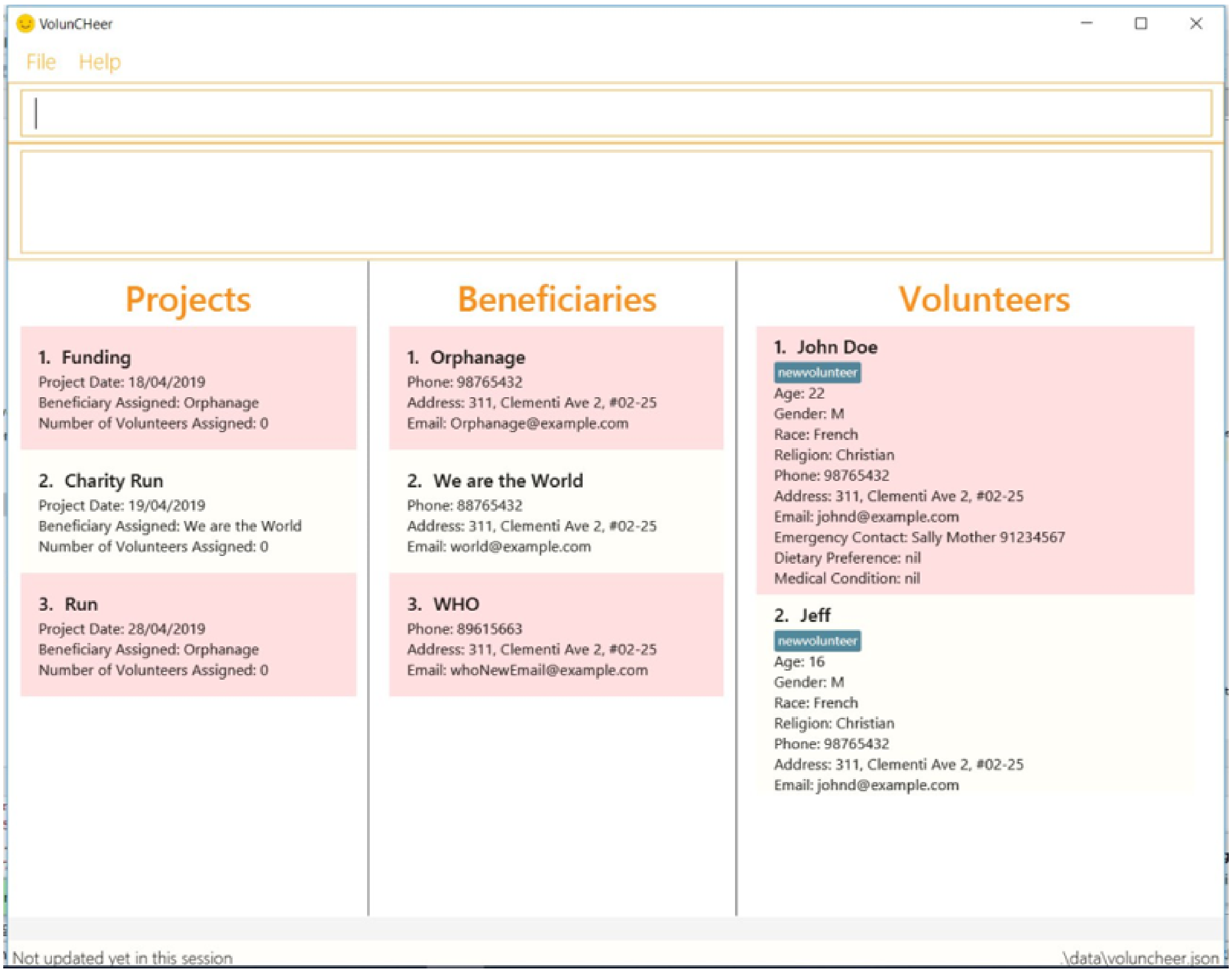
VolunCHeer - Developer Guide

By: CS2113T T08-01 Since: Feb 2019 Licence: MIT

1. Introduction

1.1. About VolunCHeer

VolunCHeer is an open-sourced Command Line Interface (CLI) management application designed for Volunteering Project Managers. We aim to help our target users alleviate the haystack of managing multiple projects, beneficiaries, and volunteers in an efficient and effective manner.



Projects

1. Funding
Project Date: 18/04/2019
Beneficiary Assigned: Orphanage
Number of Volunteers Assigned: 0

2. Charity Run
Project Date: 19/04/2019
Beneficiary Assigned: We are the World
Number of Volunteers Assigned: 0

3. Run
Project Date: 28/04/2019
Beneficiary Assigned: Orphanage
Number of Volunteers Assigned: 0

Beneficiaries

1. Orphanage
Phone: 98765432
Address: 311, Clementi Ave 2, #02-25
Email: Orphanage@example.com

2. We are the World
Phone: 88765432
Address: 311, Clementi Ave 2, #02-25
Email: world@example.com

3. WHO
Phone: 89615663
Address: 311, Clementi Ave 2, #02-25
Email: whoNewEmail@example.com

Volunteers

1. John Doe
newvolunteer
Age: 22
Gender: M
Race: French
Religion: Christian
Phone: 98765432
Address: 311, Clementi Ave 2, #02-25
Email: johnd@example.com
Emergency Contact: Sally Mother 91234567
Dietary Preference: nil
Medical Condition: nil

2. Jeff
newvolunteer
Age: 16
Gender: M
Race: French
Religion: Christian
Phone: 98765432
Address: 311, Clementi Ave 2, #02-25
Email: johnd@example.com

Not updated yet in this session
.\data\voluncheer.json

1.2. How to Contribute

There are lots of ways to contribute to VolunCHeer: coding, testing or improving our build process and tools. This developer’s guide provides information that will help you get started as a VolunCHeer contributor. Even if you are an experienced VolunCHeer developer, you will still find this guide to be useful to refer to.

If you are ready to contribute, simply create a Pull Request (PR) on our main repository.

If you have found any bugs or have ideas to improve VolunCHeer, create an issue here or contact us directly.

1.3. How to use this Developer Guide

NOTE This is a note. A note contains considerations or notices when using VolunCHeer

WARNING This is a warning. Ensure not to violate these warnings when using VolunCHeer

TIP This is a tip. A tip gives you suggestion on how to use certain features in VolunCHeer or how you can contribute to VolunCHeer

2. Setting up

2.1. Prerequisites

1. **JDK** `9` or later

   WARNING JDK `10` on Windows will fail to run tests in headless mode due to a JavaFX bug. Windows developers are highly recommended to use JDK `9`.

2. **IntelliJ** IDE

   NOTE IntelliJ by default has Gradle and JavaFx plugins installed.
   Do not disable them. If you have disabled them, go to `File` > `Settings` > `Plugins` to re-enable them.

2.2. Setting up the project in your computer

1. Fork this repo, and clone the fork to your computer
2. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
   a. Click `Configure` > `Project Defaults` > `Project Structure`
   b. Click `New…` and find the directory of the JDK
4. Click `Import Project`
5. Locate the `build.gradle` file and select it. Click `OK`
6. Click `Open as Project`
7. Click `OK` to accept the default settings
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.

9. Open `MainWindow.java` and check for any code errors
    a. Due to an ongoing issue with some of the newer versions of IntelliJ, code errors may be detected even if the project can be built and run successfully
    b. To resolve this, place your cursor over any of the code section highlighted in red. Press kbd:[ALT + ENTER], and select `Add '--add-modules=…' to module compiler options` for each error
10. Repeat this for the test folder as well (e.g. check `HelpWindowTest.java` for code errors, and if so, resolve it the same way)

2.3. Verifying the setup

1. Run the `seedu.voluncheer.MainApp` and try a few commands
2. Run the tests to ensure they all pass.

2.4. Configurations to do before writing code

2.4.1. Configuring the coding style

This project follows oss-generic coding standards. IntelliJ's default style is mostly compliant with ours but it uses a different import order from ours. To rectify,

1. Go to `File` > `Settings…` (Windows/Linux), or `IntelliJ IDEA` > `Preferences…` (macOS)
2. Select `Editor` > `Code Style` > `Java`
3. Click on the `Imports` tab to set the order
    - For `Class count to use import with '*'` and `Names count to use static import with '*'`: Set to `999` to prevent IntelliJ from contracting the import statements
    - For `Import Layout`: The order is `import static all other imports`, `import java.*`, `import javax.*`, `import org.*`, `import com.*`, `import all other imports`. Add a `<blank line>` between each `import`

Optionally, you can follow the UsingCheckstyle.adoc document to configure Intellij to check style-compliance as you write code.

3. Design

3.1. Architecture

[Architecture] | *Architecture.png*

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.

TIP The `.pptx` files used to create diagrams in this document can be found in the diagrams folder. To update a diagram, modify the diagram in the pptx file, select the objects of the diagram, and choose `Save as picture`.

`Main` has only one class called `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.

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.

[LogicClassDiagram] | *LogicClassDiagram.png*

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 `deleteVolunteer 1`.

[SDforDeletePerson] | *SDforDeletePerson.png*

Figure 3. Component interactions for `deleteVolunteer 1` command

The sections below give more details of each component.

