Mortago v2.0 - Developer Guide

By: `Team AY1920S1-CS2103T-T13-2` Since: `Sep 2019` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

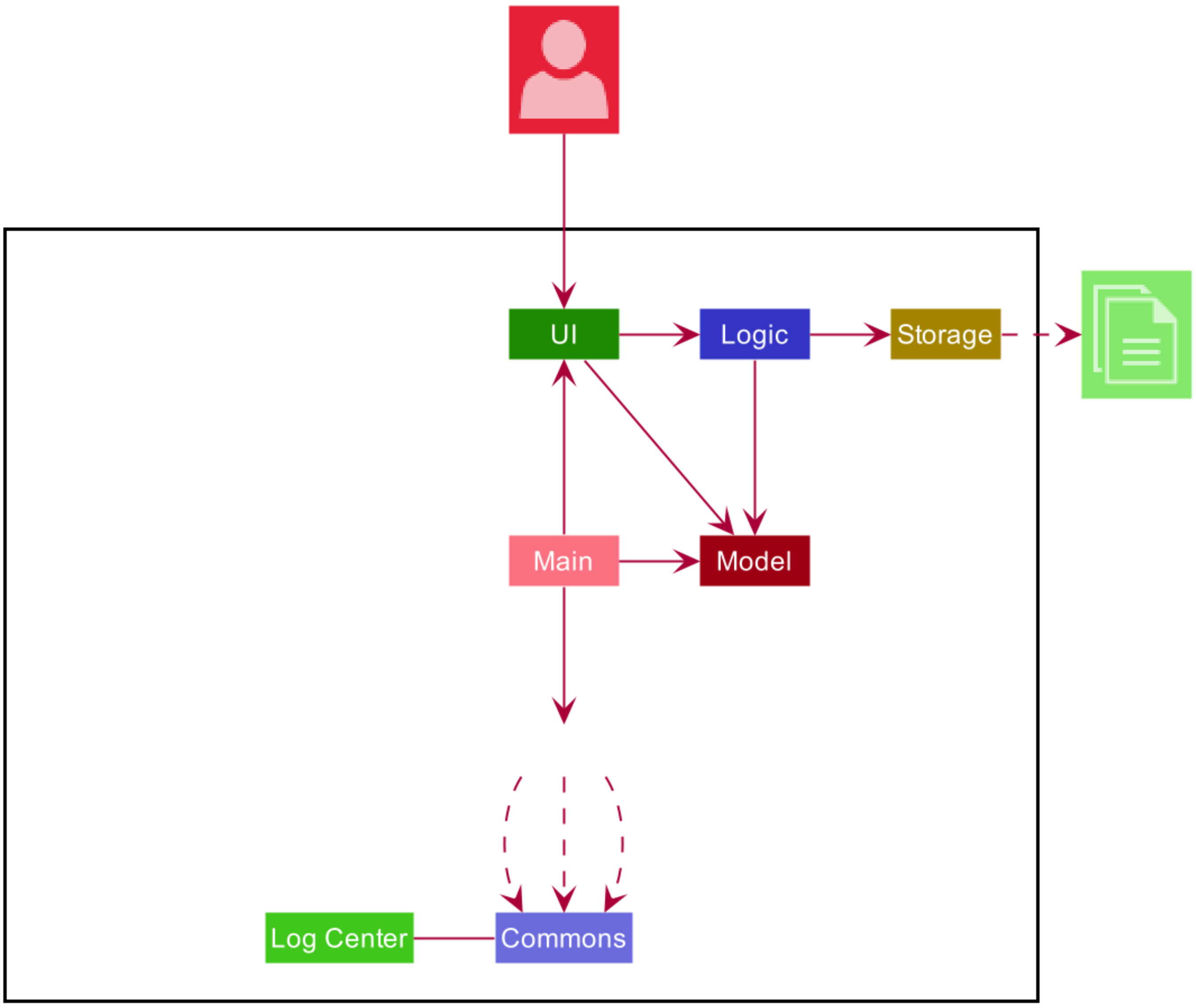


Figure 1. Architecture Diagram

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

The `.puml` files used to create diagrams in this document can be found in the diagrams (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/docs/diagrams/) folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`Main` has two classes called `Main` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/Main.java) and `MainApp` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/MainApp.java). 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.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

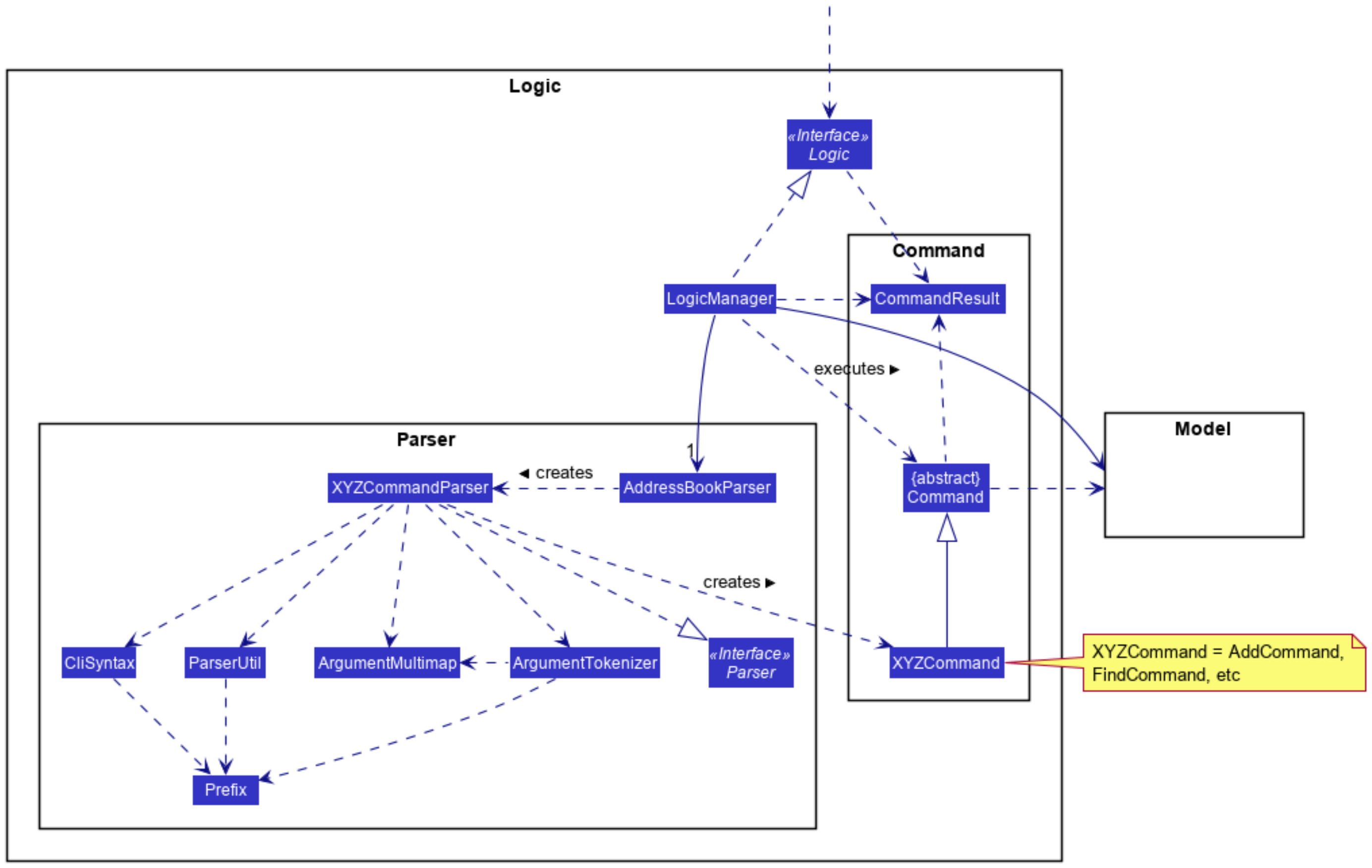


Figure 2. Class Diagram of the Logic Component

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 -b 1`.

