

WalletCLi - Developer Guide

By: Team CS2113T - W17-2 Since: Aug 2019 Licence: MIT

Table of Contents

1. Introduction	5
2. About this Developer Guide.....	5
3. Setting up	6
3.1 Prerequisites	6
3.2 Setting up the project in your computer	6
3.3 Verifying Setup.....	7
4. Design.....	8
4.1 Architecture	8
4.2 Logic Component	9
4.3 Model Component.....	10
4.4 Storage Component	10
5. Implementation	11
5.1 Recurring expenses	11
5.1.1 Current Implementation	11
5.1.2 Design Considerations.....	13
5.1.3 Future Implementation.....	13
5.2 Managing Loans	13
5.2.1 Current Implementation	13
5.2.2 Design Consideration	15
5.2.3 Future Implementation.....	15
5.3 Managing Contacts	15
5.3.1 Listing Contact.....	15
5.3.2 Adding Contact Implementation.....	16
5.3.3 Editing Contact Implementation.....	18
5.3.4 Delete Contact Implementation	19
5.3.5 Design Consideration	20
5.3.6 Future Implementation.....	20
5.4 Budget Management	20
5.4.1 Current Implementation	20
5.4.2 Design Consideration	23
5.4.3 Future Implementation.....	23
5.5 Auto Reminders	24
5.5.1 Current Implementation	24
5.5.2 Design Consideration	25
5.6 Stats.....	25

5.6.1 Current Implementation	25
5.6.2 Design Consideration	26
5.6.3 Future Implementation.....	26
5.7 Help	27
5.7.1 Current Code Implementation.....	28
5.7.2 Adding New Sections	29
5.7.3 Design Consideration	29
5.7.4 Future Implementation.....	29
5.8 Managing Expenses.....	30
5.8.1 Listing Expense.....	30
5.8.2 Adding Expense Implementation.....	30
5.8.3 Editing Expense Implementation	30
5.8.4 Delete Expense Implementation.....	31
5.8.5 Future Implementation.....	31
5.9 Export Data	32
5.9.1 Current Implementation	32
5.9.2 Design Consideration	33
5.9.3 Future Implementation.....	33
5.10 Import Data	33
5.10.1 Current Implementation	34
5.10.2 Design Consideration	34
5.10.3 Future Implementation.....	35
Appendix A - Product Scope.....	35
Appendix B - User Stories.....	36
Appendix C - Use Cases	38
Appendix D - Non-functional Requirements.....	48
Appendix E - Glossary.....	49
Appendix F - Instructions for manual testing.....	49
F.1. Launch and Shutdown.....	49
F.2. Adding an expense	49
F.3. Editing an expense	50
F.4. Deleting an expense	50
F.5. Adding a contact	50
F.6. Editing a contact.....	51
F.7. Deleting a contact	51
F.8. Adding a loan.....	52

F.9. Editing a loan.....	52
F.10. Deleting a loan	53
F.11. Settling a loan.....	53
F.12. Listing your expenses/loans/contacts.....	53
F.13. Setting/Viewing budget.....	54
F.14. Viewing statistics.....	54
F.15. Changing currency.....	55
F.16. Viewing command history.....	55
F.17. Undoing/Redoing commands	55
F.18. Import data	55
F.19. Export data	56
F.20. Help	56

1. Introduction

Welcome to **WalletCLi**!

WalletCLi is a text-based (Command Line Interface) application that caters to NUS students and staff who prefer to use a desktop application for managing both their daily expenses and loans.

WalletCLi allows its users to create daily expenses and loans and enables easy editing and deletion. Users can also pre-set their budget, and **WalletCLi** will automatically track your current expenses to ensure that its users' expenses stay within their stated budget. Expenses and loans can be efficiently managed via our intuitive category system.



WalletCLi is optimized for those who prefer to work with a Command Line Interface (CLi) and/or are learning to work more efficiently with CLi tools. Additionally, unlike traditional wallet applications, **WalletCLi** utilizes minimal resources on the user's machine while still allowing users to manage their expenses and keep track of their loans swiftly and efficiently.

2. About this Developer Guide

This developer guide provides a detailed documentation on the implementation of all the various features **WalletCLi** offers. To navigate between the different sections, you could use the table of contents above.

For ease of communication, this document will refer to expenses/loans/contacts that you might add to the application as *data*.

Additionally, throughout this developer guide, there will be various icons used as described below.

	This is a note. These are things for you to take note of when using WalletCLi .
	This is a rule. Ensure that you follow these rules to ensure proper usage of WalletCLi .

3. Setting up

This section describes the procedures for setting up **WalletCLI**.

3.1 Prerequisites

1. JDK 11 or later
2. IntelliJ IDE



IntelliJ by default has Gradle installed.

Do not disable them. If you have disabled them, go to `File > Settings > Plugins` to re-enable them.

3.2 Setting up the project in your computer

1. Fork this repo and clone the fork to your computer.
2. Open IntelliJ (if you are not in the welcome screen, click `File > Close Project` to close the existing project dialog first).
3. Set up the correct JDK version for Gradle.
 - a. Click `Configure > Project Defaults > Project Structure`
 - b. Click `New...` and find the directory of the JDK.
4. Click `Import Project`.
5. Locate the `build.gradle` file and select it. Click `OK`.
6. Click `Open as Project`.
7. Click `OK` to accept the default settings.
8. Run the `wallet.Main` class (right-click the `Main` class and click `Run Main.main()`) and try executing a few commands.
9. Run all the tests (right-click the `test` folder and click `Run 'All Tests'`) and ensure that they pass.
10. Open the `StorageFile` file and check for any code errors.
11. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with `BUILD SUCCESSFUL` message.
This will generate all the resources required by the application and tests.
12. Open `Main.java` and check for any code errors.

Due to an ongoing issue with some of the newer versions of IntelliJ, code errors may be detected even if the project can be built and run successfully.

13. To resolve this, place your cursor over any of the code section highlighted in red.
Press `ALT+ENTER`, and select Add '`--add-modules=...`' to module compiler options for each error.

3.3 Verifying Setup

1. Run the `wallet.Main` and try a few commands.
2. Run the tests to ensure they all pass.

4. Design

4.1 Architecture

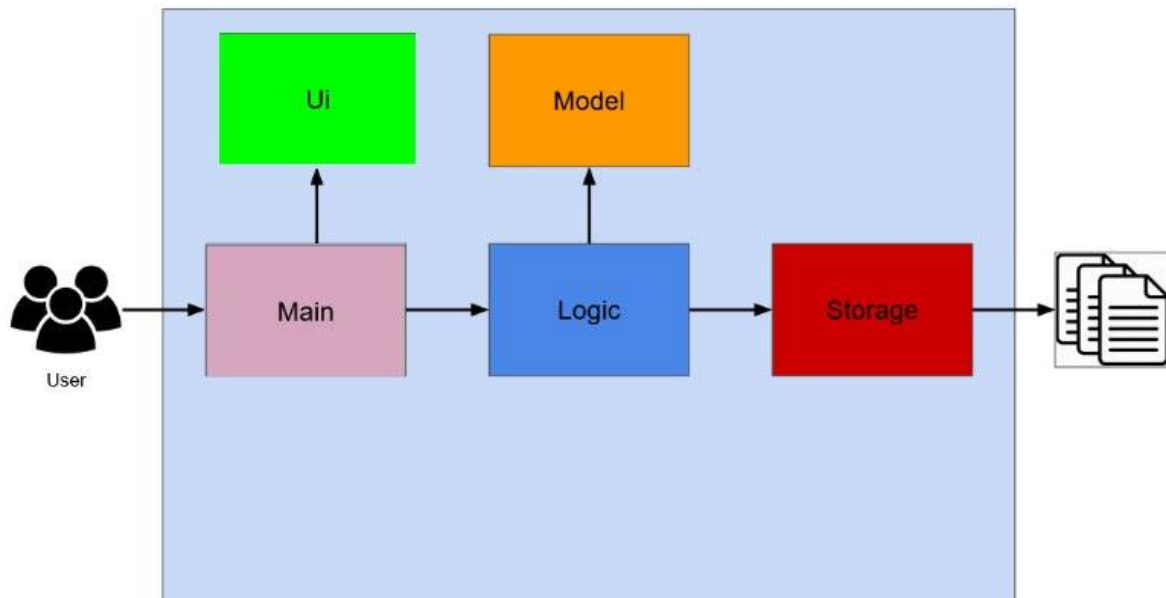


Figure 4.1.1 Architecture Diagram

The Architecture Diagram given above explains the high-level design of **WalletCLI**. Given below is a quick overview of each component.

Main is responsible for:

- At app launch: Initializes the components in the correct sequence and connect them up with each other.
- At shut down: Shuts down the components and invokes the clean-up method where necessary.

The rest of the App consists of 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.

Each of the four components

- Defines its API in an interface with the same name as the Component.
- Exposes its functionality using a {Component Name} Manager class.

4.2 Logic Component

The logic component shows the dependencies and multiplicities between *Parser* classes and *Command* classes.

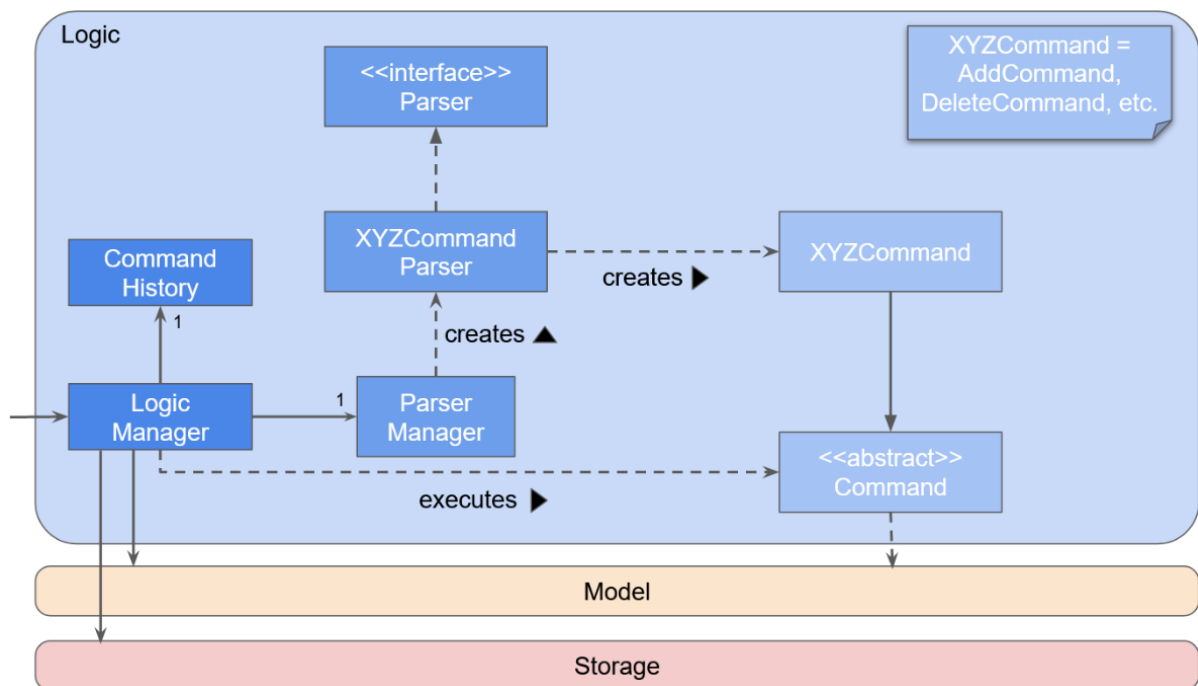


Figure 4.2.1 Structure of the Logic Component

API: LogicManager.java

The figure above shows the structure of the Logic Component.

1. Logic uses the Parser class to parse the user command.
2. This results in a command object which is executed by the LogicManager.
3. The command execution can affect the Model (e.g. adding an expense).

4.3 Model Component

The model component diagram shows dependencies and multiplicities between different *Model* classes.

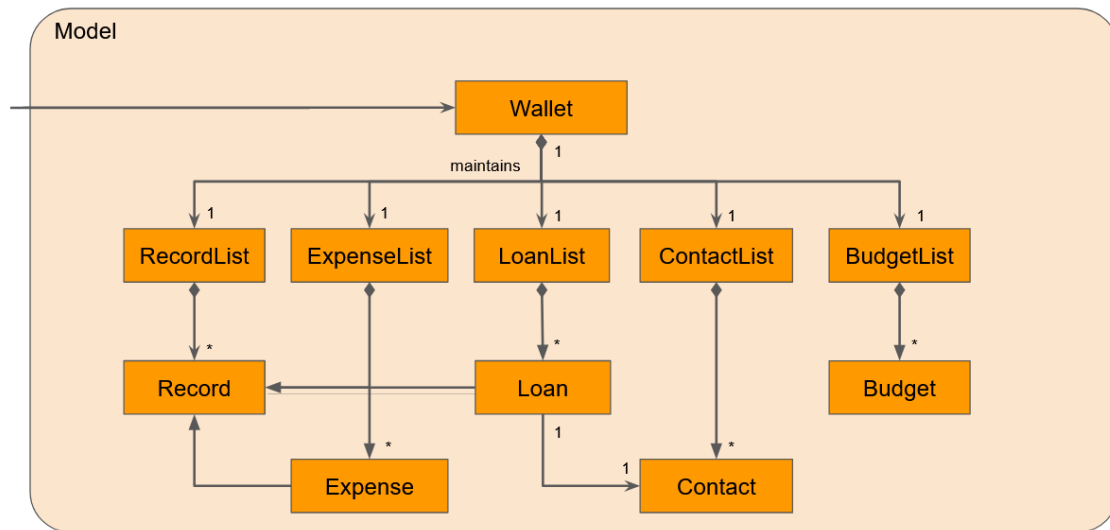


Figure 4.3.1 Structure of the Model Component

API: Wallet.java

The Model component:

- maintains the list of expenses, loans, contacts and budgets data.
- Does not depend on any of the other three components.

