Voluncheer - Developer Guide

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

1. Setting up

1.1. 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.

1.2. 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)

1.3. Verifying the setup

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

1.4. Configurations to do before writing code

1.4.1. 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.

2. Design

2.1. 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.

2.2. UI component

[UiClassDiagram] | *UiClassDiagram.png*

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.

2.3. Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `VoluncheerBookParser` 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.

2.4. Model component

[ModelClassDiagram] | *ModelClassDiagram.png*

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 (Vounteer, 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.

2.5. 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.

2.6. Common classes

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

3. Implementation

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

3.1. [Proposed] Command Line Recommendation feature

3.1.1. Proposed Implementation

The command line recommendation feature is facilitated by the `CommandLineParser`. It imports the 'CliSyntax.Java' and stores internally the `UserClosestInput` and `CommandUsed`. It updates the `UserClosestInput` and process to give suggestions.

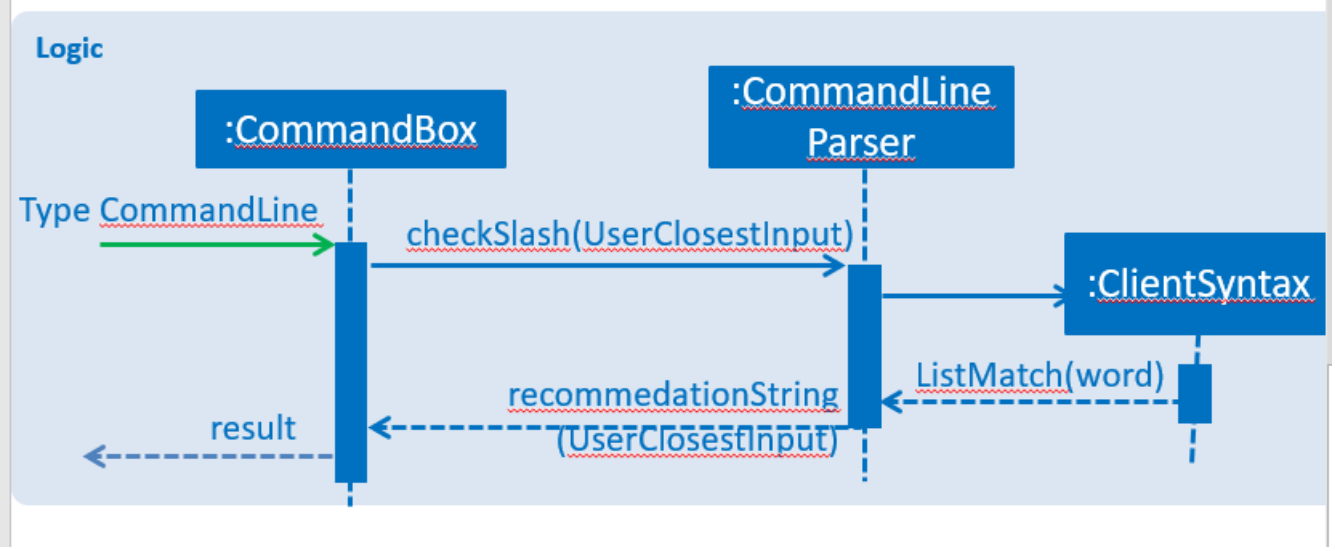
+ Given below is an example usage scenario and how the command line recommendation feature behaves at each step.

Step 1: The user types in a command keyword then type [space], the CommandLineParser is initialized which stores the `UserClosestInput` and `CommandUsed`. Then the command line show the command syntax.

Step 2: The user continues to type, `UserClosestInput` takes the input and stores. There are 2 alternatives: * If the user types a slash [\] the `UserClosestInput` waits for syntax and when syntax match, shows the recommandation list for that particular slash sub syntax.

- If the user types without the slash [\] the example remains.

Step 3: After the users type [Enter] the class is cleared.



3.1.2. Design Considerations

Aspect: How often should the `UserClosestInput` refreshs itself

- **Alternative 1 (current choice):** Everytime the [space] key is used.
  - Pros: Easy to implement.
  - Cons: Unable to dynamicly support the user.
- **Alternative 2:** Everytime a new character is type.
  - Pros: Very dynamic in the UI and supporting the user.
  - Cons: Potential to cause lagging, harder to implement.

