NUSCouples - Developer Guide

Table of Contents

By: `Team T09-B4` Since: `Feb 2018` Licence: `MIT`

Welcome to the developer guide for *NUSCouples*!

NUSCouples is a command-line desktop application targeted at couples studying at the National University of Singapore (NUS). It aims to help these couples create and remember new memories during their time in NUS.

This developer guide contains information that can help you to get started as a contributor to *NUSCouples*, as well as more about the software architecture and feature implementation of *NUSCouples*.

1. Setting up

1.1. Prerequisites

1. **JDK** `1.8.0_60` or later

NOTE	Having any Java 8 version is not enough. This app will not work with earlier versions of Java 8.

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.

1.3. Verifying the setup

1. Run the `seedu.address.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. Updating documentation to match your fork

After forking the repo, links in the documentation will still point to the `se-edu/addressbook-level4` repo. If you plan to develop this as a separate product (i.e. instead of contributing to the `se-edu/addressbook-level4`) , you should replace the URL in the variable `repoURL` in `DeveloperGuide.adoc` and `UserGuide.adoc` with the URL of your fork.

1.4.3. 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)

1.4.4. 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 [GetStartedProgramming].

Return to Table of Contents

2. Design

2.1. Architecture

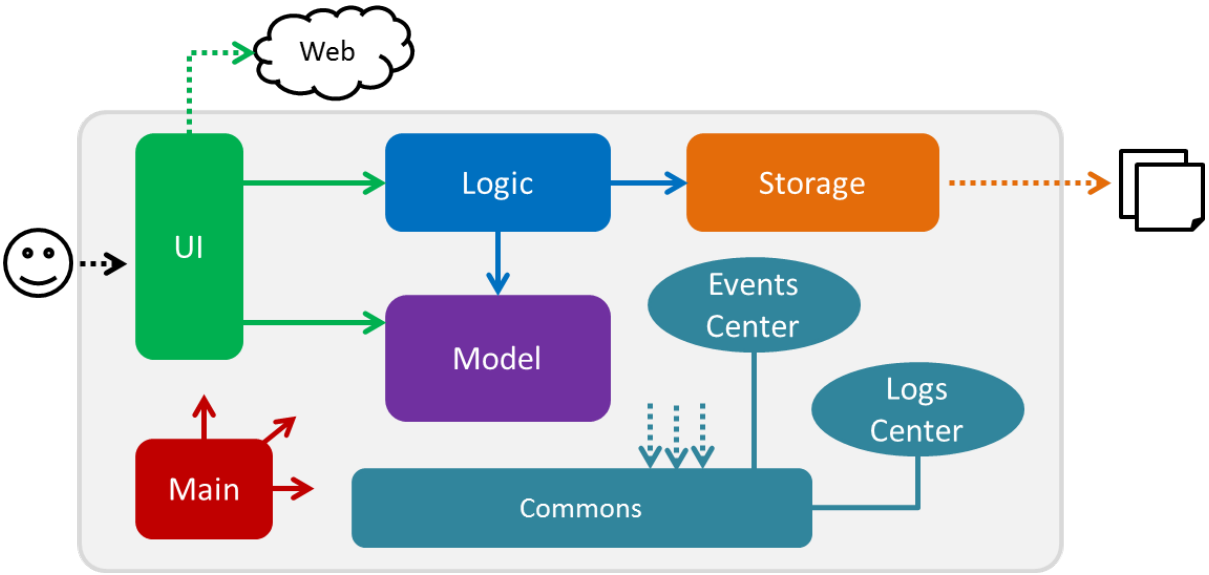


Figure 1. Architecture Diagram

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. Two of those classes play important roles at the architecture level.

- `EventsCenter` : This class (written using Google's Event Bus library) is used by components to communicate with other components using events (i.e. a form of *Event Driven* design)
- `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

Events-Driven nature of the design

The *Sequence Diagram* below shows how the components interact for the scenario where the user issues the command `delete 1`.

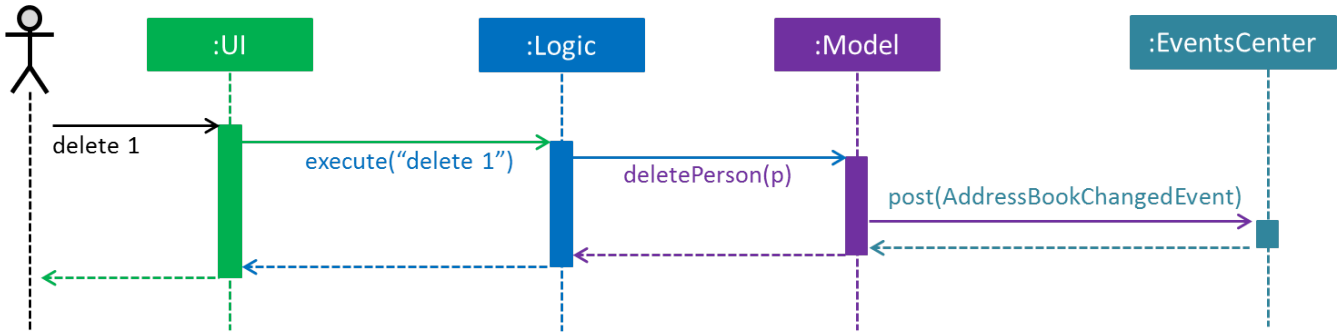


Figure 3. Component interactions for `delete 1` command (part 1)

NOTE Note how the `Model` simply raises a `AddressBookChangedEvent` when the Address Book data are changed, instead of asking the `Storage` to save the updates to the hard disk.

The diagram below shows how the `EventsCenter` reacts to that event, which eventually results in the updates being saved to the hard disk and the status bar of the UI being updated to reflect the 'Last Updated' time.

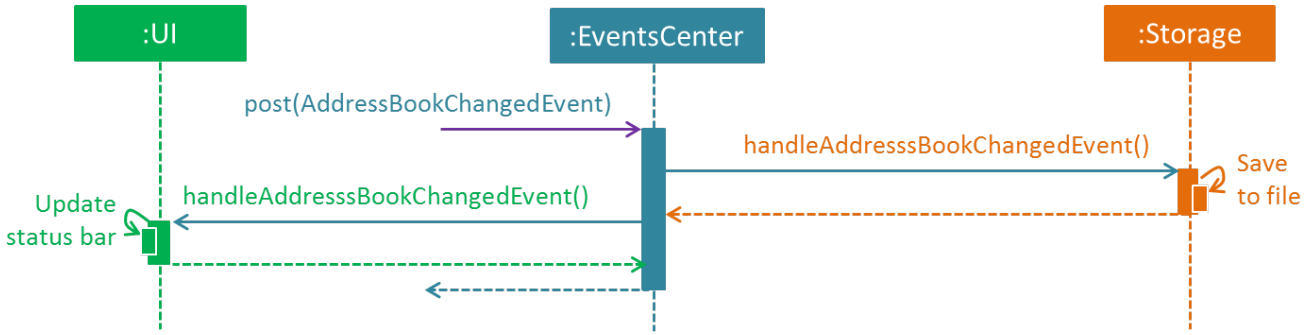


Figure 4. Component interactions for `delete 1` command (part 2)

NOTE	Note how the event is propagated through the `EventsCenter` to the `Storage` and `UI` without `Model` having to be coupled to either of them. This is an example of how this Event Driven approach helps us reduce direct coupling between components.

The sections below give more details of each component.

