Notably - Developer Guide

By: `Team Notably`   Since: `Feb 2020`   Licence: `MIT`

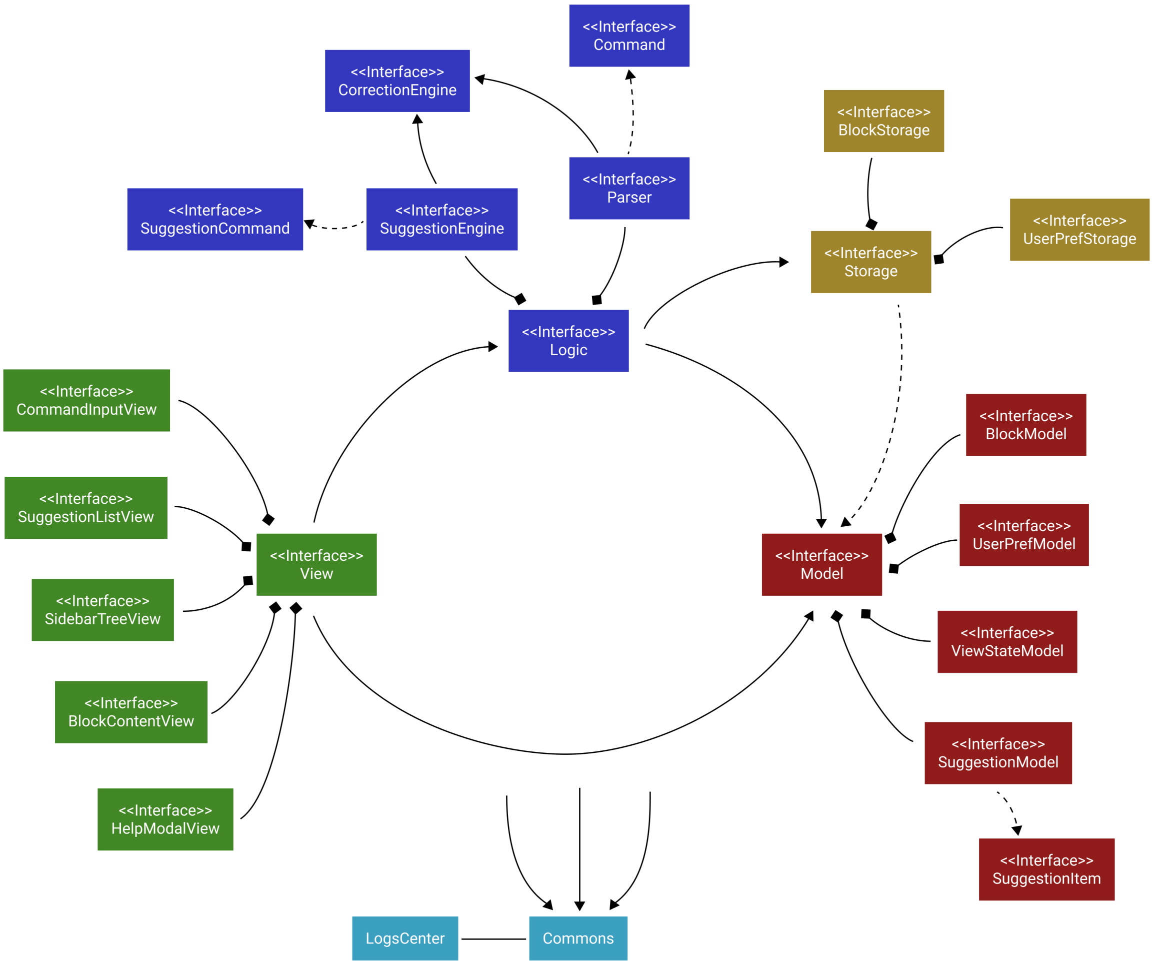
1. Setting up

Refer to the guide here.

2. Design

TIP The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

2.1. Architecture



<<Interface>> Command
<<Interface>> CorrectionEngine
<<Interface>> BlockStorage
<<Interface>> Parser
<<Interface>> SuggestionCommand
<<Interface>> SuggestionEngine
<<Interface>> Storage
<<Interface>> UserPrefStorage
<<Interface>> Logic
<<Interface>> CommandInputView
<<Interface>> BlockModel
<<Interface>> SuggestionListView
<<Interface>> View
<<Interface>> Model
<<Interface>> UserPrefModel
<<Interface>> SidebarTreeView
<<Interface>> ViewStateModel
<<Interface>> BlockContentView
<<Interface>> SuggestionModel
<<Interface>> HelpModalView
<<Interface>> SuggestionItem
LogsCenter
Commons

Figure 1. Architecture Diagram

2.1.1. Design pattern and data flow

The App is built following the Model-View-Controller design pattern.

In addition, the App's data flow is unidirectional. That is, all user interactions in `View` will trigger an appropriate handler in `Logic`, which in turn updates `Model` and `Storage`. Any data/state changes in `Model` will then propagate back to `View` automatically through JavaFX's Property and Binding.

In short, the App's data flow can be summarized as:
`View` → `Logic` → `Model` + `Storage` → `View`

2.1.2. Architecture-level components

Overall, the App consists of four components:

- **`View`**: View of the App.
- **`Logic`**: Business logic of the App.

- `Model`: In-memory representation of the App’s data/state.
- `Storage`: Reads data from, and writes data to, the hard disk.

In addition, `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.

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 defines it’s API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

2.1.3. 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 `new -t Notably -b Lorem ipsum`.

