Lee Ee Jian - Project Portfolio

[github] Role: Developer, Analyst
Responsibilities: Code Quality

PROJECT: SecureIT

This document outlines my contributions to the project **SecureIT**.

Overview

With the increasing digitisation of all businesses, managing one's digital life has become highly complex. Having realised this problem, my group and I created **SecureIT** as part of our software engineering project.

SecureIT is designed to be an all-in-one management tool for all confidential information. It targets workers in startups, and boasts management capabilities for:

- Passwords | Secret Files | Secret Notes | Credit Cards

SecureIT operates via a Command-Line Interface(CLI), and executes accordingly to the user inputted commands.

Summary of contributions

This section acts as a summary of my contributions to SecureIT.

- **Major enhancement: the ability to analyse the level of security of the users' passwords.**
  - What it does: The password analysis feature provides the ability to evaluate and subsequently view details about the security level of each of the user's passwords.
  - Justification: This feature greatly improves the utility of SecureIT because the various analysis tests allows the user to identify different weaknesses in their password and thereby make the necessary corrections to ensure the security of each account.
  - Highlights: It was highly challenging to implement the analysis feature. Some of the key considerations include:
    - Design: The design of the entire analysis component follows that of a **Command Pattern**. This greatly increases the extensibility of the analysis feature to accommodate even more types of analysis tests in the future.
    - Data structure and algorithms: The implementation of the various analysis tests requires the use of more complex concepts such as recursion and multiple pointers. The implementation also required knowledge on the use of more complex data structures such as TreeMaps and HashMaps in order to help boost the efficiency of the product.
  - Credits: [Levenshten-Distance algorithm]
- **Major enhancement: the ability to generate random secure passwords.**
  - What it does: The password generation feature provides the ability to generate customisable, and truly random passwords.
  - Justification: This feature greatly improves the utility of SecureIT because it alleviates the user of needing to think of a password whenever he needs to create a new account. More importantly, the password provided is guaranteed to be random and thereby secure.
  - Highlights: It was challenging to implement truly random password generation. In-depth research was required on how best to generate truly random passwords. The generation of passwords leverages on the `java.security.SecureRandom` API for a cryptographically strong random number generator (RNG) to make the password generation as random as possible.
- **Code contributed:** [Reposense]
- **Other contributions:**
  - Team contributions:
    - Implemented the method to copy text into the user's local clipboard, which was subsequently used by my teammates in the copy features they implemented. (examples: 1, 2, 3)
    - Reported bugs and offered suggestions for teammates. (examples: #189, #190)
  - Project management:
    - Helped create and assign some issues for each milestone in the lead up to the project submission. (examples: #29, #28, #26)

- Documentation
  - Made cosmetic improvements to the existing User Guide by creating the project logo.

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Generating a new password: `generate`

Lazy to think of a strong password?

Simply enter `generate` into the command box to get one. It's random, secure and totally hassle-free!

Try it out!

- Want to customise your password? Don't worry, we've got that covered too. You can customise the following fields:
  - Length of password (Between 4 and 25)
  - Exclusion of lower alphabets | upper alphabets | numbers | special characters
- Refer to usage format below for more details.

NOTE	You are only required to indicate for fields you wish to **exclude**!

NOTE	Please ensure **at least one character set is included**.

Format: `generate [length/LENGTH] [lower/FALSE] [upper/FALSE] [num/FALSE] [special/FALSE]`

Example: `generate` | `generate length/10` | `generate lower/false special/false`

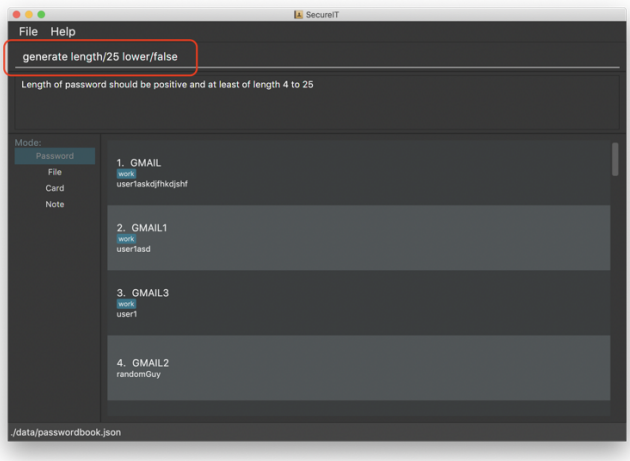

Figure 1. Example user input (left) and resultant output (right) for generate.