3.2. UI component

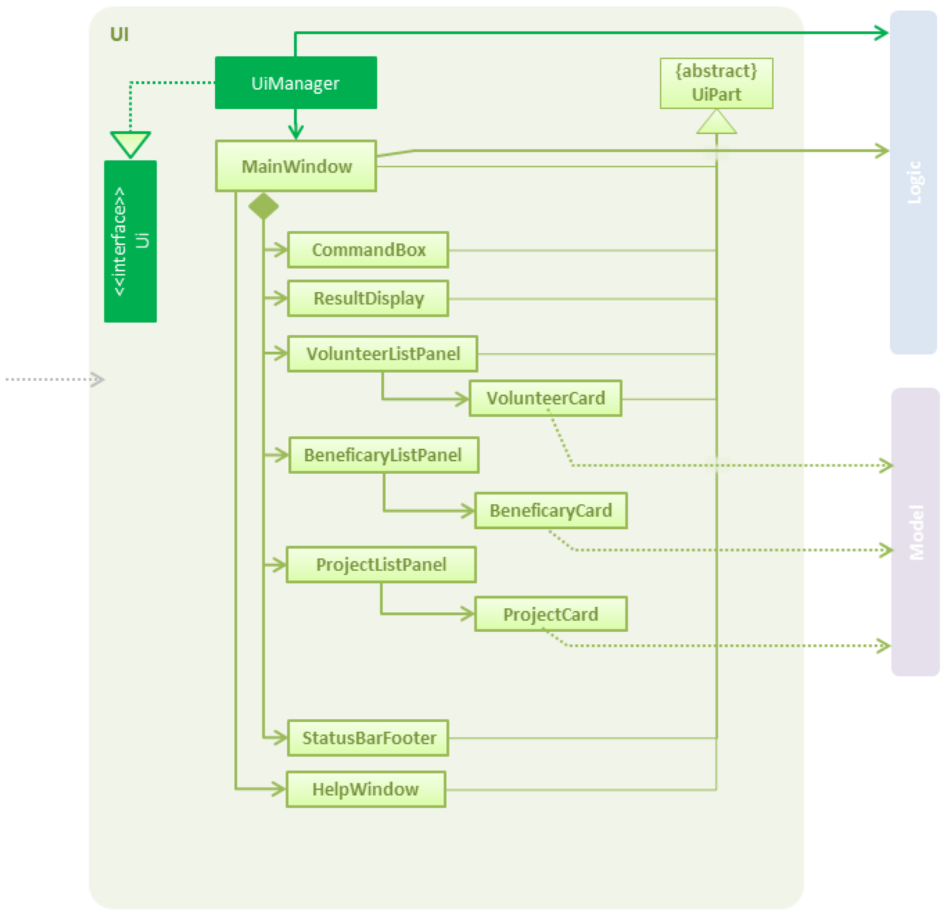


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `StatusBarFooter`, `BrowserPanel` 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` 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.

3.3. Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 5. Structure of the Logic Component

API : `Logic.java`

1. Logic uses the VolunCHeerParser 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 volunteer).
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.

3.4. Model component

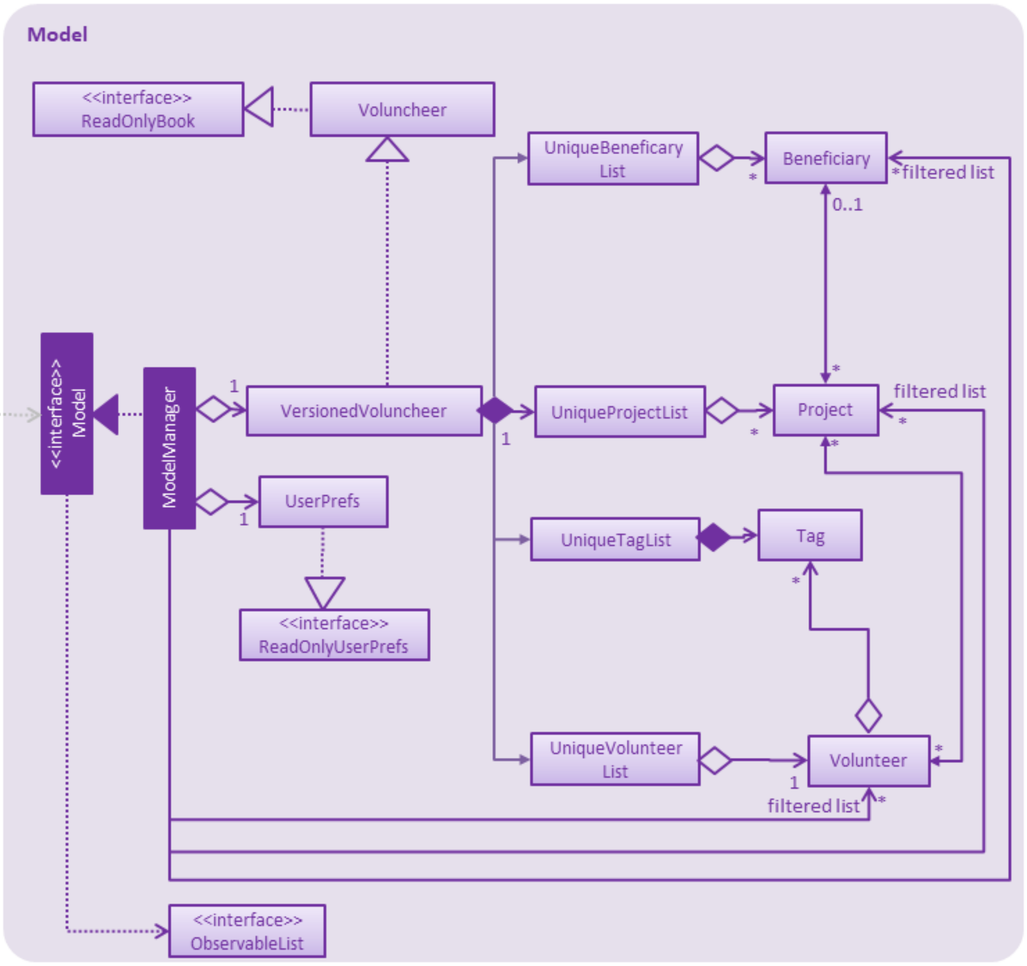


Figure 6. Structure of the Model Component

API : Model.java

The Model,

- stores a UserPref object that represents the user's preferences.
- stores the Volunteer Book, Beneficiary Book, Project Book data.
- manages the interaction and relationship between different objects (Volunteer, Beneficiary, Project)
- exposes an unmodifiable ObservableList<Object> that can be 'observed' (Object can be Vounteer,

Beneficiary, Project). 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.

3.5. Storage component

[StorageClassDiagram] | *StorageClassDiagram.png*

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 VolunCHeer Book data in json format and read it back.

3.6. Common classes

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

4. Implementation

4.1. Password Feature

4.1.1. Current Implementation

A password is required to grant access to the project manager at the start of the program. It serves as a security mechanism to protect the important data within.

At the start of the program, the UI is initialised by `UiManager`. A password window will pop out and prompt the user to enter the password.

