

# Coupon Stash v1.4 - Developer Guide

1. Setting up .....	2
2. Design .....	2
2.1. Architecture .....	2
2.2. UI component .....	4
2.3. Logic component .....	5
2.4. Model component .....	7
2.5. Storage component .....	8
2.6. Common classes .....	9
3. Implementation .....	9
3.1. Undo/Redo feature .....	9
3.2. Calendar .....	14
3.3. Coupon Archiving .....	20
3.4. Up/down arrow retrieve command history .....	24
3.5. Coupon Sorting .....	28
3.6. Coupon Reminder .....	31
3.7. Savings per use and total amount saved .....	35
3.8. Logging .....	42
3.9. Configuration .....	42
4. Documentation .....	42
5. Testing .....	42
6. Dev Ops .....	43
Appendix A: Product Scope .....	43
Appendix B: User Stories .....	43
Appendix C: Use Cases .....	47
C.1. Use Case: UC1 - Add Coupon .....	48
C.2. Use Case: UC2 - List all coupons .....	49
C.3. Use Case: UC3 - Mark a coupon as used .....	50
C.4. Use Case: UC4 - Find coupon(s) by keyword(s) .....	50
C.5. Use Case: UC5 - Edit coupon's details .....	51
C.6. Use Case: UC6 - Set reminder .....	52
C.7. Use Case: UC7 - List coupon(s) expiring before date .....	52
C.8. Use Case: UC8 - Delete coupon .....	53
C.9. Use Case: UC9 - Undo previous command .....	54
Appendix D: Non-Functional Requirements .....	54
Appendix E: Glossary .....	55
Appendix F: Instructions for Manual Testing .....	55
F.1. Launch and Shutdown .....	55
F.2. Adding a coupon .....	56

F.3. Listing coupons .....	56
F.4. Deleting a coupon .....	56
F.5. Editing a Coupon .....	56
F.6. Finding a coupon .....	57
F.7. Sorting the Coupon Stash .....	57
F.8. Listing all expiring coupons .....	57
F.9. Viewing savings .....	58
F.10. Using a coupon .....	58
F.11. Archiving a coupon .....	58
F.12. Copying a coupon .....	58
F.13. Sharing a coupon .....	59
F.14. Undoing a command .....	59
F.15. Going to a month on the calendar .....	59
F.16. Expanding a coupon .....	59
F.17. Setting currency symbol .....	60
F.18. Viewing help page .....	60
Appendix G: Efforts .....	60

By: Team F09-1

# 1. Setting up

Refer to the guide [here](#).

## 2. Design

### 2.1. Architecture

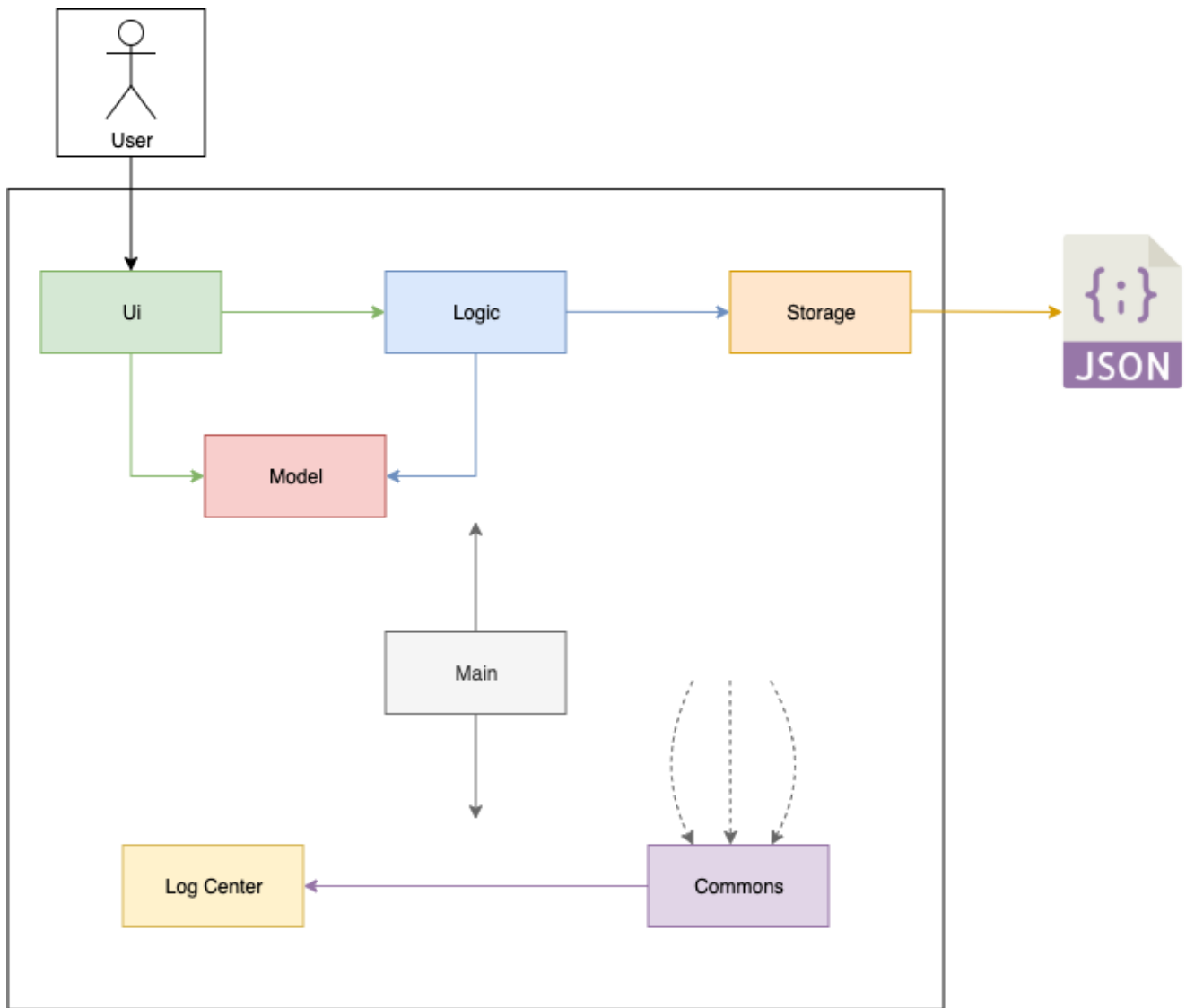


Figure 1. Architecture Diagram of Coupon Stash.

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

**Main** has two classes called **Main** and **MainApp**. It is responsible for,

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

**Commons** represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- **LogsCenter** : Used by many classes to write log messages to the App's log file.

The rest of the App consists of 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.

## How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command **delete 1**.

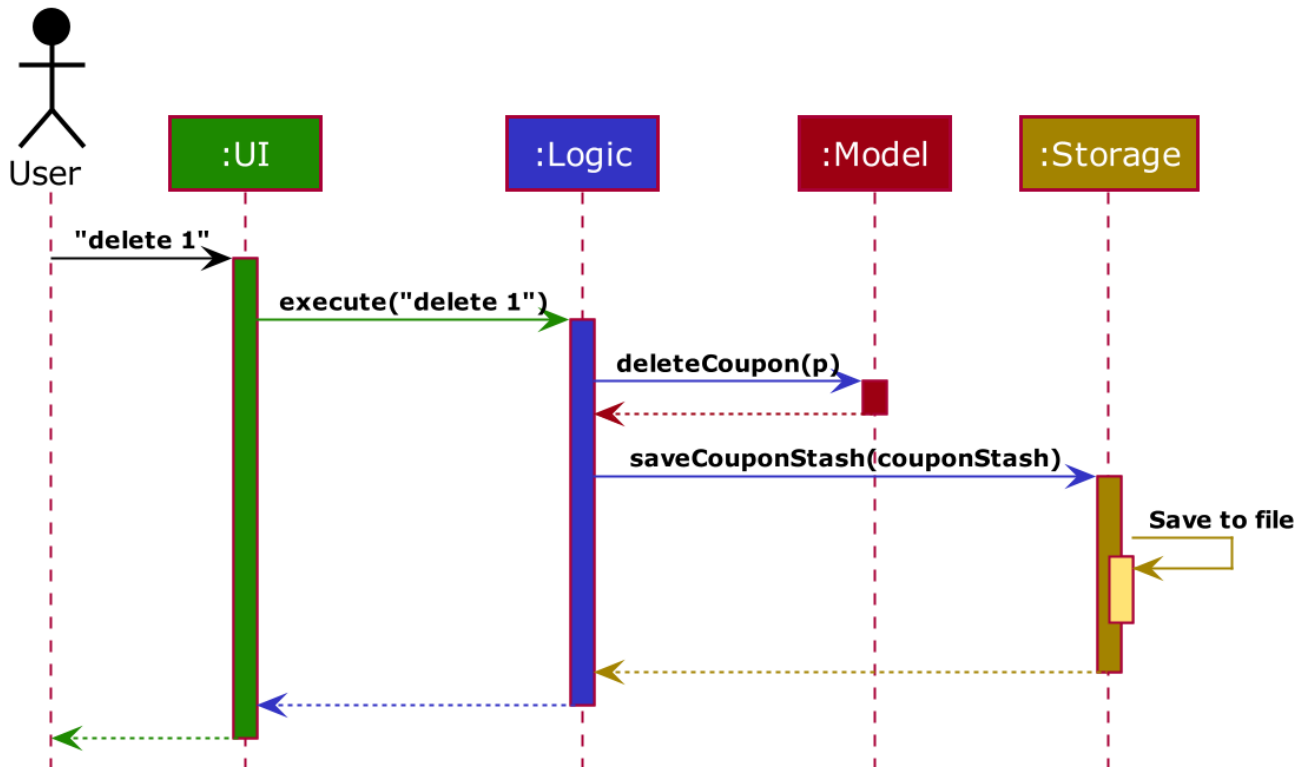


Figure 2. Component interactions for **delete 1** command.

The sections below give more details of each component.

## 2.2. UI component

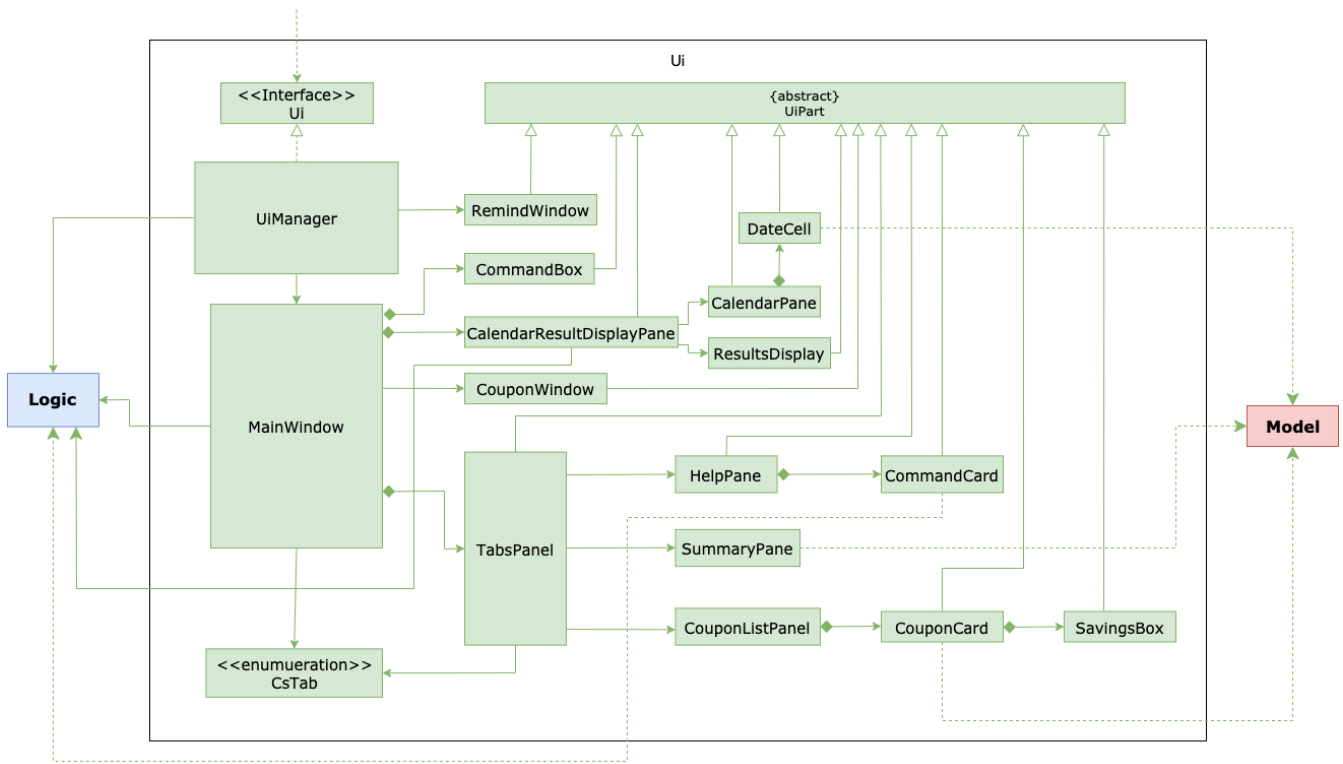


Figure 3. Structure of the UI Component.

#### API : Ui.java

The UI consists of a `MainWindow` that is made up of three main parts - `CommandBox`, `CalendarResultDisplayPanel` and a `TabsPanel`. All these, including the `MainWindow`, inherit from the abstract `UIPart` class.

The `TabsPanel` can be further divided into three tabs - `HelpPane`, `SummaryPane` and the `CouponListPanel`. The `CouponListPanel` contains the `CouponCard` that represents the `Coupons` themselves.

The `CalendarResultsDisplayPanel` is split into two UI components - the `CalendarPane` and the `ResultDisplay`. As the name suggests, the `CalendarPane` holds the calendar that allows easy visualization of expiry dates. The `ResultDisplay` provides feedback from Coupon Stash to the user after he runs a command.

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

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

## 2.3. Logic component



## NOTE

The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram. This limitation affects all of the sequence diagrams in this document.

## 2.4. Model component

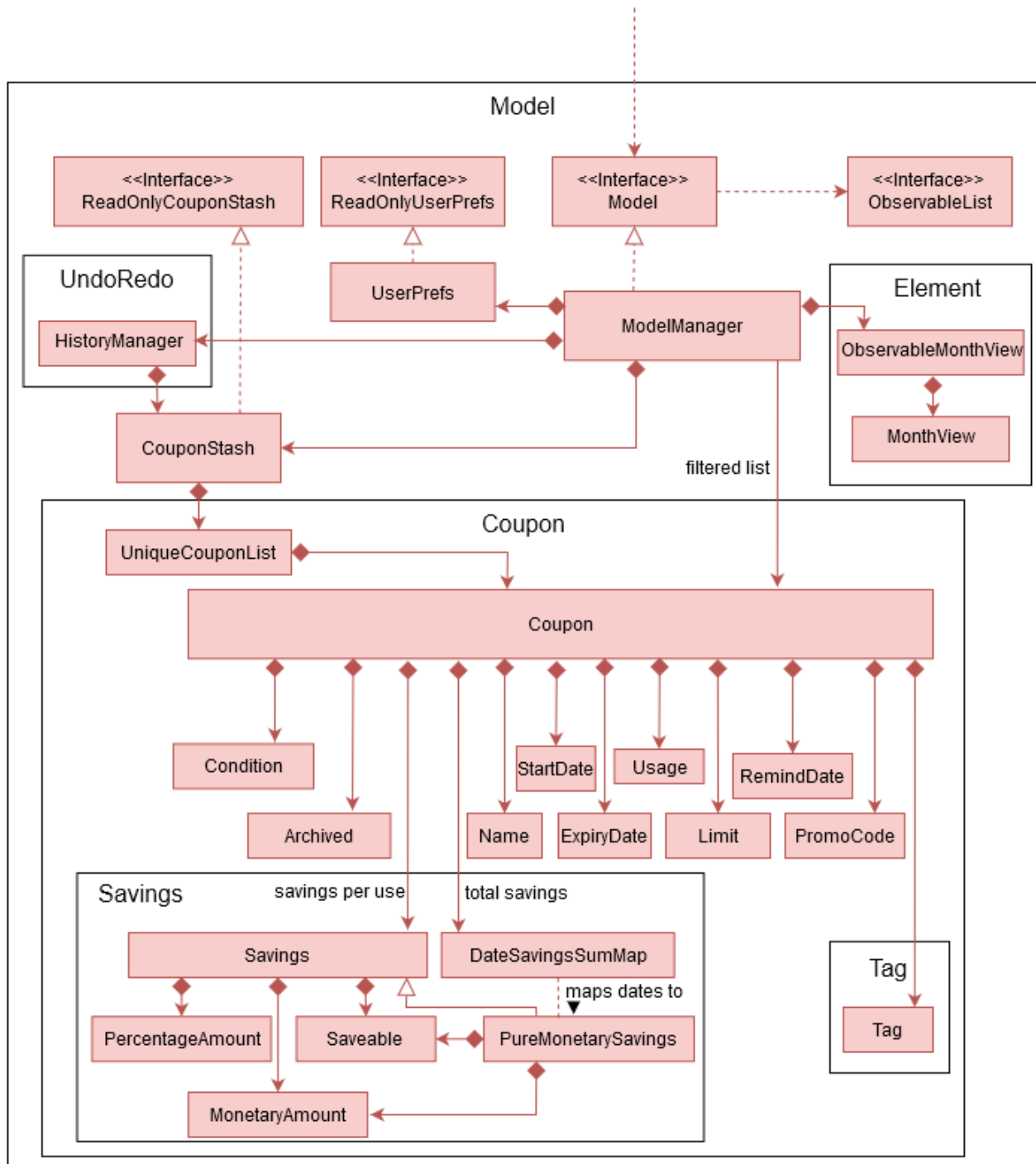


Figure 6. Structure of the Model Component.

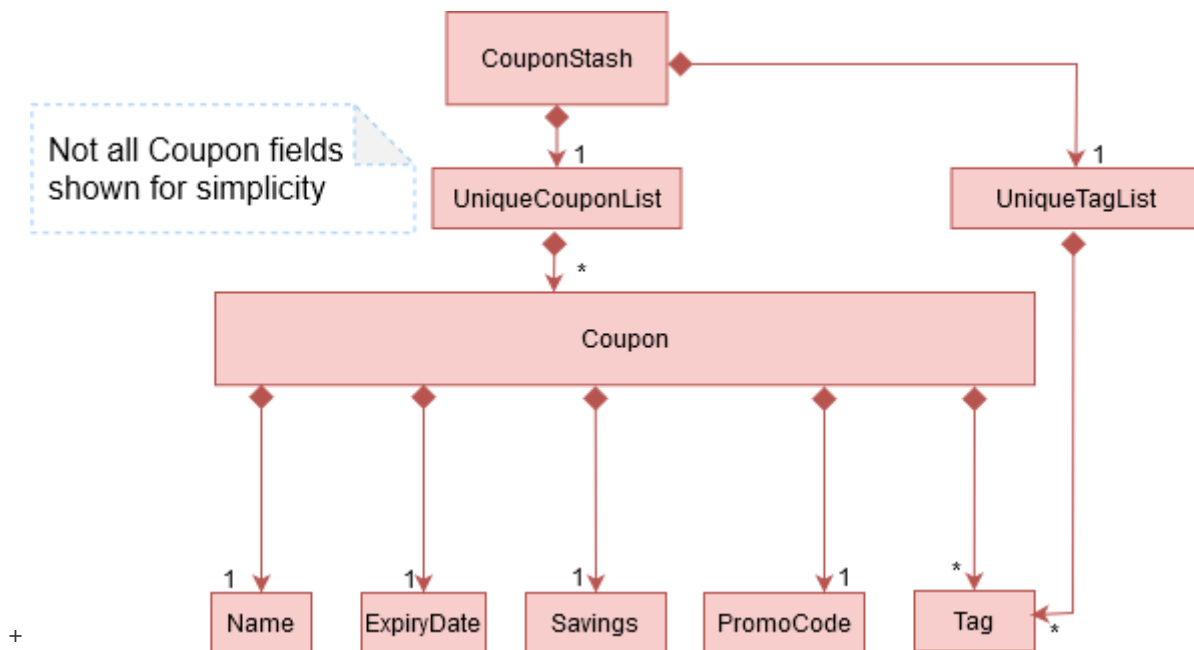
API : `Model.java`

The `Model`,

- stores a **UserPref** object that represents the user's preferences.
- stores the Coupon Stash data.
  - some examples of preferences that can be set are the money symbol, or window sizes.
- exposes an unmodifiable **ObservableList<Coupon>** that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

#### NOTE

As a more OOP model, we can store a **Tag** list in Coupon Stash, which a **Coupon** can refer to. This would allow Coupon Stash to only require one **Tag** object per unique tag, instead of each **Coupon** needing their own **Tag** object. An example of how such a model may look like is given below.