Aspect (proposed): Choices for user to quickly choose the recommendation

- **Alternative 1 (current choice):** [tab] when only 1 choice left.
  - Pros: Easy to implement.
  - Cons: Unoptimized for this particular purpose.
- **Alternative 2:** Arrow key.
  - Pros: Good for user experiences.
  - Cons: Might be difficult to implement.

3.2. Undo/Redo feature

3.2.1. Current Implementation

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

- `VersionedVoluncheerBook#commit()` — Saves the current Voluncheer book state in its history.
- `VersionedVoluncheerBook#undo()` — Restores the previous Voluncheer book state from its history.

- `VersionedVoluncheerBook#redo()` — Restores a previously undone Voluncheer book state from its history.

These operations are exposed in the `Model` interface as `Model#commitVoluncheerBook()`, `Model#undoVoluncheerBook()` and `Model#redoVoluncheerBook()` 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 `VersionedVoluncheerBook` 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#commitVoluncheerBook()`, causing the modified state of the Voluncheer book after the `delete 5` command executes to be saved in the `VoluncheerBookStateList`, 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#commitVoluncheerBook()`, causing another modified Voluncheer book state to be saved into the `VoluncheerBookStateList`.

[UndoRedoNewCommand2StateListDiagram] | *UndoRedoNewCommand2StateListDiagram.png*

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

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#undoVoluncheerBook()`, 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#canUndoVoluncheerBook()` 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#redoVoluncheerBook()`, 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 `VoluncheerBookStateList.size() - 1`, pointing to the latest Voluncheer book state, then there are no undone Voluncheer book states to restore. The `redo` command uses `Model#canRedoVoluncheerBook()` 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#commitVoluncheerBook()`, `Model#undoVoluncheerBook()` or `Model#redoVoluncheerBook()`. Thus, the `VoluncheerBookStateList` remains unchanged.

[UndoRedoNewCommand3StateListDiagram] | *UndoRedoNewCommand3StateListDiagram.png*

Step 6. The user executes `clear`, which calls `Model#commitVoluncheerBook()`. Since the `currentStatePointer` is not pointing at the end of the `VoluncheerBookStateList`, 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*

3.2.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire Voluncheer book.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `deleteVolunteer`, just save the volunteer being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of Voluncheer book states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedVoluncheerBook`.
- **Alternative 2:** Use `HistoryManager` for undo/redo

- Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
- Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.3. [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'.

3.4. [Proposed] Delete Project

_{The deleteProject is facilitated by DeleterProjectCommand Parser. deleteProject(index) removes the project with index, alongside with date attribute but beneficiary remains. if the project index is not found, DeleteProjectCommand throws ParseException.

3.5. Logging

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

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 3.6, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

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

3.6. Configuration

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

4. 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.

4.1. 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.

4.2. Publishing Documentation

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

4.3. 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.