4.4 Storage Component

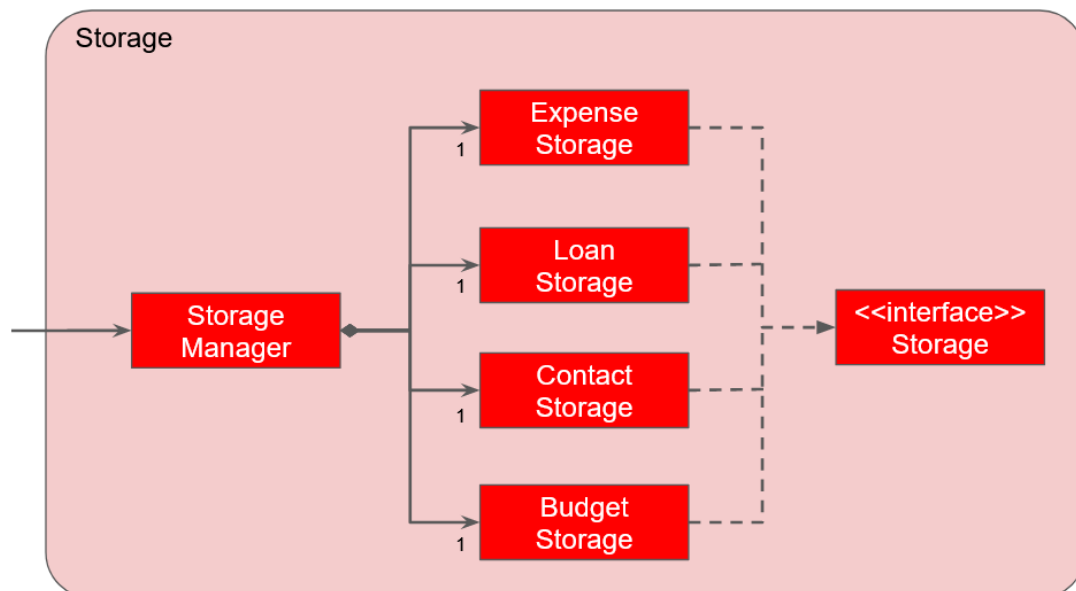


Figure 4.4.1 Structure of the Storage Component

API: StorageManager.java

The Storage component:

- can save Expense, Loan, Contact and Budget objects in csv format into text file and load them back into the application.

5. Implementation

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

5.1 Recurring expenses

The recurring mechanism allows users to add expenses that are automatically recurred by the system based on the rate of recurrence.

5.1.1 Current Implementation

The current implementation automatically updates recurring expenses until the end of the current month.

- The rate of recurrence can be daily, weekly or monthly.
- Recurrence is implemented as an additional parameter to the add command using the identifier `/r` to indicate that the expense is a recurring one.

Given below is an example usage scenario when adding a recurring expense. Assume that today's date is 10/10/2019.

1. The user executes `add expense Phone bill $40 Bills /on 05/09/2019 /r monthly` command to add a monthly recurring expense for his/her phone bill.
2. The add command parses the user input using the `AddCommandParser`, then returns a `AddCommand` object to `LogicManager` to invoke the `execute` function. This adds a new `Expense` object into the `ExpenseList`.

The following sequence diagram shows how the `add` command works:

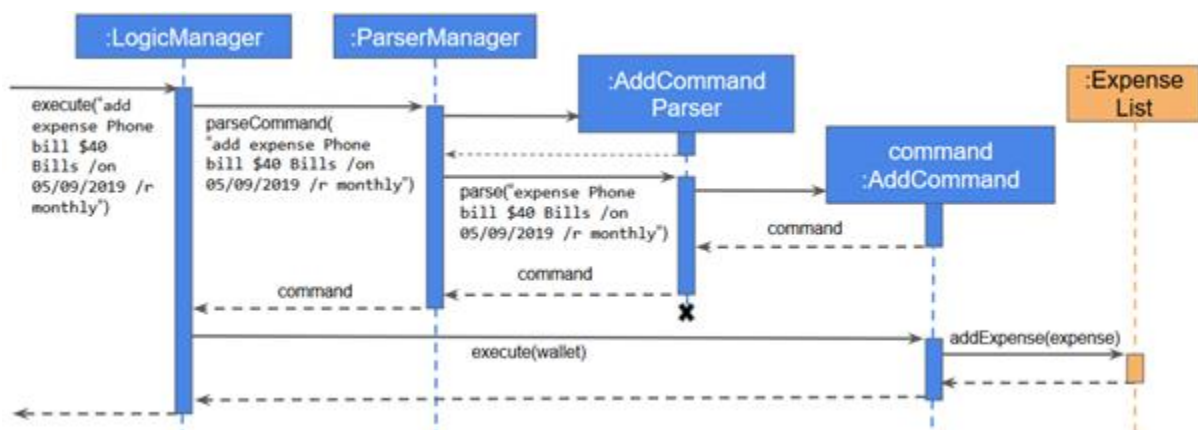


Figure 5.1.1.1 Sequence diagram when adding a new expense

1. Whenever a command is executed, the `LogicManager` class will invoke the `ExpenseParser` class to update all the recurring expenses using the `updateRecurringRecords` method.
2. The `ExpenseParser` class retrieves the `ExpenseList` object from the `wallet` class using the `getExpenseList` method.

3. After retrieving the ExpenseList, the ExpenseParser class invokes the getRecurringRecords method to filter out the recurring expenses in the list.
4. The ExpenseParser class then invokes the findExpenseIndex method to retrieve the index of the expense on the list.
5. The ExpenseParser class invokes the editExpense method to update this expense's isRecurring value to False.
6. The ExpenseParser class adds a month to the date of the recurring expense and checks if it is greater than the current month.
7. If the date is not greater than the current month, an expense is created with the same values to the initial recurring expense except for the date. The ExpenseParser class then invokes the AddCommand class to add this expense into the ExpenseList.
8. Repeat step 9 until the new date is greater than the current month.
9. Repeat steps 6 to 10 for every recurring expense in the ExpenseList

In this scenario, there is a monthly recurring expense that was just added. The system will generate another expense record with all the same values except the date being 05/10/2019.

The following sequence diagram shows how the system updates recurring expenses:

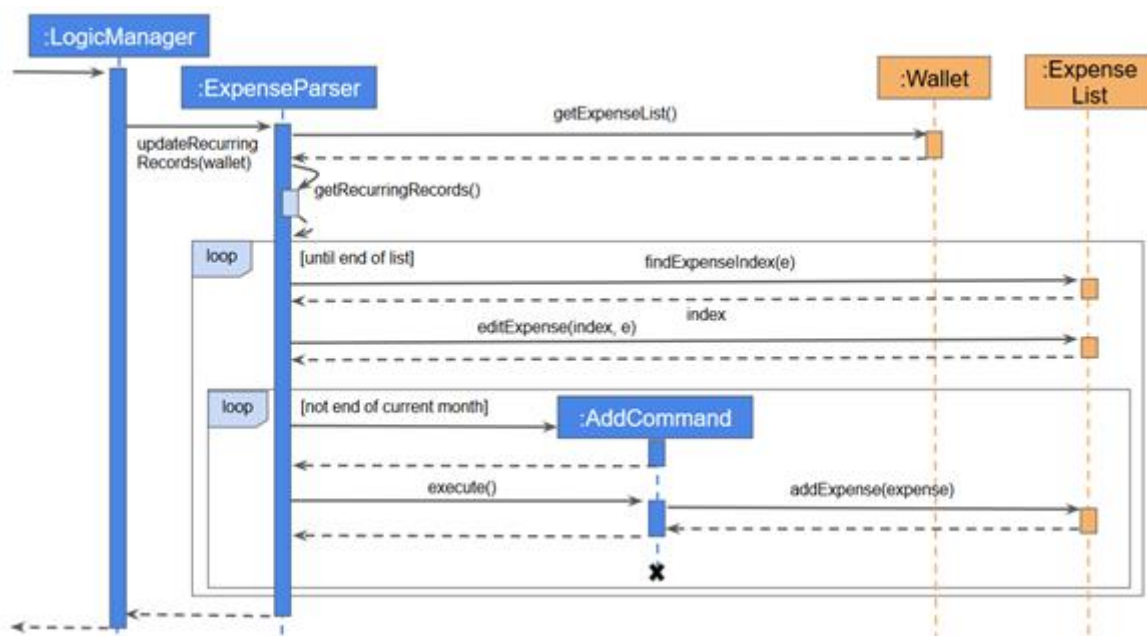


Figure 5.1.1.2 Sequence diagram updating recurring expenses

5.1.2 Design Considerations

Aspect: How the system handles recurring expenses

- **Alternative 1(current choice): System updates all recurring expenses whenever a command is executed.**

Pros: Whenever there are any changes made to any recurring expense, the system will automatically update it to the latest month.

Cons: May slow down the speed of the overall application.

- **Alternative 2: System stores this information separately first to tell itself to gradually updates recurring expenses when the current date is equal to the specified recurring expenses date.**

Pros: Less performance intensive, hence the speed of overall application will not be affected.

Cons: More memory usage despite a faster implementation.

5.1.3 Future Implementation

Provide more flexibility for setting rate of recurrence.

Currently, the rate of recurrence can only be set to daily, weekly or monthly. We can provide more flexibility to allow the users to add expenses that happens every fortnightly, every 2 days per week, or even a set number of intervals.

Allow user to set the number of times to recur.

Currently, the implementation is set to recur until the end of the current month. The limitation of this is that the user is unable to end the recurring expense at the start or middle of the month.

5.2 Managing Loans

5.2.1 Current Implementation

Users are able to manage their loans via these commands:

- `add loan <DESCRIPTION> <AMOUNT> [<date>] </l or /b> </c CONTACT ID>`
 - The Loan object takes in a description, the amount of money, the date of the loan, whether the user borrowed money from someone or lent it to somebody and last but not least, this loan must be tied to an existing contact.
- `edit loan <LOAN ID> [/d <NEW DESCRIPTION>] [/a <NEW AMOUNT>] [/t <NEW DATE>] [</l or /b>] [</c <CONTACT ID>]`
 - If the user made a mistake while adding loans, it is also possible to directly edit the loan via the ID of the loan.
 - Users can change the description, the amount, the date, whether the user borrowed money from someone or lent it to somebody and can also change the contact that the loan was tied to.
 - However, these parameters must be used in the given order.

- `done loan <LOAN ID>`
 - Allows users to settle their loans.
- `list loan`
 - Allows user to view the existing loans in a table format
- `delete loan <LOAN ID>`
 - Allows user to delete specific loans.

Given below is an example usage scenario when adding a loan. Assume that there exists a contact, Mary, whose contact ID is 1.

1. The user executes `add loan lunch $10 20/10/2019 /b /c 1`.
2. `parseCommand` is invoked and this constructs an `AddCommandParser` object.
3. The string is then parsed into the `AddCommandParser`.
4. As loans need to make reference to some existing contact, `findIndexWithId(1)` is invoked, which returns the index of the contact in the `ContactList` object.
5. With the index, `getContact(Index)` is called which returns the `Contact` object back to the `AddCommandParser` object.
6. Eventually, the command object is returned to the `LogicManager` object.

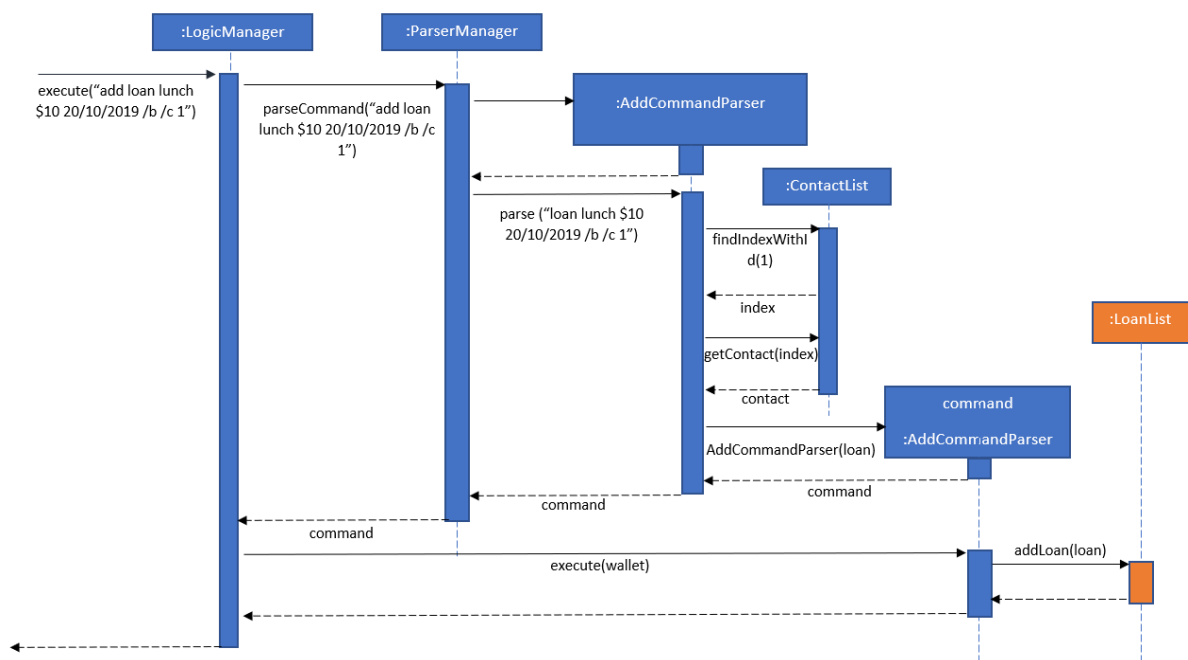


Figure 5.2.1 Sequence Diagram of Adding Loan

5.2.2 Design Consideration

Aspect: Managing Loans

- **Alternative 1: Loans need not take in a contact.**

Pros: This results in less coupling and better cohesion. Adding loans is now $O(1)$. Contact deletion will also not check whether there exists a contact in any existing loan.

