Event Planning isn’t Complicated (EPIC) - Developer Guide

By: `Team W13-B2` Since: `Mar 2018` Licence: `MIT`

1. Introduction

EPIC is an event planning tool for large organisations hosting many large-scale events (e.g. schools which host competitions, award and graduation ceremonies, seminars and talks). {more to be added}

2. Setting up

2.1. Prerequisites

Please ensure that you have the following installed on your machine:

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.

2.2. Setting up the project in your computer

Please follow these steps to set 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.

2.3. Verifying the setup

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

2.4. Configurations to do before writing code

2.4.1. Configuring the coding style

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

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

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

2.4.2. Updating documentation to match your fork

After forking the repo, links in the documentation will still point to the `CS3103JAN2018-W13-B2/main` repo. If you plan to develop this as a separate product (i.e. instead of contributing to the `CS2103JAN2018-W13-B2/main`) , you should replace the URL in the variable `repoURL` in `DeveloperGuide.adoc` and `UserGuide.adoc` with the URL of your fork.

2.4.3. 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).

2.4.4. Getting started with coding

When you are ready to start coding,

1. Get some sense of the overall design by reading Section 3.1.
2. Take a look at [GetStartedProgramming].

3. Design

3.1. Architecture

The ***Architecture Diagram*** given below explains the high-level design of the App.

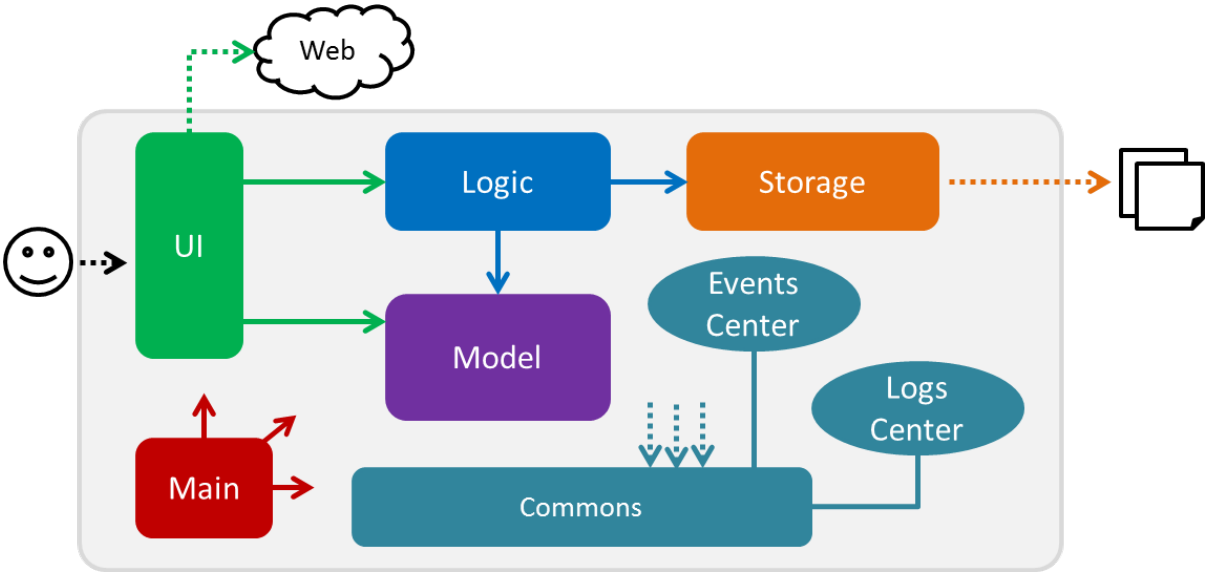


Figure 1. Architecture Diagram

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: Initializing the components in the correct sequence, and connecting them up with each other.
- At shut down: Shutting down the components and invoking 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` : This class is used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- `UI`: Interfaces with the user
- `Logic`: Executes commands
- `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.

