

Developer Guide

Table of Contents

1. **Acknowledgements**
2. **Setting up, getting started**
3. **Design**
 - 3.1 **Architecture**
 - 3.2 **UI component**
 - 3.3 **Logic component**
 - 3.4 **Model component**
 - 3.4.1 **AddressBook**
 - 3.4.1.1 **Person class diagram**
 - 3.4.2 **TransactionBook**
 - 3.4.2.1 **Transaction class diagram**
 - 3.5 **Storage component**
 - 3.6 **Common classes**
4. **Implementation**
 - 4.1 **[Proposed] Undo/redo feature**
 - 4.1.1 **Proposed Implementation**
 - 4.1.2 **Design considerations:**
 - 4.2 **[Proposed] Data import**
5. **Documentation, logging, testing, configuration, dev-ops**
6. **Appendix: Requirements**
 - 6.1 **Target User Profile**
 - 6.2 **User Stories**
 - 6.2.1 **Transaction Recording**
 - 6.2.2 **Dashboard Overview**
 - 6.2.3 **Financial Reporting**
 - 6.2.4 **Data Security and Backup**
 - 6.2.5 **Address Book**
 - 6.3 **Use Cases**
 - 6.3.1 **Use Case 1: Adding a Transaction**
 - 6.3.2 **Use Case 2: Removing a Transaction**
 - 6.3.3 **Use Case 3: Viewing All Transactions**

- 6.3.4 [Use Case 4: Editing a Transaction](#)
 - 6.3.5 [Use Case 5: Restoring Deleted Transactions](#)
 - 6.3.6 [Use Case 6: Dashboard Display](#)
 - 6.3.7 [Use Case 7: Access to Financial Reports](#)
 - 6.3.8 [Use Case 8: Customizable Reports](#)
 - 6.3.9 [Use Case 9: Deleting All Transactions](#)
 - 6.3.10 [Use Case 10: Adding a Staff Member to Address Book](#)
 - 6.3.11 [Use Case 11: Removing a Staff Member from Address Book](#)
 - 6.3.12 [Use Case 12: Editing Staff Member Information in Address Book](#)
 - 6.3.13 [Use Case 13: Deleting All Address Book Staff Members](#)
 - 6.3.14 [Use Case 14: Filtering Transactions](#)
 - 6.3.15 [Use Case 15: Sorting Transactions](#)
 - 6.4 [Non-functional Requirements \(NFR\)](#)
 - 6.5 [Glossary](#)
 - 6.6 [Appendix: Planned Enhancements](#)
 - 6.7 [Appendix: Instructions for manual testing](#)
 - 6.7.1 [Launch and shutdown](#)
 - 6.7.2 [Deleting a person](#)
 - 6.7.3 [Saving data](#)
-


Acknowledgements

- [Opencsv](#)
 - [fx-yearmonth-picker](#)
-

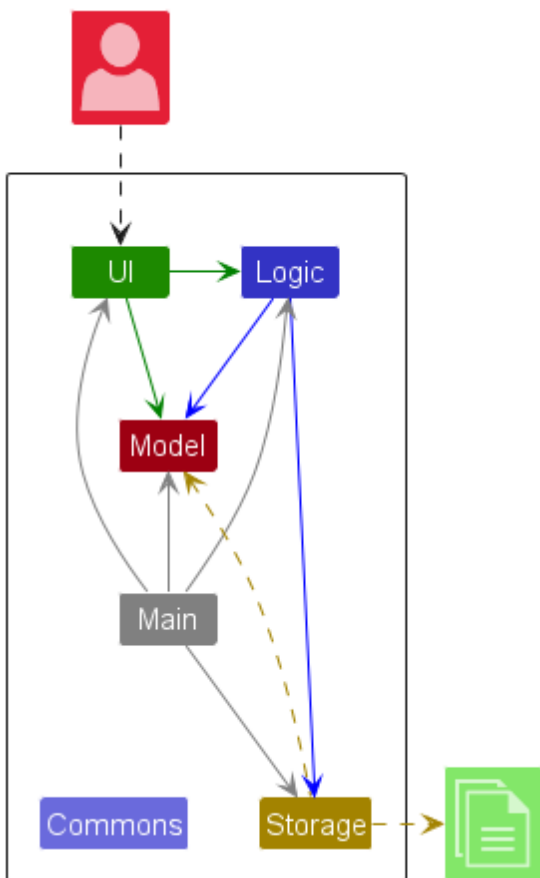
Setting up, getting started

Refer to the guide [Setting up and getting started](#).

Design

 **Tip:** The `.puml` files used to create diagrams in this document are in the `docs/diagrams` folder. Refer to the [PlantUML Tutorial](https://se-edu/guides/plantuml-tutorial) at se-edu/guides to learn how to create and edit diagrams.

Architecture



The **Architecture Diagram** given above explains the high-level design of the App.

Given below is a quick overview of main components and how they interact with each other.

Main components of the architecture

Main (consisting of classes `Main` and `MainApp`) is in charge of the app launch and shut down.

- At app launch, it initializes the other components in the correct sequence, and connects them up with each other.
- At shut down, it shuts down the other components and invokes cleanup methods where necessary.

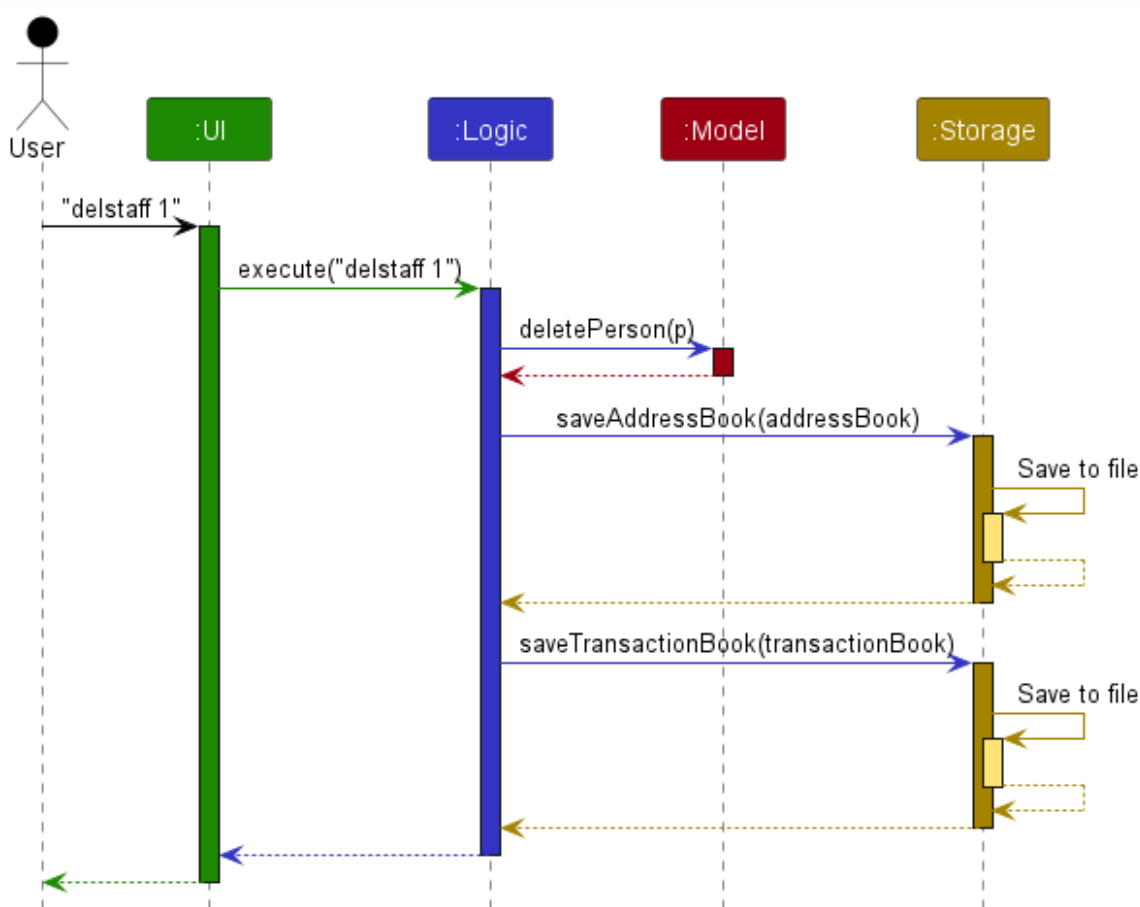
The bulk of the app's work is done by the following four components:

- **UI** : The UI of the App.
- **Logic** : The command executor.
- **Model** : Holds the data of the App in memory.
- **Storage** : Reads data from, and writes data to, the hard disk.