Analysing passwords : `analyse`

Curious to know how secure your passwords really are? Type `analyse` into the command box to find out. You might be surprised… >>>>>>> 57900bceda68ce2fd8fcf27979b3172ea508be0e

Try it out!

- To view the detailed analysis of a particular password, simply add in the `strong` prefix with the `INDEX` of the password.

NOTE	`INDEX` used should be that of an existing password.

- Refer to usage format below for more details.

Format: `analyse [strong/INDEX]`

Example: `analyse` | `analyse strong/8`

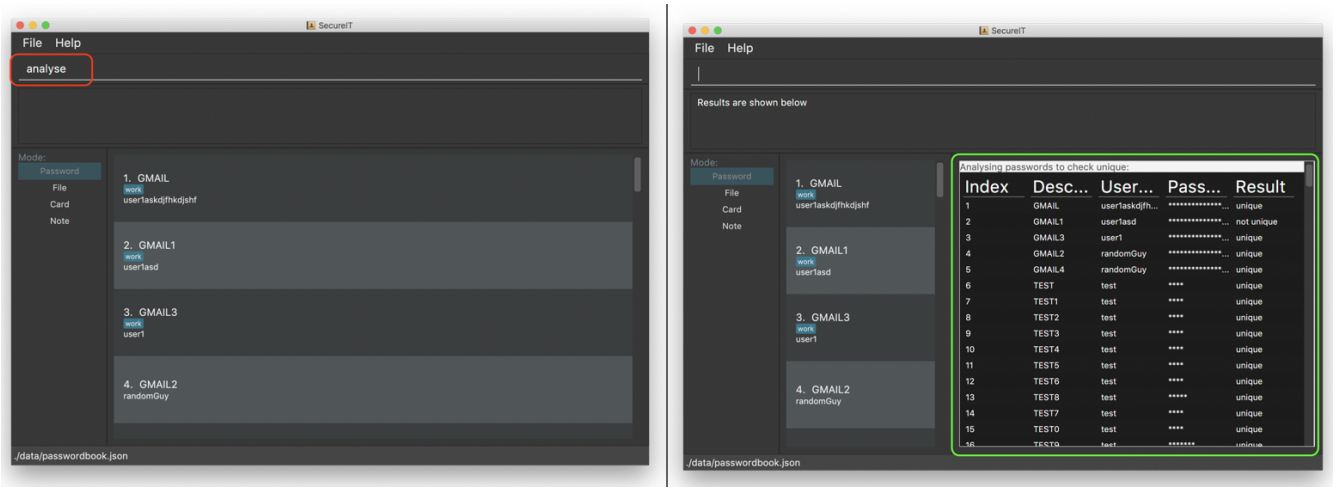

Figure 2. Example user input (left) and resultant output (right) for analyse.