Cons: Loans is no longer aware of contact.

- **Alternative 2: Loans takes in a contact. (current implementation)**

Pros: Loans is aware of contact, which makes loans easier to keep track.

Cons: Results in higher coupling and lesser cohesion. Adding loans is now $O(N)$ as it must search through the list of contacts before adding a loan object to the list of loans. Deletion of Contacts is also now $O(N)$ instead of $O(1)$ as it is necessary to search through the list of loans, ensuring that there are no loans containing the contact to be deleted.

5.2.3 Future Implementation

A future add-on to managing loans would be to sync the loans with other people's bank accounts. Since it is tied to the bank account, we will also consider the user to be able to print out proper legitimate loan paperwork.

5.3 Managing Contacts

Contacts are used in loan management to help users identify who they lend money to or borrow money from.

5.3.1 Listing Contact

To view contacts in a table form, users can use `list contact` command. Users can optionally provide `/sortby name` option to sort and list their contacts by alphabetical order. The sequence diagram below shows the main program flow of `list contact` command.

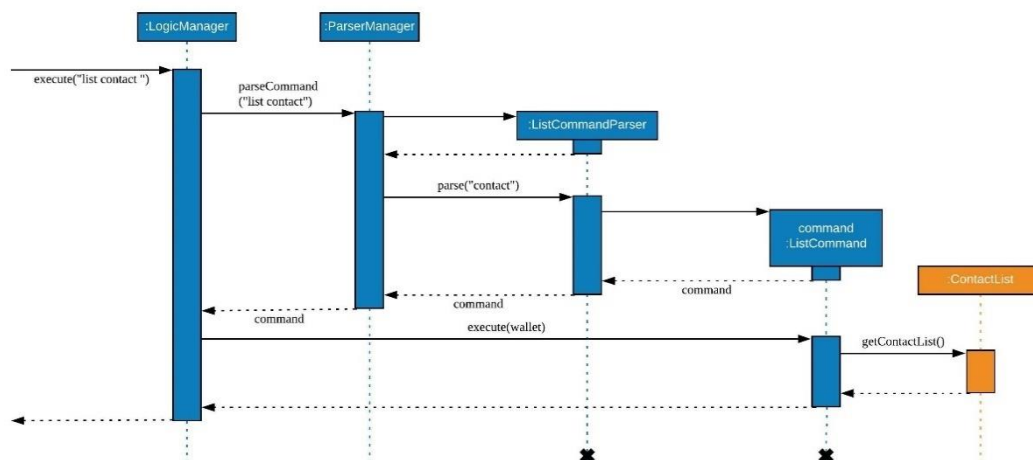


Figure 5.3.1.1 Sequence diagram for Listing Contact

The steps below will explain the sequence diagram shown above:

1. At the Logic Component, `ListCommandParser` will parse the command and check if the input is valid.
2. If the input is valid, a command object is returned to `LogicManager`.
3. `LogicManager` executes the command object and this results in the user's list of contacts to retrieved and displayed to user.
 - a. If user entered the `/sortby name` option, when the command object is executed, `sortByName()` function is called on top of `getContactList()` function to sort the list in alphabetical order. Regardless of whether the first letter is capital letter, the contacts will be sorted from letter A to letter Z as shown in the application screenshot in Figure 5.3.1.2.



ID	Name
2	Ben
7	ben
3	Jane
8	jane
6	jane tan
5	june
1	Lauren
4	Ryan Tang

Figure 5.3.1.2 Example Output for `/sortby name`

5.3.2 Adding Contact Implementation

To add contact entries into the application, users can use `add contact <Name> /p <Phone Number> /d <Details>` command. The following sequence diagram shows the implementation. The command used in the diagram is `add contact Mary Tan /p 8728 1831 /d sister`.

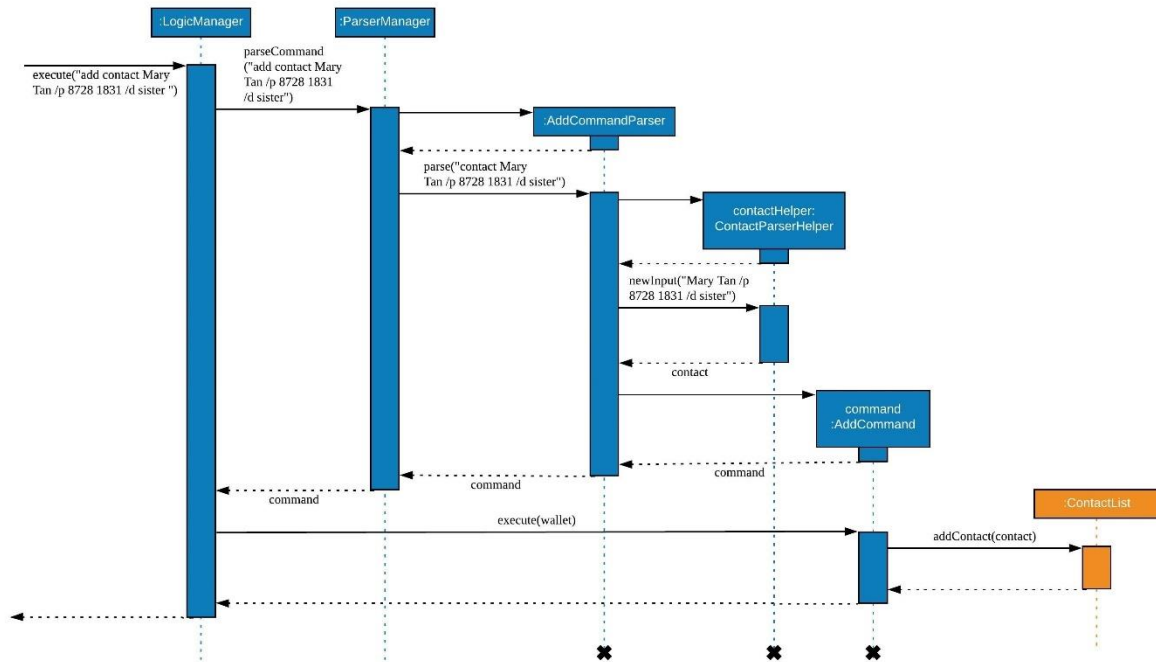




Figure 5.3.2.1 Sequence diagram for Add Contact

The steps below will explain the sequence diagram shown above:

1. At the Logic Component, AddCommandParser will parse the command and calls ContactParserHelper which processes the input into a contact object with Name, Phone Number, and Details according to the following rules:
 - a. Name must not be empty. In this case, it would be Mary Tan.
 - b. Process any arguments after /p and /d into Phone Number and Details respectively. In this case, it would be 8728 1831 and sister respectively processed.
 - c. If there is no /p, Phone Number will be set as a null value. The null value also applies if /p exists but does not have any arguments.
 - d. If there is no /d, Details will be set as a null value. The null value also applies if /d exists but does not have any arguments.
2. If the input is successfully processed, a command object which includes the contact object is returned to LogicManager.
3. LogicManager executes the command object, which adds the contact into ContactList.

	<p>At Step 2a, if Name is empty, the steps after will not be executed and an error message will display.</p>
	<p>Users can include spaces in their command arguments and the optional command line arguments (/d and /p) do not need to be entered in a particular order. ContactParserHelper will process the input accordingly.</p>

5.3.3 Editing Contact Implementation

To edit contact entries in the application, users can use `edit contact <ID> /n <Name> /p <Phone Number> /d <Details>` command. The following sequence diagram shows the implementation. The command used in the diagram is `edit contact 6 /n John /p 81007183 /d brother 123@abc.com`.

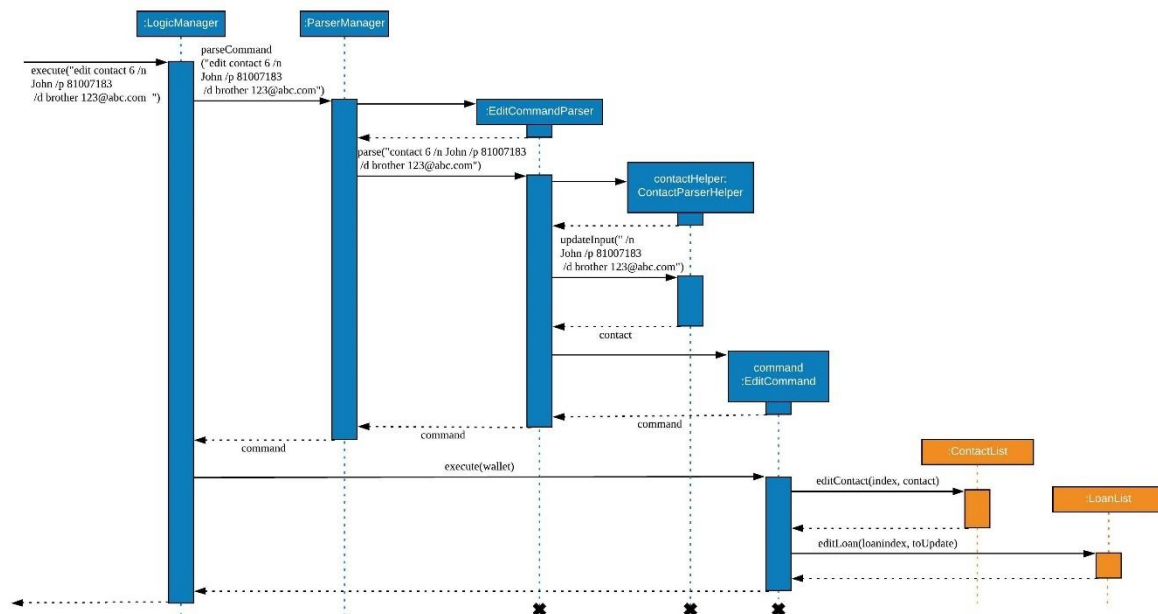





Figure 5.3.3.1 Sequence diagram for Edit Contact

The steps below will explain the sequence diagram shown above:

1. At the Logic Component, `EditCommandParser` will parse the command and calls `ContactParserHelper` which processes the input into a contact object with Name, Phone Number, and Details according to the following rules:
 - a. If there is no `/n` or no arguments for `/n`, the original Name value in the contact entry will be retained.
 - b. If there are no arguments for `/p`, Phone Number will be reset into a null value. If there is no `/p`, the original Phone Number value in the contact entry will be retained.
 - c. If there are no arguments for `/d`, Details will be reset into a null value. If there is no `/d`, the original Details value in the contact entry will be retained.
3. If the input is successfully processed, a command object which includes the contact object is returned to `LogicManager`.

4. LogicManager executes the command object, which replaces a contact entry in ContactList with the contact object according to the corresponding element index, index, that is retrieved via the ID.
5. Executing the command object also updates any loans that use the edited contact. If a loan entry uses the edited contact, a loan object, toUpdate is created. The contact object as well as other existing variable values from the loan entry, will be included in toUpdate. The loan entry in LoanList will be replaced by the toUpdate object, according to the corresponding element index, loanIndex, which is retrieved based on the ID of edited contact.

	<p>At Step 2, if no command line options are provided or detected, the contact entry is assumed to not require any edits. Any steps after Step 2 will not be executed to prevent unnecessary edits to files and error message will display.</p>
	<p>Users can include spaces in their command arguments and the optional command line arguments (/d, /n and /p) do not need to be entered in a particular order. ContactParserHelper will process the input accordingly.</p>
	<p>At Step 4, error messages will display if the ID is invalid and no contact or loan entry will be updated.</p>

5.3.4 Delete Contact Implementation

To delete contact entries, users can use `delete contact <ID>` command. The following sequence diagram shows the implementation. ID is assumed to be the value of 2.

