Cher Wei Jie - Project Portfolio

PROJECT: Delino

Overview

Delino is a desktop application for couriers to manage delivery tasks. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java and has about 10 kLoC.

Summary of contributions

- **Major enhancement**:
  - Added `return` command (Pull request: #353)
    - What it does: This command allows the user to create a new return order and add it into the Return Order List or convert a delivered order into a return order and add it into the Return Order List.
    - Justification: This feature is a must-have in Delino as it is a delivery management app. It allows the user to convert delivered orders into return orders and even create new return orders from scratch with the given parcel attributes.
    - Highlights: This feature requires a deep understanding of the AddressBook 3 (AB3) codebase and that took quite a bit of time. Two different logic execution paths had to be implemented and the testing of this feature also had to be extensive and exhaustive to ensure that no test cases were missed in the process of testing. The parametized testing functionality provided by JUnit 5 made this process to be much more convenient (https://www.baeldung.com/parameterized-tests-junit-5). Furthermore, new classes such as ReturnOrder, ReturnCommand, Parcel abstract class also had to be implemented for this feature to be fully functional. Therefore, this feature affects subsequent features to be added due to changes in inheritance of classes.
  - Added `delivered` command (Pull request: #302)
    - What it does: Allows the user to mark orders or return orders as delivered.
    - Justification: This is a must-have feature for a delivery management app like Delino. It allows the user to easily mark an order or return order as delivered after delivering the parcel.
    - Highlights: This feature required good understanding of the AB3 codebase and how the entire order list and return order list are edited to display the changes in the GUI. As such, this required a deep level of understanding of how the entire app worked so that the changes can be executed and displayed on the GUI from the start of the command till the end of execution.
  - Added the `help` command (Pull requests: #310)

        - What it does: It displays a list of all Delino's commands and llows the user to see the usage of each command in Delino and quickly be able to utilise Delino's available features.
        - Justification: This command was rather simple to implement but the main problem faced was deciding the right window size for the pop-up help window as different screens have different resolutions and this resulted in some bugs that had to be re-looked at and fixed.

- **Minor enhancement**:
    1. Updated Delino's icon image and link in the Developer Guide (Pull request: #245)
    2. Added Non-functional requirements (NFR) in the Developer Guide (Pull request: #89)
    3. Wrote Must-haves for user stories in Developer Guide (Pull request: #65)
    4. Did the first mock-up for Delino's UI using Adobe XD (Pull request: #85)
- **Code contributed**: [Functional code & Test code]
- **Other contributions**:
    - Project management:
        1. In charge of team facilitation, code integration and UI design.
        2. Set up Codacy with Jeremy to improve code quality of Delino's repository. (Pull request: #61)
        3. Ensure that project deliverables are completed on time.
    - Enhancements to existing features:
        - Adapted the given Address Book UI into Delino's first draft UI. (Pull request: #85)
    - Documentation:
        - Updated Developer Guide to include Delivered Command (Pull request: #253)
        - Updated Developer Guide to include Return Command (Pull request:)
        - Updated Developer Guide to include Help Command (Pull request:)
    - Community:
        - PRs reviewed (with non-trivial review comments): (Pull requests: #118, #191, #226)
        - Reported bugs and suggestions for other teams in the class (examples: https://github.com/Cherweijie/ped/issues)
    - Tools:
        - Set up Codacy and updated its badge to link to our repository together with Jeremy (Pull request: #61)

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.

Introduction

This user guide provides in-depth documentation on the Delino desktop application: quick start guide, features, FAQ, command summary, and glossary.

Delino is for couriers who **prefer to use a desktop app for managing their delivery tasks**. It is **optimised for those who have a strong preference towards Command Line Interface** (CLI) while still enjoying the benefits of a Graphical User Interface (GUI). Sounds interesting? Have a quick look at our tutorial [Quick Start] to get started. Have a delightful journey using Delino!

Showing the Help message - `help` [Done by - Cher Wei Jie]

- In this section, you will learn more about the `help` command and how to use it.
  Why would you want to use the `help` command? You can use this `help` command to see a summary of all available features of Delino.

How to use the Help command?

Here is how you can use the `help` command to show a summary of all available commands in Delino.

Step 1 : Type in the keyword `help`.
Step 2 : Press `Enter` on your keyboard to see the magic!

What constitutes a valid Help command?

The syntax for a valid `help` command can be seen below!

- `help`

Table 1. Possible Combinations of Help command

Scenario	Command	Result
If you want to view all the available commands in Delino.	`help`	A pop-up window will be shown that includes a summary to briefly explain all the commands that Delino offers.

Returning an order : `return` [Done by - Cher Wei Jie]

- In this section, you will learn more about the `return` command and how to use it.
  Why would you want to use the `return` command? You can use this `return` command to create a new return order to be added into the return order list.

How to use the Return command?

IMPORTANT	Return orders inserted are sorted by their delivery date and time.

Here is how you can convert an order into a return order or create a new return order by following the steps below:

Step 1 : Type in the keyword `return`.
Step 2a : If you would like to convert an existing order into a return order, provide the `TRANSACTION_ID` corresponding to the order to be converted.
Step 2b: If you would like to create a new return order in the return list, provide the `TRANSACTION_ID CUSTOMER_NAME ADDRESS PHONE_NUMBER RETURN_TIMESTAMP WAREHOUSE_LOCATION CUSTOMER_EMAIL [COMMENTS_BY_CUSTOMER] [TYPE OF ITEM]` of the parcel.
Step 3 : Press `Enter` on your keyboard to see the magic!

NOTE

- Please include a whitespace in between the keyword & transaction id or keyword & the attributes aforementioned.
  i.e. `return tid/123abcd`
  or
  `return tid/A999999 n/John Doe p/98765432 a/311 Clementi Ave 2 #02-25 S120363 e/johndoe@gmail.com rts/2020-05-05 1500 w/5 Toh Guan Rd E #02-30 S608831 c/NIL type/glass`

What constitutes a valid Return command?

The syntax for a valid `return` command can be seen below!

- `return TRANSACTION_ID RETURN_TIMESTAMP` or `return TRANSACTION_ID CUSTOMER_NAME ADDRESS CUSTOMER_EMAIL WAREHOUSE_LOCATION PHONE_NUMBER RETURN_TIMESTAMP [COMMENTS_BY_CUSTOMER] [TYPE_OF_ITEM]`

NOTE

- The `TRANSACTION_ID` refers to the transaction id of a parcel.
- The `CUSTOMER_NAME` refers to the name of the recipient of the parcel.
- The `ADDRESS` refers to the location which the return should be picked up from. It **must** have a postal code.
- The `WAREHOUSE_LOCATION` refers to the location which the return should be delivered to.
- The `PHONE_NUMBER` refers to the phone number of the customer.
- The `RETURN_TIMESTAMP` refers to the return date and time of the parcel. This field cannot be earlier than the current date and time. Also, it cannot be before the delivery time stamp of the order to be converted.
- The `COMMENTS_BY_CUSTOMER` is an optional attribute which can be included if the customer has special requests.
- The `TYPE_OF_ITEM` is an optional attribute which can be included if the item requires special attention. For example, it can be used when the item is fragile.

These are the possible combinations of the `return` command:

Table 2. Possible Combinations of Return command

Scenario	Command	Result
If you want to convert the order with Transaction Id `abc1234` into a return order and display it on returns list.	`return tid/abc1234 rts/2020-05-05 1600`	This order will be removed from the order list and be added into the returns list as a return order with the updated return time stamp.
If you want to create a new return order in the return order list.	`return tid/123abcd n/weijie a/311 Clementi Ave 2 #02-25 S120363 p/92123412 w/5 Toh Guan Rd E #02-30 S608831 rts/2020-05-05 1600`	A return order with the input attributes will be created into the return order list.

NOTE

- The `TRANSACTION_ID` given belongs to the return order that will be created.
- The `RETURN_TIMESTAMP` has to be before the delivery time stamp of the order.
- A return order will be created after executing this `return` command.
- If the parcel is an existing order, it will be removed from the order list and converted into a return order and added into the return order list.
- If the parcel is a new return order, it will be created and added into the return order list.
- All return orders will not have the `CASH_ON_DELIVERY` attribute as the item was already delivered and the money already was collected upon delivery.

WARNING

- The conversion of an order into a return order can only be done if the order was already delivered.
- The `TRANSACTION_ID` **is alphanumeric**, e.g: 123asd, 1234567, abcdef.
- The `CUSTOMER_NAME` **must consist of only alphabets**.
- The `ADDRESS` **is alphanumeric**.
- The `CUSTOMER_PHONE` **must be numeric**
- The `WAREHOUSE_LOCATION` **is alphanumeric**
- The `RETURN_TIMESTAMP` **must follow the yyyy-mm-dd format must not be earlier than the current date and time**
- The `COMMENTS_BY_CUSTOMER` **is alphanumeric**
- The `TYPE_OF_ITEM` **must** consist of only alphabets (No numbers or special characters allowed).
- The resulting created return order will not have the `CASH_ON_DELIVERY` field as the money was already collected.

Delivering an order or return order : `delivered` [Done by - Cher Wei Jie]

In this section, you will learn more about the `delivered` command and how to use it.
Why would you want to use the `delivered` command? If you have delivered an order or return order, you can mark it as delivered with the `delivered` command.

How to use the Delivered command

This section will explain the steps needed to use the `delivered` command.

Here is how you can mark the details of any order or return order by following the steps below:

Step 1 : Type in the keyword `delivered`
Step 2 : Provide the `FLAG` corresponding to the parcel order type you want to mark as delivered
Step 3 : Provide the `INDEX` of the parcel displayed on the screen that you wish to mark as delivered
Step 4 : Press `Enter` on your keyboard to see the magic!

NOTE

- Please include a whitespace in between the keyword, command flag and index. i.e. `delivered -o 1`
- If you can't see any orders use the `list` command to view existing parcel! If nothing is showing up, it means you got to `insert` or `return` some parcel and start doing work!

What constitutes a valid Delivered command

The syntax for a valid `delivered` command can be seen below!

- `delivered FLAG INDEX`

These are the possible combinations of the `delivered` command:

Table 3. Possible Combinations of Delivered command

Scenario	Command	Result
If you want to mark the first return order displayed on returns list as delivered.	`delivered -r 1`	The delivery status of the first return order displayed on the returns list will be changed to "Returned to Warehouse"
If you want to mark the second order displayed on the orders list.	`delivered -o 2`	The delivery status of the second order in the order list will be changed to "Delivered".

NOTE

- The `INDEX` given is the parcel you will be marking as delivered.
- The parcel will be marked as delivered after executing this `delivered` command.
- If the parcel is an order, its delivery status will be changed from "Not Delivered" to "Delivered".
- If the parcel is a return order, its delivery status will be changed from "Not returned to warehouse" to "Returned to warehouse".

WARNING

- The `INDEX` **must be a positive integer**, e.g: 1, 2, 3, ...
- The `INDEX` **must be in range** of the number of displayed orders.
- Only can be used when there is at least an order or return order displayed.
- The `FLAG` can only be either `-o` or `-r`, please refer to here for more information.