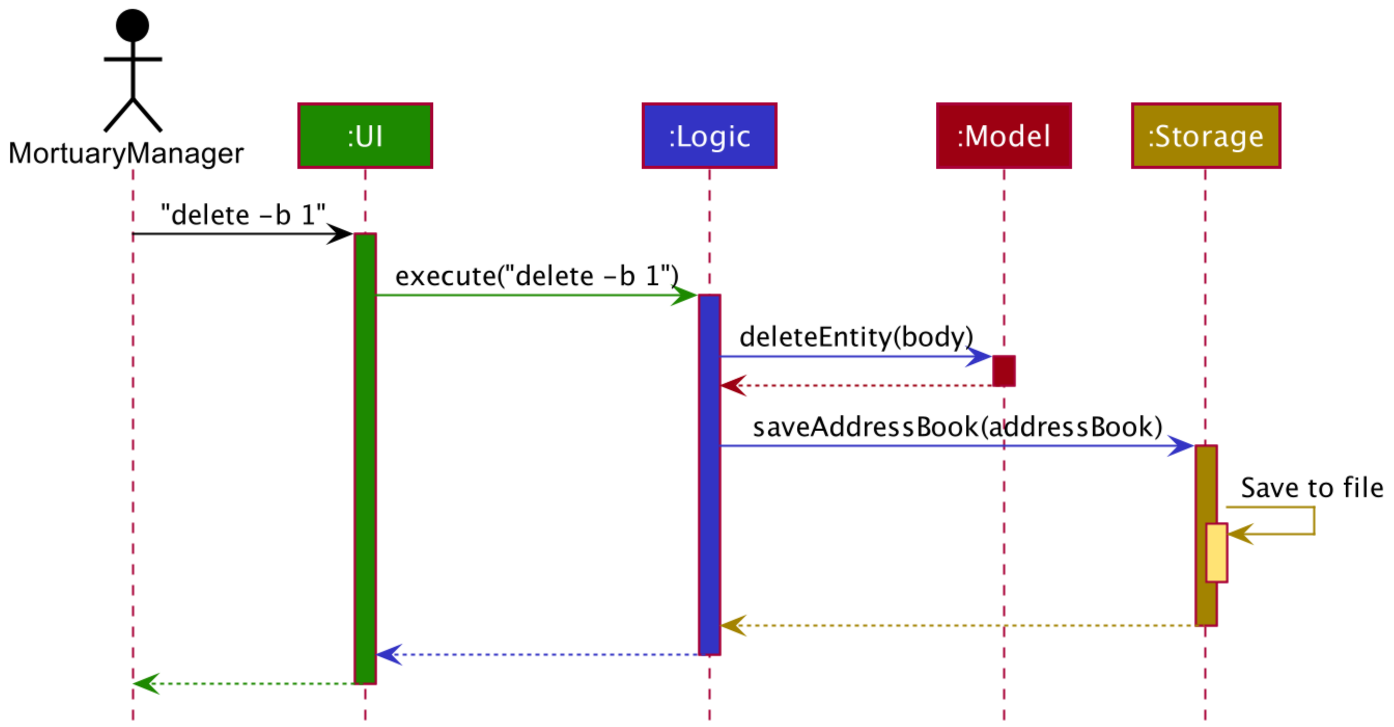


Figure 3. Component interactions for `delete -b 1` *command*

The sections below give more details of each component.

2.2. UI Component

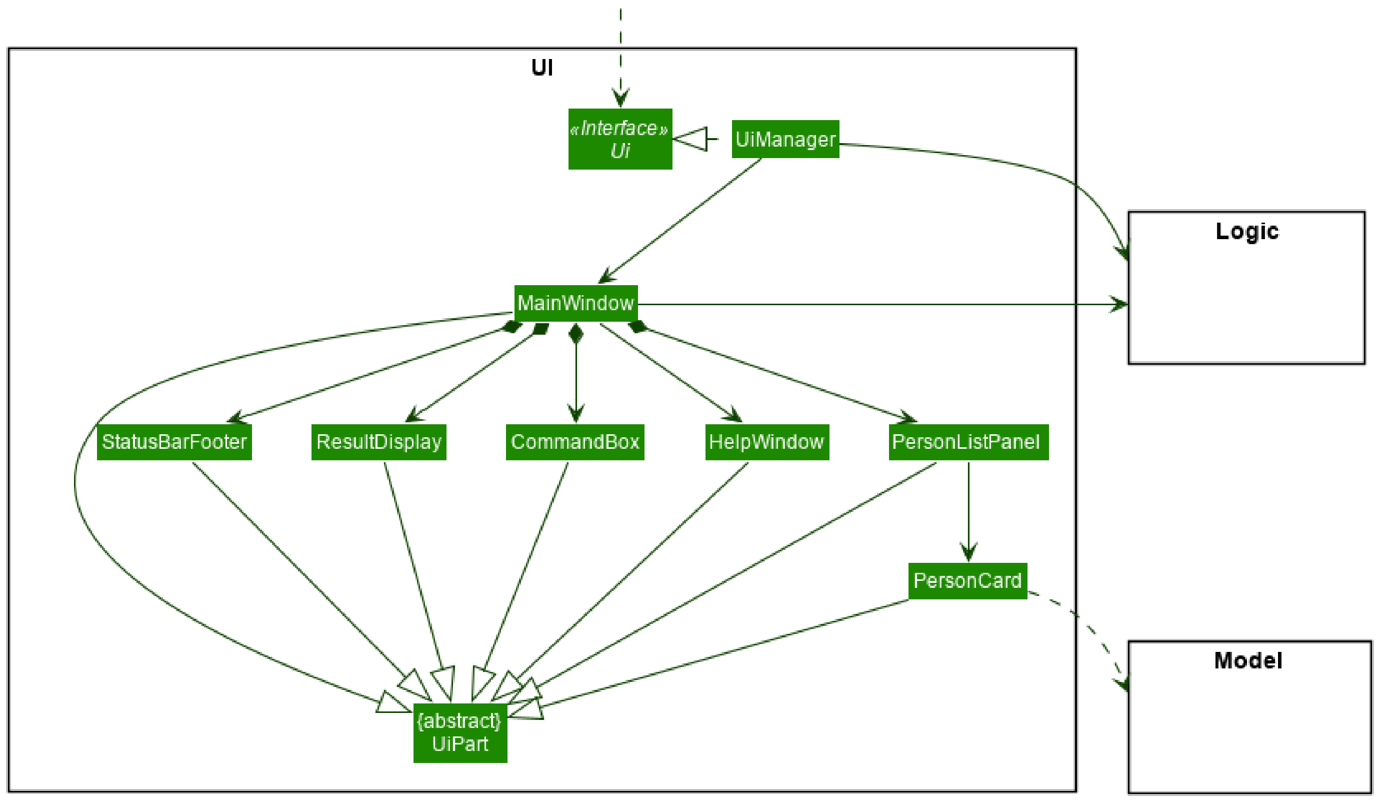


Figure 4. Structure of the UI Component

API : `Ui.java` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/ui/Ui.java)

The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox` , `ResultDisplay` , `PersonListPanel` , `StatusBarFooter` etc. All these, including the `MainWindow` , inherit from the abstract `UiPart` class.

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` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/ui/MainWindow.java) is specified in `MainWindow.fxml` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/resources/view/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

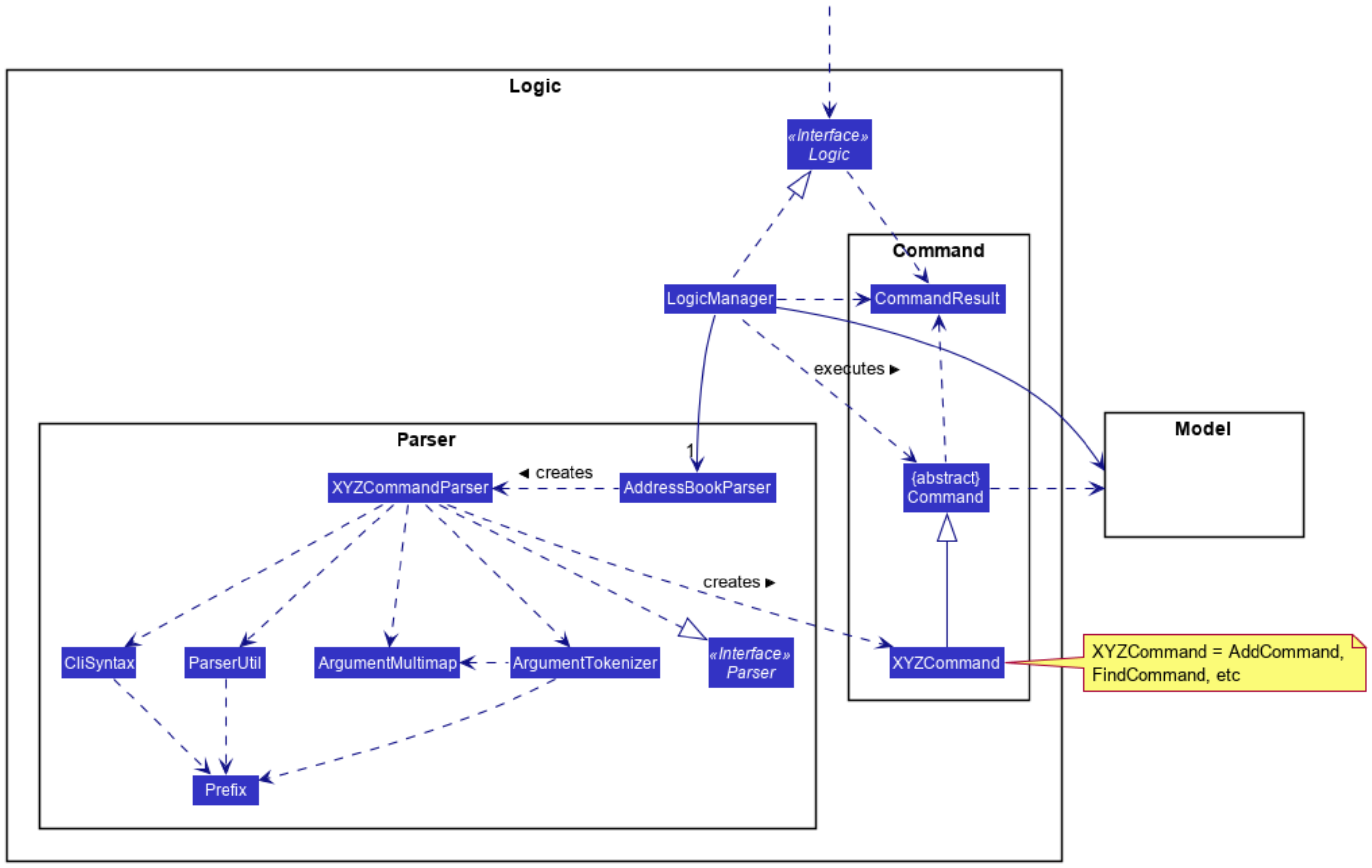


Figure 5. Structure of the Logic Component

API : `Logic.java`
(https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/logic/Logic.java)

1. `Logic` uses the `AddressBookParser` 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 a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

