UltiStudent - Developer Guide

By: `Team W09-1` 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.ultistudent.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.

1.4.2. Setting up CI

Set up Travis to perform Continuous Integration (CI) for your fork. See UsingTravis.adoc to learn how to set it up.

After setting up Travis, you can optionally set up coverage reporting for your team fork (see UsingCoveralls.adoc).

NOTE Coverage reporting could be useful for a team repository that hosts the final version but it is not that useful for your personal fork.

Optionally, you can set up AppVeyor as a second CI (see UsingAppVeyor.adoc).

NOTE Having both Travis and AppVeyor ensures your App works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based)

1.4.3. Getting started with coding

When you are ready to start coding,

1. Get some sense of the overall design by reading Section 2.1, "Architecture".

2. Take a look at Appendix A, *Suggested Programming Tasks to Get Started*.

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 `delete 1`.

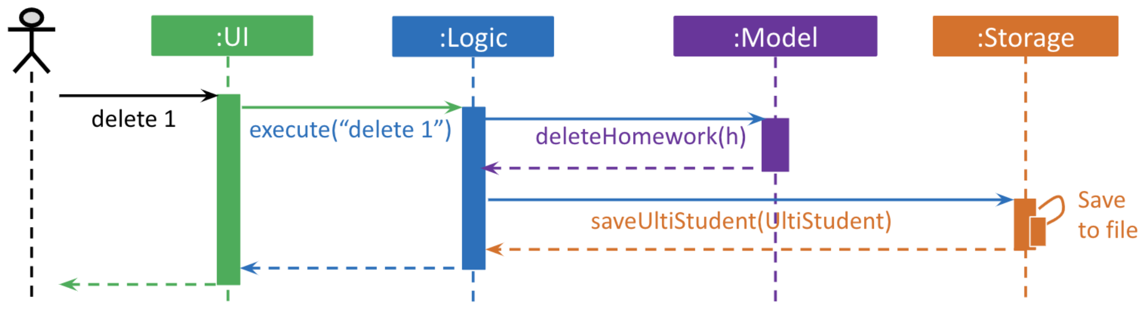


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

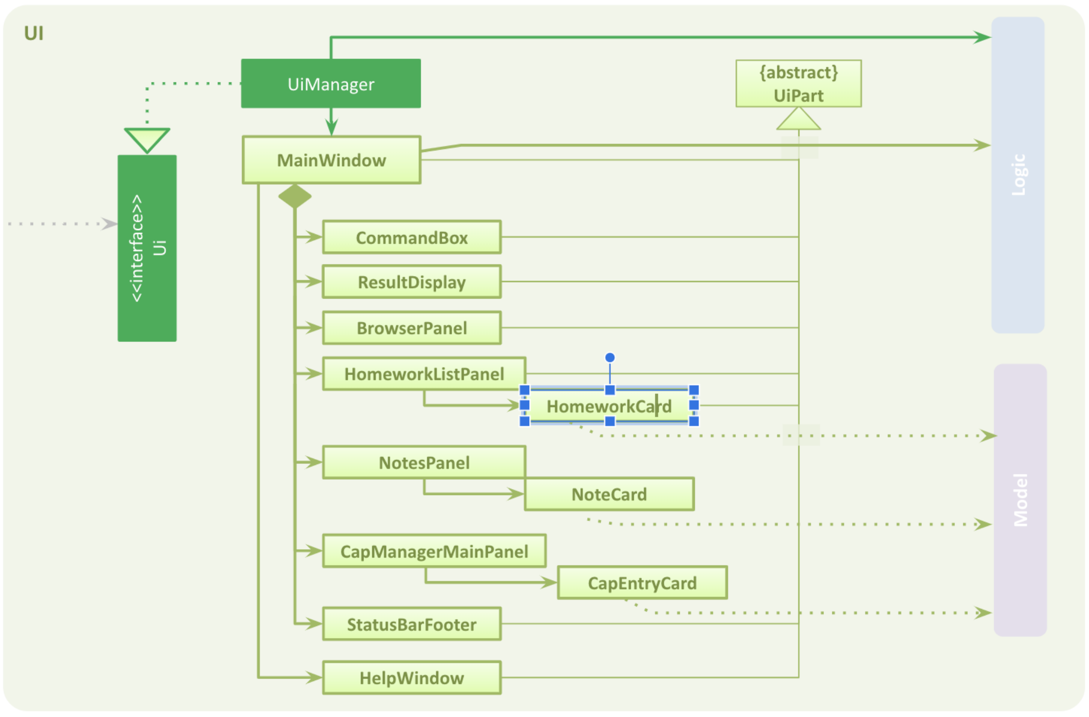


Figure 4. Structure of the UI Component

API : `Ui.java`

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

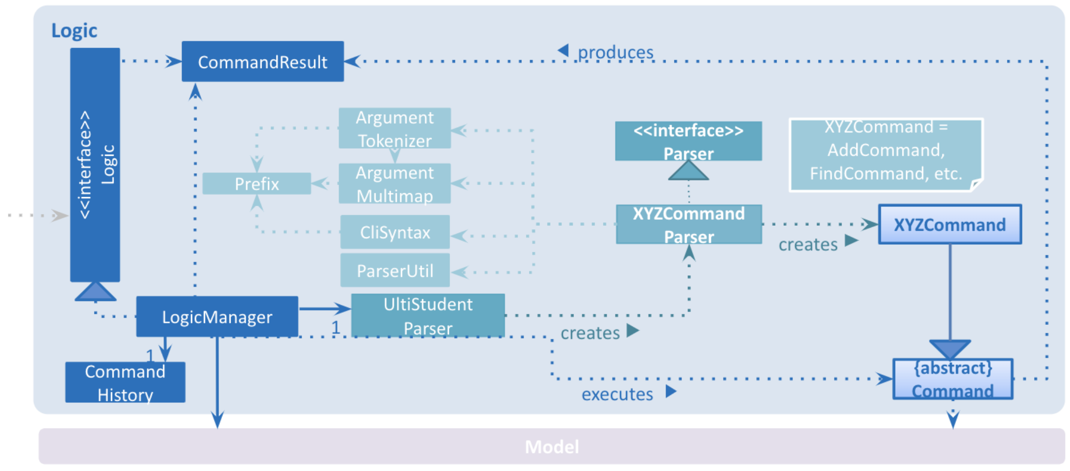


Figure 5. Structure of the Logic Component

API : `Logic.java`

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

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

[DeletePersonSdForLogic] | *DeletePersonSdForLogic.png*

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

2.4. Model component

[ModelClassDiagram] | *ModelClassDiagram.png*

Figure 7. Structure of the Model Component