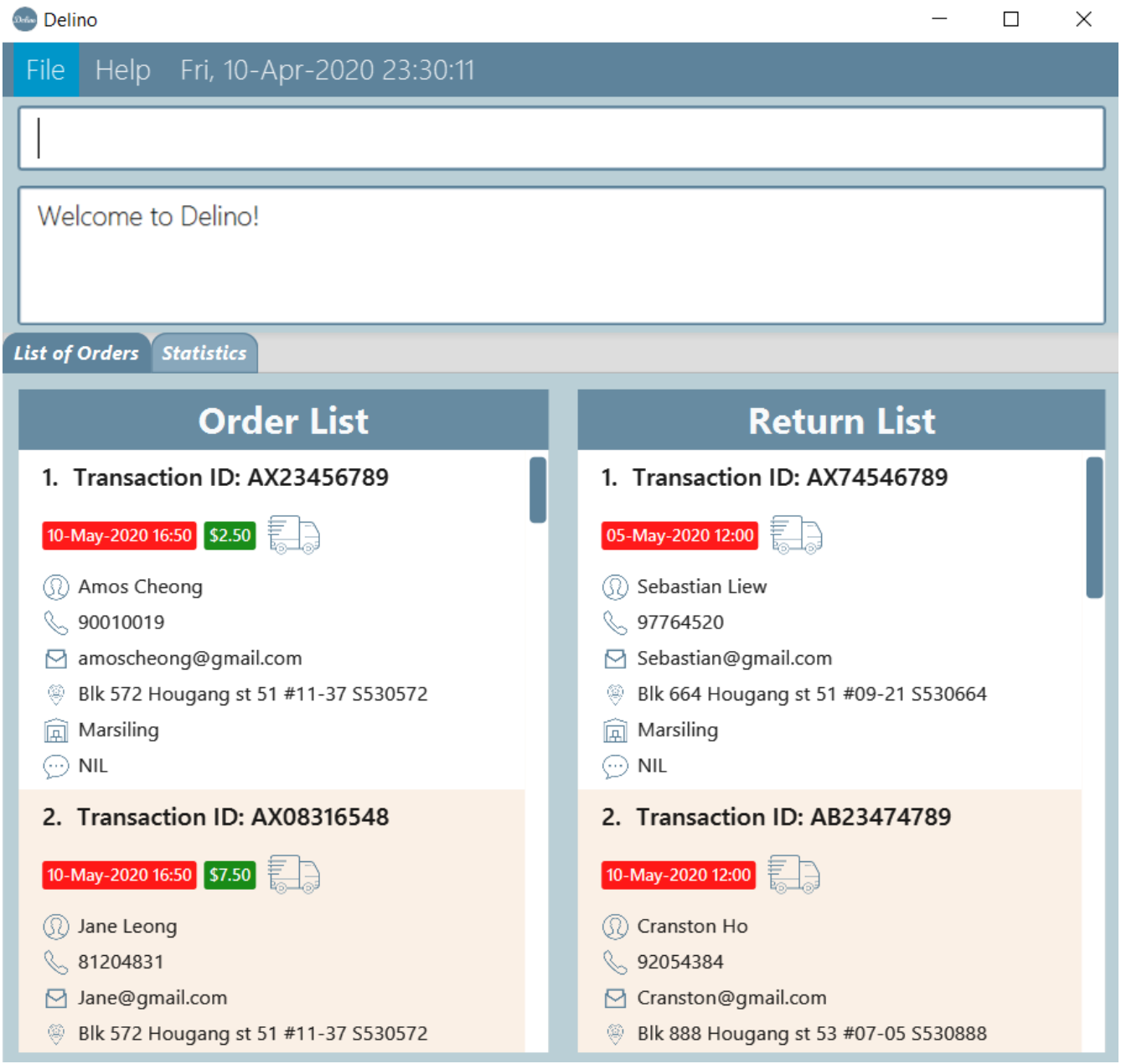


Figure 1. Graphical User Interface (GUI) of Delino

Product Information

Are you still using pen and papers or Excel to keep track of all your orders?

With Delino, you can be rest assured that it will help you to keep track of a large number of orders in the most efficient way possible. Are you interested in improving your parcel delivery management? Read further to discover the features of Delino!

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.

Design

Model component

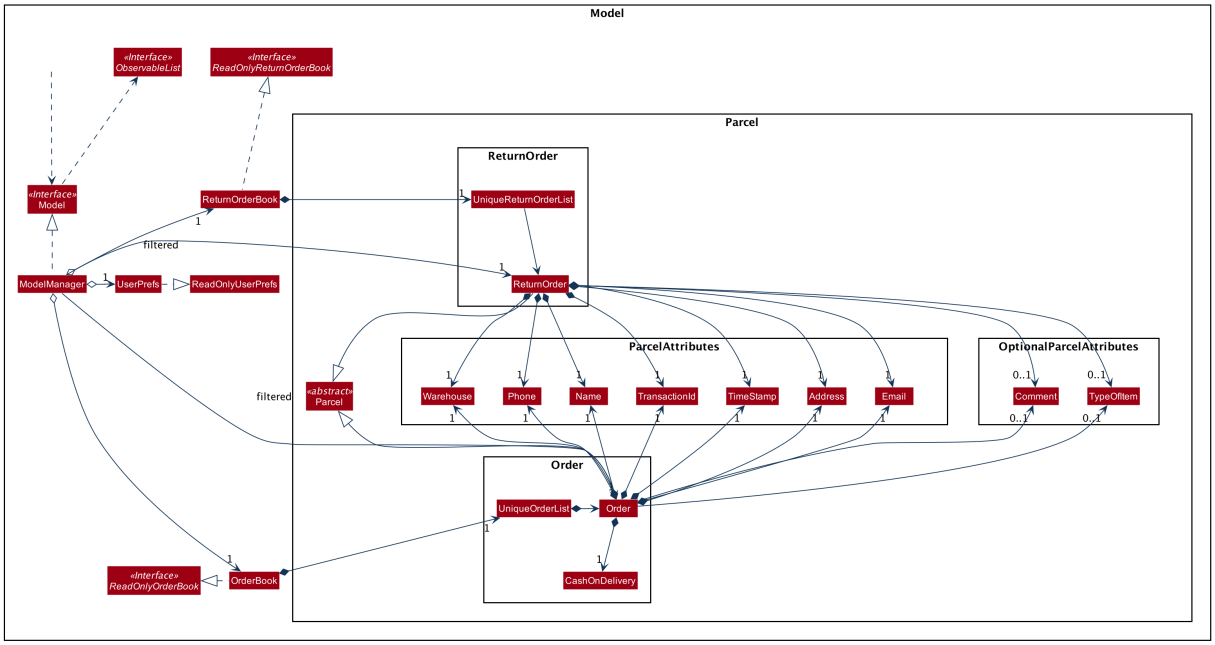


Figure 2. Structure of the Model Component

API : `Model.java`

The **Model**,

- stores a **UserPref** object that represents the user's preferences.
- stores the Order Book and Return Order Book data.
- exposes two unmodifiable lists, the `ObservableList<ReturnOrder>` and `ObservableList<Order>` that can be 'observed'.
  e.g. The UI can be bound to this list so that the UI automatically updates when the data in the list change.

- does not depend on any of the other three components.

NOTE	An **Order** class consists of ten different fields as shown in the image. Every order is part of a **UniqueOrderList** and every **UniqueOrderList** is part of an **OrderBook**. Similarly, a **ReturnOrder** class consists of nine different fields as shown in the image. Every return order is part of a **UniqueReturnOrderList** and every **UniqueReturnOrderList** is part of a **ReturnOrderBook**.

Features

Return Feature

In this section, the functionality of the `return` feature, the expected execution path, the structure of the **ReturnCommand** class and the interactions between objects with the **ReturnCommand** object will be discussed.

What is the Return Feature

The `return` feature allows the user to either:
1. Create a new return order from his/her input parcel attributes.
2. Convert an existing delivered order to a return order.

The return feature was implemented as a **ReturnCommand** in the Logic package.

The `return` command has two possible formats:

1. `return TRANSACTION_ID RETURN_TIMESTAMP` If the user provides a valid `TRANSACTION_ID` and `RETURN_TIMESTAMP` in his input, the order with the given `TRANSACTION_ID` will be converted into a return order with the same attributes but with the updated `RETURN_TIMESTAMP`. The created return order will be added into the return order list.
2. `return TRANSACTION_ID NAME ADDRESS PHONE_NUMBER EMAIL RETURN_TIMESTAMP WAREHOUSE_LOCATION [COMMENTS] [ITEM_TYPE]` If the user provides these compulsory parcel attributes, a return order with the given parcel attributes will be created and added to the return order list.

NOTE

1. All return orders do not have the `CASH_ON_DELIVERY` parcel attribute.
2. The `TRANSACTION_ID` is alphanumeric, which determines the `TRANSACTION_ID` of the resulting return order.
3. The `NAME` consists of alphabets and determines the `NAME` of the resulting return order.
4. The `ADDRESS` is alphanumeric and determines the `ADDRESS` of the resulting return order.
5. The `PHONE_NUMBER` consists of only numbers and determines the `PHONE_NUMBER` of the resulting return order.
6. The `EMAIL` is alphanumeric and determines the `EMAIL` of the resulting return order.
7. The `RETURN_TIMESTAMP` should include the date in YYYY-MM-DD format and time in 24-hour format with a whitespace in between the date and time.
8. The `WAREHOUSE_LOCATION` is alphanumeric and it determines the `WAREHOUSE_LOCATION` of the resulting return order.
9. The `[COMMENTS]` is an optional alphanumeric field and determines the `[COMMENTS]` of the resulting return order.
10. The `[ITEM_TYPE]` is an optional alphabetic field and determines the `[ITEM_TYPE]` of the resulting return order.

Execution paths of Return command

In this section, you will learn more about the execution paths for the `return` command.

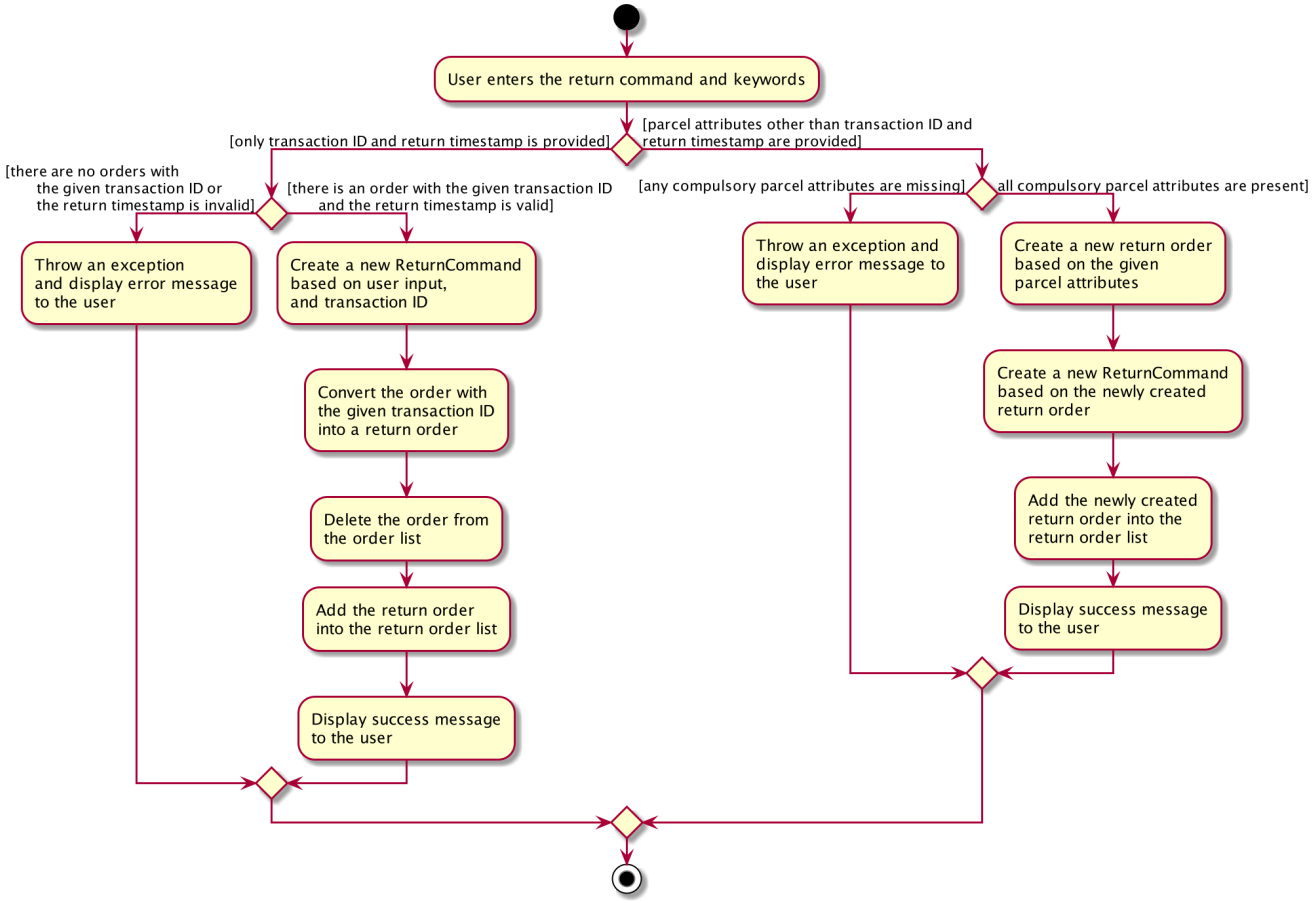
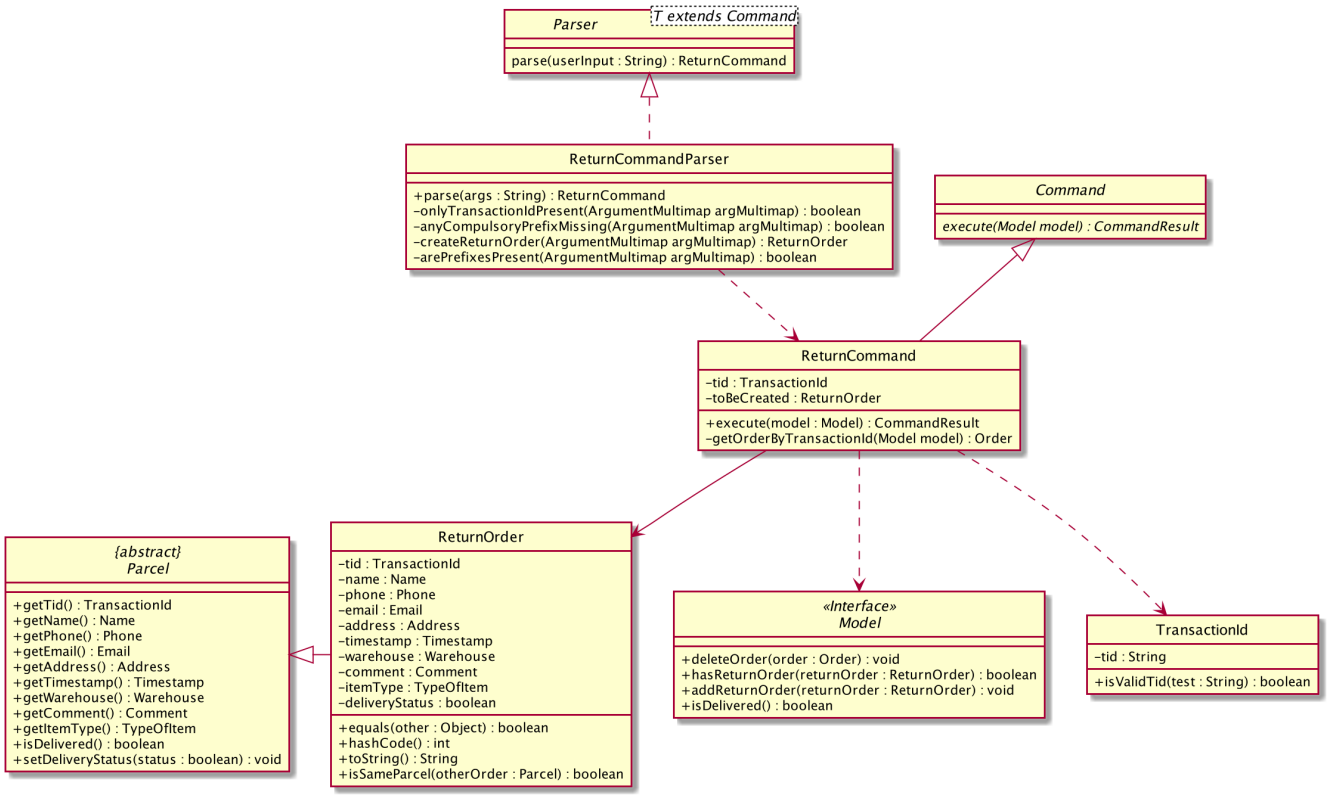