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

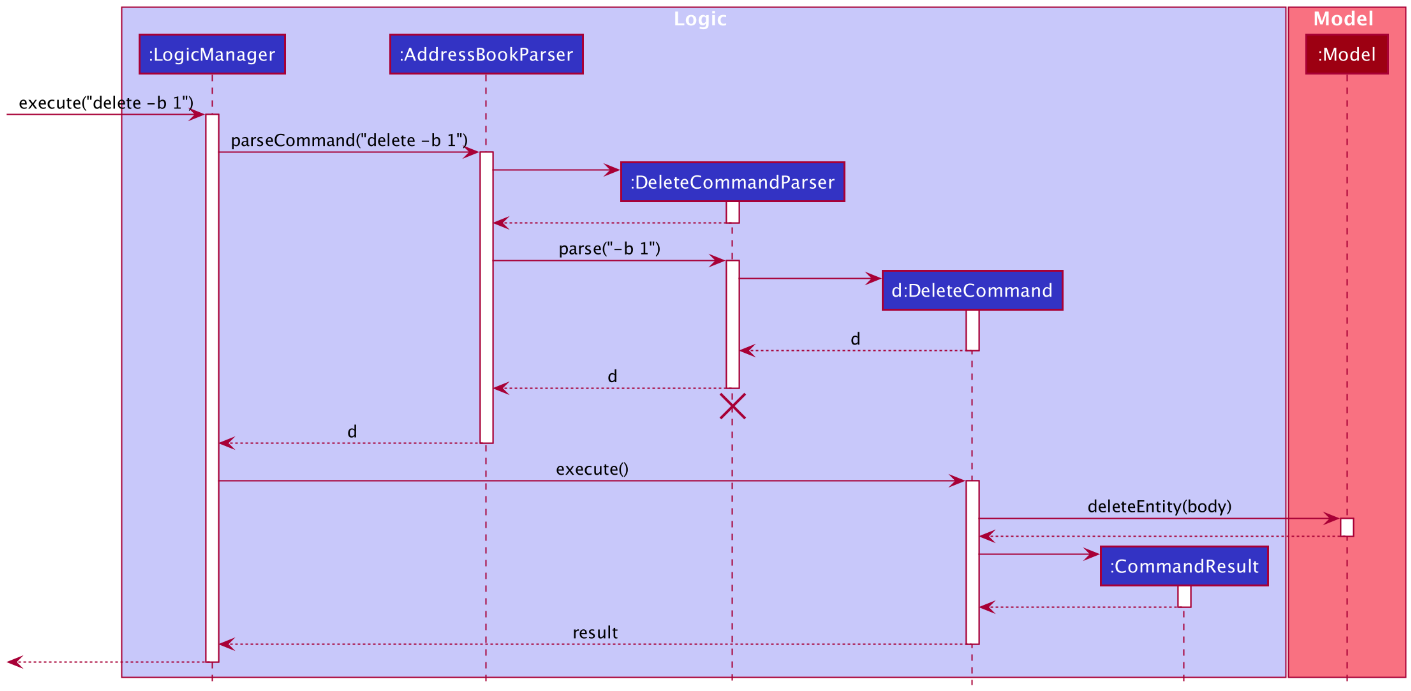


Figure 6. Interactions Inside the Logic Component for the `delete -b 1` *Command*

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.

2.4. Model Component

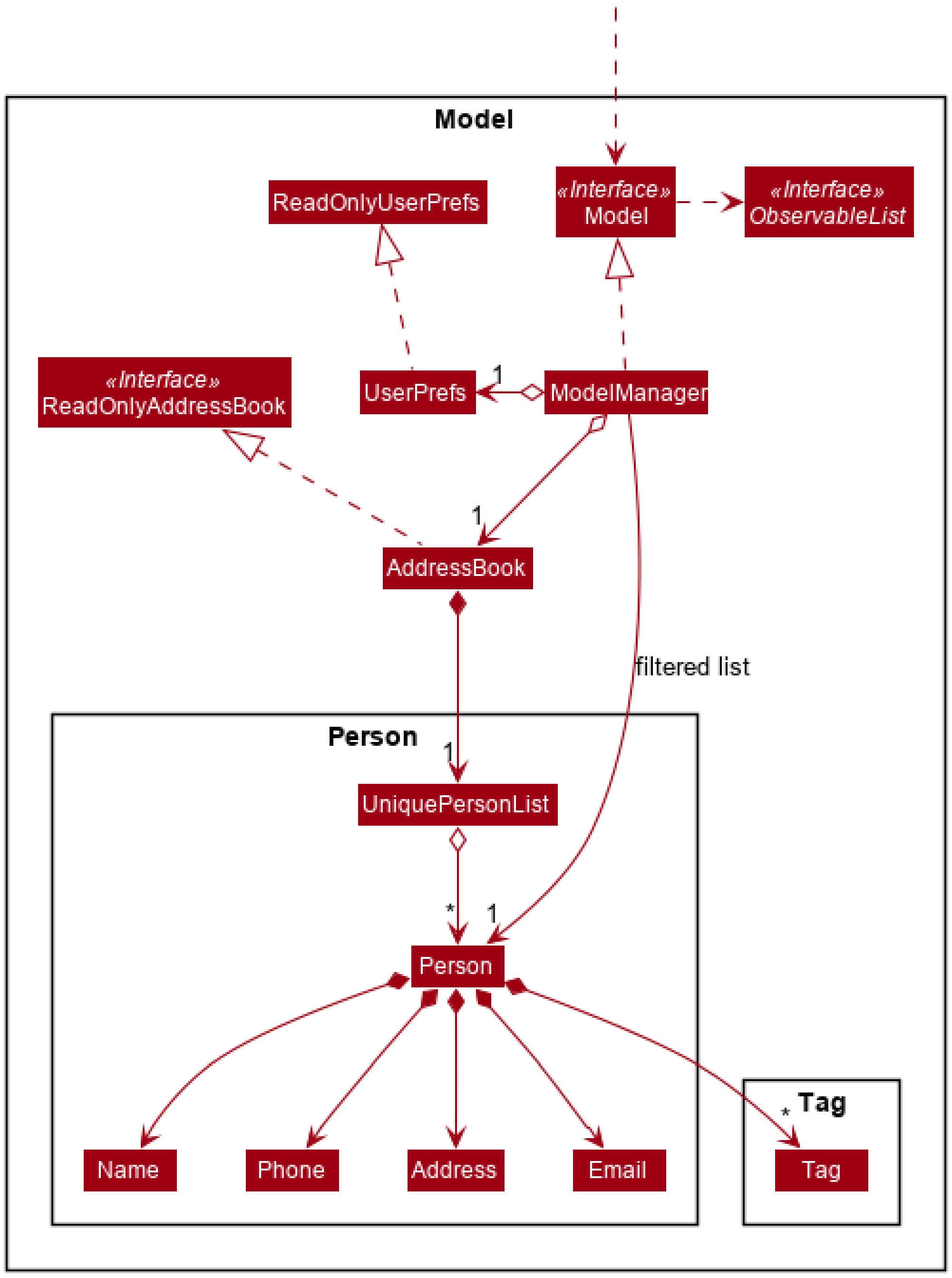


Figure 7. Structure of the Model Component

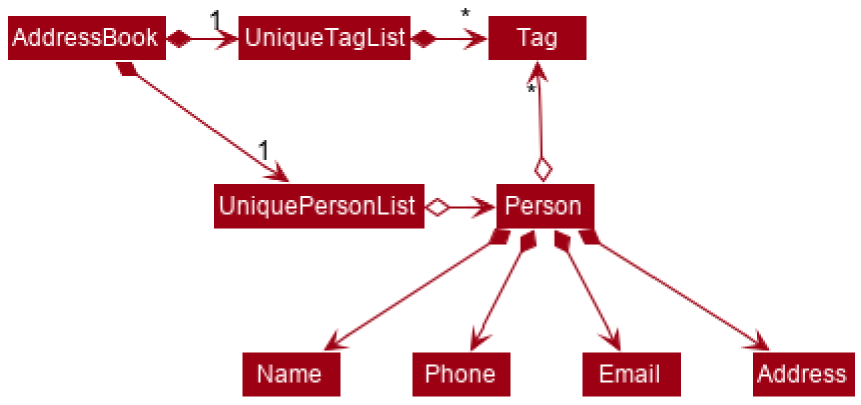
API : `Model.java`

(https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/model/Model.java)

The `Model` ,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

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



2.5. Storage Component

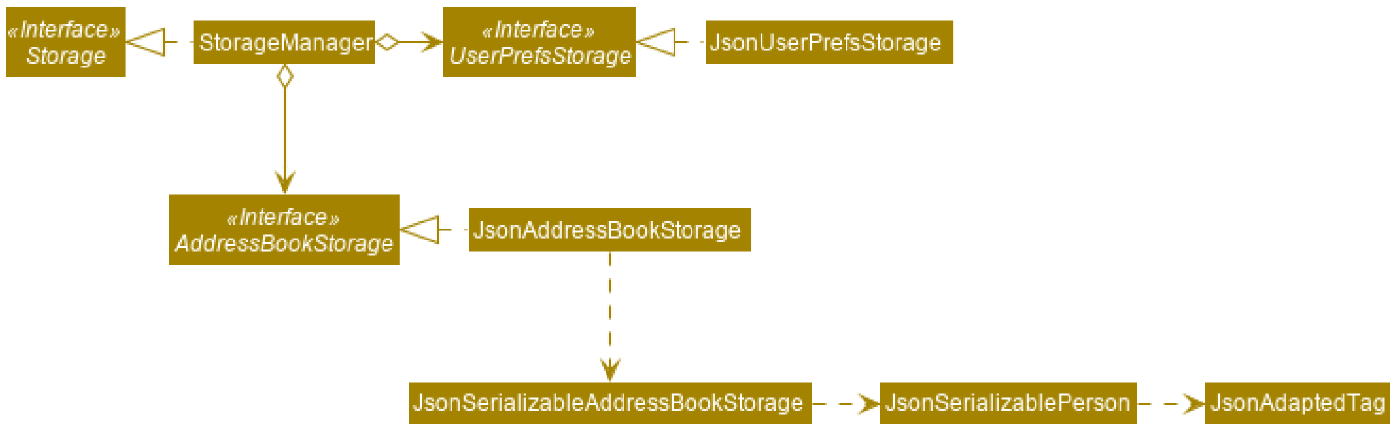


Figure 8. Structure of the Storage Component

API : `Storage.java` (https://github.com/AY1920S1-CS2103T-T13-2/tree/master/src/main/java/seedu/address/storage/Storage.java)

