

WalletCLi - Developer Guide

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

Table of Contents

1. Introduction	4
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 Future Implementation.....	15
5.3 Managing Contacts	15
5.3.1 Listing Contact.....	15
5.3.2 Adding Contact Implementation.....	15
5.3.3 Editing Contact Implementation.....	17
5.3.4 Delete Contact Implementation	19
5.3.5 Future Implementation.....	19
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
5.8 Managing Expenses.....	29
5.8.1 Listing Expense.....	29
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
6. Documentation	32
7. Testing.....	32
8. Dev ops.....	32
Appendix A - Product Scope.....	32
Appendix B - User Stories.....	33
Appendix C - Use Cases	35
Appendix D - Non-functional Requirements.....	42
Appendix E - Glossary.....	43
Appendix F - Instructions for manual testing.....	43

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 tip. Follow these suggested tips to make your life much simpler when using WalletCLi !
	This is a note. These are things for you to take note of when using WalletCLi .
	This is a signpost dictating important information. These are information that you will surely need to know to use WalletCLi efficiently.
	This is a sign-post informing caution. Please take note of these items and exercise some care.
	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.
 - i. Click `Configure > Project Defaults > Project Structure`
 - ii. 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

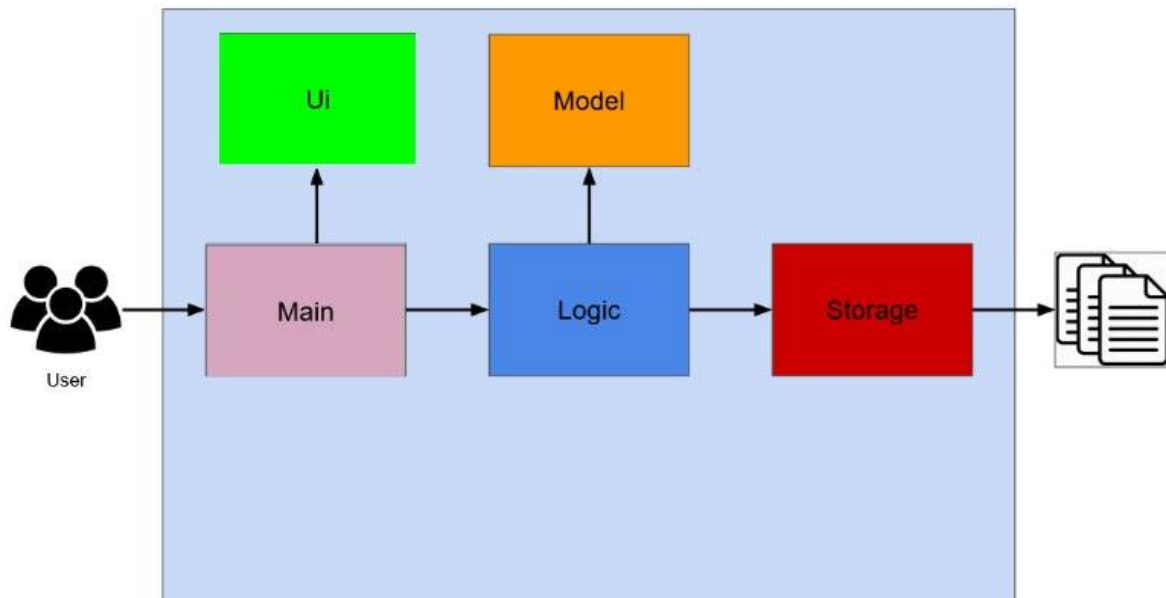


Figure 4.1.1 Architecture Diagram

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.

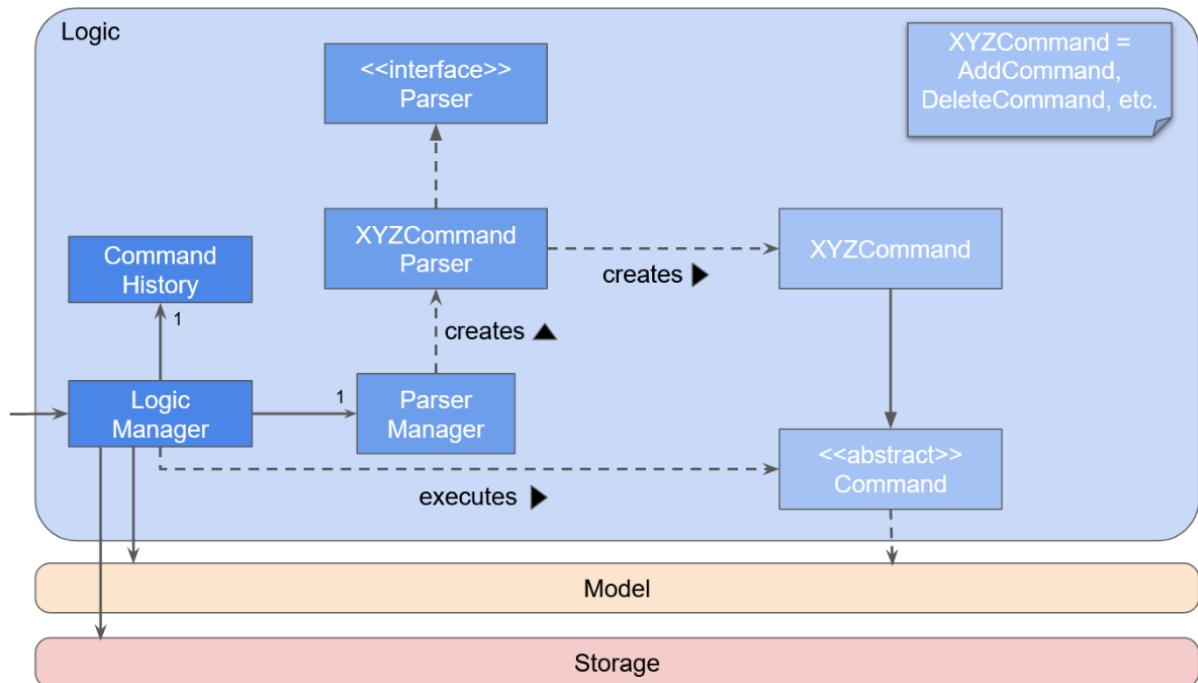


Figure 4.2.1 Structure of the Logic Component

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).

