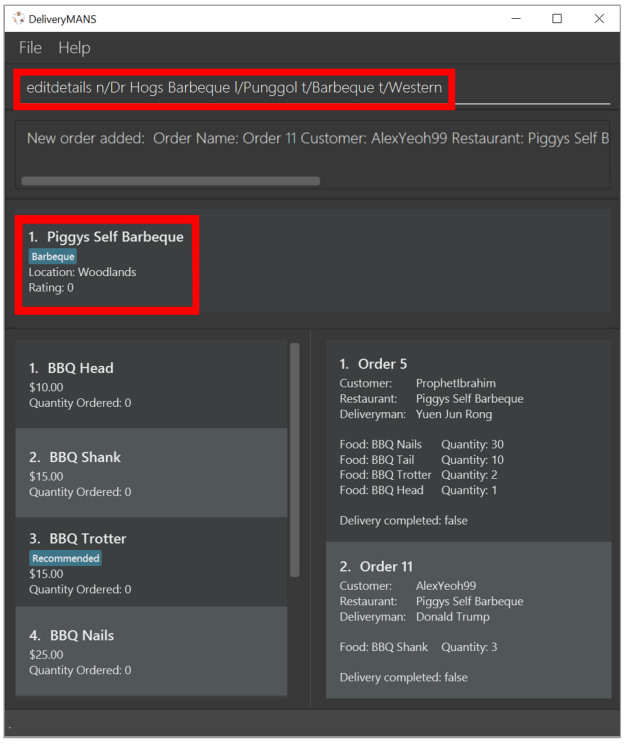
Editing restaurant's details (under EditMode): `editdetails`

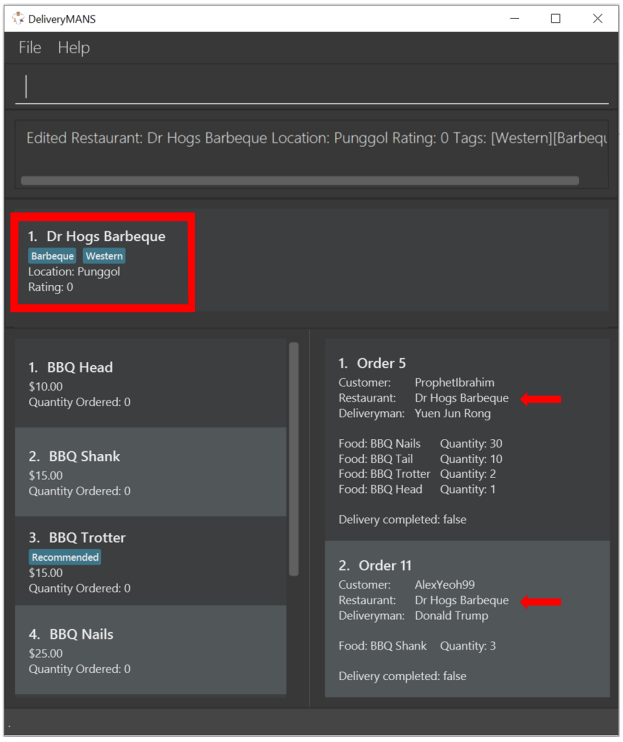
This command edits the details of the restaurant under EditMode.

Format: `editdetails [n/NAME] [l/LOCATION] [t/TAG]`

- At least one of the optional fields must be provided.
- Existing values will be updated to the input values.
- When editing tags, the existing tags of the restaurant will be removed i.e adding of tags is not cumulative.
- You can remove all the restaurant's tags by typing t/ without specifying any tags after it.

Example: `editdetails n/Dr Hogs Barbeque l/Punggol t/Barbeque t/Western`





NOTE

Note:

- Editing a restaurant's name will edit the restaurant name in all its existing orders as well.

Adding food item (under EditMode): `add`

This command adds a food item to the menu of the restaurant under EditMode.

Format: `add n/NAME a/PRICE [t/TAG]`

- You can only tag a food item as "Recommended".

Example: `add n/Chicken a/7.90 t/Recommended`

Deleting food item (under EditMode): `delete`

This command deletes the food item at the specified index in the menu of the restaurant under EditMode.

Format: `delete INDEX`

- `INDEX` must be a positive integer from 1 to n, the number of food items in the restaurant's menu.