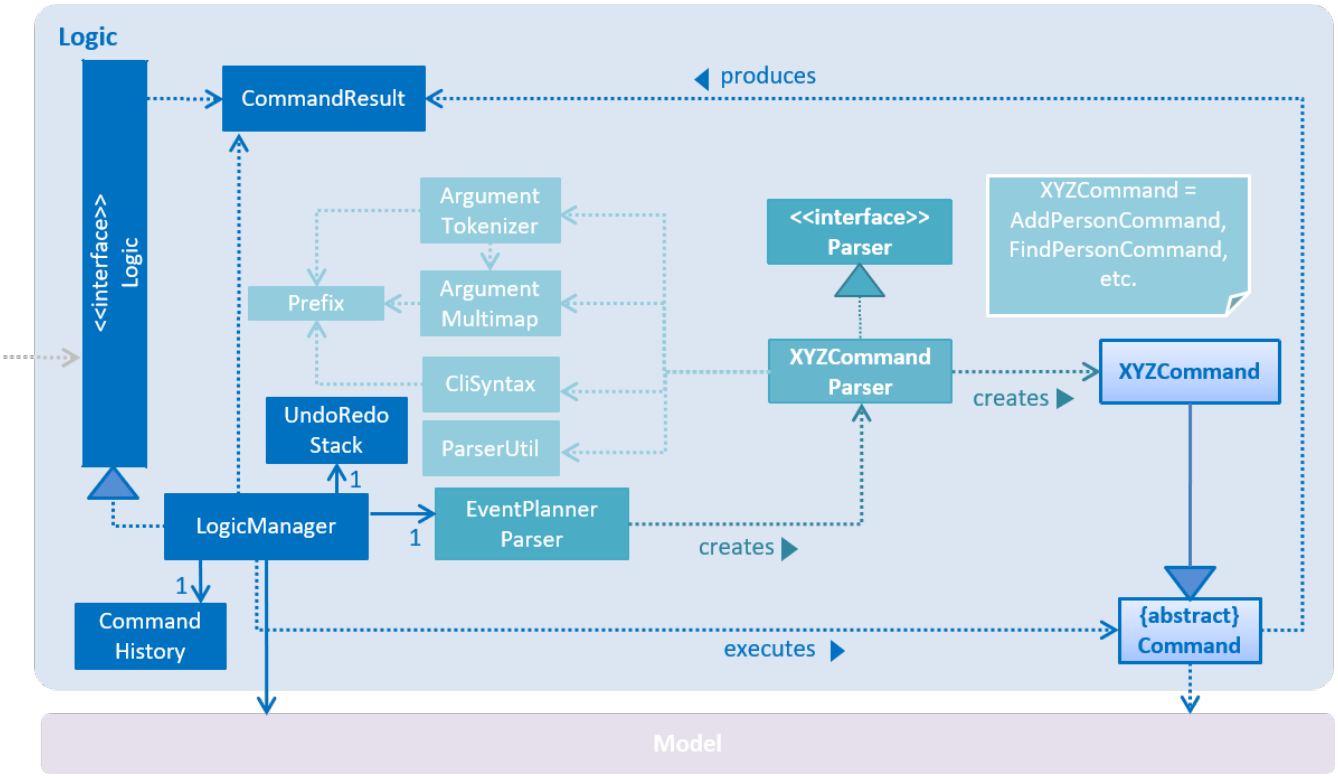


Figure 2. Class Diagram of the Logic Component

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

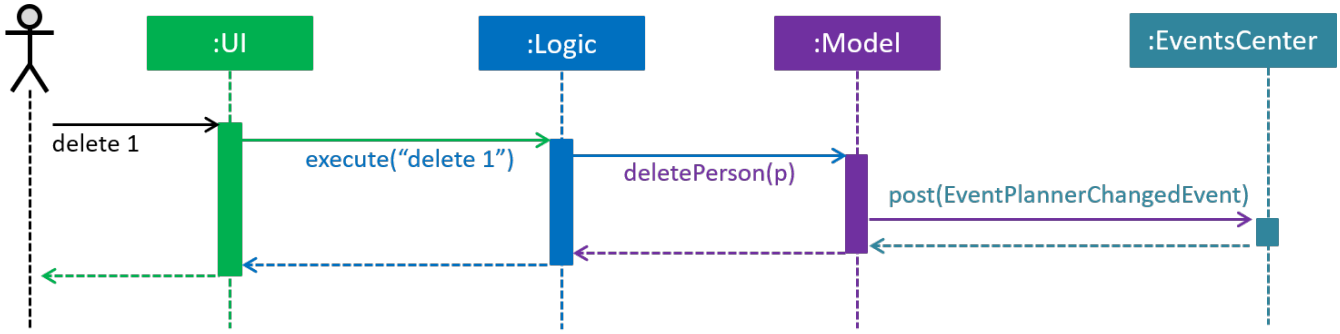


Figure 3. Component interactions for `delete 1` command (part 1)

NOTE Note how the `Model` simply raises a `EventPlannerChangedEvent` when the Event Planner data is 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.