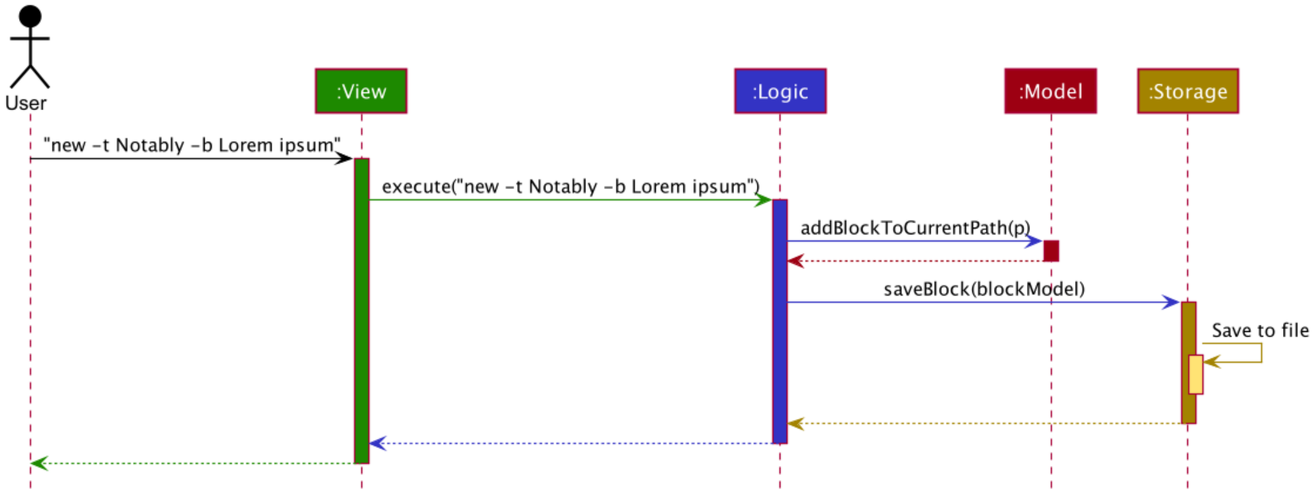


Figure 2. Component interactions for the `new -t Notably -b Lorem ipsum` command

2.2. View component

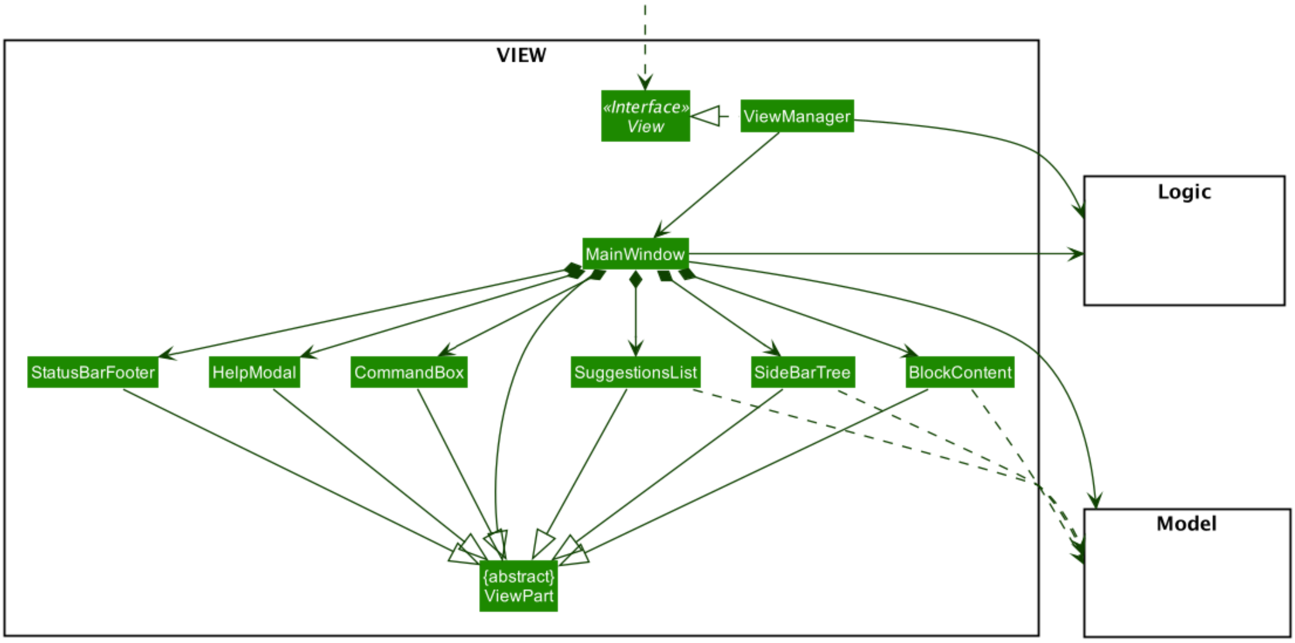


Figure 3. Structure of the View Component

API : `View.java`

The View consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `SuggestionsList`, `SideBarTree`, `HelpModal`, `BlockContent` etc. All these, including the `MainWindow`, inherit from the abstract `ViewPart` class.

The `View` component uses JavaFx framework. The layout of these View 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 `View` component,

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

2.3. Logic component

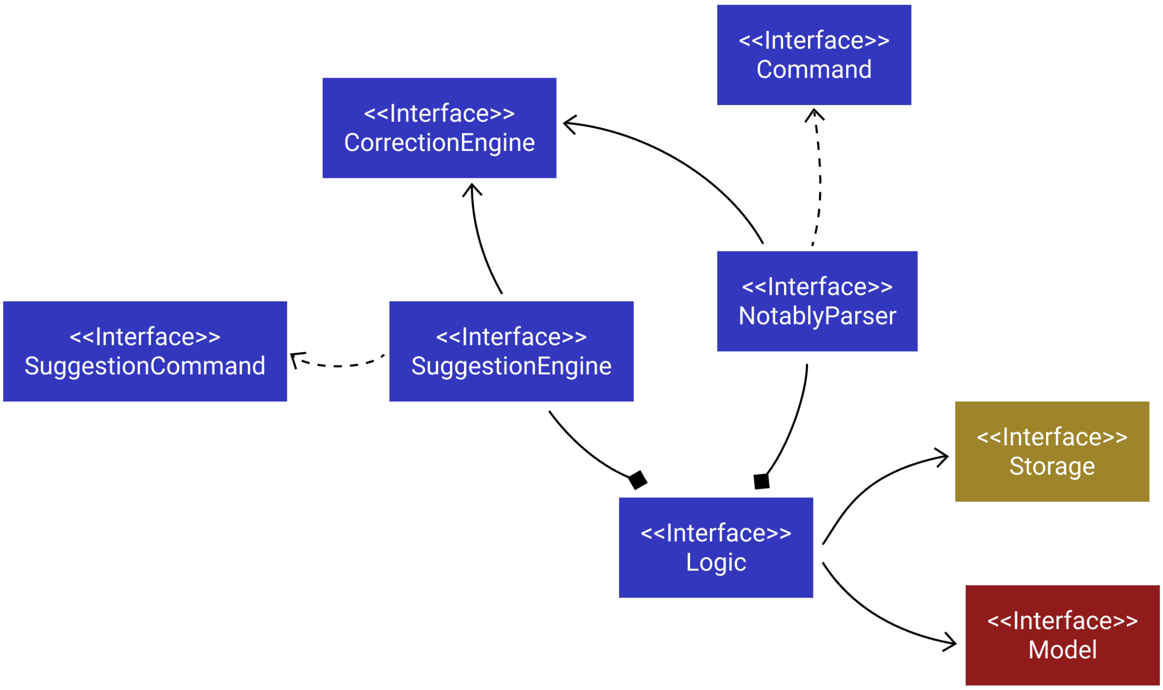


Figure 4. Architecture of Logic

API : `Logic.java`

`Logic` consists of 3 subcomponents:

- `NotablyParser`: Main parser of the App, deals with user command execution.
- `SuggestionEngine`: Deals with suggestions generation.
- `CorrectionEngine`: Deals with auto-correction.