Return to Table of Contents

2.2. UI component

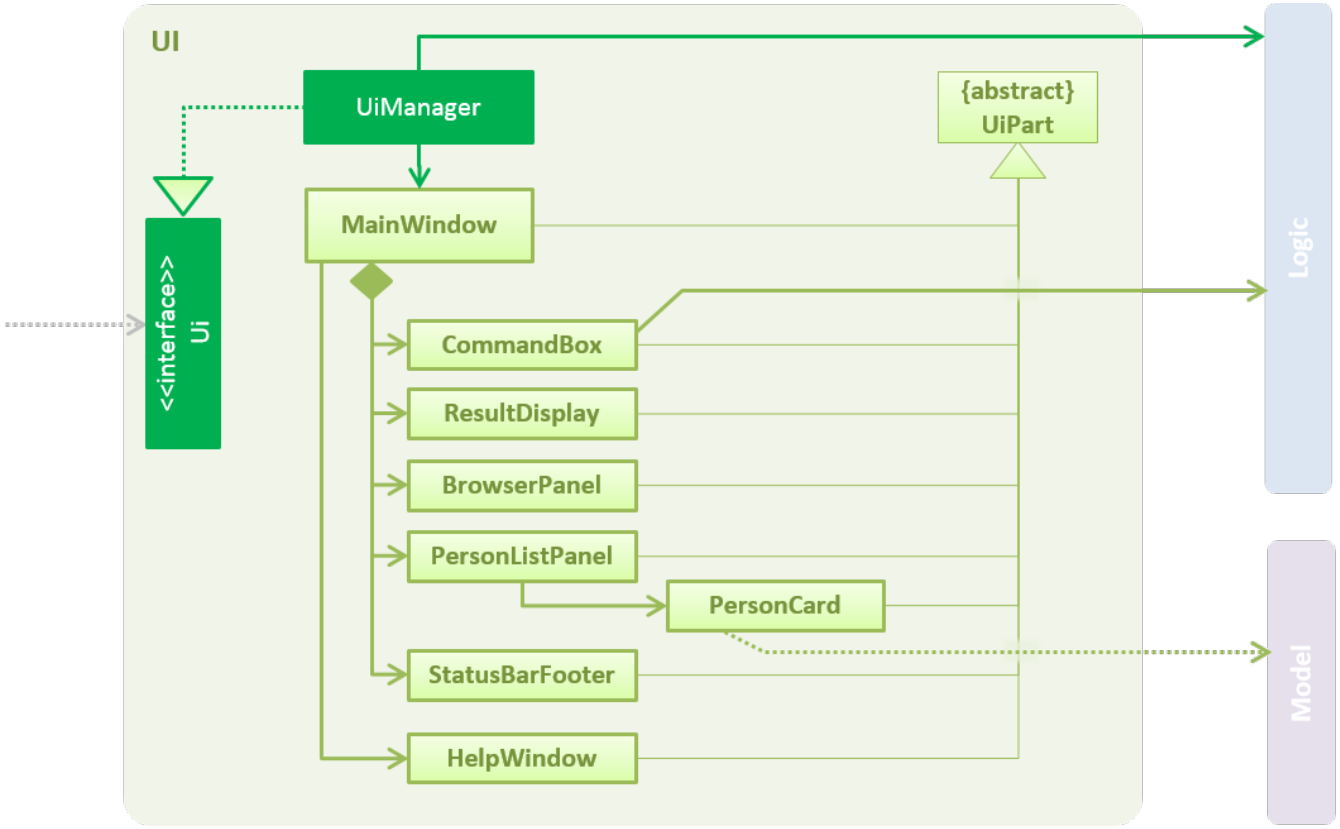


Figure 5. Structure of the UI Component

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.
- Binds itself to some data in the `Model` so that the UI can auto-update when data in the `Model` change.
- Responds to events raised from various parts of the App and updates the UI accordingly.

Return to Table of Contents

2.3. Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 6. Structure of the Logic Component

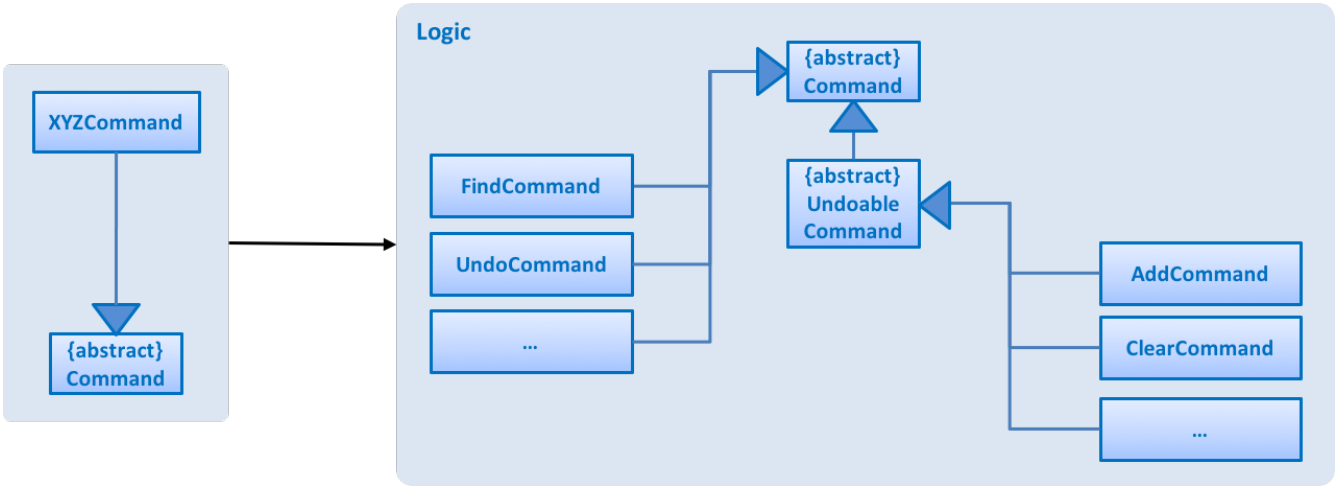


Figure 7. Structure of Commands in the Logic Component. This diagram shows finer details concerning `XYZCommand` and `Command` in Figure 6, "Structure of the Logic Component"

API : `Logic.java`

1. `Logic` uses the `AddressBookParser` 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 person) and/or raise events.
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.

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

[DeletePersonSdForLogic] | *DeletePersonSdForLogic.png*

Figure 8. Interactions Inside the Logic Component for the `delete 1` Command

Return to Table of Contents

2.4. Model component

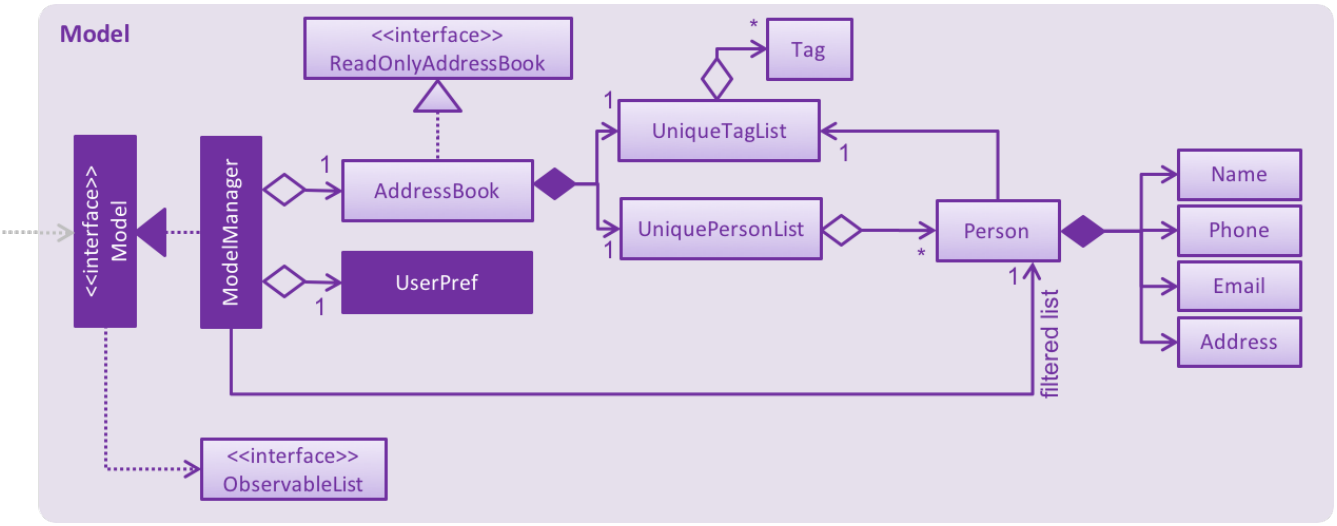


Figure 9. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` 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.