The `Storage` component,

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

2.6. Common Classes

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

3. Implementation

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

3.1. Unique Identification Number

- In Mortago, each entity is assigned a unique `IdentificationNumber` .
- `IdentificationNumbers` in Mortago consist of two parts: the `String typeOfEntity` referring to whether it is a worker, body or fridge, and `Integer idNum` referring to its unique ID number.
- Each `IdentificationNumber` is automatically generated within the application, based on the next sequential ID number available.
- The generation of unique `IdentificationNumbers` allows the user to identify different entities solely based on their `IdentificationNumber` , without relying on attributes such as `name` which may have similar duplications within the system.

3.1.1. Implementation

The generation of unique `IdentificationNumbers` is facilitated by `UniqueIdentificationNumberMaps` . The class diagram below illustrates the relation between `IdentificationNumber` and `UniqueIdentificationNumberMaps` .

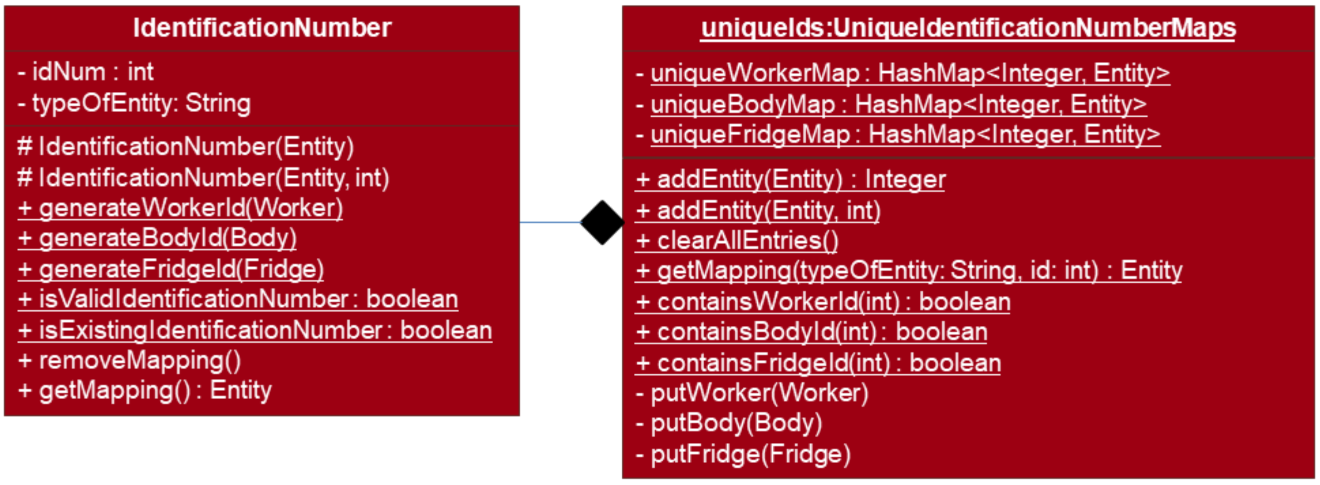

IdentificationNumber
- idNum : int
- typeOfEntity: String
# IdentificationNumber(Entity)
# IdentificationNumber(Entity, int)
+ generateWorkerId(Worker)
+ generateBodyId(Body)
+ generateFridgeId(Fridge)
+ isValidIdentificationNumber : boolean
+ isExistingIdentificationNumber : boolean
+ removeMapping()
+ getMapping() : Entity
uniqueIds:UniqueIdentificationNumberMaps
- uniqueWorkerMap : HashMap<Integer, Entity>
- uniqueBodyMap : HashMap<Integer, Entity>
- uniqueFridgeMap : HashMap<Integer, Entity>
+ addEntity(Entity) : Integer
+ addEntity(Entity, int)
+ clearAllEntries()
+ getMapping(typeOfEntity: String, id: int) : Entity
+ containsWorkerId(int) : boolean
+ containsBodyId(int) : boolean
+ containsFridgeId(int) : boolean
- putWorker(Worker)
- putBody(Body)
- putFridge(Fridge)

Figure 9. Unique `IdentificationNumber` *Class Diagram*

Determining the next free number

Note that in the diagram above, `UniqueIdentificationNumberMaps` keeps three `HashMap`, one for each entity. In each `HashMap`, the `Integer` ID number serves as the key, which maps to the entity it is assigned to. This keeps track of the numbers currently assigned to all entities and allows the next free `Integer` to be assigned to a newly added entity.

The code snippet below demonstrates how the next free number is determined.

```
private static Integer putWorker(Worker worker) {
    Set<Integer> keys = uniqueWorkerMap.keySet();
    int numOfKeys = keys.size();
    for (int id = 1; id <= numOfKeys; id++) {
        if (uniqueWorkerMap.get(id) == null) {
            uniqueWorkerMap.put(id, worker);
            return id;
        }
    }
    int newId = numOfKeys + 1;
    uniqueWorkerMap.put(newId, worker);
    return newId;
}
```

In the above `putWorker` method, the set of `keys` representing the existing ID numbers are generated and iterated through, based on the size of the `keySet`. This sequential iteration checks for any number that is not assigned to any worker due to the deletion of the latter, which removes the mapping of the ID number to the deleted worker. If there is an existing gap in the sequential iteration of numbers, this number is assigned to the newly added worker. If there is no gap available, the next highest number is assigned to the worker.

Execution sequence of generating a unique `IdentificationNumber` for a worker:

1. The user executes `add -w /name Zach …` to add a new worker to Mortago.
2. `AddCommandParser` parses the given input and calls the `Worker` constructor.
3. In the constructor, the worker's `IdentificationNumber` is created using `IdentificationNumber::generateNewWorkerId()`.
4. Consequently, `generateNewWorkerId()` creates a new `IdentificationNumber`, where the number is determined after the execution of `UniqueIdentificationNumberMaps::addEntity()`.
5. `UniqueIdentificationNumberMaps::addEntity()` subsequently calls `UniqueIdentificationNumberMaps::putWorker()`, which inserts the worker into the worker `HashMap` and returns an ID number that is currently not assigned to a worker.

The figure below illustrates the sequence diagram of the aforementioned steps.

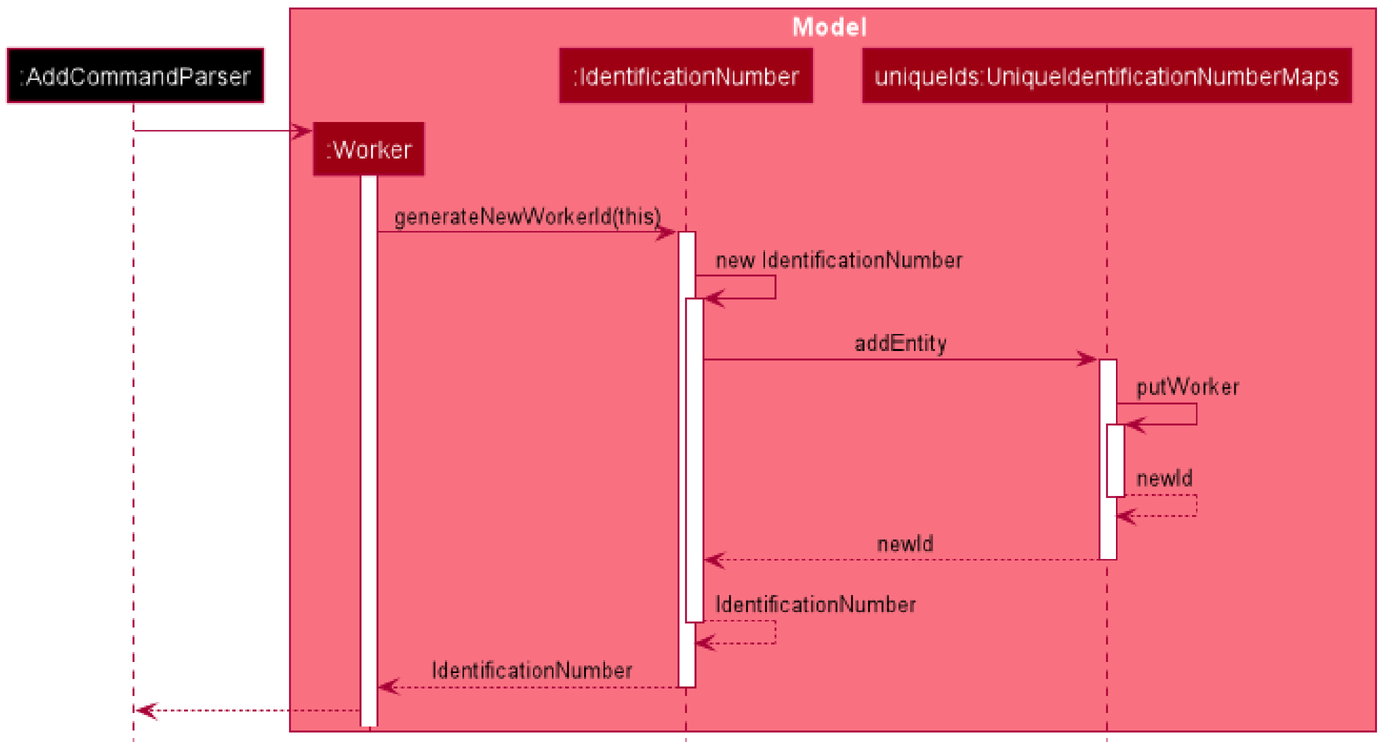


Figure 10. Generation of unique `IdentificationNumber` *Sequence Diagram*

Note that the execution sequence is similar for the addition and generation of unique `IdentificationNumber` for fridges and bodies.