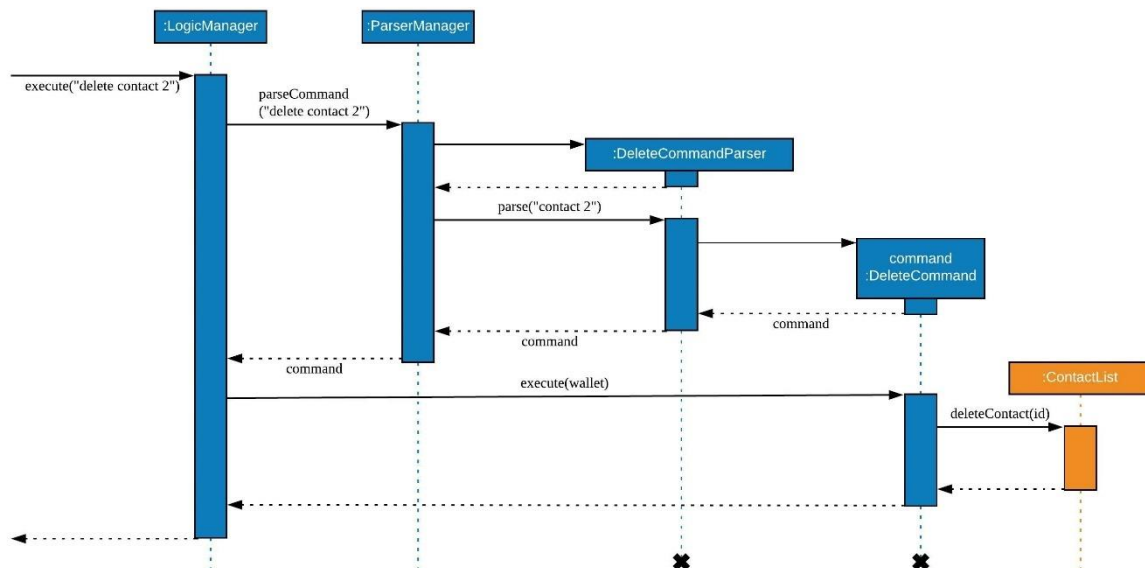




Figure 5.3.4.1 Sequence diagram for Delete Contact

The steps below will explain the sequence diagram shown above:

1. At the Logic Component, `DeleteCommandParser` will parse the command and check if the input is valid.
2. If the input is valid, a command object is returned to `LogicManager`.

3. LogicManager executes the command object, which first check if any existing loans are using the contact entry that user requests for deletion. If there are no existing loans using the contact entry, the contact will be deleted from ContactList based on ID indicated by the user.

	At Step 3, if existing loans are using contact entry, an error message will display.
	At Step 3, error messages will display if the ID is invalid and no contact entry will be deleted.

5.3.5 Design Consideration

Aspect: How data is removed from optional fields (i.e. Phone and Details)

- **Alternative 1: Provide /r option to reset phone(/p) or details(/d) into empty values e.g. edit contact 3 /p /r /d /r**

Pros: A specific command option is explicitly used for resetting fields. Easy to implement on other commands in the application to reset variable values.

Cons: Additional code implementation of one more command option required.
EditCommandParser will need to check if /r is placed at the right part of the command.

- **Alternative 2: Use /p and /d without arguments to reset phone or details into empty values (current implementation)**

Pros: Easy to implement by checking if the option is followed by an empty string (i.e. no argument).

Cons: User or program testers may accidentally reset fields into empty values if they forget to key in the arguments for the /p or /d options.

5.3.6 Future Implementation

These are features considered for future implementation:

1. Merge duplicate contacts.
2. Create more fields for the contact entry, e.g. email field, address field.

5.4 Budget Management

Budget Management involves mainly the interaction between the users and their budgets.


The section below will describe in detail the Current Implementation, Design Considerations and Future Implementation of the Budget Management.

5.4.1 Current Implementation

Users are able to interact with the budget management system via these commands:

- budget
 - Specifies the budget amount with a specific month and year.

- view
 - View budget set for a specified month and year.

	The budget command requires 2 parameters, amount and month/year, ie budget \$400 01/2019
	The view command requires 2 parameters, budget and month/year, ie view budget 01/2019

Upon invoking the `budget` command with valid parameters (refer to [UserGuide.pdf](#) for view usage), a sequence of events is executed. For clarity, the sequence of events will be in reference to the execution of a `budget $400 01/2019` command. A graphical representation is also included in the Sequence Diagram below for your reference when following through the sequence of events. The sequence of events is as follows:

1. Firstly, the `budget $400 01/2019` command is passed into the `execute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `parseCommand` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parse` function of the appropriate parser for the `budget` command which in this case, is `SetBudgetParser`.
4. After parsing is done, `SetBudgetParser` would instantiate the `SetBudgetCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `execute` function of the returned `SetBudgetCommand` object.
6. In the `execute` function of the `SetBudgetCommand` object, data will be retrieved from the `Model` component (i.e. retrieving data from the current `BudgetList` and `Budget`).
7. Now that the `SetBudgetCommand` object has the data of the budget list and the input budget, it is able to check the following conditions:
 - a. If budget amount is less than \$0
 - b. If budget amount is \$0
 - c. If budget amount is more than \$0
8. If the budget amount is a negative value, `SetBudgetCommand` object will return an error to `LogicManager`, which then returns this to the `UI` component and display the error content to the user. For this case, the displayed result will show that a negative budget amount is not accepted.
9. If the budget amount is zero fulfilled, `SetBudgetCommand` object will check for an existing budget with the same month and year and remove the entry, before sending a message to the `UI` component.

10. When the message is returned to the UI component, the UI component will display the content to the user. For this case, the displayed result will therefore be You have successfully removed your budget for January 2019 or There is no budget for removal depending on whether an existing budget with the same month and year can be found.
11. If the budget amount is a positive value, SetBudgetCommand object will check for an existing budget with the same month and year and remove the entry, before sending a message to the UI component.
12. When the message is returned to the UI component, the UI component will display the content to the user. For this case, the displayed result will therefore be You have successfully edited your budget for January 2019. or 400.00 dollars is the budget set for January 2019 depending on whether an existing budget with the same month and year can be found.

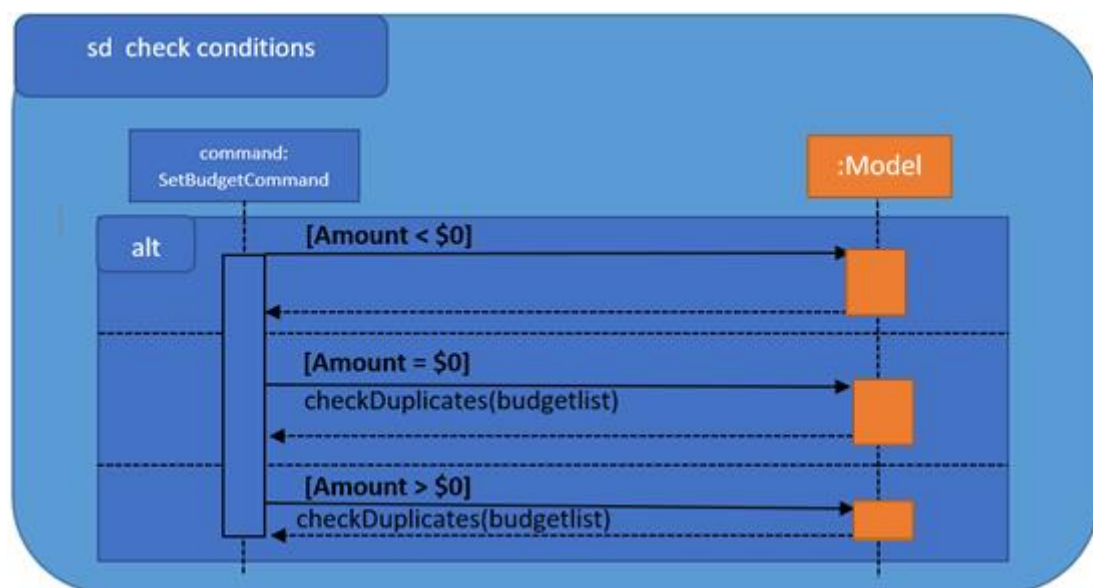
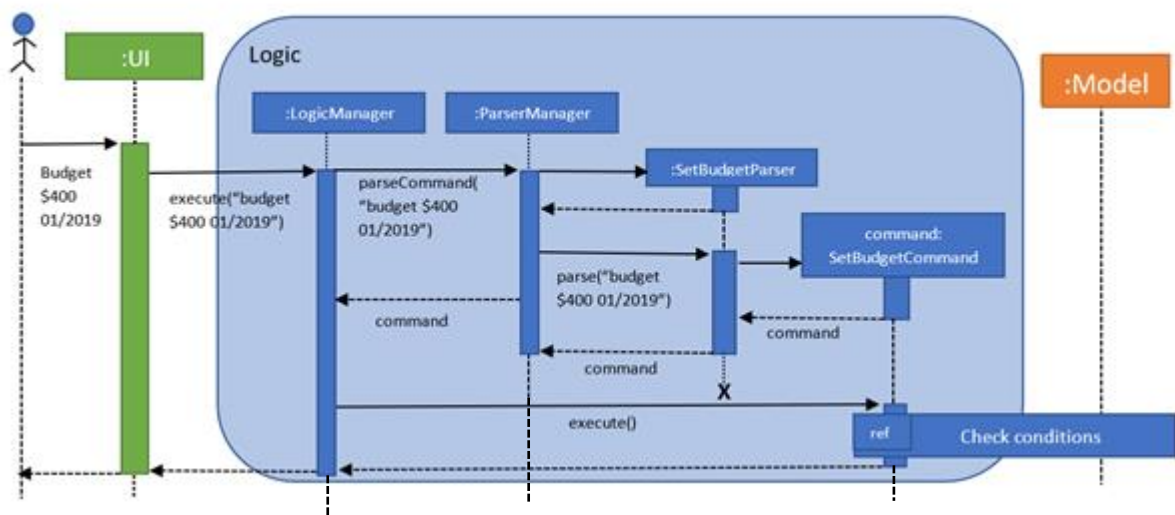


Figure 5.4.1.1a and 5.4.1.1b Sequence diagram of setting Budget

In addition to just adding a budget using the budget command, users can remove the budget by setting the budget amount for the particular month/year to \$0. For example, budget \$0 01/2019 will remove the budget for January 2019. Furthermore, a budget can be edited or overwritten when users add a budget that contains the same month and year of an existing budget.

5.4.2 Design Consideration

Aspect: Deleting/Removing budget

- **Alternative 1 (Current Selection): SetBudgetCommand to handle removal of existing budgets**

Pros: Straightforward, a budget \$0 month/year does not overlap with the actual expenditure budget and yet retains its intended function.

Cons: Easy to forget that such a feature exists.

- **Alternative 2: Delete command to handle removal of existing budgets**

Pros: Since the delete command is used to delete *data*, it will be natural for users to use delete to remove the budget as well.

Cons: delete command uses index from *data* id, budget does not use id, hence adding more confusion.

5.4.3 Future Implementation

Though the current implementation has much flexibility, there is more that can be done to elevate user experience to the next level. These are some possible enhancements:

1. A user can receive notifications whenever they enter an expense that will result in the monthly expense being too close or exceeding the stated budget.

This way, the user will think twice for future expenditures or set a lower budget for the next month to make up for the overspending.

2. Budget now extends to daily or yearly or even weekly expenses.

Provides more flexibility for users and enhancing their overall experience, since they are now able to further micromanage their expenses.

5.5 Auto Reminders

5.5.1 Current Implementation

Users can interact with the Reminder System via these commands:

- `reminder on`
 - To turn on auto reminders which remind users of existing loans that have not been settled.
 - However, if this command were to be invoked when the auto reminder system has already been called before, WalletCLi will tell the user that reminders have already been turned on.
 - In the event where there are no more unsettled loans and if the user tries to invoke this command, auto reminders will be turned off automatically as there is no need for reminders.
 - `reminder off`
 - To turn off auto reminders.
 - However, if this command were to be invoked when the auto reminder system has already been turned off, WalletCLi will tell the user that reminders have already been turned off.
 - `reminder set <TIME IN SECONDS>`
 - To turn on auto reminders which remind users of existing loans that have not been settled.
1. Upon booting up WalletCLi, the Main object will construct a LogicManager object.
 2. The LogicManager object then constructs a Reminder object.
 3. Afterwards, the Main object will invoke `getReminder()` method in the LogicManager object which returns a Reminder object back to Main.
 4. Main then invokes `autoRemindStart()` method in the Reminder object which constructs a MyThread object.
 5. `printUnsettledLoans()` method is then invoked by the ReminderThread object.

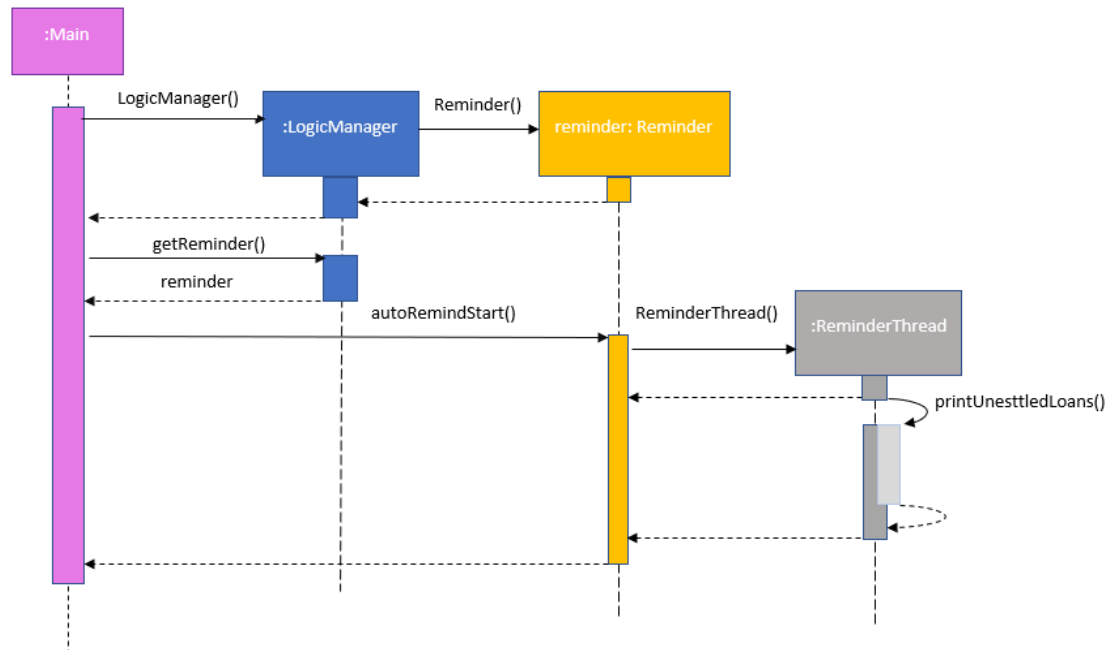


Figure 5.5.1.1 Sequence Diagram of Auto Reminder system at boot of WalletCLi.