2.3.1. NotabyParser component

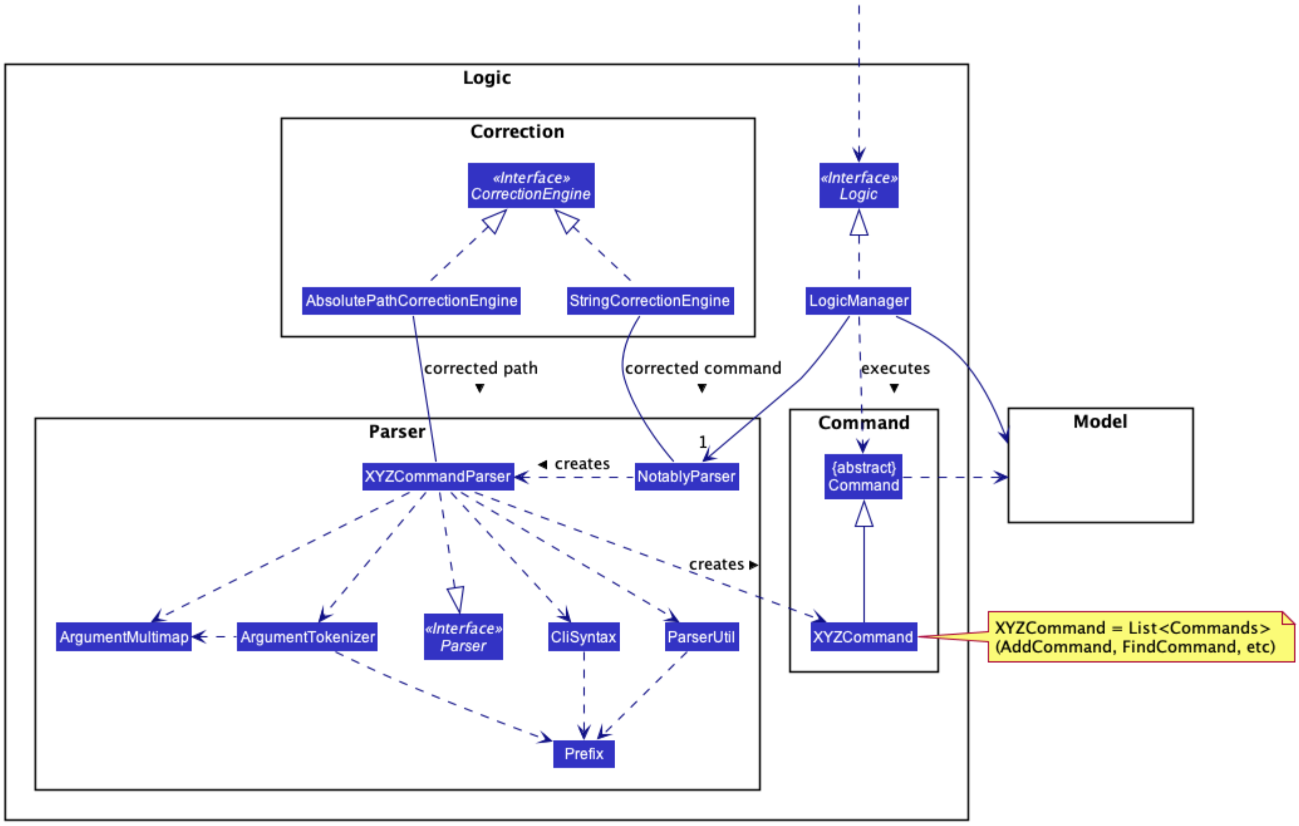


Figure 5. Class Diagram of the Logic Component

1. `Logic` uses the `NotablyParser` class to parse the user command.
2. This results in a `List<Command>` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a Note).
4. The updated model/data structure will automatically be reflected on to the `View`.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete cs2103")` API call.

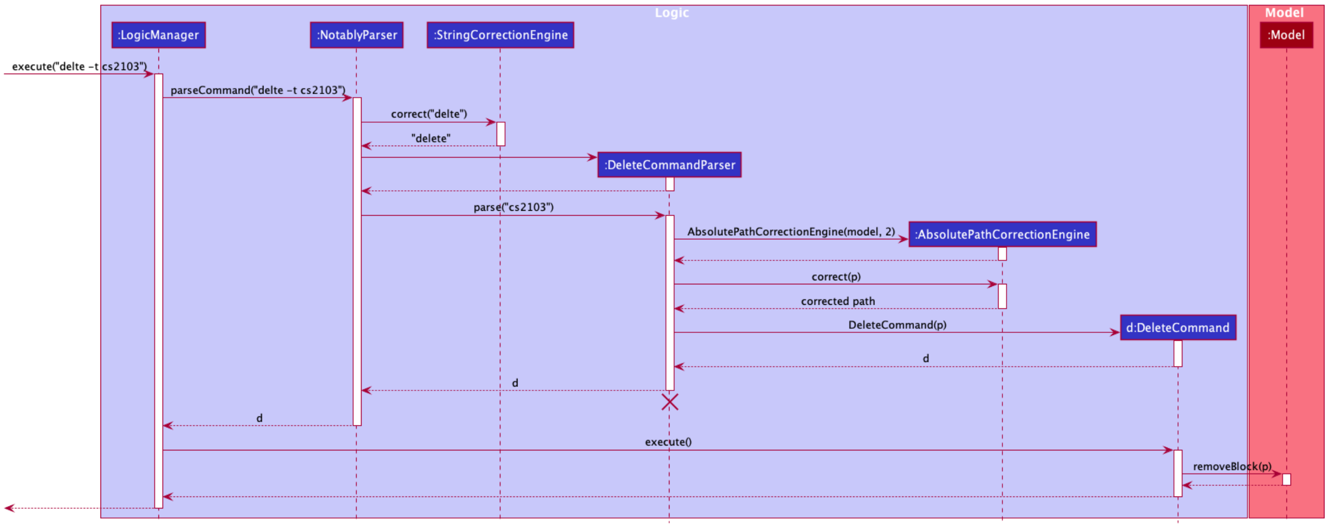


Figure 6. Interactions Inside the Logic Component for the `delete -t cs2103` Command

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.