Commons represents a collection of classes used by multiple other components.

How the architecture components interact with each other

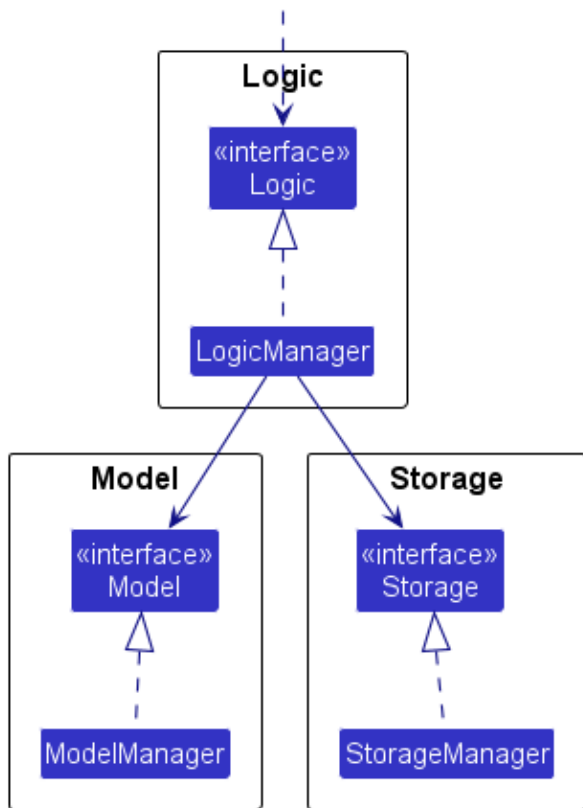
The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delstaff 1`. Note that both the address book and transaction book are saved.



Each of the four main components (also shown in the diagram above),

- defines its *API* in an **interface** with the same name as the Component.
- implements its functionality using a concrete **{Component Name}Manager** class (which follows the corresponding API **interface** mentioned in the previous point).

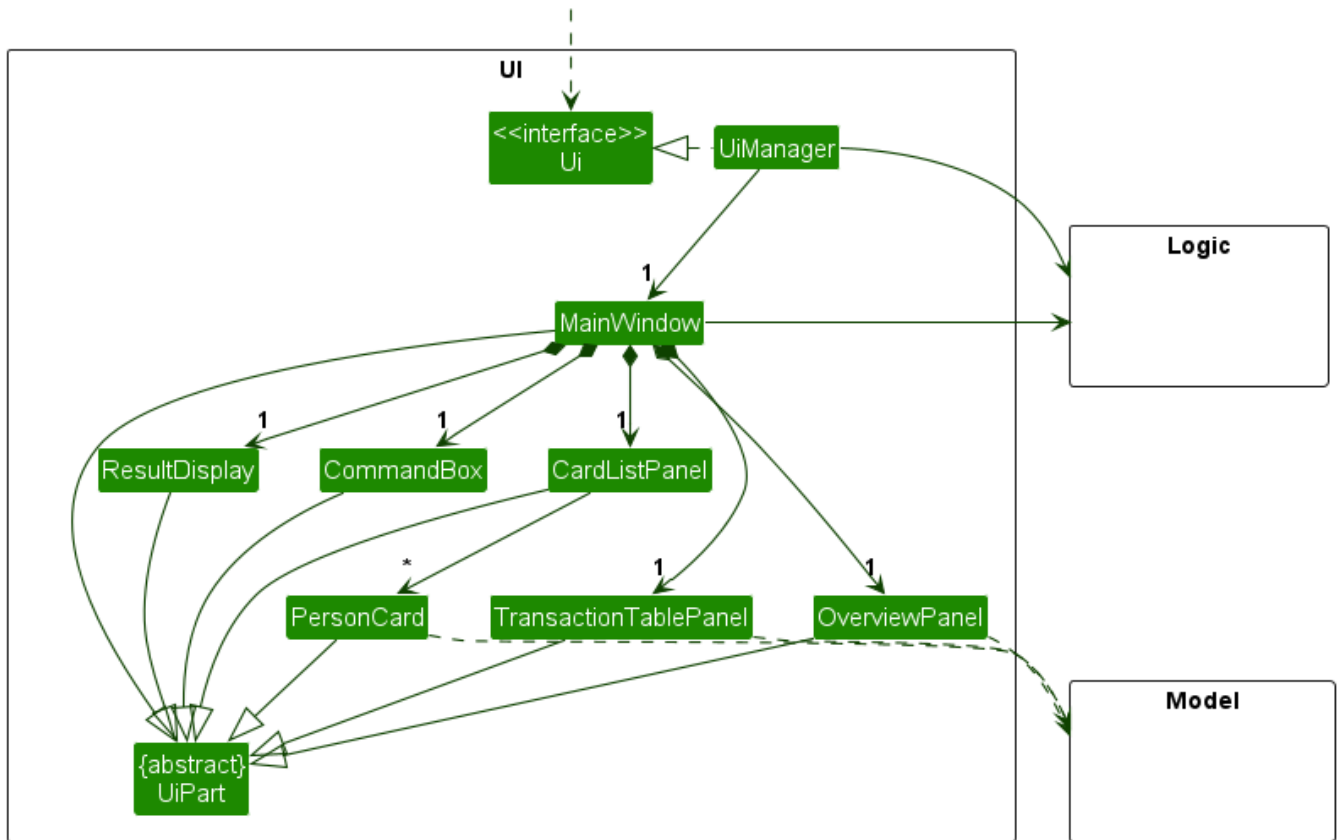
For example, the **Logic** component defines its API in the **Logic.java** interface and implements its functionality using the **LogicManager.java** class which follows the **Logic** interface. Other components interact with a given component through its interface rather than the concrete class (reason: to prevent outside component's being coupled to the implementation of a component), as illustrated in the (partial) class diagram below.



The sections below give more details of each component.

UI component

The **API** of this component is specified in `Ui.java`



The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `CardListPanel` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class which captures the commonalities between classes that represent parts of the visible GUI.

The `UI` component uses the JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