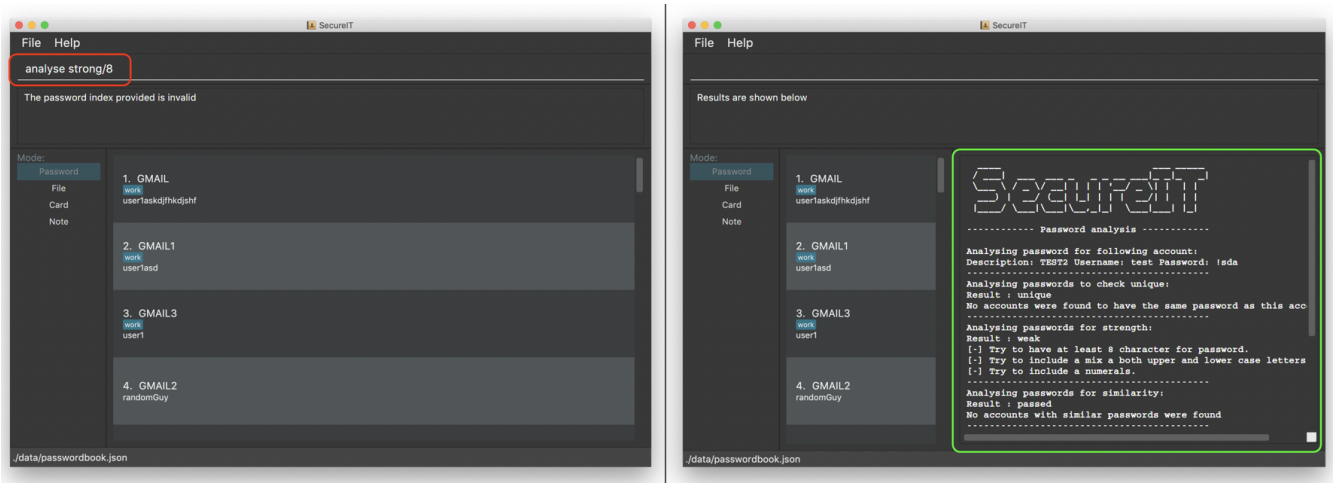

Figure 3. Example user input (left) and resultant output (right) for analyse strong.

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Generate Password

This section provides implementation details on the password generation feature.

Overview

The purpose of password generation is to provide users with a hassle-free way to generate random, secure passwords.

By default, the password generation feature will generate a random password that:

- is 10 letters long
- contains lower case alphabets, upper case alphabets, numbers, and special characters.

However, users can alternatively choose to customise any of the aforementioned fields as per their needs through their input.

The generation of truly random passwords is made possible through the use of the `java.security.SecureRandom` API. It provides a cryptographically strong random number generator (RNG) that will be used in the password generation.

Implementation

The following activity diagram summarises the general steps taken during password generation:

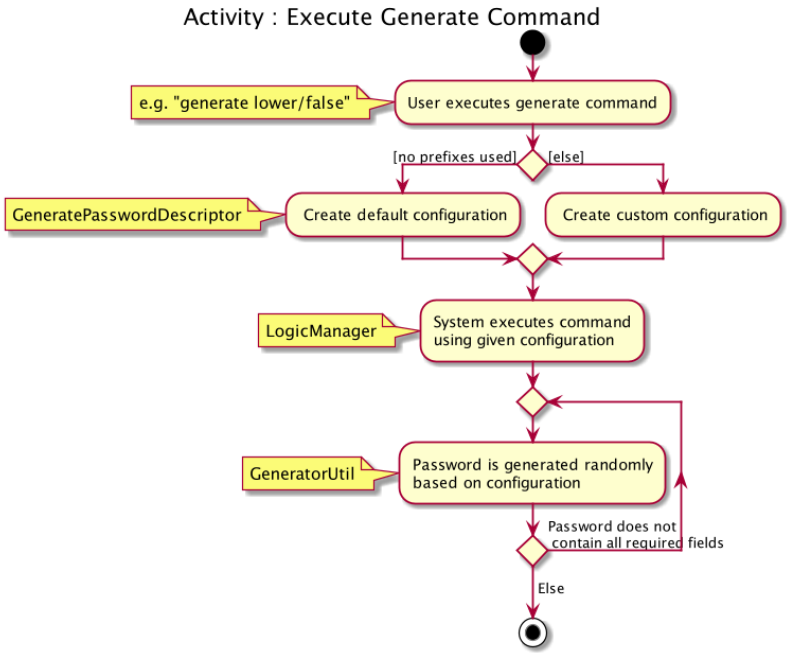

Figure 4. Summary flow of actions during password generation.