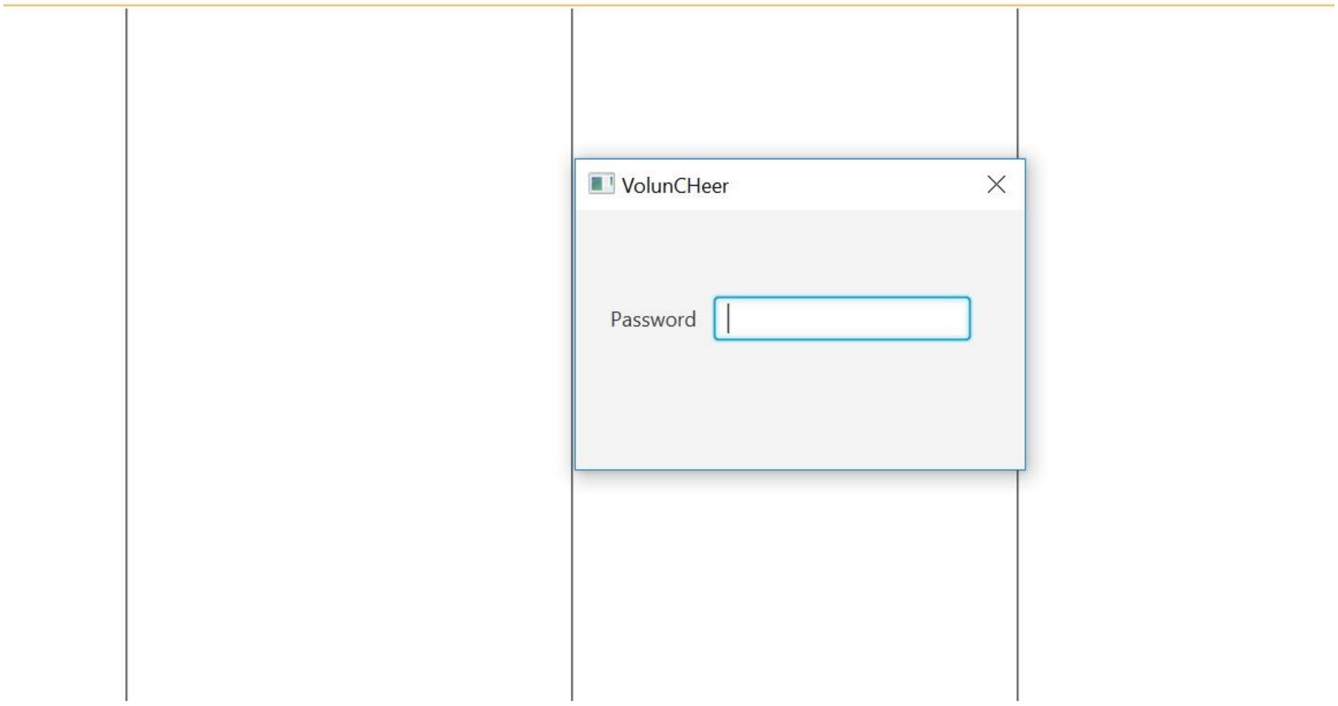
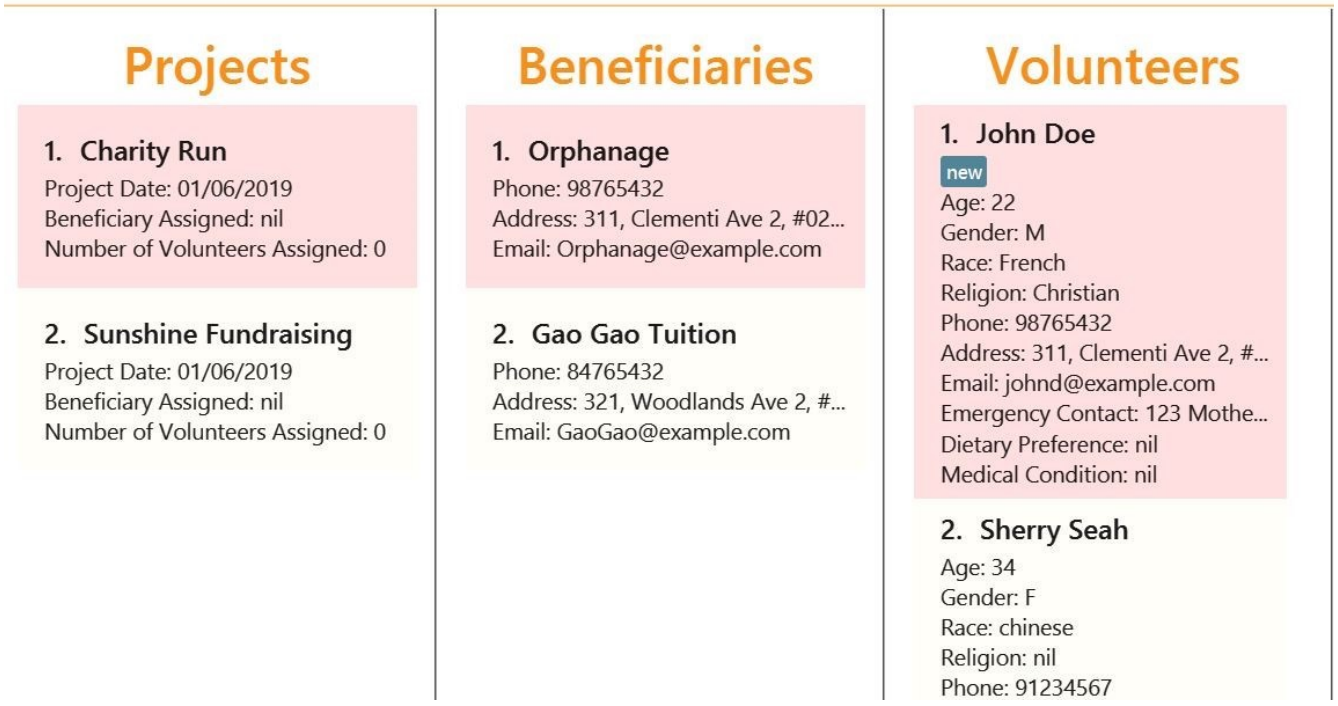


Figure 8. Program prompting user for password

The password feature uses `ValidatePassword` to check if the user input matches the password stored in the program. If the password matches, it will set boolean `user` to be true. `UiManager` will thus instantiate `fillInnerParts()`, displaying the hidden data to the project manager.



Projects

1. Charity Run
Project Date: 01/06/2019
Beneficiary Assigned: nil
Number of Volunteers Assigned: 0

2. Sunshine Fundraising
Project Date: 01/06/2019
Beneficiary Assigned: nil
Number of Volunteers Assigned: 0

Beneficiaries

1. Orphanage
Phone: 98765432
Address: 311, Clementi Ave 2, #02...
Email: Orphanage@example.com

2. Gao Gao Tuition
Phone: 84765432
Address: 321, Woodlands Ave 2, #...
Email: GaoGao@example.com

Volunteers

1. John Doe
new
Age: 22
Gender: M
Race: French
Religion: Christian
Phone: 98765432
Address: 311, Clementi Ave 2, #...
Email: johnd@example.com
Emergency Contact: 123 Mothe...
Dietary Preference: nil
Medical Condition: nil

2. Sherry Seah
Age: 34
Gender: F
Race: chinese
Religion: nil
Phone: 91234567

Figure 9. Data displayed to user upon entering the correct password

If the password is entered incorrectly for five times, the program will terminate itself.

A sequence diagram is shown below to illustrate how it works in an abstract level.

[ArchitectureDiagram password] | *ArchitectureDiagram_password.png*

Figure 10. Sequence diagram of Password Feature

4.1.2. Design Considerations

Aspect	Alternatives	Pros (+)/ Cons(-)
Implementation of Synchronization	**UI with a password field. (current choice)**	+ : It is simple and user friendly. It serves its purpose as a offline application for a single user. - : The current alternative does not allow the user to change password.
	Using Command Line Interface	+ : It allows more flexibility to implement more parameters such as custom security questions to improve its security. - : It may not be as user friendly as the UI.

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

4.2. Project Complete Feature

The complete feature allows users to indicate a project as completed.

4.2.1. Implementation

To facilitate the complete feature, an association with a new `Complete` class is added to the `Project` class:

[ProjectClassDiagram] | *ProjectClassDiagram.png*

Figure 11. Structure of the attributes of a Project in the Model component.

The diagram shows that the Project class is associated with the Complete class.

The following sequence diagram shows how the complete command works:

[CompleteSequenceDiagram] | *CompleteSequenceDiagram.png*

Figure 12. Figure Sequence diagram for the complete command.

1. The `CompleteCommandParser` parses the user input to obtain the target project index and constructs a ne `CompleteCommand` with this index.
2. The logic portion of the complete command will be executed by the `CompleteCommand` method. To mark a Project object as complete:
    1. The **CompleteCommand()** method creates a `targetProject` based on the provided project index.
    2. In the **executeCommandResult()** method then creates a `editedProject` with `Complete` attribute set to “true”. The `editedProject` is created with ProjectBuilder as shown below:

```
    Project editedProject = new
ProjectBuilder(targetProject).withComplete(true).build();
```

3. In the executeCommandResult() method

```
    model.setProject(targetProject, editedProject)
```

is called to replace `Project`’s complete attribute from “false” to “true” in the VolunCHeer in-memory.

4.2.2. Design Considerations

Aspect	Alternatives	Pros (+)/ Cons(-)
Implementation of ‘CompleteCommand’	**Add a Complete attribute to Project (current choice) -Completed projects indicated “Red”**	+ : It is easy to tag complete status as an attribute to the `Project` as we can make use of current implementations such as model.setProject(Project,Project) that sets the `Project`’s complete attribute to “true”. - : Unable to have a observable list of complete projects.
	Create a new CompletedProjectList that consists of all the complete projects, a listComplete command to show all completed tasks..	+ : Will use less memory (e.g. for deleteVolunteer, just save the volunteer being deleted). - : We must ensure that the implementation of each individual command are correct.

4.3. Assign Feature

Assigning a Beneficiary / VolunteerList to Project.

4.3.1. Implementations

Since the implementation of commands AssignBeneficiary and AssignVolunteer are similar, we will describe the implementation of AssignBeneficiary command only and provide the difference between the two.