Given below is the Sequence Diagram for interactions within the Logic component for the `execute("delete expense 1")` API call.

4.3 Model Component

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

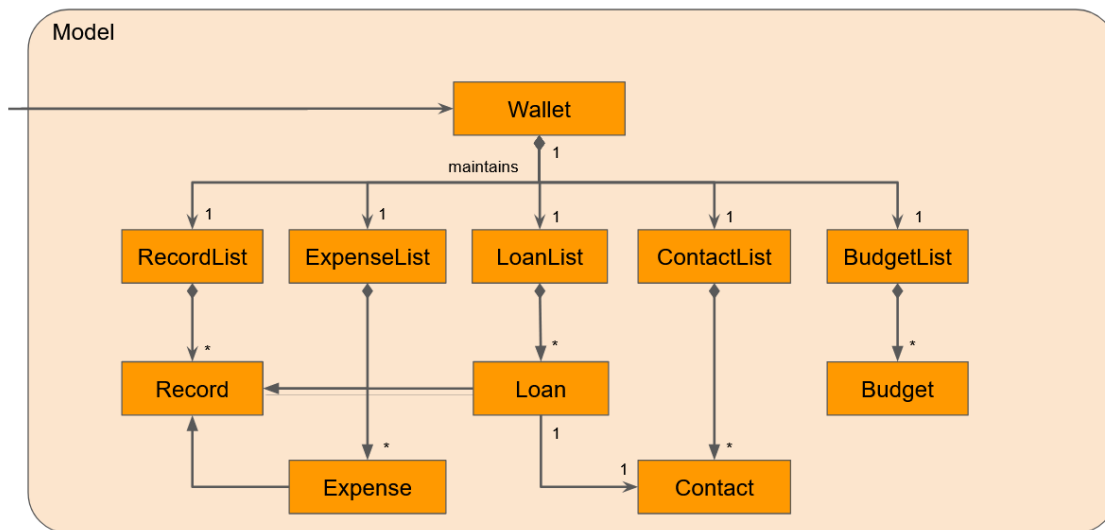


Figure 4.3.1 Structure of the Model Component

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

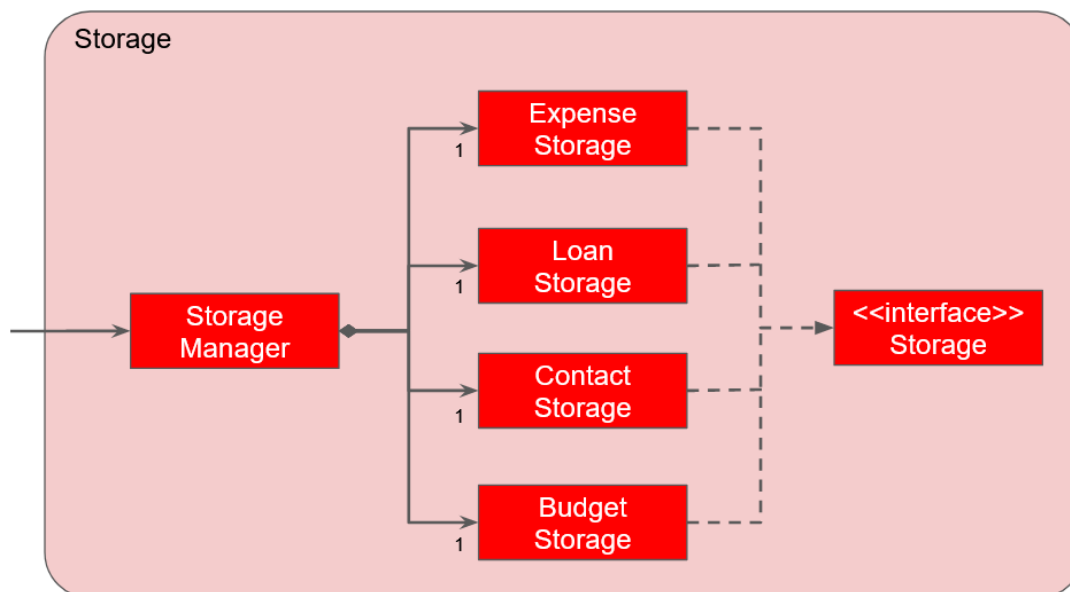


Figure 4.4.1 Structure of the Storage Component

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:

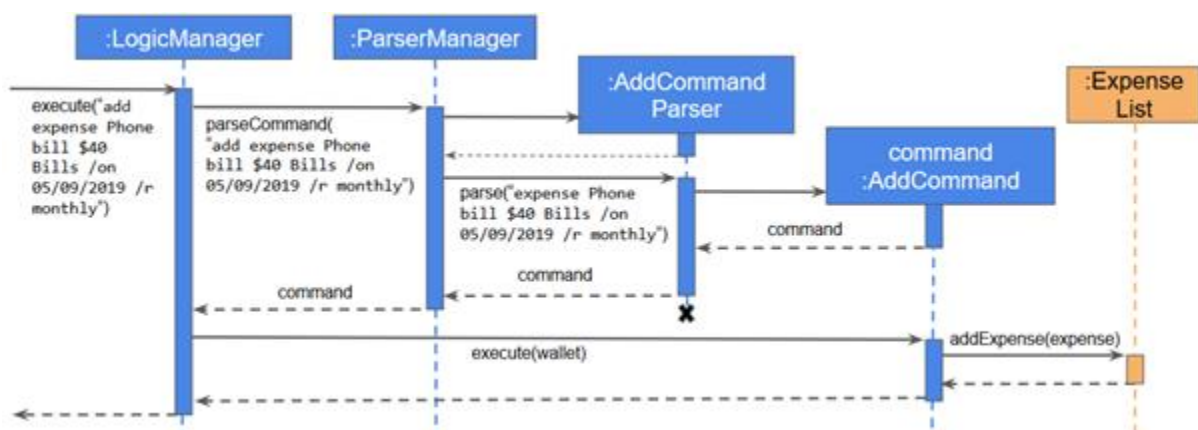


Figure 5.1.1.1 Sequence diagram when adding a new expense

3. Whenever a command is executed, the LogicManager class will invoke the ExpenseParser class to update all the recurring expenses using the updateRecurringRecords method.

4. The ExpenseParser class retrieves the ExpenseList object from the wallet class using the getExpenseList method.
5. After retrieving the ExpenseList, the ExpenseParser class invokes the getRecurringRecords method to filter out the recurring expenses in the list.
6. The ExpenseParser class then invokes the findExpenseIndex method to retrieve the index of the expense on the list.
7. The ExpenseParser class invokes the editExpense method to update this expense's isRecurring value to False.
8. The ExpenseParser class adds a month to the date of the recurring expense and checks if it is greater than the current month.
9. 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.
10. Repeat step 9 until the new date is greater than the current month.
11. 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:

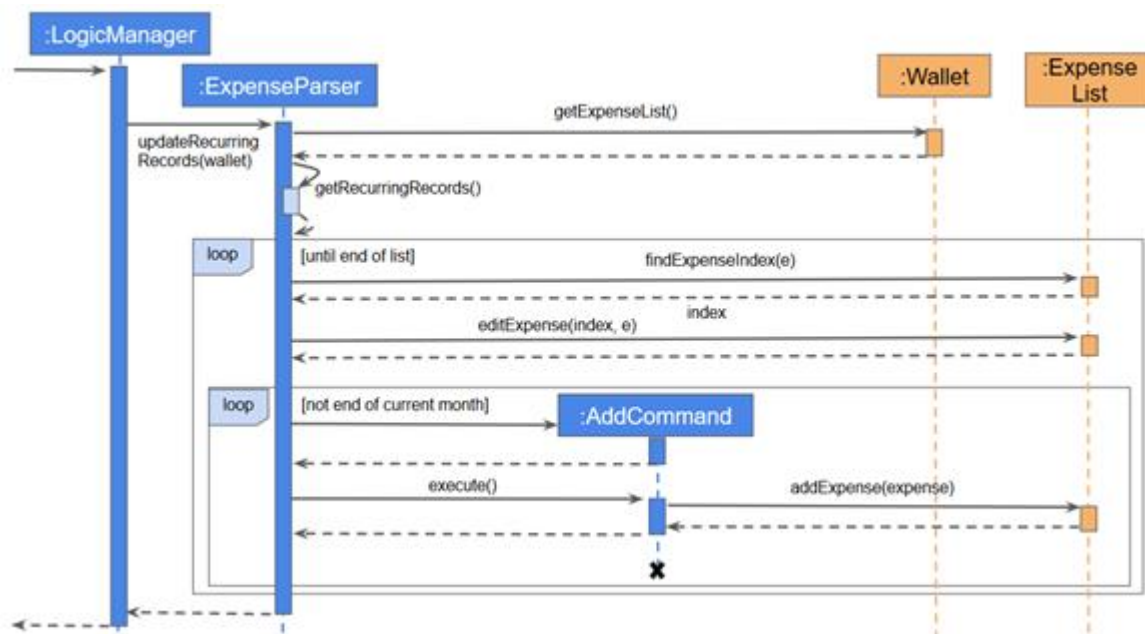


Figure 5.1.1.2 Sequence diagram updating recurring expenses

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 heavy, hence 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 <ID OF LOAN> </d NEW DESCRIPTION> </a NEW AMOUNT> </t NEW DATE> [</l or /b>] [</c> ID OF CONTACT]`
 - 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 <ID OF LOAN>
 - Allows users to settle their loans.
- list loan
 - Allows user to view the existing loans in a table format
- delete loan <ID OF LOAN>
 - 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.