API : `Model.java`

The `Model`,

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

2.5. Storage component

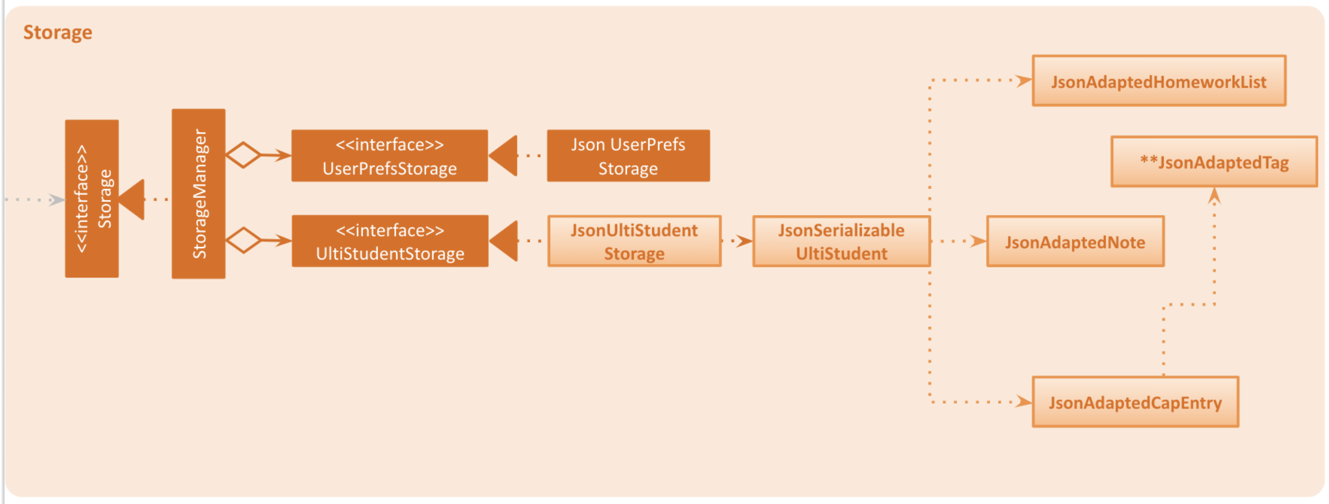


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

2.6. Common classes

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

3. Implementation

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

3.1. Undo/Redo feature

3.1.1. Current Implementation

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

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

These operations are exposed in the `Model` interface as `Model#commitUltiStudent()`, `Model#undoUltiStudent()` and `Model#redoUltiStudent()` 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 `VersionedUltiStudent` will be initialized with the initial UltiStudent state, and the `currentStatePointer` pointing to that single UltiStudent state.

[UndoRedoStartingStateListDiagram] | *UndoRedoStartingStateListDiagram.png*

Step 2. The user executes `delete 5` command to delete the 5th homework in the UltiStudent. The `delete` command calls `Model#commitUltiStudent()`, causing the modified state of the UltiStudent after the `delete 5` command executes to be saved in the `UltiStudentStateList`, and the `currentStatePointer` is shifted to the newly inserted UltiStudent state.

[UndoRedoNewCommand1StateListDiagram] | *UndoRedoNewCommand1StateListDiagram.png*

Step 3. The user executes `add mc/CS2103T …` to add a new homework. The `add` command also calls `Model#commitUltiStudent()`, causing another modified UltiStudent state to be saved into the `UltiStudentStateList`.

[UndoRedoNewCommand2StateListDiagram] | *UndoRedoNewCommand2StateListDiagram.png*

NOTE	If a command fails its execution, it will not call `Model#commitUltiStudent()`, so the UltiStudent state will not be saved into the `ultiStudentStateList`.