2.3.2. SuggestionEngine component

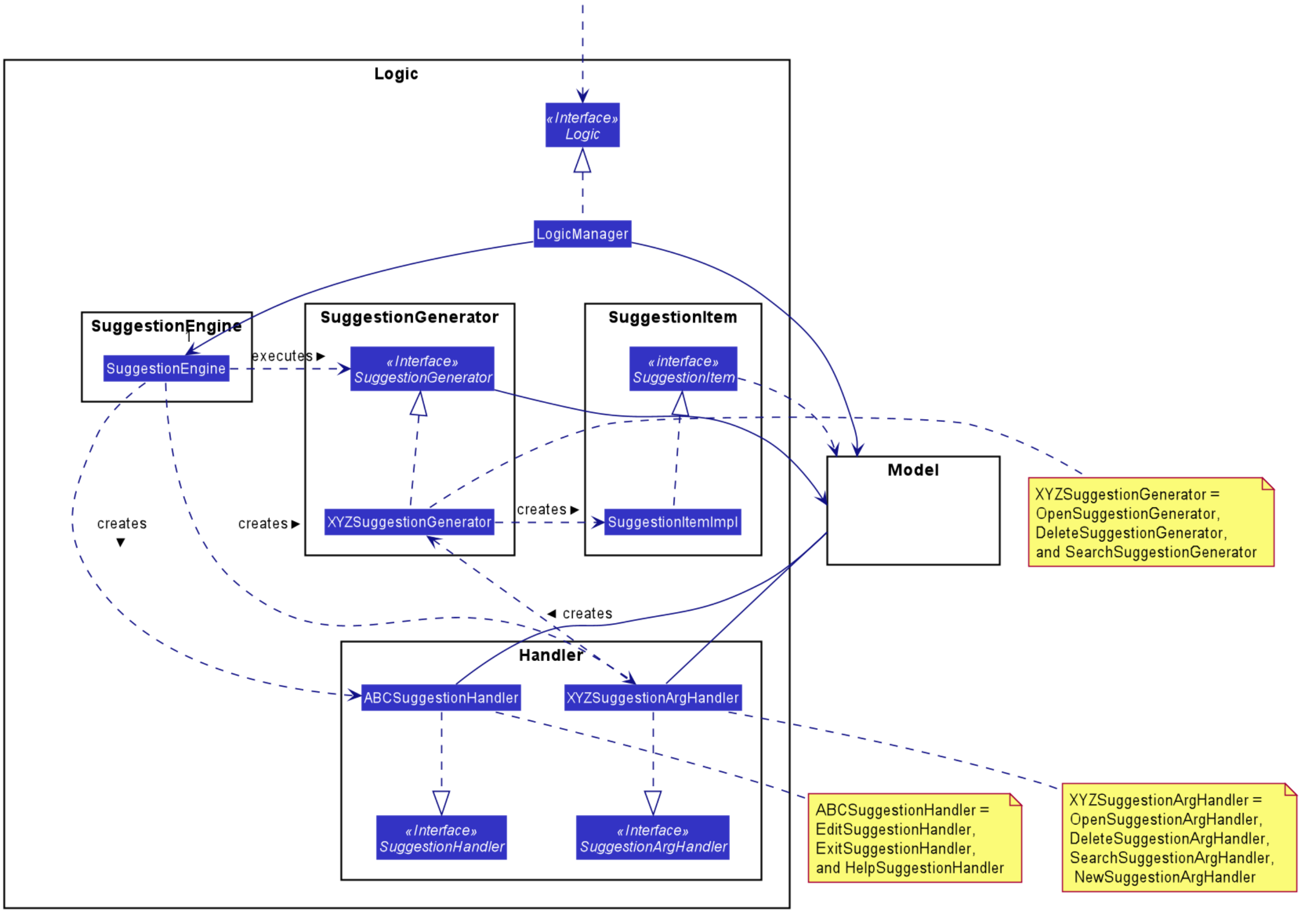


Figure 7. Class Diagram of the Suggestion Engine Component

API : `SuggestionEngine.java`

`SuggestionEngine` gives users the meaning of the command they input and a list of notes suggestions that they want to open, delete, or search.

1. `Logic` uses the `SuggestionEngine` class, to handle the user input.
2. According to the command the user inputs, `SuggestionEngine` will create a `XYZSuggestionArgHandler` or `ABCSuggestionHandler` object which implements `SuggestionArgHandler` and `SuggestionHandler` interface respectively. `XYZSuggestionArgHandler` are for commands that require argument parsing, i.e. `open`, `delete`, `search`, `new`, whereas `ABCSuggestionHandler` are for commands that do not require argument parsing, i.e. `edit`, `exit`, `help`.
3. If `SuggestionArgHandler` object is created: the `responseText` in the `Model` will be updated. This case will also result in the creation of `XYZSuggestionGenerator` object (except for `new` command) which implements `SuggestionGenerator` interface. `XYZSuggestionGenerator` is then executed by the `SuggestionEngine`.
4. If `SuggestionHandler` object is created: the `responseText` in the `Model` will be updated.
5. The `Model` could be affected in 2 ways:

- Update `responseText` of the `Model` (by the `SuggestionHandler` and `SuggestionArgHandler`): for instance, the input `open /` will set the `responseText` in the `Model` as "Open a note".
- Store a list of `SuggestionItem` in the `Model` (by the `SuggestionGenerator`).

6. The UI will then be able to retrieve the `responseText` and list of `SuggestionItem` from the `Model` to be displayed to the user.

Given below is the Sequence Diagram for interactions within the `Logic` and `Suggestion` component for the input `opne /a`.

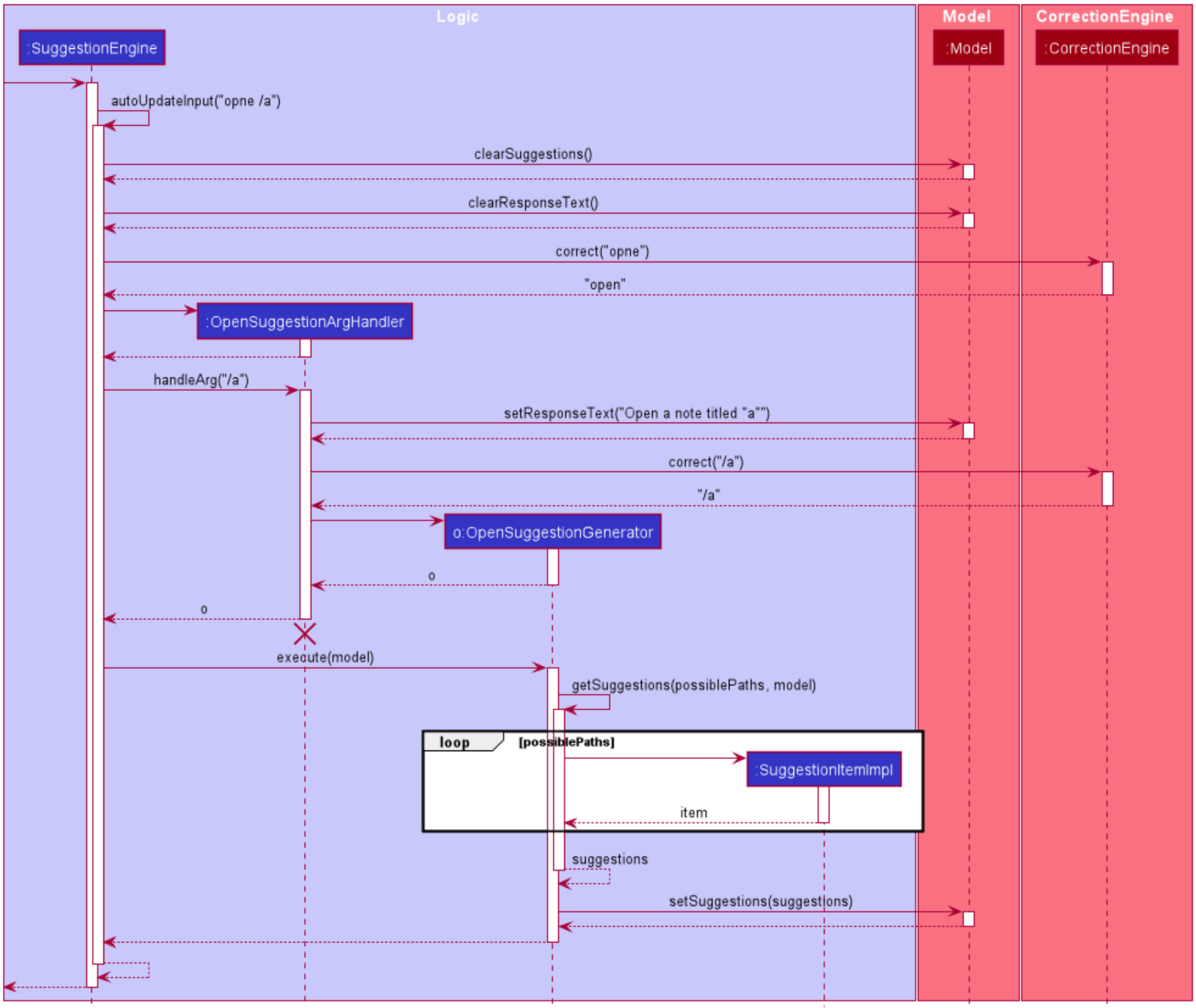


Figure 8. Interactions Inside the Logic and Suggestion Component for the input `opne /a`