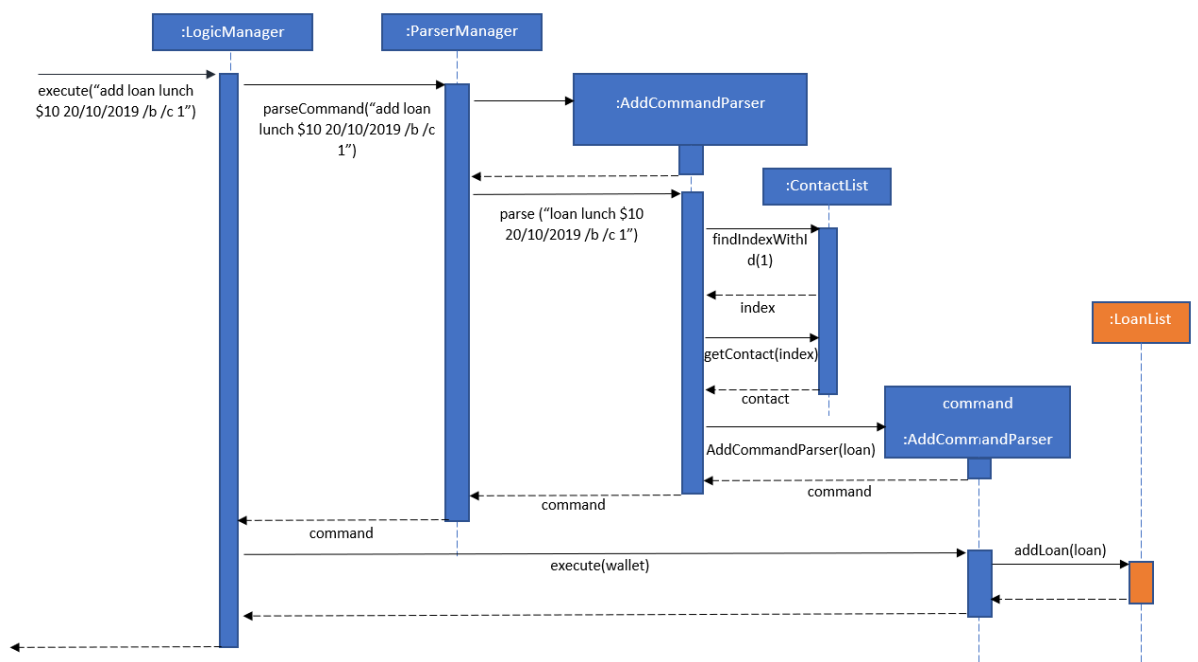


Figure 5.2.1 Sequence Diagram of Adding Loan

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.

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. Example command used here is `add contact Mary Tan /p 8728 1831 /d sister`.

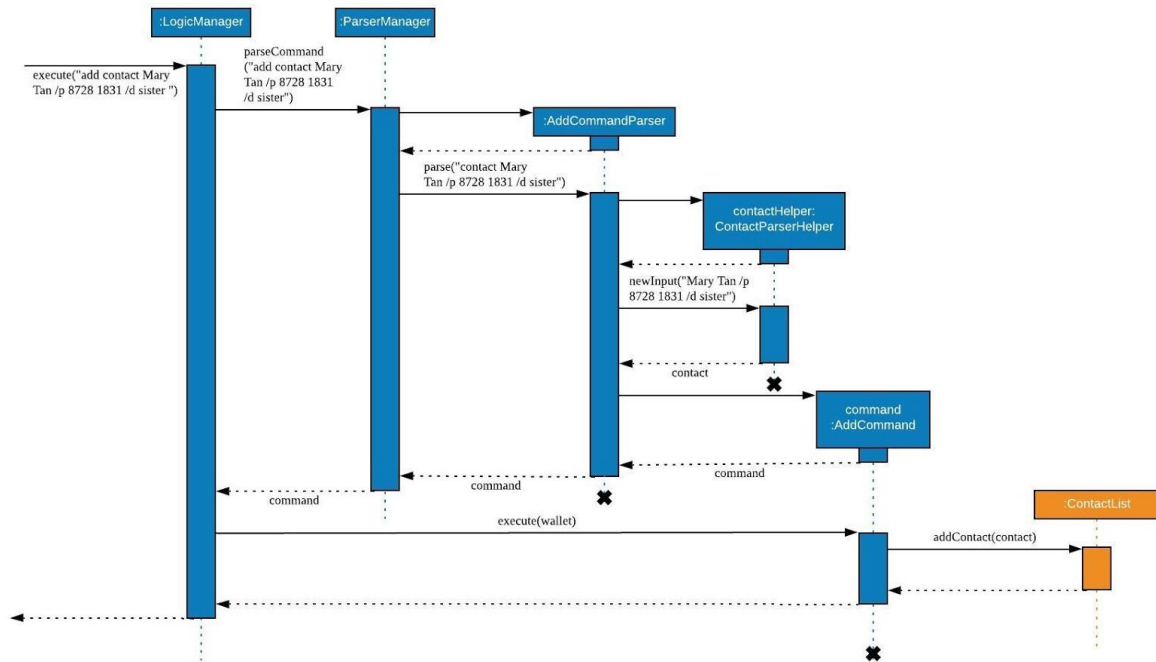




Figure 5.3.2.1 Sequence diagram for Add Contact

The steps below will explain the sequence diagram shown above:

1. “add contact Mary Tan /p 8728 1831 /d sister” command is being entered into the application.
2. 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.
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 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 are allowed to include spaces in their command arguments and the optional command line arguments (/d and /p) do not need to be</p>

	entered in a particular order. ContactParserHelper will process the input accordingly.
--	--

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. Example command used here is `edit contact 6 /n John /p 81007183 /d brother 123@abc.com`.