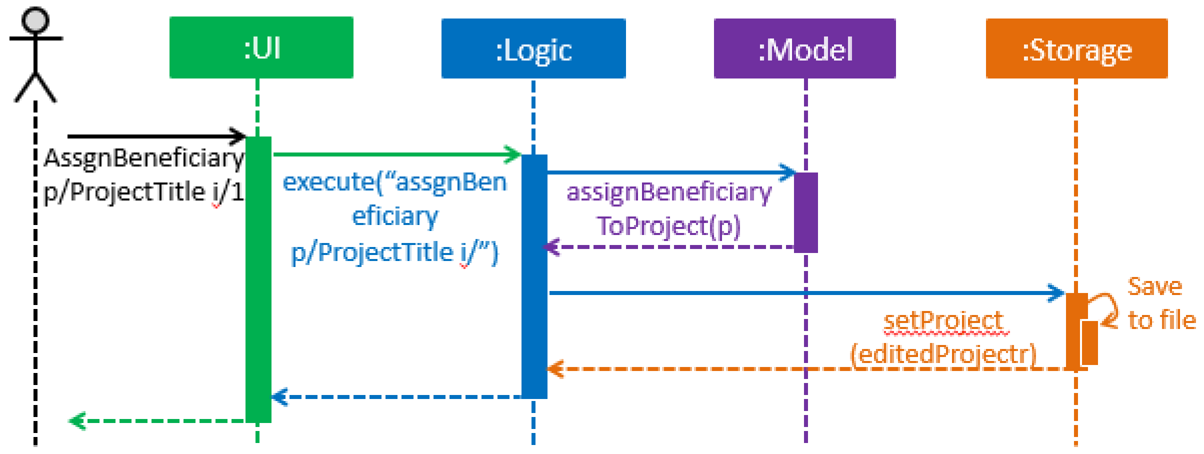

:UI
:Logic
:Model
:Storage
AssgnBeneficiary p/ProjectTitle i/1
execute("assgnBeneficiary p/ProjectTitle i/")
assignBeneficiary ToProject(p)
setProject (editedProjectr)
Save to file

Figure 13. Sequence diagram to show how the AssignBeneficiaryCommand works.

1. The **AssignBeneficiaryCommand(ProjectTitle, Index)** takes in the targetProject's projectTitle attribute and targetBeneficiary's index.
2. The **executeCommandResult()** method
    1. Sets up projectToAssign by calling a predicate to compare with the `ProjectTitle` in `FilteredProjectList`:

        ```
        model. getFilteredProjectList(). filtered(equalProjectTitle).get(0);
        ```

    2. **updateBeneficiary(model)** methods updates the `Beneficiary` object so that ProjectTitle is tracked within the Beneficiary class.
    3. editedProject is created using ProjectBuilder to take in the `Beneficiary` assigned. The following method is called to store the `Project` in VolunCHeer with specific `Beneficiary` attached to it.

        ```
        model.setProject(projectToAssign, editedProject)
        ```

4.4. Beneficiary Management Feature

4.4.1. Implementation

Beneficiary is implement in order to manage the information a benefited volunteer organization. These organizations interact with the user's organization through projects. Hence, `Beneficiary` class has a bidirectional navigability with `Project` class, as shown in the Figure 10.

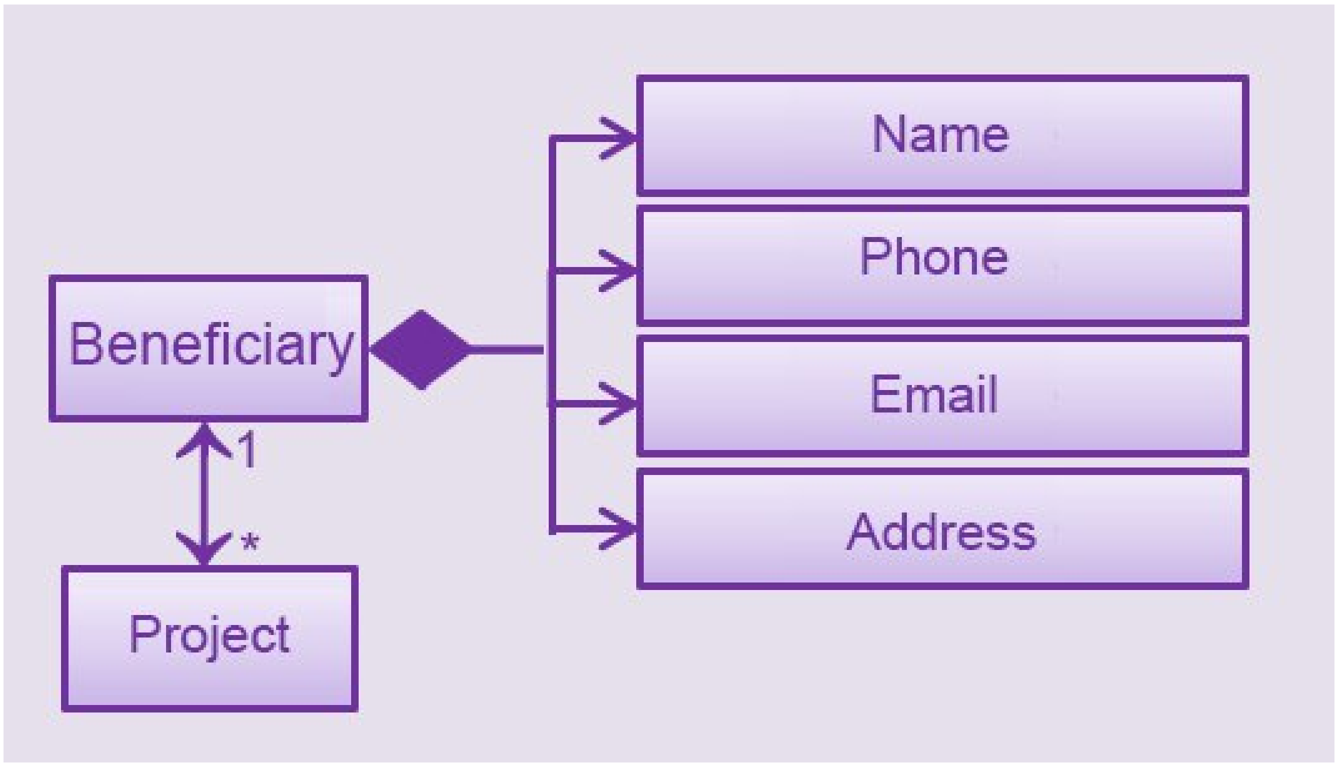


Figure 14. Structure of the `Beneficiary` *class including its attributes, and its bidirectional navigability with* `Project` *class.*

This means that if an operation such as deletion is done on a beneficiary, this should be updated on the projects that the beneficiary is assigned to. The figure below shows how the delete beneficiary command works:

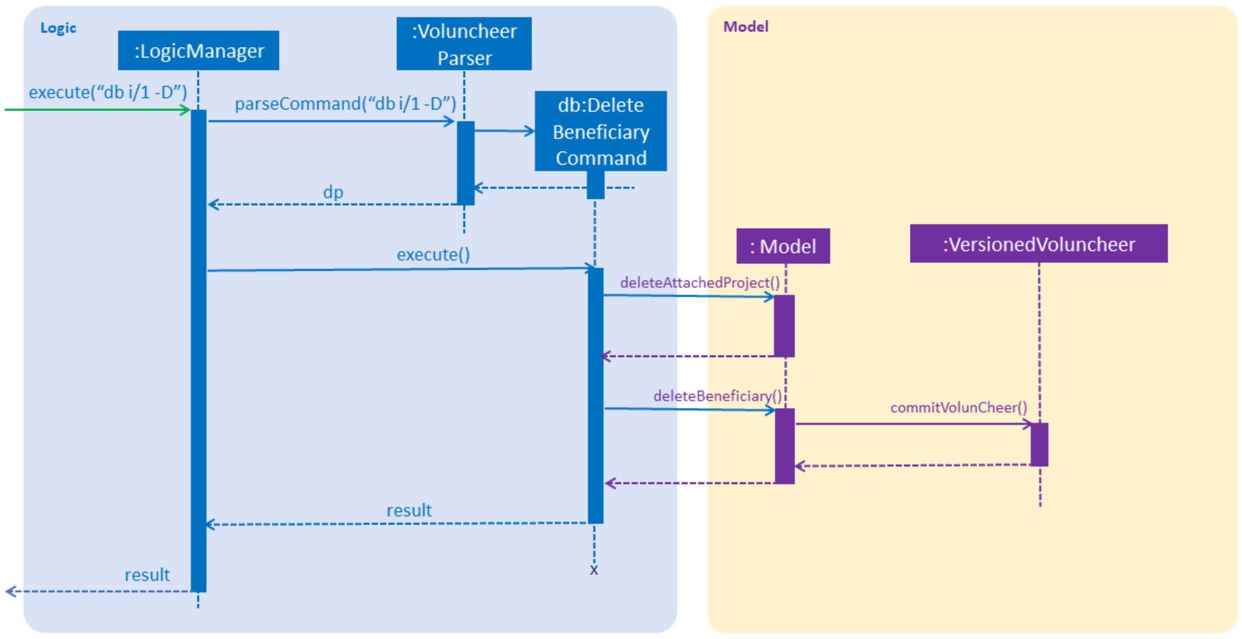


Figure 15. Beneficiary deletion sequence diagram, hard deletion mode.

NOTE "-D" indicates that the deletion is in the hard mode, meaning that the respective projects that are attached to this beneficiary will be deleted.