Return to Table of Contents

2.5. Storage component

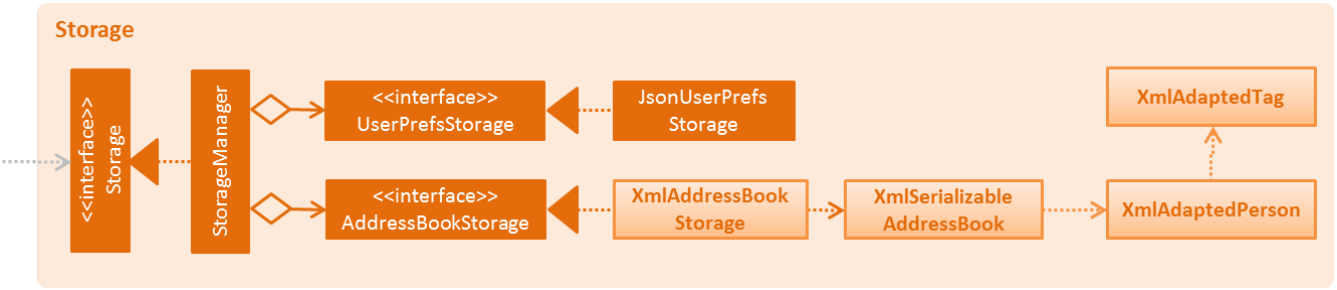


Figure 10. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

- can save the Address Book data in xml format and read it back.

Return to Table of Contents

2.6. Common classes

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

3. Implementation

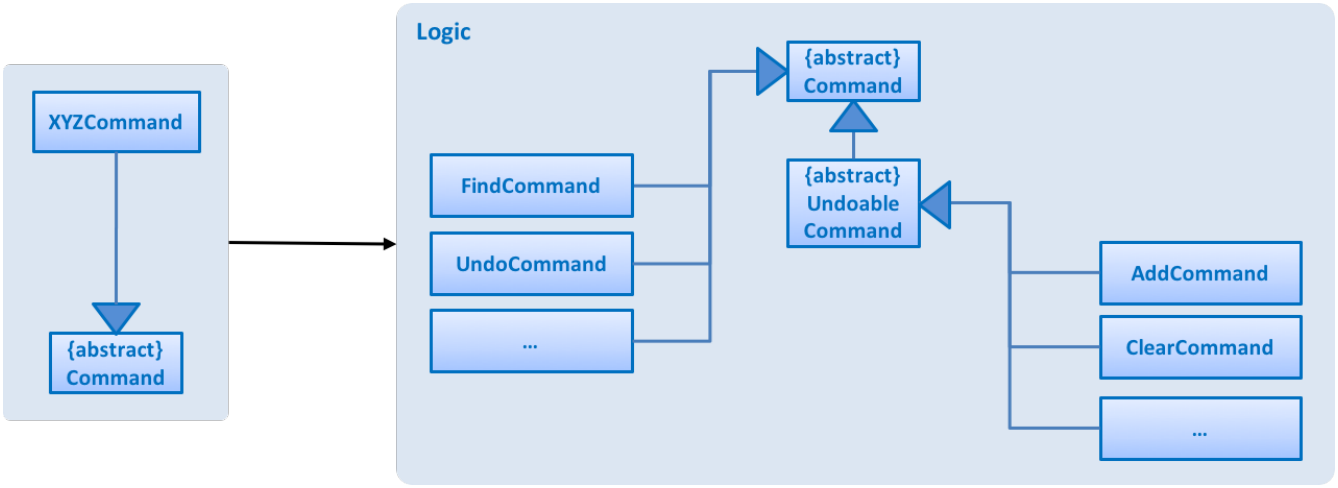
This section describes some noteworthy details on how certain features are implemented.

3.1. Undo/Redo feature

3.1.1. Current Implementation

The undo/redo mechanism is facilitated by an `UndoRedoStack`, which resides inside `LogicManager`. It supports undoing and redoing of commands that modifies the state of the address book (e.g. `add`, `edit`). Such commands will inherit from `UndoableCommand`.

`UndoRedoStack` only deals with `UndoableCommands`. Commands that cannot be undone will inherit from `Command` instead. The following diagram shows the inheritance diagram for commands:



As you can see from the diagram, `UndoableCommand` adds an extra layer between the abstract `Command` class and concrete commands that can be undone, such as the `DeleteCommand`. Note that extra tasks need to be done when executing a command in an *undoable* way, such as saving the state of the address book before execution. `UndoableCommand` contains the high-level algorithm for those extra tasks while the child classes implements the details of how to execute the specific command. Note that this technique of putting the high-level algorithm in the parent class and lower-level steps of the algorithm in child classes is also known as the template pattern.

Commands that are not undoable are implemented this way:

```
public class ListCommand extends Command {
    @Override
    public CommandResult execute() {
        // ... list logic ...
    }
}
```

With the extra layer, the commands that are undoable are implemented this way:

```
public abstract class UndoableCommand extends Command {
    @Override
    public CommandResult execute() {
        // ... undo logic ...

        executeUndoableCommand();
    }
}

public class DeleteCommand extends UndoableCommand {
    @Override
    public CommandResult executeUndoableCommand() {
        // ... delete logic ...
    }
}
```

Suppose that the user has just launched the application. The `UndoRedoStack` will be empty at the beginning.

The user executes a new `UndoableCommand`, `delete 5`, to delete the 5th person in the address book. The current state of the address book is saved before the `delete 5` command executes. The `delete 5` command will then be pushed onto the `undoStack` (the current state is saved together with the command).

[UndoRedoStartingStackDiagram] | *UndoRedoStartingStackDiagram.png*

As the user continues to use the program, more commands are added into the `undoStack`. For example, the user may execute `add n/David …` to add a new person.

[UndoRedoNewCommand1StackDiagram] | *UndoRedoNewCommand1StackDiagram.png*

> **NOTE** If a command fails its execution, it will not be pushed to the `UndoRedoStack` at all.

The user now decides that adding the person was a mistake, and decides to undo that action using `undo`.

We will pop the most recent command out of the `undoStack` and push it back to the `redoStack`. We will restore the address book to the state before the `add` command executed.

[UndoRedoExecuteUndoStackDiagram] | *UndoRedoExecuteUndoStackDiagram.png*

> **NOTE** If the `undoStack` is empty, then there are no other commands left to be undone, and an `Exception` will be thrown when popping the `undoStack`.

The following sequence diagram shows how the undo operation works:

[UndoRedoSequenceDiagram] | *UndoRedoSequenceDiagram.png*

The redo does the exact opposite (pops from `redoStack`, push to `undoStack`, and restores the address book to the state after the command is executed).

> **NOTE** If the `redoStack` is empty, then there are no other commands left to be redone, and an `Exception` will be thrown when popping the `redoStack`.

The user now decides to execute a new command, `clear`. As before, `clear` will be pushed into the `undoStack`. This time the `redoStack` is no longer empty. It will be purged as it no longer make sense to redo the `add n/David` command (this is the behavior that most modern desktop applications follow).

[UndoRedoNewCommand2StackDiagram] | *UndoRedoNewCommand2StackDiagram.png*

Commands that are not undoable are not added into the `undoStack`. For example, `list`, which inherits from `Command` rather than `UndoableCommand`, will not be added after execution:

[UndoRedoNewCommand3StackDiagram] | *UndoRedoNewCommand3StackDiagram.png*

The following activity diagram summarize what happens inside the `UndoRedoStack` when a user executes a new command:

[UndoRedoActivityDiagram] | *UndoRedoActivityDiagram.png*

3.1.2. Design Considerations

Aspect: Implementation of `UndoableCommand`

- **Alternative 1 (current choice):** Add a new abstract method `executeUndoableCommand()`
  - Pros: We will not lose any undone/redone functionality as it is now part of the default behaviour. Classes that deal with `Command` do not have to know that `executeUndoableCommand()` exist.
  - Cons: Hard for new developers to understand the template pattern.
- **Alternative 2:** Just override `execute()`
  - Pros: Does not involve the template pattern, easier for new developers to understand.
  - Cons: Classes that inherit from `UndoableCommand` must remember to call `super.execute()`, or lose the ability to undo/redo.