The steps below outline explicity the generation of passwords :

- The user input is `#parse()` by the `GeneratePasswordCommandParser`.
- Based on the user input, a `GeneratePasswordDescriptor` is instantiated.
  - If the user input includes optional prefixes to customise configuration, we modify the attributes of the `GeneratePasswordDescriptor` through various setter methods.
  - Else, the default `GeneratePasswordDescriptor` is instantiated through the static method `GeneratePasswordDescriptor#getDefaultConfiguration()`.

NOTE	The `GeneratePasswordDescriptor` is an object that encapsulates the settings of the **configuration** used for password generation.

- The `GeneratePasswordCommand` is then instantiated with the given configuration.
- Upon `GeneratePasswordCommand#execute()`, the static method `GeneratorUtil#generateRandomPassword()` is invoked:
  - Based on the configuration, the relevant character sets (lower-case alphabets, upper case alphabets, numerals, special characters) are added into a list.
  - The `GeneratorUtil` class uses the method `java.security.SecureRandom.nextInt()` to choose a random characters set, followed by a random character within that set to be included in the new password.
  - This process of choosing a character to include in the password is repeated for the length of the password.
- The generated password is then checked to see if it includes all the user requirements (ie. whether it includes all the character sets specified by the user.)
- This process of generating a password is repeated until the user requirements are met.

The activity diagram below depicts the way in which a password is generated by the method `GeneratorUtil#generateRandomPassword()`:

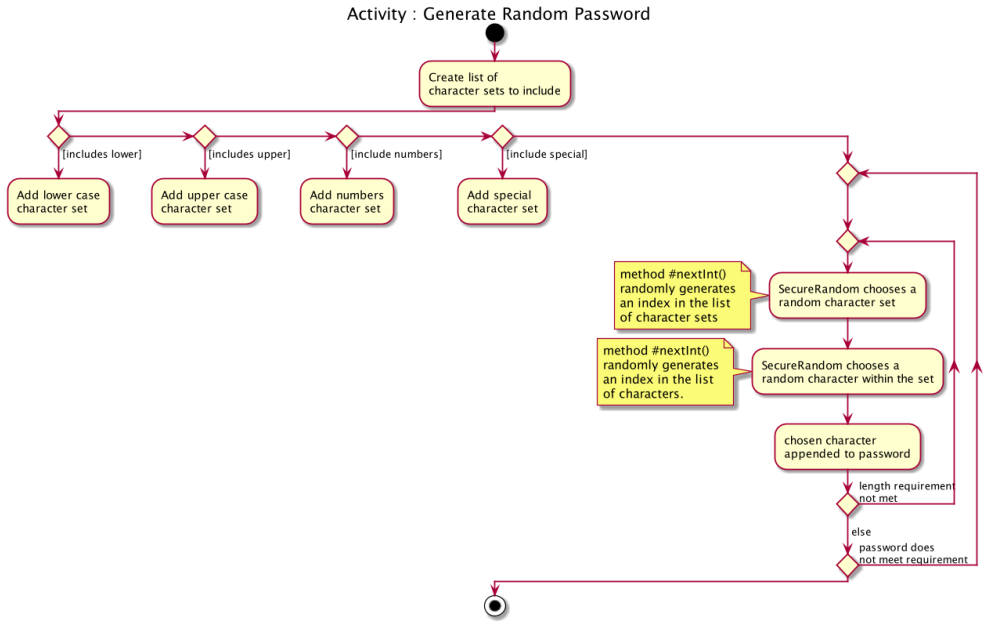

Figure 5. Summary flow of actions during the invokation of #generateRandomassword().

Design considerations

These are the considerations we came across when implementing the generate feature:

Aspect: How to generate a random password

Alternative 1 (Current choice)	**Alternative 2** How to generate a truly random password
Randomly choose a character set, followed by a character within the set to include in password: *Pros:* Randomness is achieved because there is no predictability in the way a character set, or character is chosen. *Cons:* Efficiency is compromised. Generating a password this way may produce a password that does not meet user configuration (e.g. random password does not include special characters, although it was supposed to). As such, program needs to keep producing new random password until all the user configuration are met.	**Ensuring all user configurations are met by hard coding a pattern in the way character sets are included** *Pros:* Efficiency is achieved. We are guaranteed to generate a password that matches users' configuration everytime (e.g. lower case → upper case → numeral → special case patter) *Cons:* Randomness is compromised. Generating passwords in such a pattern makes the password style very predictable, hence compromising randomness and security.
Why did we choose Alternative 1: Although less time efficient, the password generated is more random, and thus more secure. Generating a secure password is inline with our use case, and so is a more important factor.	