1. The `DeleteBeneficiaryParser` parses the index of the beneficiary that is required to delete. The Parser constructs a `DeleteBeneficiaryCommand` with constructor as shown below:

```
    public DeleteBeneficiaryCommand(Index targetIndex, boolean isHardDeleteMode) {
        this.targetIndex = targetIndex;
        this.isHardDeleteMode = isHardDeleteMode;
    }
```

2. Method **deleteAttachedProjects(model, beneficiaryToDelete)** then calls the `ModelManager` to update the deletion of the respective projects.

```
    private void deleteAttachedProjects(Model model, Beneficiary beneficiaryToDelete)
{
        HashSet<ProjectTitle> attachedProjects =
beneficiaryToDelete.getHashAttachedProjectLists();
        List<Project> projectsToDelete = new
ArrayList<>(model.getFilteredProjectList());
        for (Project p : projectsToDelete) {
            if (attachedProjects.contains(p.getProjectTitle())) {
                model.deleteProject(p);
            }
        }
    }
```

3. The `ModelManager` is then called to update the deletion of the beneficiary and update all the changes.

```
    model.deleteBeneficiary(beneficiaryToDelete);
    model.commitAddressBook();
```

In order to view the synchronization, you can observe via project pool. This is to alleviate the worries of looking at attached projects when dealing with beneficiary, as a beneficiary can have multiple projects.
However, the Beneficiary Management Feature support the viewing of these information via Summarise Command. The Summarise Command generates the summarised statistics information of beneficiary based on their activeness.

NOTE	The activeness of a beneficiary is measured by the number of projects that beneficiary has collaborated with the user's organization

The sequence diagram below shows how the Summarise Command works.

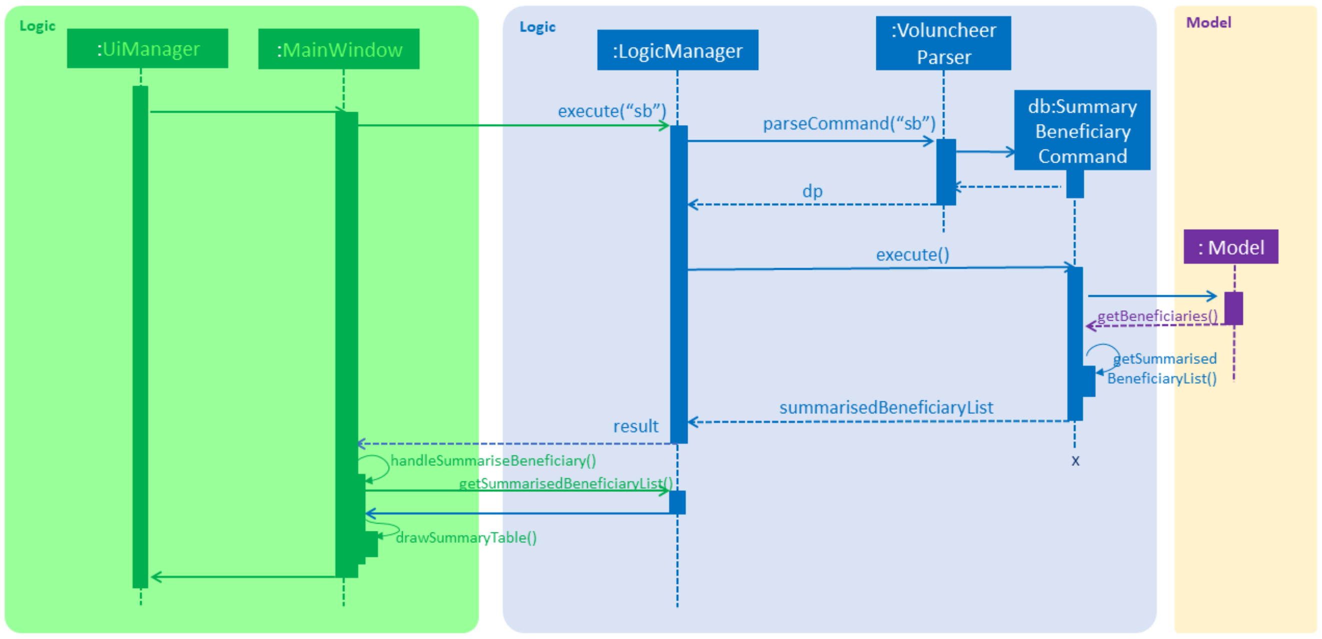


Figure 16. Summarise beneficiary command sequence diagram.

1. `SummaryBeneficiaryCommand` calls the `Model` to get the beneficiary list.
2. A summary list is generated and passed to `Logic`.
3. The Ui component which is `MainWindow` does handling of summarised list by generate a summary table and show on the screen.

4.4.2. Design Considerations

Aspect	Alternatives	Pros (+)/ Cons(-)
Implementation of Synchronization	**Update the by linear search for designed object**	+ : It is easier to implement because the code base are list based. Moreover, the use of the application is limited to only local use without a large amount of data. Hence, this method gives a good performance in the context. - : Unoptimized in terms of complexity, which requires more work for scaling of the application.
	Hash Table of the data	+ : It has a better time complexity and reduce the work in scaling stage since this data structure is more optimized (O(1) can be achieved). - : Take more resources to implement.
Display and use of attached project list	The beneficiary card shows the list	+ : The synchronization can be observed throughout the execution of commands. - : The beneficiary card is full with information and not reader friendly. Moreover, it is unnecessary to see the projects when operating single operations such as add, and edit
	Generation of summary table	+ : The summary gives a good way to look at the statistics of the beneficiary list. As it allows the dynamic of sorting in ascending or descending order of the list based on the beneficiary’s activeness - : The adaptation of Ui is required.

4.5. Filtered Export feature

4.5.1. Current Implementation

To facilitate the filtering mechanism, a new `points` integer field is added to the `Volunteer` class:

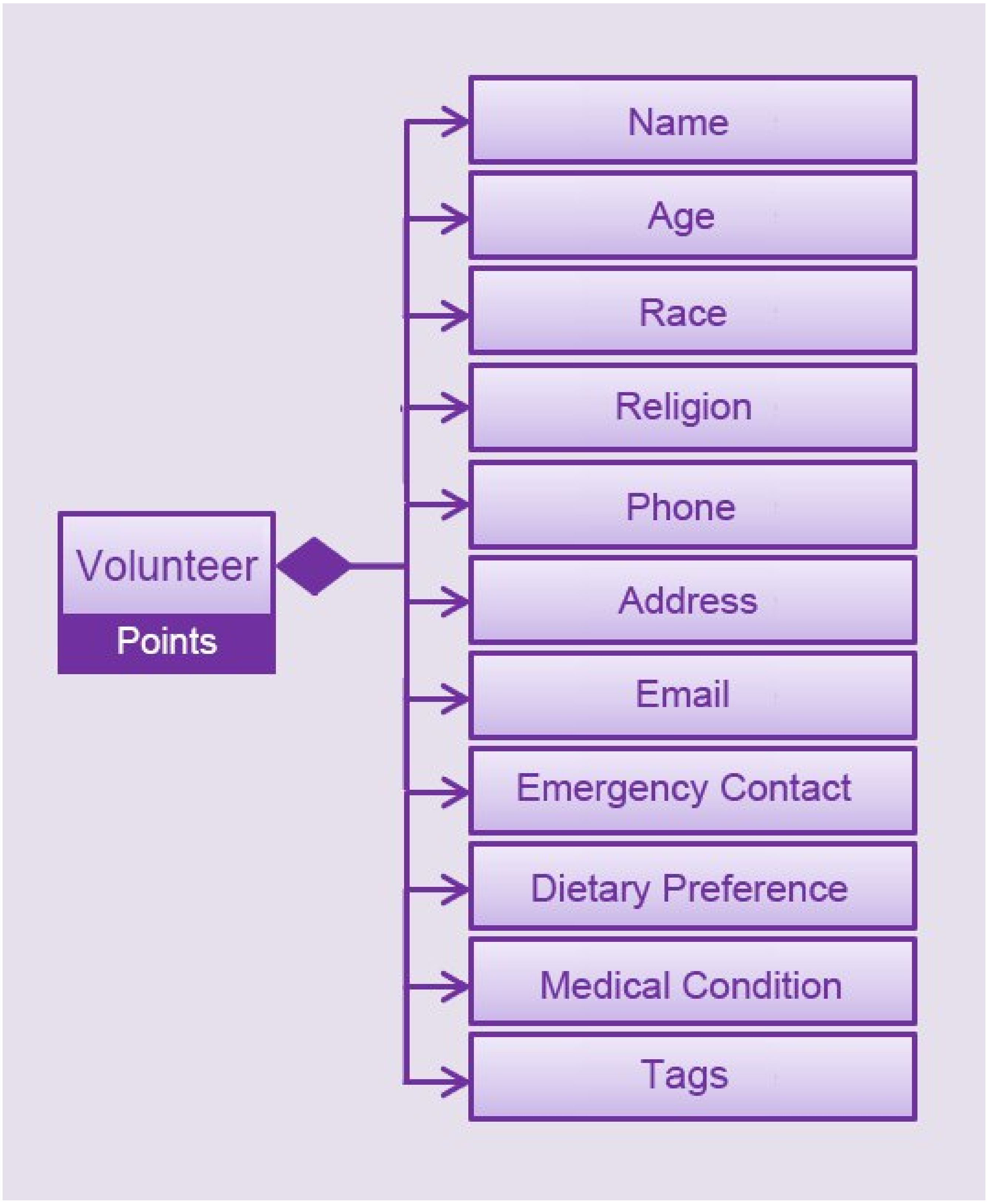


Figure 17. Volunteer class with new points field

`Points` was implemented as a integer instead of a class for ease of access. Also, it is not directly influenced by any input from the user, as input has been checked by the the other classes in `Volunteer`. Hence, no accompanying methods are necessary.