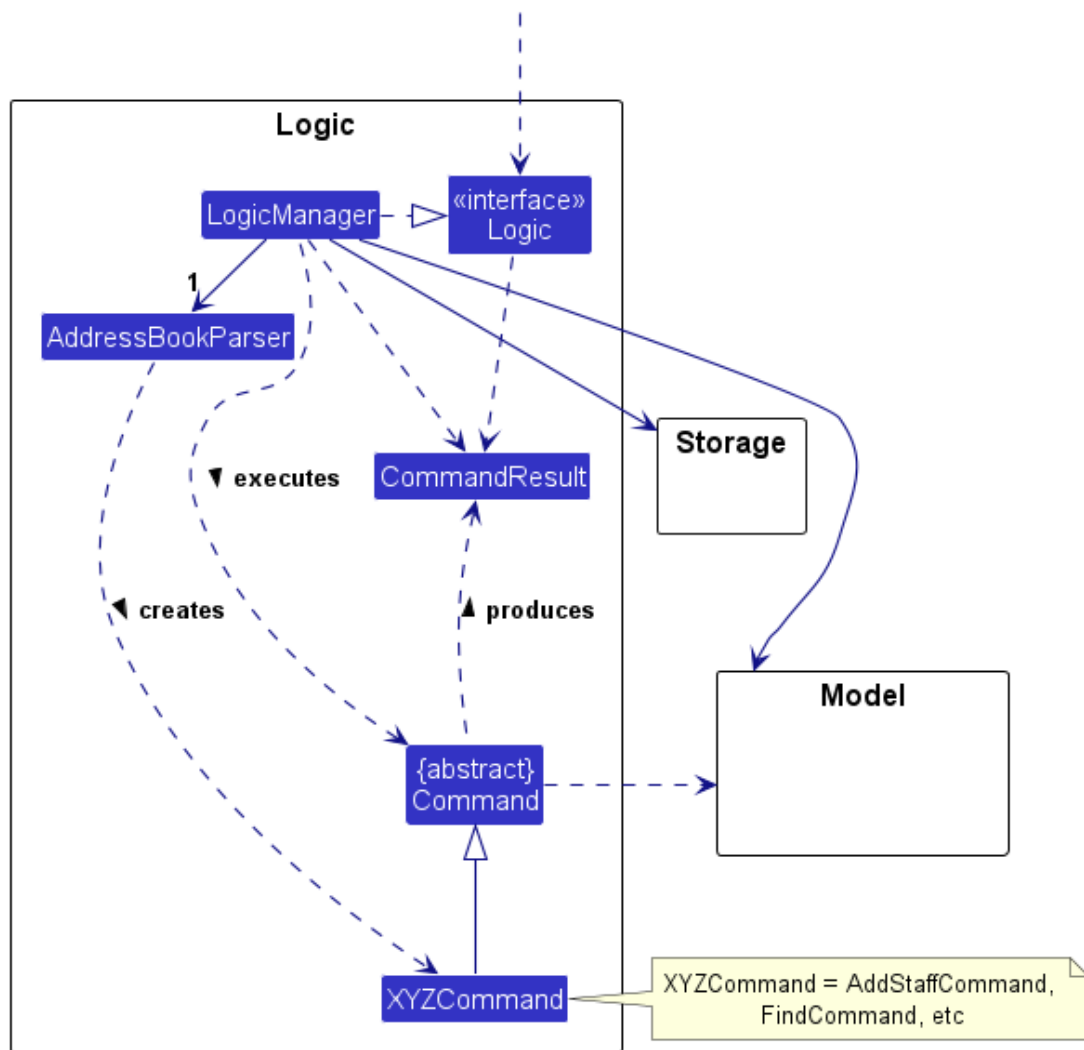
The `UI` component,

- executes user commands using the `Logic` component.
- listens for changes to `Model` data so that the UI can be updated with the modified data.
- keeps a reference to the `Logic` component, because the `UI` relies on the `Logic` to execute commands.
- depends on some classes in the `Model` component, as it displays `Person` object residing in the `Model`.

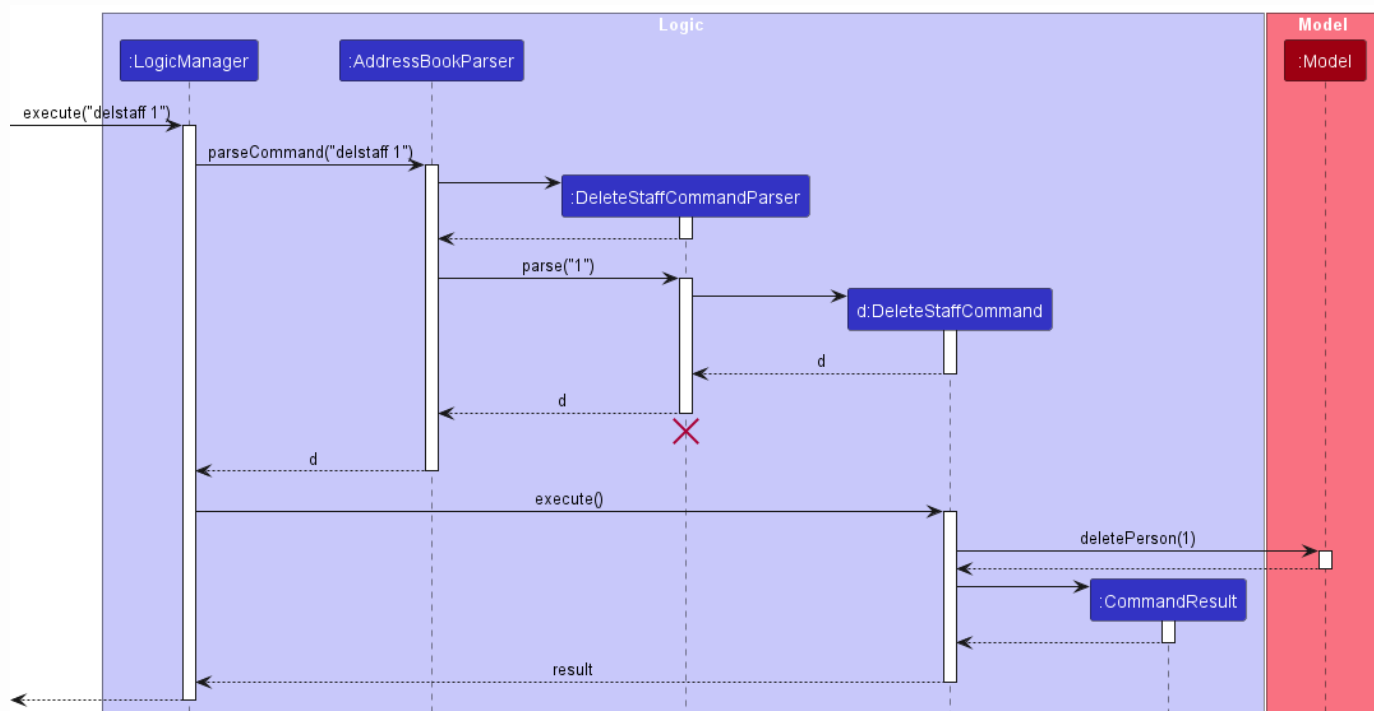
Logic component

API: `Logic.java`

Here's a (partial) class diagram of the `Logic` component:



The sequence diagram below illustrates the interactions within the **Logic** component, taking **execute("delstaff 1")** API call as an example.

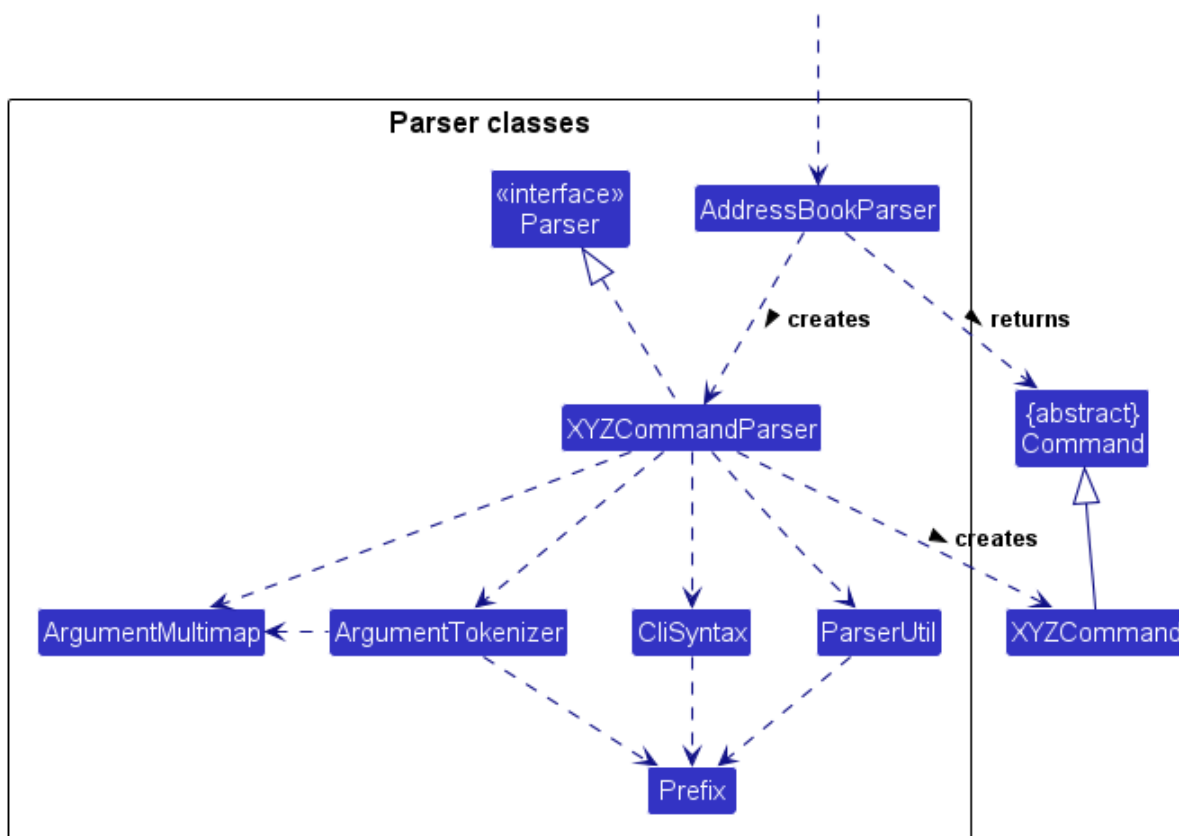


Note: The lifeline for `DeleteStaffCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

How the `Logic` component works:

1. When `Logic` is called upon to execute a command, it is passed to an `AddressBookParser` object which in turn creates a parser that matches the command (e.g., `DeleteStaffCommandParser`) and uses it to parse the command.
2. This results in a `Command` object (more precisely, an object of one of its subclasses e.g., `DeleteStaffCommand`) which is executed by the `LogicManager`.
3. The command can communicate with the `Model` when it is executed (e.g. to delete a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is returned back from `Logic`.