3.1.2. Design Considerations

Aspect: Tracking of numbers and determination of next free number

- Alternative 1: Three counters that track the total number of each entity in the system.
  - Pros: Easy to implement.
  - Cons: Does not cater for deletion of entity as deletion creates a gap.
- Alternative 2 (current choice): A `HashMap` keeping track of the ID numbers and their respective assigned entity.
  - Pros: Aside from tracking a unique `keySet` of ID numbers, `HashMap` caters for deletion of entities as the `keySet` can be iterated through to check for any gaps introduced during deletion. This also allows the assigned (mapped) entity to be accessible through the `HashMap` via the `IdentificationNumber#getMapping()` method.
  - Cons: Harder to implement, especially for unit testing since a unique `IdentificationNumber` cannot be duplicated usually.

Alternative 2 is chosen due to the comprehensive benefits of utilizing `HashMap` in terms of:

- Tracking of unique keys
- Catering for deletion of ID numbers and filling of the gap

- Increasing the ease of accessibility of mapped entities

The difficulty in testing can be circumvented by executing `UniqueIdentificationNumberMaps::clearAllEntries()` before each test to reset the `HashMaps`.

3.2. Undo/Redo Feature

3.2.1. Implementation

The core of undo/redo is placed in the undo/redo history of `ModelManager`. Its mechanism uses the Command design pattern, allowing each individual command to be undone/redone without knowing the specific command type.

Firstly, there are two instances of `CommandHistory`, stored internally as `commandHistory` and `undoHistory` in `ModelManager`. The former instance stores previously executed commands while the latter stores previously undone commands. `CommandHistory` wraps a `Deque<UndoableCommand>`. Its functions imposes a `MAX_SIZE` which determines how many commands can be stored in the command history.

In `ModelManager`, four key operations regarding `CommandHistory` are implemented:

- `ModelManager#addExecutedCommand(UndoableCommand command)` — Adds a command that was executed to the start of `commandHistory`.
- `ModelManager#getExecutedCommand()` — Removes the last command that was executed and added to `commandHistory` and returns it.
- `ModelManager#addUndoneCommand(UndoableCommand command)` — Adds a command that was undone to the start of `undoHistory`.
- `ModelManager#getUndoneCommand()` — Removes the last command that was undone and added to `undoHistory` and returns it.

In the `Model` interface implemented by `ModelManager`, these four operations are respectively exposed as `Model#addExecutedCommand(UndoableCommand command)`, `Model#getExecutedCommand()`, `Model#addUndoneCommand(UndoableCommand command)`, and `Model#getUndoneCommand()`.

Next, the `UndoableCommand` stored in the `Model` is actually a normal `Command` that changes program state. The `UndoableCommand` class is an abstract class that extends the abstract `Command` class. Commands like `AddCommand` or `UpdateCommand` extends `UndoableCommand` instead of `Command`.

A class extending `UndoableCommand` must implement an additional method, `UndoableCommand#undo(Model model)`. This means that every child class of `UndoableCommand` has a custom `undo` implementation. `` ` ```UndoableCommand#redo(Model model)` is a concrete implementation of the `redo` mechanism and is inherited by all child classes. When an `UndoableCommand` is executed or undone, it is added to the `commandHistory` or `undoHistory` of ModelManager` respectively. Therefore, an `UndoableCommand` should not be undone before it was executed or redone before it was undone.

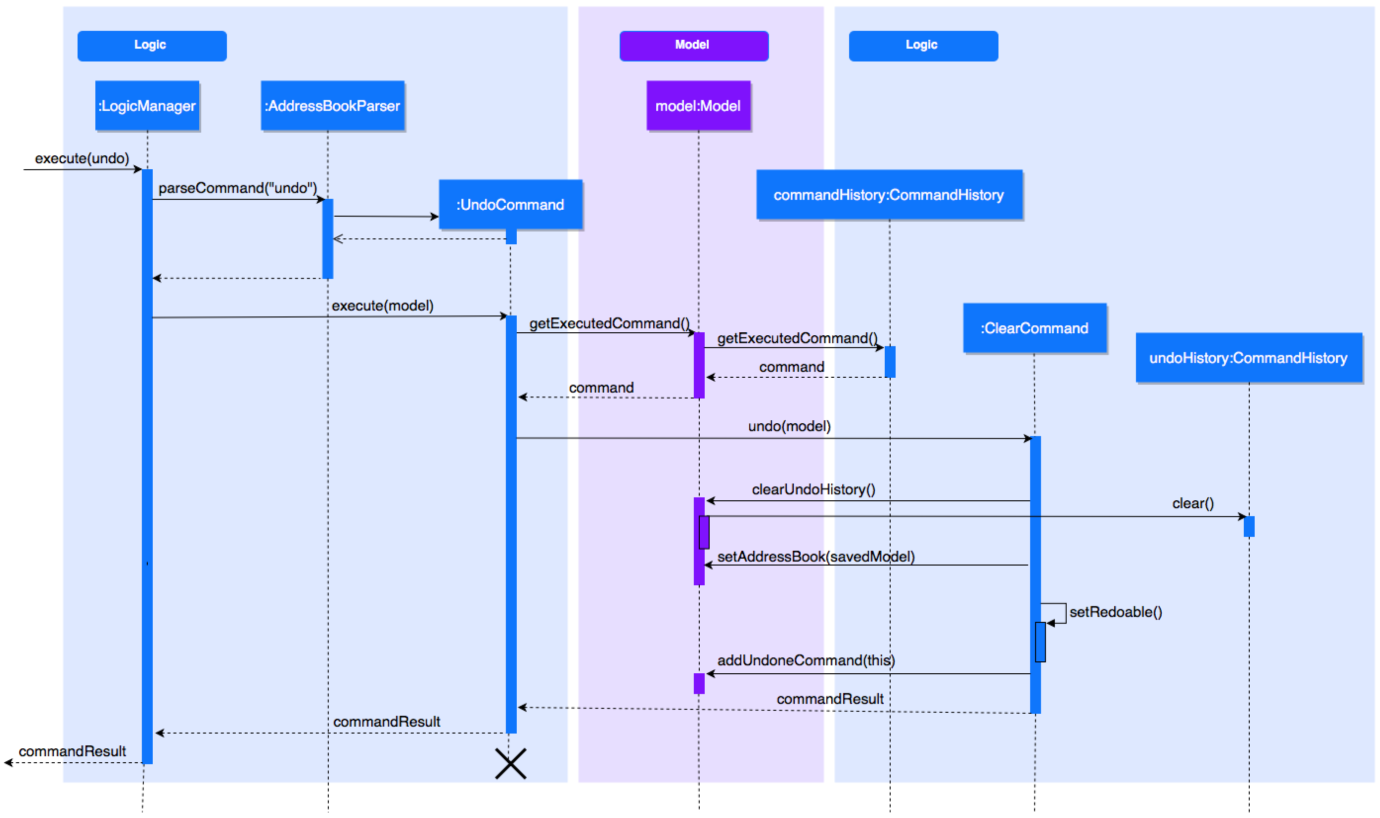
To defend against improper undoing or redoing, `UndoableCommand` contains an inner class, the enumeration `UndoableCommandState` which allows an `UndoableCommand` to have its state set to `UNDOABLE`, `REDOABLE`, and `PRE_EXECUTION`. Before a command is undone or redone, the command's state is checked for validity. These states are only changed when a `Command#execute(Model model)` or `UndoableCommand#undo(Model model)` has successfully executed. Thus, this lowers the likelihood of bugs in undo/redo.

Lastly, undo/redo is initiated when user input creates an `UndoCommand` or `RedoCommand`. When either of them are executed, they respectively get the last executed or undone command from the `CommandHistory` in `ModelManager`. As the retrieved command is an instance of `UndoableCommand`, an attempt will be made to execute `UndoableCommand#undo(Model model)` or `UndoableComman#redo(Model model)`. If it is successful, undo/redo is succesful. Otherwise, an error message is shown.

This is the mechanism of undo/redo, from start to end. Below is an example of a usage scenario that illustrates how undo/redo works.

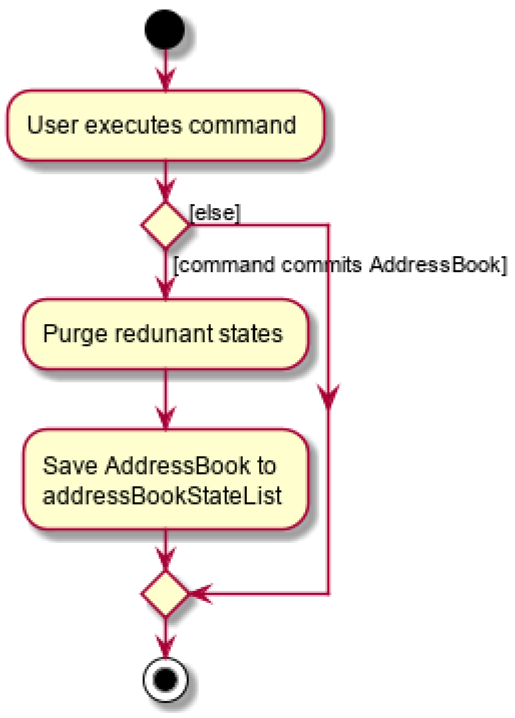
> If a command fails to execute correctly, it will not call `Model#addExecutedCommand()` or `Model#addUndoneCommand()`, thus preventing the command from being undone or redone.