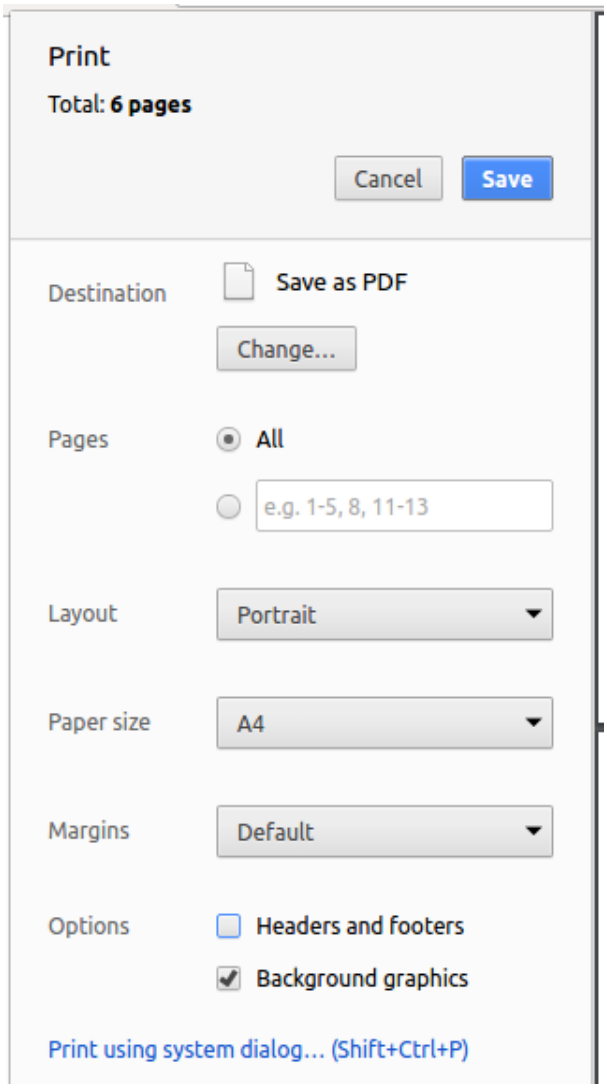


Figure 8. Saving documentation as PDF files in Chrome

4.4. 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*

Attribute name	Description	Default value
`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*

4.5. 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*

4.6. 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.

5. Testing

5.1. 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`)

5.2. 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`

5.3. 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`.

6. Dev Ops

6.1. Build Automation

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

6.2. Continuous Integration

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

6.3. Coverage Reporting

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

6.4. 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.

6.5. 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.

6.6. 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
* * *	volunteer manager	add a new volunteer	have their information in the system to manage and distribute them

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

Priority	As a …	I want to …	So that I can…
* * *	volunteer manager	add a new project with specific details	manage the project and allocate volunteers in the project
* * *	volunteer manager	edit a project	change details of the project if needed
* * *	volunteer manager	delete a project	remove projects that is abundant, cancelled or outdated
* *	volunteer manager	take attendance of volunteers for a project	keep track of volunteers's attendance
* *	volunteer manager	remind the most prioritised/ closed to dealine project	remind me to work of pay special attention to that project's progress
*	volunteer manager	have a calendar of projects on the GUI	easily visualize the timeline of work and projects
* *	volunteer manager	have a recommendation list of volunteer based on several factors	easily adding relevant volunteers in a project
* *	volunteer manager	import, export data	easily transfer the data to other machines to use
* *	volunteer manager	undo, redo	go back to my preferred state if I make a mistake
* *	volunteer manager	have autofill function on command line	type faster

Appendix C: Use Cases

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

Use case 1: Delete volunteer

MSS

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

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. VoluncheerBook 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. VoluncheerBook 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. VoluncheerBook 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. VoluncheerBook 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. VoluncheerBook shows an error message.

Use case ends.

Use case 4: Add Project

MSS

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

   Use case ends.

Extensions

2a. The command line is invalid.

2a1. VoluncheerBook shows an error message.

Use case ends.

2b. The beneficiary is not existed.

2b1. VoluncheerBook shows an error message.

2b. The date is invalid.

2b1. VoluncheerBook shows an error message.

Use case ends.

2c. The project is existed.

2c1. VoluncheerBook shows edit option.

Use case ends.

Use case 5: Edit Project

MSS

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

   Use case ends.

Extensions

2a. The project is not existed.

2a1. VoluncheerBook shows an error message.

Use case ends.

Use case 5: Find volunteer

MSS

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

   Use case ends.

Extensions

2a. There is no volunteer with that name.

2a1. VoluncheerBook returns an empty list.

Use case ends.

Use case 6: Delete Project

MSS

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

   Use case ends.

Extensions

2a. project is not existed.

2a1. VoluncheerBook shows an error message.

Use case ends.

Use case 7: export volunteer list

MSS

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

Use case ends.

Extensions

2a. file cannot be found.

2a1. VoluncheerBook shows an error message.

Use case ends.

Use case 8: export volunteer list

MSS

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

Use case ends.

Extensions

2a. the file has existed.

2a1. VoluncheerBook overwritten the file.

Use case ends.

Use case 9: export volunteer list

MSS

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

Use case ends.

Extensions

2a. the file has existed.

2a1. VoluncheerBook overwritten the file.

Use case ends.

Use case 10: Add beneficiary

MSS

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

Use case ends.

Extensions

2a. The beneficiary has existed, show edit option

Use case ends.

3a. The given command line is invalid.

3a1. VoluncheerBook shows an error message.

Use case ends.

Use case 11: 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. VoluncheerBook shows the successful sorted list.

Use case ends.

Extensions

2a. Invalid map features.

- 2b1. VoluncheerBook 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 … }