Here are the other classes in `Logic` (omitted from the class diagram above) that are used for parsing a user command:



How the parsing works:

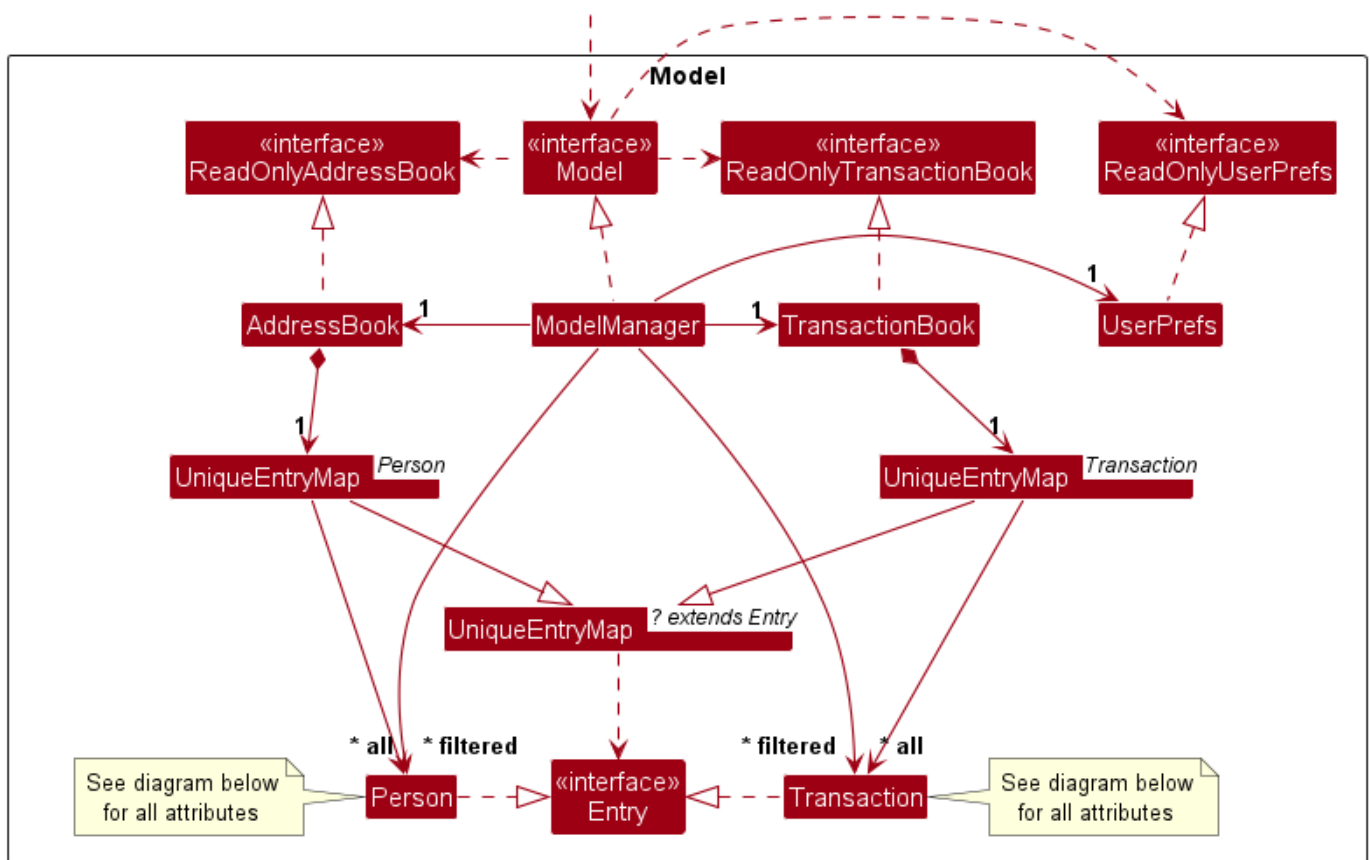
- When called upon to parse a user command, the `AddressBookParser` class creates an `XYZCommandParser` (`XYZ` is a placeholder for the specific command name e.g., `AddStaffCommandParser`) which uses the other classes shown above to parse the user

command and create a `XYZCommand` object (e.g., `AddStaffCommand`) which the `AddressBookParser` returns back as a `Command` object.

- All `XYZCommandParser` classes (e.g., `AddStaffCommandParser`, `DeleteStaffCommandParser`, ...) inherit from the `Parser` interface so that they can be treated similarly where possible e.g, during testing.

Model component

API : `Model.java`

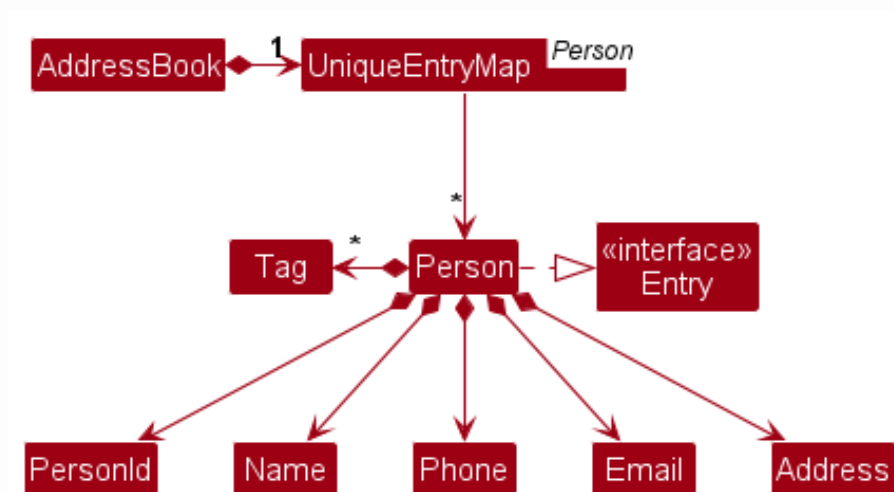


AddressBook

- Stores the address book data i.e., all `Person` objects (which are contained in a `UniqueEntryMap` object).
- Stores the currently 'selected' `Person` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Person>` 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.
- Stores a `UserPref` object that represents the user's preferences. This is exposed to the outside as a `ReadOnlyUserPref` objects.
- Does not depend on any of the other three components (as the `Model` represents data entities of the domain, they should make sense on their own without depending on other

components)