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

2.3.3. CorrectionEngine component

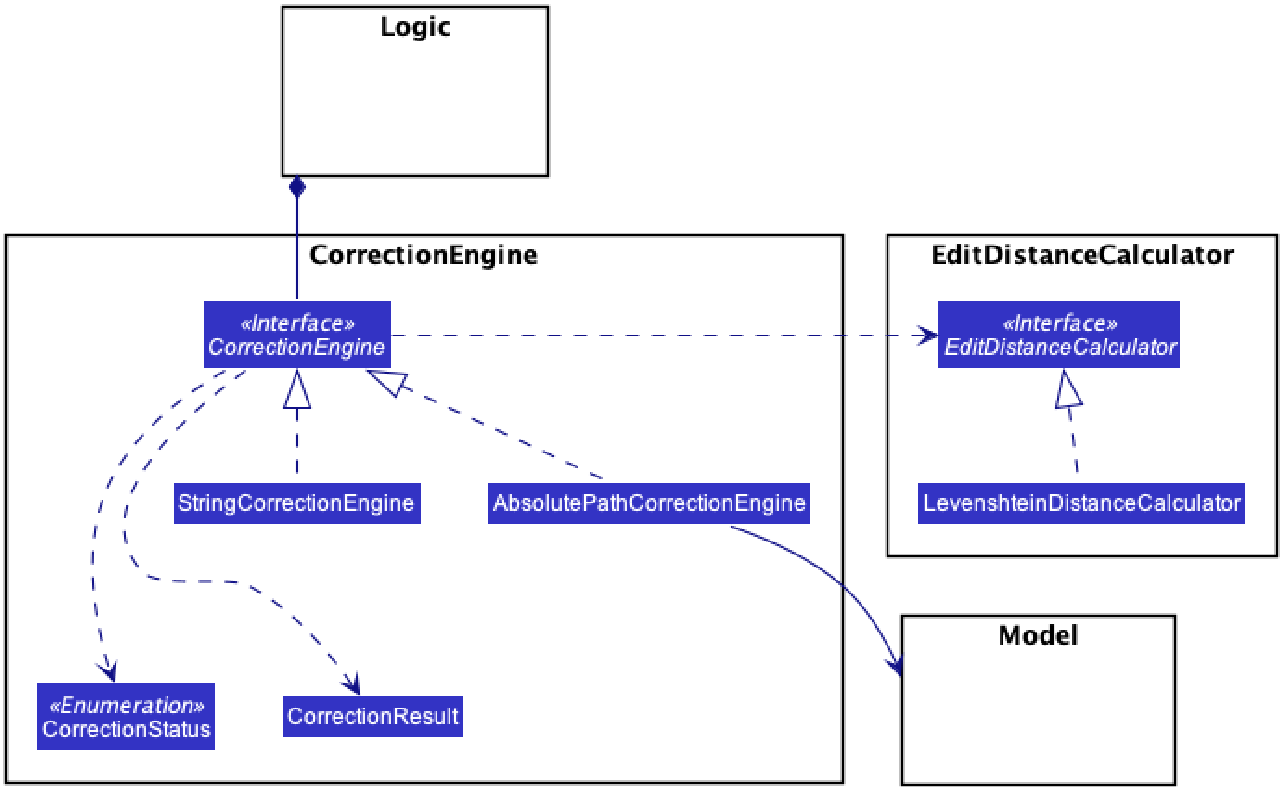


Figure 9. Class Diagram of the CorrectionEngine Component

The `CorrectionEngine` component revolves around two *API* s, namely:

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

- The `CorrectionEngine` interface, implemented by `StringCorrectionEngine` and `AbsolutePathCorrectionEngine`. Concrete implementations of `CorrectionEngine` are employed to correct an uncorrected user input.
- The `EditDistanceCalculator` interface, implemented by `LevenshteinDistanceCalculator`. Concrete implementations of `EditDistanceCalculator` are employed to calculate the edit distance between two strings.

Given below is the Sequence Diagram for interactions within the `StringCorrectionEngine` (one concrete implementation of `CorrectionEngine`) component for the `correct("uncorrected")` API call.

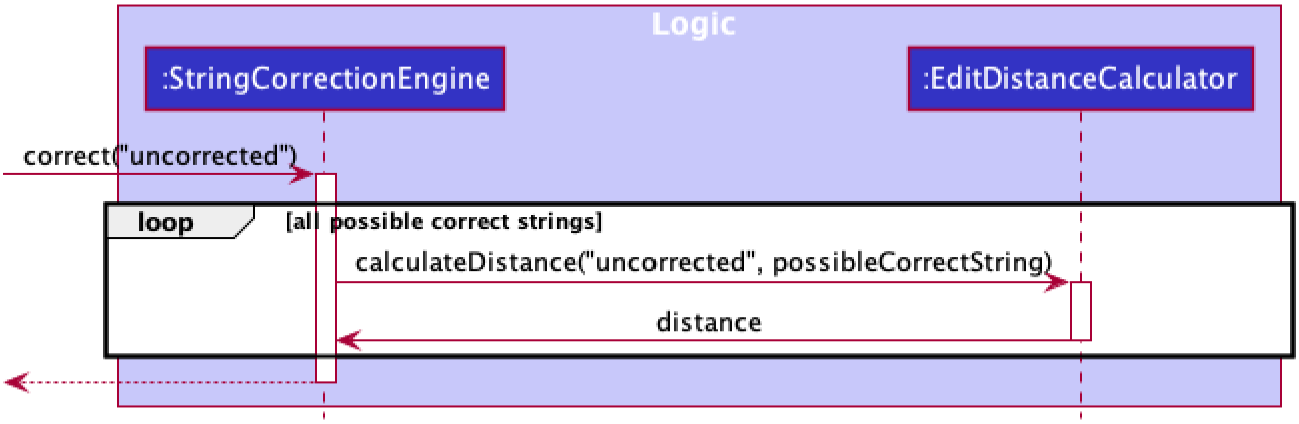

Logic
:StringCorrectionEngine
:EditDistanceCalculator
correct("uncorrected")
loop
[all possible correct strings]
calculateDistance("uncorrected", possibleCorrectString)
distance

Figure 10. Interactions inside the StringCorrectionEngine component for the `correct("uncorrected")` *call*