The sequence diagram below shows how an undo command works to undo a `ClearCommand`:



After this is done, if a `redo` command was executed, the `ClearCommand` would simply be executed again.

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



User executes command
[else]
[command commits AddressBook]
Purge redunant states
Save AddressBook to addressBookStateList

3.2.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Individual command knows how to undo/redo by itself.

    - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
    - Cons: We must ensure that the implementation of each individual command are correct.
- **Alternative 2:** Saves the entire program state.
    - Pros: Easy to implement.
    - Cons: May have performance issues in terms of memory usage.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of address book states.
    - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
    - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedAddressBook`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
    - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
    - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.3. Notification Feature

A notification is a message that Mortago will display when the status of a body is automatically updated. If the next-of-kin of a body has not been uncontactable for 24 hours, Mortago will prompt the user to contact the police for a more thorough investigation and change the status of the body from `ARRIVED` to `CONTACT_POLICE`.

3.3.1. Current Implementation

Given below is the execution process of the `NotificationCommand`:

1. The user executes `add -b /name John …` to add a new body to Mortago.
2. The add command also created a new new `Notif` using this body as the parameter.
3. The add command instantiates a new`NotifCommand` with `Notif`, `period`, `unit` as parameters. `period` is a static final integer initialized for 10 seconds and `unit` is static final TimeUnit set at TimeUnit.SECONDS.
4. `NotifCommand` creates a new static `ScheduledExecutorService` which is scheduled to update the `BodyStatus` to `CONTACT_POLICE` if there is no change in its status for the specified `period` using the `UpdateCommand`.

5. After the `BodyStatus` is updated, a `NotifWindow` will be shown to alert the user of the automatic update in `BodyStatus`.

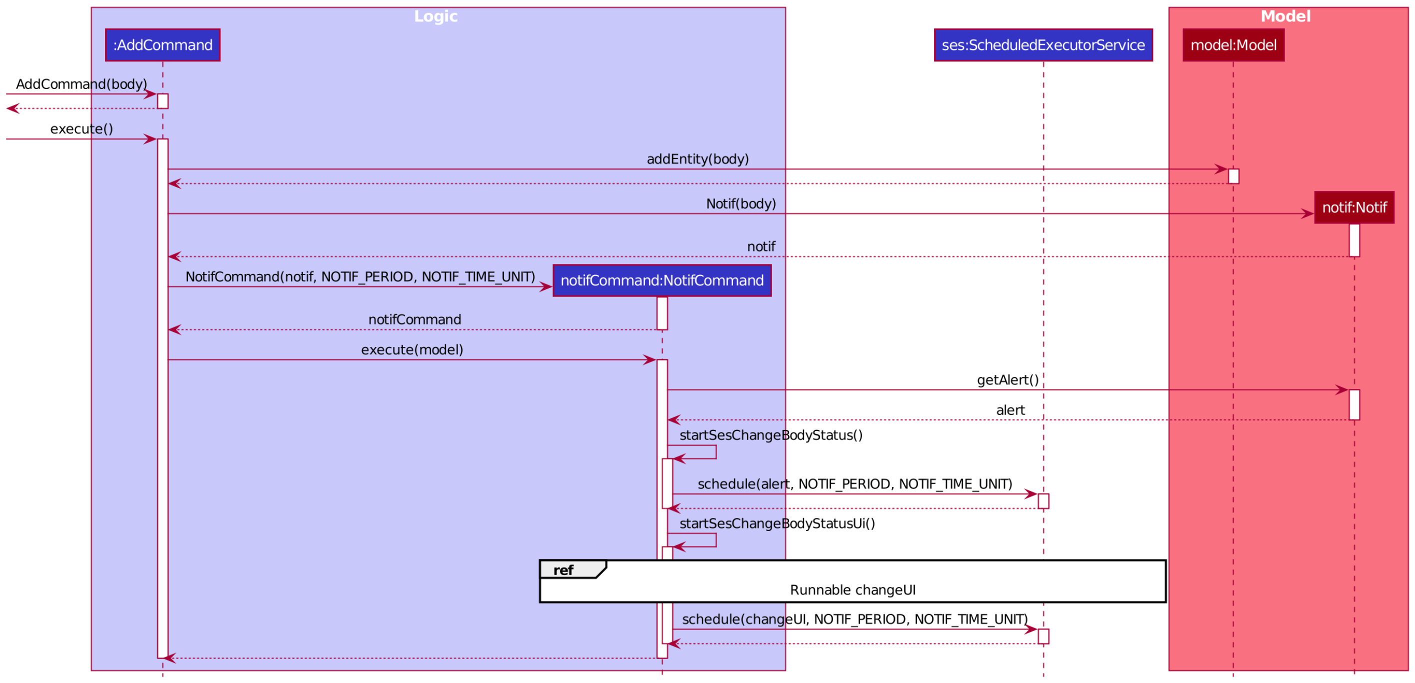

Logic
:AddCommand
ses:ScheduledExecutorService
Model
model:Model
AddCommand(body)
execute()
addEntity(body)
Notif(body)
notif:Notif
notif
NotifCommand(notif, NOTIF_PERIOD, NOTIF_TIME_UNIT)
notifCommand:NotifCommand
notifCommand
execute(model)
getAlert()
alert
startSesChangeBodyStatus()
schedule(alert, NOTIF_PERIOD, NOTIF_TIME_UNIT)
startSesChangeBodyStatusUi()
ref
Runnable changeUI
schedule(changeUI, NOTIF_PERIOD, NOTIF_TIME_UNIT)

Figure 11. Notification Command Sequence Diagram

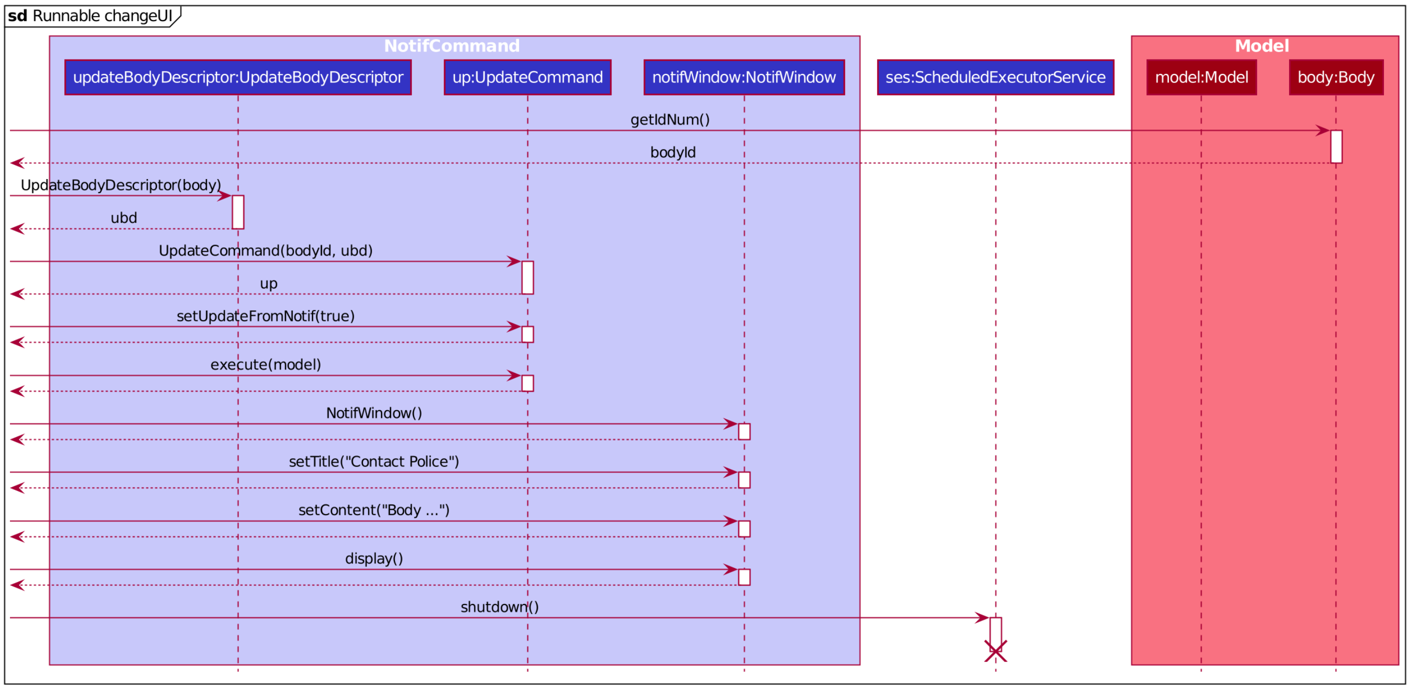


Figure 12. ChangeUI function Sequence Diagram

3.3.2. Design Considerations

Aspect: How to delay change in BodyStatus

- **Alternative 1 (current choice):** Use `ScheduledExecutorService`.
  - Pros: Does not depend on thread synchronization and avoids the need to deal with threads directly.
  - Cons: May cause memory leaks if cache is not cleared.