## 2.5. Storage component



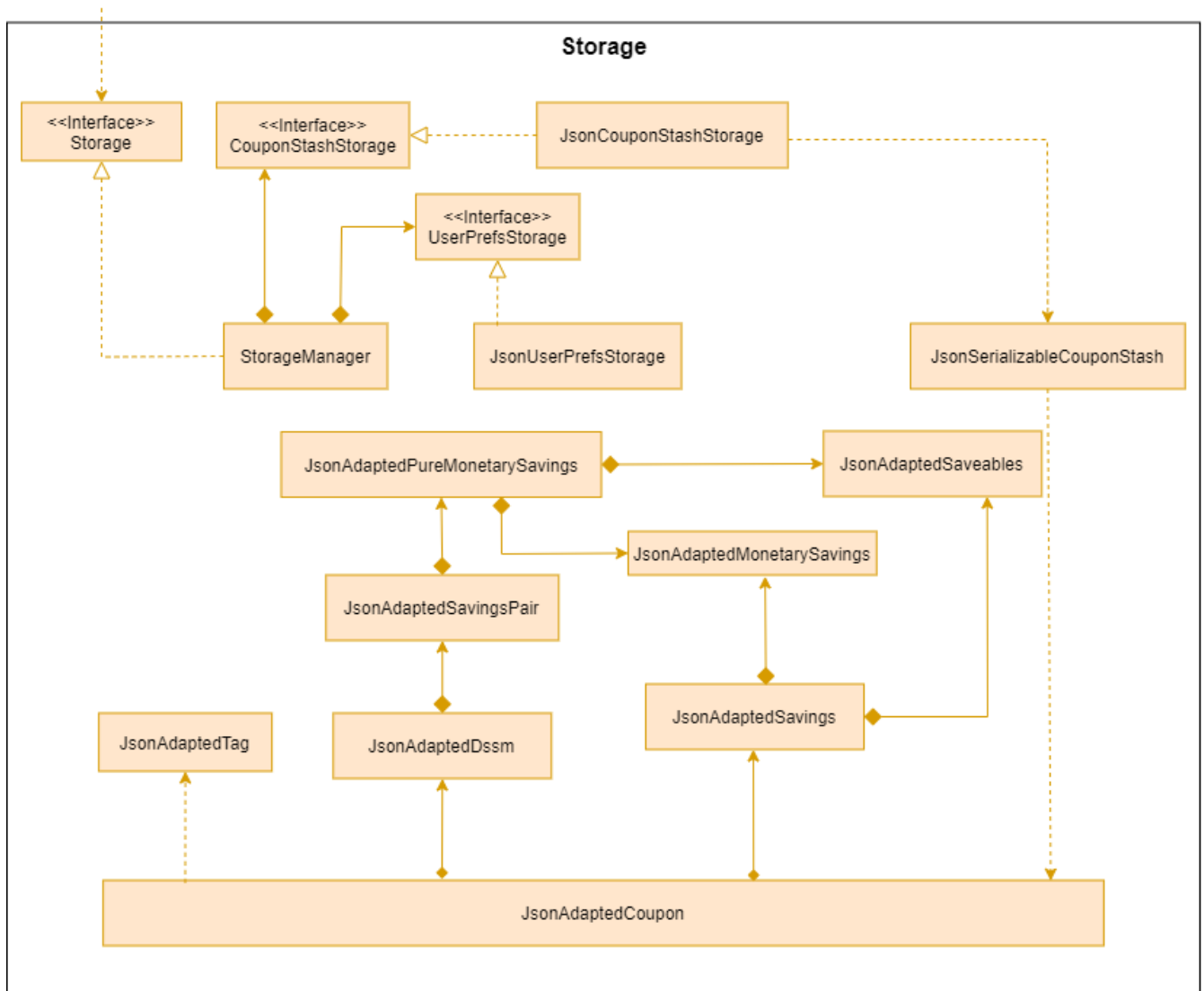


Figure 7. Structure of the Storage Component.

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Coupon Stash data in json format and read it back.

## 2.6. Common classes

Classes used by multiple components are in the `csdev.couponstash.common` package.

# 3. Implementation

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

## 3.1. Undo/Redo feature

The undo/redo mechanism is facilitated by with an undo/redo history, stored internally as an

`couponStashStateList` with a `commandTextHistory` and `currStateIndex`. All these components are encapsulated in the `HistoryManager` class. The following methods in the `Model` interface facilitates this feature:

- `Model#commitCouponStash(String commandText)` — Saves the current coupon stash state and the command text that triggered the change in state into `HistoryManager`.
- `Model#undo()` — Restores the previous coupon stash state from `HistoryManager`.
- `Model#redo()` — Restores a previously undone coupon stash state from `HistoryManager`.

### 3.1.1. Current Implementation

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

Step 1. The user launches the application for the first time. The `CouponStash` will be initialized with the initial coupon stash state, and the `currStateIndex` pointing to that single coupon stash state.

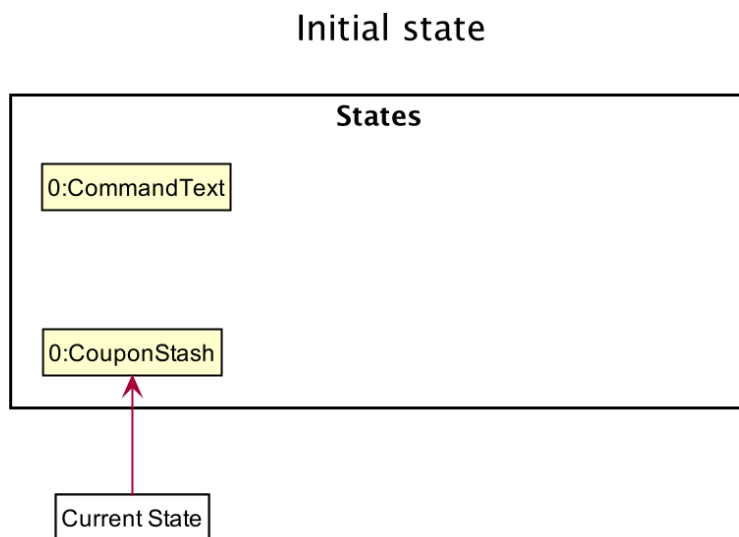


Figure 8. `CouponStash` will be initialized with the initial coupon stash state.

Step 2. The user executes `delete 5` command to delete the 5th coupon in the coupon stash. The `delete` command calls `Model#commitCouponStash(String commandText)`, causing the modified state of the coupon stash after the `delete 5` command executes to be saved in the `couponStashStateList`, and the `delete 5` command text to be stored in the `commandTextHistory`. `currStateIndex` is shifted to the newly inserted coupon stash state.

After command "delete 5"

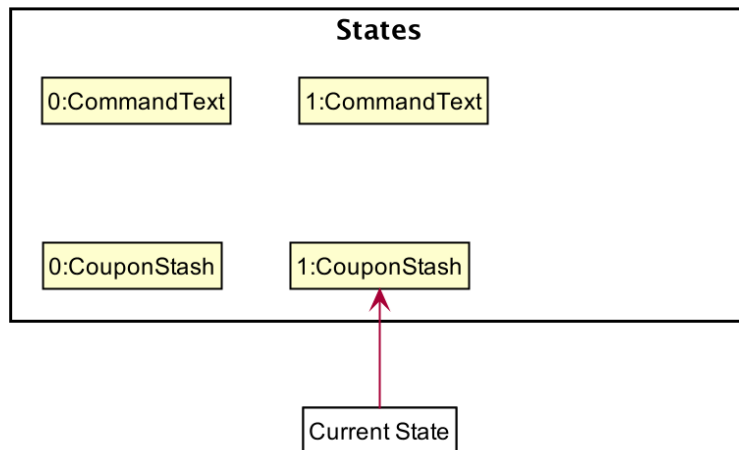


Figure 9. `currStateIndex` is shifted to the newly inserted coupon stash state.

Step 3. The user executes `add n/OMO STORE ...` to add a new coupon. The `add` command also calls `Model#commitCouponStash(String commandText)`, causing another modified coupon stash state and command text to be saved into the `couponStashStateList` and `commandTextHistory` respectively.

After command "add n/ OMO STORE..."

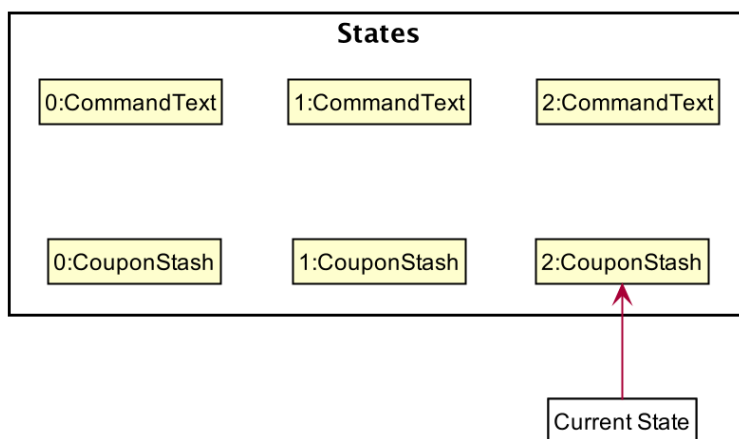


Figure 10. Modified coupon stash state and command text are saved into the `couponStashStateList` and `commandTextHistory` respectively.

#### NOTE

If a command fails its execution, it will not call `Model#commitCouponStash(String commandText)`, so the coupon stash state and command text will not be saved.

Step 4. The user now decides that adding the coupon was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoCouponStash()`, which will shift the `currStateIndex` once to the left, pointing it to the previous coupon stash state, and restores the coupon stash to that state. Plus, the command text is returned, thus allowing for the display of the command that was undone. In this case, the command undone is `add n/OMO STORE...`.

## After command "undo"

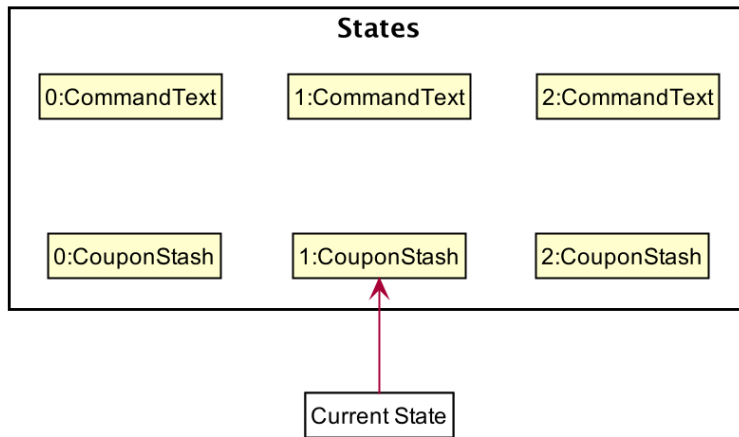


Figure 11. `currStateIndex` shifted once to the left.

### NOTE

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

The following sequence diagram shows how the undo operation works:

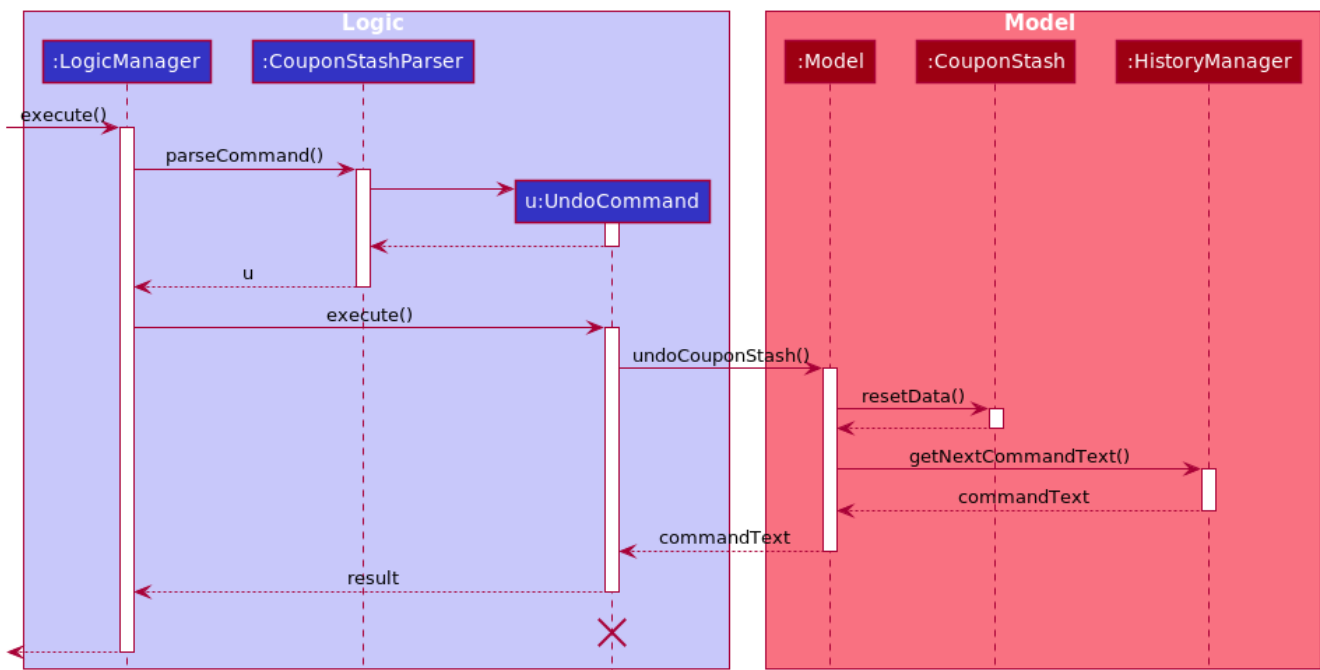


Figure 12. Undo operation sequence diagram.

The `redo` command does the opposite—it calls `Model#redoCouponStash()`, which shifts the `currStateIndex` once to the right, pointing to the previously undone state and command text, and restores the coupon stash to that state. Finally, it returns the redone command text.

## NOTE

If the `currStateIndex` is at index `couponStashStateList.size() - 1`, pointing to the latest coupon stash state, then there are no undone coupon stash states to restore. The `redo` command uses `Model#canRedoCouponStash()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command `list`. Commands that do not modify the coupon stash, such as `list`, will not call `Model#commitCouponStash()`. Thus, the `couponStashStateList` remains unchanged.

### After command "list"

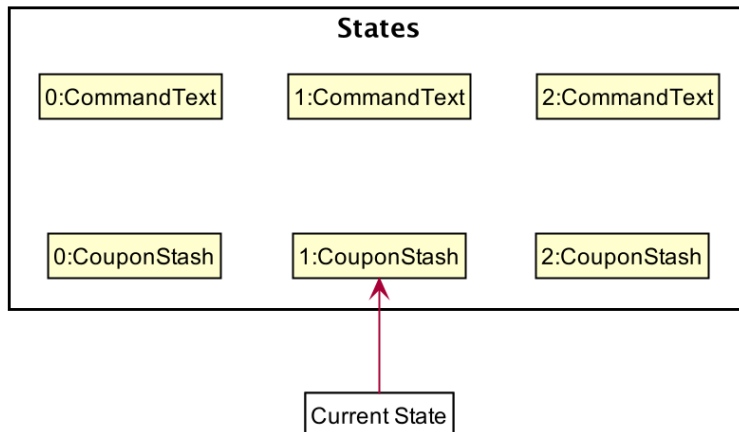


Figure 13. `couponStashStateList` remains unchanged.

Step 6. The user executes `clear`, which calls `Model#commitCouponStash()`. Since the `currStateIndex` is not pointing at the end of the `couponStashStateList`, all coupon stash states and command text history after the `currStateIndex` will be purged. We designed it this way because it no longer makes sense to redo the `add n/OMO STORE ...` command. This is the behavior that most modern desktop applications follow.

### After command "clear"

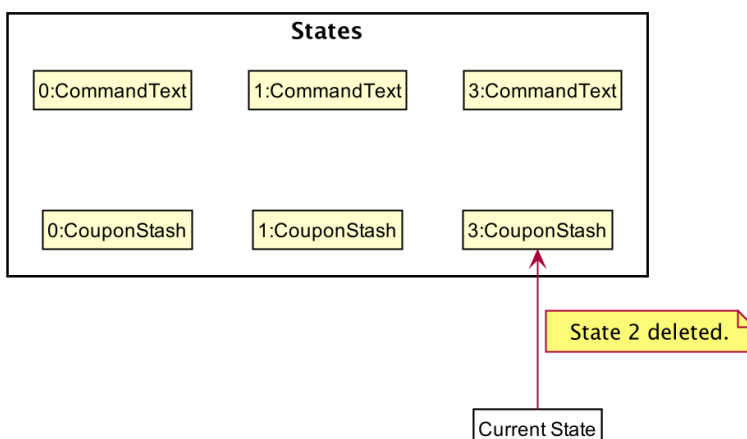


Figure 14. Command text history after the `currStateIndex` is purged.

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

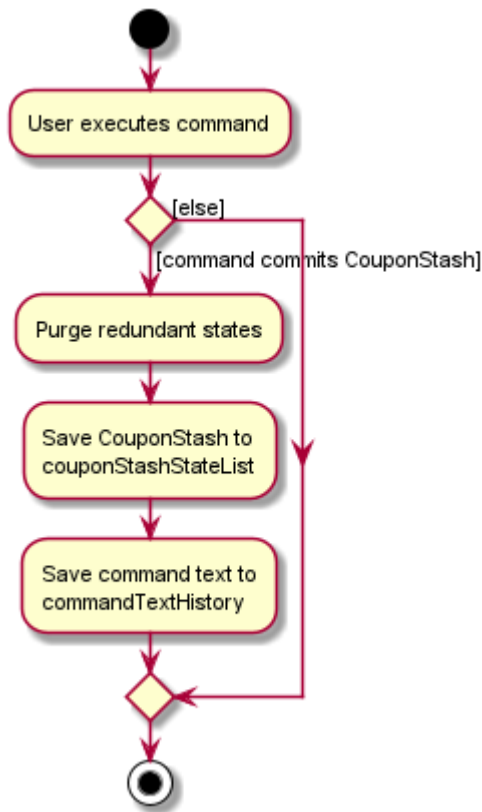


Figure 15. Activity diagram representation of how command texts are saved when they are executed.

### 3.1.2. Design Considerations

#### Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire coupon stash.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage. Plus, have to perform deep copy of coupons when saving the coupon stash so as to prevent unwanted mutations.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for **delete**, just save the coupon being deleted).
  - Cons: We must ensure that the implementation of each individual command is correct.

Alternative 1 was chosen due to its relative simplicity and extensibility. Little to no modification needs to be made to each command that can be undone, thus reducing chances of new bugs surfacing. Additionally, the ability to undo operations such as **clear** will require alternative 2 to copy the entire coupon stash too, so both alternatives will have the same memory footprint in such a context. Finally, the real world performance impact of copying all coupons vs copying one is not very huge. Thus, the more extensible and simpler alternative 1 was chosen.

## 3.2. Calendar

### 3.2.1. Current Implementation

The Calendar component provides a visual representation of the stored coupons that are expiring

over a month. It is facilitated by `CalendarPane`, `DateCell`, `ObservableList<Coupon>` and `ObservableMonthView`.

The `CalendarPane` is the controller of the Calendar on display. Users can change the month on display to show the coupons that expire during a specific month year by clicking on the arrows at the sides of the calendar's title or by using the `goto` command.