2.4. Model component

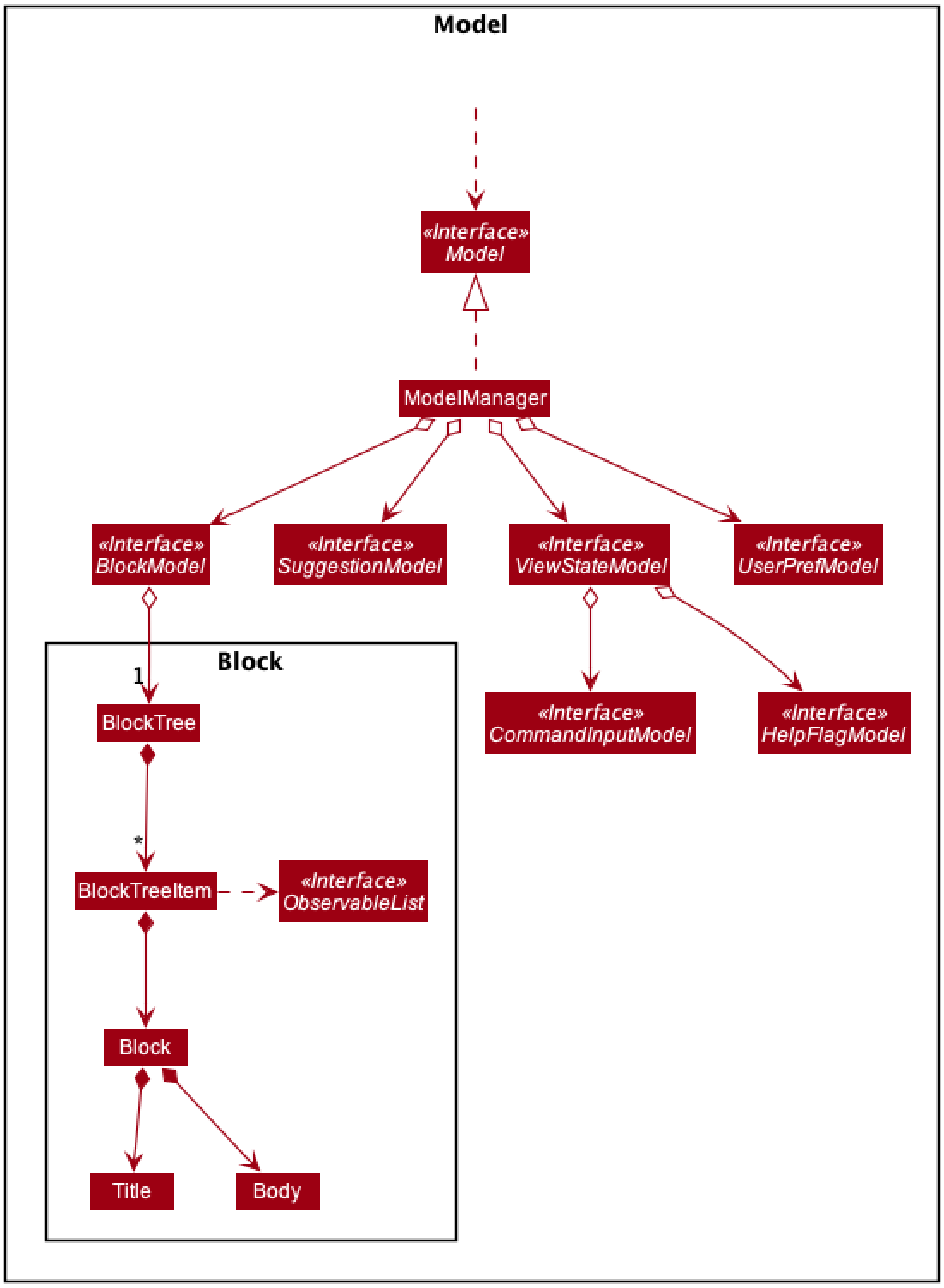


Figure 11. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores and manipulates the `BlockTree` data that represents a tree of Blocks, through BlockModel
- stores and manipulates a list of suggestions based on the user's input, through SuggestionModel
- stores the current state of the `View`, through ViewStateModel
    - stores the command input given by the user, through CommandInputModel

    - stores the state of the `help` modal being open, through HelpFlagModel

- stores `UserPref` data that represents the user's preferences, through UserPrefModel

2.5. Storage component

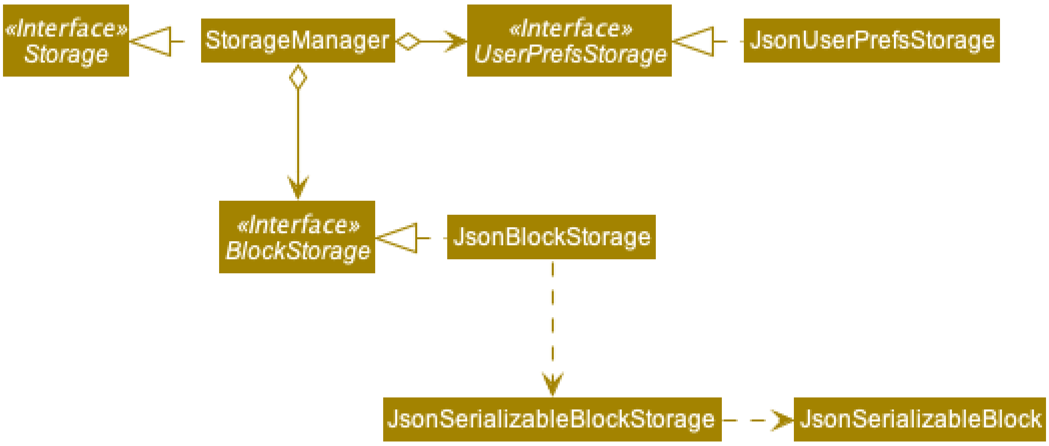


Figure 12. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in JSON format and read it back.
- can save the Block data in JSON format and read it back.

2.6. Common classes

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

2.7. 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 2.8, "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

2.8. 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. Implementation

This section describes the details on how features are implemented.

3.1. Correction Engine

3.1.1. Rationale

`CorrectionEngine` is needed to enable auto-correction of user inputs, to deliver as good typing experience as possible.

3.1.2. Current implementation

`CorrectionEngine` revolves around two *API* s, namely:

- The `CorrectionEngine` interface, implemented by `StringCorrectionEngine` and `AbsolutePathCorrectionEngine`. Concrete implementations of `CorrectionEngine` are employed to correct an uncorrected user input.
- The `EditDistanceCalculator` interface, implemented by `LevenshteinDistanceCalculator`. Concrete implementations of `EditDistanceCalculator` are employed to calculate the edit distance between two strings.

Two concrete implementations of the `CorrectionEngine` interface are, namely:

- The `StringCorrectionEngine` class, which deals with the correction of plain strings.
- The `AbsolutePathCorrectionEngine` class, which deals with the correction of absolute paths. The absolute paths here refer to the address of the notes (or blocks, as we call it) that exist in the App.

3.1.3. Design considerations

1. `CorrectionEngine` is built as a standalone module that can be used by both **`SuggestionEngine`** and **`Parser`**. This decision is made so that code duplication in relation to auto-correction is minimal.
2. Both `CorrectionEngine` and `EditDistanceCalculator` are implemented as interfaces, in an attempt to make the design of the `CorrectionEngine` component resilient to change. This design enables us to leverage on the strategy pattern to make our `CorrectionEngine` component more future-proof.

3.2. Suggestion Engine

3.2.1. Rationale

`SuggestionEngine` allows the users to traverse their notes conveniently, without having to remember the hierarchical structure of their notes. `SuggestionEngine` gives users the meaning of the command they input and a list of notes suggestions that they want to open, delete, or search.