Example: `delete 1`

NOTE

Note:

- Deleting a food item from a restaurant will NOT delete the food item in the restaurant's orders as the order was made when the food item was still available.

Adding a rating (under EditMode): `rate`

This command adds a rating to the restaurant under EditMode and updates the new average rating of all the ratings added to date.

Format: `rate RATING`

- `RATING` must be a non-negative integer from 0 to 5.

Example: `rate 4`

Exiting EditMode: `exitedit`

This command exits EditMode for the specific restaurant and returns to the list of restaurants

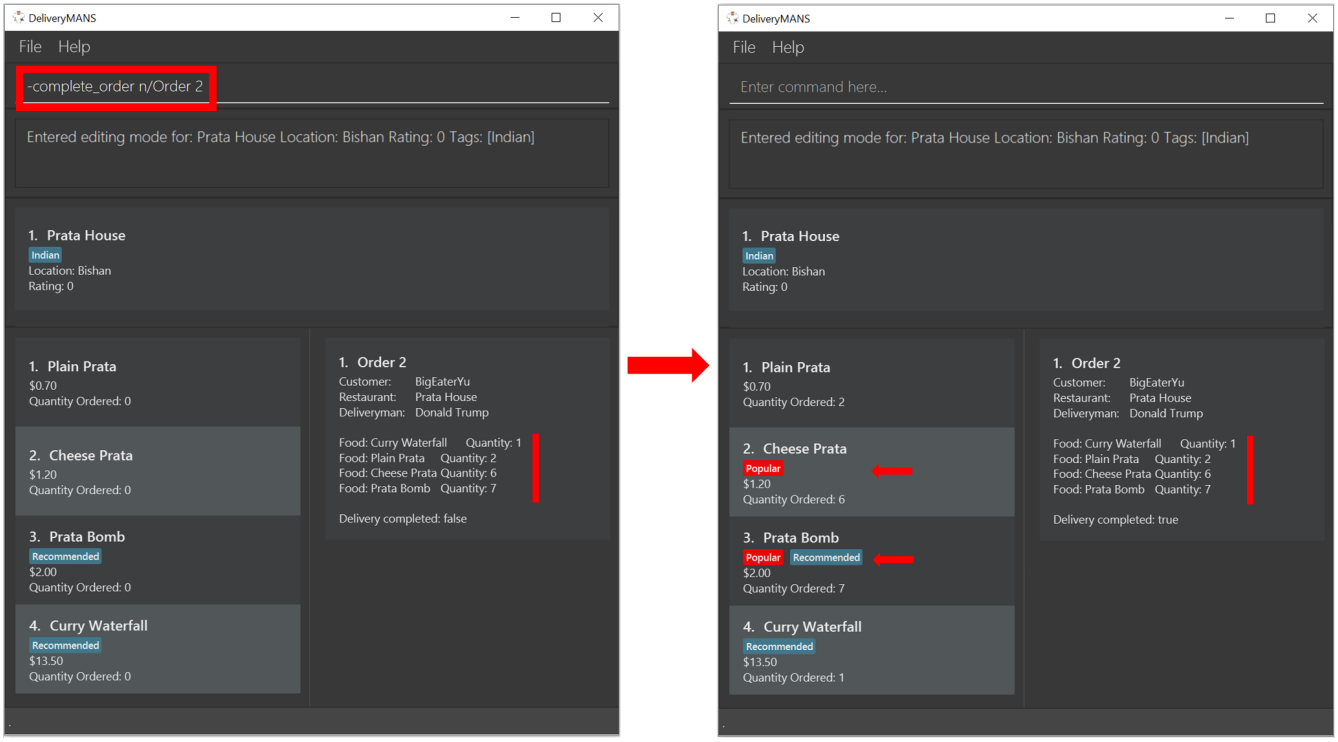
Format: `exitedit`

Auto-tagging of food item as "Popular"

Food items with quantity ordered more than 1.5 times the average quantity ordered of food items in the restaurant will be automatically tagged as "Popular".

Example (refer to image below):

- 16 food items are ordered in Order 2 (1 + 2 + 6 + 7), an average quantity ordered of 4 per food item.
- Quantity ordered of food items Cheese Prata and Prata Bomb (6 and 7 respectively) is more than 1.5 times the average quantity ordered. Automatically tagged as "Popular" upon completion of Order 2.