Each `DateCell` represents each date of the month that is currently on display. Each `DateCell` uses the `ObservableList<Coupon>` to keep a list of the coupon(s) that expires on each date. A `DateCell` with coupon(s) expiring on the date are highlighted in red and a `Datecell` that represents the `system's date` is highlighted blue.

The `ObservableList<Coupon>` is the list of filtered coupons that are currently on display in the `CouponListPanel`. They are obtained by calling the `Logic#getFilteredCouponList()` method. The list can be filtered to view all active, archived or used coupons using the `expiring` command.

The **ObservableMonthView** is the current month & year on display in the **Calendar Pane**. It is obtained by calling the **Logic#getMonthView()** method.

The class diagram below shows the interaction between classes that affects the Calendar:

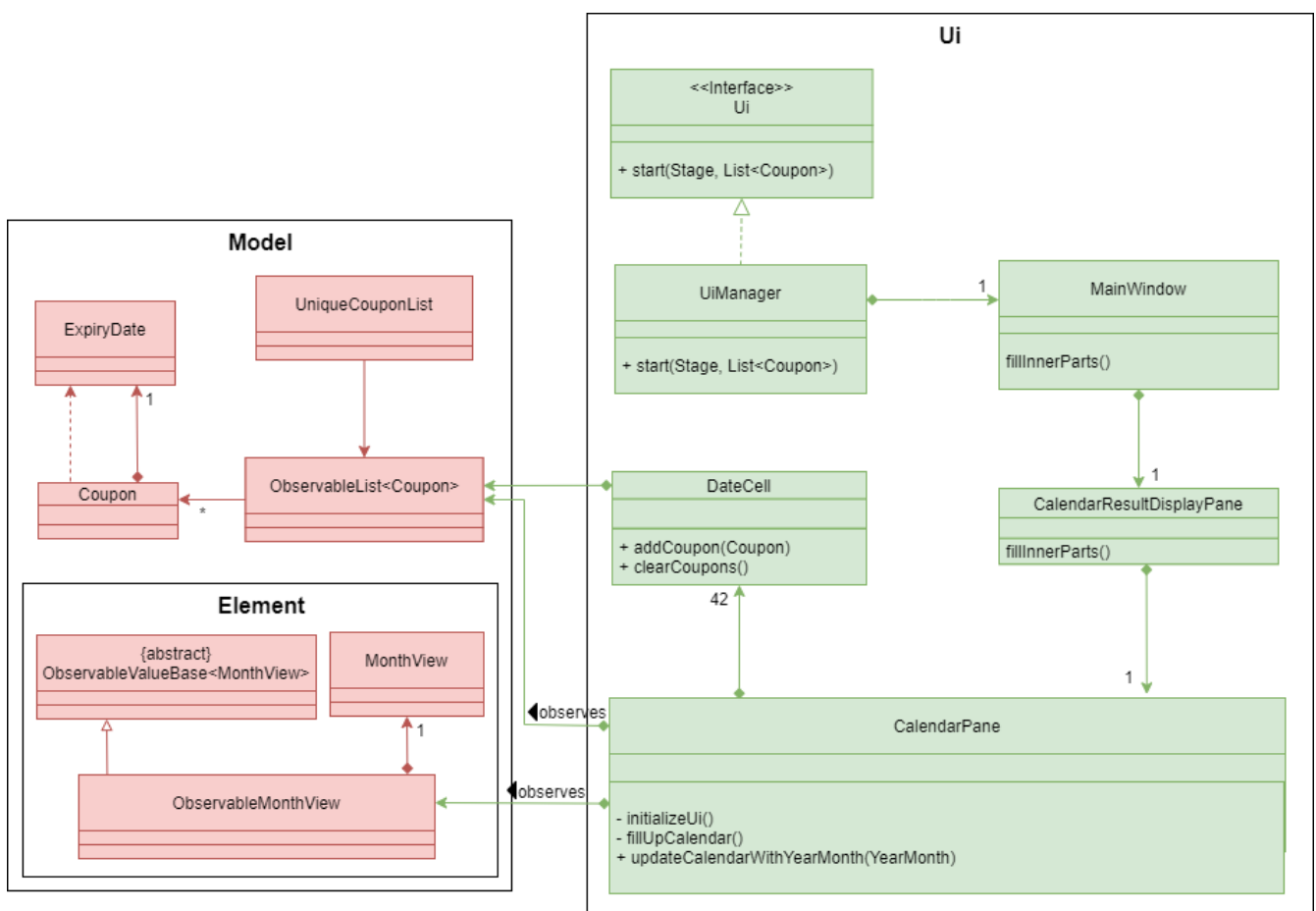


Figure 16. Overview of the class diagram representation of the Calendar.

The sequence diagrams below show how the Calendar works:

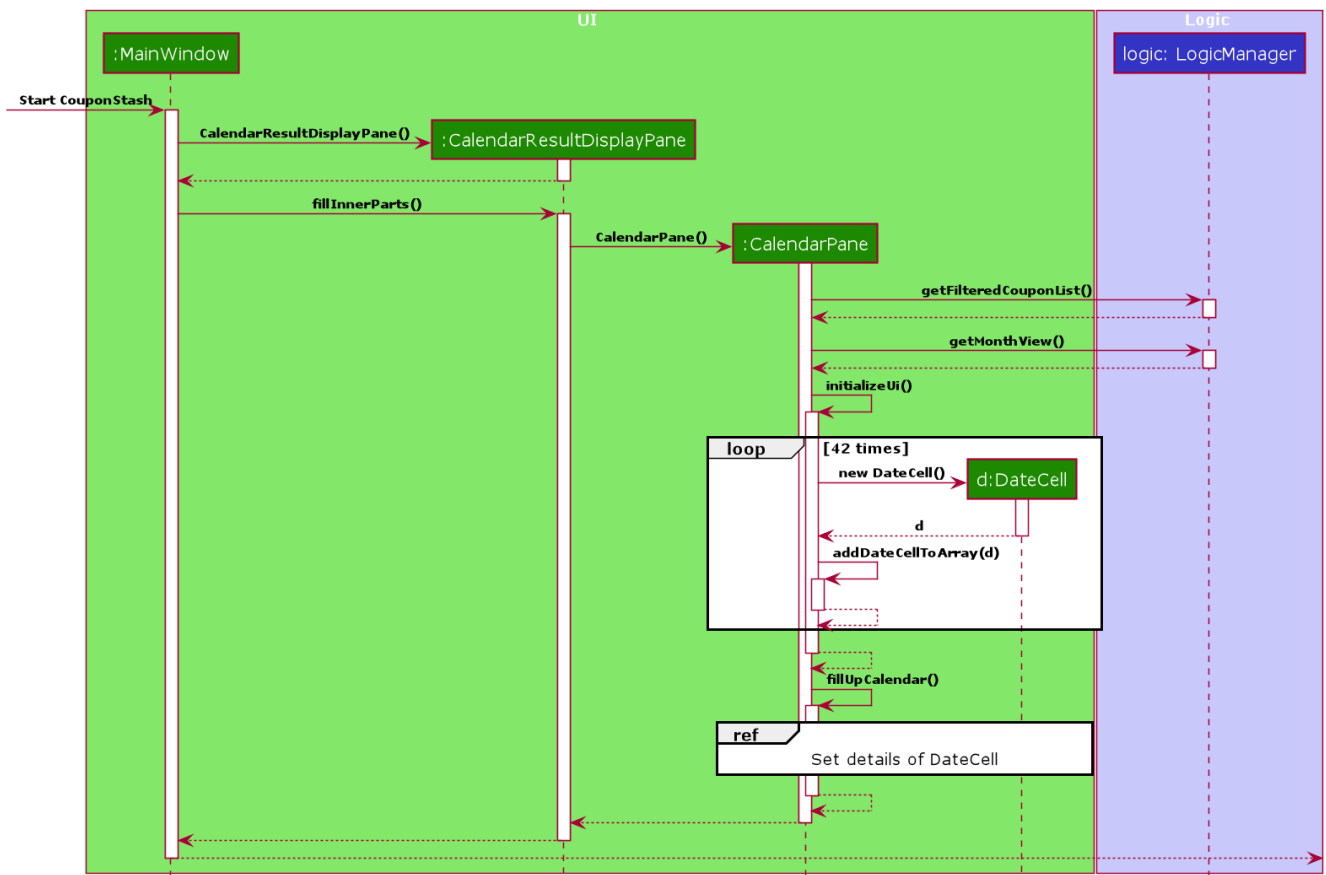


Figure 17. Sequence diagram representation of the Calendar on the startup of Coupon Stash.

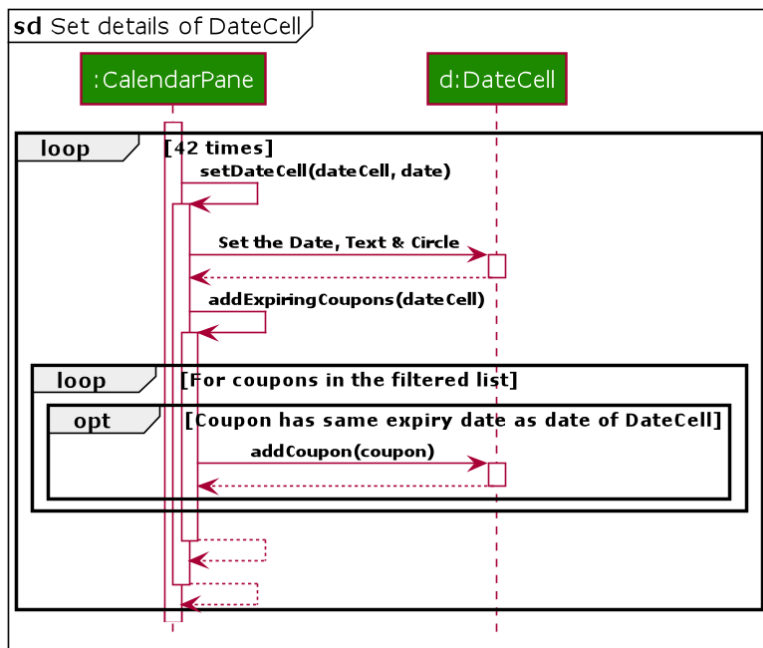


Figure 18. Sequence diagram representation of the Set details of DateCell ref frame of Calendar (Applicable to the next two diagrams).

The two scenarios below are examples of how the Calendar mechanism behaves at each step of each scenario.

### Updating the Calendar with an Updated List

The Calendar updates with the current `ObservableList<Coupon>` with commands such as the `add`,



archive, clear, delete, edit, expiring, find, list, redo, unarchive, undo and used. The following steps describes how this behavior is implemented.

Step 1. The user launches the application for the first time.

The Calendar displayed will render the saved coupon data, triggered by the initiation of the UiManager.

Step 2. The user executes a command that alters the `ObservableList<Coupon>` (any command listed above).

When a command alters the observable coupon list, the listener of the observable list detects the change and the Calendar will be updated accordingly to the list by calling the `CalendarPane#fillUpCalendar()` method.

For example, the `find` command alters the observable coupon list. It calls the `FindCommand#execute(Model, String)` method, which calls the `Model#updateFilteredCouponList(Predicate)` method. It then calls the `FilteredList<Coupon>#setPredicate(Predicate)` method that alters the observable coupon list.

#### NOTE

If a command fails its execution, it will not call the `FilteredList<Coupon>#setPredicate(Predicate)` method. Hence, the observable coupon list will not be altered and the calendar will not be altered.

The following sequence diagram shows how the Calendar updates with the observable coupon list:

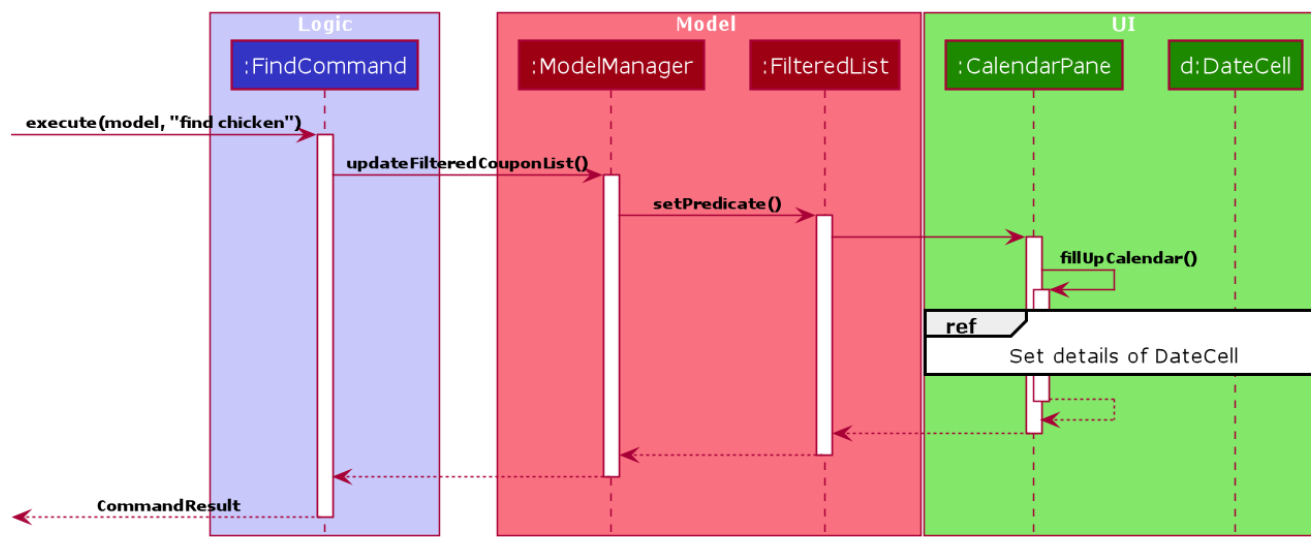


Figure 19. Sequence diagram representation of the update of the Calendar with the Coupon List for the "find chicken" Command.

### Updating the Calendar with a Different Month View

The Calendar updates with the current `ObservableMonthView` with commands such as `goto`, `expiring` and `list` or by clicking on the arrows at the sides of the calendar title. The following steps describes how this behavior is implemented.

Step 1. The user launches the application for the first time.

The Calendar displayed will render the saved coupon data, triggered by the initiation of the UiManager. The default calendar display will be set to the **system's month year**.

Step 2. The user executes a command that alters the **ObservableMonthView** (any command listed above).

When a command alters the observable month view, the listener of the observable month view detects the change and the month view display of the calendar will be updated according by calling the **CalendarPane#updateCalendarWithYearMonth** method.

For example, the **goto** command calls the **GoToCommand#execute(Model, String)** method, which calls the **Model#updateMonthView(String)** method. It then calls the **ObservableMonthView#setValue(String)** method that alters the observable month view.

- **expiring** command
  - For the **expiring** command, the Calendar will be updated accordingly to the month year of the specified date or month year with the command.
  - For example, entering these **expiring** commands **expiring my/9-2020** or **expiring e/11-9-2020** will change the month year on display to September 2020.
- **list** command
  - For the **list** command, the Calendar will be updated to the **system's month year**.

#### NOTE

If a command fails its execution, it will not call the **ObservableMonthView#setValue(String)** method. Hence, the observable month view will not be altered and the calendar will not be altered.

The following sequence diagram shows how the Calendar updates with the observable month view:

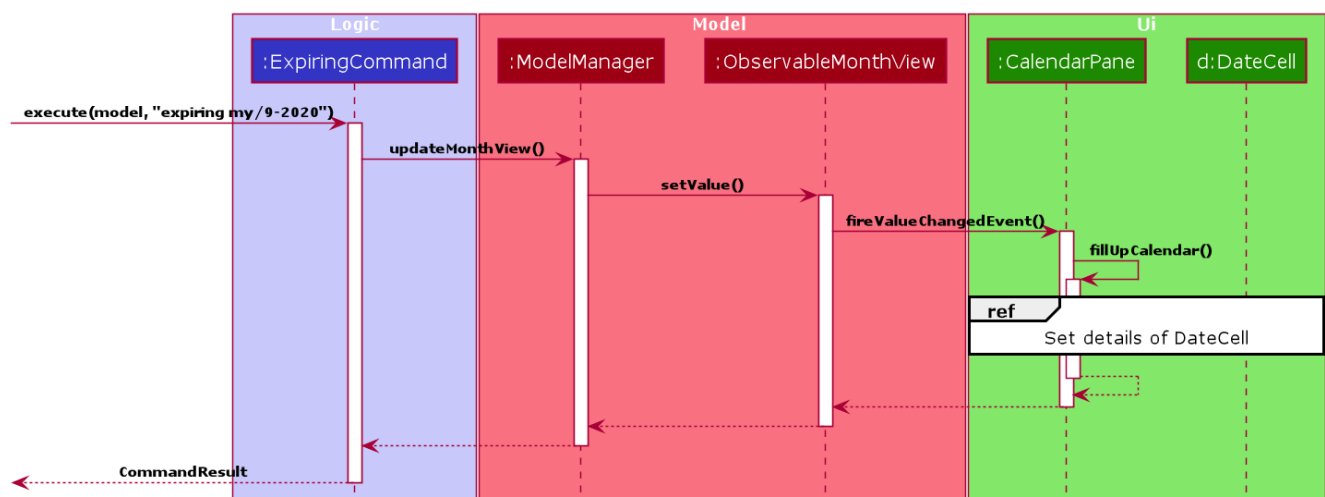


Figure 20. Sequence diagram representation of the update of the Calendar's MonthView for the "expiring chicken" Command.

Or alternatively, instead of step 2,

Step 3. The user clicks on the arrows at the sides of the calendar title to change the month year displayed.

When a click alters the observable month view, the listener of the observable month view detects the change and the month view display of the calendar will be updated according by calling the `CalendarPane#updateCalendarWithYearMonth` method.

For example, clicking on the arrow on the right calls the `CalendarPane#changeCalendarToNextMonth` method, which calls `CalendarPane#updateCalendarToNextMonth`. It then calls the `ObservableMonthView#setValue` method that alters the observable month view.

### 3.2.2. Design Considerations

#### Aspect: Information displayed on the Calendar

- **Alternative 1 (current choice):** Show expiring coupons by highlighting the dates with expiring coupon(s)
  - Pros: Cleaner view of the Calendar with minimal information & may take up less space on the `Main Window`
  - Cons: Lesser information provided with a glance
- **Alternative 2:** Show a condensed version of the coupons' details within the cell of each date
  - Pros: More information provided with a glance
  - Cons: Messy to look at when there are multiple coupons expiring on a date & may take up more space on the `MainWindow`

We decided on alternative 1, to show coupons expiring on specific dates with highlights. This is because a coupon contains much information and the calendar may look cluttered and messy, which may be aesthetically unpleasant to the user. Furthermore, the user can use the `expiring` command to search for coupons expiring on a date or month year and have a more detailed view of the coupons in the `CouponListPanel`.

#### Aspect: Whether the Calendar should update with the list

- **Alternative 1 (current choice):** Calendar updates with the filtered list
  - Pros: User can easily relate and reference to the coupons shown in the Calendar to the `CouponListPanel`
  - Cons: May overlook some coupons if the list is filtered
- **Alternative 2:** Calendar shows all the coupons in `CouponStash`
  - Pros: View of all coupons and will not overlook any coupons even when the coupon list is filtered
  - Cons: User may be confused if he/she sees a highlighted date on the Calendar when there is no coupon expiring on that date in the `CouponListPanel`

We decided on alternative 1, for the calendar to update with the list in the `CouponListPanel`. This is because this follows the Observer Pattern Design Principle. Furthermore, this will not confuse the user when the user sees a highlighted date on the Calendar when there is no coupon expiring on that date in the `CouponListPanel`.

## 3.3. Coupon Archiving

When physical coupons are expired or exhausted, they would usually be thrown away, or kept in the archive. Coupon Stash simulates this archive, storing these coupons in the app so that the user can still keep track of it, and the savings they generated.