3.2.2. Current implementation

	SuggestionArgHandler	**SuggestionHandler**
Purpose	Handles the arguments part of the user input and updates the `responseText` in the `Model` according to the user's command input	Updates the `responseText` in the `Model` according to the user's command input
Commands	`open`, `delete`, `search`, `new`	`edit`, `exit`, `help`
Suggestion Generation	Yes, by `SuggestionGenerator` (except for `new` command)	No

1. `Logic` uses the `SuggestionEngine` class, to handle the user input.
2. According to the command the user inputs, `SuggestionEngine` will create a `XYZSuggestionArgHandler` or `ABCSuggestionHandler` object which implements `SuggestionArgHandler` and `SuggestionHandler` interface respectively. `XYZSuggestionArgHandler` are for commands that require argument parsing, i.e. `open`, `delete`, `search`, `new`, whereas `ABCSuggestionHandler` are for commands that do not require argument parsing, i.e. `edit`, `exit`, `help`.
3. If `SuggestionArgHandler` object is created: the `responseText` in the `Model` will be updated. This case will also result in the creation of `XYZSuggestionGenerator` object (except for `new` command) which implements `SuggestionGenerator` interface. `XYZSuggestionGenerator` is then executed by the `SuggestionEngine`.
4. If `SuggestionHandler` object is created: the `responseText` in the `Model` will be updated.
5. The `Model` could be affected in 2 ways:
   - Update `responseText` of the `Model` (by the `SuggestionHandler` and `SuggestionArgHandler`): for instance, the input `open /` will set the `responseText` in the `Model` as "Open a note".
   - Store a list of `SuggestionItem` in the `Model` (by the `SuggestionGenerator`).
6. The UI will then be able to retrieve the `responseText` and list of `SuggestionItem` from the `Model` to be displayed to the user.

3.2.3. Design considerations

Aspect 1: Design with respect to the whole architecture

1. `SuggestionEngine` is segregated from `Parser` in order to differentiate the logic when the user has finished typing and pressed kbd:[Enter] (which will be handled by `Parser`) in contrast to when the user presses the keyboard kbd:[down] button and kbd:[Enter] to take in the suggestion item.
2. In order to keep the App's data flow unidirectional, `SuggestionEngine` will update the `responseText` (which tells the user the meaning of his command) and the list of `SuggestionItem` into the `Model`. Thus, by not showing the `responseText` and suggestions immediately to the UI, `SuggestionEngine` will not interfere with the `View` functionality.
3. `SuggestionArgHandler`, `SuggestionHandler`, `SuggestionGenerator`, `SuggestionItem`, and `SuggestionModel` are implemented as interfaces, in an attempt to make the design of the `SuggestionEngine` component resilient to change.

Aspect 2: Implementation of suggestions generation

- **Alternative 1:** Have a `SuggestionCommandParser` interface and `SuggestionCommand` interface to parse each of the command, update `responseText` in the `Model`, and give suggestions.
    - Pros: This provides a consistency for all the commands, where each command has a `XYZSuggestionCommandParser` and `XYZSuggestionCommand` class.
    - Cons: The `SuggestionCommandParsers` of the commands that do not require parsing of user input (`edit`, `exit`, `help`) end up passing a `userInput` argument that is not being used anywhere, which makes this design unintuitive. Moreover, since the updating of the `responseText` in the `Model` can be done in each `SuggestionCommandParser`, the ``SuggestionCommand`s of `edit``, `exit`, and `help` end up to be redundant.
- **Alternative 2 (current choice):** Create 2 separate interface to handle commands with input parsing and those without, and name it as a `SuggestionArgHandler` and `SuggestionHandler` respectively.
    - Pros: This solves the cons discussed in Alternative 1, as this design gives a separate implementation for the commands with input parsing and those without. It does not force the `Handler` to parse the user input when there is no need to. The naming `Handler` also does not restrict the functionality of the interface and classes to just parse an input, but allows for a flexibility in executing other functionality such as updating the `responseText` in the `Model`.

3.3. Paths

Given below is the implementation detail of the Path feature and some alternative design considerations.

3.3.1. Current Implementation

The `Path` interface represents the directory of a `Block` in our data structure. A path can exist in 2 forms namely :

1. AbsolutePath

2. RelativePath

An AbsolutePath is a path that takes its reference from the root `/` block.
While a RelativePath takes it reference from the current directory that is opened.

Currently the user is given the freedom to provide any of the 2 forms when using the `open`, `delete` command.
Given the following DataStructure below.

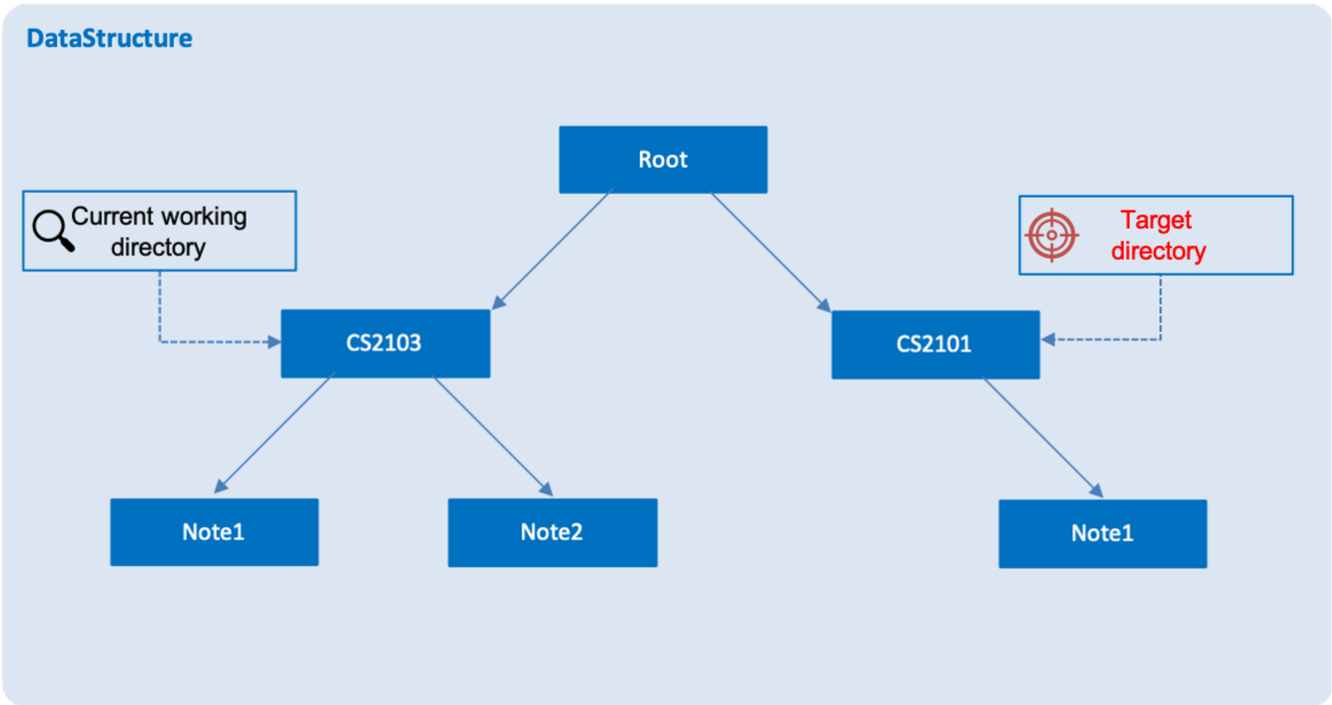


Figure 13. DataStructure example to illustrate Path

Using `AbsolutePath` `open /CS2101` and using `RelativePath` `open ../CS2101` would yield the same result Design Consideration.