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

[UndoRedoExecuteUndoStateListDiagram] | *UndoRedoExecuteUndoStateListDiagram.png*

NOTE	If the `currentStatePointer` is at index 0, pointing to the initial UltiStudent state, then there are no previous UltiStudent states to restore. The `undo` command uses `Model#canUndoUltiStudent()` 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#redoUltiStudent()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the UltiStudent to that state.

NOTE	If the `currentStatePointer` is at index `UltiStudentStateList.size() - 1`, pointing to the latest UltiStudent state, then there are no undone UltiStudent states to restore. The `redo` command uses `Model#canRedoUltiStudent()` 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 UltiStudent, such as `list`, will usually not call `Model#commitUltiStudent()`, `Model#undoUltiStudent()` or `Model#redoUltiStudent()`. Thus, the `UltiStudentStateList` remains unchanged.

[UndoRedoNewCommand3StateListDiagram] | *UndoRedoNewCommand3StateListDiagram.png*

Step 6. The user executes `clear`, which calls `Model#commitUltiStudent()`. Since the `currentStatePointer` is not pointing at the end of the `UltiStudentStateList`, all UltiStudent states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add mc/CS2101 …` 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*

3.1.2. Design Considerations

Aspect: How undo & redo executes

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

3.2. [Proposed] Data Encryption

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

3.3. 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 3.4, “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