NOTE

Note:

- Food items' quantity ordered will only be updated upon completion of an order.

Contributions to the Developer Guide

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

EditMode for restaurants

A `Restaurant` object contains many attributes. On top of a `Name`, `Location`, `Rating` and a list of `Tag`, it also includes a list of `Food` as its menu, as well as a list of `Order`. This makes it difficult to edit an entire `Restaurant` object using just one `Command`.

The EditMode feature allows editing of a specific `Restaurant` object's details (name, location, rating, tags), menu and orders individually under 1 interface, using different commands.

Implementation

Model:

In addition to the `filteredRestaurantList` that contains all the restaurants in the restaurant database, `ModelManager` now contains a `editingRestaurantList`, which holds and allows access to the single restaurant currently under EditMode.

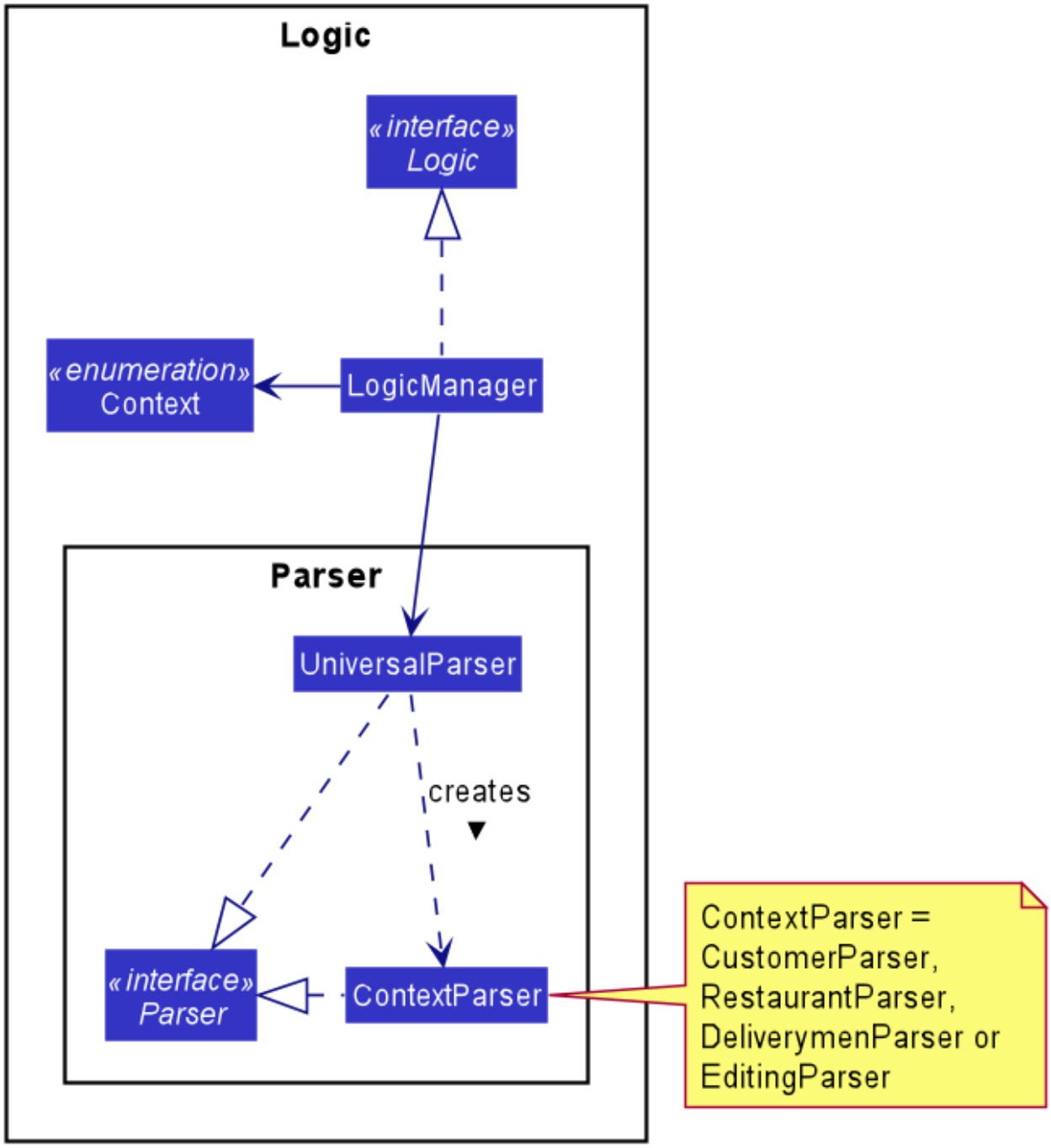
When user inputs the `editmode INDEX` command:

- The restaurant referenced by the `INDEX` in the list of restaurants will be placed in the `editingRestaurantList` via the function call `Model#setEditingRestaurant(Restaurant editingRestaurant`.
- Subsequent commands that edit the restaurant, such as `AddFoodCommand`, `DeleteFoodCommand`, `AddRatingCommand` and `EditDetailsCommand` will create a new restaurant with the edited attributes.
- The outdated restaurant will be replaced with the new edited restaurant in both the `filteredRestaurantList` and `editingRestaurantList` via the function call `Model#setRestaurant(Restaurant oldRestaurant, Restaurant newRestaurant)`.

Logic:

The `Logic` for EditMode is facilitated by `Context` enum type, which contains the following constants: `GLOBAL`, `CUSTOMER`, `RESTAURANT`, `DELIVERYMEN` and `EDITING`. It determines the `Context` the application is in, as well as the commands the user can access. It is contained inside `LogicManager` as an attribute. `EditModeCommand` is only accessible in `Context.RESTAURANT`, and entering a valid `EditModeCommand` will change the `Context` to `Context.EDITING`.

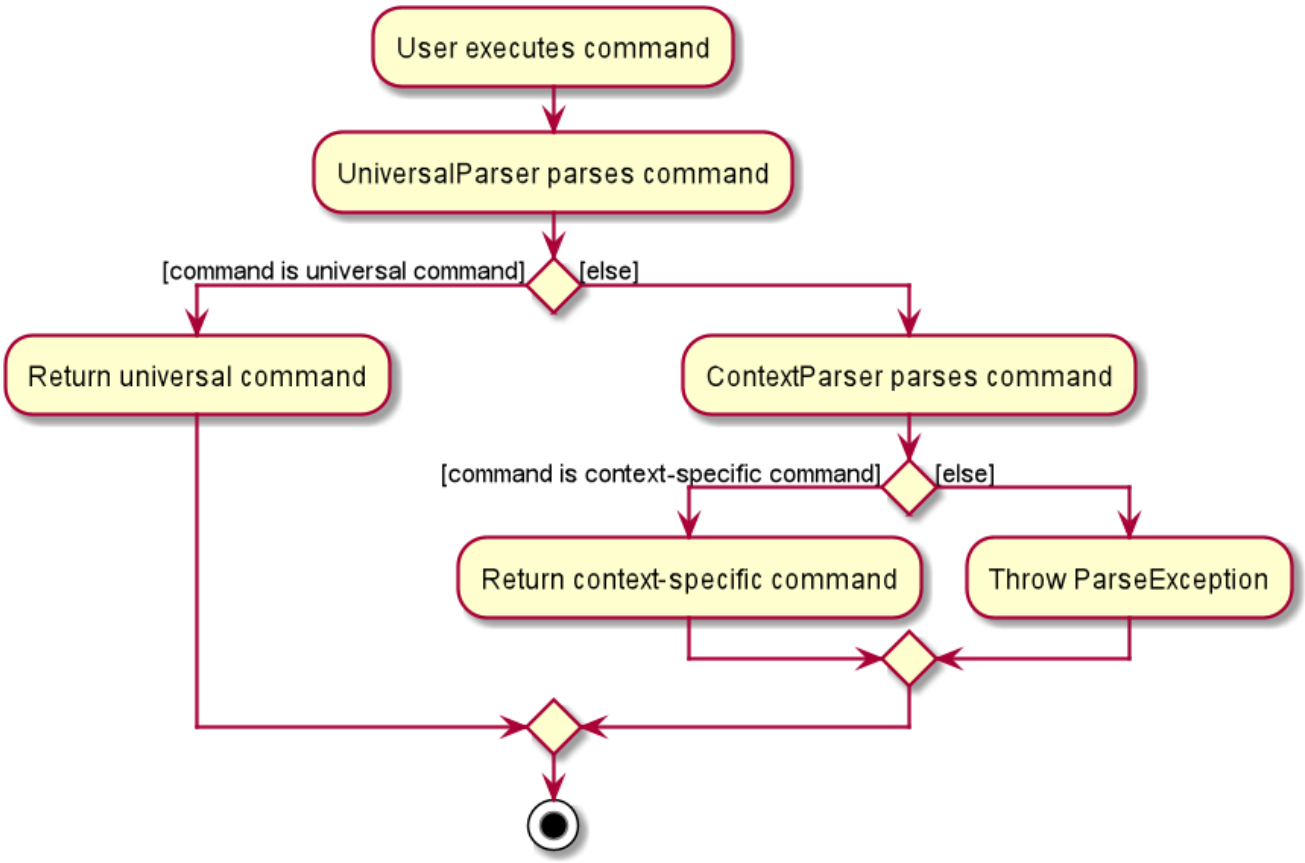
The following class diagram shows the relevant structure of `Logic` and `Parser`:



When the user inputs a command:

- `userInput` will always be parsed by `UniversalParser` first, regardless of the current `Context`. The reason for this is to check for universal commands, which are accessible in all `Context`.
- Subsequently, if the command word in `userInput` matches none of the universal commands, then `UniversalParser` will create a context specific `Parser` based on the current `Context`, i.e. `CustomerParser`, `RestaurantParser`, `DeliverymenParser`, `EditingParser`, which takes over and parses the `userInput`. Any context switching command will then change the `Context` in `LogicManager`.

The following activity diagram summarises what happens when the user enters a command:



When user inputs the `editmode INDEX` command:

- `UniversalParser` will parse it first.
- Since `editmode` matches none of the universal commands, `UniversalParser` will create a new `RestaurantParser` (since current `Context` is `Context.RESTAURANT` as `EditModeCommand` is only accessible in said `Context`).
- The new `RestaurantParser` will then parse the `userInput` and subsequently change the `Context` in `LogicManager` to `Context.EDITING`, unlocking commands to edit the restaurant, such as `AddFoodCommand`, `DeleteFoodCommand`, `AddRatingCommand` and `EditDetailsCommand`.

UI:

Commands that change the UI will either:

- Pass its command class to `MainWindow` in the UI package if the command doesn't change the `Context`
- Pass the new `Context` to `MainWindow` in the UI package if the command changes the `Context`.

as the second parameter of the `CommandResult` returned by the command.

When `MainWindow` receives the `CommandResult`, it will extract out either the command class or `Context` from `CommandResult` and make changes to the UI accordingly via the function call `MainWindow#changeDisplay(Context context)` or `MainWindow#changeDisplay(Class commandClassName)`.