Analyse Password

This section provides implementation details on the analyse password feature.

Overview

The purpose of analysing passwords is to provide users with information on the level of security of their passwords.

The user can either :

1. View a summary table of results of all password by inputting `"analyse"` OR
2. View a detailed review of a specific password by inputting `"analyse strong/INDEX"`

SecureIT analyses 6 core components of every `Password`, using various `Analysers`. The following table summarises the functionality of each `Analyser`:

Analyser:	**Purpose:**
UniqueAnalyser	Checks that every password is unique.

StrengthAnalyser	Checks the complexity of every password.
SimilarityAnalyser	Checks that no two accounts share a password that is at least 70% similar.
DictionaryAnalyser	Checks that password does not contain any commonly-used passwords. (e.g. "password")
SequenceAnalyser	Checks that password does not contain any commonly-used sequences. (e.g. "ABC")
KeyboardAnalyser	Checks that password does not contain any commonly-used keyboard patterns. (e.g. "qwerty")

The following class diagram is the current structure of the `Analyser` component.

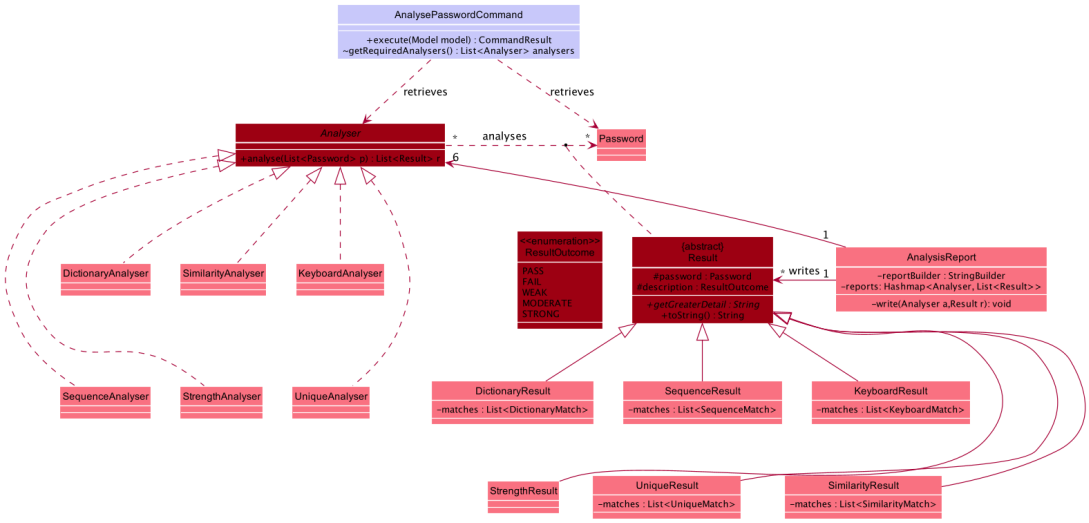

Figure 6. Class diagram depicting the structure of the Analyser component.

In summary,

1. The analysis of passwords is initiated upon `#execute()` of the `AnalysePasswordCommand`.
2. Each `Analyser` analyses every password in the password book.
3. The analysis of each password produces the respective `Result` for that password.
4. These `Results` are compiled and written into a `AnalysisReport`, which is then shown to the user in the form of a summary table.
5. Users can opt to view a detailed report for a specific password to get more information on the respective `Result`.

Implementation

On `#execute()`, `AnalysePasswordCommand` retrieves the current list of passwords via `Model#getFilteredPasswordList()`. It also retrieves the required analysers via `#getRequiredAnalysers()`.