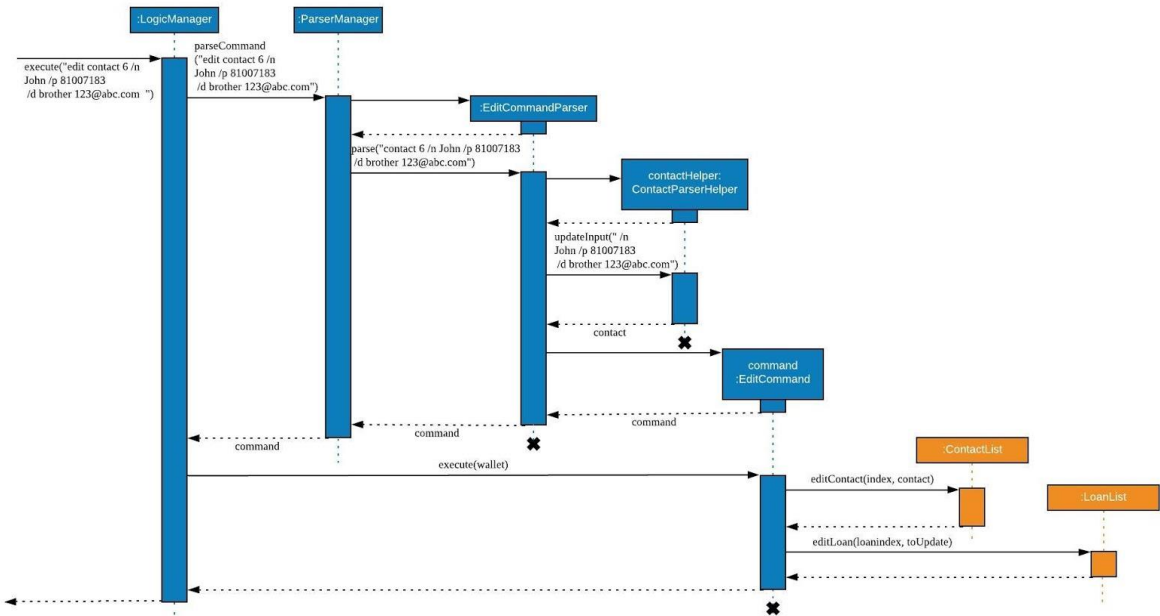





Figure 5.3.3.1 Sequence diagram for Edit Contact

The steps below will explain the sequence diagram shown above:

1. “edit contact 6 /n John /p 81007183 /d brother 123@abc.com” command is being entered into the application.
2. 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) Process any arguments after `/n`, `/p` and `/d` into `Name`, `Phone Number` and `Details` respectively. In this case, it would be `John`, `81007183` and `brother 123@abc.com` respectively processed.
 - b) If there is no `/n` or no arguments for `/n`, the original `Name` value in the contact entry will be retained.
 - c) 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.
 - d) 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. `Contact`, as well as other variable values from the loan entry, will be included in `toUpdate`. The loan entry in `LoanList` will be replaced with the `toUpdate` object, according to the corresponding element index, `loanIndex`, which is retrieved based on the `ID` of edited contact.

	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.
	Users can include spaces in their command arguments and the optional command line arguments (<code>/d</code> , <code>/n</code> and <code>/p</code>) do not need to be entered in a particular order. <code>ContactParserHelper</code> will process the input accordingly.
	At Step 4, error messages will display if the <code>ID</code> is invalid and no contact or loan entry will be updated.

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.

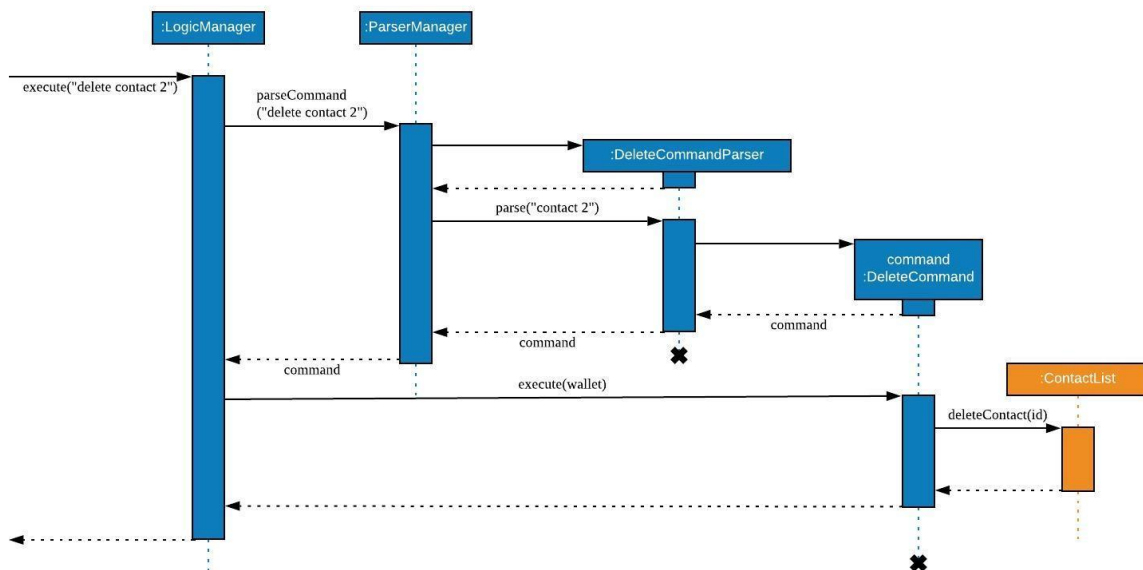




Figure 5.3.4.1 Sequence diagram for Delete Contact

The steps below will explain the sequence diagram shown above:

1. "delete contact 2" command is being entered into the application.
2. At the Logic Component, DeleteCommandParser will parse the command and check if the input is valid. At this step, the parse() command will check if any existing loans are using the contact entry that user requests for deletion.
3. If no existing loans are using the contact entry, a command object is returned to LogicManager.
4. LogicManager executes the command object, which deletes the contact entry from ContactList based on ID indicated by the user.

	At Step 2, if existing loans are using contact entry, the steps after will not be executed and an error message will display.
	At Step 4, error messages will display if the ID is invalid and no contact entry will be deleted.

5.3.5 Future Implementation

These are features considered for future implementation:

1. Implement /sortby option for users to view names in alphabetical order when listing 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 <code>budget \$40 01/2019</code>
	The view command requires 2 parameters, budget and month/year, ie <code>view budget 01/2019</code>

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.