Figure 3. Return Command Activity Diagram

There are three possible execution paths for the `return` command

1. User provides an invalid `return` command input
   This will result in a parse exception and an error message will be displayed to the user.
2. User provides a valid `return` command input with a valid `TRANSACTION_ID`, i.e.
   `return TRANSACTION_ID`
   If the order with the given `TRANSACTION_ID` is delivered, it will be converted to an existing order with the given `TRANSACTION_ID` into a return order.
   This return order will then be added into the return order list.
3. User provides a valid `return` command input with all compulsory parcel attributes, i.e.
   `return TRANSACTION_ID NAME ADDRESS PHONE_NUMBER EMAIL RETURN_TIMESTAMP WAREHOUSE_LOCATION [COMMENTS] [ITEM_TYPE]`
   If the given `TRANSACTION_ID` does not exist as an order or return order, this will create a new return order based on the given parcel attributes and the resulting return order will be added to the return order list.
   If the given `return TRANSACTION_ID` already exists as an order or return order, an error message will be displayed to the user that an order or return order already exists in the order list or return order list respectively.
4. User provides a valid `return` command input with an invalid `TRANSACTION_ID`
   This will result in a parse exception and an error message will be displayed to the user.
5. User provides a valid `return` command but one or more of the compulsory parcel attributes is/are invalid. This will result in a parse exception and an error message will be displayed to the user.

Structure of Return Command



The above class diagram shows the structure of the **ReturnCommand** and its associated classes and interfaces. Some methods and fields are not included because they are not extensively utilised in **ReturnCommand**; such as public static fields and getter/setter methods.

Interactions between objects when Return Command is executed

The sequence diagrams for the `return` command are shown below.

Sequence Diagram for converting an order into a return order

The sequence diagrams for the **Return Command** are shown below.

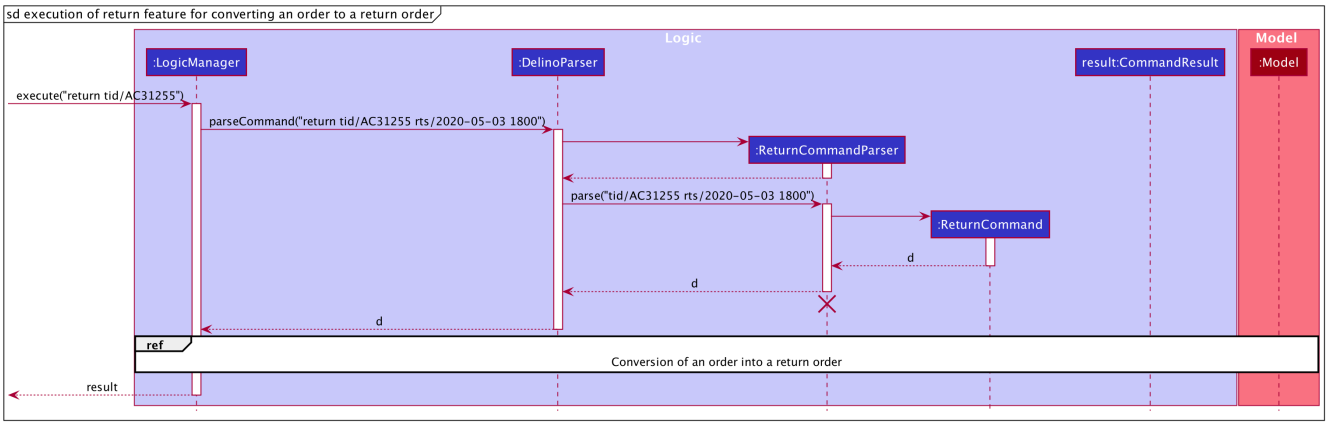


Figure 4. Return Command Sequence Diagram

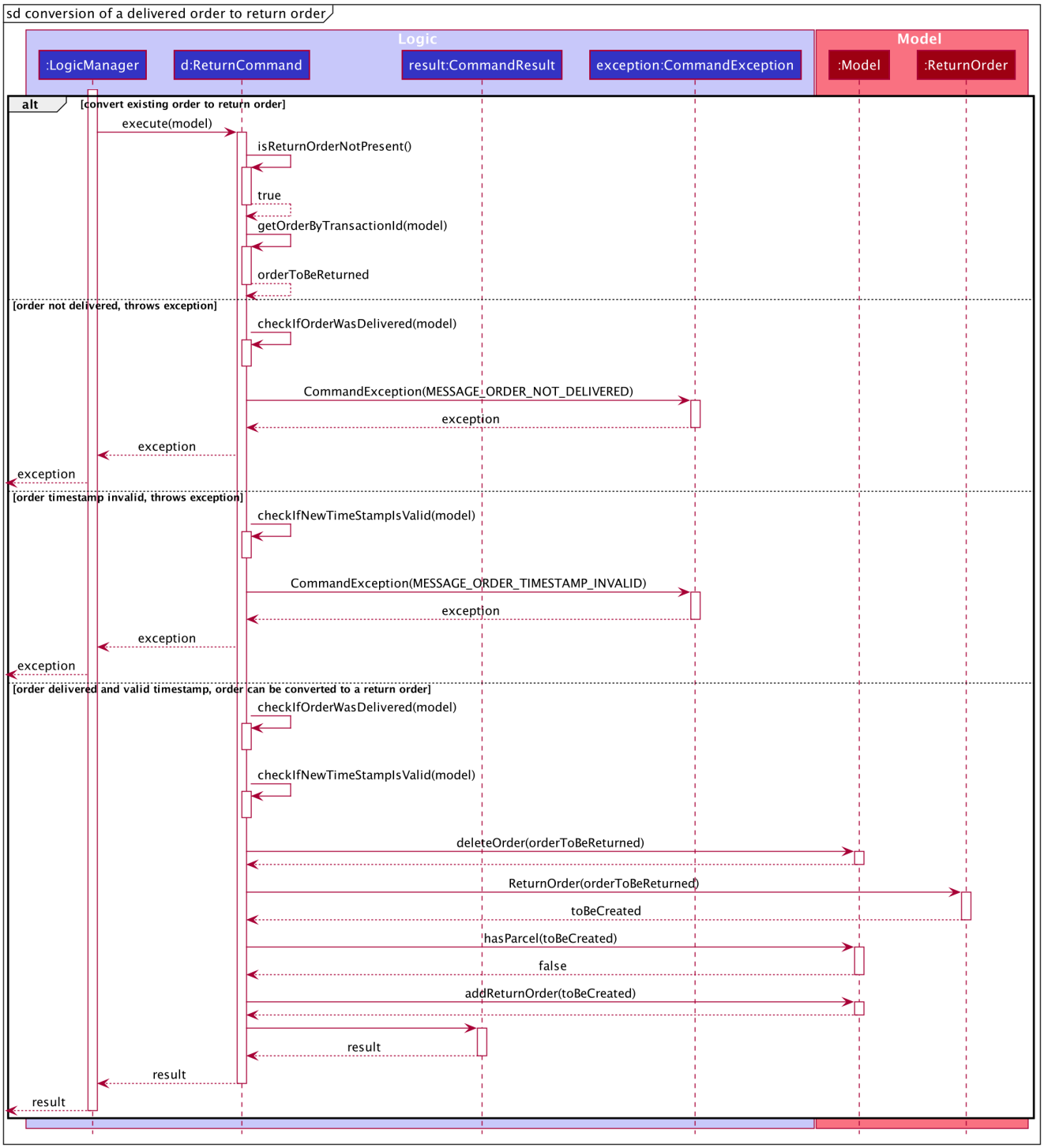


Figure 5. Execution of Return Command to convert a delivered Order into a Return Order

The arguments typed into Delino by the user will first be done by the `execute` method in `LogicManager`. After which, an `DelinoParser` object will be created to parse the input which is determined by the command word via the parseCommand method. In this case, it is the `return` command word that will be parsed.

Then, a `ReturnCommandParser` object will be created to parse the arguments after removing the command word `return` from the user's input. Based on the command word `return`, a `ReturnCommand` object will be created.

Subsequently, the parseCommand method in `LogicManager` will continue to create a CommandResult based on the validity of the user's input; which is determined by the `execute` method in

`ReturnCommand`.

The `execute` method of `ReturnCommand` will first check if the return order in the constructor of ReturnCommand is present. In this case, since we are converting an order into a return order, the return order will not be present in the constructor of `ReturnCommand` and the isReturnOrderNotPresent() method will return true.

If the given `TRANSACTION_ID` exists in the order list, the getOrderByTransactionId(model) method will attempt to create a new Order object from the model's Order list based on the given transaction ID, i.e. orderToBeReturned.

The checkIfOrderWasDelivered(model) method checks if the newly created Order is delivered. If the order was not delivered, it will throw a command exception and display an error message to the user.

If the order was delivered, Delino will proceed to check if the timestamp input was valid. If it was invalid, an exception will be thrown and an error message will be displayed to the user.

If the order was delivered and the timestamp input was valid, the deleteOrder(orderToBeReturned) method will be triggered to delete the order from the model's order list. Also, a new return order will be created based on the ReturnOrder's constructor that takes in an Order, i.e. ReturnOrder(orderToBeReturned). This creates a new Return Order object, toBeCreated.