3.4. 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

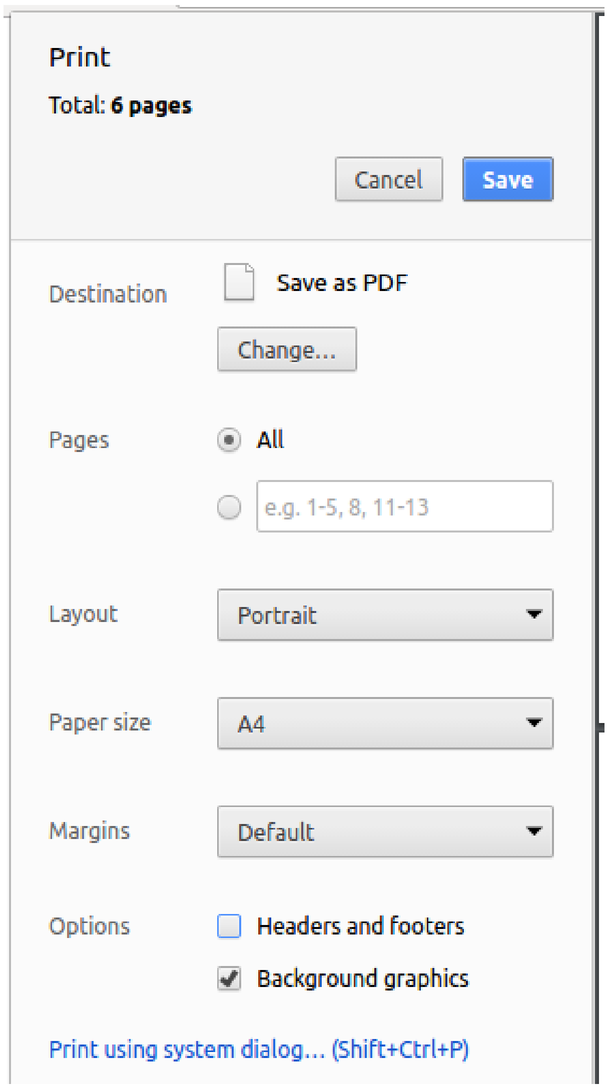
See UsingTravis.adoc to learn how to deploy GitHub Pages using Travis.

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.


Print

Total: **6 pages**

Destination: Save as PDF

Pages: [✓ All]

Layout: Portrait

Paper size: A4

Margins: Default

Options: ☐ Headers and footers ☑ Background graphics

Figure 9. 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.ultistudent.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.ultistudent.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.ultistudent.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.ultistudent.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, UltiStudent 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: Suggested Programming Tasks to Get Started

Suggested path for new programmers:

1. First, add small local-impact (i.e. the impact of the change does not go beyond the component) enhancements to one component at a time. Some suggestions are given in Section A.1, “Improving each component”.
2. Next, add a feature that touches multiple components to learn how to implement an end-to-end feature across all components. Section A.2, “Creating a new command: `remark`” explains how to go about adding such a feature.

A.1. Improving each component

Each individual exercise in this section is component-based (i.e. you would not need to modify the other components to get it to work).

`Logic` component

Scenario: You are in charge of `logic`. During dog-fooding, your team realize that it is troublesome for the user to type the whole command in order to execute a command. Your team devise some strategies to help cut down the amount of typing necessary, and one of the suggestions was to implement aliases for the command words. Your job is to implement such aliases.

TIP Do take a look at Section 2.3, “Logic component” before attempting to modify the `Logic` component.

1. Add a shorthand equivalent alias for each of the individual commands. For example, besides typing `clear`, the user can also type `c` to remove all persons in the list.

- Hints
  - Just like we store each individual command word constant `COMMAND_WORD` inside `*Command.java` (e.g. `FindCommand#COMMAND_WORD`, `DeleteCommand#COMMAND_WORD`), you need a new constant for aliases as well (e.g. `FindCommand#COMMAND_ALIAS`).
  - `AddressBookParser` is responsible for analyzing command words.
- Solution
  - Modify the switch statement in `AddressBookParser#parseCommand(String)` such that both the proper command word and alias can be used to execute the same intended command.
  - Add new tests for each of the aliases that you have added.
  - Update the user guide to document the new aliases.
  - See this PR for the full solution.