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address 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 `delete`, just save the person being deleted).
    - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Type of commands that can be undone/redone

- **Alternative 1 (current choice):** Only include commands that modifies the address book (`add`, `clear`, `edit`).
    - Pros: We only revert changes that are hard to change back (the view can easily be re-modified as no data are * lost).
    - Cons: User might think that undo also applies when the list is modified (undoing filtering for example), * only to realize that it does not do that, after executing `undo`.
- **Alternative 2:** Include all commands.
    - Pros: Might be more intuitive for the user.
    - Cons: User have no way of skipping such commands if he or she just want to reset the state of the address * book and not the view. **Additional Info:** See our discussion here.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use separate stack for undo and redo
    - Pros: Easy to understand 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 `UndoRedoStack`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
    - Pros: We do not need to maintain a separate stack, 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.

Return to Table of Contents

3.2. [Proposed] Data Encryption

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

Return to Table of Contents

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 [Implementation-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

Return to Table of Contents

3.4. Configuration

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

Return to Table of Contents

3.5. [Proposed] Calendar Google API Feature

3.5.1. Proposed Implementation (Appearing in V2.0)

The Calendar Viewer mechanism is facilitated by `Google Calendar API` and reside in the `ModelManager`. It supports viewing/add/editing/deleting capability that modifies the state of *NUSCouples*. Firstly, it uses OAuth 2.0 endpoints to allow users to share specific data with the application while keeping their usernames, passwords, and other information private. For example, an application can use OAuth 2.0 to obtain permission from users to store files in their Google Drives which sync to the calendar. This implementation requires the user to connect to the internet because *NUSCouples* needs to open the system browser and supply a local redirect URI to handle responses from Google's authorization server.

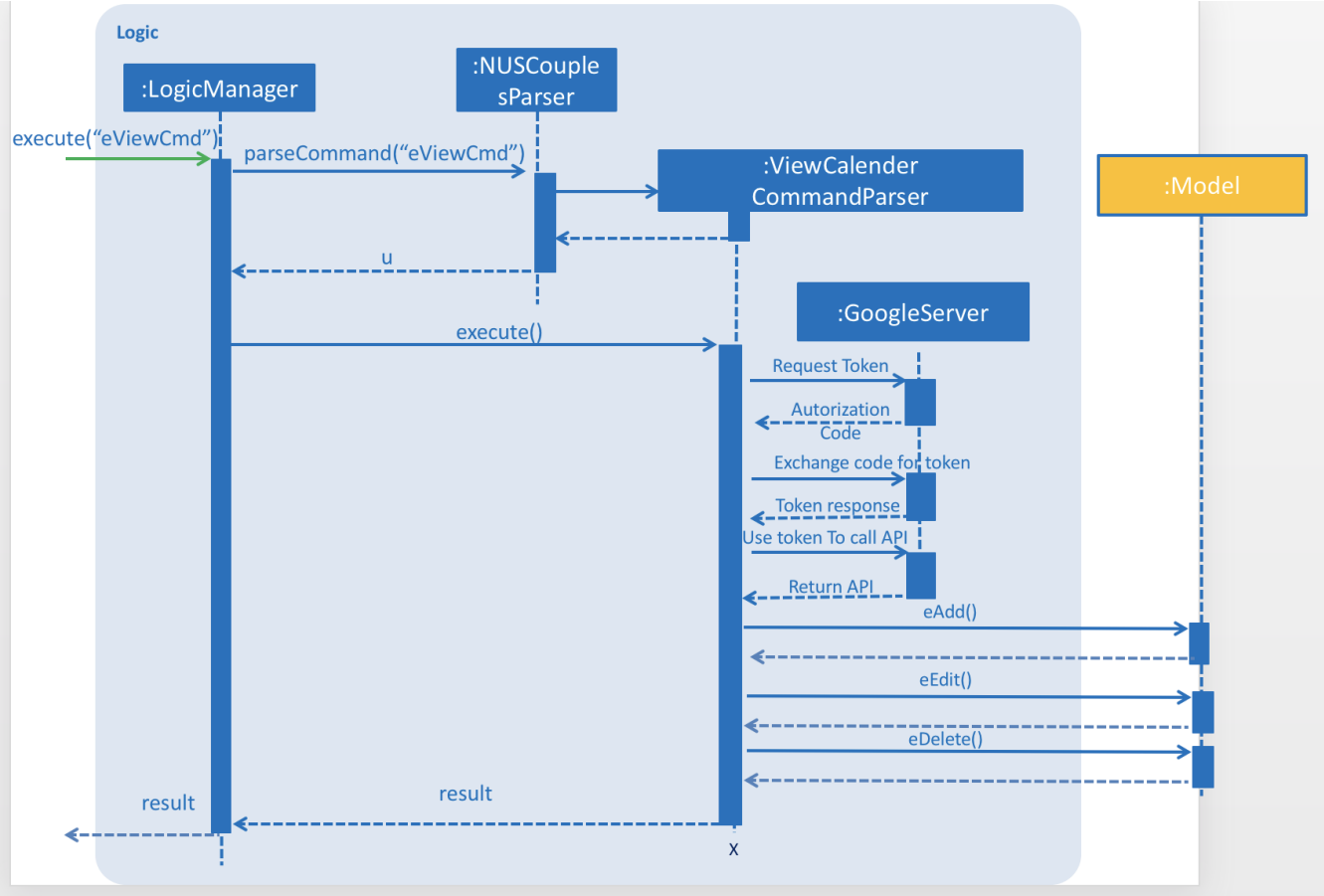
Basic steps

All applications follow a basic pattern when accessing a Google API using OAuth 2.0. At a high level, this are the four steps:

No.	Description
Step 1:	Obtain OAuth 2.0 credentials from the Google API Console. Visit the Google API Console to obtain OAuth 2.0 credentials such as a client ID and client secret that are known to both Google and your application. The set of values varies based on what type of application you are building. For example, a JavaScript application does not require a secret, but a web server application does.
Step 2:	Obtain an access token from the Google Authorization Server. Before your application can access private data using a Google API, it must obtain an access token that grants access to that API. A single access token can grant varying degrees of access to multiple APIs. A variable parameter called scope controls the set of resources and operations that an access token permits. During the access-token request, your application sends one or more values in the scope parameter. There are several ways to make this request, and they vary based on the type of application you are building. For example, a JavaScript application might request an access token using a browser redirect to Google, while an application installed on a device that has no browser uses web service requests. Some requests require an authentication step where the user logs in with their Google account. After logging in, the user is asked whether they are willing to grant the permissions that your application is requesting. This process is called user consent. If the user grants the permission, the Google Authorization Server sends your application an access token (or an authorization code that your application can use to obtain an access token). If the user does not grant the permission, the server returns an error. It is generally a best practice to request scopes incrementally, at the time access is required, rather than up front. For example, an app that wants to support purchases should not request Google Wallet access until the user presses the “buy” button; see Incremental authorization.

No.	Description
Step 3:	After an application obtains an access token, it sends the token to a Google API in an HTTP authorization header. It is possible to send tokens as URI query-string parameters, but we don't recommend it, because URI parameters can end up in log files that are not completely secure. Also, it is good REST practice to avoid creating unnecessary URI parameter names. Access tokens are valid only for the set of operations and resources described in the scope of the token request. For example, if an access token is issued for the Google+ API, it does not grant access to the Google Contacts API. You can, however, send that access token to the Google+ API multiple times for similar operations.
Step 4:	Refresh the access token, if necessary. Access tokens have limited lifetimes. If your application needs access to a Google API beyond the lifetime of a single access token, it can obtain a refresh token. A refresh token allows your application to obtain new access tokens

The sequence diagram below shows interactions within the `Logic` Component for Outh 2.0 endpoints:



3.5.2. Design Considerations

Aspect: Implementation of Google Calendar feature

- **Alternative 1 (current choice):** Data are not save locally.
  - Pros: User does not worry about getting data lost
  - Cons: User unable to retrieve the Calendar if internet is not connected
- **Alternative 2:** Save Data locally
  - Pros: User does not worry their Calendar is unable to connected to Google.
  - Cons: The latest Calendar events might not have been synchronized.

Aspect: Using Open-source or proprietary Calendar API

- **Alternative 1 (current choice):** Using Google API (open source)
  - Pros: I will learnt more even if I failed at the end of the project and Google API is more versatile
  - Cons: Tedious to implement it.
- **Alternative 2:** Using Restful API (proprietary)
  - Pros: Easier to implement due to everything is assisted.
  - Cons: Restrictive, need more money for more features to add on.