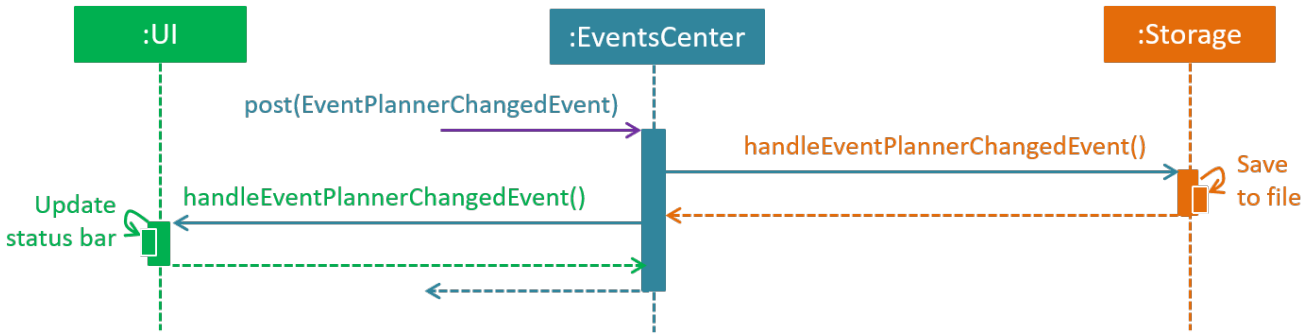


Figure 4. Component interactions for `delete 1` command (part 2)

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.