### 3.3.1. Current Implementation

The archiving of coupons is facilitated by the **Archived** attribute of a coupon. The following methods in the **CouponStash**, **Coupon**, **Usage**, **UsedCommand** class and the **Model** interface facilitates this feature:

- **CouponStash#archiveExpiredCoupons()** — Archives any coupon in the **CouponStash** that has expired, and returns a new updated **CouponStash**.
- **Coupon#increaseUsageByOne()** - Increases the usage of a coupon by one.
- **Usage#isAtLimit** - Returns true if the current usage is at its limit (abstracted by the **Limit** field).
- **UsedCommand#execute()** - Executes the **used** command input by the user.
- **Model#PREDICATE\_SHOW\_ALL\_ACTIVE\_COUPONS** - A **Predicate** function that filters out archived coupons from a given **CouponStash**.

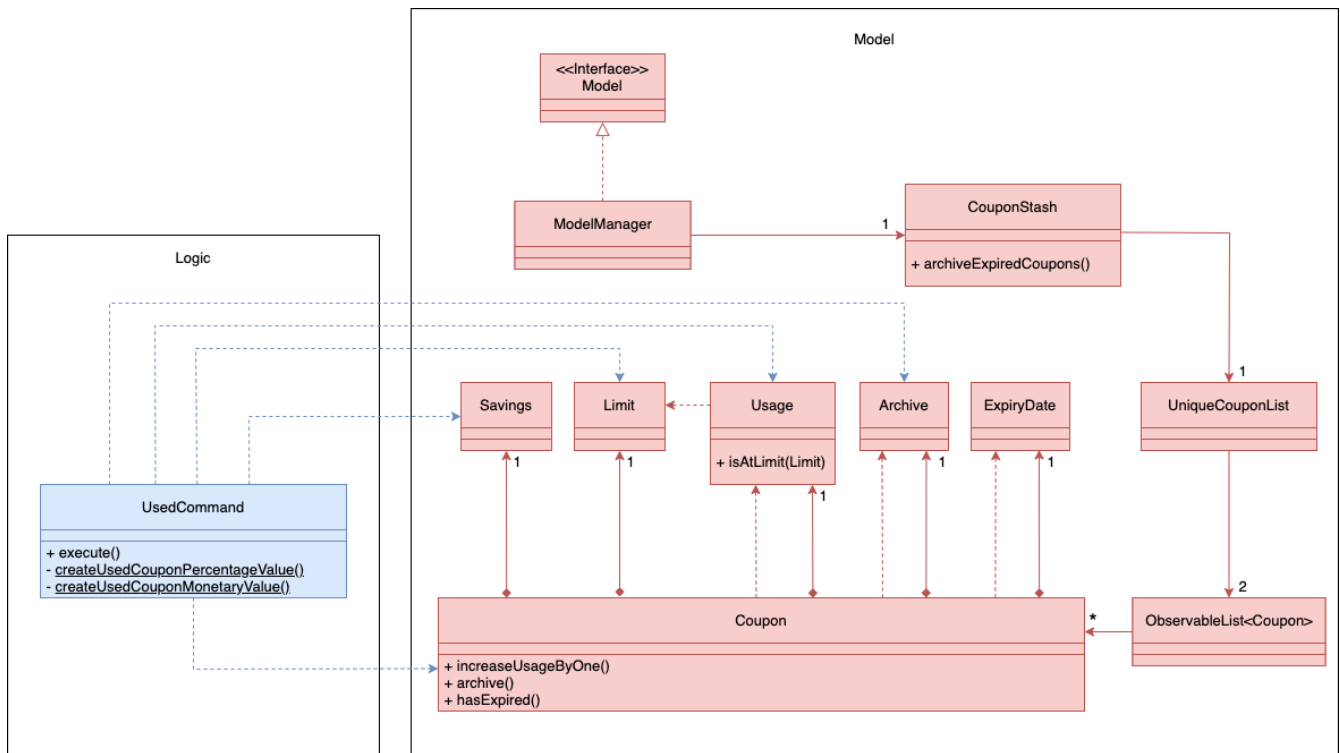


Figure 21. Overview class diagram representation of the coupon archiving implementation.

Given below is two example usage scenarios and how the archiving mechanism behaves at each step of each scenario. An activity diagram is provided first to describe the general events that will lead to an automatic archiving of coupons by Coupon Stash.

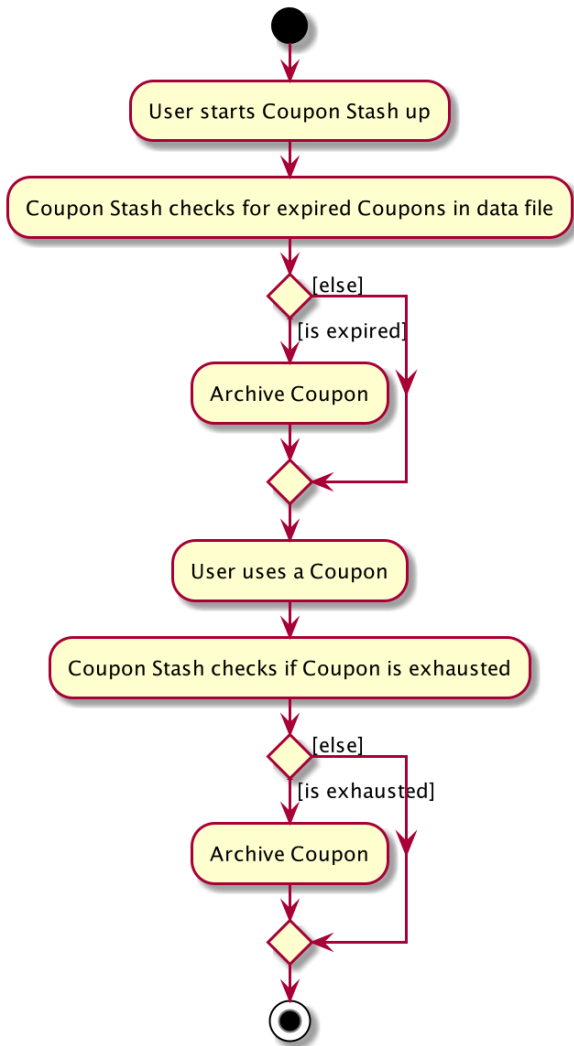


Figure 22. Activity diagram representation of the general flow of archiving of coupons in Coupon Stash

### Archiving of Expired Coupons

Expired coupons are automatically archived by Coupon Stash upon start up of the application. The following steps describe how this behaviour is implemented.

Step 1. The user launches the application for the first time. The initiation of `ModelManager` will also trigger the initiation of `CouponStash` with any available saved data.

Step 2. The method `CouponStash#archiveExpiredCoupons` will be called from the newly initiated `CouponStash`, and have its `UniqueCouponList` mapped to a function that archive coupons that has expired before the date of opening the application, and returns a new updated `CouponStash`. This mapping function is facilitated by `Coupon#hasExpired()` and `Coupon#archive()`.

Step 3. The `ModelManager` will proceed to filter out the archived coupons from the newly updated `CouponStash`, and return a filtered list of active coupons. This filtering is facilitated by the predicate `Model#PREDICATE_SHOW_ALL_ACTIVE_COUPONS`.

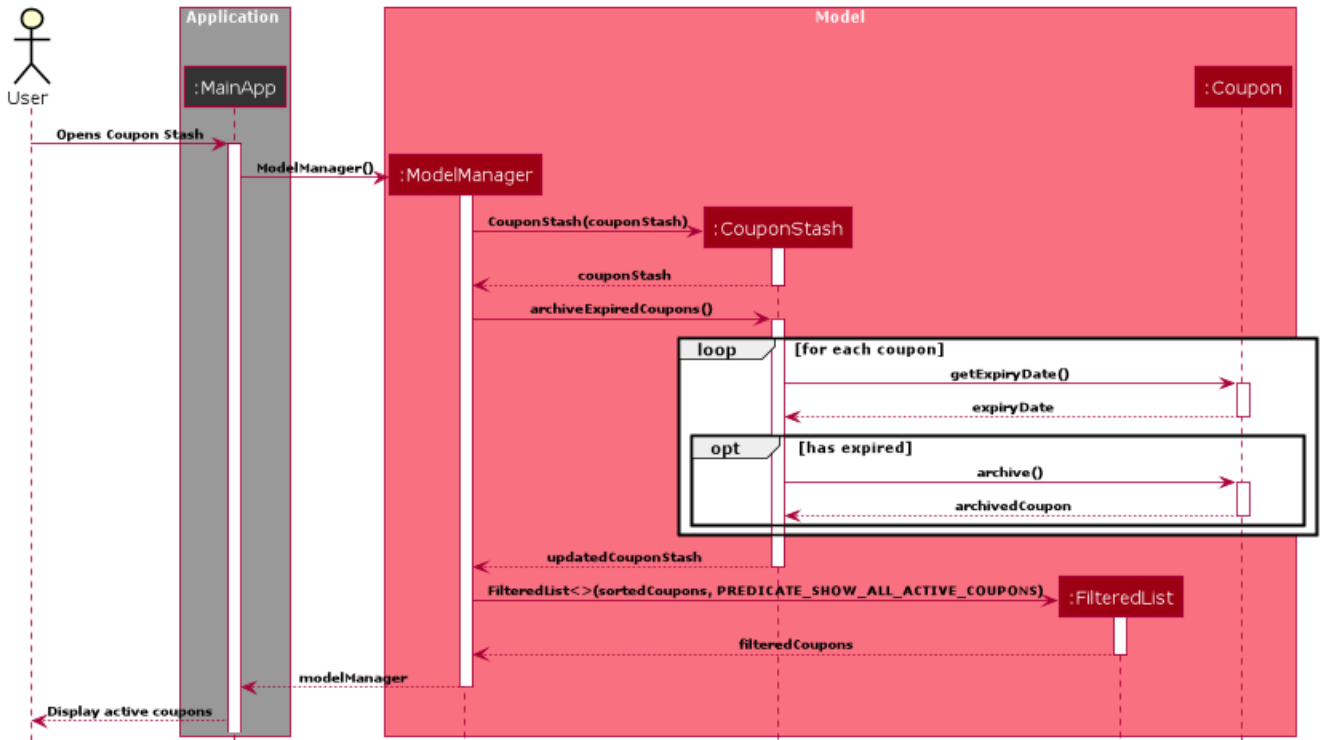


Figure 23. Sequence diagram representation of archiving expired coupons

## Archiving of Exhausted Coupons

Coupons that have exhausted its usages will be automatically archived by the application. The following steps describe how this behaviour is implemented.

Step 1. The user uses a **Coupon** in the current observable **CouponStash** with the command **used 1**. **UsedCommand** is created with the parsed arguments, and executed. The particular **Coupon** will then have its **Usage** increased by one by calling **Coupon#increaseUsageByOne()**.

Step 2. The **Coupon** will then be checked if its **Usage** has reached its **Limit**, using the **Usage#isAtLimit()** method. For the purpose of this explanation, we assume that the coupon being used has a usage **Limit** of 1 and a previous **Usage** value of 0, with savings in **MonetaryAmount**.

Step 3. The **Coupon** will have a new **Archived** value, which will be set to **true** if the **Usage** has indeed reached its **Limit**. This is facilitated by **Coupon#archive()**.

Step 4. The **CouponStash** will be updated with this used **Coupon** with the **ModelManager#setCoupon()** method. Under the hood of this method, the current **FilteredList** will be updated to show active **Coupons** only, facilitated by the predicate **Model#PREDICATE\_SHOW\_ALL\_ACTIVE\_COUPONS**.

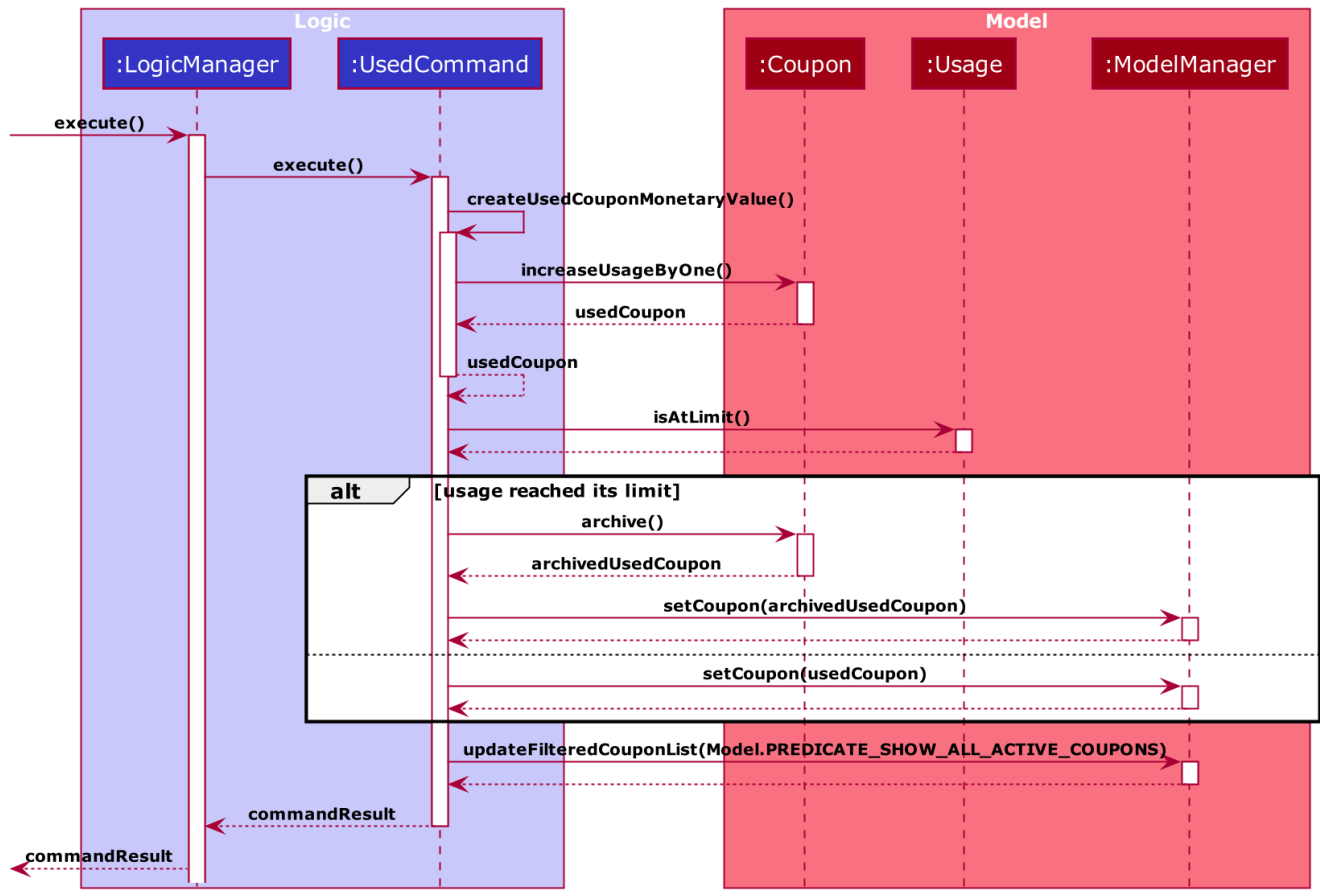


Figure 24. Sequence diagram representation of archiving exhausted coupons

### 3.3.2. Design Considerations

#### Aspect: The implementation to store archived coupons

- **Alternative 1 (current choice):** **Coupon** contains an **Archived** field
  - Pros: Easy to implement, lower maintainability.
  - Cons: Saved data may get considerably huge after heavy usage of application.
- **Alternative 2:** Archived **Coupons** are stored in another separate data file.
  - Pros: Separates the logic between the two different **CouponStash**, e.g. ability to limit the functions on archived **Coupons**
  - Cons: Sharply increases the maintainability and coupling of the application with two data files.

Alternative 1 was chosen, due to the cons of Alternative 2. While a separate file is akin to having two separate stashes of coupons, this would increase the overall complexity of the application. **Logic** and **Model** would have to deal with another set of data, and **Commands** may have to split up the logic for different data sets. Furthermore, while saved data will be larger for Alternative 1, it should only affect the performance of starting Coupon Stash up, since most of the interactions with the program is with active coupons.

## 3.4. Up/down arrow retrieve command history

### 3.4.1. Current Implementation

The retrieving of command history via the up and down arrow keys is facilitated by the `CommandTextHistory` class. The command history is stored internally as a `LinkedList` used as a stack with a `currIndex` tracking the next command in the history to return. The following methods and attributes in the `CommandTextHistory` class facilitates this feature:

- `CommandTextHistory#add(String commandText)`
- `CommandTextHistory#getDown()`
- `CommandTextHistory#getUp()`
- `CommandTextHistory#commandTextHistory`
- `CommandTextHistory#currIndex`

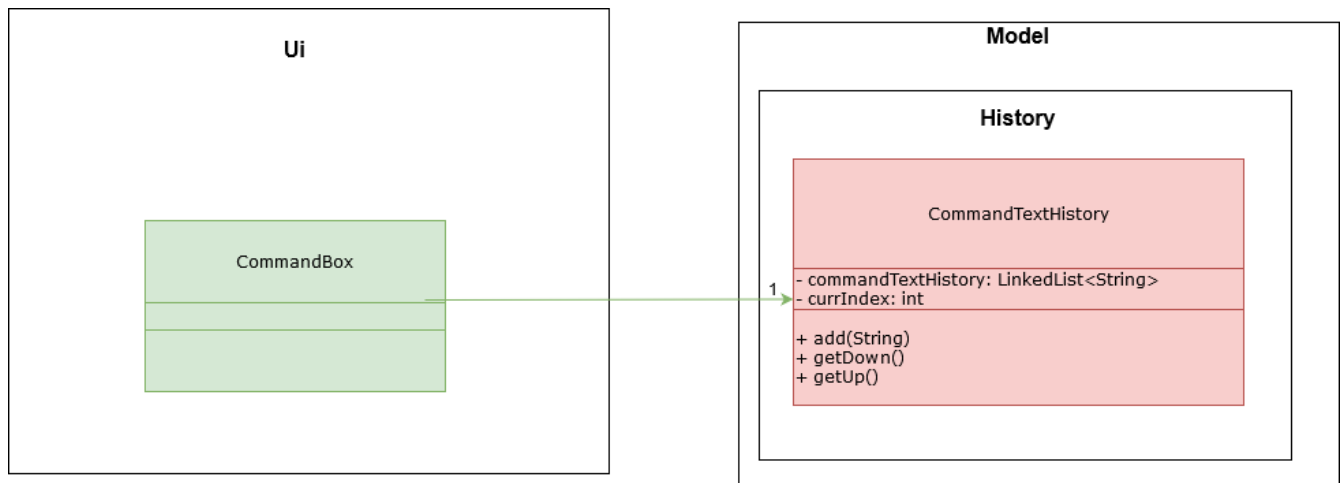


Figure 25. Class diagram representation of the command history retrieving function.

Given below is an example usage scenario and how the up/down button presses behaves at each step.

Step 1. The user launches the application for the first time. The `CommandTextHistory` is initialized with a stack containing only an empty string (`""`), and the `currIndex` is set to `0`.

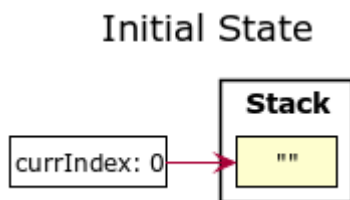


Figure 26. Stack containing only an empty string

Step 2. The user executes `delete 1`. `CommandBox#handleCommandEntered()` will call `CommandTextHistory#add(String commandText)` to save the entered command into the stack contained in `CommandTextHistory`. The top of the stack (i.e. the empty string) is popped off first, before the entered command is pushed onto the stack. Then, the empty string is pushed onto the stack again, thus ensuring that the empty string stays at the top of the stack. Note that `currIndex` is not affected.



## After command "delete 1"

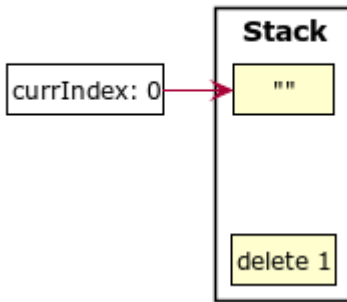


Figure 27. Stack after executing `delete 1`

Step 3. The user executes `delete 2`. `CommandBox#handleCommandEntered()` will also save the entered command into the stack contained in `CommandTextHistory`. As in the previous step, the new command is pushed to the top of the stack, just below the empty string.