Return to Table of Contents

3.6. Calendar Viewer feature

3.6.1. Proposed Implementation

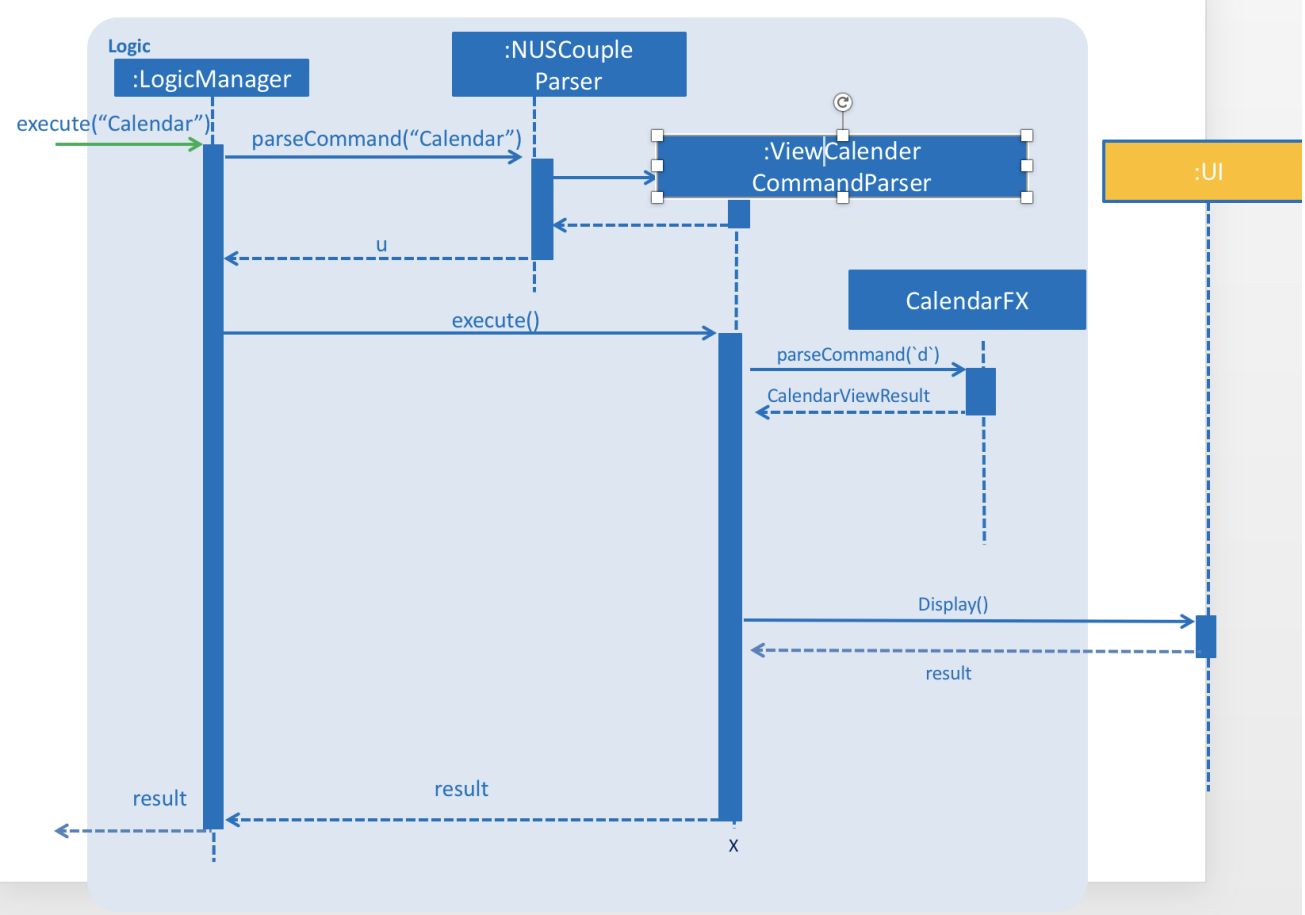
The Calendar Viewer feature is implemented by 'BrowserPanel', which will reside in 'UI'.

The idea of implementing this Calendar Viewer feature that sits ontop of Browser panel UI is to give the user a first look at the upcoming events first before going on to other features on the App.

Users are able to select different views such as in Days, Weeks, Months or Years by adding a prefix 'c', 'w', 'm', 'y' after adding 'Cal' or 'Calendar' behind.

This Calendar Interface is created and designed by CalendarFX. Through their GUI interface, i manipulate and massage the codes to allow user to enter commands to change the views since this is a CLI interface APP.

The sequence diagram below shows interactions within the `Logic` Component for the `execute' API call.



3.6.2. Design Considerations

Aspect: Implementation of Calendar View

- **Alternative 1 (current choice):** Display only current month event.
  - Pros: Easier to implement and Neater rather than displaying more than 1 mth.
  - Cons: Need to input cmd to filter through other month.
- **Alternative 2:** Don't display any month until user defines.
  - Pros: More interaction.
  - Cons: The UI will be blank at initial stage which is ugly.

Aspect: Calendar View performance

- **Alternative 1 (current choice):** Requires RAM of at least 6GB and above.
  - Pros: Faster retrieval and display the Calendar out
  - Cons: Need to buy more memory
- **Alternative 2:** Don't display Year.
  - Pros: Reduce latency
  - Cons: User can't add event to next year

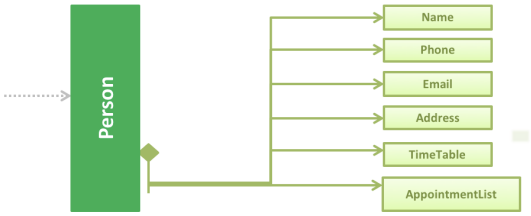
3.7. Calendar Add Appointment feature

3.7.1. Proposed Implementation

The Calendar Add Appointment feature is implemented by 'Appointment', which will reside in 'Model'.

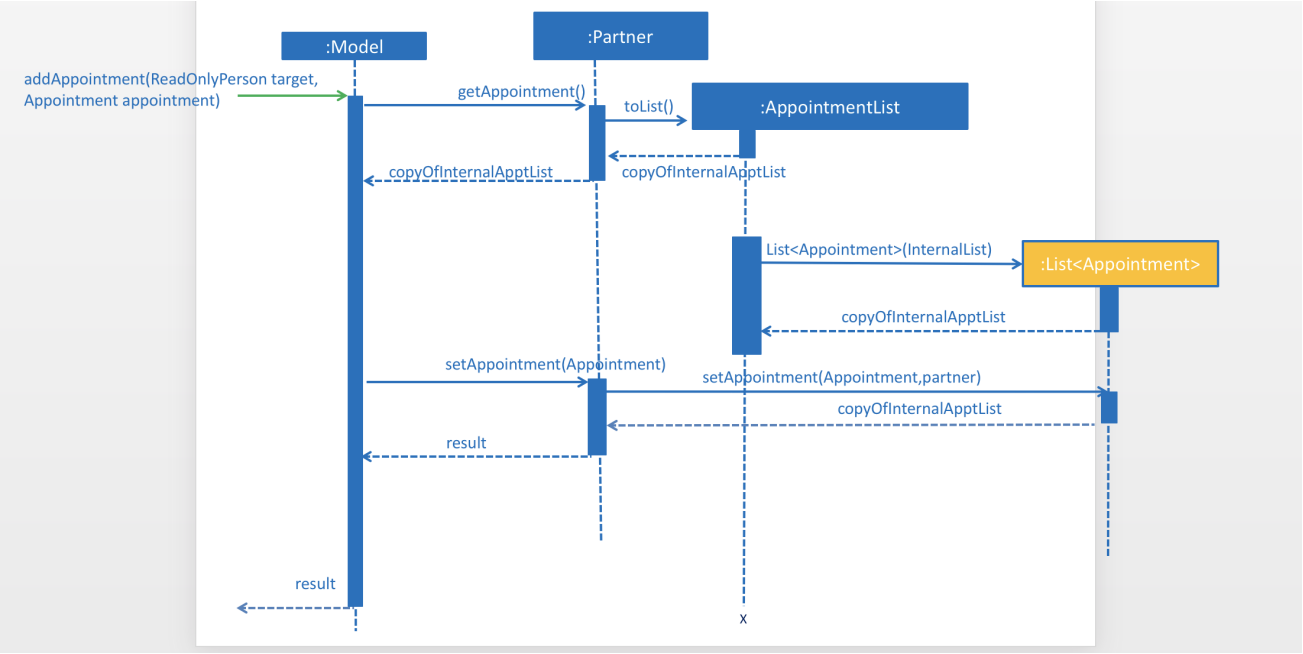
The idea of implementing Add Appointment feature is to allow the user to add his/her event on the Calendar.