3.2. UI component

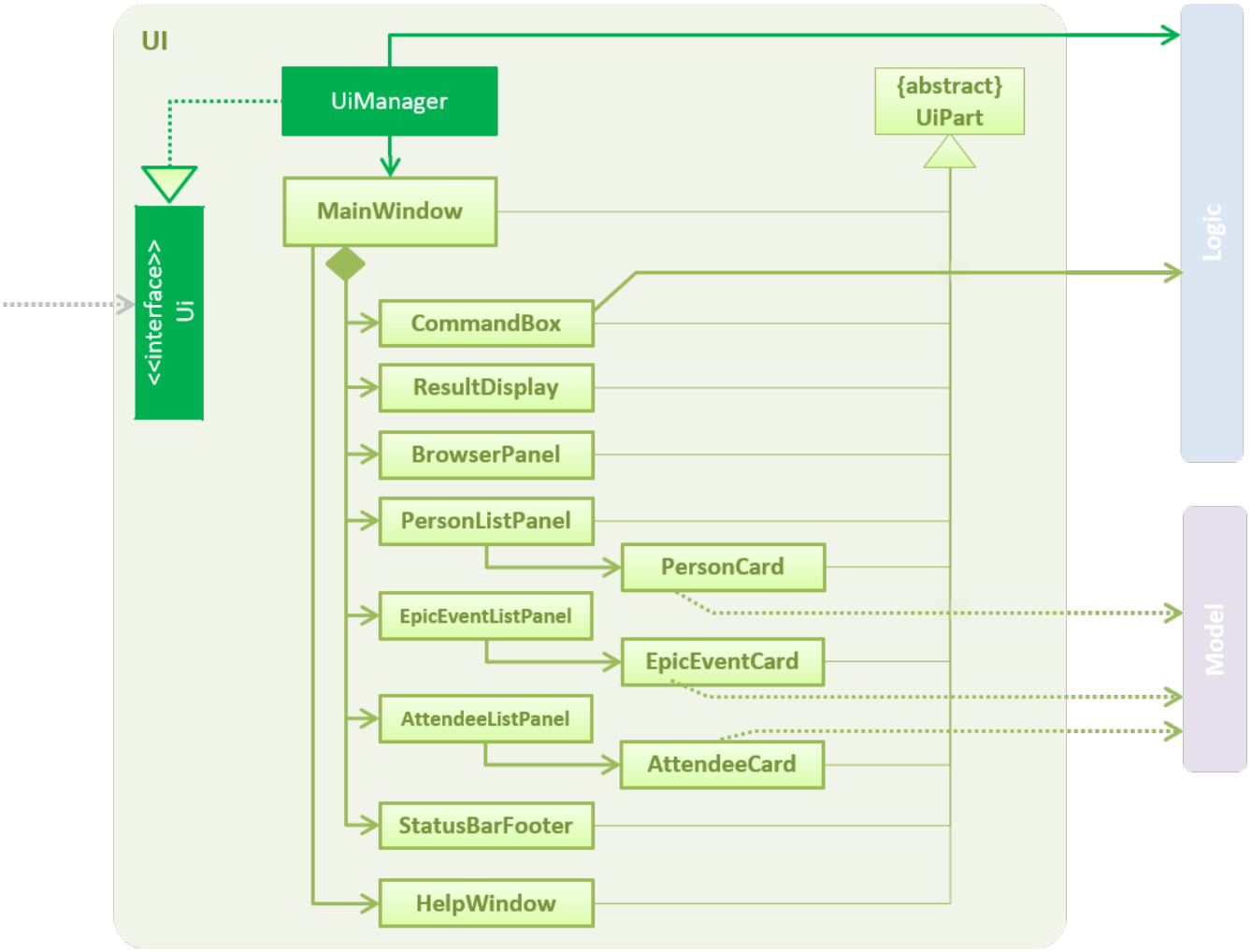


Figure 5. Structure of the UI Component

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.