5.5.2 Design Consideration

Aspect: Managing Reminders

- **Alternative 1: Reminders are only set to remind users at the start of the program**

Pros: Easy to implement as it will only be called once at the start.

Cons: However, this defeats the purpose of having a reminder system in the first place. If it is only called once at the start, it is an insignificant feature.

- **Alternative 2: Reminders are set to remind users at timed intervals.**

Pros: Provides more customisability to reminders and does not get affected by the parent process.

Cons: The child process will print all unsettled loans even if the user is still typing in the command halfway, disrupting user input.

5.6 Stats

Stats involves reflecting visualised detailed data for the users.

The section below will describe in detail the Current Implementation, Design Considerations and Future Implementation of the Stats system.

5.6.1 Current Implementation

Users are able to interact with the stats system via this command:

- View stats
 - View visualised data for expenses

Upon invoking the `view stats` command, a sequence of events is executed. The sequence of events is as follows:

1. Firstly, the `view` command is passed into the `execute` function of `LogicManager` to be parsed.
2. `LogicManager` then invokes the `parseCommand` function of `ParserManager`.
3. `ParserManager` in turn invokes the `parse` function of the appropriate parser for the `budget` command which in this case, is `ViewCommandParser`.
4. After parsing is done, `ViewCommandParser` would instantiate the `ViewCommand` object which would be returned to the `LogicManager`.
5. `LogicManager` is then able to invoke the `execute` function of the returned `ViewCommand` object.
6. In the `execute` function of the `ViewCommand` object, expenses will be retrieved from the `Model` component
7. Now that the `ViewCommand` object has information of the expenses list, it requests the instantiated `UI` object to invoke the method `drawStats()`
8. `drawStats()` method starts a thread to start drawing pie charts and bar graphs based on current data on the `Expenses List`.

5.6.2 Design Consideration

Aspect: Main stats function

- **Alternative 1: Main stats function to be done in the ViewCommand**

Pros: Reduce the complexity of `ViewCommand`.

Cons: The generation of stats takes a long time with more expenses, so there will be a threshold that will result in the `execute` function to not return false, which will then cause an error message to appear in the UI.

- **Alternative 2 (Current Selection): Use a separate thread to do the main statistic functions**

Pros: No error will be generated given high volume of expenses

Cons: Race condition will occur in the UI if users give another set of commands that will generate UI.

5.6.3 Future Implementation

Though the current implementation has much flexibility, there is more that can be done to elevate user experience to the next level. These are some possible enhancements:

1. A user can export the stats on a .txt or .pdf file

This way, the user can better keep track of their expenses, as they can export and save their data daily, monthly or yearly.

5.7 Help

The `help` command aims to help users to understand the different command syntaxes and usages. All commands are being grouped based on the program feature they are related to. Each program feature has its own help section. The user can view the list of available sections and their indexes through the `help` command. When the user executes `help <SECTION INDEX>` command in the application, **WalletCLI** will display the content of the help section, based on the section index indicated in `<SECTION INDEX>`.

To provide an easier way for developers to update the help sections, each section has its own content stored in a text file under `/src/main/resources`. **WalletCLI** will load all content from the text files into in-app memory at the start of the application by referencing the file names from `/src/main/resources/helppaths.txt`.

Compiling the application to jar file via Gradle should add all files under `/src/main/resources` into the jar by default. The following table shows the current list of in-app help section indexes and names against the text file it will retrieve. This information is also stored under `/src/main/resources/helppaths.txt`.

Index	Section Name	Text File Name
1	General	general.txt
2	Expense	expense.txt
3	Loans	loan.txt
4	Contacts	contact.txt
5	Command History	cmdhistory.txt

For each command in the text files, most of the content is formatted with `[field] [value]` format, delimited by pipeline character (`|`). The following shows the content and formatting required for each command in the text files. Currently, there isn't any admin console to update these files yet.

1. Header

- a. Format: `--[Command Header]--`

- b. Example:

`--Add Loan--`

2. Command Syntax

- a. Format: `command | [syntax]`

- b. Example:

`command | add loan <DETAILS> <AMOUNT> [DATE] </1 OR /b>`

3. Description of Command
 - a. Format: desc | [description]
 - b. Example:
desc | add new loan entry
4. (Optional) For parameters which need further explanation or need a specific format
 - a. Format: [parameter] | [explanation or format]
 - b. Example:
DATE | dd/mm/yyyy
 - i. /l | use this if entry indicates the amount you lend
 - ii. /b | use this if entry indicates the amount you borrow
 - c. Note: Insert these contents between Command Syntax and Description of Command

Pipeline character are used to split the fields from its value when the application prints out the help section. Hence, do not include additional pipeline characters in any fields or values.

5.7.1 Current Code Implementation

The section describes how the in-app help section is implemented in terms of code. The command used in the diagram is `help 5` which currently retrieve commands related to Command History.

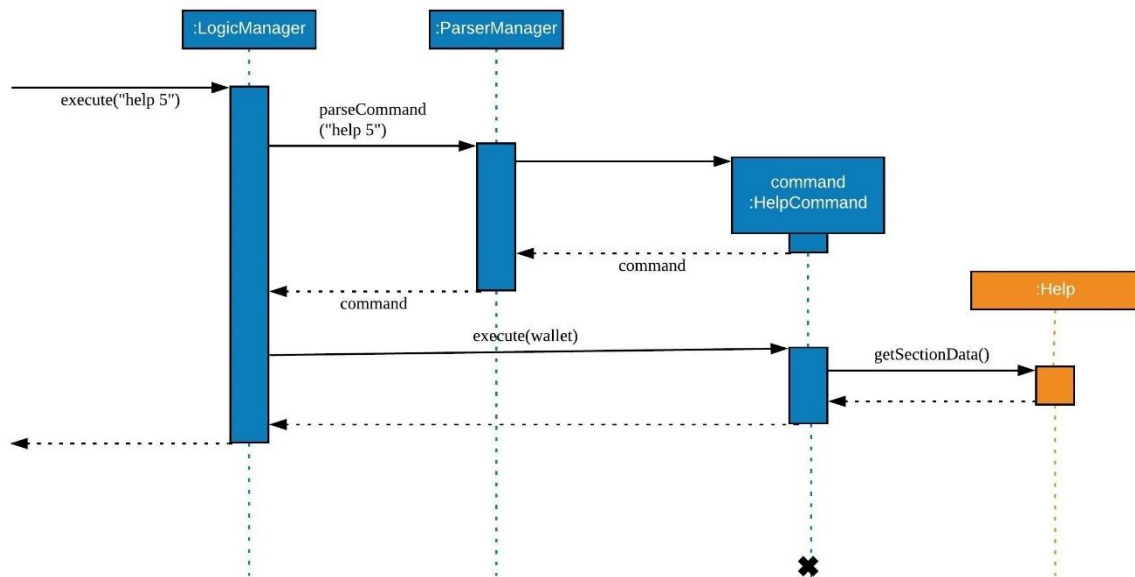


Figure 5.7.1.1 Sequence diagram for Help

The main flow of the help section is implemented according to the sequence diagram shown above. Note that there is no parser for the help command. The steps below further explain the sequence diagram shown above:

1. At the Logic Component, `LogicManager` will parse the command with `ParserManager` and check if the command is valid.

2. If the command is valid, a command object is returned to LogicManager.
3. When LogicManager executes the command object, **WalletCLI** will check if the user input is a valid help section index.
4. If it is a valid help section index, **WalletCLI** will retrieve and display the corresponding section content from the in-app memory through `getSectionData()` function.
 - a. Each help section is stored as a `Help` object in the application memory, where each object consists of a section name and the section content read from their corresponding text file.



At Step 4, if the user inputs an invalid index, an error message will display.

5.7.2 Adding New Sections

`/src/main/resources/helppaths.txt` will need to be updated if new text files (i.e. help sections) are added into `/src/main/resources`. Type in the new section name and relative path in the following format: [Section Name], [Relative Path].

Example: Command History, `/cmdhistory.txt`

5.7.3 Design Consideration

Aspect: Storage of Help Section content

- **Alternative 1: Store all command syntaxes and examples into a single file.**

Pros: At the start of the application, program only need to retrieve a single file and process all the commands into in-app memory.

Cons: The size of file may grow very big if more commands are added to the file in the future, which causes the editing of the file or finding one command to be harder.

- **Alternative 2: Group similar command syntaxes and examples into their own file (i.e. section). (current implementation)**

Pros: When updating syntaxes, it is easier to find the file where a command is stored, since developers can filter out the sections that the command does not belong to.

Cons: Planning of groups is required such that all commands can have suitable section to be added to. Program need to retrieve more files and process them into in-app memory.

5.7.4 Future Implementation

These are features considered for future implementation:

1. Admin console to update content in help section.
2. Provide additional option for user to get help for a specific command instead of finding and reading off a section.

5.8 Managing Expenses

Expenses are used to allow users to record how much they spend and to keep track of their monthly budgeting.

5.8.1 Listing Expense

To view expenses in a table form, users can use `list expense` command.

5.8.2 Adding Expense Implementation

To add expense entries into the application, users can use `add expense <DESCRIPTION> $<AMOUNT> <CATEGORY> [/on <DATE>] [/r <RECURRENCE RATE>]` command.

Given below is an example usage scenario when adding an expense. The example command used here is `add expense Lunch $8 Food /on 10/10/2019`.

1. The user executes the command `add expense Lunch $8 Food /on 10/10/2019`.
2. The add command parses the user input using the `AddCommandParser`, then returns an `AddCommand` object to `LogicManager` to invoke the `execute` function. This adds a new `Expense` object into the `ExpenseList`.

The following sequence diagram shows how the add command works:

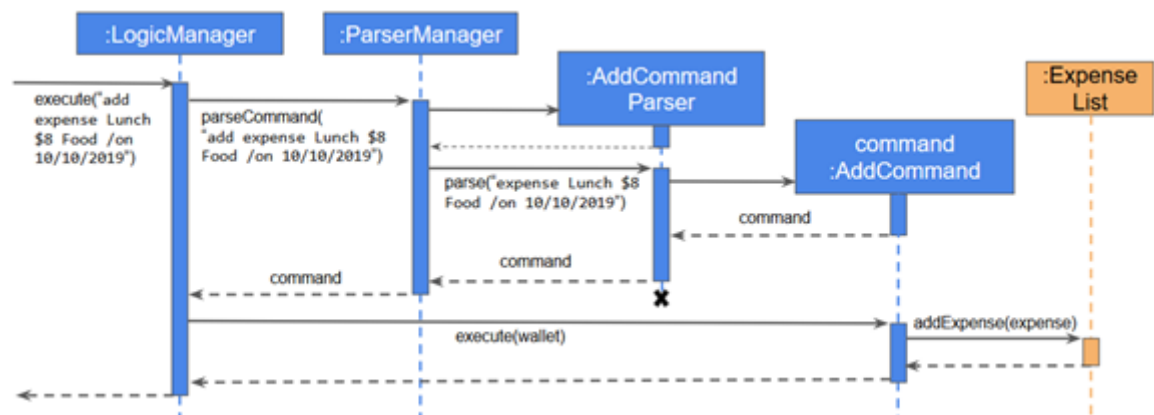


Figure 5.8.2.1 Sequence diagram when adding an expense

5.8.3 Editing Expense Implementation

To edit an expense entry in the application, users can use `edit expense <EXPENSE ID> [/d <DESCRIPTION>] [/a <AMOUNT>] [/c <CATEGORY>] [/t <DATE>] [/r <RECURRENCE RATE>]` command.

Given below is an example usage scenario when editing an expense. The example command used here is `edit expense 1 /d Dinner /a 4.5 /c Food /t 12/10/2019`.

1. The user executes the command `edit expense 1 /d Dinner /a 4.5 /c Food /t 12/10/2019`.
2. The edit command parses the user input using the `EditCommandParser`, then returns an `EditCommand` object to `LogicManager` to invoke the `execute` function.
3. This edits the `Expense` object with the new values given by the user in the `ExpenseList`.