`Model` component

Scenario: You are in charge of `model`. One day, the `logic`-in-charge approaches you for help. He wants to implement a command such that the user is able to remove a particular tag from everyone in the UltiStudent, but the model API does not support such a functionality at the moment. Your job is to implement an API method, so that your teammate can use your API to implement his command.

TIP Do take a look at Section 2.4, “Model component” before attempting to modify the `Model` component.

1. Add a `removeTag(Tag)` method. The specified tag will be removed from everyone in the UltiStudent.

- Hints
  - The `Model` and the `AddressBook` API need to be updated.
  - Think about how you can use SLAP to design the method. Where should we place the main logic of deleting tags?
  - Find out which of the existing API methods in `AddressBook` and `Person` classes can be used to implement the tag removal logic. `AddressBook` allows you to update a person, and `Person` allows you to update the tags.
- Solution
  - Implement a `removeTag(Tag)` method in `AddressBook`. Loop through each person, and remove the `tag` from each person.
  - Add a new API method `deleteTag(Tag)` in `ModelManager`. Your `ModelManager` should call `AddressBook#removeTag(Tag)`.
  - Add new tests for each of the new public methods that you have added.
  - See this PR for the full solution.

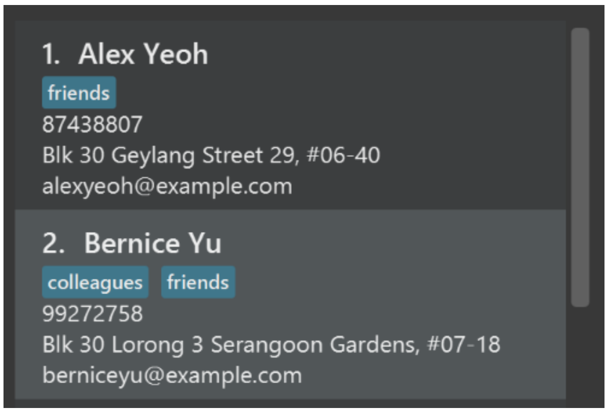
`Ui` component

Scenario: You are in charge of `ui`. During a beta testing session, your team is observing how the users use your UltiStudent application. You realize that one of the users occasionally tries to delete non-existent tags from a contact, because the tags all look the same visually, and the user got confused. Another user made a typing mistake in his command, but did not realize he had done so because the error message wasn't prominent enough. A third user keeps scrolling down the list, because he keeps forgetting the index of the last person in the list. Your job is to implement improvements to the UI to solve all these problems.

TIP Do take a look at Section 2.2, "UI component" before attempting to modify the `UI` component.

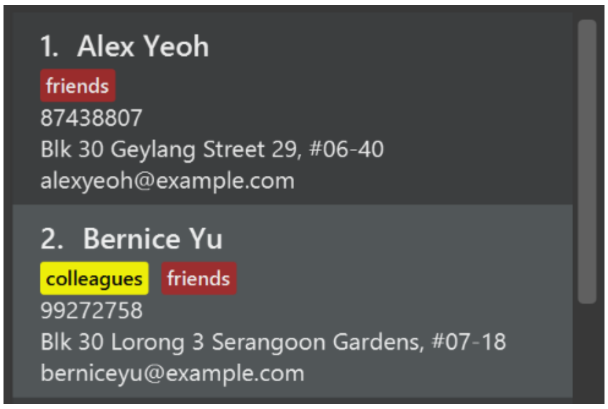
1. Use different colors for different tags inside person cards. For example, `friends` tags can be all in brown, and `colleagues` tags can be all in yellow.