3.3. Logic component

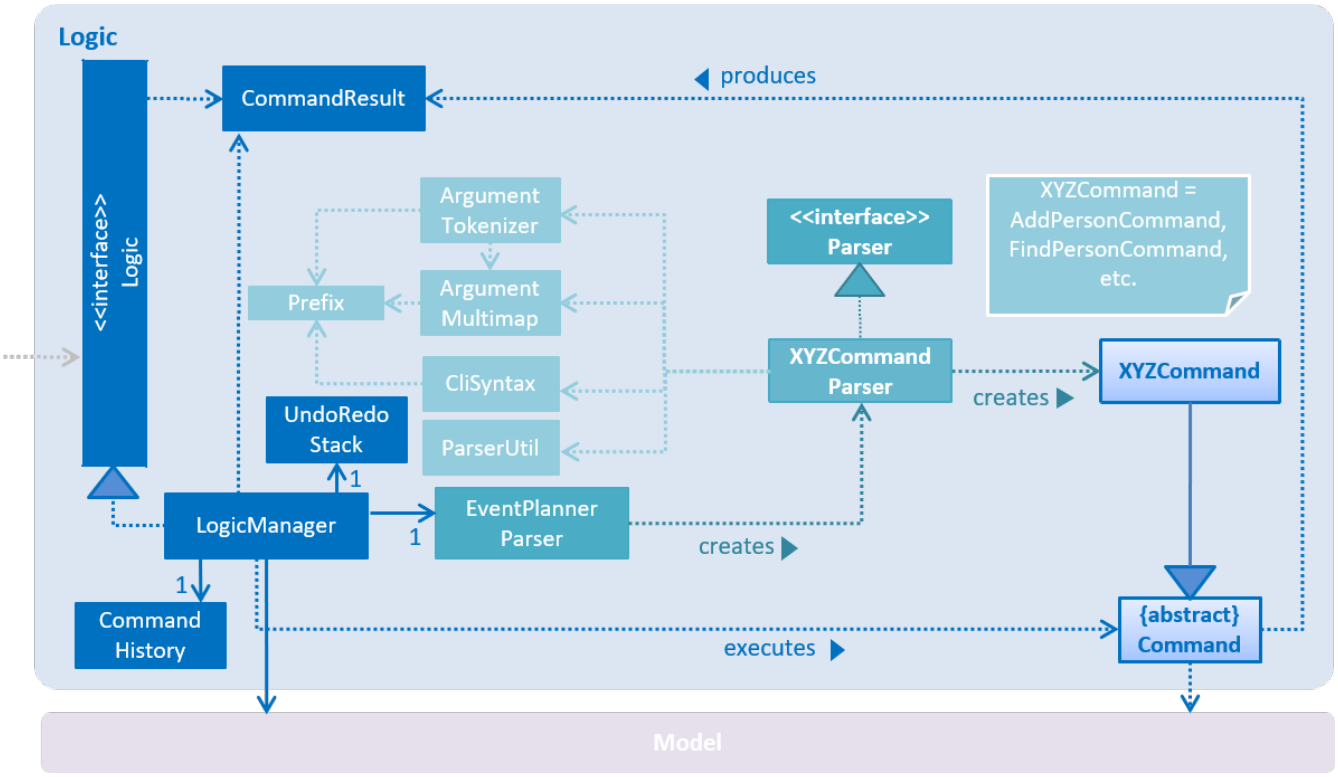


Figure 6. Structure of the Logic Component

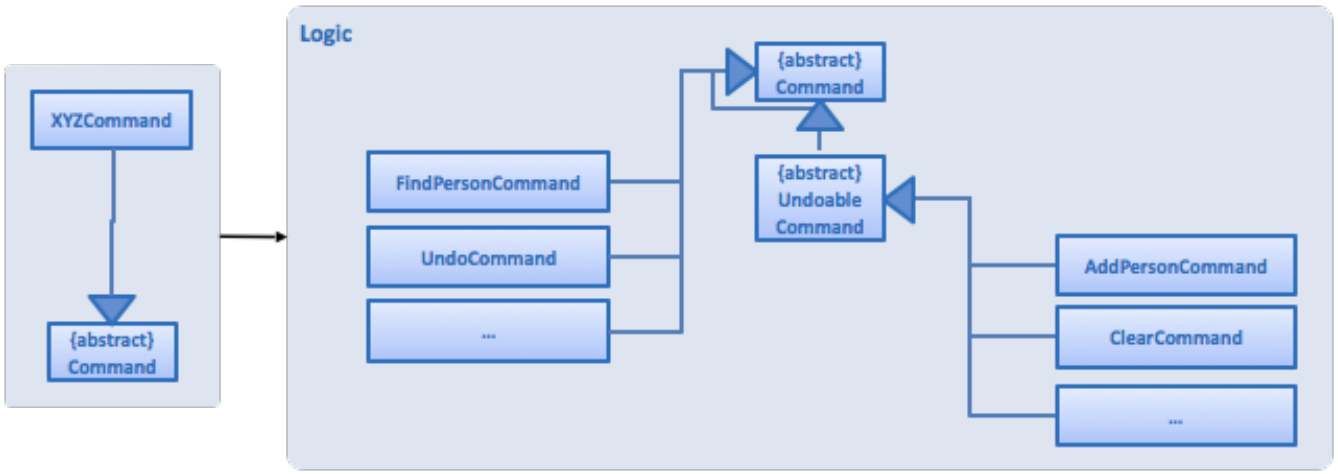


Figure 7. Structure of Commands in the Logic Component. This diagram shows finer details concerning `XYZCommand` and `Command` in Figure 6

API : `Logic.java`

1. `Logic` uses the `EventPlannerParser` 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.