The following sequence diagram shows how the edit command works:

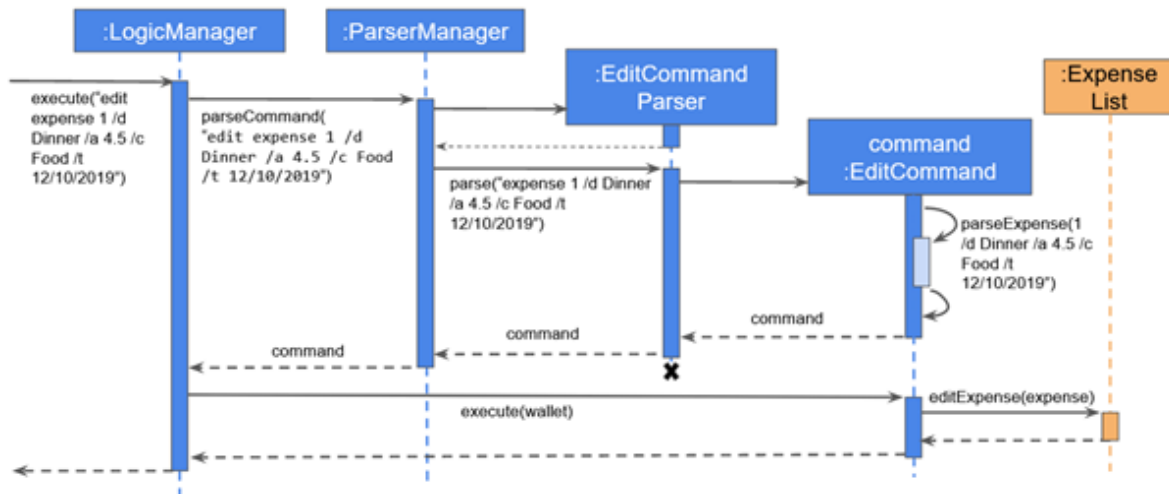


Figure 5.8.3.1 Sequence diagram when editing an expense

5.8.4 Delete Expense Implementation

To delete an expense entry, user can use `delete expense <EXPENSE ID>` command.

Given below is an example usage scenario when deleting an expense. The example command used here is `delete expense 1`.

1. The user executes the command `delete expense 1`.
2. The delete command parses the user input using the `DeleteCommandParser`, then returns a `DeleteCommand` object to `LogicManager` to invoke the `execute` function.
3. This deletes the `Expense` object with the given `EXPENSE ID` in the `ExpenseList`.

The following sequence diagram shows how the delete command works:

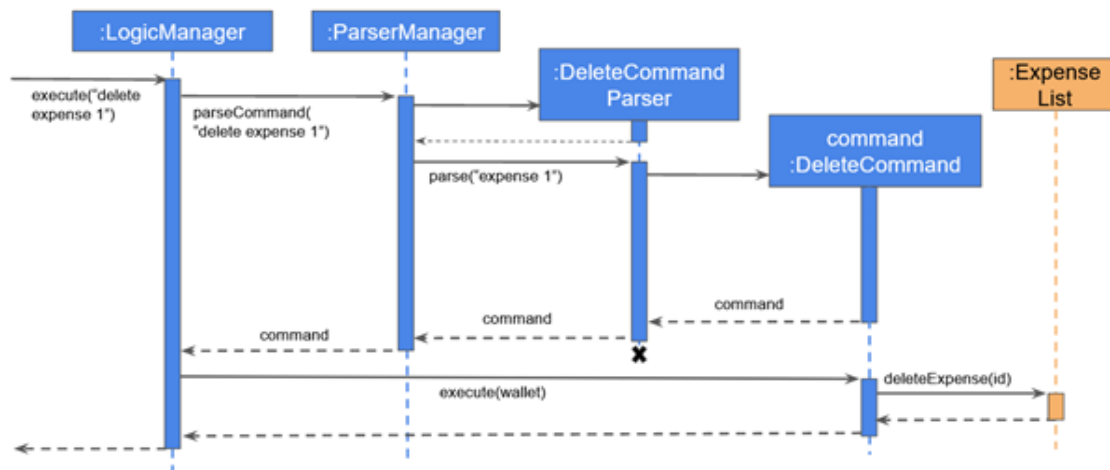


Figure 5.8.4.1 Sequence diagram when deleting an expense

5.8.5 Future Implementation

These are features considered for future implementation:

- Allow for more flexible syntax when using the commands, input parameters do not need to be in a specific order.
- Parse different formats of date for user input.

5.9 Export Data

The section explains the code implementation of exporting data to csv files in **WalletCLI**. The command for exporting expenses is `export expense <MONTH/YEAR>`. The command for exporting loans is `export loan`. To see how data is formatted in csv files, see the Export section in User Guide.

5.9.1 Current Implementation

The following sequence diagram shows the main flow of exporting data. Example command used here is `export expense 10/2019`.

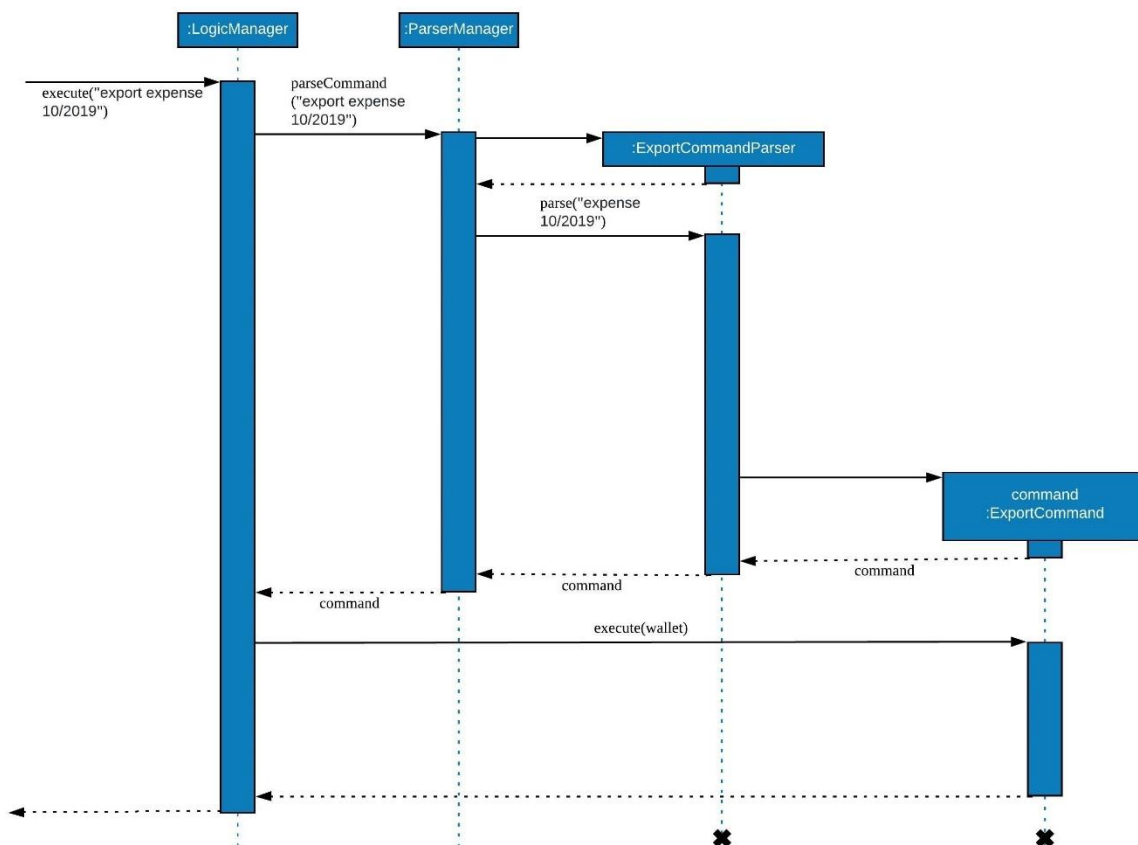


Figure 5.X.1 Sequence diagram for Export Data

The steps below further explain the sequence diagram and logic of code:

1. At the Logic Component, `ExportCommandParser` will parse the command and extracts the type of data user wants to export (either `expense` or `loan`) and stores it into a variable called `type`.
2. `ExportCommandParser` will retrieve either the expense or loan records in the application based on the type of data requested. In this case, it will retrieve expense records from the month of October 2019.

3. A list, known as `data`, is created. `ExportCommandParser` will save the retrieved records into `data`.
 - a. If user requests for expenses, the month requested and budget values (if any) are separately added into `data`. In addition, `ExportCommandParser` will sum the total amount spent from the records retrieved in Step 2 and add the sum into `data`.
4. `ExportCommandParser` returns a `command` object to the `LogicManager`. In the `command` object, `data` and `type` are stored.
5. `LogicManager` executes the `command` object.
6. A new csv file will be created in the directory where the jar file of **WalletCLI** is running from. `data` will be written into the csv file using `CSVWriter.writeAll()` method from a third party library, `OpenCSV`. The file name will include the data type as well as the date and time at which the data is being exported. E.g. `walletCLI-expenses-08.11.2019-10.11.33.csv`



At Step 6, error messages will display if files can't be written and the process will be terminated.

5.9.2 Design Consideration

Aspect: How files are being named

- **Alternative 1: Use an id to name an exported file. E.g. `WalletCLI-expenses-id000001.csv`**

Pros: Users can differentiate the files being exported to their **WalletCLI** home directory. This can prevent `OpenCSV` from overwriting files with identical names, since all filenames are now unique.

Cons: `ExportCommand` must always generate a unique id for the file. Id cannot be reused.

- **Alternative 2: Use date and time to name an exported file. E.g. `WalletCLI-expenses-08.11.2019-10.11.33.csv` (current implementation)**

Pros: Users can differentiate the files being exported to their **WalletCLI** home directory. Users can see the file creation date and time from the file name.

Cons: During code implementation, must ensure that the local date and time is correctly retrieved in order to name the file.

5.9.3 Future Implementation

These are features considered for future implementation:

1. Allow users to sort the loan or expense records before exporting them into csv files.
2. Provide an option to export data into xls files such that identical data can be merged into one cell when viewed in Microsoft Excel.

5.10 Import Data

The section explains the code implementation of importing data from csv files into **WalletCLI**. Command for importing records is `import <loan OR expense> <FILENAME>`. To know what formatting is required in the csv files before importing, see the Import section in User Guide.

5.10.1 Current Implementation

The following sequence diagram shows the main flow of importing data. Example command used here is `import loan importLoans.csv`.

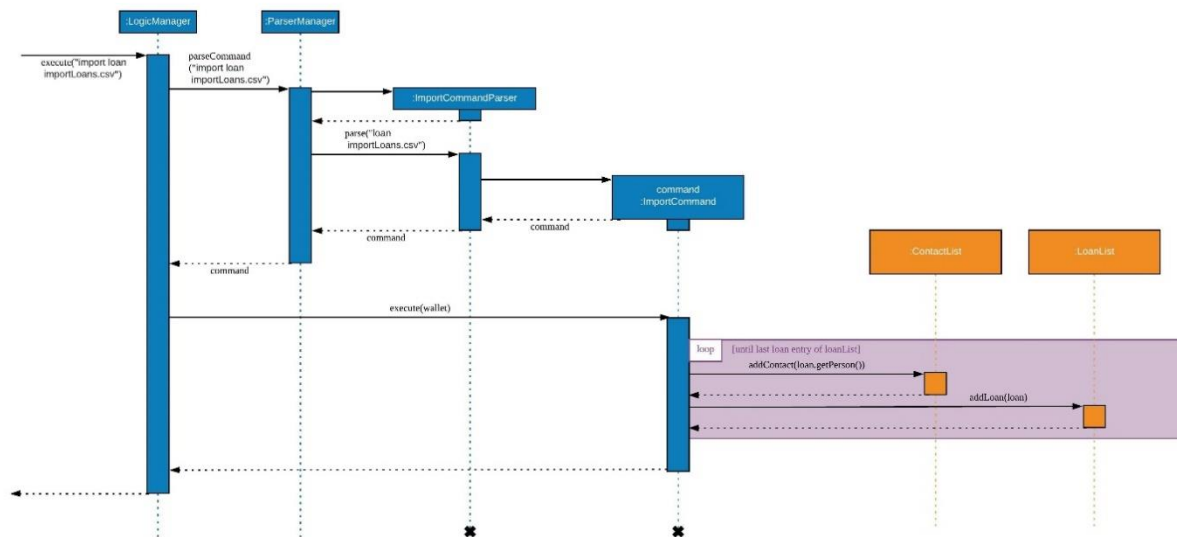


Figure 5.X.1 Sequence diagram for Import Data

The steps below further explain the sequence diagram and logic of code:

1. At the Logic Component, `ImportCommandParser` will parse the command to check if it is valid.
2. `ImportCommandParser` will attempt to find the file based on the `FILENAME` provided by user by searching for it in the directory from which the application is running. In this case, it would be `importLoans.csv`.
3. If file is found, using `CSVReader` from a third-party library, `OpenCSV`, `ImportCommandParser` will read each record from each row of the file and checks if it is formatted correctly.
4. `ImportCommandParser` will save the retrieved records into a list, called `loanList`.
5. `ImportCommandParser` returns a `command` object to the `LogicManager`. In the `command` object, `loanList` is stored.
6. `LogicManager` executes the `command` object.
7. As shown in the diagram above, new contacts and loans will be created from `loanList`. The new records are added to `ContactList` and `LoanList`, which stores the contact and loan data respectively. In the event if the data type was expenses instead, new expenses will be added to `ExpenseList` instead.



At Step 2 and 3, error messages will display if the file can't be read or if the records in the file is formatted incorrectly. The process will be terminated and steps after will not be executed.

5.10.2 Design Consideration

Aspect: File Format Requirements

- **Alternative 1: Use json file to import data.**

Pros: Users do not need to worry about how the variables are being ordered in each record.

Cons: Json is not a commonly used file format. Third-party library required to interpret the syntax. User required to repeatedly key in the variable names for each record.

- **Alternative 2: Use csv file to import data. (current implementation)**

Pros: Users can use popular and commonly used spreadsheet software e.g. Microsoft Excel, Google Sheets to easily build their csv files.

Cons: Third-party library required to interpret the syntax. Order of variables is important. As csv file require a character to separate the variable values in each row, this separator character cannot be used in data fields. User required to leave input for optional variables as blank if entry does not require the optional field. E.g. Consecutive commas in "value 1,,value3" is required to signal to the application that the user does not need the optional variable in the middle.