## After command "delete 2"

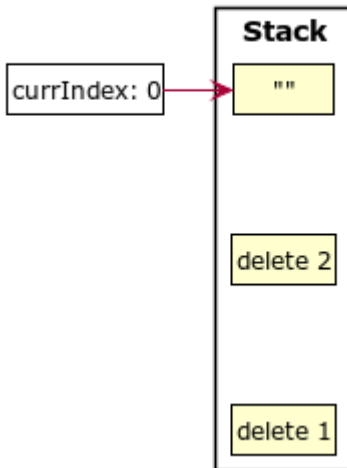


Figure 28. Stack after executing `delete 2`.

Step 3. Now, the user decides to delete the second coupon again. We press the arrow key up once, and `CommandBox#commandTextField` has a listener that calls `CommandTextHistory#getUp()`. The `currIndex` is incremented, and then the command text pointed to by `currIndex` is returned and displayed in the program command box.

After pressing "up" arrow key

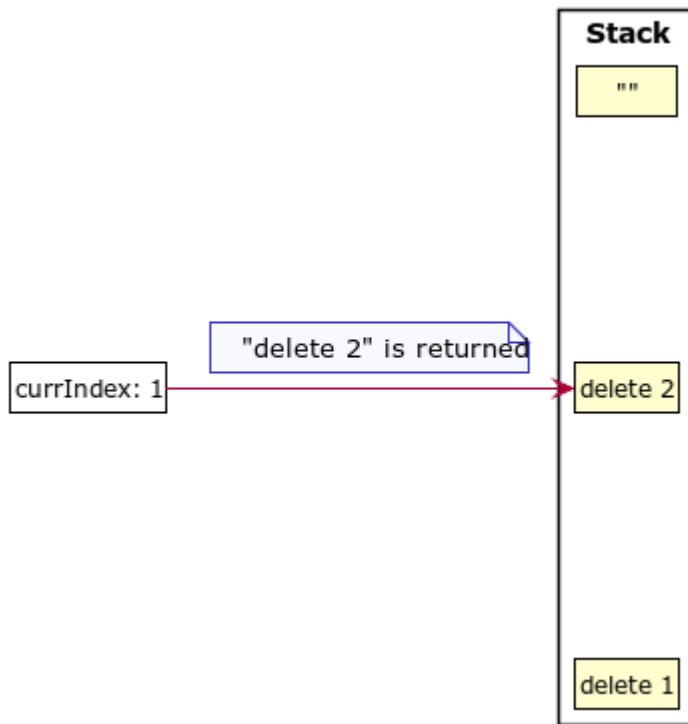


Figure 29. After pressing the "up" arrow key.

Step 4. The user then executes the retrieved command (**delete 2**). As in the previous steps, this newly executed command is pushed to the top of the stack just below the empty string. However, in such a case when the **currIndex** is not 0 and does not point to the top of the stack, it is reset to 0.

After executing "delete 2" again

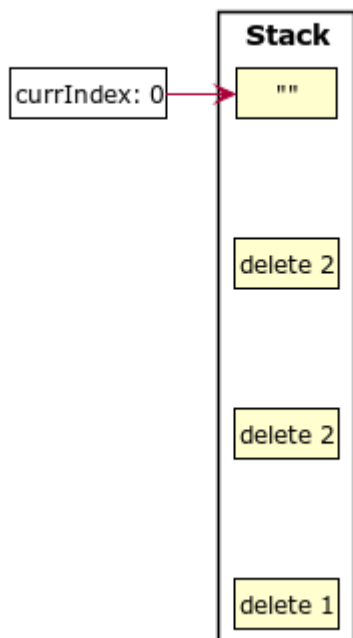


Figure 30. Stack after executing **delete 2** again.

**NOTE**

If the **currStateIndex** is pointing to the top of the stack, then there are no previous commands to retrieve. Thus, the up button will simply return the empty string. No changes to the stack and **currIndex** will be effected.

The down arrow key does the opposite, it will lead to the calling of `CommandTextHistory#getDown()`, which shifts the `currIndex` one item higher (i.e. decrement the `currIndex` by 1), before returning the command text pointed by the updated `currIndex`.

**NOTE**

If the `currIndex` is at index `commandTextHistory.size() - 1`, pointing to the bottom of the stack, there is no next command to retrieve when pressing the down key. Thus, the down button will simply return the command text currently being pointed to by the `currIndex`. No changes to the stack and `currIndex` will be affected.

Below is a sequence diagram describing the events that happen when a user presses a key.

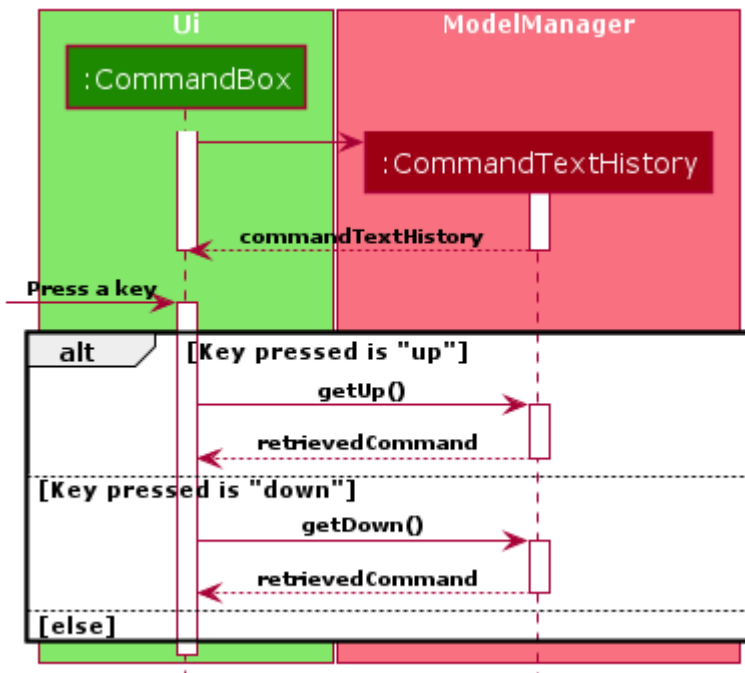


Figure 31. Sequence diagram representing retrieval of command text history with the up and down arrow keys.

Below is a sequence diagram describing the events that happen when a executes a command text, thus triggering the saving of a command text into `CommandTextHistory`.

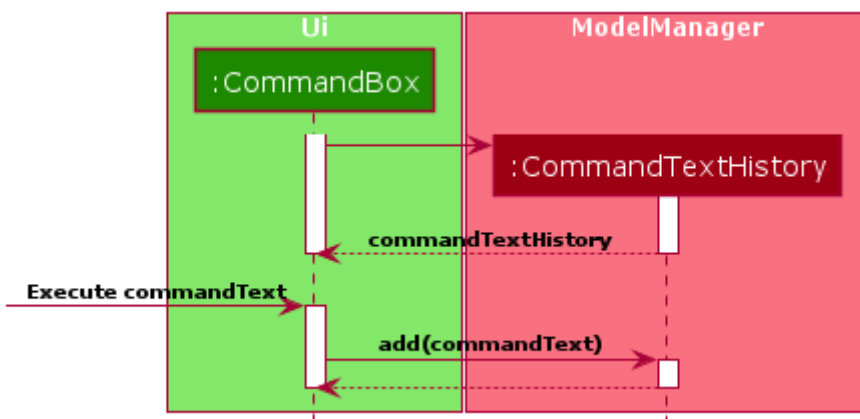


Figure 32. Sequence diagram representing the saving of a command text.

### 3.4.2. Design Considerations

#### Aspect: Data structure to support the key actions

- **Alternative 1 (current choice):** Use `LinkedList` as a stack to store the command text history.
  - Pros: `LinkedList` is a better data structure that allows for more efficient operations supported by stacks.
- **Alternative 2:** Use `ArrayList` as a stack to store the command text history.
  - Pros: `ArrayList` is more recognizable to people who are relatively new to Java, thus reducing confusion.
  - Cons: Stack operations are less efficient on `ArrayList` s.

## 3.5. Coupon Sorting

### 3.5.1. Current implementation

The sorting of coupons in the coupon stash is facilitated by the following static variables in the `SortCommand` class and this methods in the `Model` interface and `SortedList` class.

- `SortCommand#NAME_COMPARATOR` - Comparator that sorts coupons by name in ascending order.
- `SortCommand#EXPIRY_COMPARATOR` - Comparator that sorts coupons by expiry date in ascending order.
- `SortCommand#REMINDER_COMPARATOR` - Comparator that sorts coupons by remind date in ascending order.
- `Model#sortCoupons(Comparator<Coupon> comparator)` - Sorts the `ObservableList` of coupons that are stored in `Model` according to the order decided by the passed in `comparator`.
- `SortedList#setComparator(Comparator<Coupon> comparator)` - Sets the comparator that determines the order of the coupons inside the sorted list.

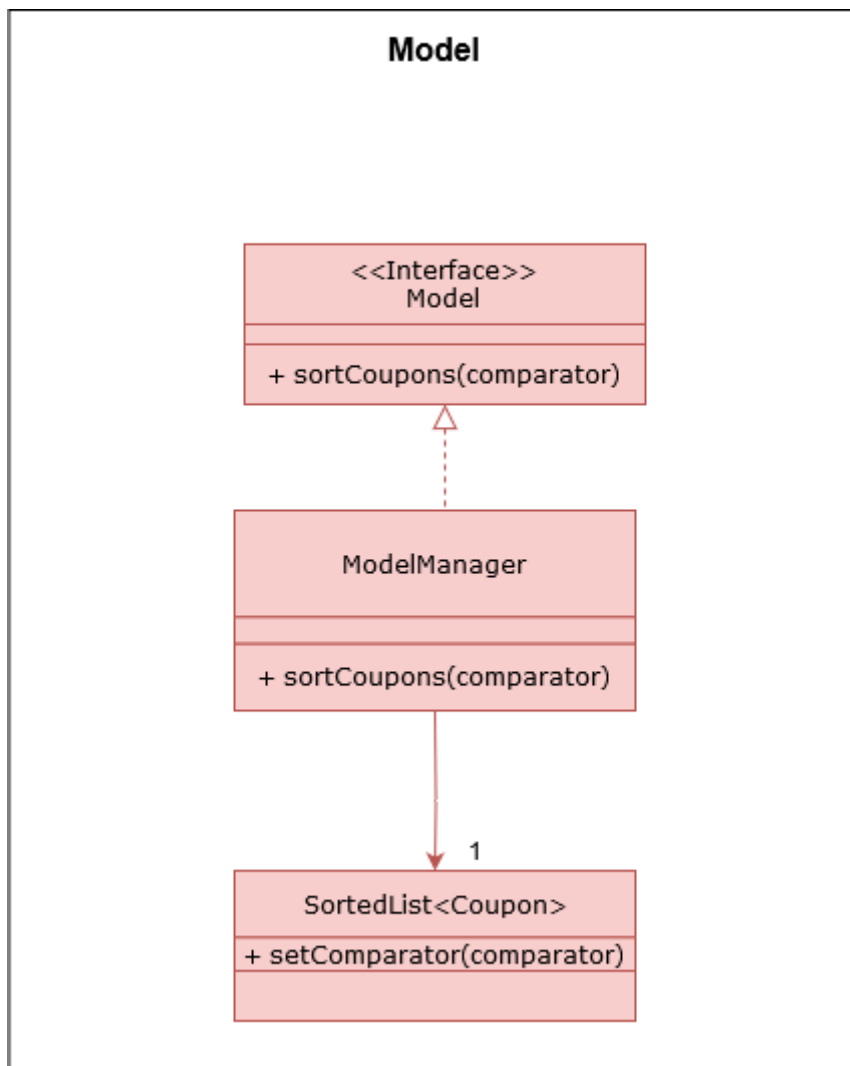


Figure 33. Overview of the class diagram representation of the coupon sorting implementation.

When a **sort** command is executed, the field to sort by is indicated by the inputted prefix. The sequence diagram below describes what happens when a **sort** command is run.

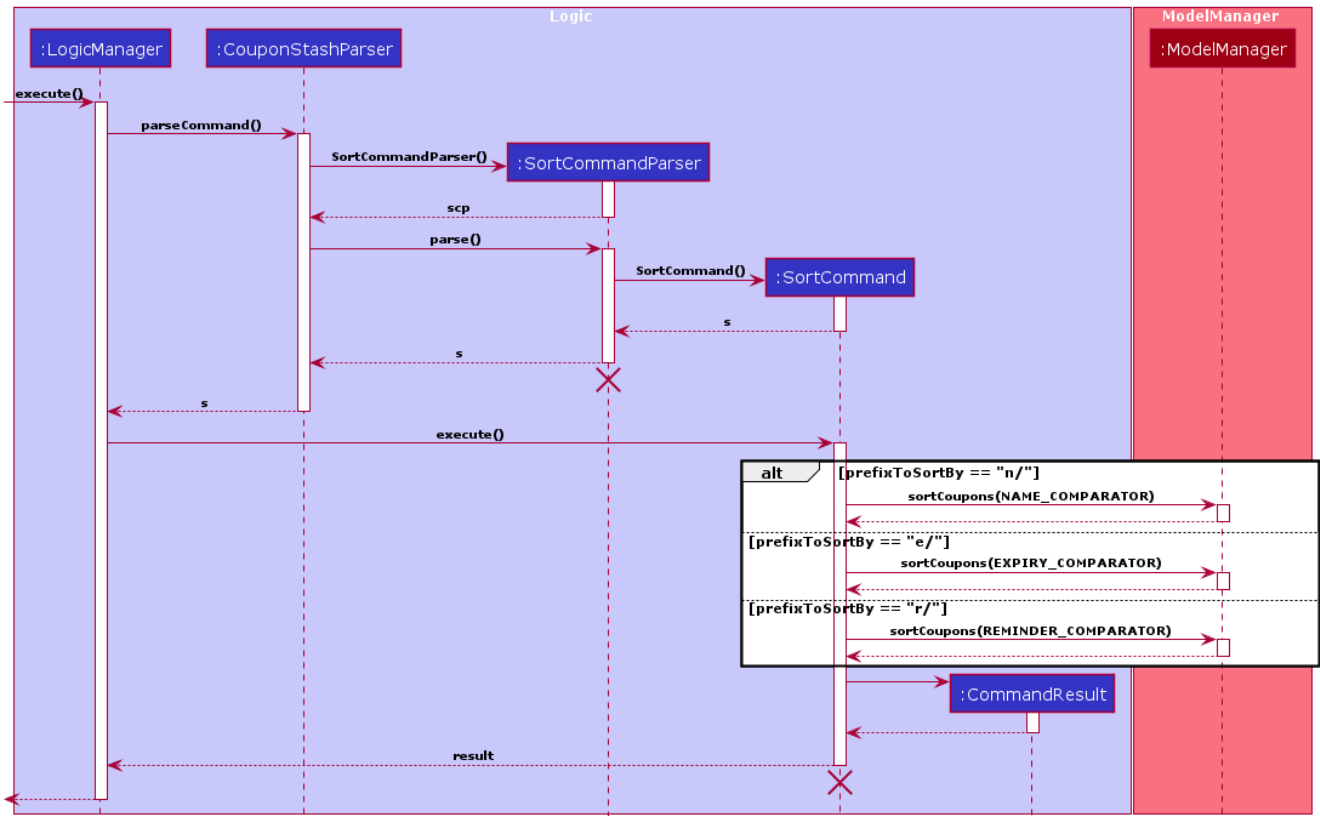


Figure 34. Sequence diagram describing the process of sorting coupons.

Depending on the prefix to sort by, `ModelManager#sortCoupons()` will be called with the relevant comparator as its argument. The `ModelManager#sortCoupons()` method subsequently calls the `SortedList#setComparator()` method (not shown in the above diagram), which leads to a change of the comparator of the `SortedList` stored in `ModelManager`, thus triggering a sort of the `SortedList`.

### 3.5.2. Design Considerations

#### Aspect: Persistent or non - persistent sort?

- **Alternative 1 (current choice):** Make sorting non - persistent.
  - Pros: Sorting is faster as no write to disk is needed to make the new order persistent. Additionally, with the coupons being sorted by the time they are added to the coupon stash by default, there is no way to restore this order without storing the time a coupon was added to the stash. Thus, the non - persistent approach shines here as restoring the original order of the coupon stash is as trivial as reopening the program.
  - Cons: If a user prefers a particular default sorting order for their coupons, they have to retype the `sort` command each time the program is launched or each time a coupon is added or edited.
- **Alternative 2:** Make sorting persistent.
  - Pros: Gives users more freedom over the default order of their coupons.
  - Cons: Can be unnecessarily complicated to implement a hidden field stating a coupon's addition time just so users can revert to the default order. Additionally, it can be confusing to users when there are so many different ways to sort.

In our usage during testing, we have never had the urge to have a default sorting order when the program is launched. Plus, we feel that the simplicity of excluding a sort by default order function will be well favored by users, and thus we chose alternative 1.

## 3.6. Coupon Reminder

To ensure users are aware of expiring coupons and maximise their saving, Coupon Stash reminds the user through a pop-up window, upon launching the application.

To achieve this feature, the following methods in the `RemindDate` and `RemindWindow` classes are used.

- `RemindDate#isToday()` - Check if the `RemindDate` is today.
- `RemindWindow#filterRemindCoupons()` - Filters out all `RemindDates` that are not today from `RemindWindow`.
- `RemindWindow#constructRemindCoupons()` - Creates a `String` of coupons that have their `RemindDates` today. This `String` is used in the displayed reminder window.
- `RemindWindow#showIfAny()` - Shows the reminder window if there are coupons to be reminded of today. If there are no coupons that have to be reminded today, no window will be shown.

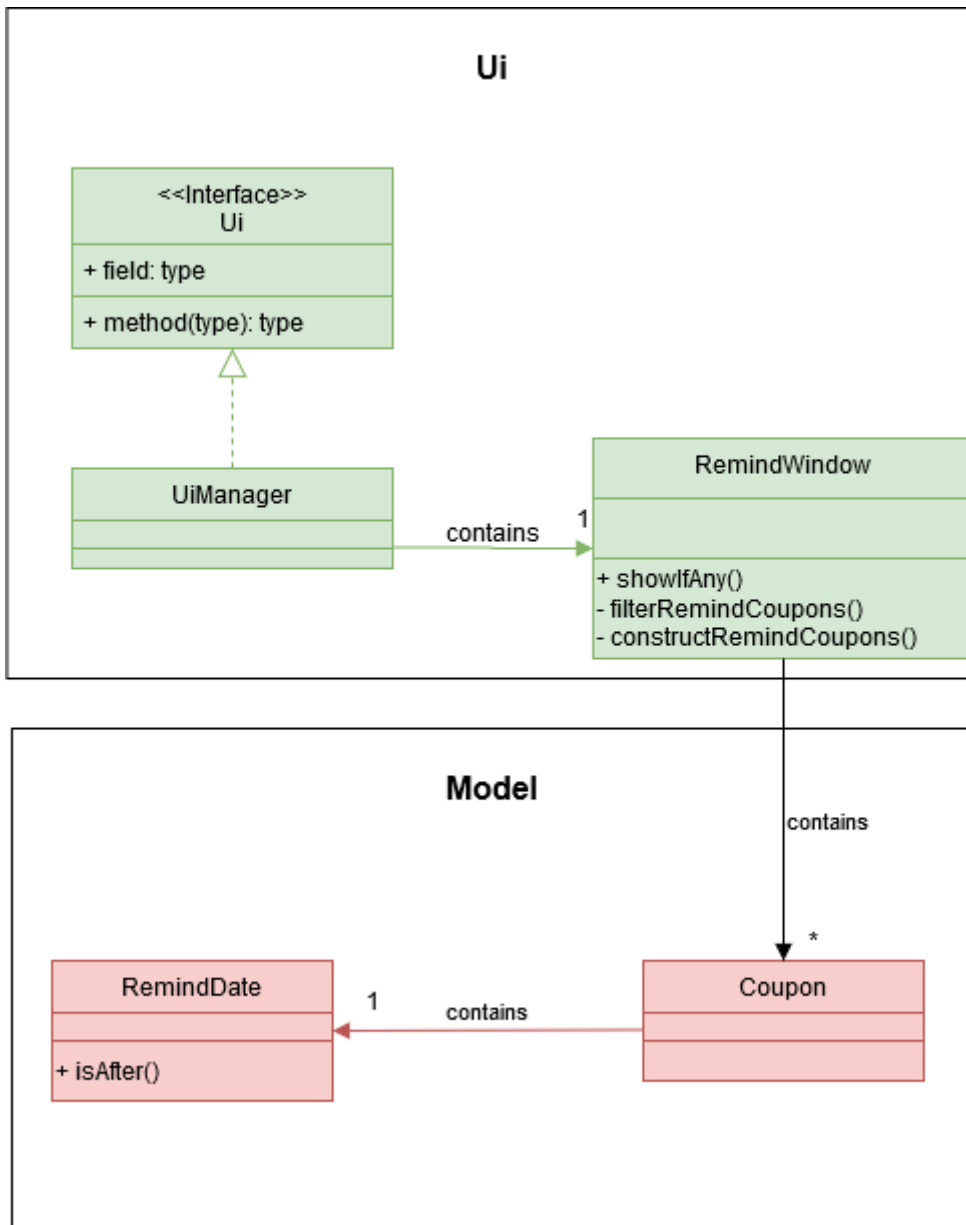


Figure 35. Overview of the class diagram representation of the reminder checking implementation.