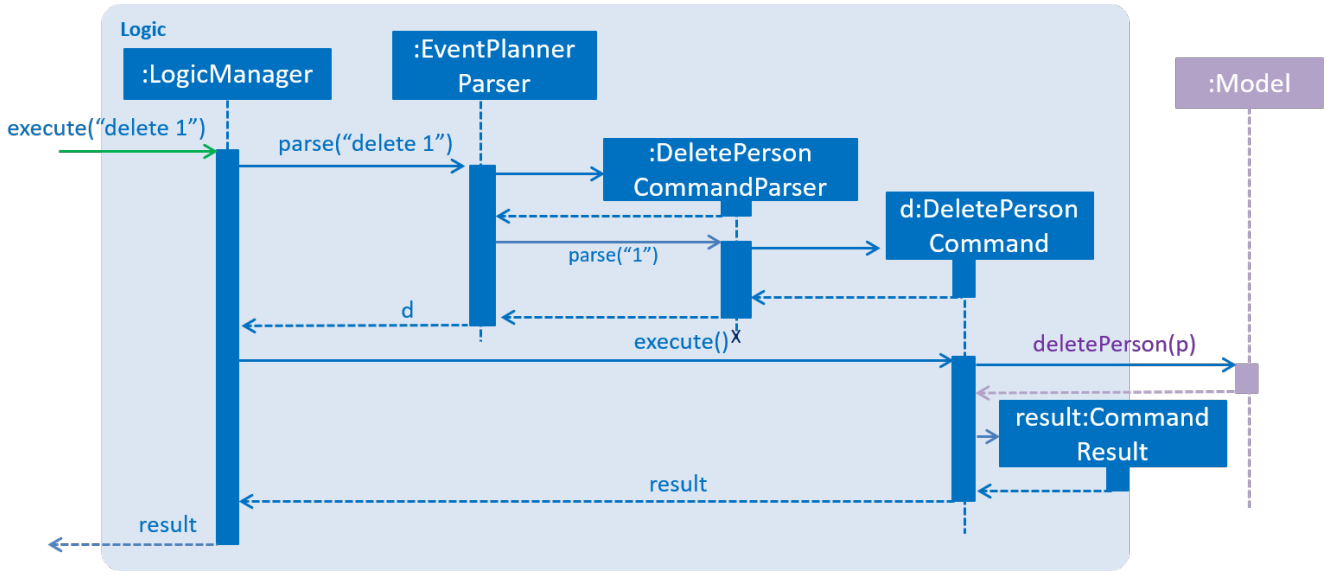


Figure 8. Interactions Inside the Logic Component for the `delete 1` Command

3.4. Model component

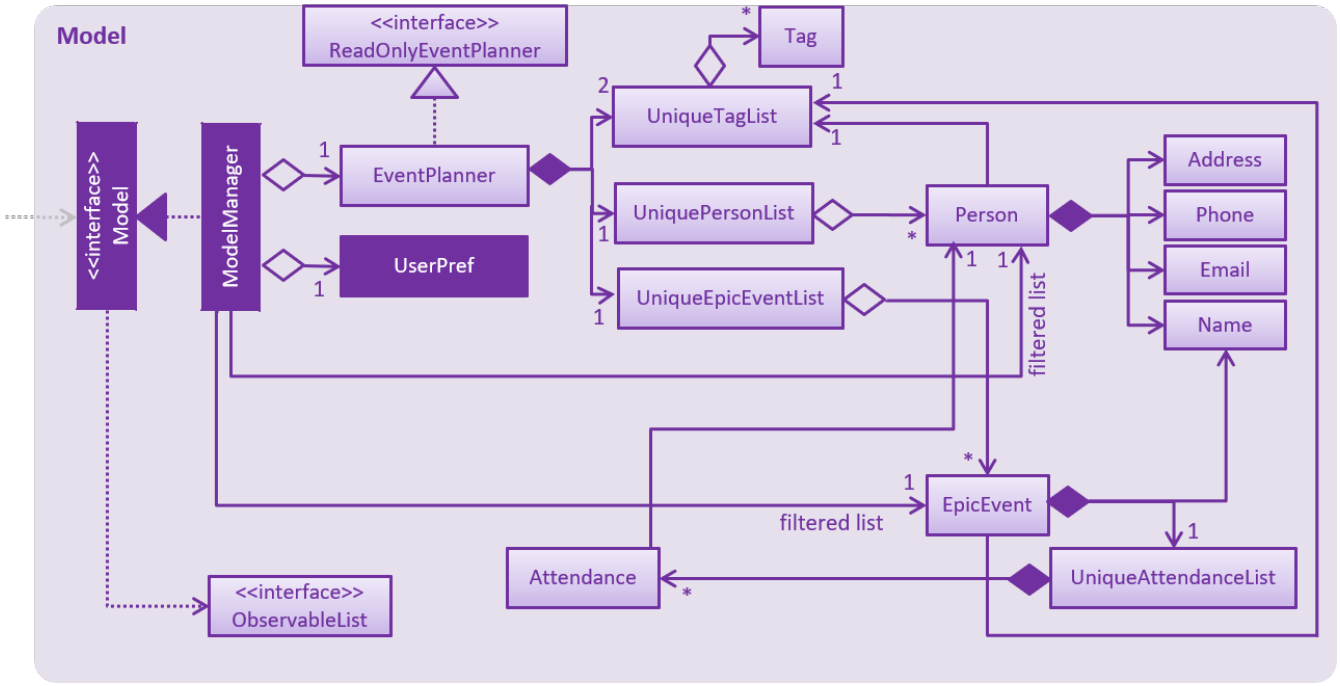


Figure 9. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Event Planner 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.
- exposes an unmodifiable `ObservableList<EpicEvent>` 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.