The AppointmentList class holds an internal list that holds the Appointment class. Appointments are made up of 3 internal variables.



- Description: It Holds a string about the appointment. This String will be used to identify which Appointment.
- Start Date: It Holds the starting time of the appointment. Used for sorting appointments and UI.
- End Date: Holds the end time of the appointment. Used for UI.

The sequence diagram below shows interactions within the `Model` Component for the `AddAppointment' API call.



3.7.2. Design Considerations

Aspect: Implementation of Add Event View

- **Alternative 1 (current choice):** Display in Calendar UI.
    - Pros: Easier to implement and Neater.
    - Cons: Doesn't display all events listed on the partner.
- **Alternative 2:** Create a List to display all events.
    - Pros: User can have an overview of all the events listed
    - Cons: The UI will be blank at initial stage which is ugly.

Aspect: Allow duplicate events

- **Alternative 1 (current choice):** On same time of the same day.
    - Pros: User can plan which one is more important to attend
    - Cons: Ambiguous as there are multiple similar events on the same time frame
- **Alternative 2:** Restrict user to add same event on the same time frame.
    - Pros: More neat looking
    - Cons: User cant compare or manage well

3.8. Calendar Cancel Appointment feature

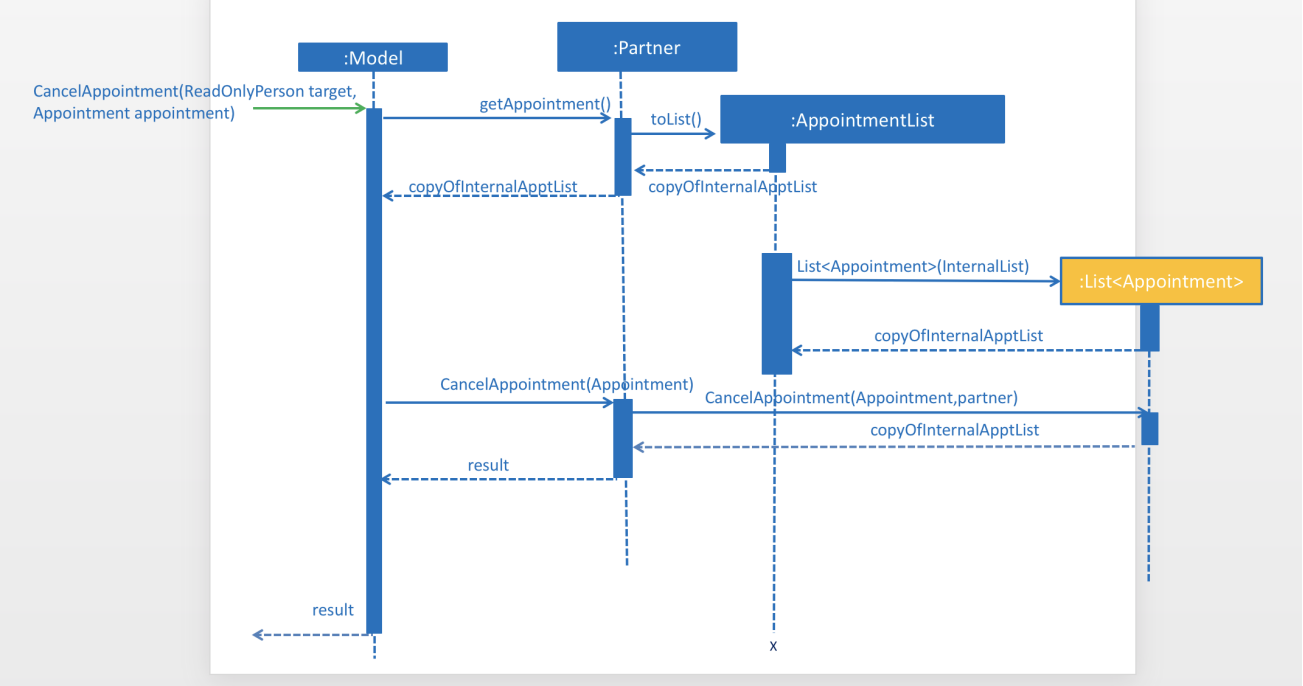
3.8.1. Proposed Implementation

The Calendar Cancel Appointment feature is implemented by 'Appointment', which will reside in 'Model'.

The idea of implementing a Cancel Appointment feature is to allow the user to remove his/her event on the Calendar when needed.

When the user calls to cancel an appointment in the Calendar, the model will ask AppointmentList to return a list of Appointments. However, the AppointmentList will only return a mutable copy of the AppointmentList back to the caller, as a practice of Defensive Programming.

The sequence diagram below shows interactions within the `Model` Component for the `CancelAppointment' API call.



3.8.2. Design Considerations

Aspect: Implementation of Add Event View

- **Alternative 1 (current choice):** Display in Calendar UI.
  - Pros: Easier to implement and Neater.
  - Cons: Have to cancel one by one and thus calendar will refresh over and over again.
- **Alternative 2:** Create a List to display all events.
  - Pros: User can have an overview of all the events listed to cancel
  - Cons: The UI will be blank at initial stage which is ugly.

3.9. Timetable View and Compare feature

3.9.1. Adding a Timetable

The Timetable Viewer feature is implemented by `Timetable`, which will reside in `ModelManager`.

Users are able to add a shortened NUSMods timetable URL to their existing partner in *NUSCouples*.

```
Sample shortened NUSMods URL: http://modsn.us/wNuIW
```

We pass the shortened URL through a `HttpURLConnection` to get the expanded URL.

```
Sample expanded NUSMods URL: https://nusmods.com/timetable/sem-
2/share?CS2101=SEC:C01&CS2103T=TUT:C01&···
```

The expanded NUSMods URL can be generalised and represented in the format `···/timetable/sem-`

`[SEM_NUM]/share?[MODULE_CODE]=[LESSON_TYPE]:[CLASS_NUM]&[MODULE_CODE]=[LESSON_TYPE]:[CLASS_NUM]&…`

We can parse this expanded NUSMods URL to get the `SEM_NUM`, as well as the `MODULE_CODE`, `LESSON_TYPE` and `CLASS_NUM` for each of the modules in the timetable.

Using NUSMods API, we can get the `WEEK_TEXT`, `DAY_TEXT`, `START_TIME`, `END_TIME` and `VENUE` of each module.

The following diagram shows how the Timetable class is represented.

A TimetableModule represents one NUSMods module.
The TimetableModuleSlots represents a particular class session of a TimetableModule. (e.g. Tutorial, Lecture, etc)

3.9.2. Design Considerations

Aspect: Implementation of add NUSMods timetable URL

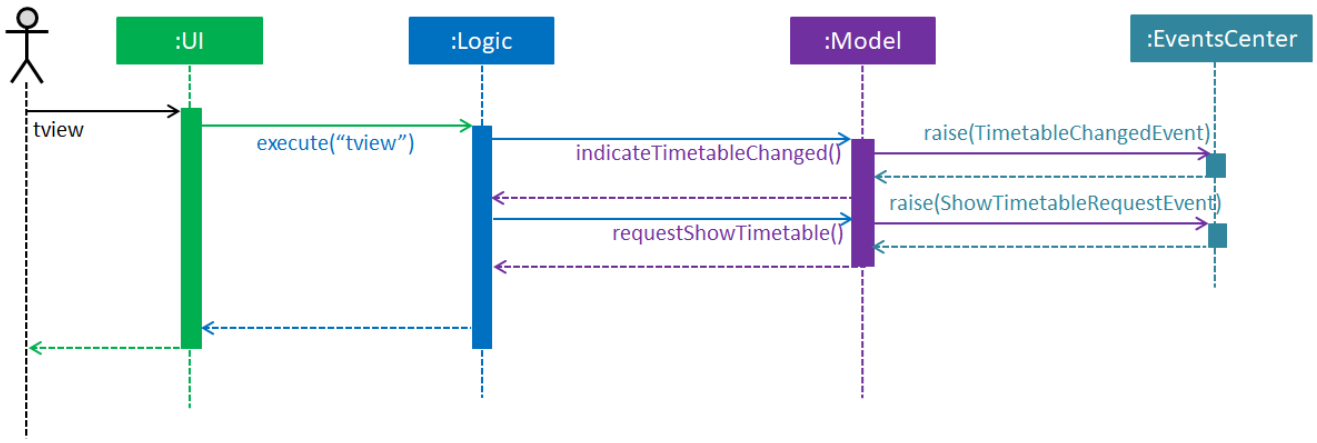
- **Alternative 1 (current choice):** Accept short URLs only
  - Pros: Easier to implement.
  - Cons: Less user friendly as users can only add one type of URL.
- **Alternative 2:** Accept both short URLs and expanded URLs
  - Pros: More user friendly as users have the choice to add either short or expanded URLs.
  - Cons: Difficult to check if given expanded NUSMods URL is a valid.