Subsequently, this newly created ReturnOrder object toBeCreated, will be checked against the model's return order list using the hasParcel(toBeCreated) method. If it exists, a command exception will be thrown and an error message will be displayed to the user.

If the ReturnOrder does not exist in the model's return order list, the newly created ReturnOrder object, toBeCreated, will be added to the model's return order list using the addReturnOrder(toBeCreated) method.

Finally, a new CommandResult will be created to display the success message to the user for converting a delivered order to a return order. `

Sequence Diagram for creating a new return order

The sequence diagrams for the **Return Command** are shown below.

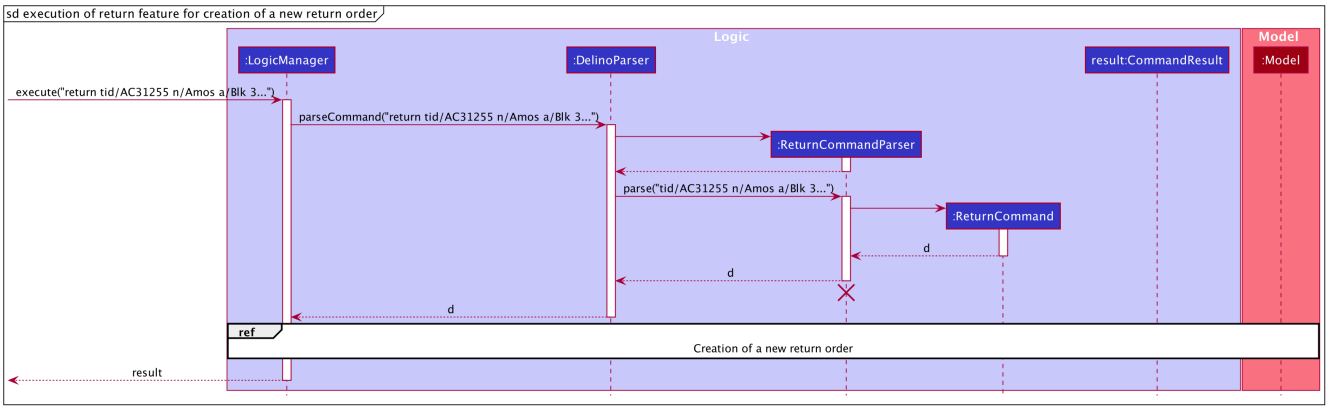


Figure 6. Return Command Sequence Diagram

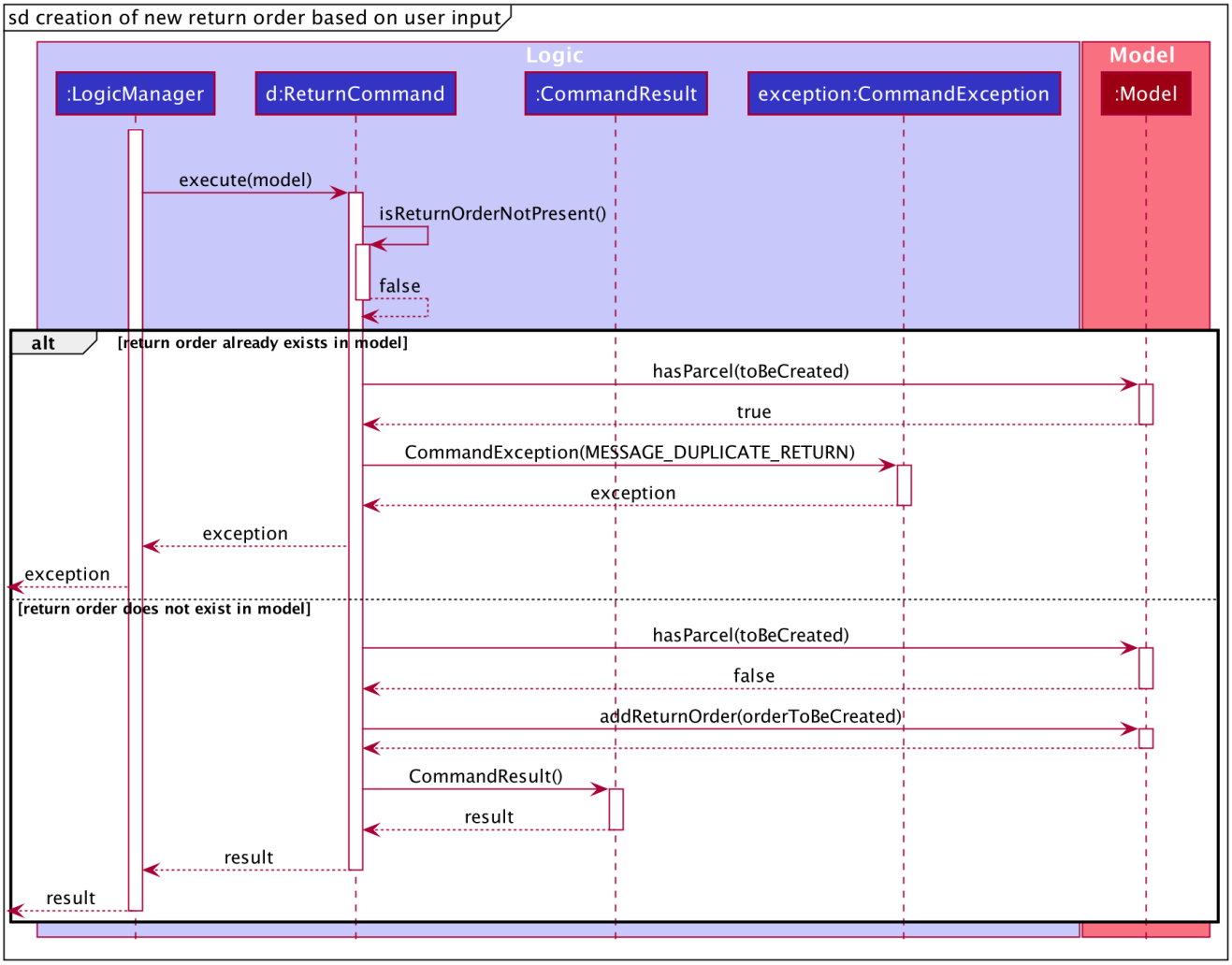


Figure 7. Execution of Return Command to create a new Return Order

The arguments typed into Delino by the user will first be done by the `execute` method in **LogicManager**. After which, an **DelinoParser** object will be created to parse the input which is determined by the command word via the parseCommand method. In this case, it is the `return` command word that will be parsed.

Then, a **ReturnCommandParser** object will be created to parse the arguments after removing the command word `return` from the user's input. Based on the command word `return`, a **ReturnCommand** object will be created.

Subsequently, the parseCommand method in **LogicManager** will continue to create a CommandResult based on the validity of the user's input; which is determined by the `execute` method in **ReturnCommand**.

The `execute` method of **ReturnCommand** will first check if the return order in the constructor of **ReturnCommand** is present. In this case, since we are creating a new return order from the given parcel attributes, a return order will be created and it will be used in the constructor of **ReturnCommand** and the isReturnOrderNotPresent() method will return false.

Also, a new return order will be created based on the **ReturnOrder**'s constructor that takes in an **Order**, i.e. ReturnOrder(orderToBeReturned). This creates a new **ReturnOrder** object, toBeCreated.

Subsequently, this newly created **ReturnOrder** object toBeCreated, will be checked against the

model's return order list using the hasParcel(toBeCreated) method. If it exists, a command exception will be thrown and an error message will be displayed to the user.

If the **ReturnOrder** does not exist in the model's return order list, the newly created **ReturnOrder** object, toBeCreated, will be added to the model's return order list using the addReturnOrder(toBeCreated) method.

Finally, a new **CommandResult** will be created to display the success message to the user for creating a new return order with the given parcel attributes.

Help Feature

In this section, the functionality of the `help` feature, the expected execution path, the structure of the `HelpCommand` class and the interactions between objects with the `HelpCommand` object will be discussed.

What is the Help Feature

The `help` feature was implemented as the **HelpCommand** in the logic package.
The `help` feature allows users to save the trouble of adding the delivery orders and the return orders one by one when they have large amount of delivery orders or return orders to add into Delino.

Execution paths of the Help command

The execution path of the **HelpCommand** is shown below:

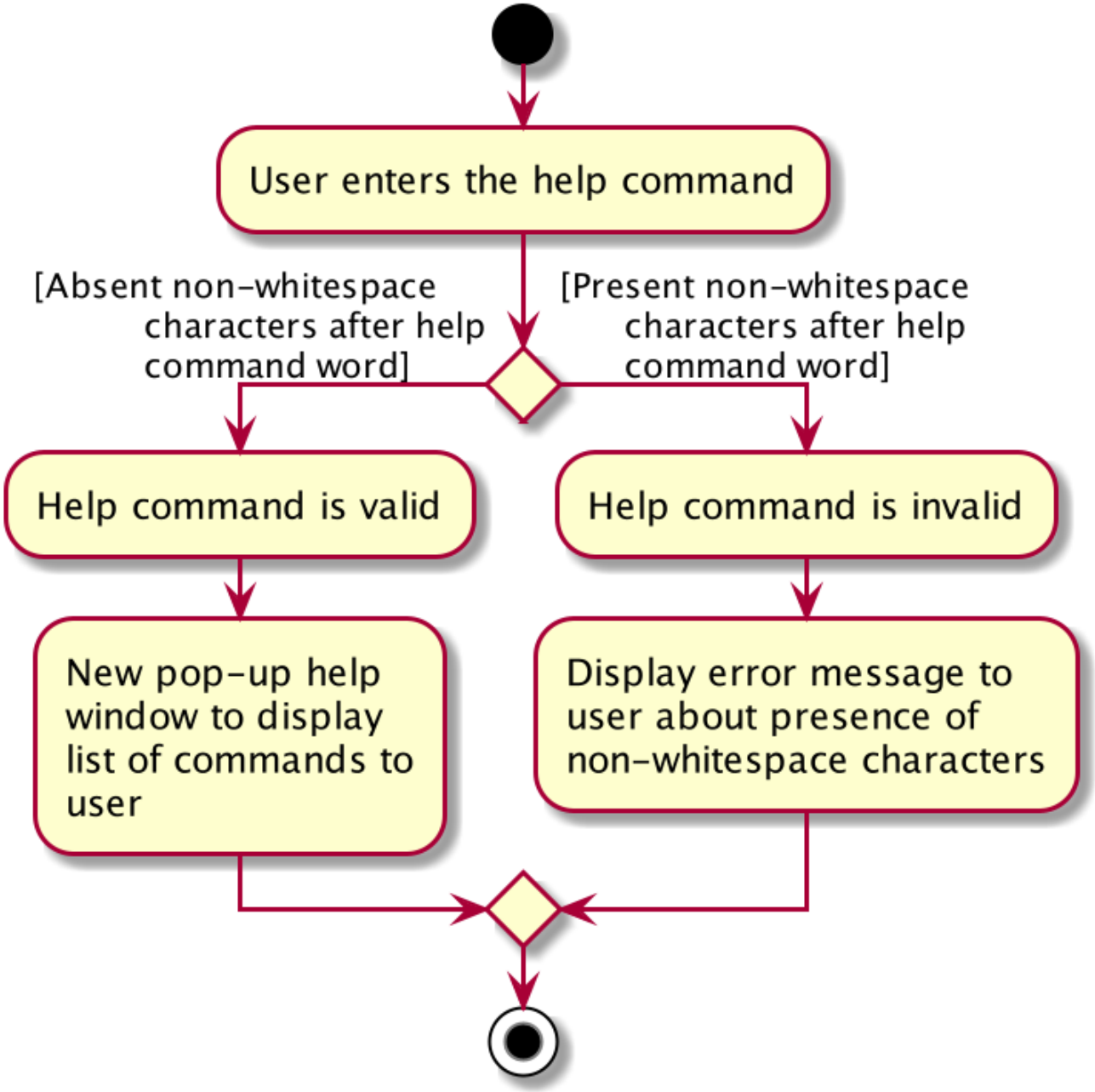


Figure 8. Help Command Activity Diagram

After the user enters the help command word, there will be a validation check to ensure that there are no non-whitespace characters following after the `help` command word so that the help command can be processed as a valid command.