This feature revolves around 3 commands:

1. `map` Command

2. `sort` Command
3. `exportV` Command

For the `map` command, the user inputs the specific criteria to map `Volunteers` on, as well as the points. Upon execution of the `map` command, the following sequence diagram shows how the map command works:

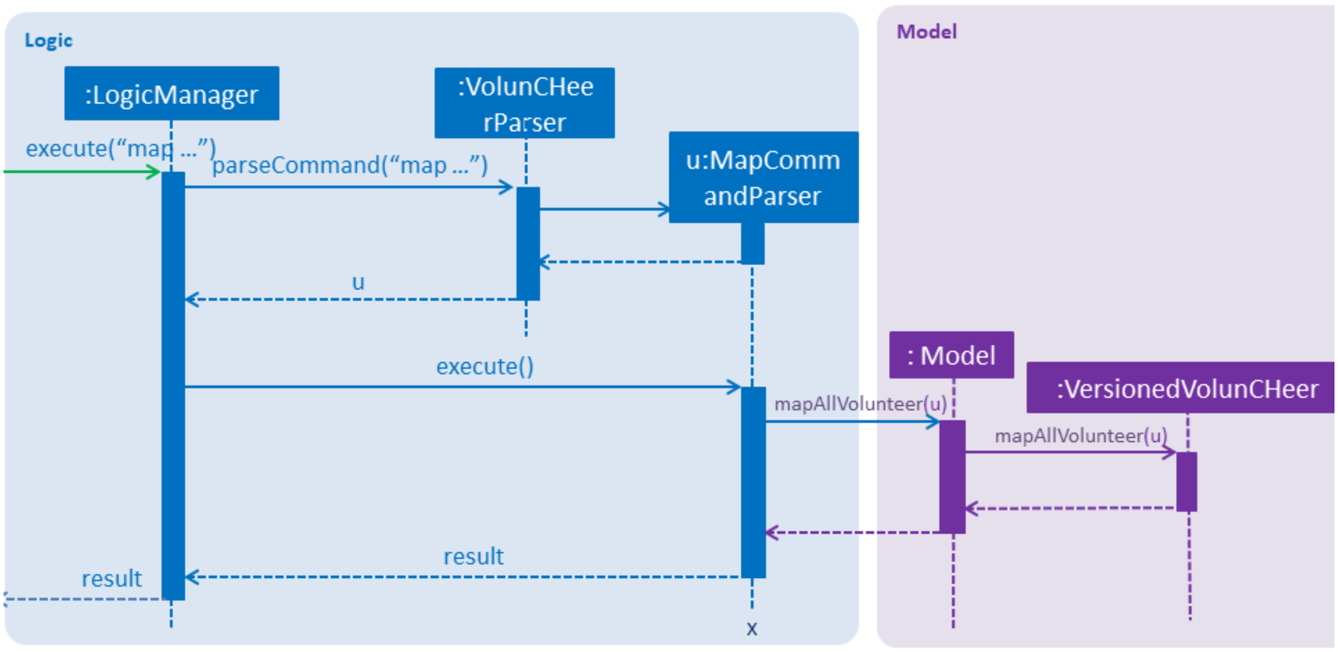


Figure 18. Sequence diagram of the map command

`MapCommandParser` will check the given arguments for correctness, such as proper points or valid comparator. It then creates a `MapObject` and stores the given arguments in a `Pair` of <points, conditions> and passes it to `MapCommand`, which passes it to `Model` by calling `mapAllVolunteers`.

Within the model, `mapAllVolunteers()` is as shown:

```
public void mapAllVolunteer(MapObject map) {
        versionedAddressBook.getVolunteerList().forEach(volunteer -> {
            volunteer.resetPoints();
            volunteer.addPoints(checkAge(map, volunteer));
            volunteer.addPoints(checkRace(map, volunteer));
            volunteer.addPoints(checkMedical(map, volunteer));;
        });
    }
```

The `checkAge`, `checkRace` and `checkMedical` methods check each `Volunteer` and return the given points for that criteria, which `addPoints` adds to them.

`Sort` Command

For the `Sort` command, the `Model` calls the `UniqueVolunteerList` internal `sortByPoints` method. This method uses the standard `FXCollections.sort` on the `internalList`, which immediately reflects in the UI.

```
    public void sortByPoints() {
        FXCollections.sort(internalList, (new Comparator<Volunteer>() {
            public int compare (Volunteer s1, Volunteer s2) {
                return s2.getPoints() - s1.getPoints();
            }
        }));
    }
```

The custom comparator sorts `Volunteers` in descending order of points.

`Export` Command

The `exportV` command writes certain parts of volunteers data based on provided crtieria. It takes on various parameters such as [NUMBER OF VOLUNTEERS], [PREFIX OF DATA REQUIRED 1][PREFIX OF DATA REQUIRED 2] … .

The `ExportVolunteerCommandParser` checks that at least 1 type of data and the number of `Volunteers` is given. It then stores the prefixes in a list called `prefixToBePrinted` and returns the the list and the number of volunteers as a `Pair` to `ExportVolunteer`. The code snippet below shows how the main command is implemented.

```
File output = new File("Export.csv");
        List<String[]> volunteerData = new ArrayList<>();
        volunteerData = model.addData(numVolunteers, prefixToBePrinted);
        try (PrintWriter pw = new PrintWriter(output)) {
            volunteerData.stream()
                    .map(this::toCsv)
                    .forEach(pw::println);
        } catch (IOException e) {
            throw new CommandException("Error writing to file");
        }
```

A `List` of `String` arrays is used to store each line of `Volunteer`. The `addData` method goes through the `Volunteer` list and collects the specified fields into a `String` array, which is appended to another `List` of `String` arrays and returned. The `toCsv` method formats the data into CSV-friendly data.

Below are certain considerations made when designing the filtered export feature.

Aspect	Alternatives	Pros (+)/ Cons(-)
Sorting the internal volunteer list	**Using a SortedList wrapper around the unmodifiable list**	+ : Easy to implement. + Will not affect the actual data. - : The new SortedList has to be added to the UI, or constantly swapped around with the usual list of volunteers.
Using PrintWriter to write out data to CSV	**Using an open-source library such as openCSV to handle the writing.**	+ : Easier to understand and code for any new developers. + openCSV will handle special characters in data. - : External library is required to be installed. - Data to be written is already checked and cleaned to be free of special characters, hence it is not necessary.

4.6. Undo/Redo feature

4.6.1. Current Implementation

The undo/redo mechanism is facilitated by `VersionedVolunCHeer`. It extends `VolunCHeer` with an undo/redo history, stored internally as an `voluncheerBookStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedVolunCHeer#commit()` — Saves the current VolunCHeer book state in its history.
- `VersionedVolunCHeer#undo()` — Restores the previous VolunCHeer book state from its history.
- `VersionedVolunCHeer#redo()` — Restores a previously undone VolunCHeer book state from its history.

These operations are exposed in the `Model` interface as `Model#commitVolunCHeer()`, `Model#undoVolunCHeer()` and `Model#redoVolunCHeer()` respectively.

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 `VersionedVolunCHeer` will be initialized with the initial VolunCHeer book state, and the `currentStatePointer` pointing to that single VolunCHeer book state.

[UndoRedoStartingStateListDiagram] | *UndoRedoStartingStateListDiagram.png*

Step 2. The user executes `deleteVolunteer 5` command to delete the 5th Volunteer in the VolunCHeer book. The `deleteVolunteer` command calls `Model#commitVolunCHeer()`, causing the modified state of the VolunCHeer book after the `delete 5` command executes to be saved in the `VolunCHeerStateList`, and the `currentStatePointer` is shifted to the newly inserted VolunCHeer book state.

[UndoRedoNewCommand1StateListDiagram] | *UndoRedoNewCommand1StateListDiagram.png*

Step 3. The user executes `add n/David …` to add a new volunteer. The `add` command also calls `Model#commitVolunCHeer()`, causing another modified VolunCHeer book state to be saved into the `VolunCHeerStateList`.

[UndoRedoNewCommand2StateListDiagram] | *UndoRedoNewCommand2StateListDiagram.png*

NOTE If a command fails its execution, it will not call `Model#commitVolunCHeer()`, so the VolunCHeer book state will not be saved into the `VolunCHeerStateList`.

Step 4. The user now decides that adding the volunteer was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoVolunCHeer()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous VolunCHeer book state, and restores the VolunCHeer book to that state.

[UndoRedoExecuteUndoStateListDiagram] | *UndoRedoExecuteUndoStateListDiagram.png*

NOTE If the `currentStatePointer` is at index 0, pointing to the initial VolunCHeer book state, then there are no previous VolunCHeer book states to restore. The `undo` command uses `Model#canUndoVolunCHeer()` 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:

[UndoRedoSequenceDiagram] | *UndoRedoSequenceDiagram.png*

The `redo` command does the opposite — it calls `Model#redoVolunCHeer()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the VolunCHeer book to that state.