Each `Analyser` has it's own implementation of `#analyse()`, which will subsequently be invoked by the `AnalysePasswordCommand` given the current list of passwords.

The following sequence diagram breaks down the general flows of events stated above, and in the context of a `DictionaryAnalyser`:

> **NOTE** For the following sequence diagram, the other `Analysers` are instantiated in a similar fashion in `#getRequiredAnalsyers()` and hence omitted to the make the diagram less congested.

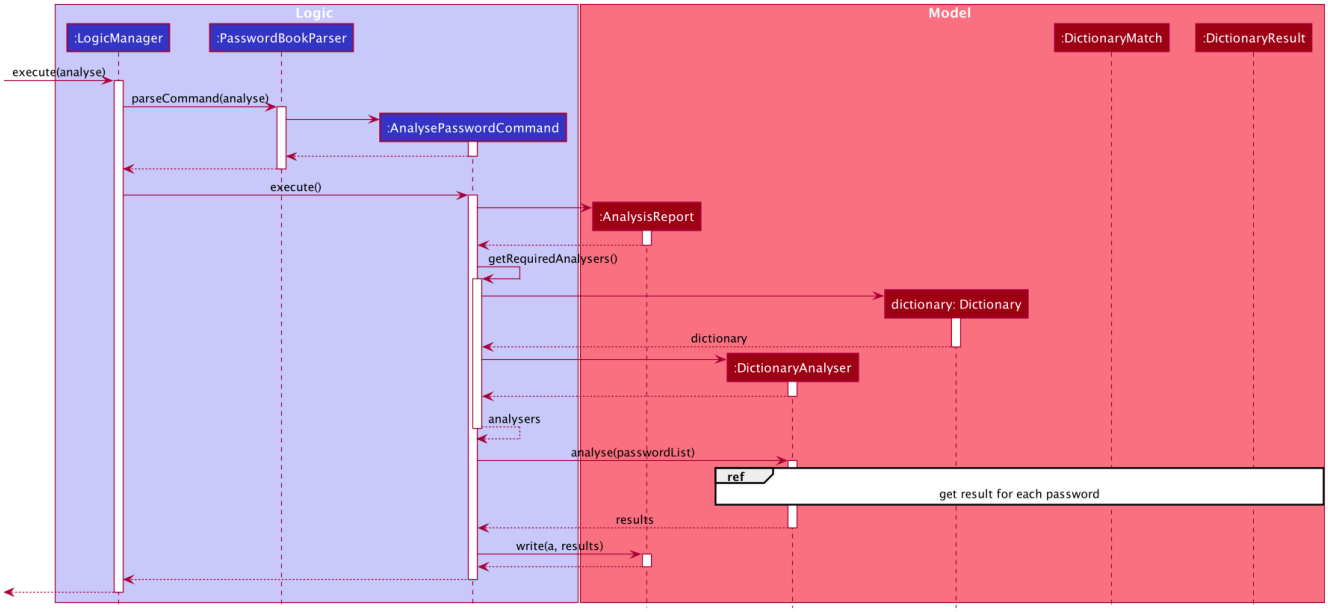

Figure 7. Sequence diagram depicting the flow of analysing the list of passwords, in the context of DictionaryAnalyser.

In the following discussion, we will be reviewing how `Results` are produced for each password, in the context of the `DictionaryAnalyser`.

In the case of the `DictionaryAnalyser#analyse()`, every password is checked to see if it contains any subsequence that is listed as a commonly used password in the instantiated `Dictionary`. This is done in the internal method `#getAllMatches()` within `#analyse()`.

> **NOTE** The `Dictionary` is an object that maps commonly used passwords to their ranking in terms of how commonly used they are. (e.g. "123456" is mapped to rank 1 in the Dictionary because it is the most commonly used password)

If there exists a subsequence that tests positive when looked up against the `Dictionary` , a `DictionaryMatch` is created to note down the details of the subsequence.

As long as a `DictionaryMatch` is found, then the password has failed the analysis. A `DictionaryResult` with the attribute `ResultOutcome.FAIL` will be created for that password. This process is repeated for every password. Following, the list of `DictionaryResult` are returned to be compiled and written by the `AnalysisReport`.