Aspect: Data Structure to support implementation of Timetable

- **Alternative 1 (current choice):** Store information by days of the week and by modules taken
  - Pros: Easy to add new functions on top of this implementation, more flexible.
  - Cons: May be a bit messy to implement due to the need to manage both structures.
- **Alternative 2:** Store information by days of the week
  - Pros: Easy to add new functions on top of this implementation such as compare timetables by days.
  - Cons: Have to sort information by day during parsing which can be tedious.
- **Alternative 3:** Store information by modules taken
  - Pros: Easier to implement due to how NUSMods API is structured.
  - Cons: Difficult to extract out information for a particular time slot on a particular day.

3.9.3. Viewing a Timetable

The following image shows how the tview Command works.



The `TimetableChangedEvent` is handled by `StorageManager` which will save the new timetable details into the relevant timetable display files.

```
    @Subscribe
    public void handleTimetableChangedEvent(TimetableChangedEvent event) {
        setUpTimetableDisplayFiles(event.timetable.getTimetableDisplayInfo());
        setUpTimetablePageHtmlFile();
        raise(new ShowTimetableRequestEvent());
    }
```

The `ShowTimetableRequestEvent` is handled by both `ListPanel` and `MainWindow`. The following code snippets show how they are handled.

```
    @Subscribe
    private void handleShowTimetableRequestEvent (ShowTimetableRequestEvent event) {
        logger.info(LogsCenter.getEventHandlingLogMessage(event));
        scrollTo(PARTNER_INDEX); // selects Partner in ListPanel
    }
```

```
    @Subscribe
    private void handleShowHelpEvent(ShowHelpRequestEvent event) {
        logger.info(LogsCenter.getEventHandlingLogMessage(event));
        handleHelp();
    }

    public void handleShowTimetable() {
        browserPanel.loadTimetablePage(); // Loads Timetable Page in Browser Panel
        if (!browserPlaceholder.getChildren().contains(browserPanel.getRoot())) {
            browserPlaceholder.getChildren().add(browserPanel.getRoot());
        }
    }
```

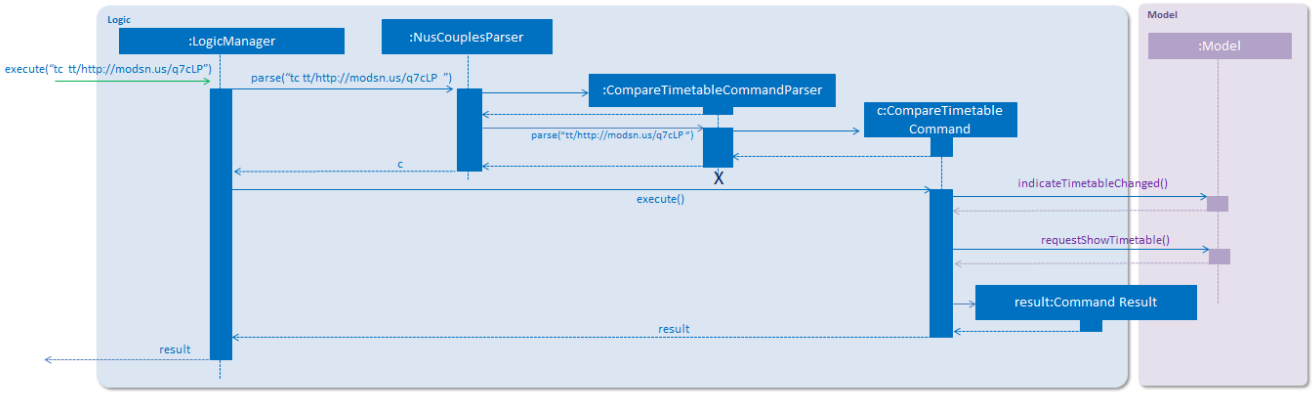
3.9.4. Design Considerations

Aspect: Implementation of storing Timetable Information

- **Alternative 1 (current choice):** Stores Information in a HTML file. Edits the javascript array in the HTML file to change the contents of the tables.
  - Pros: Easy to implement.
  - Cons: GUI will be a static web page.
- **Alternative 2:** Use JavaFX
  - Pros: Provides a friendlier GUI (able to drag and drop table view).
  - Cons: Takes longer to load and display.

3.9.5. Comparing Timetables

The sequence diagram below shows interactions within the `Logic` Component for the `execute("tc tt/http://modsn.us/q7cLP")` API call.



Similar to Viewing a Timetable, the CompareTimetableCommand raises two Events: `ShowTimetableRequestEvent` and `TimetableChangedEvent`. This updates the relevant files and refreshes the Timetable Page displayed.

Return to Table of Contents

3.10. [Proposed] Journal feature

3.10.1. Current Implementation

The journal feature is facilitated by an `ObservableList of <JournalEntry>` in Journal, which resides inside `ModelManager`. It allows the user to create and save journal entries in xml format. A `JournalEntry` contains only two variables, a String corresponding to the `date` the entry was created and a String containing its `text` contents.

Suppose that the user has just launched the application for the first time. The `Journal` will be empty at the beginning. On the `jnew` command, the app will check if the journal contains a `JournalEntry` corresponding to the current local date in the form (yyyymmdd). If it exists, its data (date and text)

is read from the `JournalEntry` and a copy of it is opened in a new `JournalWindow`. If it does not exist, a new `JournalWindow` is created. When the window is closed, a `handleJournalClose` method is called. If the `TextArea` is not empty, a `SaveEntryEvent` is raised. This event will pass the data from the `JournalWindow` in the form of a `JournalEntry` to the event handler.

Currently, the `LogicManager` will handle the SaveEntryEvent` by adding the `JournalEntry` to the `Journal`. If a `JournalEntry` with the same date exits, it will overwrite the text.

NOTE	The user cannot choose to save the `JournalEntry` under a different date. The same applies to editing past journal entries. The reason why this is implemented in this was is because a journal is a record of your thoughts and feelings in this moment. If you change your mind, that is your thoughts and feelings in a different moment. Thus, in order for the journal to be an accurate record of your thoughts and feelings each day, we have chosen to only allow the user to edit the journal corresponding to the current date.

The following sequence diagram shows how the newJournal operation works:

[NewJournalSequenceDiagram] | *NewJournalSequenceDiagram.png*

3.10.2. Design Considerations

Aspect: Implementation of `JournalWindow`

- **Alternative 1:** Use javafx to directly make a new window.
  - Pros: Easy to implement. Only requires a few lines of code in one or two files.
  - Cons: Not consistent with the rest of the app. Needs more effort to maintain when changes are made.
- **Alternative 2:** Make use of the UI framework.
  - Pros: Consistent with rest of app.
  - Cons: Harder to implement. Requires understanding of the UI component. Required minor edits in many files.

Aspect: Naming of journal entries

- **Alternative 1 (current choice):** Automatically uses current date "YYYYMMDD" as file name.
  - Pros: No need to worry about duplicate names. Easy to implement filtering (can filter by value easily).
  - Cons: Lack of personalisation. Hard to distinguish between files.
- **Alternative 2:** Allow user to name journal entries.
  - Pros: User can distinguish between files easily.
  - Cons: If duplicate names are allowed, we need to distinguish them with another method. If duplicate names are not allowed, user may struggle to find unique names for every entry.