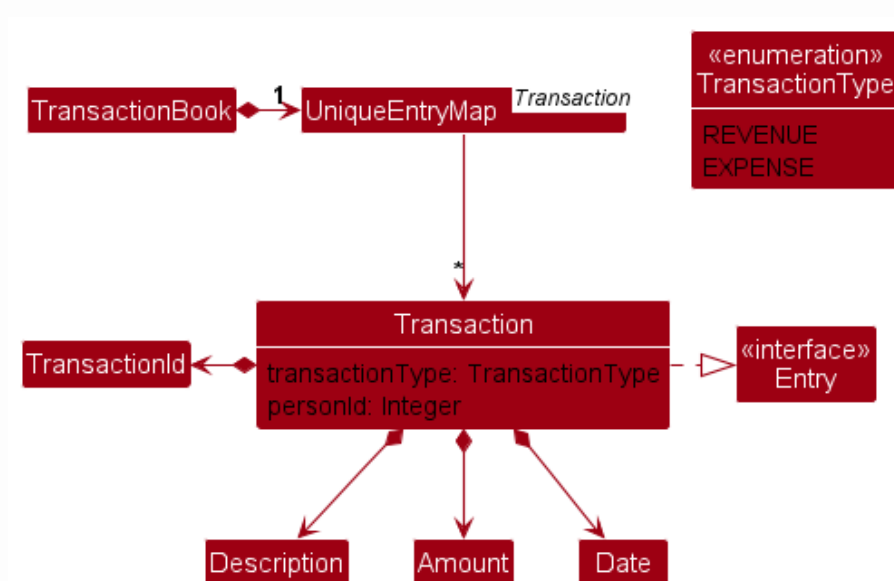
Person class diagram



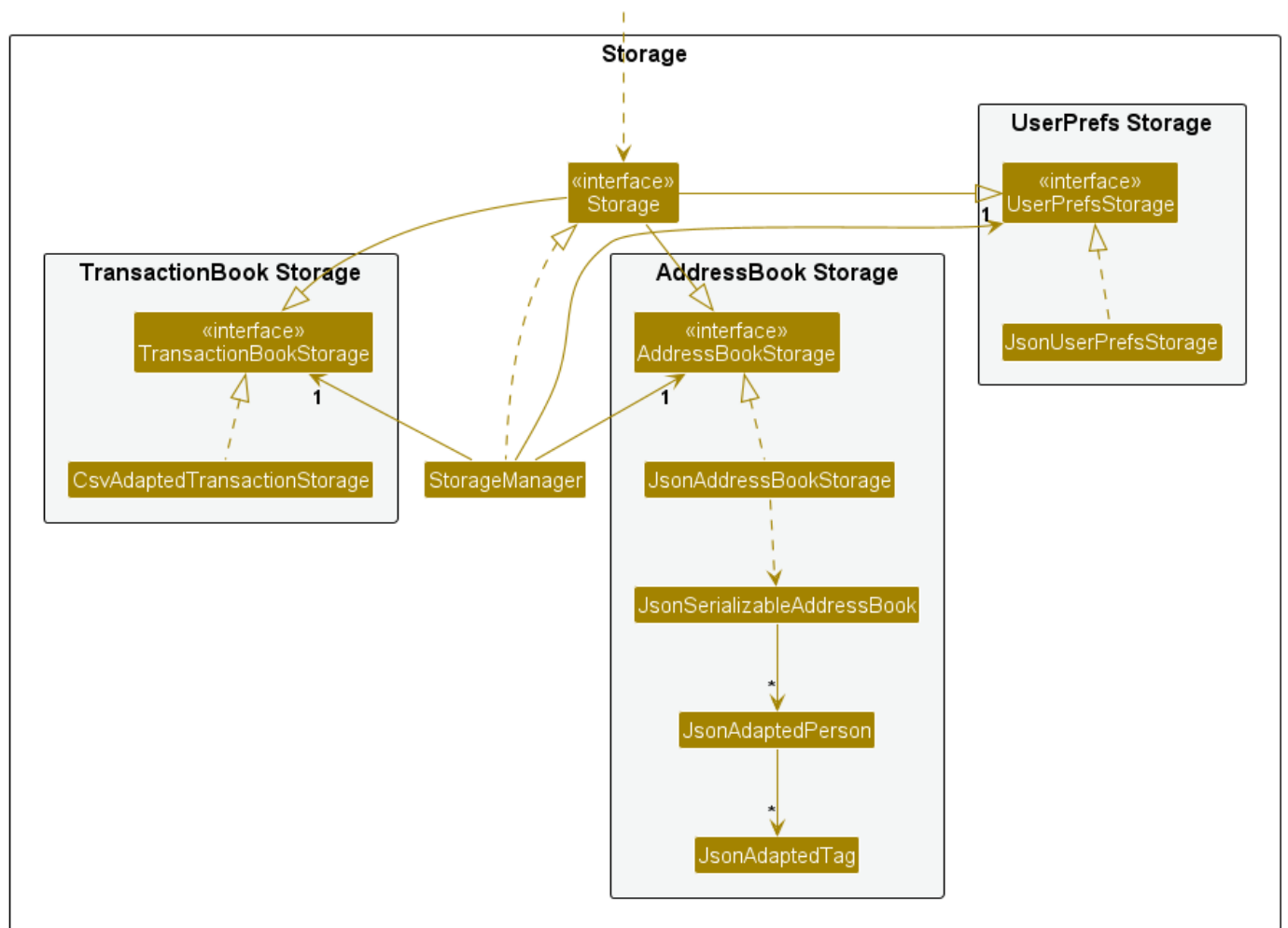
TransactionBook

- Introduces a new component, the `TransactionBook`, responsible for managing financial transactions.
- Utilizes a `UniqueEntryMap` to associate transactions with their unique `transactionId`.
- Similar to the `AddressBook`, maintains an `ObservableList<Transaction>` that can be observed by external entities, facilitating automatic updates in the UI.
- In addition to a *filtered* list like in `AddressBook`, the `TransactionBook` also maintains a *sorted* list of transactions, which is used to sort filtered transactions by relevant attributes.

Transaction class diagram



Storage component



The `Storage` component,

- can save address book data and user preference data in JSON format, and saves transaction book data in CSV format. It can also read them back into corresponding objects.
- inherits from `AddressBookStorage`, `TransactionBookStorage` and `UserPrefsStorage`, which means it can be treated as either one (if only the functionality of only one is needed).
- depends on some classes in the `Model` component (because the `Storage` component's job is to save/retrieve objects that belong to the `Model`)

Common classes

Classes used by multiple components are in the `transact.commons` package.

Implementation

This section describes some noteworthy details on how certain features are implemented.

[Proposed] Undo/redo feature

Proposed Implementation

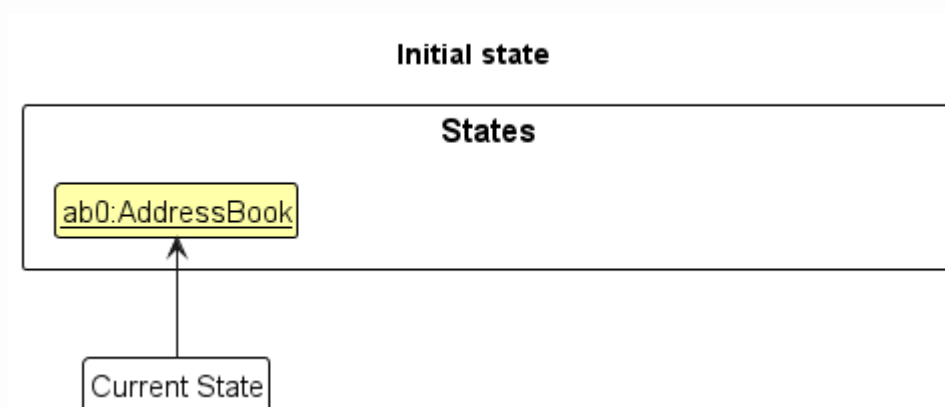
The proposed undo/redo mechanism is facilitated by `VersionedAddressBook`. It extends `AddressBook` with an undo/redo history, stored internally as an `addressBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedAddressBook#commit()` — Saves the current address book state in its history.
- `VersionedAddressBook#undo()` — Restores the previous address book state from its history.
- `VersionedAddressBook#redo()` — Restores a previously undone address book state from its history.

These operations are exposed in the `Model` interface as `Model#commitAddressBook()`, `Model#undoAddressBook()` and `Model#redoAddressBook()` respectively.

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

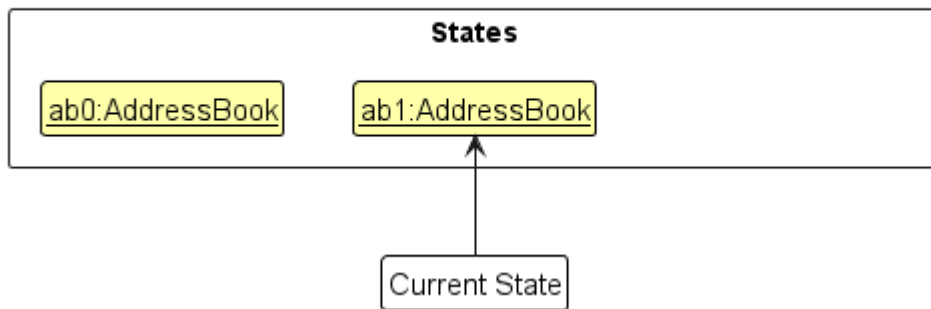
Step 1. The user launches the application for the first time. The `VersionedAddressBook` will be initialized with the initial address book state, and the `currentStatePointer` pointing to that single address book state.



Step 2. The user executes `delete 5` command to delete the 5th person in the address book. The `delete` command calls `Model#commitAddressBook()`, causing the modified state of the address book after the `delete 5` command executes to be saved in the

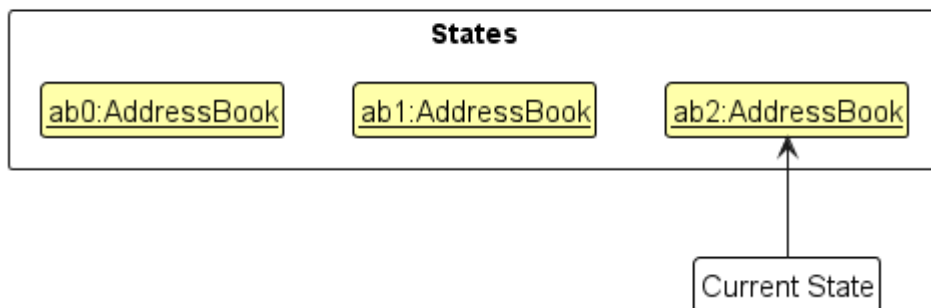
`addressBookStateList`, and the `currentStatePointer` is shifted to the newly inserted address book state.