It is also worth mentioning that `DictionaryAnalyser` is capable of identifying commonly used passwords even in leet-ed variations (e.g. p@5sw0rd).

This is made possible because of the method `#LeetUtil.translateLeet()`, which is a recursive algorithm designed to return all possible un-leet variations, given a leet-ed password.

The following sequence diagram depicts the flow of events mentioned above:

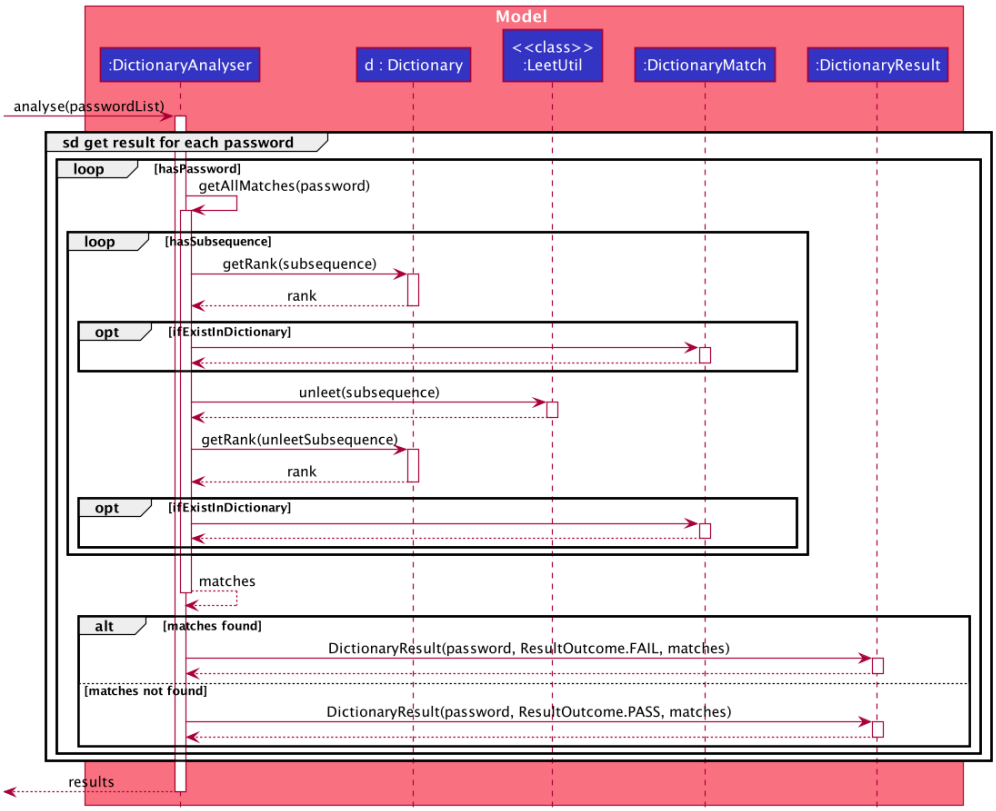

Figure 8. Sequence diagram depicting the flow of getting all Match objects for a given password.

Design Considerations

Aspect: How analyse/analyse strong executes

Alternative 1 (Current choice)	Alternative 2
Always analyse the entire list of Password objects for every "analyse" command, even if the list of Passwords was unchanged. *Pros:* Easy to implement, not required to check state if the current list of Passwords has been modified. *Cons:* May have performance issues in terms of speed of the programme.	**Save in memory the result produced by the Analyser objects, and update result upon modification of list of Passwords.** *Pros:* Performance of programme will be a lot faster and efficient. *Cons:* Hard to implement. Have to keep track of state of the list of Password objects and check if the list has been modified from the last time they were analysed.
Why did we choose Alternative 1: From a more practical point of view, users are not expected to analyse their passwords on a very regular basis, so it may be inefficient use of memory to constantly save the results. Also, considering the fact that each password is capped at a length of 25, time performance will not be affected significantly when analysing each password.	