5.10.3 Future Implementation

These are features considered for future implementation:

1. Merge identical contacts when importing the list of loans. Currently, a new contact is created from each loan entry in the csv file.
2. Allow users to choose an existing contact from the application to tag a loan entry that is read from the csv file.
3. Allow users to import a list of budget values that they set for each month into the application.

Appendix A - Product Scope

Target user profile:

- NUS students
- has a need to organise and manage a significant number of expenses
- prefers desktop apps over other types
- prefers having a completely offline wallet application
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage expenses, budgets and loans faster than a typical mouse/GUI driven app and caters to users who prefer an offline solution due to the current technology climate where information privacy/data protection has become an uncertainty.

Appendix B - User Stories

This section describes the user stories that the **WalletCLI** developer team ideated. These user stories were used to decide on the desired features for **WalletCLI**. The user stories are categorized into different priorities for implementation:

- High (must have) - ***
- Medium (nice to have) - **
- Low (unlikely to have) - *

Priority	As a ...	I can ...	So that I ...
***	user	Add loans by specifying the amount and contact	Can know who owes me money
***	Forgetful user	Update status of loans by indicating it is being settled	Know which loans are settled
***	user	Delete entries from the list of loans	Can remove unwanted or old entries from the list
***	user	Record my expenses	Can track my daily expenses
***	user	Delete my expenses record	Can remove records that were added by mistake
***	Organised user	See my expenses for each category	Can plan my budget for each expense category
***	user	Set category expenses	Know which category I have spent most of my money on
***	user	View my expenses according to category	Can know how much I spend on each category per month
***	Power user	Export my expenses record	Can keep a record of my expenses
***	user	Update my expenses record	Can consistently keep track of my money
***	user	Add contacts	Can keep track of their personal information
***	user	Edit my contacts	Can update any changes to their information
***	user	Delete my contacts	Can remove contacts that were added by mistake or no longer in use
***	New user	View the help section	Learn how to use the expenses app in command line interface

***	user	View the total sum of money that I have	Can track how much money I have left in my wallet
***	user	view colour coded categories	Can easily view the different types of categories
***	user	Choose to view my entire expenses and loans or view specific loans and expenses by date	Can view my expenses and loans history
**	Forgetful user	Set reminders to pay my bills	Will not forget to pay the bills
**	Lazy user	Add recurring expenses	Can have expenses added monthly/daily/yearly automatically on the application
**	user	Use the app to take photos of receipts	Can input my expenses later
**	user	Start new sessions of 'WallerCLI' that have their own saved states	Can keep track of each one individually and differently
**	user	Convert currency	Do not need to calculate it manually
**	user	Set expenses goals by specifying how much I want to spend per week or month	Will not overspend
**	user	Redo command	Can add multiple of the same entry
**	user	Undo command	Can go back to the previous state if there are mistakes.
**	user	View command history	Can know the previously executed commands.
**	user	Export Data	Can share data with other people
**	user	Import Data	Can import existing records into application
*	user	Receive cashback when I save money	Can lower my expenses
*	user	Show off to my friends how much I saved per month on social media	Can encourage people to do so too
*	user	Get notifications from the application	Can be notified of important expenses or if i don't have much money left
*	user	Sync my record to my bank account	Can have more convenience

*	user	Make direct transactions when shopping online	Do not need to actively use a card for online transaction
---	------	---	---

Appendix C - Use Cases

This section describes the Use Cases for some of our implemented features. (For all use cases below, the System is `walletCLI` and the Actor is the user unless specified otherwise)

Use Case 1: Adding an expense

- **MSS:**
 1. User inputs `add expense` command with all required parameters.
 2. System adds the expense into the expense list.

Use case ends.
- **Extensions:**
 - 1a. System detects parameters in the wrong order.
 - 1a1. System outputs an error message.

Use case ends.
 - 1b. System detects missing required parameters in the given input.
 - 1b1. System outputs an error message.

Use case ends.

Use Case 2: Editing an expense

- **MSS:**
 1. User inputs `edit expense` command with ID of expense and new values.
 2. System updates the expense in the expense list.
 3. System outputs the edited expense with the updated values.

Use case ends.
- **Extensions:**
 - 1a. System detects ID of expense is invalid.
 - 1a1. System outputs an error message.

Use case ends.
 - 1b. System detects input parameters in the wrong order.
 - 1b1. System outputs an error message.

Use case ends.

1c. System detects no parameters in the given input

1c1. System outputs an error message.

Use case ends.

Use Case 3: Deleting an expense

- **MSS:**

1. User inputs `delete expense` command with the ID of expense to delete.

2. System deletes the expense and updates the expense list.

3. System outputs the deleted expense with its values.

Use case ends.

- **Extensions:**

1a. System detects ID of expense does not exist.

1a1. System outputs an error message.

Use case ends.

1b. System detects no parameters in the given input.

1b1. System outputs an error message.

Use case ends.

Use Case 4: Listing all expenses

- **MSS:**

1. User inputs `list expense` command.

2. System outputs all expenses along with their values.

Use case ends.

- **Extensions:**

1a. System detects input parameters are in the wrong order.

1a1. System outputs an error message.

Use case ends.

1b. System detects no parameters in the given input.

1b1. System outputs an error message.

Use case ends.

Use Case 5: Setting a budget for the month

- **MSS:**
 1. User inputs budget command with the required parameters.
 2. System sets and updates the budget for the given month and outputs the new budget for the month.

Use case ends.
- **Extensions:**
 - 1a. System detects input parameters are in the wrong order.
 - 1a1. System outputs an error message.

Use case ends.
 - 1b. System detects no parameters in the given input.
 - 1b1. System outputs an error message.

Use case ends.

Use Case 6: Adding a loan

- **MSS:**
 1. User inputs add loan command with all required parameters.
 2. System adds the loan into the loan list.

Use case ends.
- **Extensions:**
 - 1a. System detects input parameters are in the wrong order.
 - 1a1. System outputs an error message.

Use case ends.
 - 1b. System detects missing required parameters in the given input.
 - 1b1. System outputs an error message.

Use case ends.

Use Case 7: Editing a loan

- **MSS:**
 1. User inputs edit loan command with ID of loan and new values as parameters.
 2. System modifies and updates the loan in the loan list.
 3. System outputs the edited loan with the updated values.

Use case ends.
- **Extensions:**

- 1a. System detects ID of loan does not exist.
 - 1a1. System outputs an error message.Use case ends.
- 1b. System detects input parameters are in the wrong order.
 - 1b1. System outputs an error message.Use case ends.
- 1c. System detects no parameters in the given input.
 - 1c1. System outputs an error message.Use case ends.

Use Case 8: Deleting a loan

- **MSS:**
 - 1. User inputs `delete loan` command with the ID of loan to delete.
 - 2. System deletes the loan and updates the loan list.
 - 3. System outputs the deleted loan with its values.Use case ends.
- **Extensions:**
 - 1a. System detects ID of loan does not exist.
 - 1a1. System outputs an error message.Use case ends.
 - 1b. System detects no parameters in the given input.
 - 1b1. System outputs an error message.Use case ends.

Use Case 9: Listing all loans

- **MSS:**
 - 1. User inputs `list loan` command with the required parameters.
 - 2. System outputs all loans along with their values.Use case ends.
- **Extensions:**
 - 1a. System detects input parameters are in the wrong order.
 - 1a1. System outputs an error message.

Use case ends.

1b. System detects no parameters in the given input.

1b1. System outputs an error message.

Use case ends.

Use Case 10: Setting time for an auto reminder

- **MSS:**

1. User inputs `reminder set` command with the required parameters (eg. `Time-in-seconds`)

Example command: `reminder set 3600`

2. System outputs a string, indicating auto reminder is properly set with the appended value.

Use case ends.

- **Extensions:**

1a. System detects input parameters are in the wrong order.

1a1. System outputs an error message.

Use case ends.

1b. System detects no parameters in given input.

1b1. System outputs an error message.

Use case ends.

Use Case 11: Turning off the auto reminder

- **MSS:**

1. User inputs `reminder off` command.

2. System outputs a string, indicating auto reminder is turned off.

Use case ends.

- **Extensions:**

1a. System detects an error if auto reminder is already turned off.

1a1. System outputs an error message.

Use case ends.

Use Case 12: Turning on the auto reminder

- **MSS:**

1. User inputs `reminder on` command.

2. System outputs a string, indicating auto reminder is switched on.

Use case ends.

- **Extensions:**

1a. System detects an error if auto reminder is already turned on.

1a1. System outputs an error message.

Use case ends.

Use Case 13: Undo commands

- **MSS:**

1. User inputs undo command.

2. System outputs a message, indicating the command has been undone.

Use case ends.

- **Extensions:**

1a. System detects an error if there are no previous commands.

1a1. System outputs an error message.

Use case ends.

Use Case 14: Redo commands

- **MSS:**

1. User inputs redo command.

2. System outputs a string, indicating the command has been redone.

Use case ends.

- **Extensions:**

1a. System detects an error if there are no commands after the current state.

1a1. System outputs an error message.

Use case ends.

Use Case 15: View command history

- **MSS:**

1. User inputs history command.

2. System outputs a history of commands (buffer size: 10)

Use case ends.

Use Case 16: View specific data

- **MSS:**
 1. User inputs `view` command with a date as a parameter.
 2. System outputs loans and expenses that are related to that date

Use case ends.
- **Extensions:**
 - 1a. System detects an error if there are no parameters.
 - 1a1. System outputs an error message.

Use case ends.
 - 1b. System detects an error if the parameter is in the wrong format.
 - 1b1. System outputs an error message

Use case ends.

Use Case 17: View expenses statistics

- **MSS:**
 1. User inputs `view stats` command.
 2. System outputs visuals in the form of pie charts and bar graphs based on all expenses.

Use case ends.

Use Case 18: Add Contact

- **MSS:**
 1. User inputs `add contact` command with required parameters.
 2. System adds contact into contact list.

Use case ends.
- **Extensions:**
 - 1a. System detects an error if the parameters are in the wrong format.
 - 1a1. System outputs an error message.

Use case ends.

Use Case 19: Edit Contact

- **MSS:**
 1. User inputs `edit contact` command with ID of contact and new values as parameters.

2. System updates contact with new values in the contact list.

Use case ends.

- **Extensions:**

- 1a. System detects an error if the parameters are in the wrong format.

- 1a1. System outputs an error message.

Use case ends.

- 1b. System detects an error if the ID that user input is invalid.

- 1b1. System outputs an error message.

Use case ends.

Use Case 20: Delete Contact

- **MSS:**

1. User inputs `delete contact` command with ID of contact to be deleted.
2. System deletes contact from the contact list.

Use case ends.

- **Extensions:**

- 1a. System detects an error if existing loans are using the contact being requested for deletion.

- 1a1. System outputs an error message.

Use case ends.

- 1b. System detects an error if the ID that user input is invalid.

- 1b1. System outputs an error message.

Use case ends.

Use Case 21: List all contacts

- **MSS:**

1. User inputs `list contact` command.
2. System displays contacts from contact list.

Use case ends.

- **Extensions:**

- 1a. System detects an error if the parameters in the command is wrong.

- 1a1. System outputs an error message.

Use case ends.

Use Case 22: View list of help sections

- **MSS:**
 1. User inputs `help` command.
 2. System shows a list of available help sections

Use case ends.

Use Case 23: View help section

- **MSS:**
 1. User inputs `help` command with index of help section.
 2. System displays help section content.

Use case ends.
- **Extensions:**
 - 1a. System detects an error if the index that user input is invalid.
 - 1a1. System outputs an error message.

Use case ends.

Use Case 24: Export Data

- **MSS:**
 1. User inputs `export` command with the type of data they want to export.
 2. System exports data to a csv file created in user's **WalletCLI** home directory.

Use case ends.
- **Extensions:**
 - 1a. System detects an error if the parameters in the command is wrong.
 - 1a1. System outputs an error message.

Use case ends.
 - 2a. System detects an error if the csv file cannot be created.
 - 2a1. System outputs an error message.

Use case ends.

Use Case 25: Import Data

- **MSS:**

1. User inputs `import` command with the type of data they want to import and the filename where their data is stored.
2. System searches for the user's file in their **WalletCLI** home directory based on the filename.
3. System checks if the records are formatted correctly in the file.
4. System imports the records into application. Loan records will be imported into loan list. Expense Records will be imported into expense list.

Use case ends.

- **Extensions:**

- 1a. System detects an error if the parameters in the command is wrong.

- 1a1. System outputs an error message.

Use case ends.

- 2a. System detects an error if the file cannot be found.

- 2a1. System outputs an error message.

Use case ends.

- 3a. System detects an error if the records are in a wrong format.

- 2a1. System outputs an error message.

Use case ends.

Use Case 26: Generating Default Data

- **MSS:**

1. User inputs `generate` command
2. System deletes all existing storage files and replaces it with the default storage files and data.

Appendix D - Non-functional Requirements

This section describes the non-functional requirements of **WalletCLI**.

1. The application should work on any mainstream OS as long as it has Java 11 or higher installed.
2. The application should work on both 32-bit and 64-bit environments.
3. The application should be able to hold up to over a hundred entries of expenses, loans and contacts without a noticeable sluggishness in performance for typical usage.
4. 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.
5. The system should respond relatively quickly to user commands so as to not make the user wait around; this is an advantage of using **WalletCLI**.
6. The system should take up relatively little space on the local machine so as to cater to all users and OS.
7. The system should be easy to use, intuitive and simple, such that any student regardless of past experience with wallet/expenses application is able to use it.
8. The system should be flexible to allow all kinds of expenses that target users might have.
9. The data should be encrypted to prevent private data such as contact information from being accessed.
10. The application should work even without any Internet connection


Appendix E - Glossary

This section further explains some terms/words used in **WalletCLI**.