If there are non-whitespace characters following `help` command word, a **ParseException** object will be created and thrown to the user by displaying an error message.

If there are no non-whitespace characters following the `help` command word, a new **HelpCommand** object will be created and a **CommandResult** object will be created subsequently to display the success message to the user.

Structure of Help Command

The following diagram shows the overview structure of the **HelpCommand** Class Diagram:

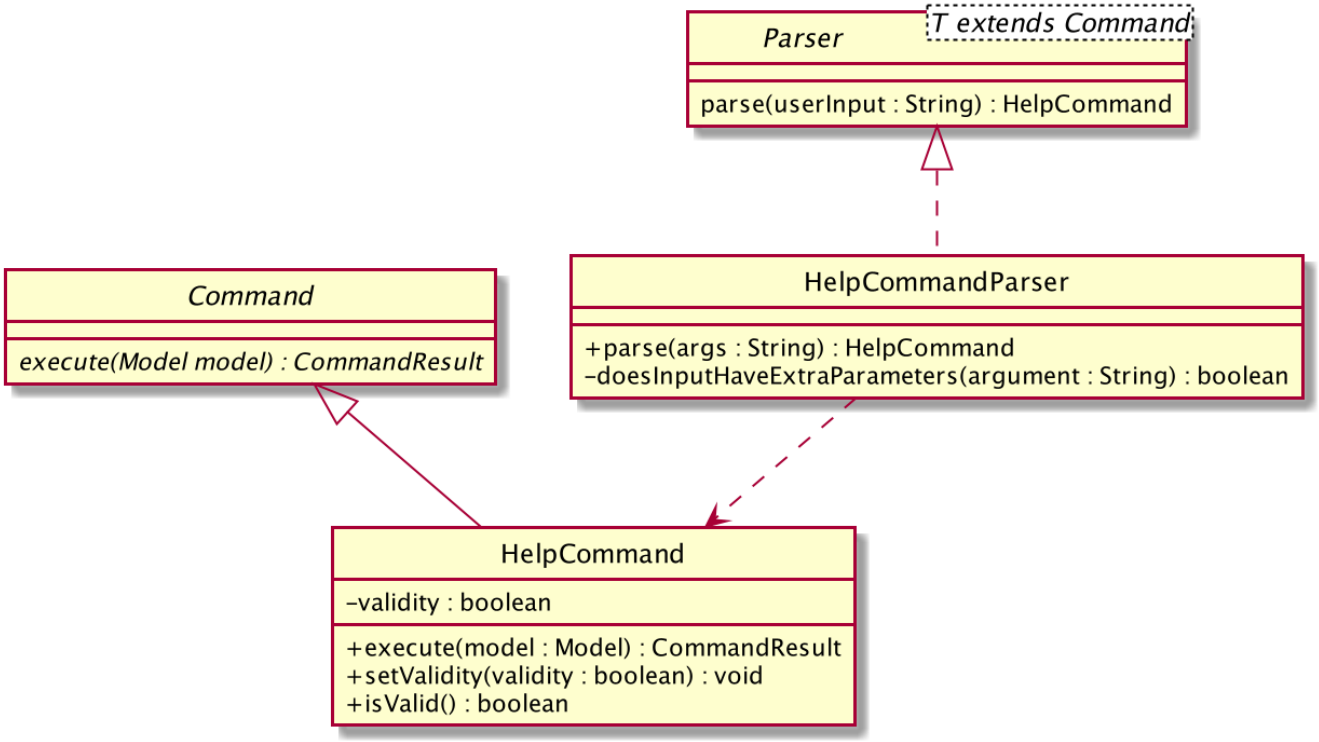


Figure 9. Help Command Class Diagram

The above class diagram shows the structure of the **HelpCommand** and its associated classes and interfaces. Some methods and fields are not included because they are not extensively utilised in **HelpCommand**; such as public static fields and getter/setter methods.

Interactions between objects when Help Command is executed

Sequence Diagram for executing the Help Command

The sequence diagrams for the `help` command are shown below.

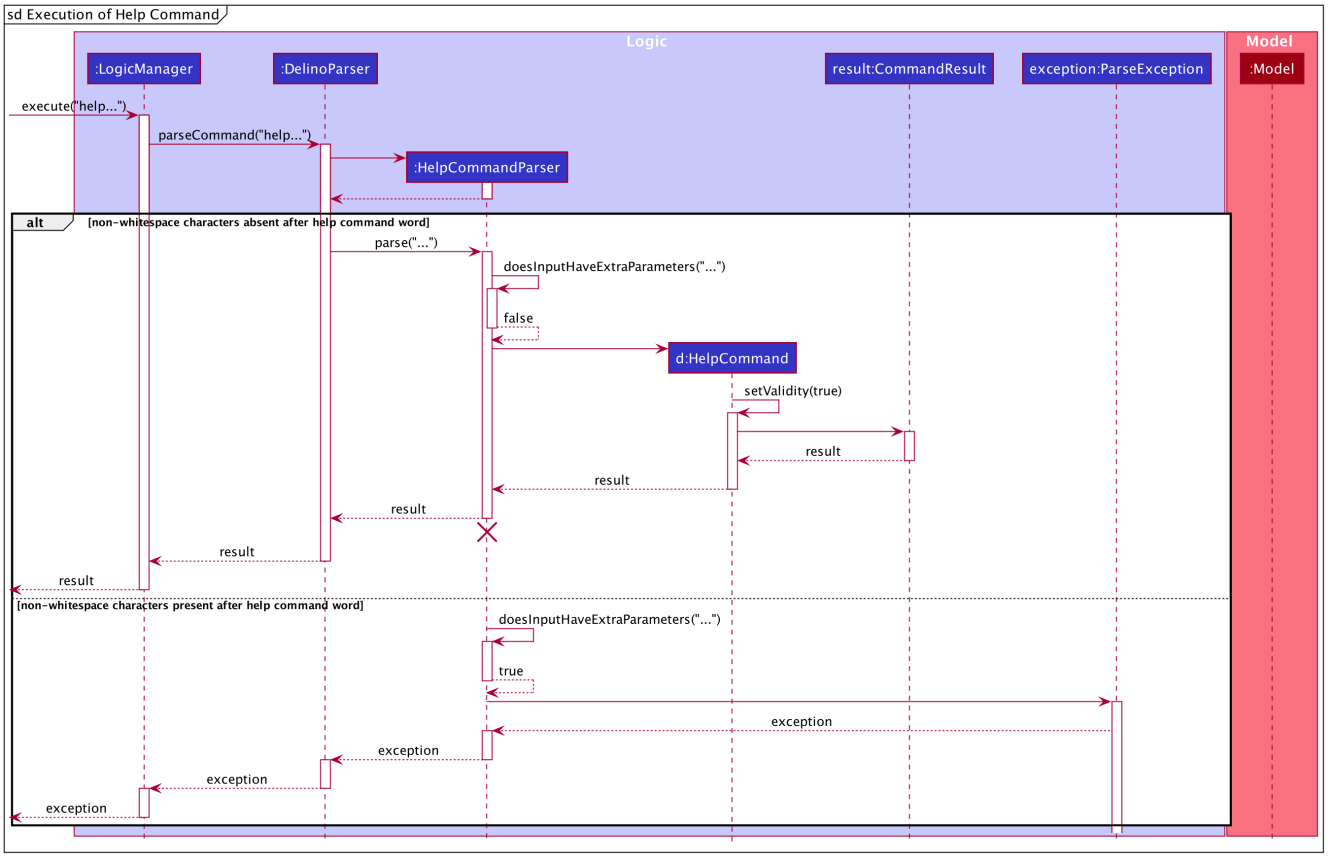


Figure 10. Help Command Sequence Diagram

The arguments typed into Delino by the user will first be done by the `execute` method in **LogicManager**. After which, an **DelinoParser** object will be created to parse the input which is determined by the command word via the parseCommand method. In this case, it is the `return` command word that will be parsed.

Then, a **HelpCommandParser** object will be created to parse the arguments after removing the command word `help` from the user's input. Based on the command word `help`, a **HelpCommand** object will be created. The parse() method in **HelpCommand** will check the validity of the user's input to see if there are any non-whitespace characters following the `help` command word.

Subsequently, the parseCommand method in **LogicManager** will continue to create a **CommandResult** based on the validity of the user's input.

If the user input is invalid, i.e. there are non-whitespace characters after the `help` command word, a **ParseException** object will be created in the `parse` method in **HelpCommandParser** and an error message will be displayed to the user.

If the user input is valid, i.e. there are no non-whitespace characters after the `help` command word, the `parse` method of **HelpCommandParser** will return a new **HelpCommand**.

Then, a new **CommandResult** will be created based on the user input. This will then display the success message to the user.

Delivered Feature

In this section, the functionality of the `delivered` feature, the expected execution path, the structure

of the **DeliveredCommand** class and the interactions between objects with the **DeliveredCommand** will be discussed.

What is the Delivered feature

The `delivered` function allows the user to mark orders or return orders as delivered after delivering an order or a return order.

The `delivered` feature was implemented as the **DeliveredCommand** in the logic package.
The `delivered` function requires a valid `FLAG` and a valid `INDEX`.
i.e. `delivered INDEX FLAG`

The `FLAG` can either be '-o' or '-r', which indicates which list (order list or return order list respectively) to mark the parcel from. The `FLAG` is only valid when either '-o' and '-r' is used. All other inputs will be regarded as invalid.

The `INDEX` is a positive integer that determines which order or return order to be marked as delivered. The `INDEX` is only valid if it is a positive integer and if it is not bigger than the size of the order list or return order list, depending on the `FLAG`that is provided. For instance, if the '-o' `FLAG` is provided, the `INDEX` should not be greater than the size of the order list.