3.3.2. Design Consideration

Aspect: Implementation of `Path` :

- Alternative 1(Current choice): Have 2 separate class implementing `Path`, which is `AbsolutePath` and `RelativePath`.
  - Pros: More readable and OOP, each class can have their individual validity REGEX.
- Alternative 2: Implement a single class `PathImpl` and have a boolean flag `isAbsolute` to tell if its a Relative or Absolute path.

Aspect: Logical equivalence of `RelativePath` :

- Alternative 1(Current choice): Relative path `CS2103/../note1` would be equivalent to `note1`. This was deem to be
  - Pros: More readable and OOP, each class can have their individual validity REGEX.
- Alternative 2: Relative path `CS2103/../note1` would not be logically equivalent to `note1`.

3.4. Tree Data Structure

Notably aims to provide end user a neat and well-organized workspace to store their notes. This is done by creating a tree structure; allowing users to create folder-like paths to organize their notes and group them into categories to their own liking.

3.4.1. Rationale

While this can be done with a linear data structure (a simple list), a linear list of notes would require more work to establish the relationship between groups of notes. A tree data structure supports this better, giving a clearer distinction while also establishing a form of hierarchy (as seen in the design example below).

On top of that, observability must be ensured so that the UI can update with any changes that happen on the tree (and its nodes) and also the data within each node.

3.4.2. Current Implementation

A custom tree data structure that supports observability has been implemented. The tree (referred to as `BlockTree`) is made up of tree nodes (referred to as `BlockTreeItem`). The tree is observable such that if any change occurs on any of the tree's nodes, the change event will bubble upwards to the root node. Hence, the root node serves as the entry point for the `BlockTree`.

Each BlockTreeItem contains 3 primary components:

- a reference to its parent
- an ObservableList of its children
- User's note data (referred to as `Block` data) consisting of:
    - `Title` of the note
    - `Body` content of the note (optional)

When manipulating the `BlockTree`, the execution of any operation is always split in this order:

1. Navigate to the specfied path
2. Open the block at the specified path
3. Execute the operation on the block that is currently open

3.4.3. Design Considerations

Aspect: `BlockTreeItem` vs Folders to represent path structure

Current choice: `BlockTreeItem` Pros: No need for an additional class. Having a separate `folder` object would also require a separate UI View since folders should not contain any block data. Cons: Somewhat unconventional design. User might be unfamiliar with the intention on first use, without proper explanation

Aspect: Root should also be a `BlockTreeItem`

Pros: Seamless transition to JSON storage Cons: Need to add constraint to ensure that the root `BlockTreeItem` does not contain any `Body` and is also unmodifiable

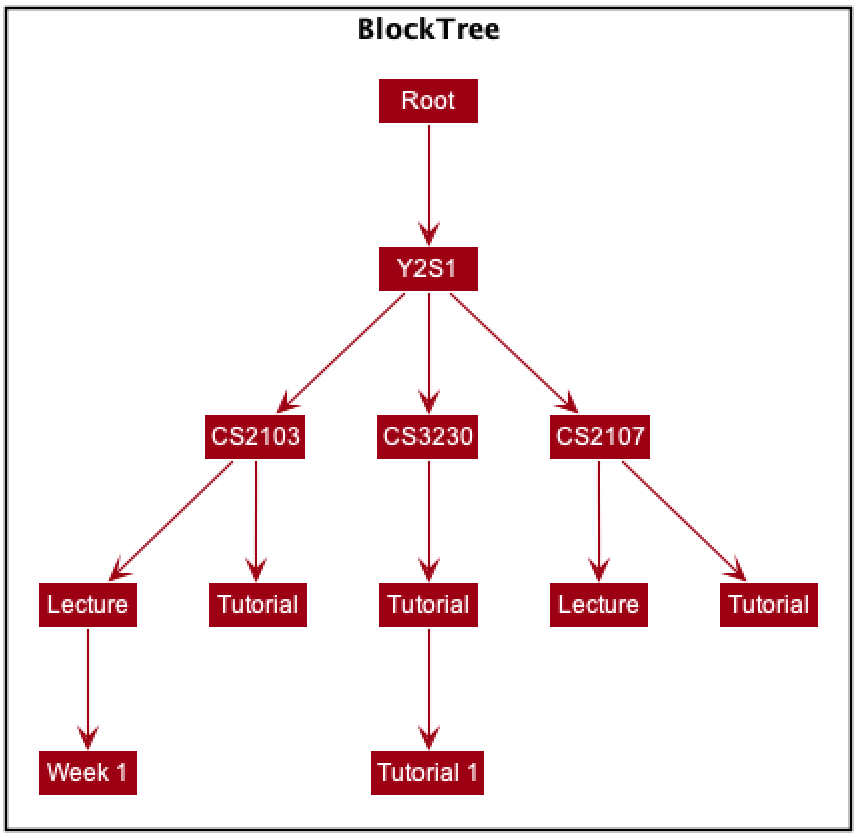


Figure 14. Tree Data Structure Design Example

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:

- Students that has a need to take notes and organize them into categories

- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: Take and manage notes faster than a typical mouse/GUI driven app

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…
* * *	student	traverse my notes in a file system-like manner	so that I can skim through my sea of notes and drafts without any problem.
* * *	student	search my notes by their content	I won’t have to remember the exact titles I had given my notes.
* * *	impatient student	alias a path to a folder	do not have to memorise and type out the entire file structure when accessing a nested note
* *	student	can view the relevant search results	so that I don’t need to worry about remembering the exact location and title of notes
* *	student	reliably type search commands(not error-prone)	focus on searching my notes rather than ensuring my commands are exact

Priority	As a …	I want to …	So that I can…
*	student	export my notes into PDF documents	share/print my notes effortlessly.

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `Notably` and the **Actor** is the `user`, unless specified otherwise)

Use case: Search notes using the Auto-suggestion feature

MSS

1. User types in a keyword of a note’s content that he wants to open.
2. Notably lists out the relevant search results, with the most relevant at the top of the list (based on the keyword’s number of occurrences in the note).
3. User chooses one of the suggested notes.
4. Notably opens the chosen note.

   Use case ends.

Extensions

2a. No suggestion is being generated.

2a1. Notably displays a response text, indicating that the user is trying to search through all of the notes using that particular keyword.

2a2. Since the empty suggestion conveys that the keyword cannot be found, the user enters a new data.

Steps 2a1-2a2 are repeated until the data entered is correct. Use case resumes from Step 3.

Use case: Open/ Delete notes using the Auto-suggestion feature

MSS

1. User types in an incomplete path or title of a note.
2. Notably lists out suggestions of notes.

3. User chooses one of the suggested notes.
4. Notably opens/ deletes the chosen note.

   Use case ends.

Extensions

1a. Path or title contains invalid character(s) (symbols - or `)

1a1. Notably displays a response text, indicating that the path or title is invalid.

1a2. User enters a new data.

Steps 1a1-1a2 are repeated until the data entered is correct. Use case resumes from Step 2.

1b. Path or title does not exist

1b1. Notably displays a response text, indicating that the user is trying to open/ delete the note with the particular path or title that the user inputs.

1b2. Notably does not generate any suggestions, which means the note cannot be found.

1b3. User enters a new data.

Steps 1b1-1b3 are repeated until the data entered is correct. Use case resumes from Step 2.

===

{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 notes without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

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

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 … }

F.2. 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 … }