1. Data: expenses/loans/contacts that you might add to the application.
2. MSS: Main Success Scenario (MSS) describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong. This is also called the Basic Course of Action or the Main Flow of Events of a use case.
3. Use case: a specific situation in which a product or service could potentially be used.

Appendix F - Instructions for manual testing

Given below are instructions to test the app manually.

	These instructions only provide a starting point for testers to work on; testers are expected to do more <i>exploratory</i> testing.
---	--

F.1. Launch and Shutdown

1. Initial launch
 - a. Download the jar file and copy into an empty folder
 - b. Start the app by running the jar file with the following command:
`java -jar {jar file name}`
Expected: You should see the GUI appear in a few seconds. The window size may not be optimum.
 - c. To create default data, type in `generate` command

F.2. Adding an expense

1. Adding a non-recurring expense
 - a. Test case: add expense Lunch \$5 Food
Expected: New expense (Lunch with today's date) is added into the expense list. Details of the added expense shown in the output message.
 - b. Test case: add expense T-shirts \$25 Clothes /on 01/11/2019
Expected: Expense is **not** added into the expense list (Invalid category provided). Error details shown in the output message.
2. Adding a recurring expense
 - a. Test case: add expense Phone bill \$80 Bills /on 01/10/2019 /r monthly
Expected: New expense (Phone bill with date "01/10/2019") is added into the expense list. Details of the added expense shown in the output message. Another expense (Phone bill with date "01/11/2019") is also added into the expense list.

F.3. Editing an expense

1. Editing an expense

- a. Prerequisites: Populate expense list with sample data using `generate` command, then list all expenses using the `list expense` command.
- b. Test case: `edit expense 5 /d Dinner /t 10/10/2019 /a 10`
Expected: The expense with ID of 5 is edited with its description changed to "Dinner", date changed to "10/10/2019" and amount changed to "10". Details of the edited expense shown in the output message.
- c. Test case: `edit expense 8 /a 45`
Expected: The expense with ID of 8 is edited with its amount changed to "45". Details of the edited expense shown in the output message.

F.4. Deleting an expense

1. Deleting an expense

- a. Prerequisites: Populate expense list with sample data using `generate` command, then list all expenses using the `list expense` command.
- b. Test case: `delete expense 10`
Expected: The expense with ID of 10 is deleted from the expense list. Details of the deleted expense shown in the output message.

F.5. Adding a contact

1. Adding a contact

- a. Test case: `add contact`
Expected: No contact added. Error Message displays.
- b. Test case: `add contact /p 9728 1831 /d sister`
Expected: No contact added. Error Message displays.
- c. Test case: `add contact Mary Tan /d /p`
Expected: Contact with Mary Tan as name is added to contact list. Details of added contact shown in output message.
- d. Test case: `add contact Mary Tan`
Expected: Contact with Mary Tan as name is added to contact list. Details of added contact shown in output message.
- e. Test case: `add contact Mary Tan /p 9728 1831 /d sister`
Expected: Contact with Mary Tan as name, 9728 1831 as phone number and sister as details is added to contact list. Details of added contact shown in output message.
- f. Test case: `add contact Test /d /d`
Expected: Contact with Test as name and /d as details is added to contact list. Details of added contact shown in output message.

F.6. Editing a contact

1. Editing a contact

- a. Prerequisites: Add contacts with `add contact` command or have multiple existing contacts in contact list. Alternatively, populate contact list with sample data using `generate` command.
- b. Test case: `edit contact`
Expected: No contact edited. Error Message displays.
- c. Test case: `edit contact /n James`
Expected: No contact edited. Error Message displays.
- d. Test case: `edit contact 1`
Expected: No contact edited. Error Message displays.
- e. Test case: `edit contact 1 2 3 /n James`
Expected: No contact edited. Error Message displays.
- f. Test case: `edit contact x /n John` (where x is an invalid id)
Expected: No contact edited. Error Message displays.
- g. Test case: `edit contact 1 /n /p /d`
Expected: Phone number and details of first contact in list resets to empty value. Name of first contact in list will be retained. Details of updated contact shown in output message.
- h. Test case: `edit contact 1 /n John /p (+65)62457183 /d brother 123@abc.com`
Expected: Updates first contact in list with the John as name, (+65)62457183 as phone number and brother 123@abc.com as details. Details of updated contact shown in output message.
- i. Test case: `edit contact 1 /n Test /p /p`
Expected: Updates first contact in list with Test as name and /p as phone number. Details of updated contact shown in output message.

F.7. Deleting a contact

1. Deleting a contact

- a. Prerequisites: Add contacts with `add contact` command or have multiple existing contacts in contact list. Alternatively, populate contact list with sample data using `generate` command. Ensure no loan entries using the second contact in list. Add loan entry with `add loan lunch $100 21/11/2019 /b /c 1`, so that there is at least one loan entry using the first contact in list.
- b. Test case: `delete contact`
Expected: No contact in list deleted. Error Message displays.
- c. Test case: `delete contact x` (where x is an invalid id)
Expected: No contact in list deleted. Error Message displays.
- d. Test case: `delete contact 1`
Expected: First contact not deleted from contact list, since there is loan entry using it. Error Message displays.

- e. Test case: delete contact 2
Expected: Second contact deleted from contact list. Details of deleted contact shown in output message.
- f. Test case: delete contact 1 2 3
Expected: No contact in list deleted. Error Message displays.

F.8. Adding a loan

1. Adding a loan

- a. Prerequisites: The contact tied to the loan must exist inside the contact list. Assuming there exist only one contact whose id is 1.
- b. Test case: add loan lunch \$10 10/10/2019 /b /c 1
Expected: New loan (Lunch with the date set) is added into the loan list. Details of the added loan is shown in the output message.
- c. Test case: add loan lunch \$10 10/10/2019 /b /c 2
Expected: Loan is **not** added into the loan list (Invalid contact). Error details shown in the output message.
- d. Test case: add loan lunch /b /c 1
Expected: Loan is **not** added into the loan list (Insufficient parameters due to missing amount and date). Error details shown in the output message.
- e. Test case: add loan <param 1> ... <param 6> (for any param 1 to 6 is missing)
Expected: Loan is **not** added into the loan list (Insufficient parameters due to missing amount and date). Error details shown in the output message.

F.9. Editing a loan

1. Editing a loan

- a. Test case: edit loan 1 /d dinner
Expected: Loan with ID 1 has its description set to 'dinner'. Details of the edited loan is shown in the output message.
- b. Test case: edit loan 1 /a 5
Expected: Loan with ID 1 has its amount set to '\$5'. Details of the edited loan is shown in the output message
- c. Test case: edit loan 1 /t 09/10/2019
Expected: Loan with ID 1 has its date set to '09/10/2019'. Details of the edited loan is shown in the output message
- c. Test case: edit loan 1 /l
Expected: Loan with ID 1 has its loan set to 'Lent to'. Details of the edited loan is shown in the output message
- d. Test case: edit loan 1 /c 2
Expected: Loan with ID 1 has its contact set to the contact containing an ID of 2. Details of the edited loan is shown in the output message

- e. Test case: `edit loan 1` (If all parameters are missing)
Expected: Loan is **not** edited and updated in the loan list (Insufficient parameters). Error details shown in the output message.

F.10. Deleting a loan

1. Deleting a loan

- a. Test case: `delete loan 1`
Expected: Loan with ID 1 will be deleted. Details of the deleted loan is shown in the output message.
- b. Test case: `delete loan`
Expected: Loan is **not** deleted in the loan list (no id specified). Error details shown in the output message.

F.11. Settling a loan

1. Settling a loan

- a. Test case: `done loan 1`
Expected: Loan with ID 1 will be mark as settled. Details of the settled loan is shown in the output message.
- b. Test case: `done loan`
Expected: Loan is **not** settled in the loan list (no id specified). Error details shown in the output message.

F.12. Listing your expenses/loans/contacts

1. Listing data

- a. Prerequisites: Populate all the lists with sample data using `generate` command.
- b. Test case: `list loan`
Expected: All loans will be displayed in a table format.
- c. Test case: `list loan /sortby date`
Expected: All loans will be displayed in a table format and sorted according to their date.
- d. Test case: `list contact`
Expected: All contacts will be displayed in a table format.
- e. Test case: `list contact /sortby name`
Expected: All contacts will be displayed in a table format. Contacts are sorted into alphabetical order.
- f. Test case: `list contact /sortby date`
Expected: No contact from contact list displayed. Error Message displays.
- g. Test case: `list contact /sortby`
Expected: No contact from contact list displayed. Error Message displays.
- h. Test case: `list expense`
Expected: All expenses will be displayed in a table format.

F.13. Setting/Viewing budget

1. Setting a budget

- a. Prerequisites: add expense food \$25 Food /on 01/01/2019
- b. Test case: budget \$500 01/2019
Expected: Since there is an existing budget, an output message is shown that asks user if they would like to take into account their existing expenses. If user replies "yes", New budget is added into the budget list with budget amount being \$475. Details of the added budget shown in the output message.
- c. Test case: budget \$500 02/2019
Expected: New budget is added into the budget list. Details of the added budget shown in the output message.
- d. Test case: budget \$0 01/2019
Expected: Budget in the budget list from test case (b) will be deleted. Details of the deleted budget shown in the output message
- e. Test case: budget \$-100 01/2019
Expected: Budget is **not** added into the budget list (Invalid amount provided). Error details shown in the output message.

2. Viewing a budget

- a. Prerequisites: budget \$25 01/2019 (adds a budget amount of \$25 to 01/2019)
- b. Test case: view budget 01/2019
Expected: Details of the existing budget shown in the output message.
- c. Test case: view budget 02/2019
Expected: Error details shown in the output message. (Non-existing budget)

F.14. Viewing statistics

1. Viewing combined expenses stats

- a. Prerequisites: Some existing expenses
- b. Test case: view stats
Expected: A pie chart and bar graph of total expenses based on their category shown.

2. Viewing specified expenses stats

- a. Prerequisites: Some existing expense for January 2019
- b. Test case: view stats 01/2019
Expected: A pie chart and bar graph of expenses made on January 2019 based on their category shown.

F.15. Changing currency

1. Changing all data into specified currency
 - a. Prerequisites: Some existing data
 - b. Test case: `currency south korea`
Expected: Amount in expenses and loans in their respective lists are converted into korean won currency rates. Details of the currency change shown in the output message.
 - c. Test case: `currency abcd`
Expected: currency is **not** changed in both loans and expense list (Invalid country provided). Error details shown in the output message.

F.16. Viewing command history

- a. Test case: `history`
Expected: A list of commands that affected the storage files.

F.17. Undoing/Redoing commands

- a. Prerequisites: type in any command that modifies the storage (`add`, `edit`, `done`, `delete`)
- b. Test case: `undo`
Expected: System will revert the state.
- c. Test case: `redo`
Expected: System will move forward by one state.

F.18. Import data

1. Import records
 - a. Prerequisites: CSV file in **WalletCLi** home directory which is formatted correctly according to requirements listed under Import Section in User Guide.
 - b. Test case: `import expense FILENAME` (where FILENAME is a valid filename)
Expected: New expense records added to expense list. Details of new expense records displayed. Details of new budget will be calculated and displayed.
 - c. Test case: `import loan FILENAME` (where FILENAME is a valid filename)
Expected: New loan records added to loan list. Details of new loan records displayed.
 - d. Test case: `import expense`
Expected: No data imported. Error Message displays.
 - e. Test case: `import loan`
Expected: No data imported. Error Message displays.
 - f. Test case: `import`
Expected: No data imported. Error Message displays.

- g. Test case: `import budget`
Expected: No data imported. Error Message displays.
- h. Test case: `import expense FILENAME` (where `FILENAME` is an invalid filename)
Expected: No data exported. Error Message displays.
- i. Test case: `import loan FILENAME` (where `FILENAME` is an invalid filename)
Expected: No data exported. Error Message displays.

F.19. Export data

1. Export records
 - a. Populate expense and loan list with sample data using `generate` command.
 - b. Test case: `export loan`
Expected: CSV created in **WalletCLi** home directory with filename in format of `WalletCLi-loans-[date exported]-[time exported].csv`. CSV populated with loan records if there were records being exported.
 - c. Test case: `export expense mm/yyyy` (where `mm/yyyy` is a valid month)
Expected: CSV created in **WalletCLi** home directory with filename in format of `WalletCLi-expenses-[date exported]-[time exported].csv`. CSV populated with expense records if there were records being exported.
 - d. Test case: `export expense`
Expected: No data exported. Error Message displays.
 - e. Test case: `export`
Expected: No data exported. Error Message displays.
 - f. Test case: `export budget`
Expected: No data exported. Error Message displays.
 - g. Test case: `export expense mm/yyyy` (where `mm/yyyy` is an invalid month)
Expected: No data exported. Error Message displays.

F.20. Help

1. List out available help sections
 - a. Prerequisites: Help section data from text files in resource folder of jar file successfully loaded into the application.
 - b. Test case: `help`
Expected: List of help sections displayed.
2. Display content of a help section
 - a. Test case: `help x` (where `x` is an invalid index)
Expected: No help section content displayed. Error Message displays.
 - b. Test case: `help x` (where `x` is valid index)
Expected: Help section content displayed with command syntaxes and description. Examples of command usages may be displayed as well.