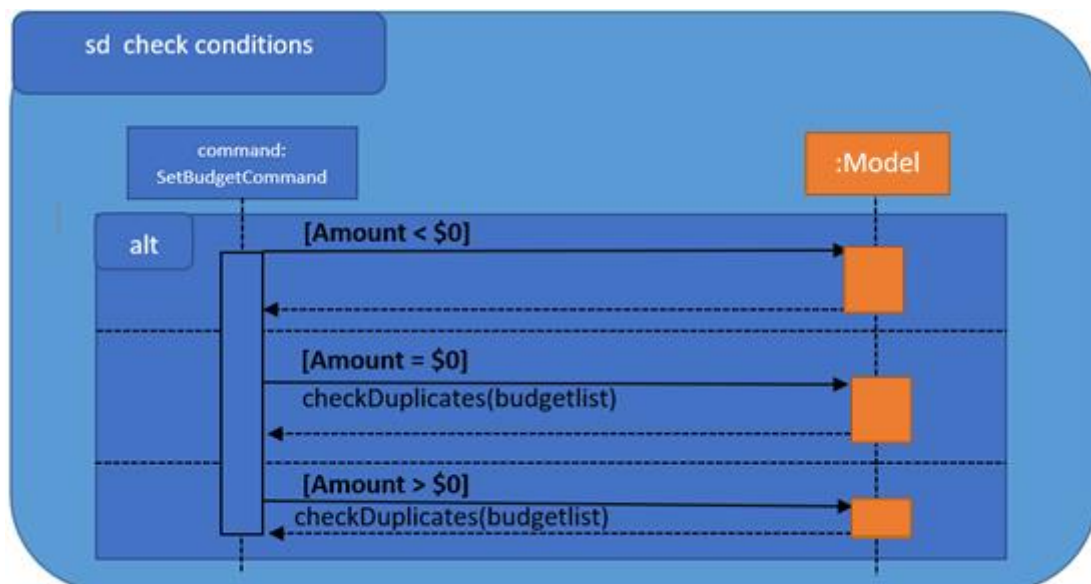
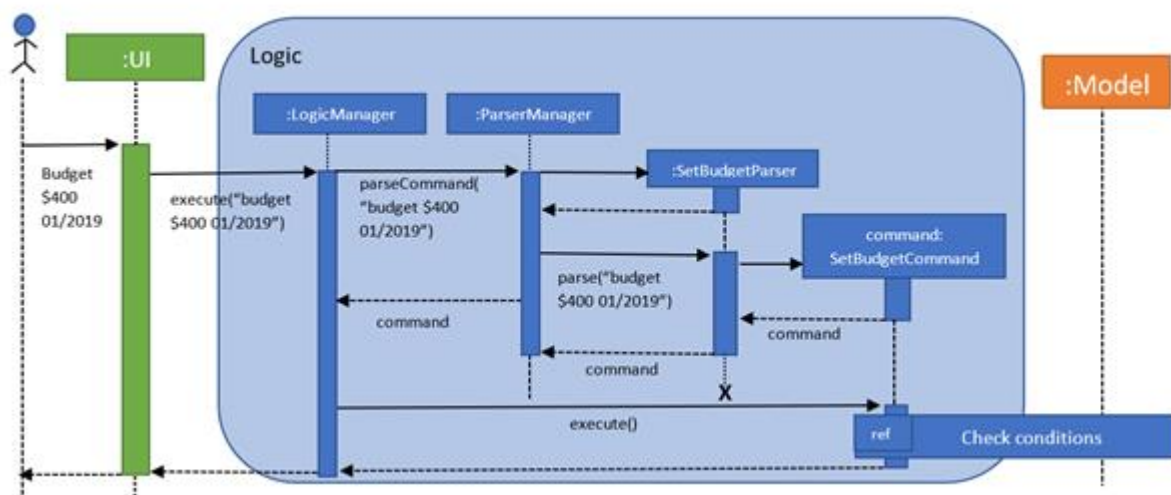


Figure 5.4.1.1a and 5.4.1.1b Sequence diagram of setting Budget

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.