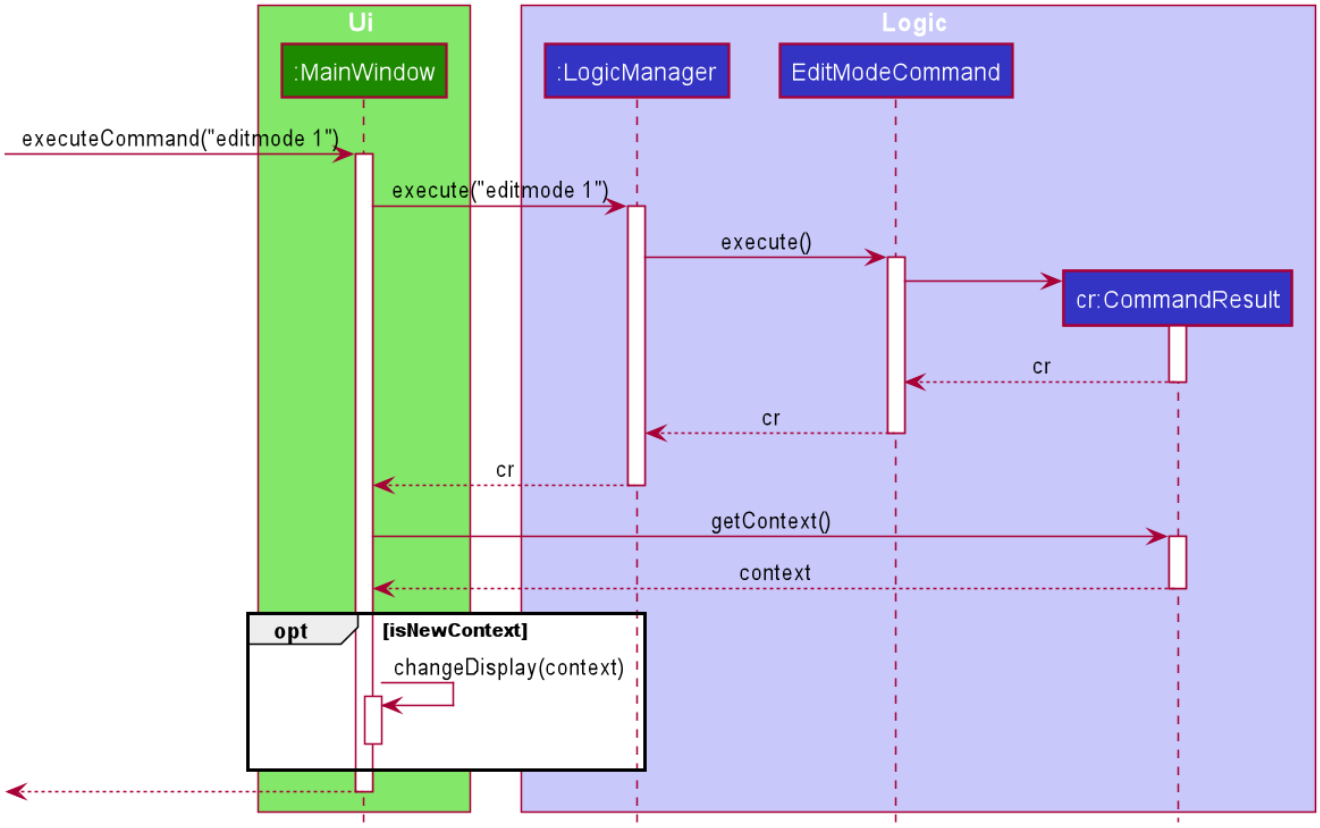
When user inputs the `EditModeCommand`:

- Since `Context` is changed to `Context.EDITING`, it will be passed as the second parameter of the

`CommandResult` returned by the `EditModeCommand`.

- Upon receiving this new `Context`, `MainWindow` will call the function `changeDisplay(Context.EDITING)` to change the UI. An extra `StackPane` showing the restaurant under EditMode will be displayed, while the `SplitPane` displaying the list of restaurants originally will now be filled with the restaurant's `Food` menu and `Order` list.

The following sequence diagram summarises how the `EditModeCommand` changes the UI:



Design Considerations

Aspect: Structure of `Logic` and `Parser`

- **Current:** `LogicManager` contains only the `UniversalParser`, which then creates a context-specific parser depending on the current `Context` in `LogicManager`.
  - Pros: Checking whether `userInput` is a universal command only needs to be done once in `UniversalParser`
  - Cons: Doesn't make as much sense for `UniversalParser` to be able to create other context-specific parsers.
- **Alternative:** Instead of containing only the `UniversalParser`, which then creates the other 4 context-specific parsers, `LogicManager` contains all 5 parsers.
  - Pros: Makes more sense to have `LogicManager` containing all 5 parsers which parses `userInput` individually based on the current context.
  - Cons: Checking whether `userInput` is a universal command needs to be repeated in each parser.

Aspect: Changing of UI

- **Current:** If the `Command` that changes the UI does not change the `Context`, pass its command name to `MainWindow` instead of creating a new `Context` to signal a change in UI.
  - Pros: Does not create unnecessary `Context`.
  - Cons: 2 method signatures are needed for `MainWindow#changeDisplay` to change the UI.
- **Alternative:** A new `Context` is created for every `Command` that changes the UI.
  - Pros: Only 1 method signature is needed for `MainWindow#changeDisplay` to change the UI.
  - Cons: Creates many unnecessary `Context` that `LogicManager` will never be in.