NOTE If the `currentStatePointer` is at index `VolunCHeerStateList.size() - 1`, pointing to the latest VolunCHeer book state, then there are no undone VolunCHeer book states to restore. The `redo` command uses `Model#canRedoVolunCHeer()` 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 VolunCHeer book, such as `list`, will usually not call `Model#commitVolunCHeer()`, `Model#undoVolunCHeer()` or `Model#redoVolunCHeer()`. Thus, the `VolunCHeerStateList` remains unchanged.

[UndoRedoNewCommand3StateListDiagram] | *UndoRedoNewCommand3StateListDiagram.png*

Step 6. The user executes `clear`, which calls `Model#commitVolunCHeer()`. Since the `currentStatePointer` is not pointing at the end of the `VolunCHeerStateList`, all VolunCHeer book states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add n/David …` command. This is the behavior that most modern desktop applications follow.

[UndoRedoNewCommand4StateListDiagram] | *UndoRedoNewCommand4StateListDiagram.png*

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

[UndoRedoActivityDiagram] | *UndoRedoActivityDiagram.png*

4.6.2. Design Considerations

Aspect	Alternatives	Pros (+)/ Cons(-)
How undo & redo executes	**Saves the entire VolunCHeer book.**	+ : Easy to implement. - : May have performance issues in terms of memory usage.
	Individual command knows how to undo/redo by itself.	+ : Will use less memory (e.g. for `deleteVolunteer`, just save the volunteer being deleted). - : We must ensure that the implementation of each individual command are correct.
Data structure to support the undo/redo commands	**Use a list to store the history of VolunCHeer book states.**	+ : Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.+ - : Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedVolunCHeer`.
	Use `HistoryManager` for undo/redo	+ : We do not need to maintain a separate list, and just reuse what is already in the codebase. - : 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.

4.7. [Proposed] Project Calendar

_{The projectcalendar mechanism takes the projectTitle and projectDate attribute of the project list and apply them into - Google Calendar API such that the UI now includes a calendar interface and projects sorted according to date. The API has a dependency on Google API Client Library and build.gradle file compiles 'com.google.api-client:google-api-client:1.25.0'.

4.8. 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 4.9, "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

4.9. Configuration

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

5. Documentation

We use asciidoc for writing documentation.

NOTE We chose asciidoc over Markdown because asciidoc, although a bit more complex than Markdown, provides more flexibility in formatting.

5.1. Editing Documentation

See UsingGradle.adoc to learn how to render `.adoc` files locally to preview the end result of your edits. Alternatively, you can download the AsciiDoc plugin for IntelliJ, which allows you to preview the changes you have made to your `.adoc` files in real-time.

5.2. Publishing Documentation

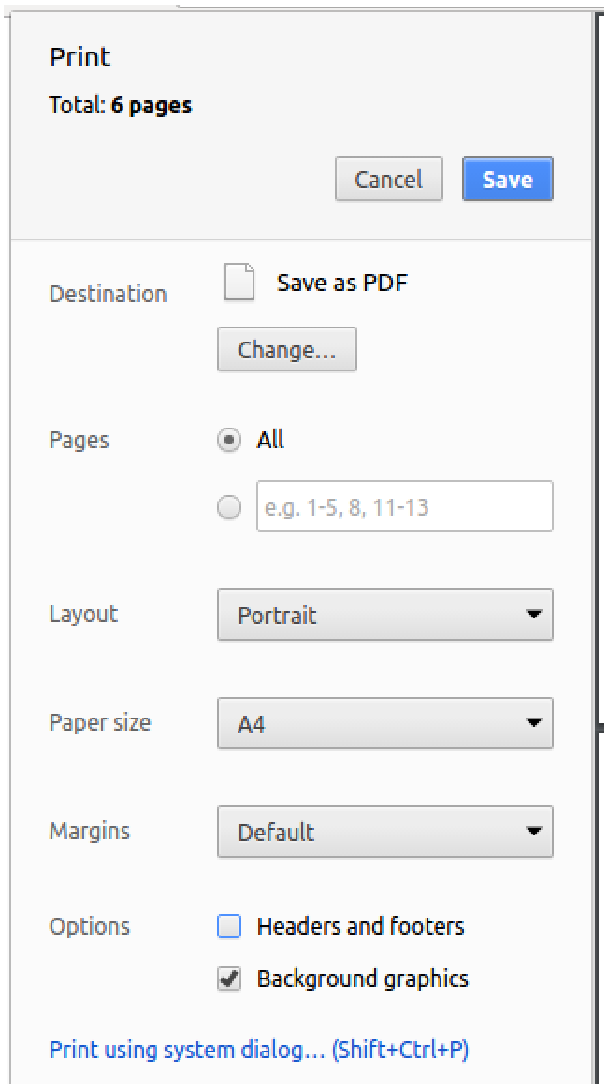
See UsingTravis.adoc to learn how to deploy GitHub Pages using Travis.

5.3. Converting Documentation to PDF format

We use Google Chrome for converting documentation to PDF format, as Chrome's PDF engine preserves hyperlinks used in webpages.

Here are the steps to convert the project documentation files to PDF format.

1. Follow the instructions in UsingGradle.adoc to convert the AsciiDoc files in the `docs/` directory to HTML format.
2. Go to your generated HTML files in the `build/docs` folder, right click on them and select `Open with` → `Google Chrome`.
3. Within Chrome, click on the `Print` option in Chrome's menu.
4. Set the destination to `Save as PDF`, then click `Save` to save a copy of the file in PDF format. For best results, use the settings indicated in the screenshot below.



Print

Total: 6 pages

Destination: Save as PDF

Pages: (● All) ○ e.g. 1-5, 8, 11-13

Layout: Portrait

Paper size: A4

Margins: Default

Options: ☐ Headers and footers; ☑ Background graphics

Figure 19. Saving documentation as PDF files in Chrome

5.4. Site-wide Documentation Settings

The `build.gradle` file specifies some project-specific asciidoc attributes which affects how all documentation files within this project are rendered.

TIP Attributes left unset in the `build.gradle` file will use their **default value**, if any.

Table 1. List of site-wide attributes

Attribute name	Description	Default value
`site-name`	The name of the website. If set, the name will be displayed near the top of the page.	*not set*
`site-githuburl`	URL to the site's repository on GitHub. Setting this will add a "View on GitHub" link in the navigation bar.	*not set*
`site-seedu`	Define this attribute if the project is an official SE-EDU project. This will render the SE-EDU navigation bar at the top of the page, and add some SE-EDU-specific navigation items.	*not set*

5.5. Per-file Documentation Settings

Each `.adoc` file may also specify some file-specific asciidoc attributes which affects how the file is rendered.

Asciidoctor's built-in attributes may be specified and used as well.

TIP Attributes left unset in `.adoc` files will use their **default value**, if any.

Table 2. List of per-file attributes, excluding Asciidoctor's built-in attributes

Attribute name	Description	Default value
`site-section`	Site section that the document belongs to. This will cause the associated item in the navigation bar to be highlighted. One of: `UserGuide`, `DeveloperGuide`, `LearningOutcomes`*, `AboutUs`, `ContactUs` ** Official SE-EDU projects only*	*not set*
`no-site-header`	Set this attribute to remove the site navigation bar.	*not set*

5.6. Site Template

The files in `docs/stylesheets` are the CSS stylesheets of the site. You can modify them to change some properties of the site's design.

The files in `docs/templates` controls the rendering of `.adoc` files into HTML5. These template files are written in a mixture of Ruby and Slim.

WARNING Modifying the template files in `docs/templates` requires some knowledge and experience with Ruby and Asciidoctor’s API. You should only modify them if you need greater control over the site’s layout than what stylesheets can provide. The SE-EDU team does not provide support for modified template files.

6. Testing

6.1. Running Tests

There are three ways to run tests.

TIP The most reliable way to run tests is the 3rd one. The first two methods might fail some GUI tests due to platform/resolution-specific idiosyncrasies.

Method 1: Using IntelliJ JUnit test runner

- To run all tests, right-click on the `src/test/java` folder and choose `Run 'All Tests'`
- To run a subset of tests, you can right-click on a test package, test class, or a test and choose `Run 'ABC'`

Method 2: Using Gradle

- Open a console and run the command `gradlew clean allTests` (Mac/Linux: `./gradlew clean allTests`)

NOTE See UsingGradle.adoc for more info on how to run tests using Gradle.

Method 3: Using Gradle (headless)

Thanks to the TestFX library we use, our GUI tests can be run in the *headless* mode. In the headless mode, GUI tests do not show up on the screen. That means the developer can do other things on the Computer while the tests are running.

To run tests in headless mode, open a console and run the command `gradlew clean headless allTests` (Mac/Linux: `./gradlew clean headless allTests`)

6.2. Types of tests

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include,
   a. *System Tests* that test the entire App by simulating user actions on the GUI. These are in the `systemtests` package.
   b. *Unit tests* that test the individual components. These are in `seedu.VolunCHeer.ui` package.
2. **Non-GUI Tests** - These are tests not involving the GUI. They include,

a. *Unit tests* targeting the lowest level methods/classes.
   e.g. `seedu.VolunCHeer.commons.StringUtilTest`

b. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working).
   e.g. `seedu.VolunCHeer.storage.StorageManagerTest`

c. Hybrids of unit and integration tests. These test are checking multiple code units as well as how the are connected together.
   e.g. `seedu.VolunCHeer.logic.LogicManagerTest`

6.3. Troubleshooting Testing

Problem: `HelpWindowTest` fails with a `NullPointerException`.