To make sense of how coupon reminder functions, let's dive into the specifics of RemindDate class, RemindCommand class and RemindWindow class.

### 3.6.1. Implementation of editing a coupon's RemindDate

The following activity diagram depicts what happens when the user runs a **edit** command to edit a coupon's **RemindDate**.



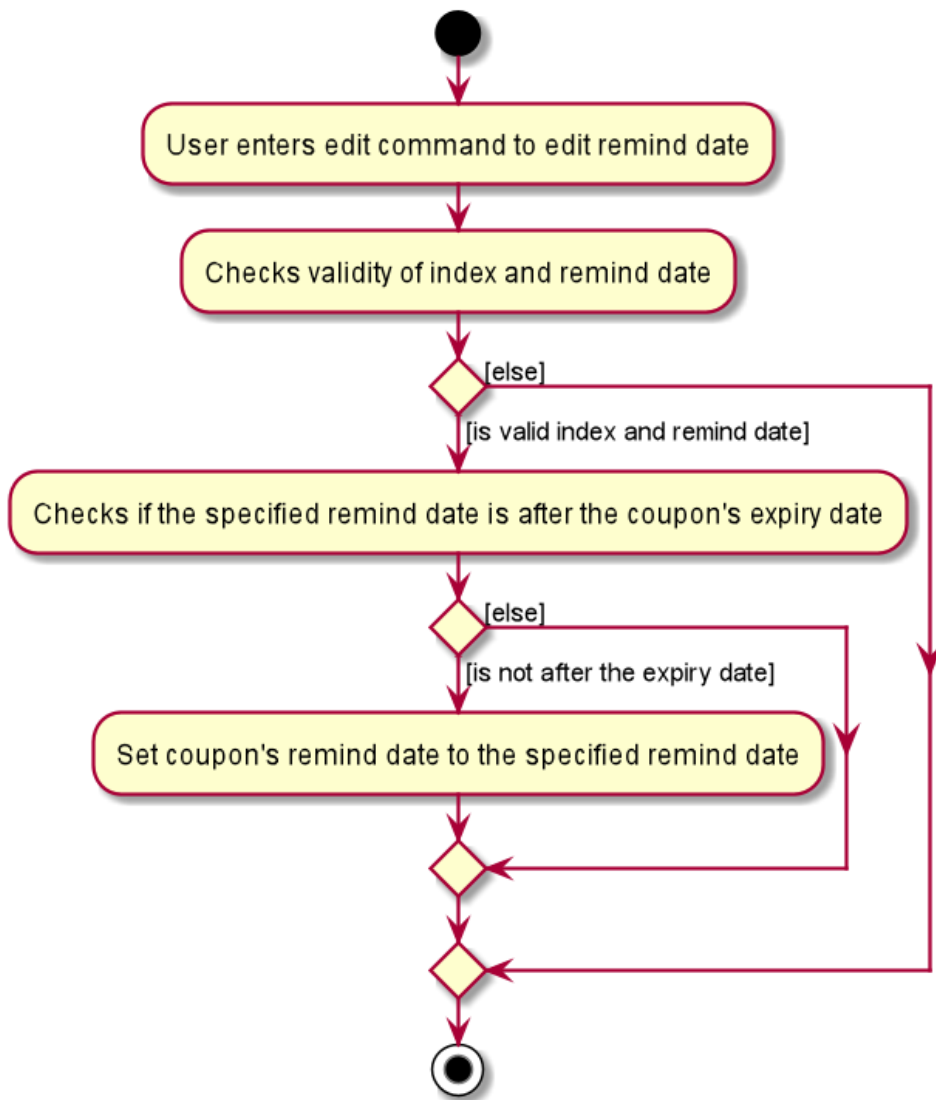


Figure 36. Activity Diagram representation of the flow of editing the remind date of a coupon.

### 3.6.2. Implementation of reminder pop up

After establishing the remind dates for all the coupons, the next step is ensure that there will be a reminder pop up (if necessary) upon opening the application.

The following steps describe how to reminder pop up works:

1. The user launches Coupon Stash. The `start` method in `MainApp` class will kick start the program by setting up the stage, along with the saved data.
2. This will trigger the `start` method in `UiManager`, which leads to the creation of a new `RemindWindow` instance, with a `List` of all coupons currently stored passed in as a parameter.
3. In the constructor of `RemindWindow`, coupons that do not have a remind date of today are filtered out.
4. After filtering the coupons, if there are coupons to be reminded today, their information will be concatenated into a `String` that is displayed in the reminder window.

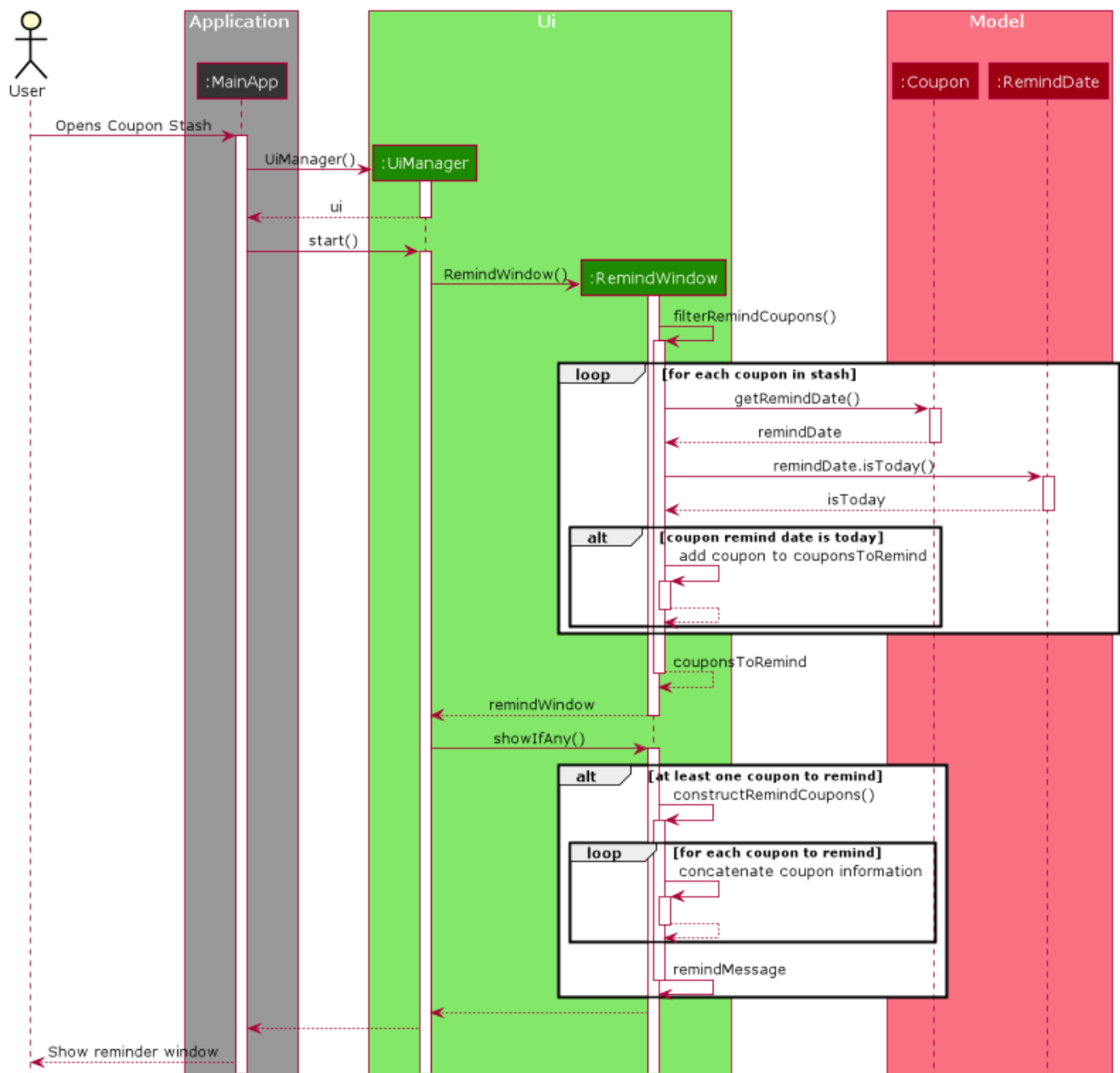


Figure 37. Sequence diagram describing the process of opening the reminder window.

### 3.6.3. Design consideration

#### Aspect: How to keep track of coupon remind date

- **Alternative 1 (current choice):** Coupon contains a **Remind** field.
  - Pros: Code implementation is easier and this makes the remind date more visible to the user since it is a field.
  - Cons: Coupon display may get very cluttered with the addition of this extra field.
- **Alternative 2 :** Store the remind dates of all coupons in a separate data file. Coupons can be stored in a hash table with their remind dates as keys.
  - Pros: No need to clutter coupon display with additional fields. Plus, it is efficient to list all coupons that have to be reminded for on a certain day as the coupons are stored in a hash table.

- Cons: Hard to maintain two separate data files that have shared components (in this case coupons)

All in all, we chose alternative 1 as we feel that it is good for users to be able to view the remind date of a coupon in the coupon view. Additionally, all speed-ups and efficiency of storing the remind dates in a separate data file is nullified by the fact that we still need to loop through all coupons to display their remind dates on the calendar component. Thus, to make it easier to extend the program in the future, we decided against adding another data file which can make extension more complicated, and chose to work with alternative 1.

## 3.7. Savings per use and total amount saved

To allow users to keep track of how much they have saved (after all, the whole point of coupons is to offer certain tangible benefits, encouraging purchases by customers), Coupon Stash automatically tracks the user's savings as they use their coupons that are handled in the application.

To achieve this, **Coupons** have to store two different fields:

1. Amount of savings each use of a coupon provides
2. Total amount of savings accumulated from using a certain coupon

### 3.7.1. Class structure of Savings

Just for reference, the image below shows the class diagram for the **Savings** class. It is compulsory for each **Coupon** to contain an **Savings** object, that represents what the user would gain from 1 use of that **Coupon**.

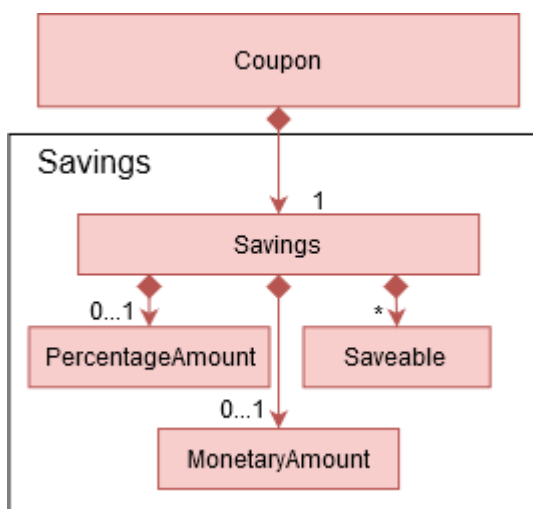


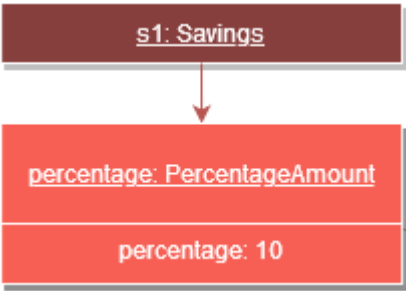
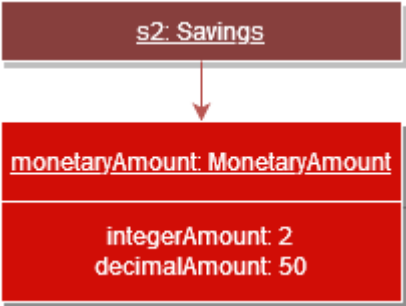
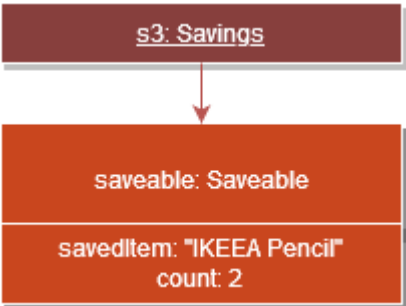

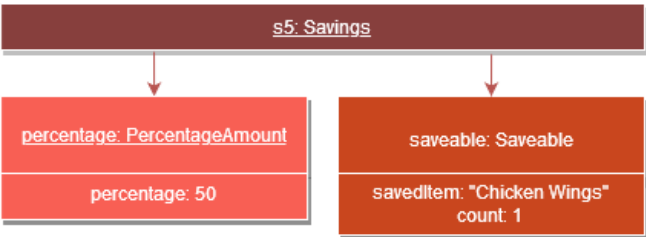
Figure 38. Class diagram describing the structure of **Savings**.

A **Savings** object can hold a **PercentageAmount**, **MonetaryAmount** or **Saveables**, which represents discounts like "\$5 off", "10% off" and "free door gift" respectively.

The table below shows which are valid **Savings** objects, and which are not.

Table 1. A list of different **Savings** objects, which could be valid or invalid.

Object Diagram	Comments
----------------	----------

<p><i>Savings with percentage amount.</i></p>  <pre> graph TD     s1["s1: Savings"] --&gt; p1["percentage: PercentageAmount"]     p1 --&gt; v1["percentage: 10"] </pre>	Valid
<p><i>Savings with monetary amount.</i></p>  <pre> graph TD     s2["s2: Savings"] --&gt; m1["monetaryAmount: MonetaryAmount"]     m1 --&gt; v1["integerAmount: 2&lt;br/&gt;decimalAmount: 50"] </pre>	Valid
<p><i>Savings with saveable items.</i></p>  <pre> graph TD     s3["s3: Savings"] --&gt; s1["saveable: Saveable"]     s1 --&gt; v1["savedItem: 'IKEEA Pencil'&lt;br/&gt;count: 2"] </pre>	Valid
<p><i>Savings without any fields present.</i></p>  <pre> graph TD     s4["s4: Savings"] </pre>	Invalid: <b>Savings</b> must have at least one field
<p><i>Savings with percentage amount and a saveable item.</i></p>  <pre> graph TD     s5["s5: Savings"] --&gt; p1["percentage: PercentageAmount"]     s5 --&gt; s1["saveable: Saveable"]     p1 --&gt; v1["percentage: 50"]     s1 --&gt; v2["savedItem: 'Chicken Wings'&lt;br/&gt;count: 1"] </pre>	Valid: <b>Savings</b> can have both a percentage amount and saveables

<p><i>Savings with monetary amount and multiple saveable items.</i></p>	<p>Valid: <b>Savings</b> can hold more than one <b>Saveable</b></p>
<p><i>Savings with monetary amount and percentage amount.</i></p>	<p>Invalid: <b>Savings</b> cannot have both a <b>MonetaryAmount</b> and <b>PercentageAmount</b></p>

As can be seen from the table, **Savings** cannot be completely empty, and **Savings** cannot have both a **MonetaryAmount** and **PercentageAmount** (it does not make much sense to have a voucher that says "10% and \$5 off").

### 3.7.2. **PureMonetarySavings** and **DateSavingsSumMap**

In order to calculate the total amount saved, **Coupons** also store information about how much the user saves, and storage is done at the moment the user uses the coupon. This information is stored in the form of **PureMonetarySavings**, which is a subclass of **Savings** that never holds **PercentageAmounts**. The class diagram below illustrates this.

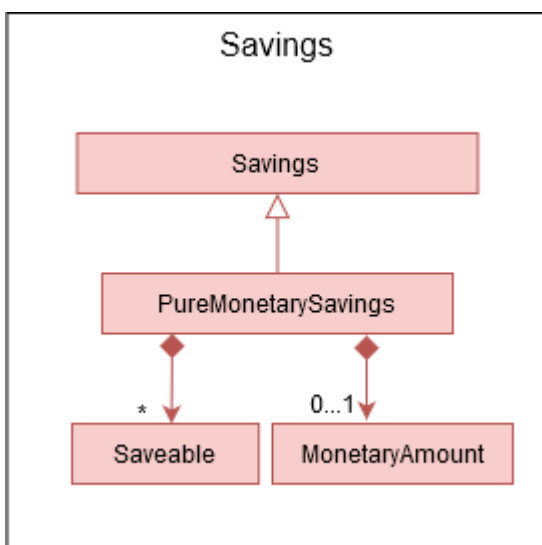


Figure 39. Class diagram describing the structure of **PureMonetarySavings**.

The reason why **PercentageAmounts** are not allowed in accumulated savings is because a percentage discount is a relative value that depends on the original price of the product, and cannot be easily added up in a way that allows users to accurately measure how much they have saved from their coupons.

**PureMonetarySavings** are stored in a **DateSavingsSumMap**, which is a hash table that links the current date (**LocalDate**) to the savings earned (**PureMonetarySavings**) on that date. Each **Coupon** holds a **DateSavingsSumMap**. The next image shows the class diagram of the **DateSavingsSumMap**.

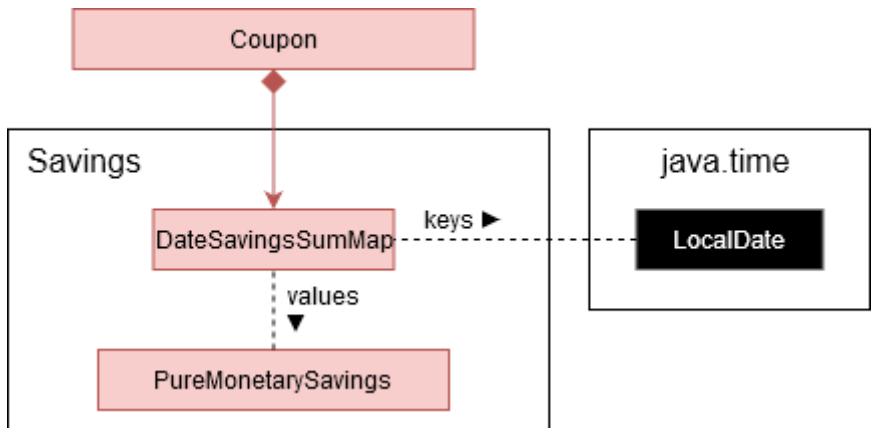


Figure 40. Class diagram describing the structure of **DateSavingsSumMap**.

The following section describes the processes that follow whenever a user marks a Coupon as "used" with the **used** command.

### 3.7.3. Implementation of used command

When the user enters a **used** command, the actions taken by Coupon Stash change depending on whether the Coupon's Savings stores a MonetaryAmount of PercentageAmount. The following activity diagram shows what happens when the user runs a used command.