Execution Paths of Delivered Command

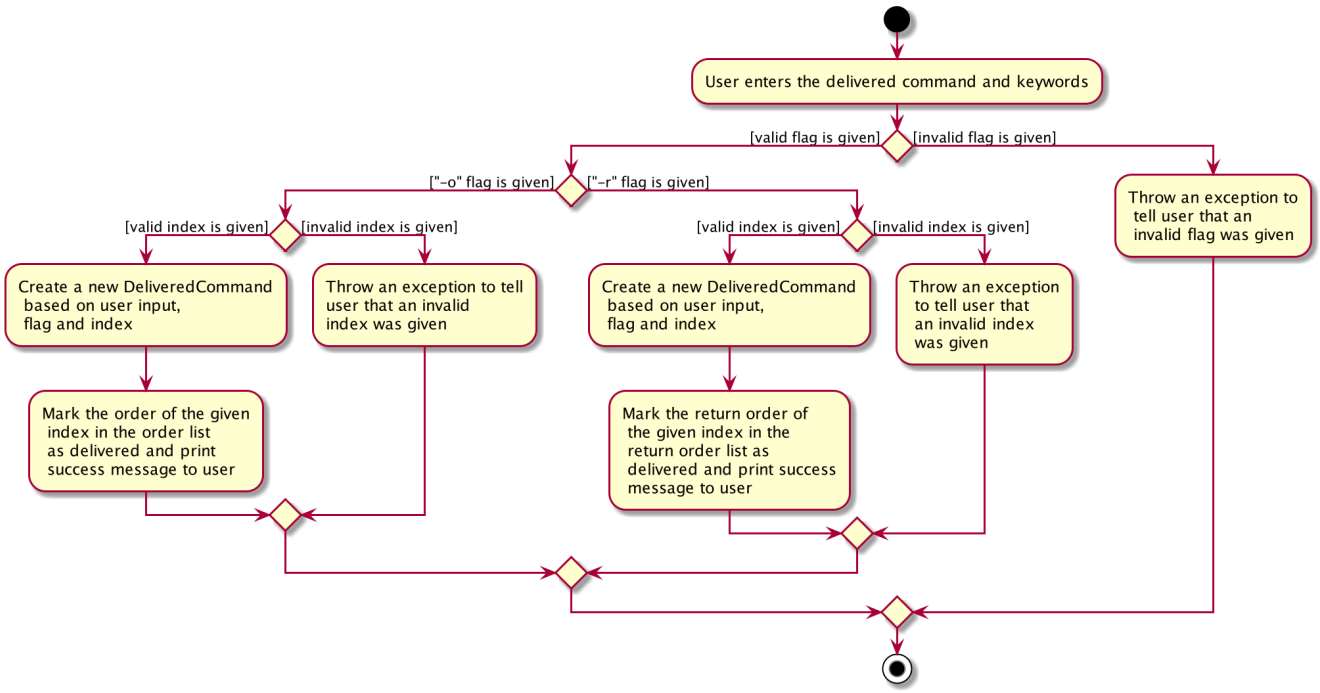
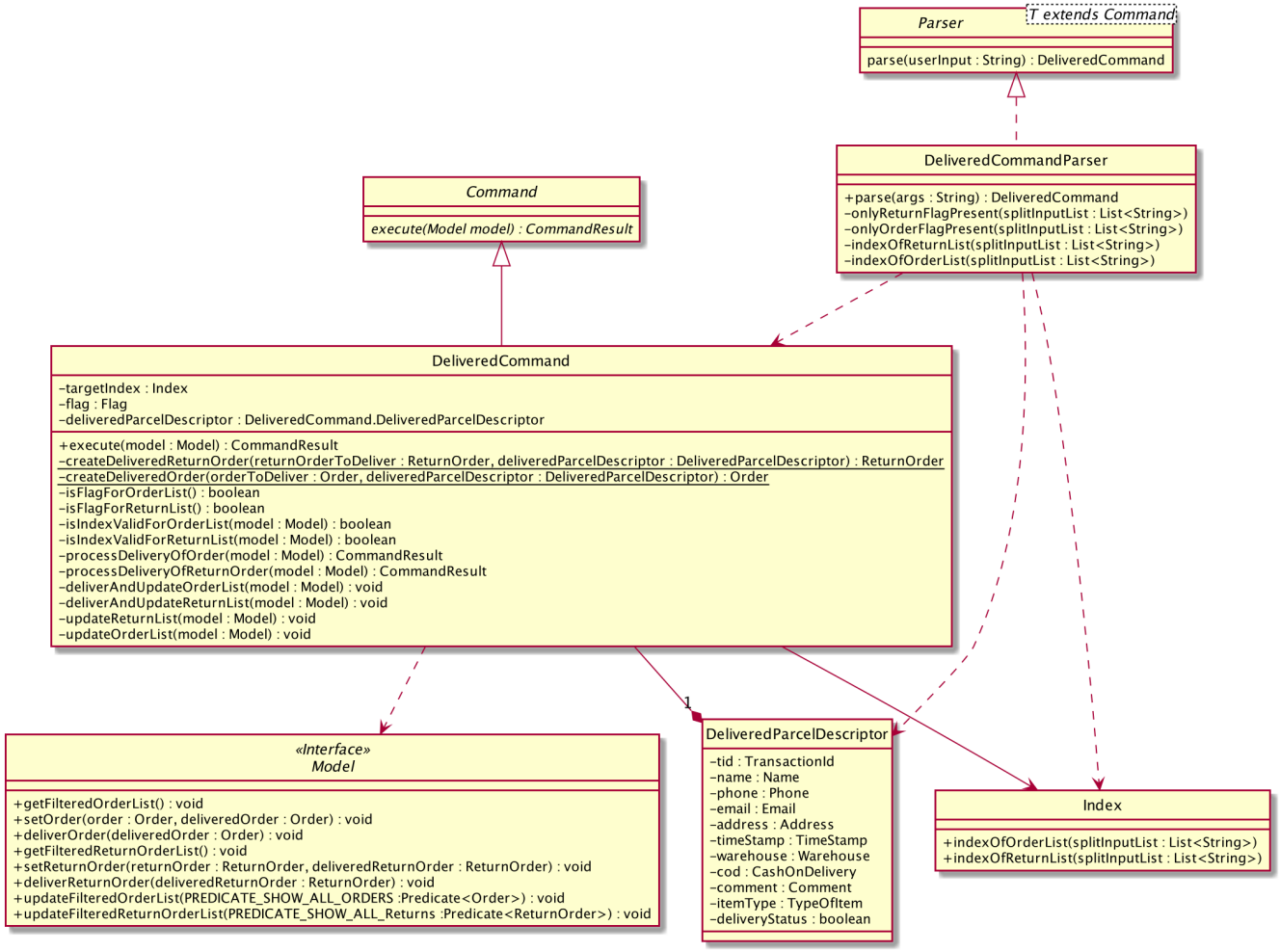


Figure 11. Activity Diagram of the Delivered Command

The above activity diagram shows the logic behind the **DeliveredCommand** which is determined in the **DeliveredCommandParser** class when the user inputs the command word `delivered` to activate the `delivered` feature.

Structure of Delivered Command



The above class diagram shows the structure of the **DeliveredCommand** and its associated classes and interfaces. Some methods and fields are not included because they are not extensively utilised in **DeliveredCommand**; such as public static fields and getter/setter methods.

Interactions between Delivered command and its associated objects

The sequence diagrams for the `delivered` command are shown below.

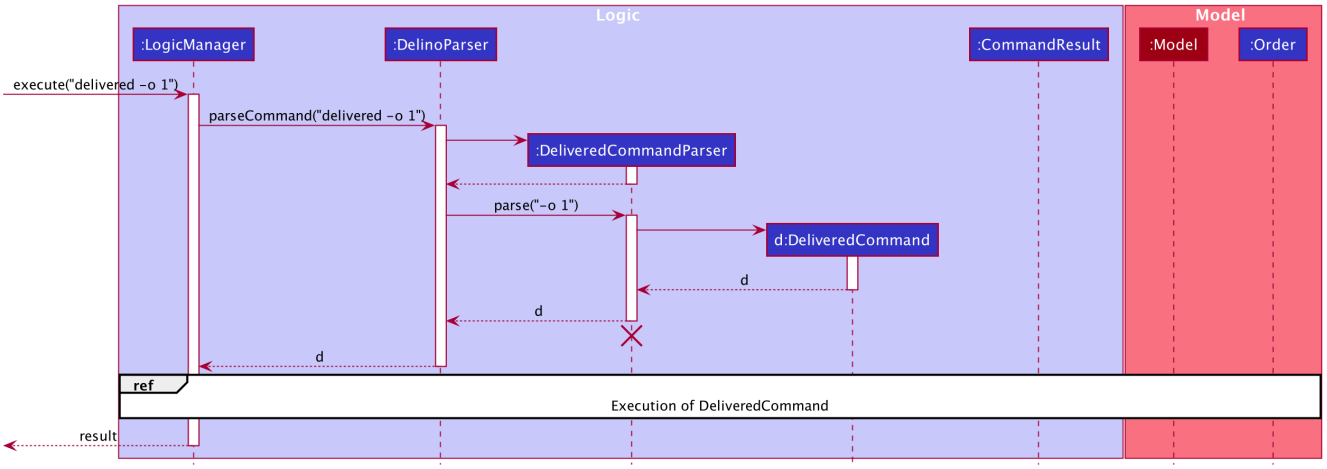


Figure 12. Delivered Command Sequence Diagram

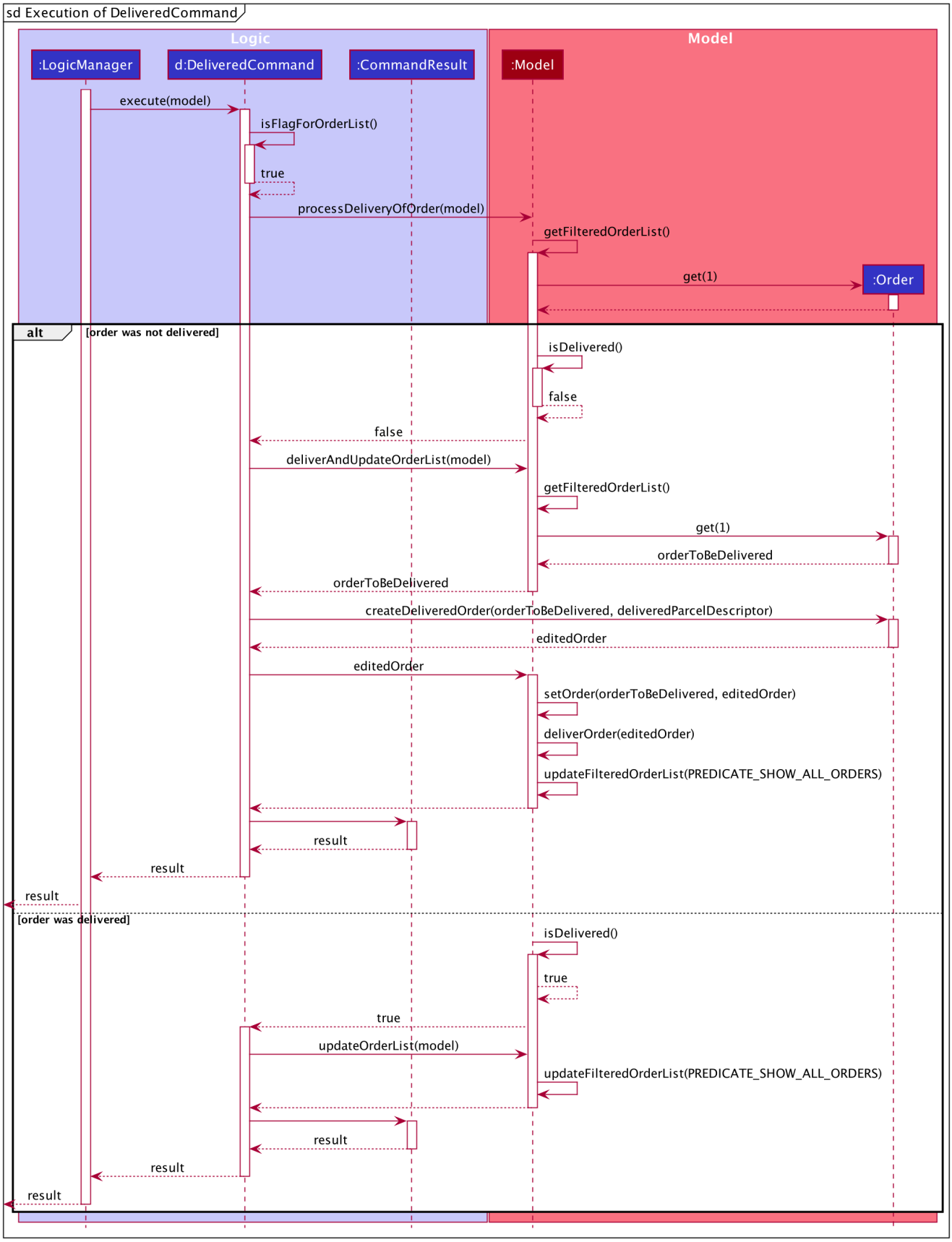


Figure 13. Execution of Delivered Command

The arguments typed into Delino by the user will first be done by the `execute` method in `LogicManager`. After which, an `DelinoParser` object will be created to parse the input which is determined by the command word via the parseCommand method. In this case, it is the `delivered` command word that will be parsed.

Then, a **DeliveredCommandParser** object will be created to parse the arguments after removing the command word `delivered` from the user's input. Based on the command word `delivered`, a **DeliveredCommand** object will be created.

Subsequently, the parseCommand method in **LogicManager** will continue to create a **CommandResult** based on the validity of the user's input; which is determined by the `execute` method in **DeliveredCommand**.

The `execute` method of **DeliveredCommand** will first check if a valid `FLAG` is present in the user's input. If the `FLAG` is not valid, a **CommandException** will be thrown to the user to tell him/her that their input was invalid and tell them the format which their input should follow.

If a valid `FLAG` is present, this will trigger the processDeliveryOfOrder method in **DeliveredCommand** which will check if a valid `INDEX` is present in the user's input.

If the `INDEX` is not valid, processDeliveryOfOrder method will throw a **CommandException** to the user; telling him/her that their input was invalid and the format that their input should follow. i.e. `delivered FLAG INDEX`

If both `FLAG` and `INDEX` are valid, an **Order** or **ReturnOrder** object will be created based on the `FLAG`. The `INDEX` will determine which order or return order to take from the order list or return order list respectively using the appropriate getter method. The `Order` or `ReturnOrder` object will be checked to see if it was delivered using the `checkIfOrderWasDelivered(model)` method.

If the **Order** or **ReturnOrder** was already delivered, this will call the updateOrderList(model) or updatedReturnOrderList(model) method respectively in **DeliveredCommand** and a new instance of **CommandResult** will be created to tell the user that the order or return order was delivered.