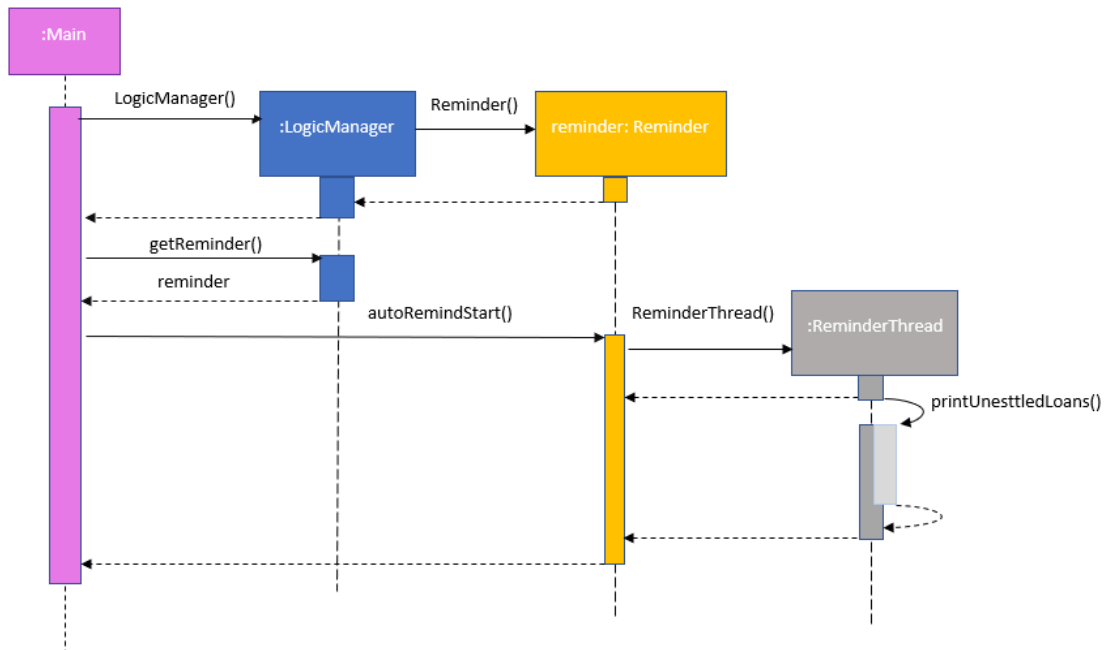


Figure 5.5.1.1 Sequence Diagram of Auto Reminder system at boot of WalletCLI.

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, `ViewCommandParse` 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 section aims to help users to understand the different command syntax and usage.

When a user executes `help` command in the application and keys in a help section index, **WalletCLi** will retrieve the corresponding text file that contains the help section content from `/src/main/resources`. 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. The information of section names against text file names 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 format required for each command when updating 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

/l, use this if entry indicates the amount you lend others
/b, use this if entry indicates the amount you borrow from others

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. a

5.7.1 Current Code Implementation

The section describes how the in-app help section is implemented in terms of code.

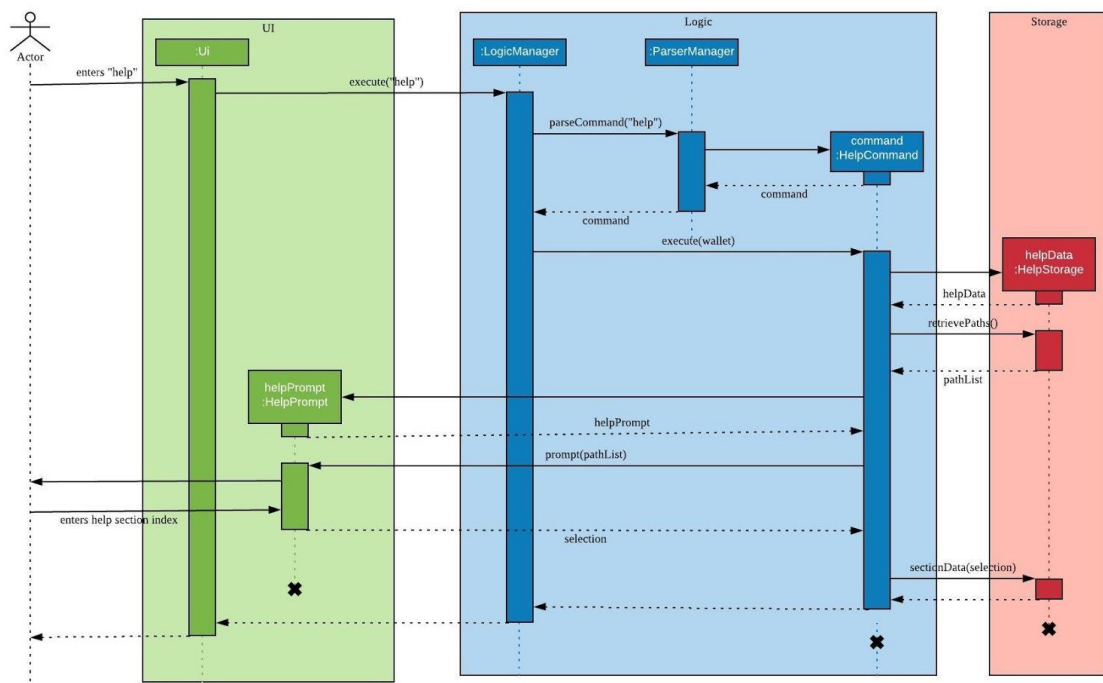





Figure 5.7.1.1 Sequence diagram for Help

The main flow of the help section is implemented according to the sequence diagram shown above. The interaction between components modifies the flow of the architecture depicted in section 4.1. The steps below further explain the sequence diagram shown above:

1. User inputs `help` command into the application, and it is passed into the `execute` function of `LogicManager`.
2. `LogicManager` invokes the `parseCommand` function of `ParserManager`.
3. `ParserManager`, in turn, retrieves a new `HelpCommand` object that returns to the `LogicManager`.
4. `LogicManager` invokes the `execute` function of the returned `HelpCommand` object.

5. In the `execute` function of the `HelpCommand` object, it first retrieves a `HelpStorage` object.
6. The `HelpCommand` object invokes `retrievePaths` function in `HelpStorage` which reads `/src/main/resources/helppaths.txt` and returns a list of sections names against the relative path of the corresponding text file. The variable for the list is called `pathList`.
7. The `HelpCommand` object retrieves a `HelpPrompt` object
8. The `HelpCommand` object invokes the `prompt` function while parsing in `pathList`.
9. `HelpPrompt` object will list out the section names from `pathList` and prompt User to key in the index of the help section they require. This index, known as `selection`, will be returned to `HelpCommand`.
10. The `HelpCommand` object invokes `sectionData` function in `HelpStorage` while parsing in the value of `section`.
11. According to the index value specified in `section`, `HelpStorage` retrieves the corresponding relative file path from `pathList` to open and print the corresponding file content from `/src/main/resources` folder for the User to read.