After command "delete 5"



Step 3. The user executes `add n/David ...` to add a new person. The `add` command also calls `Model#commitAddressBook()`, causing another modified address book state to be saved into the `addressBookStateList`.

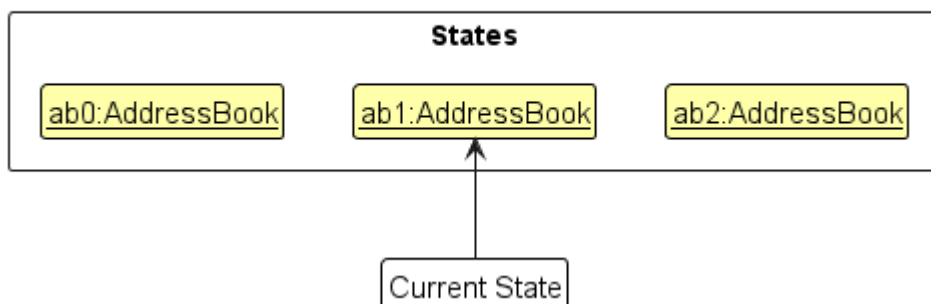
After command "add n/David"



Note: If a command fails its execution, it will not call `Model#commitAddressBook()`, so the address book state will not be saved into the `addressBookStateList`.

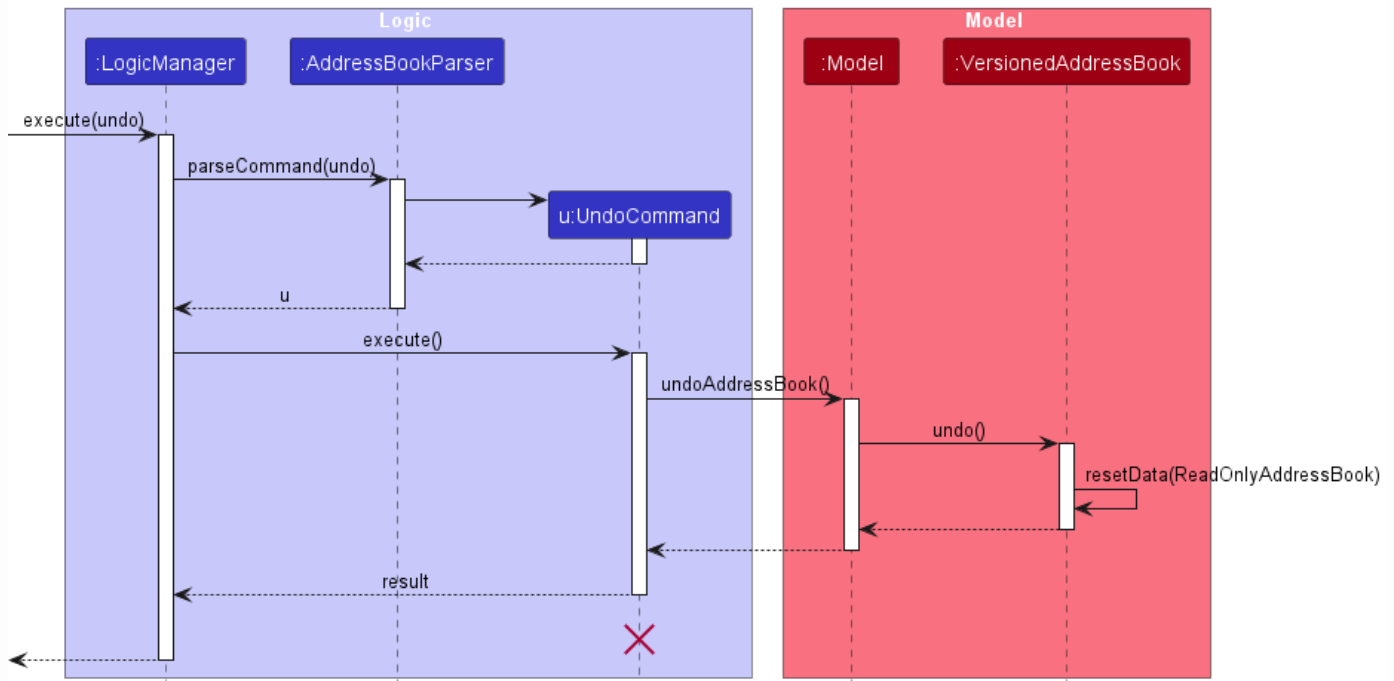
Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoAddressBook()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous address book state, and restores the address book to that state.

After command "undo"



Note: If the `currentStatePointer` is at index 0, pointing to the initial `AddressBook` state, then there are no previous `AddressBook` states to restore. The `undo` command uses `Model#canUndoAddressBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:



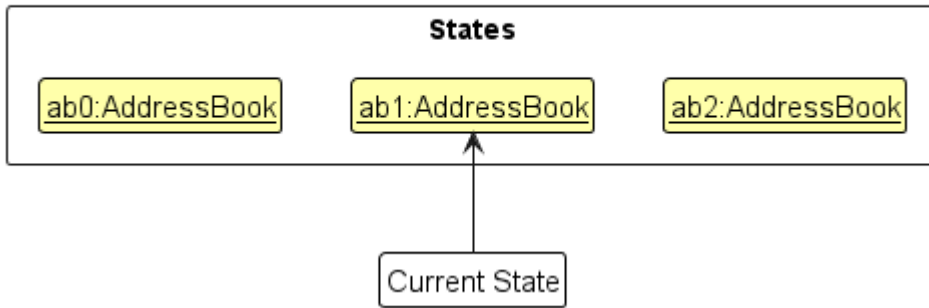
Note: The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The `redo` command does the opposite — it calls `Model#redoAddressBook()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the address book to that state.

Note: If the `currentStatePointer` is at index `addressBookStateList.size() - 1`, pointing to the latest address book state, then there are no undone `AddressBook` states to restore. The `redo` command uses `Model#canRedoAddressBook()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

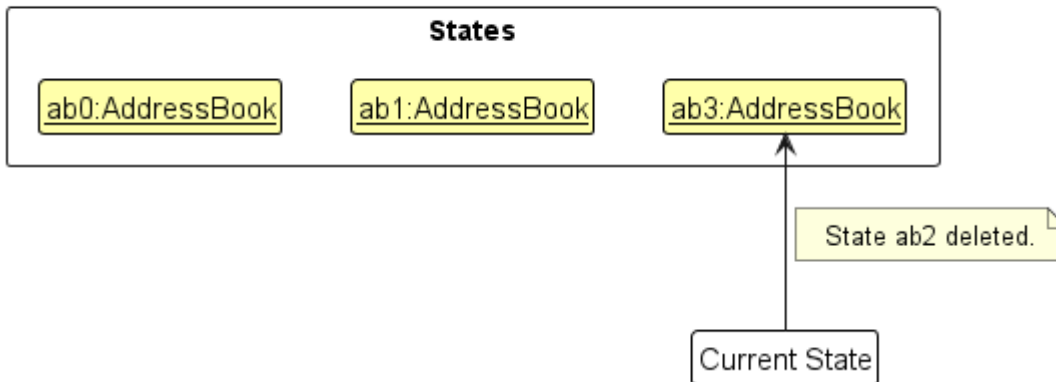
Step 5. The user then decides to execute the command `list`. Commands that do not modify the address book, such as `list`, will usually not call `Model#commitAddressBook()`, `Model#undoAddressBook()` or `Model#redoAddressBook()`. Thus, the `addressBookStateList` remains unchanged.

After command "list"



Step 6. The user executes `clear`, which calls `Model#commitAddressBook()`. Since the `currentStatePointer` is not pointing at the end of the `addressBookStateList`, all address book states after the `currentStatePointer` will be purged. Reason: It no longer makes sense to redo the `add n/David ...` command. This is the behavior that most modern desktop applications follow.