If the **Order** or **ReturnOrder** was not delivered, this will call the deliverAndUpdateOrderList(model) or deliverAndUpdateReturnOrderList(model) respectively in **DeliveredCommand**. In these methods, the particular **Order** or **ReturnOrder** will be retrieved from the model using the getFilteredOrderList() or getFilteredReturnOrderList() method. Based on the retrieved **Order** or **ReturnOrder**, a new **Order** or **ReturnOrder** with the `delivered` delivery status will be instantiated using the createDeliveredOrder or createDeliveredReturnOrder methods respectively.

Then, the setOrder or setReturnOrder method will be called to replace the original **Order** or **ReturnOrder** object respectively in model. The deliverOrder or deliverReturnOrder method will be called to to set the delivery status of the object to delivered. Then, the updateFilteredOrderList() method or updateFilteredReturnOrderList() method to update the list in the model.

Based on the new updates, a new **CommandResult** object will be instantiated to print the message success to the user.

Use Cases

Use case: UC08 - Request for help

MSS

1. User requests to list all commands in Delino
2. Delino opens a new window after execution of the help command.
3. Delino display the list of commands and a button to provide user a link to Delino's User Guide

   Use case ends.

Extensions

1a. Delino detects invalid syntax.

   1a1. Delino shows an error message to tell user the right way to use the help command.

   Use case ends.

1b. Delino detects additional non-whitespace characters after the command word, help.

   1b1. Delino shows an error message to tell user the right way to use the help command.

   Use case ends.

Use case: UC09 - Importing order details

MSS

1. User wants to import a specific orders from an external file.
2. User requests to import orders from an external file.
3. Delino checks for file existence.
4. Delino imports all orders from the external file.
5. Delino displays all orders imported.

   Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

   2a1. Delino shows an error message.

   Use case ends.

3a. Delino detects invalid file path.

   3a1. Delino shows the invalid file path error message

   Use case ends.

4a. Delino is unable to open the file.

   4a1. Delino shows permission denied error message.

Use case ends.

Use case: UC10 - Listing all orders

MSS

1. User wants to view all orders.
2. User requests to view the list of orders.
3. Delino display list of orders.

Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

2a1. Delino shows an error message.

Use case ends.

3a. Delino detects no orders.

3a1. Delino shows empty order list message.

Use case ends.

Use case: UC11 - Returning an order

MSS

1. User wants to return an order or create a new return order.
2. User requests to return an order or create a new return order.
3. Either an order will be converted to a return order or a new return order will be created
4. Delino displays the updated return order list with the new return order.

Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

2a1. Delino shows an error message.

Use case ends.

2b. Delino detects invalid parcel attributes.

2b1. Delino shows an error message to the user.

Use case ends.

2c. Delino detects invalid `TRANSACTION_ID`, i.e. order with the given `TRANSACTION_ID` does not exist.

2c1. Delino shows an error message to the user.

Use case ends.

2d. Delino detects invalid `RETURN_TIMESTAMP`, i.e. order with the given `RETURN_TIMESTAMP` is bfeore the delivery time stamp of the order.

2d1. Delino shows an error message to the user.

Use case ends.

2e. Delino detects missing parcel attributes.

2e1. Delino shows an error message to the user.

Use case ends.

Use case: UC12 - Obtain orders in a postal sector

MSS

1. User wants to obtain all orders in a specified postal sector.
2. User requests to obtain orders in a specified postal sector
3. Delino obtains all orders located in the postal sector
4. Delino display the list of orders

Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

2a1. Delino shows an error message.

Use case ends.

3a. Delino detects no orders.

3a1. Delino shows empty order list message.

Use case ends.

Use case: UC13 - Show statistics

MSS

1. User requests to see the statistics of orders.
2. Delino opens a window that contains the statistics.

   Use case ends.

Appendix A: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 350 orders without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. The system should be able to respond within three seconds.
5. The system should work without internet access.
6. A user should be able to get all the information he/she needs within four commands.
7. A user should be able to familiarise himself/herself within an hour of usage.
8. The data cannot be stored in a Database Management System (DBMS).
9. The system should work once downloaded and should not depend on a remote server.
10. The application should not be larger than 50Mb.
11. The application should be crash during its execution. It should show warning messages instead of crashing.

Appendix B: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Returns

An order that is rejected and needs to be returned to the warehouse

Invalid syntax

Any syntax used that does not correspond to the required format

Status Bar

Refers to the display field showing the results of an executed command

Table 4. Command Prefix

Prefix	Meaning	Used in the following Command(s)
ot/	Order Type	Import

Prefix	Meaning	Used in the following Command(s)
tid/	Transaction ID	Edit, Import, Insert, Return, Search
n/	Customer Name	Edit, Import, Insert, Return, Search
a/	Address	Edit, Import, Insert, Return, Search
p/	Phone Number	Edit, Import, Insert, Return, Search
e/	Email	Import, Insert, Edit, Return, Search
dts/	Delivery Date And Time	Edit, Import, Insert, Return, Search
rts/	Return Date and Time	Import, Return, Search
w/	Warehouse Location	Edit, Import, Insert, Return, Search
cod/	Cash On Delivery	Edit, Insert, Search
c/	Comments by Customer	Edit, Import, Insert, Return, Search
type/	Type of Item	Edit, Import, Insert, Return, Search

Table 5. Possible Command Flags

Flag	Meaning	Used in the following Command(s)
-f	Force clear, no user confirmation will be requested	Clear
-o	Order flag, Operation on order list	Clear, Delete, Delivered, Edit, Nearby, Search
-r	Return Order flag, Operation on return order list	Clear, Delete, Delivered, Edit, Nearby, Search

Appendix C: Product Survey

PingLocker

Pros:

- You can take a picture of the parcel and it is automatically added to the application for tracking.
- You can also receive weekly statistics on your delivery tasks and time saved.

- The application interface looks good and easy to navigate.

Cons:

- Required to scan the parcels one by one, which can be slow and tedious.
- There does not seem to be a feature to keep track of parcels in a particular area/region.

Pickupp

Pros:

- You can take a picture of the parcel barcode to add to the application for tracking.
- There is language support for chinese language (in addition to english)

Cons:

- There are no statistics available.
- Required to scan the parcels one by one, which can be slow and tedious.

Appendix D: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

Launch Delino

1. Initial launch
   a. Ensure that you have Java 11 installed in your computer
   b. Download the latest Delino.jar
   c. Copy the jar file to the folder you would like to use as a home address for Delino application
   Expected: Shows the GUI of the Delino App. The window size may not be optimum

Inserting an order

1. Insert a minimum of 2 orders
   a. Insert command format: `insert tid/TRANSACTION_ID n/CUSTOMER_NAME a/ADDRESS p/PHONE_NUMBER e/EMAIL ts/DELIVERY_DATE_&_TIME w/WAREHOUSE_LOCATION cod/CASH_ON_DELIVERY [c/COMMENTS_BY_CUSTOMER] [type/TYPE_OF_ITEM]`
   b. Test case: `insert tid/9876543210 n/John Doe a/Blk 572 Hougang st 51 #10-33 S530572 p/98766789 e/johndoe@example.com ts/2020-02-20 1300 w/Yishun cod/$4`
   Expected: Inserts an order with the above details to the list and displayed on the GUI
   c. Test case: `insert tid/1023456789 n/Amos Cheong a/Blk 572 Hougang st 51 #11-37 S530572`

`p/9001 0019 e/amoscheong@example.com ts/2020-03-10 1650 w/Marsiling cod/$5 c/Leave it at the riser type/glass`
Expected: Inserts the order to the list, including the item type and the order comment

d. Test case: Invalid Syntax
Expected: No order is added. Error details shown in the response message. A help message displayed for user to insert accordingly. Status bar remain unchanged

e. Test case: Insert order with existing Transaction ID in list
Expected: An error will occur and a message will be displayed, stating that order with duplicate ID cannot be inserted into the list

Returning an order

1. Convert a delivered order into a return order or create a new return order

   a. Return command format: `return tid/TRANSACTION_ID n/CUSTOMER_NAME a/ADDRESS p/PHONE_NUMBER e/EMAIL rts/RETURN_TIMESTAMP w/WAREHOUSE_LOCATION [c/COMMENTS_BY_CUSTOMER] [type/TYPE_OF_ITEM]`
   OR `return tid/TRANSACTION_ID rts/RETURN_TIMESTAMP`

   b. Test case: `return tid/9876543210 n/John Doe a/Blk 572 Hougang st 51 #10-33 S530572 p/98766789 e/johndoe@example.com rts/2020-02-20 1300 w/Yishun`
   Expected: Creates a new return order with the above details and adds it into the return order list. This will be displayed on the GUI

   c. Test case: `return tid/1023456789 n/Amos Cheong a/Blk 572 Hougang st 51 #11-37 S530572 p/9001 0019 e/amoscheong@example.com rts/2020-03-10 1650 w/Marsiling c/Leave it at the riser type/glass`
   Expected: Creates a new return order with the above details, including type and comments and adds it into the return order list. This will be displayed on the GUI

   d. Test case: `return tid/1023456789 rts/2020-05-12 1300`
   Expected: Checks if an order with the given Transaction ID exists. If it exists and it was delivered, check if the given return time stamp is after the delivery time stamp. If it is both delivered and the return time stamp is after the delivery time stamp, the order will be converted into a return order and added into the return order list.

   e. Test case: Invalid Syntax
   Expected: No return order is added. Error details shown in the response message. A help message displayed for user to use the return command accordingly.

   f. Test case: Return order with invalid Transaction ID in order list
   Expected: An error will occur and a message will be displayed, stating that order with the given Transaction ID.

   g. Test case: Return order with invalid return time stamp in order list
   Expected: An error will occur and a message will be displayed, stating that order with the given Transaction ID. cannot be found in the order list.

   h. Test case: Return order with missing parcel attributes
   Expected: An error will occur and a message will be displayed, indicating the right message usage of the return feature.

i. Test case: Order to be converted was not delivered
Expected: An error will occur and a message will be displayed, indicating that the order was not delivered and cannot be returned.

Deleting an order

1. Deleting an order with respect to the current list displayed
    a. Delete command format: `delete FLAG INDEX`
    b. Prerequisites: List all orders using the `list` command. Multiple orders in the list
    c. Test case: `delete -o 1`
    Expected: The first order item in the current order list will be removed. Details of the deleted order will be displayed in the response box
    d. Test case: `delete -r 2`
    Expected: The second item in the current return order list will be removed. Details of the deleted order will be displayed in the response box
    e. Test case: `delete 20`
    Expected: No order is deleted as no `FLAG` is provided.
    An error message will be displayed in the response box.
    f. Test case: `delete -r INVALID_INDEX`
    Expected: No order is deleted. An error message will be displayed in the response box, indicating that the index cannot be found in the list

Edit orders

1. Edit the details of the delivery order by specifying the `FLAG` (order type), `INDEX` (parcel number displayed), `ORDER_ATTRIBUTE_PREFIX` (the field user want to change) and `NEW_VALUE` (the new value that user want to replace the old ones with). For detailed implementation explanation click here.
    a. Prerequisite: Call the `list` command to show something or `insert`/`return` command to add something before you can `edit`.
    b. Edit command format: `edit FLAG INDEX ORDER_ATTRIBUTE_PREFIX/NEW_VALUE`
    c. Test case: `edit -r 1 n/Xuan En`
    Expected: The first index customer's name is changed to Xuan En.
    d. Test case: `edit -o 2 p/99521654`
    Expected: The second index phone number is changed to 9952 1654.
    e. Test case: `edit -o 1 a/Blk 123 Pasir Ris street 51 #12-23 S510123`
    Expected: The first index is edited where the address of the customer of the order will be changed to Blk 123 Pasir Ris Street 51 #12-23 S510123.
    f. Test case: `edit -r 2 n/Mr Tan p/98776655 a/Blk 888 Jurong East street 2 #01-02 S609601`
    Expected: The second index of the list is edited. The name is changed to Mr Tan, phone number changed to 98776655 and address will be changed to Blk 888 Jurong East street 2 #01-02 S609601.

g. Test case: `edit -o 1 dts/09/08/2020`
Expected: The delivery date of the first index of the customer will be rescheduled to 09/08/2020.

h. Test case: `edit -r 1 rts/02/02/1900`
Expected: The response box will display an error message as it is impossible to put a date that is already passed.