- **Alternative 2:** Use `Thread.sleep`
  - Pros: Straightforward way to delay a thread.
  - Cons: May quickly run into OutOfMemory error.

3.3.3. Aspect: How to reflect change in BodyStatus on UI

- **Alternative 1 (current choice):** Use `UpdateCommand` .
  - Pros: Easy to implement.
  - Cons: Increases `NotifCommand's dependency.
- **Alternative 2: Directly make changes to the model.
  - Pros: Does not increase dependency .
  - Cons: Cause a lot of repetition of code.

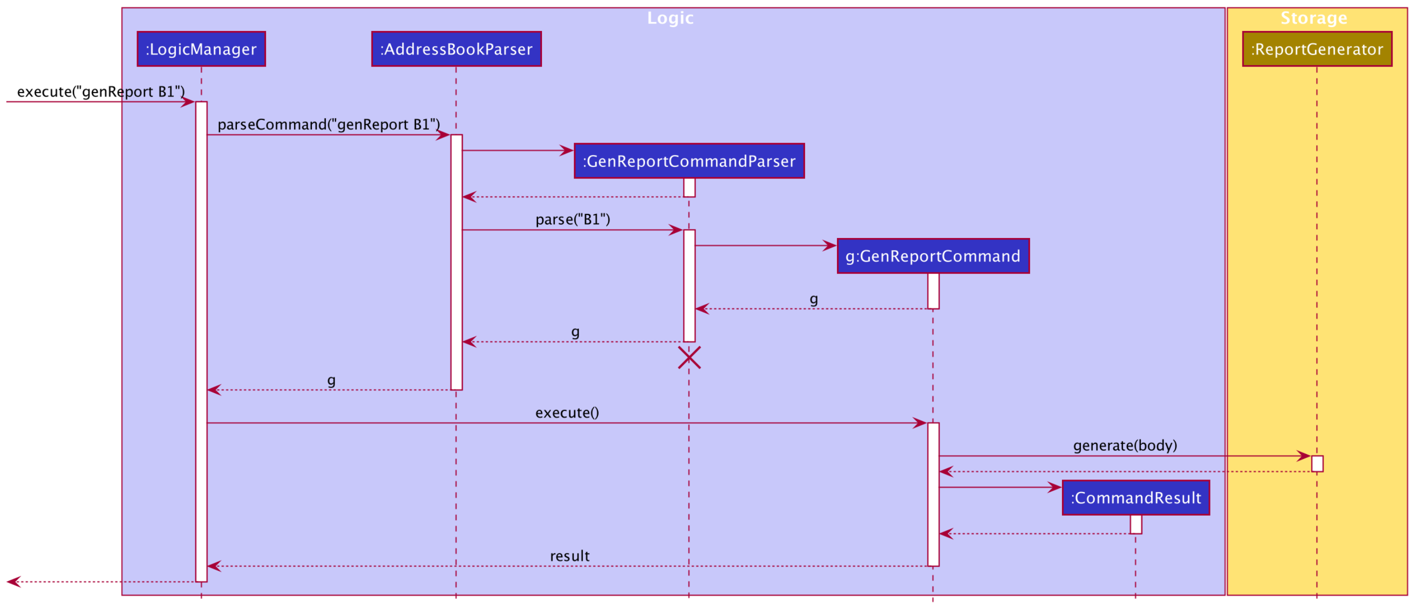
3.4. Generate PDF Feature

3.4.1. Implementation

The generate PDF feature is facilitated by `ReportGenerator` . It extends `Mortago` with the ability to create a report, supported by iText external library. Additionally, it implements the following operations:

- `ReportGenerator#generate(body)` — Creates report in a PDF file for the specific body.
- `ReportGenerator#generateAll()` — Creates report in a PDF file for all bodies registered in Mortago.

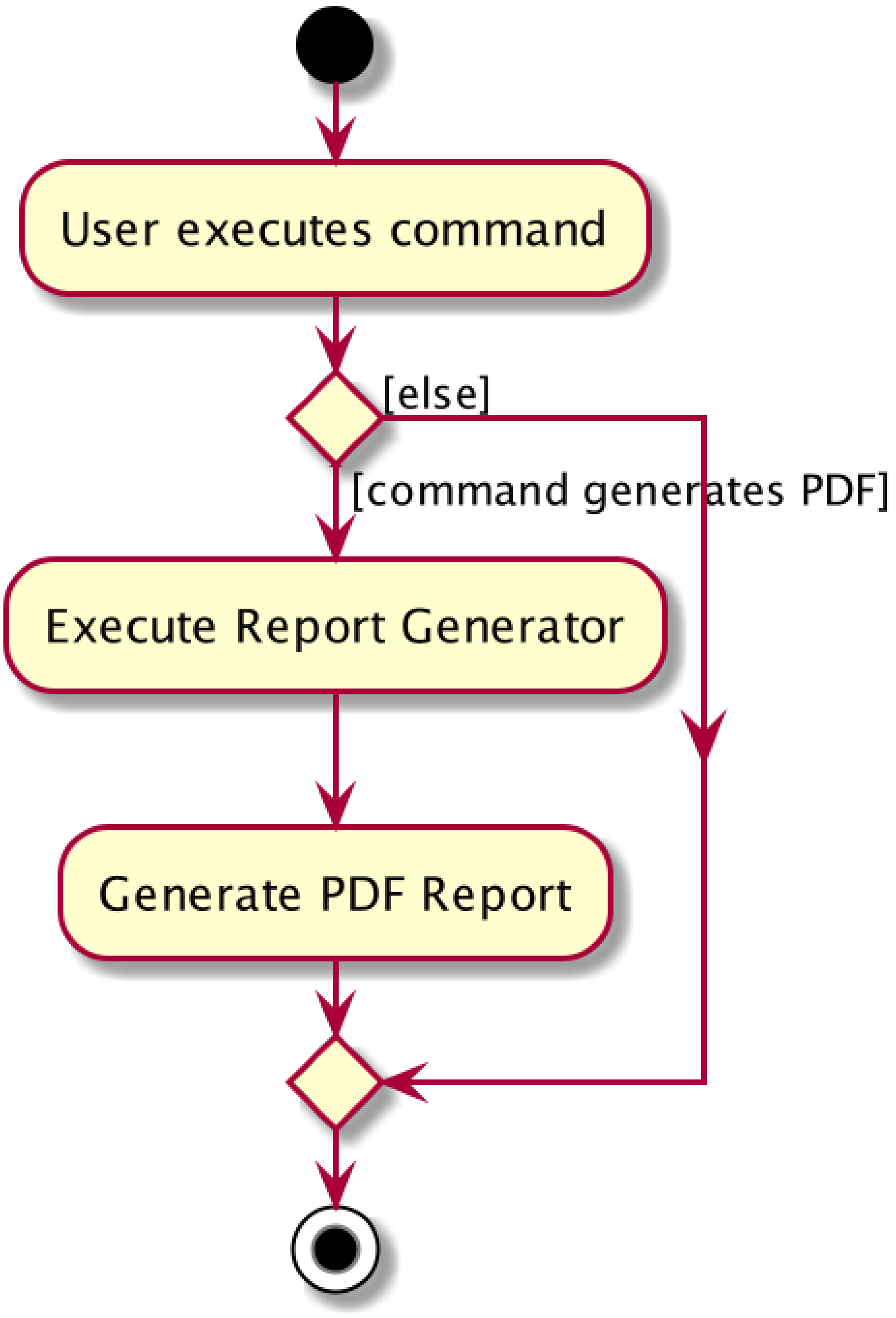
The following sequence diagram shows how the execute operation works:



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

The `genReport <BODY_ID>` command calls `ReportGenerator#execute()` , which creates the document.

The following activity diagram summarizes what happens when a user executes a `genReport <BODY_ID>` command:



3.4.2. Design Considerations

Aspect: How generate report executes

- **Alternative 1 (current choice):** Creates a PDF file.

    - Pros: Implementation is easy.
    - Cons: Implementation must ensure that each individual body attribute is correct.
- **Alternative 2:** Creates a Word Document file.
    - Pros: Implementation allows user to edit the contents of the report.
    - Cons: Implementation defeats the purpose of being automated.

Aspect: How report is formatted

- **Alternative 1 (current choice):** Uses tables to organise related details in the report.
    - Pros: Implementation allows report to be organised, increases readability for user.
    - Cons: Implementation is tedious.
- **Alternative 2:** Lists all attributes in the report without any formatting.
    - Pros: Implementation is easy.
    - Cons: Implementation decreases readability for user.

3.5. Logging

We are using `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 3.6, “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

3.6. Configuration

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

3.7. User Interface and Dashboard Enhancement

The dashboard of Mortago plays a key part in presenting a sleek, organised, and concise overview to the entities in the system. Thus, designing an aesthetic and functional dashboard is a crucial aspect for Mortago.

3.7.1. Surface Design and Coloring

Mortago draws upon the guidelines specified in Material.io (https://material.io/design/color/dark-theme.html#) to design a dark theme that enhances visual ergonomics via the minimization of eye strain due to the bright luminance emitted by screens.

Visual Contrast Between Surfaces and Text

With reference to Web Content Accessibility Guidelines' (WCAG), the guidelines recommend that the contrast level between dark surfaces and white text to be at least 15:8:1 to ensure visual accessibility. Thus, the following color values are set within Mortago:

Aspect of Mortago	Color	Preview
Background	derive(#121212, 25%)	⬛
Surface	#121212	⬛
Primary	#FF7597	🟥

The background color uses the `derive` function in `JavaFX`, which computes a color that is brighter or darker than the original, based on the brightness offset supplied. Here, notice that the background color is brighter than the surface color, as opposed to what Material.io dictates. This is because bulk of the screen space in the Mortago is taken up by surfaces to optimize the amount of information available to the user, hence by giving surfaces a darker brightness, this improves **overall** accessibility and visual ergonomics. In addition, this achieves elevation between surfaces and the background, which is aided by drop shadows by surfaces as well.