   Before


1. Alex Yeoh
friends
87438807
Blk 30 Geylang Street 29, #06-40
alexyeoh@example.com

2. Bernice Yu
colleagues friends
99272758
Blk 30 Lorong 3 Serangoon Gardens, #07-18
berniceyu@example.com

After

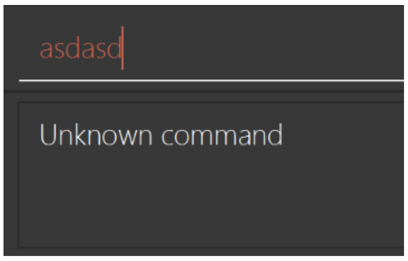

1. Alex Yeoh
friends
87438807
Blk 30 Geylang Street 29, #06-40
alexyeoh@example.com

2. Bernice Yu
colleagues friends
99272758
Blk 30 Lorong 3 Serangoon Gardens, #07-18
berniceyu@example.com

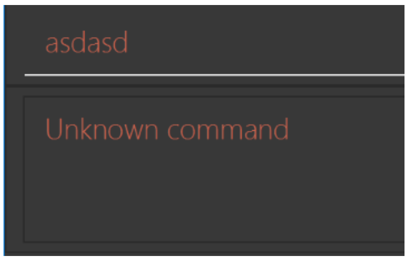
- Hints
  - The tag labels are created inside the `PersonCard` constructor (`new Label(tag.tagName)`). JavaFX's `Label` class allows you to modify the style of each Label, such as changing its color.
  - Use the .css attribute `-fx-background-color` to add a color.
  - You may wish to modify `DarkTheme.css` to include some pre-defined colors using css, especially if you have experience with web-based css.
- Solution
  - You can modify the existing test methods for `PersonCard` 's to include testing the tag's color as well.
  - See this PR for the full solution.
    - The PR uses the hash code of the tag names to generate a color. This is deliberately designed to ensure consistent colors each time the application runs. You may wish to expand on this design to include additional features, such as allowing users to set their own tag colors, and directly saving the colors to storage, so that tags retain their colors even if the hash code algorithm changes.

2. Modify `NewResultAvailableEvent` such that `ResultDisplay` can show a different style on error (currently it shows the same regardless of errors).

Before


```
asdasd
```
```
Unknown command
```

After



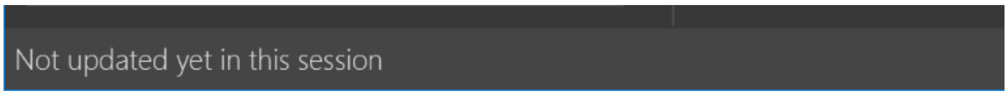
asdasd

Unknown command

- Hints
  - `NewResultAvailableEvent` is raised by `CommandBox` which also knows whether the result is a success or failure, and is caught by `ResultDisplay` which is where we want to change the style to.
  - Refer to `CommandBox` for an example on how to display an error.
- Solution
  - Modify `NewResultAvailableEvent` 's constructor so that users of the event can indicate whether an error has occurred.
  - Modify `ResultDisplay#handleNewResultAvailableEvent(NewResultAvailableEvent)` to react to this event appropriately.
  - You can write two different kinds of tests to ensure that the functionality works:
    - The unit tests for `ResultDisplay` can be modified to include verification of the color.
    - The system tests `AddressBookSystemTest#assertCommandBoxShowsDefaultStyle()` and `AddressBookSystemTest#assertCommandBoxShowsErrorStyle()` to include verification for `ResultDisplay` as well.
  - See this PR for the full solution.
    - Do read the commits one at a time if you feel overwhelmed.

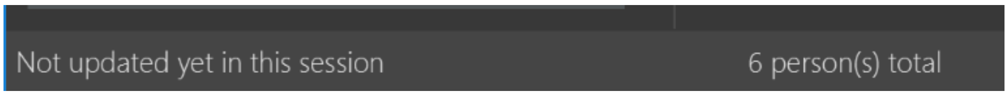
3. Modify the `StatusBarFooter` to show the total number of people in the UltiStudent.