Import orders

1. Import a new list of orders from a .csv file given by the company

   a. Import command format: `import NAME_OF_FILE.csv`

   b. Prerequisites : The import file must be a `.csv file` and the `csv file` should be inside `data` folder which is the same directory as the JAR file. Otherwise, it will cause the app to raise an exception and print the error message. Should not import a file that is non-existent

   c. Test case: `import customers_20_02_2020.csv`
   Expected: In the response box, a message will appear to indicate that the import is successful. At the same time, the contents of the .csv file will be shown to the user in the form of a list of orders

List orders

1. List all the delivery orders for the user. The type of orders to be listed is dependent on the command input from the user

   a. Test case: `list`
   Expected: List all the delivery orders, showing all completed and uncompleted orders.

   b. Test case: `list done`
   Expected: List all completed delivery orders.

   c. Test case: `list undone`
   Expected: List all uncompleted delivery orders.

   d. Test case: `list ANY_WORD_OTHER_THAN_UNDONE_AND_DONE`
   Expected: An error will occur, a message will appear in the response box, indicating an invalid list command

Search an order

1. Search an order based on the `KEYWORD` and `FLAG`(if any) given. For a more detailed explanation of the implementation click here.

   a. Search command format: `search FLAG KEYWORD`

   b. Test case: `list done`
   `search -r Jeremy Loh`
   Expected: Specifically search for any return order from the return order list that has any field matching the keyword of either `Jeremy`, `Loh` or both and print it to the user.

c. Test case: `list undone`
`search -o tid/asj2od3943`
Expected: Specifically search for any order from the order list that has a Transaction id of `asj2od3943` and print it out to the user.

d. Test case: `list`
`search n/Jeremy`
Expected: Specifically search for both return order or order from both list that has one word of the name matching the key word `Jeremy`.

e. Test case: `list`
`search`
Expected: An error message will appear in the response box, stating that the argument cannot be empty and there are no changes to the list itself.

Mark parcel as delivered

1. Mark a parcel's delivery status as delivered based on its `INDEX` and `FLAG` whenever the parcel has been delivered.

   a. Delivered command format: `delivered FLAG INDEX`

   b. Prerequisite: Ensure that your order list or return order list has at least one order.

   c. Test case: `delivered -o 1`
   Expected: The first order in the currently displayed order list will be marked as delivered

   d. Test case: `delivered -r 2`
   Expected: The second return order in the currently displayed return order list will be marked as delivered

   e. Test case: Invalid syntax
   Expected: No parcel is marked as delivered. The error message will be displayed on the error response box describing the error

Clear orders

1. Clear all orders while all orders are listed

   a. Clear command format: `clear [FLAG]`

   b. Test case: `clear`
   Expected: Confirmation message will display in status message.

      i. If **Yes** button is pressed, the both order and return order lists will be cleared. Notify the user that both order lists have been cleared in the status message.

      ii. If **No** button is pressed, no order list is cleared.

   c. Test case: `clear -f`
   Expected: Both order list and return order list will be cleared. Notify the user that both order lists have been cleared in the status message.

   d. Test case: `clear -f -r`
   Expected: Only return order list will be cleared. Notify the user that return order list has

been cleared in the status message.

e. Test case: `clear -r -r -f`
Expected: Only return order list will be cleared. Notify the user that return order list has been cleared in the status message.

f. Test case: `clear -r -o`
Expected: Invalid command input, as both `-r` and `-o` cannot be in a single command.
Error details shown in the response message. A help message displayed for the user to type the correct command. Status bar remains unchanged

g. Test case: `clear -r-f`
Expected: Invalid command input, as space is required in between flags.
Error details shown in the response message. A help message displayed for the user to type the correct command. Status bar remains unchanged

Help

1. Display a list of available commands to user

   a. Test case: `help`
   Expected: A list of commands will be displayed in a new pop-up window and the response box will indicate a successful command.

   b. Test case: Invalid syntax
   Expected: An error will occur and the response box will show an error message

Show

1. Opens a window which shows the statistics of the current lists of orders. It displays information such as earnings made, orders delivered and orders returned (Including a PieChart).

   a. Test case: `show all`
   Expected: All statistical information of all the orders shown in the statistics tab.

   b. Test case: `show today`
   Expected: All statistical information today shown in the statistics tab.

   c. Test case: `show today 2020-12-03`
   Expected: All statistical information between the dates shown in the statistics tab.

   d. Test case: `show 2020-12-03`
   Expected: All statistical information for the given date shown in the statistics tab.

   e. Test case: `show 2020-12-03 2021-01-01`
   Expected: All statistical information within the dates shown in the statistics tab.

   f. Test case: Invalid syntax
   Expected: An error will occur and the response box will show an error message.

Obtain all nearby orders based on area or postal sector

There are two possible search criteria for nearby orders

1. Obtain all orders located in the same postal sector.
   The postal sector to search for is given by the user.
   A **postal sector** is the first **two** digits of a six digit Singapore postal code.
   The list of postal sectors and their corresponding general locations can be found here.
   a. Nearby command format: `nearby [FLAG] POSTAL_SECTOR`
   b. Prerequisites: Should call a list command before calling nearby. The nearby command will search based on the current list
   c. Test case: `nearby -o 79`
      Expected: Obtain all orders located in postal sector 79 (Seletar)
   d. Test case: `nearby -r 07`
      Expected: Obtain all return orders located in postal sector 07 (Anson, Tanjong Pagar)
   e. Test case: `nearby -o 99`
      Expected: An error will occur as the given postal sector is invalid
   f. Test case: `nearby -o 600`
      Expected: An error will occur as the given postal sector is invalid
   g. Test case: `nearby -o`
      Expected: An error will occur as it is an invalid syntax (no postal sector is provided)
2. Obtain all orders located in the same area.
   There are **5** different areas in Singapore: Central, East, North East, West, North
   a. Nearby command format: `nearby [FLAG] AREA`
   b. Prerequisites: Should call a list command before calling nearby. The nearby command will search based on the current list
   c. Test case: `nearby -o central`
      Expected: Obtain all orders located in central area of Singapore
   d. Test case: `nearby -r east`
      Expected: Obtain all return orders located in east area of Singapore
   e. Test case: `nearby -o north east`
      Expected: Obtain all orders located in east area of Singapore
   f. Test case: `nearby -r west`
      Expected: Obtain all return orders located in east area of Singapore
   g. Test case: `nearby -r north`
      Expected: Obtain all return orders located in east area of Singapore
   h. Information about each area was obtained from this website

Saving data

1. Manual Saving is not required as data is already saved in the hard disk after any commands that change the data

Exit

1. Exits the Delino App using the `exit` command
    a. Test case: `exit`
    Expected: The GUI window will be closed
    b. Test case: Adding any other words as the second argument of the `exit` command Expected: The response box will display an invalid command message

Use case: UC11 - Returning an order

MSS

1. User wants to return an order or create a new return order.
2. User requests to return an order or create a new return order.
3. Either an order will be converted to a return order or a new return order will be created
4. Delino displays the updated return order list with the new return order.

Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

2a1. Delino shows an error message.

Use case ends.

2b. Delino detects invalid parcel attributes.

2b1. Delino shows an error message to the user.

Use case ends.

2c. Delino detects invalid `TRANSACTION_ID`, i.e. order with the given `TRANSACTION_ID` does not exist.

2c1. Delino shows an error message to the user.

Use case ends.

2d. Delino detects invalid `RETURN_TIMESTAMP`, i.e. order with the given `RETURN_TIMESTAMP` is bfeore the delivery time stamp of the order.

2d1. Delino shows an error message to the user.

Use case ends.

2e. Delino detects missing parcel attributes.

2e1. Delino shows an error message to the user.

Use case ends.

Use case: UC04 - Mark order or return order as delivered

MSS

1. User wants to mark an order or return order as delivered.
2. User request to mark order or return order as delivered.
3. Delino changes the delivery status of the specified order or return order to delivered.
4. Delino will display an updated order list or return order list.

Use case ends.

Extensions

2a. Delino detects invalid syntax from user input.

2a1. Delino shows an error message.

Use case ends.

2b. Delino unable to detect any parcel with the `INDEX` provided.

2b1. Delino shows error message to the user.

Use case ends.

2c. Delino unable to detect valid `INDEX` provided.

2c1. Delino shows error message to the user.

Use case ends.

2d. Delino unable to detect valid `FLAG` provided.

2d1. Delino shows error message to the user.

Use case ends.

Glossary

Table 6. Command Prefix

Prefix	Meaning	Used in the following Command(s)
ot/	Order Type	Import
tid/	Transaction ID	Edit, Import, Insert, Return, Search
n/	Customer Name	Edit, Import, Insert, Return, Search
a/	Address	Edit, Import, Insert, Return, Search
p/	Phone Number	Edit, Import, Insert, Return, Search
e/	Email	Import, Insert, Edit, Return, Search
dts/	Delivery Date And Time	Edit, Import, Insert, Return, Search
rts/	Return Date and Time	Import, Return, Search
w/	Warehouse Location	Edit, Import, Insert, Return, Search
cod/	Cash On Delivery	Edit, Insert, Search
c/	Comments by Customer	Edit, Import, Insert, Return, Search
type/	Type of Item	Edit, Import, Insert, Return, Search

Table 7. Possible Command Flags

Flag	Meaning	Used in the following Command(s)
-f	Force clear, no user confirmation will be requested	Clear
-o	Order flag, Operation on order list	Clear, Delete, Delivered, Edit, Nearby, Search
-r	Return Order flag, Operation on return order list	Clear, Delete, Delivered, Edit, Nearby, Search

Appendix G: Instructions for Manual Testing

Mark parcel as delivered

1. Mark a parcel's delivery status as delivered based on its `INDEX` and `FLAG` whenever the parcel has been delivered.
   a. Delivered command format: `delivered FLAG INDEX`
   b. Prerequisite: Ensure that your order list or return order list has at least one order.
   c. Test case: `delivered -o 1`
   Expected: The first order in the currently displayed order list will be marked as delivered
   d. Test case: `delivered -r 2`
   Expected: The second return order in the currently displayed return order list will be marked as delivered
   e. Test case: Invalid syntax
   Expected: No parcel is marked as delivered. The error message will be displayed on the error response box describing the error === Returning an order
2. Convert a delivered order into a return order or create a new return order
   a. Return command format: `return tid/TRANSACTION_ID n/CUSTOMER_NAME a/ADDRESS p/PHONE_NUMBER e/EMAIL rts/RETURN_TIMESTAMP w/WAREHOUSE_LOCATION [c/COMMENTS_BY_CUSTOMER] [type/TYPE_OF_ITEM]`
   OR `return tid/TRANSACTION_ID rts/RETURN_TIMESTAMP`
   b. Test case: `return tid/9876543210 n/John Doe a/Blk 572 Hougang st 51 #10-33 S530572 p/98766789 e/johndoe@example.com rts/2020-02-20 1300 w/Yishun`
   Expected: Creates a new return order with the above details and adds it into the return order list. This will be displayed on the GUI
   c. Test case: `return tid/1023456789 n/Amos Cheong a/Blk 572 Hougang st 51 #11-37 S530572 p/9001 0019 e/amoscheong@example.com rts/2020-03-10 1650 w/Marsiling c/Leave it at the riser type/glass`
   Expected: Creates a new return order with the above details, including type and comments and adds it into the return order list. This will be displayed on the GUI
   d. Test case: `return tid/1023456789 rts/2020-05-12 1300`
   Expected: Checks if an order with the given Transaction ID exists. If it exists and it was delivered, check if the given return time stamp is after the delivery time stamp. If it is both delivered and the return time stamp is after the delivery time stamp, the order will be converted into a return order and added into the return order list.
   e. Test case: Invalid Syntax
   Expected: No return order is added. Error details shown in the response message. A help message displayed for user to use the return command accordingly.
   f. Test case: Return order with invalid Transaction ID in order list
   Expected: An error will occur and a message will be displayed, stating that order with the given Transaction ID.
   g. Test case: Return order with invalid return time stamp in order list
   Expected: An error will occur and a message will be displayed, stating that order with the given Transaction ID. cannot be found in the order list.

h. Test case: Return order with missing parcel attributes
   Expected: An error will occur and a message will be displayed, indicating the right message usage of the return feature.

i. Test case: Order to be converted was not delivered
   Expected: An error will occur and a message will be displayed, indicating that the order was not delivered and cannot be returned. === Help

3. Display a list of available commands to user

   a. Test case: `help`
      Expected: A list of commands will be displayed in a new pop-up window and the response box will indicate a successful command.

   b. Test case: Invalid syntax
      Expected: An error will occur and the response box will show an error message