The primary color is desaturated to reach a WCAG’s AA standard of minimally 4:5:1. This facilitates a mild yet impressionable visual aspect to Mortago while minimizing eye strain, as saturated colors can cause optical vibrations against the dark surface and exacerbate eye strain.

3.7.2. Replacing Windows Title Bar

In spirit with producing a sleek design, the standard Windows platform title bar was removed. This exposes the user interface to become one that is self-contained, while providing extra space at the top.

The following code snippet was placed in `MainApp#start()` to achieve this.

```
primaryStage.initStyle(StageStyle.TRANSPARENT);
```

Note that this has to be done before the stage is shown. Otherwise, the application will close automatically upon running.

However, with this removal, the default Windows functions such as the default OS close button will be inevitably removed as well. Hence, these buttons will have to be rebaked into the application.

3.8. [Proposed] Statistics Feature

3.8.1. Proposed Implementation

The statistics feature appears as a line chart of the number of bodies admitted over the past 10 days and is facilitated by `LineChartPanel`. It extends `UiPart` with an internal storage of the number of bodies admitted per day over the past 10 days. The line chart is part of the user interface and is initialised automatically when Mortago is launched.

3.9. [Proposed] Data Encryption

{Explain here how the data encryption feature will be implemented}

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

Target user profile:

- has a need to manage a significant number of bodies
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: Mortago replaces and improves upon the traditional whiteboard system. It provides a convenient dashboard for the mortuary manager to keep track of all bodies and fridges, outstanding tasks, and alerts. Mortago unites the different aspects of a mortuary and allows the mortuary management to be more accurate in managing tasks, calculate the availability of space, and generates reports automatically.

Appendix B: User Stories

Priorities: High (must have) - `* * *`, Medium (nice to have) - `* *`, Low (unlikely to have) - `*`

Priority	As a …	I want to …	So that I can…
`* * *`	mortuary manager	keep track of all bodies and fridges in a single dashboard using the dashboard command	look out for any outstanding work and keep myself up to speed
`* * *`	mortuary manager	have a dynamically updated dashboard	reduce errors as compared to manually updating a whiteboard
`* * *`	mortuary manager	key new bodies into the system	keep track of them

Priority	As a …	I want to …	So that I can…
* * *	mortuary manager	sort the bodies by certain characteristics	view bodies of a speciic category and generate statistics easily
* * *	mortuary manager	filter the bodies by certain characteristics	view bodies of a certain category and generate statistics easily
* * *	mortuary manager	update the status of each and every worker, body and fridge	know when (date & time) was each step of the process completed and the findings of each process (eg. cause of death)
* * *	mortuary manager	delete a worker, body and fridge	remove a worker when he quits, remove a wrong entry of the body, or remove a fridge
* * *	mortuary manager	switch between the dashboard and the detail views	view information in an appropriate format
* * *	mortuary manager	view all free and vacant fridges	keep track of the overall vacancy of the morgue

Priority	As a …	I want to …	So that I can…
* * *	mortuary manager	view all registered bodies	view all bodies in the morgue
* * *	mortuary manager	view all registered body parts	view all body parts in the morgue
* * *	mortuary manager	view all the commands the app is capable of	look at all the commands in one go
* * *	mortuary manager	view emergency hotlines	be efficient and respond quickly to emergencies
* * *	mortuary manager	read up on the use of a specific command of the app	understand a specific command which the app offers in more detail
* *	mortuary manager	be alerted to bodies unclaimed after 24hours	know when to start the administrative process for donation to medical research
* *	mortuary manager	receive routine reports from the app automatically	need not manually write in each and every single report

Priority	As a …	I want to …	So that I can…
* *	mortuary manager	assign workers to tasks	know who was responsible for a task
* *	mortuary manager	can undo my previous tasks	conveniently undo any wrong commands
* *	mortuary manager	redo my previous tasks	conveniently redo any undone commands
* *	mortuary manager	add new or existing workers	keep track of all the workers in the mortuary
* *	mortuary manager	add new or existing fridge	keep track of all the fridges in the mortuary
* *	mortuary manager	be able to see a history of changes	know if anything was inputted wrongly in the past
* *	mortuary manager	create mortuary bills with the app automatically	need not manually write in each and every single bill
* *	mortuary manager	view bills for past reports and individual reports	easily obtain past bills for my own reference / authorities / third party

Priority	As a …	I want to …	So that I can…
* *	mortuary manager	archive processed cases on a regular interval	review past cases when such a need arises
* *	mortuary manager	add comments and feedback to workplace processes	review these feedback and improve on them
* *	mortuary manager	feel happy when I see a beautifully designed dashboard	keep my mood up throughout the day
* *	mortuary manager	make sure that everything is organised and in order	be praised by my higher ups

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is `Mortago` and the **Actor** is the `mortuary manager`, unless specified otherwise)

Use case: View dashboard

MSS

1. Mortuary manager requests to view the dashboard
2. Mortago displays the dashboard.

   Use case ends.

Use case: Add body

MSS

1. Mortuary manager requests to add a body
2. Mortago adds the body into the system

   Use case ends.

Extensions

1a. Duplicate body found.

   1a1. Mortago shows an error message.

   Use case restarts at step 1.

1b. Mandatory fields are missing.

   1b1. Mortago shows an error message.

   Use case resumes at step 1.

Use case: Delete body

MSS

1. Mortuary manager requests to list bodies.
2. Mortago shows a list of bodies.
3. Mortuary manager requests to delete a specific body in the list.
4. Mortago deletes the body from the system.

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

   3a1. Mortago shows an error message.

   Use case resumes at step 2.

Use case: Find entry

MSS

1. Mortuary manager switches to the desired view (bodies or workers).
2. Mortuary manager specifies word to search.
3. Mortago shows a list of entries whose names matches the word.

   Use case ends.

Extensions

3a. The list is empty.

Use case ends.

Use case: Filter entries

MSS

1. Mortuary manager switches to the desired view (bodies or workers).
2. Mortuary manager specifies criteria for filter.
3. Mortago shows a list of entries that matches the criteria.

   Use case ends.

Extensions

3a. The list is empty.

Use case ends.

Use case: Sort entries

MSS

1. Mortuary manager switches to the desired view (bodies or workers).
2. Mortuary manager specifies criteria for sorting.
3. Mortago shows a list of entries sorted according to the specified criteria.

   Use case ends.

Extensions

3a. The list is empty.

Use case ends.

Use case: Generate report

MSS

1. Mortuary manager requests to generate report for a specific body.
2. Mortago creates a new PDF report with body ID as the title.

   Use case ends.

Extensions

1a. The given body ID is invalid.

1a1. Mortago shows an error message.

Use case ends.

Use case: Alert for unclaimed bodies.

MSS

1. Mortuary manager wants to be reminded of the next line of action if next of kin cannot be contacted within 24 hours.
2. Mortago maintains a record of all the alerts about unidentified and unclaimed bodies until their status is changed.
3. Mortago shows pop-up alerts after 24 hours from the point of arrival of the body in the mortuary.

   Use case ends.

Extensions

1a. There are no alerts

Use case ends.

Use case: Undoing a previous command

MSS

1. Mortuary manager requests to undo the previous command.
2. Mortago undoes the command.
3. Mortago updates the GUI to reflect the new changes.

   Use case ends.

Extensions

2a. There is no command to undo.

Use case ends.

2b. An error occurred when undoing the command.

- 2b1. Mortago shows an error message and nothing is changed.

  Use case ends.

Use case: Redoing an undone command

MSS

1. Mortuary manager requests to redo the last undone command.
2. Mortago redoes the command.
3. Mortago updates the GUI to reflect the new changes.

   Use case ends.

Extensions

2a. There is no command to redo.

Use case ends.

2b. An error occurred when undoing the command.

- 2b1. Mortago shows an error message and nothing is changed.

  Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `11` or above installed.
2. Should be able to hold up to 1000 bodies without a noticeable sluggishness in performance for typical usage.
3. A mortuary manager 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.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Body

A corpse

Worker

An employee in the mortuary

Fridge

A fridge used to store a body in the mortuary

Command-Line-Interface (CLI)

A text-based user interface (UI) used to view and manage computer files

Graphical User Interface (GUI)

An interface through which a user interacts with electronic devices such as computers, hand-held devices and other appliances. This interface uses icons, menus and other visual indicator (graphics) representations to display information and related user controls, unlike text-based interfaces, where data and commands are in text

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

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

G.1. Launch and Shutdown

1. Initial launch

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

    b. Double-click the jar file
    Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.

2. Saving window preferences

    a. Resize the window to an optimum size. Move the window to a different location. Close the window.

    b. Re-launch the app by double-clicking the jar file.
    Expected: The most recent window size and location is retained.

{ more test cases … }

G.2. Deleting a Body

1. Deleting a body while all bodies are listed

    a. Prerequisites: List all bodies using the `list -b` command. Multiple bodies in the list.

    b. Test case: `delete -b 1`
    Expected: Body with body ID number 1 is deleted from the list. Details of the deleted body shown in the status message. Timestamp in the status bar is updated.

    c. Test case: `delete -b 0`
    Expected: No body is deleted. Error details shown in the status message. Status bar remains the same.

    d. Other incorrect delete commands to try: `delete -b`, `delete -b x` (where x is larger than the list size) *{give more}*
    Expected: Similar to previous.

{ more test cases … }

G.3. Saving Data

1. Dealing with missing/corrupted data files

    a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }

Last updated 2019-10-24 12:46:18 UTC