3.5. Storage component

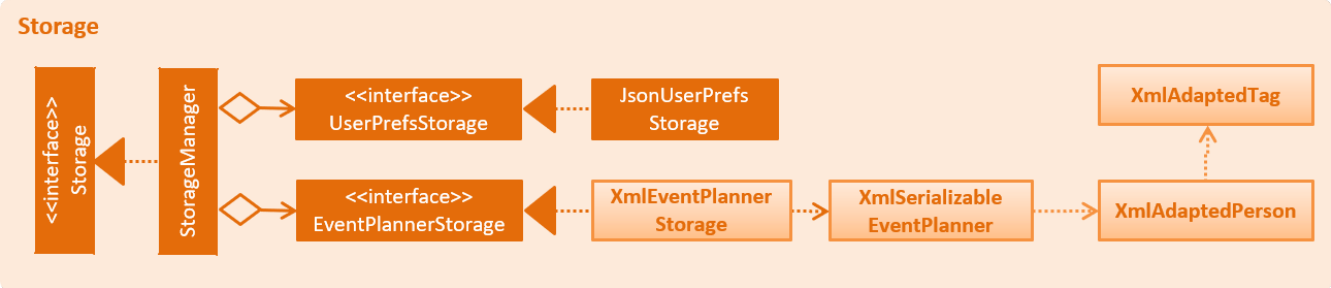


Figure 10. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

3.6. Common classes

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

4. Implementation

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

4.1. Edit Person/Event feature

4.1.1. Previous Implementation

In AddressBook-Level4, the `edit` command was performed by creating a new `Person` and passing it to a `UniquePersonList` in the model, which would then replace the to-be-edited `Person` with it.

```
    int index = internalList.indexOf(personToEdit);
    Person editedPerson = new Person(name, ...);
    internalList.set(index, editedPerson);
```

4.1.2. Current Implementation

In EPIC, the `edit` and `edit-event` commands are now implemented in a mutable manner - instead of replacing the to-be-edited `Person/EpicEvent` with the new one, we edit the details of the to-be-edited `Person/EpicEvent` directly.

```
    int index = internalList.indexOf(personToEdit);
    Person editedPerson = new Person(name, ...);
    internalList.get(index).setPerson(editedPerson); // setPerson edits internal
details using those of the supplied Person
```

4.1.3. Design Considerations

Aspect: Implementation of `edit`

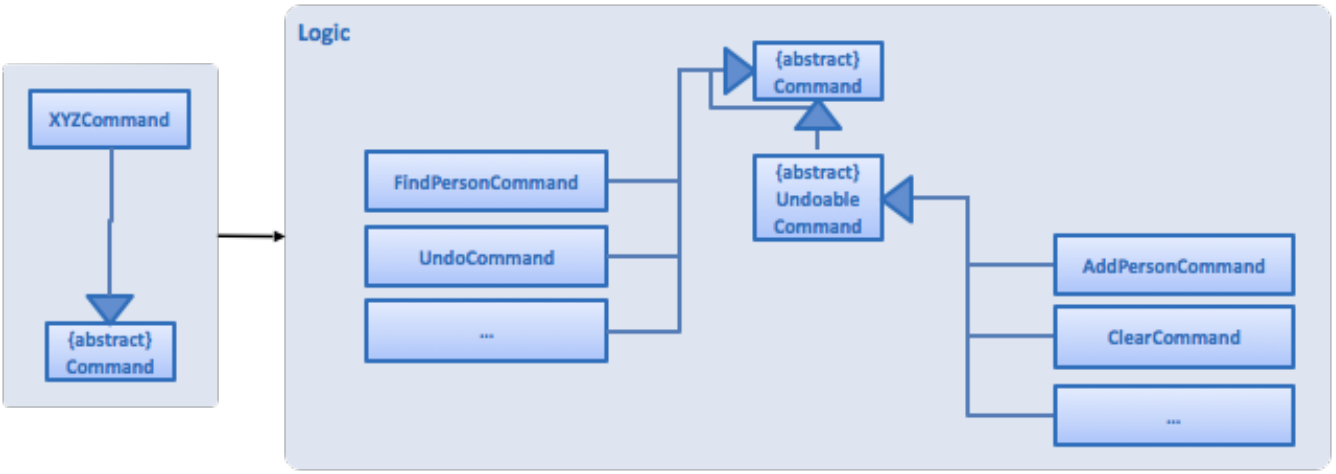
- **Alternative 1 (current choice):** Edit in a mutable manner
  - Pros: Since EPIC has both `EpicEvent` and `Person` objects, which maintain references to one another, editing a `Person/EpicEvent` in this manner automatically updates the `EpicEvent/Person` objects that is associated with it.
  - Cons: Implementation of `undo` will be more difficult.
- **Alternative 2:** Edit in an immutable manner
  - Pros: Implementation of `undo` is easier, since we can just replace the current `EventPlanner` with the previous one.
  - Cons: Editing a `Person/EpicEvent` will require passing a copy of the newly-created `Person/EpicEvent` to all objects associated with the to-be-edited version, introducing significant overhead