- Reason: One of its dependencies, `HelpWindow.html` in `src/main/resources/docs` is missing.
- Solution: Execute Gradle task `processResources`.

7. Dev Ops

7.1. Build Automation

See UsingGradle.adoc to learn how to use Gradle for build automation.

7.2. Continuous Integration

We use Travis CI and AppVeyor to perform *Continuous Integration* on our projects. See UsingTravis.adoc and UsingAppVeyor.adoc for more details.

7.3. Coverage Reporting

We use Coveralls to track the code coverage of our projects. See UsingCoveralls.adoc for more details.

7.4. Documentation Previews

When a pull request has changes to asciidoc files, you can use Netlify to see a preview of how the HTML version of those asciidoc files will look like when the pull request is merged. See UsingNetlify.adoc for more details.

7.5. Making a Release

Here are the steps to create a new release.

1. Update the version number in `MainApp.java`.
2. Generate a JAR file using Gradle.

3. Tag the repo with the version number. e.g. `v0.1`
4. Create a new release using GitHub and upload the JAR file you created.

7.6. Managing Dependencies

A project often depends on third-party libraries. For example, VolunCHeer Book depends on the Jackson library for JSON parsing. Managing these *dependencies* can be automated using Gradle. For example, Gradle can download the dependencies automatically, which is better than these alternatives:

a. Include those libraries in the repo (this bloats the repo size)

b. Require developers to download those libraries manually (this creates extra work for developers)

Appendix A: Product Scope

Target user profile:

- manager of a volunteer organization such as shool's CCAs, CIP office
- has a need to manage significant number of volunteers but not attached exclusively to any other volunteering program
- has a need to manage a significant number of interested beneficiaries who want to connect to the volunteers
- has a need to manage multiple projects
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: * manage volunteers, beneficiaries, projects' details 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…
`* * *`	new user	see usage instructions	refer to instructions when I forget how to use the App

Priority	As a …	I want to …	So that I can…
* * *	volunteering project manager	add a new volunteer	have their information in the system to manage and distribute them
* * *	volunteering project manager	delete an existing volunteer	remove the volunteer that no longer needs
* * *	volunteering project manager	edit a volunteer	update information of volunteer
* * *	volunteering project manager	find a volunteer by name	locate details of the volunteer without having to go through the entire list
* * *	volunteering project manager	hide private contact details by default	minimize chance of someone else seeing them by accident
* * *	volunteering project manager	sort volunteer list by name	locate a the volunteer easily
* * *	volunteering project manager	add a beneficiary	have their infomation in the system to manage
* * *	volunteering project manager	add beneficiary’s description	have a description of beneficiary to refer to
`* `	volunteering project manager	highlight details/ keywords in the beneficiary’s description	read and scan through the information easily
* * *	volunteering project manager	delete a beneficiary	remove beneficary
* * *	volunteering project manager	edit a beneficiary	update details if there is any changes

Priority	As a …	I want to …	So that I can…
* * *	volunteering project manager	sort the beneficiary by name or more	easily manage the list of beneficiary
* *	volunteering project manager	see the summary of beneficiary based on their activeness	gain overview of beneficiaries to collaborate with or seek funding from
* * *	volunteering project manager	add a new project with specific details	manage the project and allocate volunteers in the project
* * *	volunteering project manager	edit a project	change details of the project if needed
* * *	volunteering project manager	delete a project	remove projects that is abundant, cancelled or outdated
* *	volunteering project manager	take attendance of volunteers for a project	keep track of volunteers's attendance
* *	volunteering project manager	remind the most prioritised/ closed to dealine project	remind me to work of pay special attention to that project's progress
*	volunteering project manager	have a calendar of projects on the GUI	easily visualize the timeline of work and projects
* *	volunteering project manager	have a recommendation list of volunteer based on several factors	easily adding relevant volunteers in a project
* *	volunteering project manager	import, export data	easily transfer the data to other machines to use

Priority	As a …	I want to …	So that I can…
* *	volunteering project manager	undo, redo	go back to my preferred state if I make a mistake
*	user	have autofill function on command line	type faster

Appendix C: Use Cases

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

Use case 1: Delete volunteer

MSS

1. User requests to list volunteers
2. VolunCHeer shows a list of volunteers
3. User requests to delete a specific volunteer in the list
4. VolunCHeer deletes the volunteer

   Use case ends.

Extensions

  2a. The list is empty.

  Use case ends.

  3a. The given index is invalid.

    3a1. VolunCHeer shows an error message.

    Use case resumes at step 2.

Use case 2: Add volunteer

MSS

1. User requests to add a volunteer, including name, age, email, address, etc.
2. VolunCHeer shows the successful add message

   Use case ends.

Extensions

2a. The volunteer has existed, show edit option

Use case ends.

3a. The given command line is invalid.

3a1. VolunCHeer shows an error message.

Use case ends.

Use case 3: Edit volunteer

MSS

1. Users requests to find a volunteer.
2. User requests to edit the volunteer.
3. VolunCHeer shows the successful edit message.

Use case ends.

Extensions

1a. The volunteer cannot be found

Use case ends.

2a. Given index for edit command is invalid.

2a1. VolunCHeer shows an error message.

Use case ends.

Use case 4: Add Project

MSS

1. Users requests to add a project.
2. VolunCHeer shows the successful add message.

Use case ends.

Extensions

2a. The command line is invalid.

2a1. VolunCHeer shows an error message.

Use case ends.

2b. The beneficiary is not existed.

2b1. VolunCHeer shows an error message.

2b. The date is invalid.

2b1. VolunCHeer shows an error message.

Use case ends.

2c. The project is existed.

2c1. VolunCHeer shows edit option.

Use case ends.

Use case 5: Edit Project

MSS

1. Users requests to edit a project.
2. VolunCHeer shows the successful edit message.

   Use case ends.

Extensions

2a. The project is not existed.

2a1. VolunCHeer shows an error message.

Use case ends.

Use case 6: Find volunteer

MSS

1. Users requests to find (a) volunteer/volunteers by name.
2. VolunCHeer shows the list of volunteers who share the name.

   Use case ends.

Extensions

2a. There is no volunteer with that name.

2a1. VolunCHeer returns an empty list.

Use case ends.

Use case 7: Delete Project

MSS

1. User requests to delete a specific project by name
2. VolunCHeer deletes the project

   Use case ends.

Extensions

   2a. project is not existed.

      2a1. VolunCHeer shows an error message.

      Use case ends.

Use case 8: export volunteer list

MSS

1. User requests to import a volunteer file
2. VolunCHeer imports the volunteer file to the volunteer list

   Use case ends.

Extensions

   2a. file cannot be found.

      2a1. VolunCHeer shows an error message.

      Use case ends.

Use case 9: export volunteer list

MSS

1. User requests to export a volunteer file
2. VolunCHeer exports new volunteer data file

   Use case ends.

Extensions

   2a. the file has existed.

      2a1. VolunCHeer overwritten the file.

Use case ends.

Use case 10: export volunteer list

MSS

1. User requests to export a volunteer file
2. VolunCHeer exports new volunteer data file

   Use case ends.

Extensions

2a. the file has existed.

2a1. VolunCHeer overwritten the file.

Use case ends.

Use case 11: Add a beneficiary

MSS

1. User requests to add a beneficiary.
2. VolunCHeer shows the successful add message

   Use case ends.

Extensions

2a. The beneficiary has existed, show error message

Use case ends.

2b. The given command line is invalid.

2b1. VolunCHeer shows an error message.

Use case ends.

Use case 12: Edit a beneficiary

MSS

1. Users requests to edit a beneficiary.
2. VolunCHeer shows the successful edit message.

   Use case ends.

Extensions

2a. The beneficiary is not existed.

2a1. VolunCHeer shows an error message.

Use case ends.

Use case 13: Delete a beneficiary (soft delete)

MSS

1. Users requests to delete a beneficiary.
2. VolunCHeer shows the successful delete message.

Use case ends.

Extensions

2a. The beneficiary is not existed.

2a1. VolunCHeer shows an error message.

Use case ends.

2b. The beneficiary has attached projects.

2b1. VolunCHeer shows an error message.

Use case ends.

Use case 14: Sort volunteers based on PRIORITY_SCORE

MSS

1. User uses "map" command to calculate PRIORITY_SCORE.
2. User requests to make a sorted list of volunteers based on PRIORITY_SCORE.
3. VolunCHeer shows the successful sorted list.

Use case ends.

Extensions

2a. Invalid map features.

- 2b1. VolunCHeer shows error message. Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `9` or higher installed.
2. Should be able to hold up to 1000 volunteers 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.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

A contact detail that is not meant to be shared with others

Appendix F: Product Survey

VolunCHeer

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: 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.

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 volunteer

1. Deleting a volunteer while all volunteers are listed

   a. Prerequisites: List all volunteers using the `list` command. Multiple volunteers in the list.

   b. Test case: `deleteVolunteer 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: `deleteVolunteer 0`
   Expected: No volunteer is deleted. Error details shown in the status message. Status bar remains the same.

   d. Other incorrect delete commands to try: `deleteVolunteer`, `deleteVolunteer 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 … }