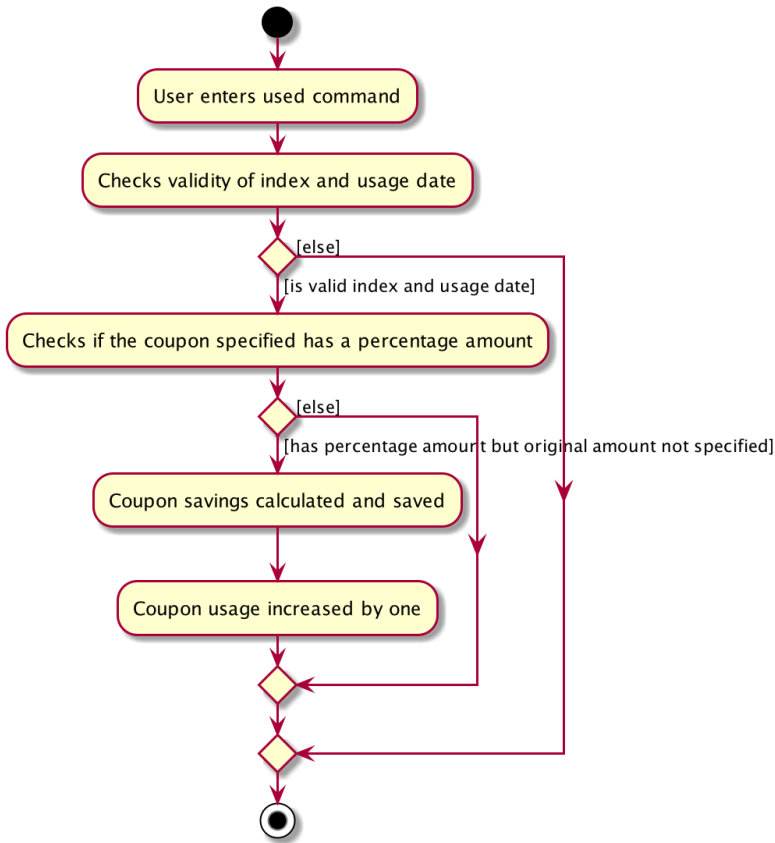


Figure 41. Activity diagram showing the execution of a used command.

In terms of the implementation, the next two images shows the sequence diagram that models the successful execution of a used command within the actual program components.

More specifically, the used command executed is **used 1 \$100**, and the state of the system is such that a **Coupon** with **PercentageAmount** in its **Savings** (no **MonetaryAmount**) and with **Usage** not at its **Limit** is located at index 1. Also, the money symbol set in the user preferences would be **\$**, which makes this command a valid one that will execute successfully.

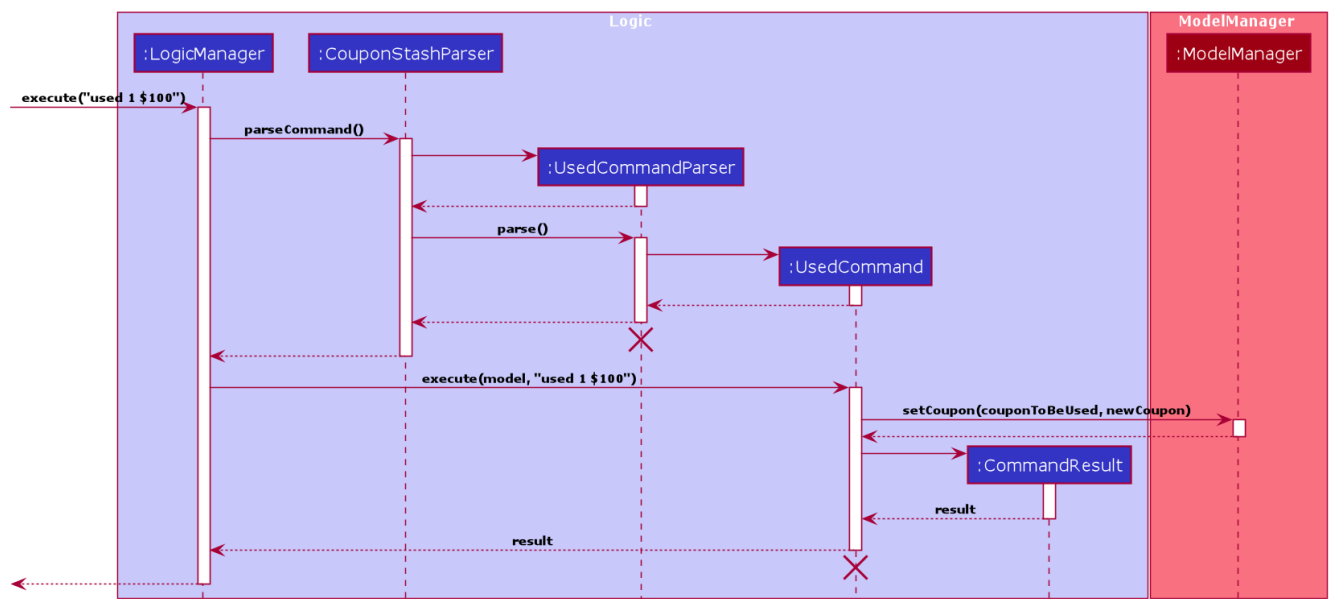


Figure 42. Sequence diagram showing how a **UsedCommand** is executed.

The money symbol set in the user preferences is retrieved by **CouponStashParser**, which passes it to

`UsedCommandParser` that will use this symbol to parse the `used` command.

Also, within `UsedCommand`, the `UsedCommand#execute()` method will cause the creation of a new `Coupon` with the correct recorded number of uses and amount of savings earned. The next sequence diagram shows how a successful `UsedCommand#execute()` method produces the new total savings value for the new `Coupon`.

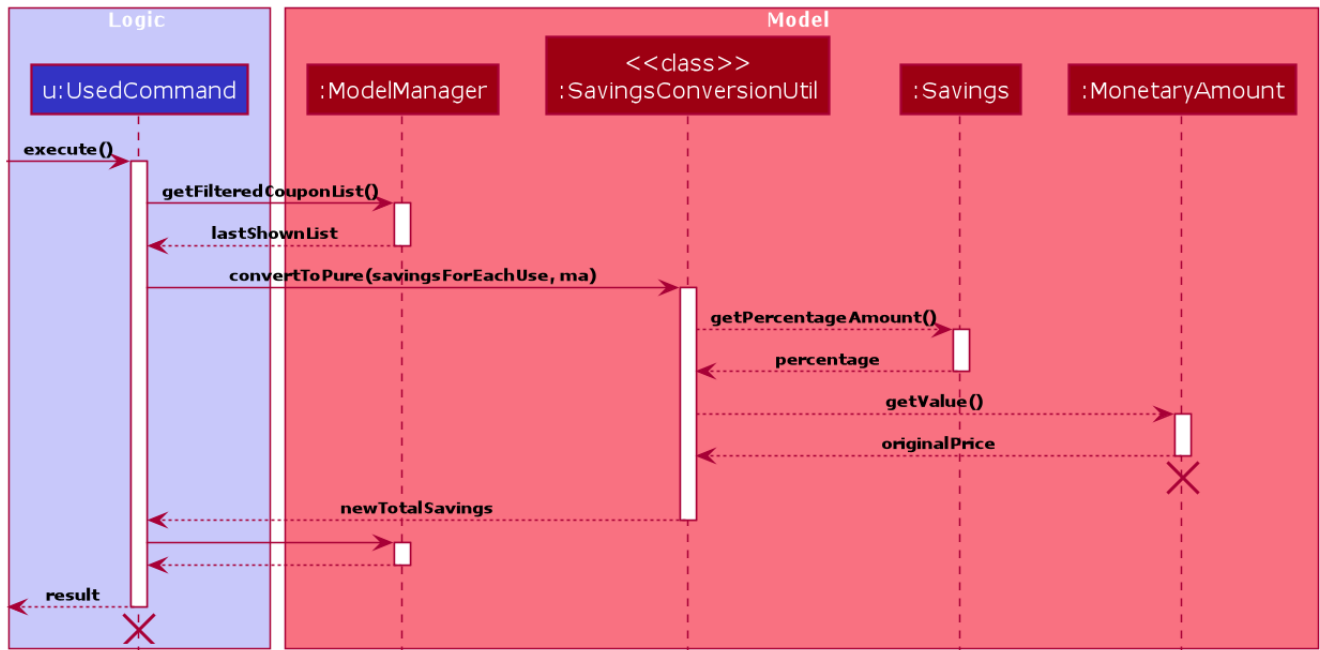


Figure 43. Sequence diagram showing how `UsedCommand` updates the total savings.

In the end, the total savings value of the `Coupon` is updated. This total savings is represented by a `DateSavingsSumMap`.

One key implementation within the `UsedCommand` is the checks that it has to make to ensure the valid usage of a `Coupon`. Below is an activity diagram to show the flow of checks within the `UsedCommand#execute()` method.

### 3.7.4. Implementation of saved command

Now that we have seen how the `used` command works, we can look at how the `saved` command works. While `used` stores the amount of savings that the user has earned on a particular day, `saved` retrieves the amount of savings earned as recorded by Coupon Stash, given a particular time period.

The saved command works similarly to the used command, where a `SavedCommandParser` will be created by `Logic` to split up the raw `String` into its arguments, creating a `SavedCommand`. Let's look at how a `SavedCommand` would be executed.



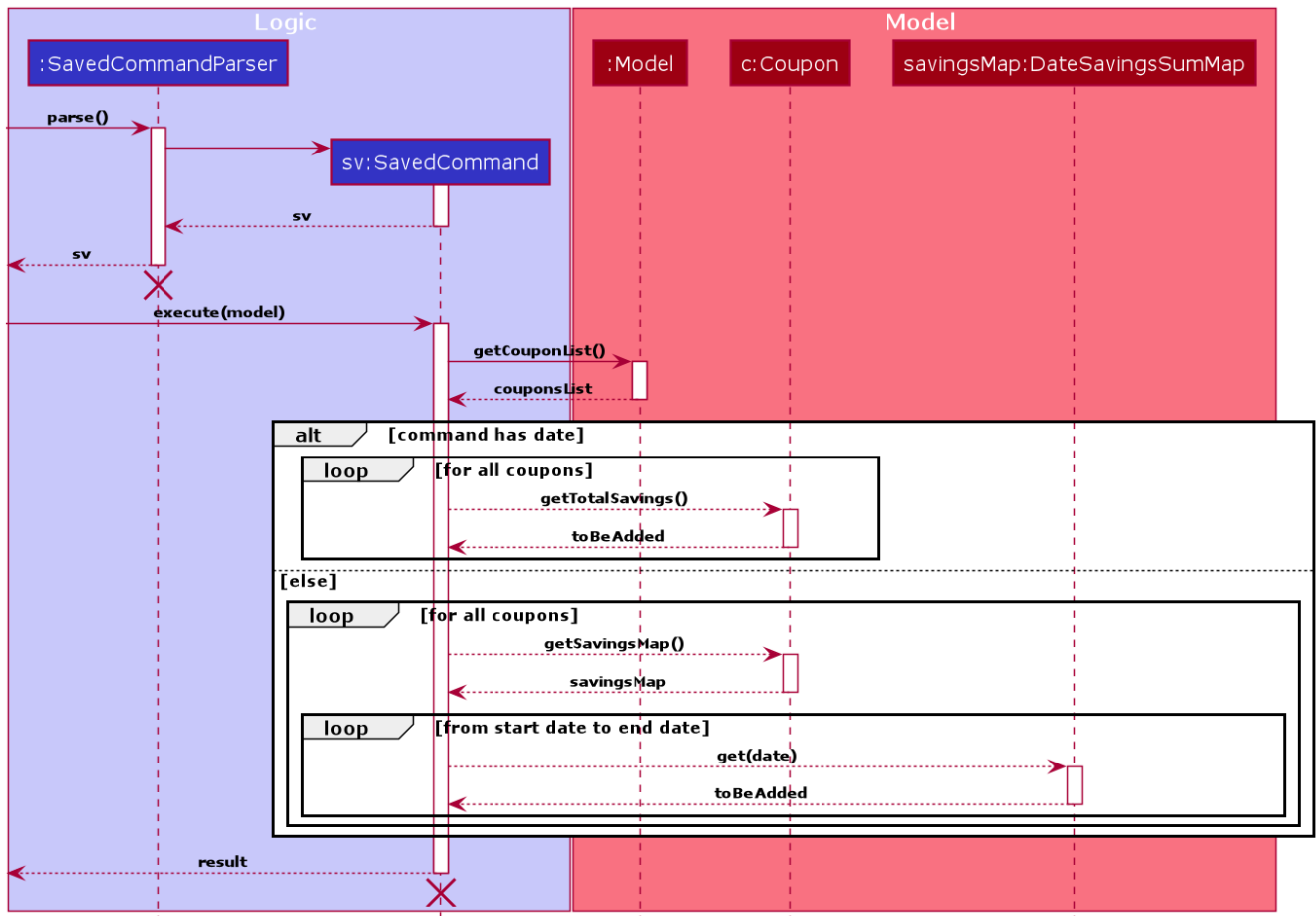


Figure 44. Sequence diagram showing the execution of a `SavedCommand`.

Hence the `SavedCommand` loops through all `Coupons` to add up the savings earned from a particular time period, or from all dates if no time period is specified.

### 3.7.5. Design considerations

Based on the User Stories, there is a desire for tracking how much one has saved by using Coupon Stash, as well as for viewing total savings easily. Below are some alternative implementations of savings tracking and viewing that were considered by the developers, but were rejected in favour of the current implementation.

Alternatives:

- Restrict each `Savings` to a concrete monetary value

This would make the implementation of `Savings` much simpler, as there would not be a need for separate classes like `PercentageAmount`, `MonetaryAmount` and `Saveables`. However, this might burden the user with calculating how much they would save in terms of dollars and cents, when many coupons and discounts come in the form of certain percentage reductions of the original price, as well as free gifts or benefits that cannot be translatable to a concrete monetary amount.

Hence, it was decided to rely on a few different representations of `Savings` that can be earned from using a `Coupon`, as well as a `Savings` class that could refer to any of these representations, or even a logical combination of these representations.

- Each `Coupon` stores a `MonetaryAmount`, `PercentageAmount` and `Saveables` directly

This would eliminate the need for the intermediary `Savings` class and reduce complication in the program code slightly. But, it would be difficult to ensure that at least one such field exists in the `Coupon`, or guarantee that the `Coupon` would have one such field.

The `Coupon` class would have to hold the logic for determining whether it had a valid combination of `MonetaryAmount`, `PercentageAmount` and `Saveables`, which does violate Single Responsibility Principle as the `Coupon` class now has another reason to change (if we would want to allow both `MonetaryAmount` and `PercentageAmount` on a `Coupon` for instance).

Hence the `Savings` class was decided to handle this responsibility, as well as abstract away the implementation details of the multiple possible values and combinations of these values. This allows the `Coupon` to think in terms of an entire `Savings` object, rather than handle multiple different scenarios depending on which fields it has.

## 3.8. Logging

We are using the `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See [Section 3.9, “Configuration”](#))
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

### Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

## 3.9. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

# 4. Documentation

Refer to the guide [here](#).

# 5. Testing

Refer to the guide [here](#).

## 6. Dev Ops

Refer to the guide [here](#).

## Appendix A: Product Scope

- Bargain hunter that has accumulated many coupons
- Likes to use desktop applications
- Would rather type a command than click a button
- Fast typist
- Enjoys using command-line interface

**Value proposition:** Manage coupons faster than a typical mouse/GUI driven app

## Appendix B: User Stories

Priorities:

\* \* \* \* - epic (must have) | | \* \* \* - rare (nice to have) | | \* \* - comon (unlikely to have) | | \* - rabak (will negatively affect the application)

Table 2. User stories and their priorities

Priority	As a ...	I want to ...	so that I can ...
* * * *	forgetful student	keep track of all the <i>promo codes</i> /coupons	redeem it at their respective stores.
* * * *	SoC student	quickly input the coupons that I collected from welfare packs	have a digital record of all the coupon in a safe place
* * * *	user	get a list of all the vouchers/ <i>promo codes</i> that are expiring soon	make use of them before they expire

Priority	As a ...	I want to ...	so that I can ...
* * * *	user	track how many times I can use the <i>promo codes/coupons</i>	use them multiple times if possible
* * * *	user	track how much I have saved from using these <i>promo codes/coupons</i>	know how much I save within a period.
* * * *	user	have an overview of when my coupons are expiring	use them before they expire.
* * * *	thrifty student with student loan	apply discount codes/coupons	maximise my savings
* * * *	highly competent SoC student	execute simple tasks like add, sorting and finding a coupon	showcase how easy it is to use command-line
* * * *	organized student	have a easy visualisation representation of all my coupons	can efficiently update any coupons' details
* * * *	store owner	able to search for coupons by store	customers do not waste too much time finding their coupons
* * * *	conscientious couponer	want to be reminded of the soon-to-be expire coupon	use it before it expires

Priority	As a ...	I want to ...	so that I can ...
* * *	command-line enthusiast	make use of my fast typing speed to organise my coupons in seconds	spend the rest of my time drinking over a lack of friends
* * *	canteen stall owner	promote my store by giving out coupons and vouchers	students can benefit from my amazing culinary skills
* * *	business owner	let potential consumers discover my discount codes/coupons	advertise and market my products/services
* * *	user	track how much I have spent from using these <i>promo codes</i> /coupons	plan my expenses for the month
* * *	financial-aid SoC student	quickly store the <i>promo code</i> shared by my peers and use them later for critical necessity like KBBQ and escape room	maximise my savings
* * *	exchange student attached to SoC	keep track of the good deals in Singapore	explore Singapore on a tight budget
* * *	time-conscious student	use command line to access my coupons	spend more time with my family

Priority	As a ...	I want to ...	so that I can ...
***	lazy student	input coupon details with ease	life is worth living
***	influencer	keep track of all my client's coupon code	share the codes at my IG
***	digital nomad	access all the coupons while I am on the go	reduce my spending
**	consumer	check if the store has any ongoing discount/promotions before making payment	save some money from it
**	bargain hunter	know which coupon requires group purchase	quickly share it to my peers
**	SoC lecturer	share my wealth of coupons with students	students will think I am cool and hip instead of another boring lecturer
**	exchange student	find the best food and attractions in Singapore easily	make good use of my time here
**	block head	share relevant club's coupons to all my hall members	get more financial support from respective sponsors

Priority	As a ...	I want to ...	so that I can ...
**	mobile phone user	email the coupon details to myself	easily access them when I'm outside
*	SoC cleaner	make use of the rubbish that students always leave behind after orientation camps	make use of necessary services like Korean BBQ and escape rooms
*	mother of 5 SoC students	look out for the hottest deals in town	finance my childrens' education
*	computing student	save data such that it is easily parsable	create alternative clients
*	infosec student	encrypt all coupons in one place	prevent hackers to hack my coupons

## Appendix C: Use Cases

This is a list of Use-Cases for Coupon Stash, a coupon stash application. Primary actor is the user.