Before



Not updated yet in this session

After



Not updated yet in this session 6 person(s) total

- Hints
    - `StatusBarFooter.fxml` will need a new `StatusBar`. Be sure to set the `GridPane.columnIndex` properly for each `StatusBar` to avoid misalignment!
    - `StatusBarFooter` needs to initialize the status bar on application start, and to update it accordingly whenever the UltiStudent is updated.
- Solution
    - Modify the constructor of `StatusBarFooter` to take in the number of persons when the application just started.
    - Use `StatusBarFooter#handleAddressBookChangedEvent(AddressBookChangedEvent)` to update the number of persons whenever there are new changes to the addressbook.
    - For tests, modify `StatusBarFooterHandle` by adding a state-saving functionality for the total number of people status, just like what we did for save location and sync status.
    - For system tests, modify `AddressBookSystemTest` to also verify the new total number of persons status bar.
    - See this PR for the full solution.

`Storage` component

Scenario: You are in charge of `storage`. For your next project milestone, your team plans to implement a new feature of saving the UltiStudent to the cloud. However, the current implementation of the application constantly saves the UltiStudent after the execution of each command, which is not ideal if the user is working on limited internet connection. Your team decided that the application should instead save the changes to a temporary local backup file first, and only upload to the cloud after the user closes the application. Your job is to implement a backup API for the UltiStudent storage.

TIP Do take a look at Section 2.5, “Storage component” before attempting to modify the `Storage` component.

1. Add a new method `backupAddressBook(ReadOnlyAddressBook)`, so that the UltiStudent can be saved in a fixed temporary location.

    - Hint
        - Add the API method in `AddressBookStorage` interface.
        - Implement the logic in `StorageManager` and `JsonAddressBookStorage` class.
    - Solution
        - See this PR for the full solution.

A.2. Creating a new command: `remark`

By creating this command, you will get a chance to learn how to implement a feature end-to-end, touching all major components of the app.

Scenario: You are a software maintainer for `addressbook`, as the former developer team has moved on to new projects. The current users of your application have a list of new feature requests that they hope the software will eventually have. The most popular request is to allow adding additional comments/notes about a particular contact, by providing a flexible `remark` field for each contact, rather than relying on tags alone. After designing the specification for the `remark` command, you are convinced that this feature is worth implementing. Your job is to implement the `remark` command.

A.2.1. Description

Edits the remark for a person specified in the `INDEX`.
Format: `remark INDEX r/[REMARK]`

Examples:

- `remark 1 r/Likes to drink coffee.`
  Edits the remark for the first person to `Likes to drink coffee.`
- `remark 1 r/`
  Removes the remark for the first person.

A.2.2. Step-by-step Instructions

[Step 1] Logic: Teach the app to accept 'remark' which does nothing

Let's start by teaching the application how to parse a `remark` command. We will add the logic of `remark` later.

Main:

1. Add a `RemarkCommand` that extends `Command`. Upon execution, it should just throw an `Exception`.
2. Modify `AddressBookParser` to accept a `RemarkCommand`.

Tests:

1. Add `RemarkCommandTest` that tests that `execute()` throws an Exception.
2. Add new test method to `AddressBookParserTest`, which tests that typing "remark" returns an instance of `RemarkCommand`.

[Step 2] Logic: Teach the app to accept 'remark' arguments

Let's teach the application to parse arguments that our `remark` command will accept. E.g. `1 r/Likes to drink coffee.`

Main:

1. Modify `RemarkCommand` to take in an `Index` and `String` and print those two parameters as the error message.
2. Add `RemarkCommandParser` that knows how to parse two arguments, one index and one with prefix 'r/'.
3. Modify `AddressBookParser` to use the newly implemented `RemarkCommandParser`.

Tests:

1. Modify `RemarkCommandTest` to test the `RemarkCommand#equals()` method.
2. Add `RemarkCommandParserTest` that tests different boundary values for `RemarkCommandParser`.
3. Modify `AddressBookParserTest` to test that the correct command is generated according to the user input.

[Step 3] Ui: Add a placeholder for remark in `PersonCard`

Let's add a placeholder on all our `PersonCard` s to display a remark for each person later.