Return to Table of Contents

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.

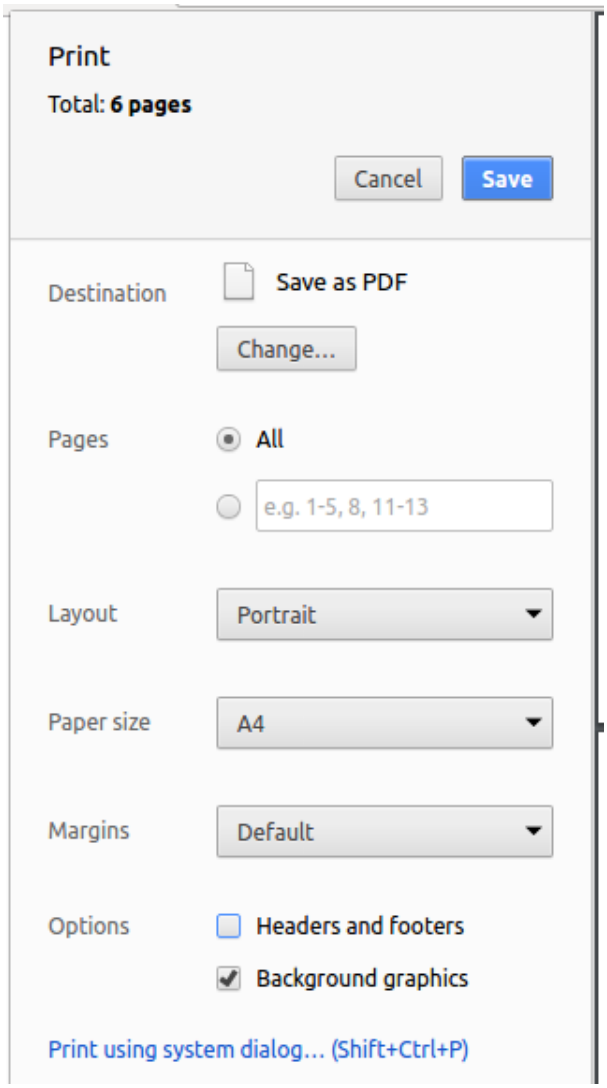


Figure 11. Saving documentation as PDF files in Chrome

Return to Table of Contents

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.address.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.address.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.address.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.address.logic.LogicManagerTest`

5.3. Troubleshooting Testing

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

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

Return to Table of Contents

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, Address Book depends on the Jackson library for XML 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)

Return to Table of Contents

Appendix A: Product Scope

Target user profile:

- is currently a student in a relationship with another student in NUS

- needs to remember special dates and moments
- needs to schedule meetings to find a good time to meet
- prefer desktop apps over other types
- can type fast
- prefer typing over mouse input
- are reasonably comfortable using CLI apps

Value proposition: all-in-one desktop app to help NUS couples make and remember memories with each other

Feature contribution:

Assignee	Major	Minor
Chen Xing	Scheduler: This app allows user to schedule//delete/view planned meetings	Accessibility: Reduce the effort when user enters command on the command box through custom keystrokes. Tracker: The system will update the time on the footer to show when the changes have been made.
Marlene	Timetable viewer: To help you keep updated with your partner's school schedule	Timetable comparator: To help couples identify common breaks during school term so they can plan meetings during their free time
Samuel	Journal: Allow couples to record their thoughts and feelings to remember the time spent with their partner	Tag (emotions) : Add/delete tags to journal entries (happy, sad, angry, funny)
Daniel	Send motivational picture: Motivate partner by sending a motivational picture via the browser panel	Command aliases: Enable shortcut keywords for commands

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…
* * *	pair of NUS students in a relationship	add/view my partner's NUS timetable for the week	keep track of my partner's free time
* * *	person in a relationship	View,Add and Delete special events/ meetings (e.g. Valentine's day) in the same month	better plan for these dates
* * *	person in a relationship	add/view Journal entries	remember special memories
* * *	pair of NUS students in a relationship	locate my partner from his NUS timetable	easily meet up with my partner on campus
* *	user who values their privacy	encrypt *NUSCouples* save data	prevent strangers from reading personal information such as my journal entries
* *	As a part of a couple in NUS	encrypt *NUSCouples* save data	prevent strangers from reading personal information such as my journal entries
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	user	able to execute keystrokes to lighten their typing	keep their spirits up

Priority	As a …	I want to …	So that I can…
*	user with many persons in the address book	sort persons by name	locate a person easily
*	user who likes to customize things	change the theme or customize text color	
*	experienced user	have shortcut keys	do the same thing in a shorter time
*	user	gets updated upon the changed of event	so he can be verified that changes has been made

Appendix C: Use Cases

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

Use case: Authenticate User with Google

MSS

1. User are required to generate and download their credential from google API credentials: Google Dashboard and import into project resource directory.
2. NusCouples use the credential to authenticate with Google API using Auth2.0.

   Use case ends.

Extensions

1a. The partner already has an existing google calendar hosted in google.

   1a1. NUSCouples redirects to google calendar account to authenticate using the user credential.

   1a2. User confirms change.

   Use case resumes at step 2.

1b. The given credential is invalid.

   1b1. NUSCouples shows an error message and close.

   Use case ends.

Use case: View Calendar of User

MSS

1.User inside browser panel enters the command to view calendar at different view(s). 2.NUSCouples displays the Calendar

+ Use case ends.

Extensions

1a. There is an existing user in NUSCouples.

1a1. *NUSCouples* request to display Calendar of day/week/month/year from CalendarFX.

1a2. *NUSCouples* populate the calendar on the browser panel

Use case resumes at step 2.

Use case ends.

Use case: Add Event on Calendar

MSS

1.User inside browser panel enters the command to add appointment. 2.NUSCouples update the appointment list. 3.NUSCouples displays the Calendar of all the events in the appointment list and update the change of event on the footer.

Use case ends.

Extensions

1a. User didn't specify date of event

1a1. *NUSCouples* shows an error

2a. NUSCouples add the new appointment into the list

Use case ends.

Use case: Delete Event on Calendar

MSS

1.User inside browser panel enters the command to delete appointment. 2.NUSCouples update the appointment list. 3.NUSCouples displays the Calendar of all the events in the appointment list.

Use case ends.

Extensions

1a. User didn’t specify date of event

1a1. *NUSCouples* shows an error

2a. NUSCouples cannot find the appointment into the list

2a1. NUSCouples displays error message

Use case ends.

Use case: Keyboard Accessibility

MSS

1.User enters a long command but want’s to add something at the front or back 2.User press Shift Ctrl (move cursor to the front) Shift Alt (move cursor to the back)

Use case ends.

Extensions

1a. User is using MAC

1a1. enter `option` key instead of `Alt` key

2a. User doesn’t have keystroke Shift/Alt/Ctrl.

2a1. This feature can’t be used

Use case ends.

Use case: View Timetable of Partner

MSS

1. User requests to view timetable of his/her partner.
2. *NUSCouples* displays the timetable.

   Use case ends.

Extensions

1a. The specified person does not have a timetable.

- 1a1. *NUSCouples* shows an error message.

Use case ends.

Use case: Add New Journal Entry

MSS

1. User requests to create new journal entry.
2. *NUSCouples* opens new journal window.
3. User enters text and closes journal window.
4. *NUSCouples* saves new journal entry in journal.

   Use case ends.

Extensions

- 1a. Journal entry with the current local date exists.
  - 1a.1 *NUSCouples* reads data and a new copy of the journal entry is opened in a journal window.
    Use case resumes at step 3.
- 3a. Journal entry with the current local date exists.
  - 3a.1 *NUSCouples* overwrites journal entry in journal.
    Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `1.8.0_60` 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.
4. Should be usable by a novice after reading the User Guide.
5. Should be able to handle any invalid inputs.
6. Should respond to user inputs within 2 seconds.
7. Should be able to work on both 32-bit and 64-bit environments.
8. Should have commands that are intuitive and easy to remember.
9. Should be able to control almost everything from the CLI.
10. The application should be connected to the internet.
11. Should able to store <100 appointments

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Third-Party API

NUSMods, CalendarFX

Private contact detail

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

Appendix F: Product Survey

Product Name

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 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 … }

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 … }