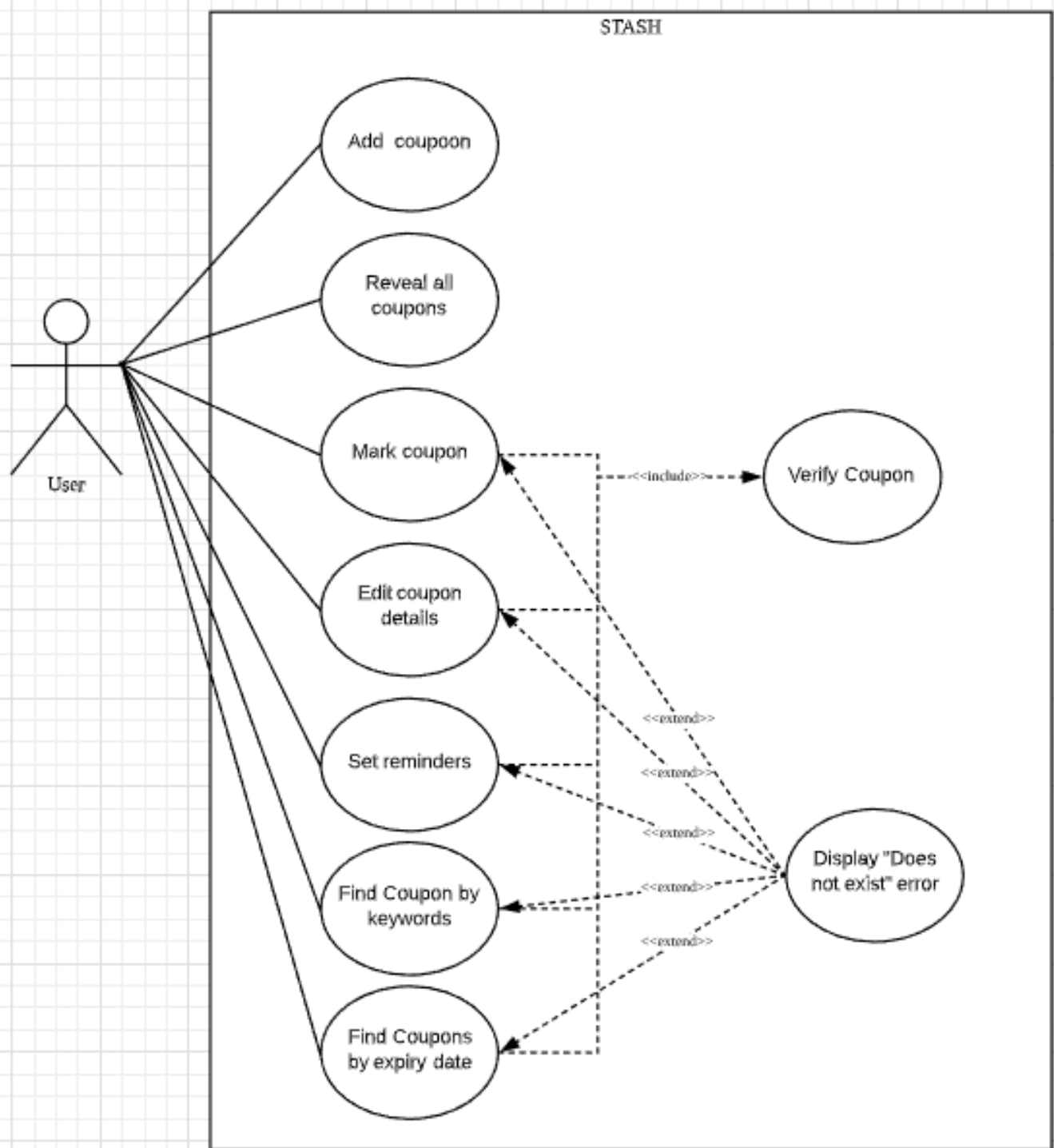


Figure 45. Use Cases Overview for Coupon Stash

## C.1. Use Case: UC1 - Add Coupon

**Actor:** user

**Precondition:** User has opened the application

This use case describes how a user uses Coupon Stash to add a new coupon entry.

### MSS

1. User keys in command to add coupon.
2. Coupon Stash adds coupon.



3. Coupon Stash informs user that a coupon is added.

Use case ends.

### Extensions

1a. Coupon Stash detects an invalid format in the entered data.

1a1. Coupon Stash requests the user to re-enter the details.

1a2. User enters new data.

Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

1a3. User enters wrong data twice.

1a4. Coupon Stash clears command line.

Use case ends.

## C.2. Use Case: UC2 - List all coupons

**Actor:** user

**Precondition:** User has opened the application

This use case describes how a user uses Coupon Stash to list out all the coupon entries.

### MSS

1. User keys in command to list all the coupons.

2. Coupon Stash lists out all coupons.

3. Coupon Stash informs to user of the number of coupons found in the list.

Use case ends.

### Extensions

1a. Coupon Stash detects an invalid format in the entered data.

1a1. Coupon Stash requests the user to re-enter the details.

1a2. User enters new data.

Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

1a3. User enters wrong data twice.

1a4. Coupon Stash clears command line.

Use case ends.

1b. Coupon Stash detects that the coupon list is empty.

1b1. Coupon Stash informs the user that the list is empty.

Use case ends

### C.3. Use Case: UC3 - Mark a coupon as used

**Actor:** user

**Pre-condition:** User has opened the application

#### MSS

1. User keys in command to <u>list all coupons (UC2)</u>.
2. User marks coupon as used.
3. Coupon Stash marks the coupon as used.
4. Coupon Stash informs the user the specific coupon that is successfully used.

Use case ends.

#### Extensions

1a. Coupon Stash detects an invalid format in the entered data.

1a1. Coupon Stash requests the user to re-enter the details with the correct format.

1a2. User enters new data.

Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

1b. Coupon Stash detects that the specified coupon does not exist.

1b1. Coupon Stash requests the user to enter an index that corresponds with an existing coupon.

1b2. User enters a new index.

Use case resumes from step 2.

1c. Coupon Stash detects that the specified coupon has been previously marked as done.

1c1. Coupon Stash informs user that the coupon has been previously marked as done.

Use case ends.

### C.4. Use Case: UC4 - Find coupon(s) by keyword(s)

**Actor:** user

**Pre-condition:** User has opened the application

This use case describes how a user uses Coupon Stash to find the coupon(s) with keyword(s).

#### MSS

1. User keys in command to find a coupon based on keyword(s).
2. Matched coupons are displayed.
3. Coupon Stash informs user the number of coupons found.

Use case ends.

### Extensions

1a. Coupon Stash detects an invalid format in the entered data.

1a1. Coupon Stash requests the user to re-enter the details with the correct format.

1a2. User enters new data.

Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

1b. Coupon Stash detects that the specified coupon does not exist.

1b1. Coupon Stash requests the user to enter an index that corresponds with an existing coupon.

1b2. User enters a new index.

Use case resumes from step 2.

## C.5. Use Case: UC5 - Edit coupon's details

**Actor:** user

**Precondition:** User has opened the application

This use case describes how a user uses Coupon Stash to edit details of an existing coupon.

### MSS

1. User keys in command to <u>list all coupons (UC2)</u>.
2. User edits an existing coupon.
3. Coupon Stash updates the coupon details.
4. Coupon Stash informs the user which coupon has been edited.

Use case ends.

### Extensions

2a. Coupon Stash detects an invalid format in the entered data.

2a1. Coupon Stash requests the user to re-enter the details with the correct format.

2a2. User enters new data.

Steps 2a1 - 2a2 are repeated twice until the data entered are correct.

Use case resumes from step 3.

2b. Coupon Stash detects that the specified coupon does not exist.

2b1. Coupon Stash requests the user to enter an index that corresponds with an existing coupon.

2b2. User enters a new index.

Use case resumes from step 3.

## C.6. Use Case: UC6 - Set reminder

**Actor:** user

**Precondition:** User has opened the application

This use case describes how a user uses Coupon Stash to set reminders for an existing coupon.

### MSS

1. User keys in command to <u>list all coupons (UC2)</u>.
2. User sets a reminder for an existing coupon.
3. On the day of the input date, a pop up will appear to remind the user about the coupon.

Use case ends.

### Extensions

2a. Coupon Stash detects an invalid format in the entered data.

2a1. Coupon Stash requests the user to re-enter the details with the correct format.

2a2. User enters new data.

Steps 2a1 - 2a2 are repeated twice until the data entered are correct.

Use case resumes from step 3.

2b. Coupon Stash detects that the specified coupon does not exist.

2b1. Coupon Stash requests the user to enter an index that corresponds with an existing coupon.

2b2. User enters a new index.

Use case resumes from step 3.

## C.7. Use Case: UC7 - List coupon(s) expiring before date

**Actor:** user

**Precondition:** User has opened the application

This use case describes how a user uses Coupon Stash to find the coupon(s) expiring before the input expiry date.

### MSS

1. User keys in command to find a coupon based on expiry date.
2. Matched coupons are displayed. Coupon Stash informs the user the number of coupons expiring before the specified date. Use case ends.

### Extensions

1a. Coupon Stash detects an invalid format in the entered data.

1a1. Coupon Stash requests the user to re-enter the details with the correct format.

1a2. User enters new data.

Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

1b. Coupon Stash detects that the specified coupon does not exist.

1b1. Coupon Stash requests the user to enter an index that corresponds with an existing coupon.

1b2. User enters a new index.

Use case resumes from step 2.

## C.8. Use Case: UC8 - Delete coupon

**Actor:** user

This use case describes how a user uses Coupon Stash to delete an existing coupon.

### MSS

1. User <u>list all coupons (UC2)</u>.
2. User deletes an existing coupon.
3. User confirms its decision during confirmation.
4. Coupon Stash removes the coupon.

Use case ends.

### Extensions

2a. Coupon Stash detects an invalid format in the entered data.

2a1. Coupon Stash requests the user to re-enter the details with the correct format.

2a2. User enters new data.

Steps 2a1 - 2a2 are repeated twice until the data entered are correct.

Use case resumes from step 2.

2b. Coupon Stash detects that the specified coupon does not exist.

2b1. Coupon Stash requests the user to enter an index that corresponds with an existing

coupon.

2b2. User enters a new index.

Use case resumes from step 2.

## C.9. Use Case: UC9 - Undo previous command

**Actor:** `user`

This use case describes how a user undo the previous command in Coupon Stash.

### MSS

1. User keys in command to undo a previous command.
2. User confirms its decision during confirmation.
3. Coupon Stash undo the previous command.
4. Coupon Stash informs the user which command has been undone.

Use case ends.

### Extensions

- 1a. Coupon Stash detects an invalid format in the entered data.
  - 1a1. Coupon Stash requests the user to re-enter the details with the correct format.
  - 1a2. User enters new data.Steps 1a1 - 1a2 are repeated twice until the data entered are correct.

Use case resumes from step 1.

## Appendix D: Non-Functional Requirements

1. Coupon Stash works on `common operating systems (OS)` that have `Java 11` or above installed.
2. Coupon Stash can store at least 500 coupons without crashing the application.
3. Coupon Stash can operate without noticeable lag (~2s) when entering commands or interacting with the UI.
4. Coupon Stash caters to users who have above average typing speed, and these users should be able to get tasks completed faster in the application by typing, rather than using the mouse and the UI.
5. Coupon Stash source code should be covered by tests as much as possible.
6. Coupon Stash should be easy to use for users, who are not familiar with coding.
7. All monetary amounts should be accurate up to 2 decimal places.
8. Coupon Stash should be portable.
9. Data files should remain unchanged when transferring from OS to OS.

10. Coupon Stash works perfectly without access to the internet.
11. Coupon Stash supports various types of coupons (e.g. promotional codes, QR code, or barcode) (coming in in v2.0)

## Appendix E: Glossary

- **Coupon Stash** - the program that makes handling your coupons easier, and also the subject matter of this Developer Guide.
- **common operating system** - refers to the most widely seen operating systems for desktop computers.
- **Java 11** - the 11th version of the highly popular Java platform and programming language, on which many software applications are built upon.
- **lag** - the phenomenon where some arbitrary user input takes a noticeable and vexatious amount of time to effect a change in the application state.
- **monetary amount** - any currency amount (for example, 10.55 may represent 10 dollars and 55 cents, or 10 pounds and 55 pence, or 10 pesos and 55 centavos).
- **operating system** - a fundamental software application that runs on a computer, supporting basic functions such as ability to manage computer memory, to allow users to use the device without concern for such technical details.
- **OS (Operating System)** - see operating system.
- **promo code** - short for promotional code, usually refer to an unique string of letters and numbers that can be entered in some mobile application to redeem certain benefits.
- **SoC (School of Computing)** - the School of Computing at the National University of Singapore.
- **source code** - a set of instructions, written in a programming language that determine the final application's internal and external behaviour.
- **UI (User Interface)** - a catch-all term referring to how a computer system and a coupon interacts, usually referring to specific elements displayed on the computer screen that the user may interact with such as buttons or text boxes, as well as areas where the computer application displays certain outputs to the user.

## Appendix F: Instructions for Manual Testing

Given below are instructions to test the app manually.

### NOTE

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

### F.1. Launch and Shutdown

- Initial launch
  - a. Download the jar file and copy into an empty folder

- b. Head over to your local Command Line Interface (CLI), and change to the directory where the jar file was saved.
- c. Type `java - jar CouponStash.jar` in the CLI.  
Expected: Shows the GUI with a set of sample coupons.

## F.2. Adding a coupon

- Adding a coupon
  - a. Test case: `add n/Popular Bookstore e/31-12-2020 s/10%`  
Expected: Coupon Popular Bookstore is added to the list. `Start Date` is set to the `system's date`, while the Remind Date is set to 3 days before the specified `Expiry Date`. In this case, it would be 28-12-2020.

## F.3. Listing coupons

- List different type of coupons
  - a. Test case: `list`  
Expected: All active coupons are displayed on the list.
  - b. Test case: `list a/`  
Expected: All archived coupons are displayed on the list.
  - c. Test case: `list u/`  
Expected: All previously used coupons are displayed on the list.

## F.4. Deleting a coupon

- Deleting a coupon while all coupons are listed
  - a. Prerequisites: List all coupons using the `list` command. Have at least one coupon in the list.
  - b. Test case: `delete 1`  
Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
  - c. Test case: `delete 0`  
Expected: No coupon is deleted. Error details shown in the status message.
  - d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size)  
Expected: Similar to previous.

## F.5. Editing a Coupon

- Editing a coupon while all coupons are listed
  - a. Prerequisites: List all coupons using the `list` command. Have at least one coupon in the list.
  - b. Test case: `edit 1 p/ILOVESTASH`  
Expected: The first coupon will have its Promo Code changed to `ILOVESTASH`.
  - c. Test case: `edit 1 e/31-12-2021`



Expected: The first coupon will have its Expiry Date changed to **31-12-2021**.

- d. Test case: **edit 1 l/0**

Expected: The first coupon will now have unlimited usage.

- e. Test case: **edit 1 u/10**

Expected: An error message will be thrown, explaining that usage cannot be edited.

- f. Test case: **edit 1 s/\$10 s/Water Bottle**

Expected: The first coupon will have its savings changed to \$10, and include a Water Bottle item.

## F.6. Finding a coupon

- Finds certain coupons by name in Coupon Stash.
  - a. Prerequisites: Have some coupons in Coupon Stash, preferably the sample data.
  - b. Test case: **find adidas**  
Expected: Only the coupon named "Adidas" shows up.
  - c. Test case: **find grabfood**  
Expected: Only the coupon named "GrabFood" shows up.

## F.7. Sorting the Coupon Stash

- Sorts coupons by name, expiry date or remind date in Coupon Stash.
  - a. Prerequisites: Have some coupons in Coupon Stash, preferably the sample data. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: **sort n/**  
Expected: The Coupon Stash appears in this order: Adidas, Gong Cha, GrabFood, Lazada, LiHO, Shopee
  - c. Test case: **sort e/**  
Expected: The Coupon Stash appears in this order: GrabFood (30-04-2020), LiHO (31-05-2020), Gong Cha (30-06-2020), Lazada (30-09-2020), Shopee (31-10-2020), Adidas (31-12-2020)
  - d. Test case: **sort r/**  
Expected: The Coupon Stash appears in this order: GrabFood (27-04-2020), LiHO (28-05-2020), Gong Cha (27-06-2020), Lazada (27-09-2020), Shopee (28-10-2020), Adidas (28-12-2020)

## F.8. Listing all expiring coupons

- Shows coupons that are expiring on a certain date or a certain month.
  - a. Prerequisites: Have some coupons in Coupon Stash, preferably the sample data. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: **expiring e/31-10-2020**  
Expected: Shopee (expires on 31-10-2020) appears as the only coupon.
  - c. Test case: **expiring my/4-2020**  
Expected: GrabFood (expires 30-04-2020) appears as the only coupon

## F.9. Viewing savings

- Views savings accumulated in the application
  - a. Prerequisites: Have some coupons in Coupon Stash, preferably the sample data. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: `saved`  
Expected: The command result box displays "In total, you have saved \$60.00 as well as earned 1x USB-C cable, 1x iPhone case."
  - c. Test case: `saved d/11-3-2020`  
Expected: The command result box displays "You saved \$7.30 on 11 March 2020." (this savings record is attached to the "Gong Cha" coupon).
  - d. Test case: `saved sd/16-3-2020 e/23-3-2020`  
Expected: The command result box displays "You saved \$17.00 between 16 March 2020 and 23 March 2020."

## F.10. Using a coupon

- Marks a coupon as used and registers the savings. Depending on whether the coupon is a percentage or monetary amount, you may be required to enter in the original price of the item.
  - a. Prerequisites: Have some coupons in Coupon Stash that have usage not at the limit, preferably the sample data. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: `list, used 6 $100`  
Expected: The command result box displays "Used Coupon: GrabFood" and sayings for today increases by \$40 (40% of \$100 saved).

## F.11. Archiving a coupon

- Marks a coupon as archived and hides it from the normal view.
  - a. Prerequisites: Have some coupons in Coupon Stash that are not already archived. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: `list, archive 5`  
Expected: The command result box displays "Archived Coupon: LiHO" and LiHO disappears from the displayed coupons.
  - c. Test case: `list a/, unarchive 1`  
Expected: The command result box displays "Unarchived Coupon: LiHO" and LiHO reappears in the displayed coupons.

## F.12. Copying a coupon

- Recreates a coupon as an add command that can be copied and shared to other users of Coupon Stash.
  - a. Prerequisites: Have some coupons in Coupon Stash that are not already archived. The

following test cases assumes the commands are run on a set of unmodified sample data.

- b. Test case: `list, copy 1`

Expected: Command result box displays "Copied coupon: Adidas" and `add n/Adidas p/30ADIDAS e/31-12-2020 s/30.0% s/Adidas Cap 1/1` is added to the operating system's clipboard.

## F.13. Sharing a coupon

- Takes an image of the coupon as it appears in Coupon Stash and opens the file save dialogue box of your operating system so you can save the image to your computer.
  - a. Prerequisites: Have some coupons in Coupon Stash that are not already archived. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: `list, share 2`  
Expected: The operating system prompts you to save "Gong Cha.png" to a location on your computer.

## F.14. Undoing a command

- Undoes the previous command entered, only if the command resulted in a change to the coupons.
  - a. Prerequisites: At least one command that modified a coupon was previously entered.
  - b. Test case: `add n/Test e/1-1-5050 s/$40, undo` Expected: A new coupon named "Test" is added, and then removed again once `undo` is run.

## F.15. Going to a month on the calendar

- Navigates the calendar to jump to a certain month and year, without the need for clicking buttons in the User Interface.
  - a. Prerequisites: None
  - b. Test case: `goto my/4-5678`  
Expected: The calendar shows the month of April, in the year 5678.

## F.16. Expanding a coupon

- Opens a new window with full details of the Coupon, such as a list of all the saveable items.
  - a. Prerequisites: Have at least one coupon displayed in Coupon Stash. The following test cases assumes the commands are run on a set of unmodified sample data.
  - b. Test case: `list, expand 2`  
Expected: A window appears that shows the full history of coupon usage for the Gong Cha coupon.

## F.17. Setting currency symbol

- Changes the currency symbol used by Coupon Stash when displaying amounts saved and when interpreting monetary amounts in commands.
  - a. Prerequisites: None. The following test cases assumes the commands are run with the unmodified sample user preferences.
  - b. Test case: `setcurrency ms/RM`  
Expected: Command result box displays "Money symbol changed from \$ to RM!" and coupons shown are updated to display savings in RM instead of \$.

## F.18. Viewing help page

- Opens the user guide of Coupon Stash in another window.
  - a. Prerequisites: None.
  - b. Test case: `help`  
Expected: A web browser window opens, showing the Coupon Stash User Guide.

# Appendix G: Efforts

Coupon Stash is the first team-based software engineering project for all of us. Without much experience, we faced many difficulties in coordinating our features and codes. Since Coupon Stash is User Interface intensive, we had to learn JavaFX comprehensively to fulfil the high standards that we set for ourselves. Learning a new Framework was as good as learning a new language in itself, taking us some time to understand how to fully utilize it.

Each of us would push commits every other day every week to try and complete our tasking too. While our product is similar to AB3 in terms of data fields, we push it further by adding functionality to these fields. The intertwining of these functionality was the most difficult part about this project. We would usually try to sort out the implementation on chat, but there are times where one of us would mess up the implementation somewhere afterwards, causing a bit of debugging. We also try to apply as much of our learning from the textbook, but again without much practice, it would usually take us a few iterations and discussions to correct our code. Nonetheless, we learnt a lot from this project throughout these 7 weeks.

What we believe we did well in was getting our project workflow right from the start. Using Github's project board, issue tracker and branch protections, we were able to help each other spot mistakes, ensuring that our master branch was always working. Even though we started as strangers, the long weekly discussions that we have brought us close! We are definitely proud of the product that we created, and believe that it serves its purpose well.