After command "clear"



The following activity diagram summarizes what happens when a user executes a new command:



Design considerations:

Aspect: How undo & redo executes:

- **Alternative 1 (current choice):** Saves the entire address book.
 - Pros: Easy to implement.
 - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
 - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
 - Cons: We must ensure that the implementation of each individual command are correct.

{more aspects and alternatives to be added}

[Proposed] Data import

The data import feature will enable users to seamlessly bring in external data into the application, supporting formats such as CSV and JSON. The process involves several key steps:

1. User Command or Interface:

- Define a user command or interface to initiate the data import process.
- Users can invoke the import feature through a specific command, such as "import," followed by the path to the external file.

2. Parsing External Data:

- Implement parsers for the supported data formats (e.g., CSV, JSON).
- These parsers, such as `CsvAdaptedTransactionStorage` and `JsonAddressBookStorage`, are responsible for interpreting the external data.
- Extract relevant information from the external file and convert it into the internal format used by the application.

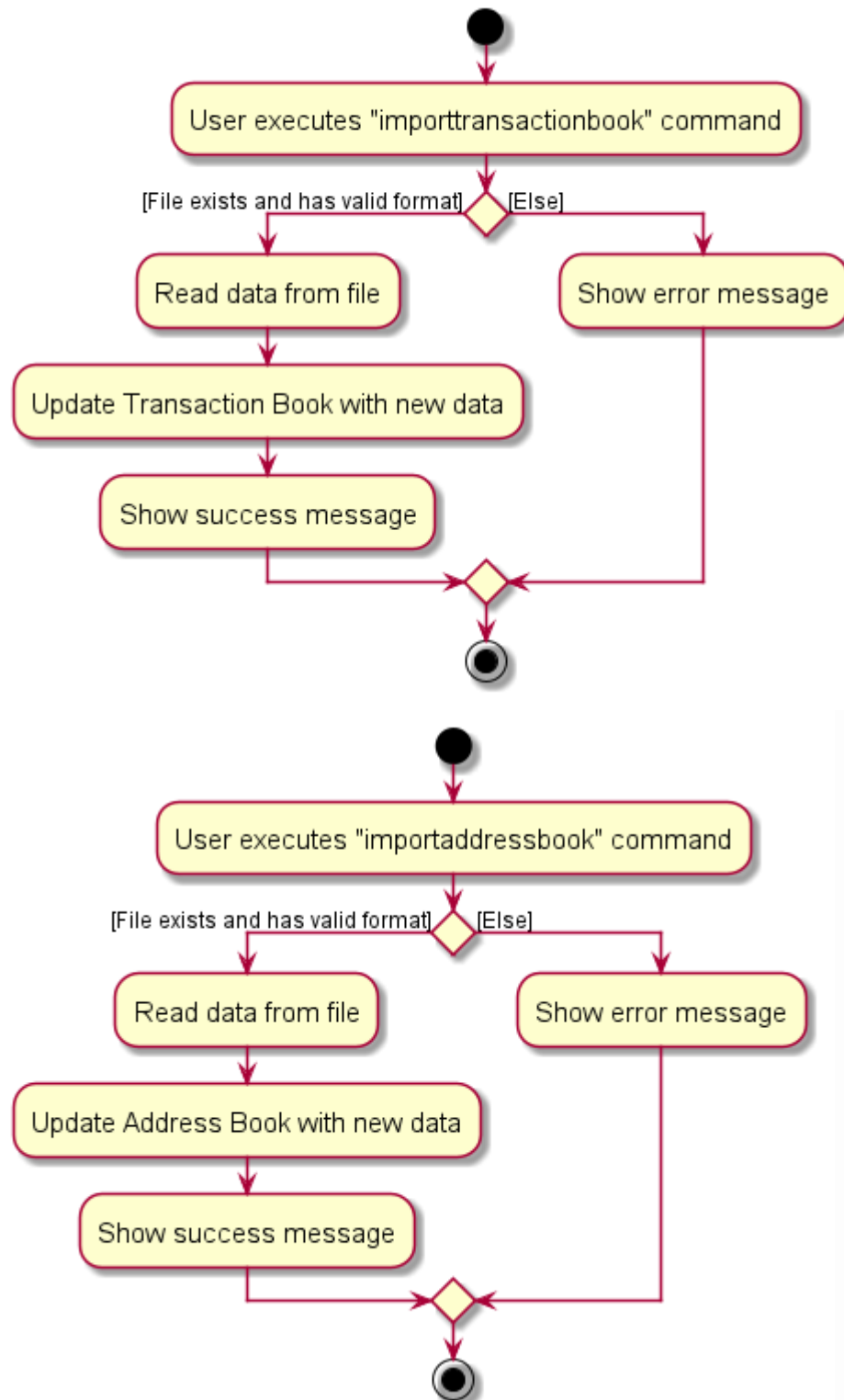
3. Updating Internal Model:

- Update the internal model components, such as `AddressBook` and `TransactionBook`, with the imported data.
- Utilize methods like `readTransactionBook(Path path)` in `CsvAdaptedTransactionStorage.java` and `readAddressBook(Path path)` in `JsonAddressBookStorage(Path filePath)` for transaction book and address book importation, respectively.
- Ensure proper validation of the imported data and handle potential conflicts with existing data appropriately.

4. Feedback to User:

- Implement a robust feedback mechanism to inform users about the success or failure of the import operation.
- Assign different types of exceptions to various format errors or inconsistencies with the required format.
- Provide detailed feedback on any issues or conflicts encountered during the import process.

- Consider generating a `CommandResult` that reflects the outcome of the import operation, indicating success or specifying the encountered errors.



The "import transactionbook" and "import addressbook" features shall be designed as a subclass of the Command class, responsible for coordinating the entire import process. When `execute()`, the command class invokes the above-mentioned methods for reading and updating the internal model. The `CommandResult` generated by the command depends on the success or failure of the import operation, providing valuable feedback to the user.

Documentation, logging, testing, configuration, dev-ops

- [Documentation guide](#)
 - [Testing guide](#)
 - [Logging guide](#)
 - [Configuration guide](#)
 - [DevOps guide](#)
-

Appendix: Requirements

Target User Profile

Tran\$act is specifically tailored for accountants and finance professionals in small businesses who have the following characteristics:

- **Need to manage a significant number of transactions:** Tran\$act is designed to handle a high volume of financial transactions efficiently.
- **Fast typists:** Users are expected to be proficient in typing, as the application primarily relies on keyboard inputs for data entry and interaction.
- **Prefer CLI interactions:** Our users prefer command-line interactions over graphical user interfaces, as it allows for quicker data entry and navigation.
- **Reasonably comfortable with CLI apps:** Users should have a basic understanding of using command-line applications.
- **Required to produce financial reports:** Tran\$act can automatically produce financial reports given the transactions that have been entered.
- **Required to sort or filter transactions:** Tran\$act can automatically filter and sort transactions based on the user's needs.
- **Value Proposition:** Quickly add company revenue and expenses via the CLI. Easily keep track of company profits and visualize them. Cheap solution for small businesses with limited capital.

User Stories

Priority	As a ...	I want to ...	So that I can...
***	new user	see usage instructions	refer to instructions when I forget how to use the App
***	user	add a new person	
***	user	delete a person	remove entries that I no longer need
***	user	find a person by name	locate details of persons without having to go through the entire list
**	user	hide private contact details	minimize chance of someone else seeing them by accident
*	user with many persons in the address book	sort persons by name	locate a person easily
Transaction Recording			
***	user	add a new transaction	
***	user	remove a transaction	
***	user	view all transactions	
**	user	edit transactions through the software	
**	user	edit transactions without opening the software	

Priority	As a ...	I want to ...	So that I can...
Financial Reporting			
*	user who needs to analyze transactions	have a variety of financial reports	
*	user who needs to share data with others	generate reports downloadable in common formats (PDF, CSV)	
*	user	restore from a backup	undo large changes
Data Security and Backup			
*	user who needs to keep sensitive data confidential	encrypt the data	ensure the security and privacy of financial data
*	user who does not have reliable hardware	have automated backups	prevent unnecessary data loss
*	user	restore from a backup	undo large changes

Transaction Recording

1. Adding a Transaction

- As an accountant, I want to add a new financial transaction quickly.
- I should be able to specify the amount, type (revenue or expense), date, and optionally, the associated person for the transaction.

2. Removing a Transaction

- As an accountant, I want to delete a transaction when necessary.
- I should be able to remove a transaction from the records to correct mistakes or manage data.

3. Viewing All Transactions

- As an accountant, I want to see a list of all recorded transactions for reference.
- This list will provide an overview of all financial activities in one place.