	At Step 9, if the user inputs an invalid index, the steps after will not be executed and an error message will display.
	At Step 9, User can exit the help prompt by entering 0 and steps after will not be executed.
	At Step 6 and 11, error messages will display if files can't be read and steps after will not be executed.

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`

5.8 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:

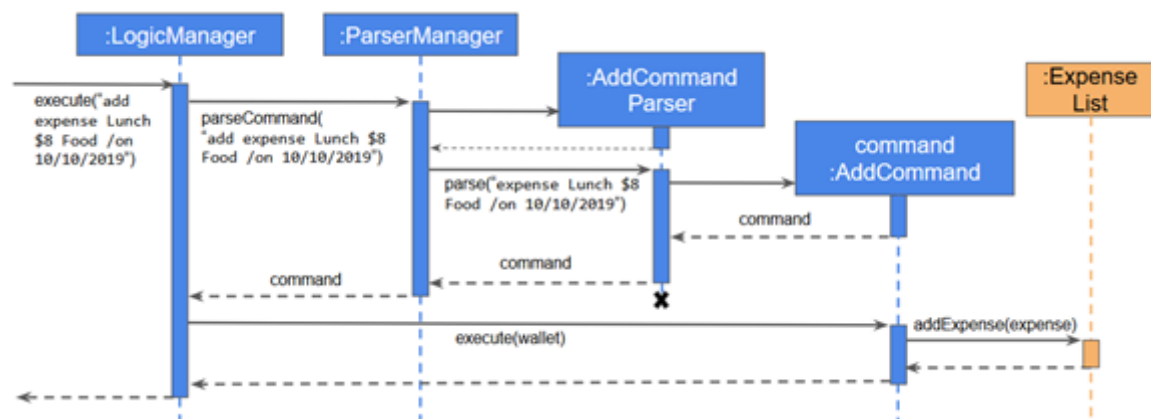


Figure 5.8.2.1 Sequence diagram when adding an expense

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:

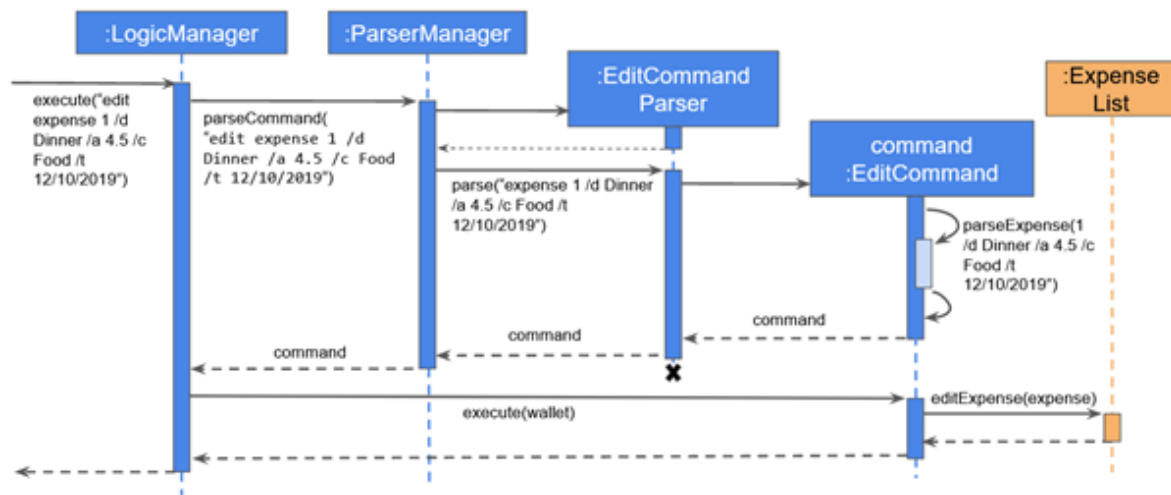


Figure 5.8.3.1 Sequence diagram when editing an expense

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:

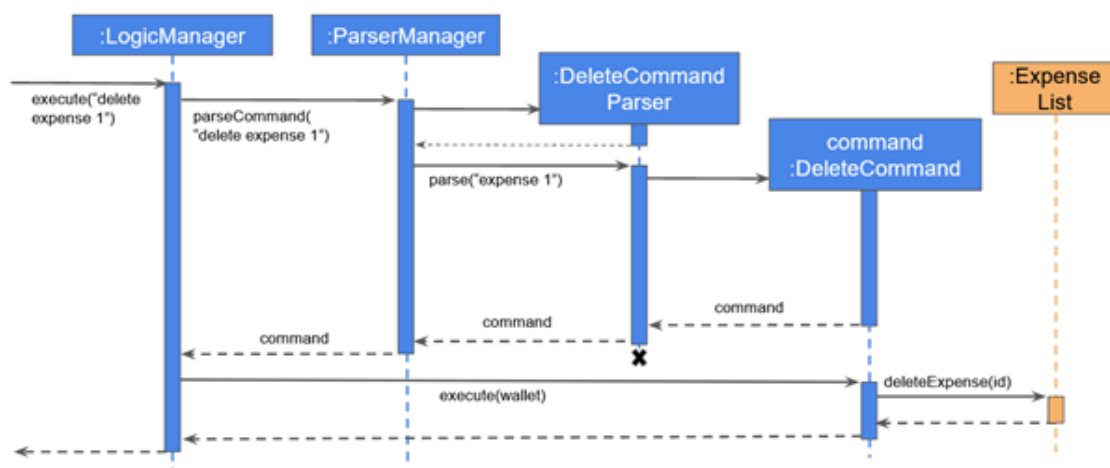


Figure 5.8.4.1 Sequence diagram when deleting an expense

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.

6. Documentation

7. Testing

8. Dev ops

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	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.

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