Main:

1. Add a `Label` with any random text inside `PersonListCard.fxml`.
2. Add FXML annotation in `PersonCard` to tie the variable to the actual label.

Tests:

1. Modify `PersonCardHandle` so that future tests can read the contents of the remark label.

[Step 4] Model: Add `Remark` class

We have to properly encapsulate the remark in our `Person` class. Instead of just using a `String`, let's follow the conventional class structure that the codebase already uses by adding a `Remark` class.

Main:

1. Add `Remark` to model component (you can copy from `Address`, remove the regex and change the names accordingly).
2. Modify `RemarkCommand` to now take in a `Remark` instead of a `String`.

Tests:

1. Add test for `Remark`, to test the `Remark#equals()` method.

[Step 5] Model: Modify `Person` to support a `Remark` field

Now we have the `Remark` class, we need to actually use it inside `Person`.

Main:

1. Add `getRemark()` in `Person`.
2. You may assume that the user will not be able to use the `add` and `edit` commands to modify the

remarks field (i.e. the person will be created without a remark).

3. Modify `SampleDataUtil` to add remarks for the sample data (delete your `data/addressbook.json` so that the application will load the sample data when you launch it.)

[Step 6] Storage: Add `Remark` field to `JsonAdaptedPerson` class

We now have `Remark` s for `Person` s, but they will be gone when we exit the application. Let's modify `JsonAdaptedPerson` to include a `Remark` field so that it will be saved.

Main:

1. Add a new JSON field for `Remark`.

Tests:

1. Fix `invalidAndValidPersonAddressBook.json`, `typicalPersonsAddressBook.json`, `validAddressBook.json` etc., such that the JSON tests will not fail due to a missing `remark` field.

[Step 6b] Test: Add withRemark() for `PersonBuilder`

Since `Person` can now have a `Remark`, we should add a helper method to `PersonBuilder`, so that users are able to create remarks when building a `Person`.

Tests:

1. Add a new method `withRemark()` for `PersonBuilder`. This method will create a new `Remark` for the person that it is currently building.
2. Try and use the method on any sample `Person` in `TypicalPersons`.

[Step 7] Ui: Connect `Remark` field to `PersonCard`

Our remark label in `PersonCard` is still a placeholder. Let's bring it to life by binding it with the actual `remark` field.

Main:

1. Modify `PersonCard`'s constructor to bind the `Remark` field to the `Person` 's remark.

Tests:

1. Modify `GuiTestAssert#assertCardDisplaysPerson(…)` so that it will compare the now-functioning remark label.

[Step 8] Logic: Implement `RemarkCommand#execute()` logic

We now have everything set up… but we still can't modify the remarks. Let's finish it up by adding in actual logic for our `remark` command.

Main:

1. Replace the logic in `RemarkCommand#execute()` (that currently just throws an `Exception`), with the

actual logic to modify the remarks of a person.

Tests:

1. Update `RemarkCommandTest` to test that the `execute()` logic works.

A.2.3. Full Solution

See this PR for the step-by-step solution.

Appendix B: Product Scope

Target user profile:

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

Value proposition: manage contacts faster than a typical mouse/GUI driven app

Appendix C: 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
`* * *`	user	add a new person	
`* * *`	user	delete a person	remove entries that I no longer need
`* * *`	user	find a person by name	locate details of persons without having to go through the entire list

Priority	As a …	I want to …	So that I can…
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
*	user with many persons in the UltiStudent	sort persons by name	locate a person easily

{More to be added}

Appendix D: Use Cases

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

Use case: Delete person

MSS

1. User requests to list persons
2. AddressBook shows a list of persons
3. User requests to delete a specific person in the list
4. AddressBook deletes the person

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. AddressBook shows an error message.

      Use case resumes at step 2.

{More to be added}

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

{More to be added}

Appendix F: Glossary

Mainstream OS
Windows, Linux, Unix, OS-X

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

Appendix G: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix H: 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.

H.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 … }

H.2. Deleting a person

1. Deleting a person while all persons are listed

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

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

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

{ more test cases … }

H.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 … }