4. **Editing a Transaction**

- As an accountant, I need the ability to edit transaction details.
- I should be able to make corrections or updates to transaction records as needed.

5. **Restoring Deleted Transactions**

- As an accountant, I want a safety net for accidental deletions.
- I should be able to retrieve transactions I have mistakenly deleted from a "bin" or archive.

6. **Fast Data Entry**

- As a fast typist, I want shortcuts and efficient data entry methods.
- This will enable me to record transactions, including revenue and expenses, quickly via the CLI.

7. **Sorting Transactions**

- As an accountant, I want to be able to sort through my transactions.
- This will enable me to view my transactions in the order I want them to be viewed in.

8. **Filtering Transactions**

- As an accountant, I want to be able to filter my transactions.
- This will enable me to see the transactions I want to see.

9. **Exporting Transactions**

- As an accountant, I want to be able to export my transactions into csv format.
- This will allow me to share transaction data with stakeholders, or use the data to do further analysis in Excel.

Dashboard Overview

1. **Dashboard Display**

- As an accountant, I want to see a clear and concise dashboard upon opening the app.
- The dashboard should display total revenue, total expenses, net profit for the selected period (usually monthly), and a breakdown of expenses by sector to improve cost efficiency.

Financial Reporting

1. **Access to Financial Reports**

- As an accountant who analyzes transactions, I need access to various financial reports.
- I should be able to generate income statements, balance sheets, and cash flow statements.

2. **Customizable Reports**

- As an accountant who shares data with stakeholders, I want to generate customizable reports.
- I should be able to create reports in common formats (PDF, CSV, Excel) to share with others.

Data Security and Backup

1. Data Security

- As a user who values data privacy, I expect the app to secure financial data.
- The app should implement security measures, potentially including encryption, to protect sensitive information.

2. Automated Backups

- As a user concerned about data loss, I want the option for automated backups.
- The app should allow me to set up automated backups to prevent data loss due to hardware issues.

3. Undo and Restore

- As a user prone to mistakes, I need the ability to undo actions or restore from backups.
- This feature will help me recover from errors or data corruption.

Address Book

1. Exporting Staff List

- As a user who manages staff, I want to export the staff list into json format.
- This will help me share contact information with associates in a file format that can be sent.

2. Importing Staff List

- As a user who needs efficiency, I want to import a staff list into the address book.
- This will help me quickly access contact information for employees and associates.

3. Adding and Editing People

- As a user who manages relationships, I need to add, edit, and remove people from the address book.
- This allows me to keep the address book up to date and accurate.

4. Viewing Address Book

- As a user who relies on contact information, I want to view the entire address book.
- This provides easy access to contact details for individuals in the address book.

Use Cases

Use Case 1: Adding a Transaction

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant specifies the transaction details, including the amount, type (revenue or expense), date, and optionally, the associated person.
 2. The system validates the input data.
 3. The system records the transaction in the database.
- Use case ends

Extensions:

2a. Input data is invalid

- 2a1. The system displays an error message.
- Use case ends

Use Case 2: Removing a Transaction

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant specifies a transaction to remove.
 2. The system removes the transaction from the transaction book.
- Use case ends

Extensions:

- 2a. Transaction to remove does not exist
 - 2a1. The system displays an error message.
- Use case ends

Use Case 3: Viewing All Transactions

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to view all transactions.
 2. The system retrieves and displays a list of all recorded transactions.
- Use case ends

Use Case 4: Editing a Transaction

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant enters command with the details of the transaction to edit.
 2. The system validates the input data.
 3. The system saves the updated transaction in the database.
- Use case ends

Extensions:

- 2a. Input data is invalid
 - 2a1. The system displays an error message.
- Use case ends

Use Case 5: Restoring Deleted Transactions

Value proposition: Quickly add company inflow and outflow via the CLI. Easily keep track of company profits and expenses associated with staff and visualize them.

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to restore deleted transactions.
2. The system presents a list of previously deleted transactions.
3. The accountant selects a transaction to restore.

4. The system restores the selected transaction to the active records.
Use case ends

Use Case 6: Dashboard Display

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects option to display dashboard.
2. The dashboard shows total revenue, total expenses, net profit for the selected period (usually monthly), and a breakdown of expenses by sector.
Use case ends

Use Case 7: Access to Financial Reports

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to access financial reports.
2. The system generates and displays financial reports, including income statements, balance sheets, and cash flow statements.
Use case ends

Use Case 8: Customizable Reports

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to generate customizable reports.
2. The system displays options for the accountant to customize the report parameters.
3. The system generates and displays the customized report in common formats (PDF, CSV, Excel).
Use case ends

Use Case 9: Deleting All Transactions

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to delete all transactions.
 2. The system permanently deletes all recorded transactions from the database.
- Use case ends

Use Case 10: Adding a Staff Member to Address Book

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant enters command to add a new staff member to the address book with relevant details.
 2. The system validates the input data.
 3. The system adds the new staff member to the address book.
- Use case ends

Extensions:

- 2a. Input data is invalid
 - 2a1. The system displays an error message.
- Use case ends

Use Case 11: Removing a Staff Member from Address Book

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant enters the command to remove a staff member from the address book.
 2. The system removes the selected staff member from the address book.
- Use case ends

Extensions:

- 2a. Staff to remove does not exist
 - 2a1. The system displays an error message.Use case ends

Use Case 12: Editing Staff Member Information in Address Book

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant enters command to edit a staff member with the relevant details.
 2. The system validates the updated information.
 3. The system saves the updated staff member information in the address book.
- Use case ends

Extensions:

- 2a. Input data is invalid
 - 2a1. The system displays an error message.Use case ends

Use Case 13: Deleting All Address Book Staff Members

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant selects the option to delete all address book staff members.
 2. The system permanently deletes all staff members from the address book.
- Use case ends

Use Case 14: Filtering Transactions

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant specifies the parameters for which they want to filter for transactions by.
 2. System displays only transactions that fit the parameters specified.
- Use case ends

Use Case 15: Sorting Transactions

Actor: Accountant

Preconditions: The accountant is logged into the Tran\$act application.

MSS:

1. The accountant specifies the parameters for which they want to sort transactions by.
 2. System sorts the transactions based on the parameters specified.
- Use case ends

Non-functional Requirements (NFR)

- Tran\$act should support the storage of at least 1000 transactions per month to accommodate the needs of small businesses.
- Should work on any *mainstream* OS as long as it has Java 11 or above installed.
- Should be able to hold up to 1000 transactions without a noticeable sluggishness in performance for typical usage.
- 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.

Glossary

- **Accountant:** A professional who is responsible for managing a company/institution's financial records.
 - **Mainstream OS:** Windows, Linux, Unix, OS-X.
 - **Revenue:** Money received, such as sales revenue.
 - **Expense:** Costs incurred, including staff salaries and product costs.
 - **Transaction:** An exchange of money, which includes buying and selling activities.
-

Appendix: Planned Enhancements

In addition to proposed features in the [implementation](#) section, we have listed additional planned enhancements below:

- Allow export of transactions with filter and sort rules applied
 - Adding sub-categories to transactions, such as "cost of goods sold", "operating expenses", "taxes", "capital expenditures", etc.
 - Allow renaming of the file before exporting
 - Add separation between command results in results box for easier distinguishing
 - Add more graph types on dashboard for better visualisation
-

Appendix: Instructions for manual testing

Given below are instructions to test the app manually.

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

Launch and shutdown

1. Initial launch

1. Download the jar file and copy into an empty folder
2. Double-click the jar file Expected: Shows the overview GUI with a set of sample transactions loaded in. The window size may not be optimum.

2. Saving window preferences

1. Resize the window to an optimum size. Move the window to a different location. Close the window.
2. Re-launch the app by double-clicking the jar file. Expected: The most recent window size and location is retained.

Deleting a person

1. Deleting a person

1. Prerequisites: View the staff list using the `view s` command.
2. Test case: `delstaff 1` Expected: Staff with id 1 is deleted from the list. Details of the deleted contact shown in the status message.
3. Test case: `delstaff 100` Expected: No staff is deleted. Error details shown in the status message.
4. Other incorrect delete commands to try: `delete`, `delete x` (where x is not associated with any staff id) Expected: Similar to previous.

Saving data

1. Test Case: Click on `export transactions` in menu bar

1. Expected: File chooser pops up, allowing you to choose location where transactions will be saved. `transaction.csv` file will be created at chosen location.