4.2. Undo/Redo feature

4.2.1. 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 event planner (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 generating the opposite command 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:

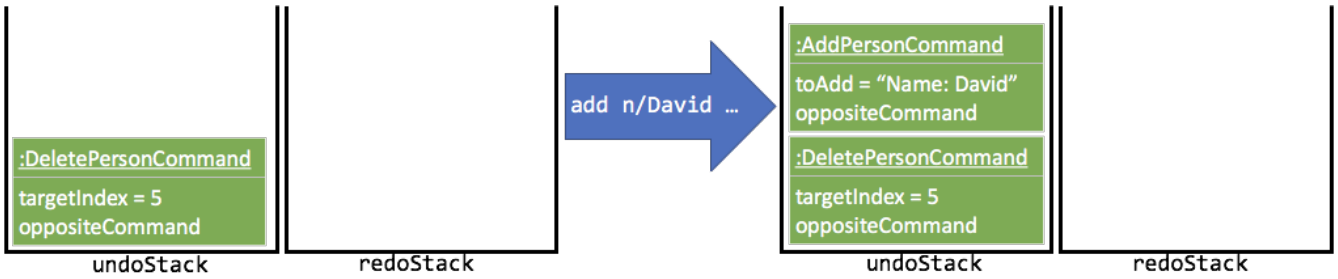
```
public abstract class UndoableCommand extends Command {
    @Override
    public CommandResult execute() {
        // ... undo logic ...

        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 event planner. The current state of the event planner 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).

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.

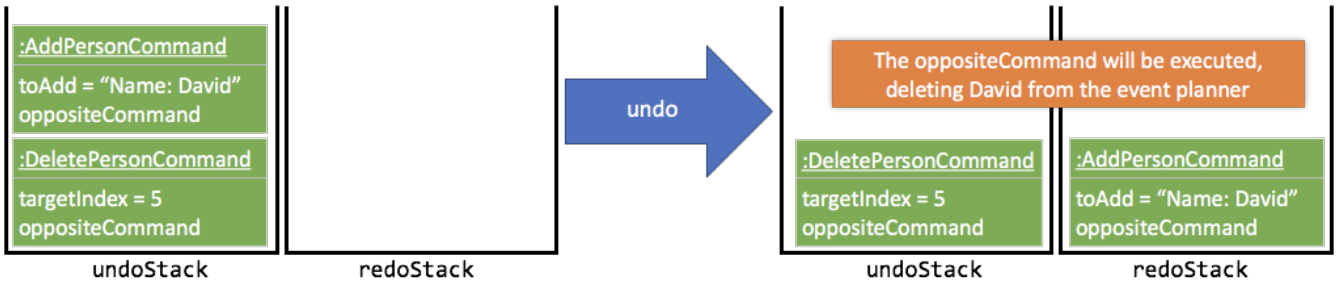


NOTE If a command fails its execution, it will not be pushed to the `UndoRedoStack` at all.

NOTE The `oppositeCommands` for the `AddPersonCommand` and `DeletePersonCommand` above are different! The former is a `DeletePersonCommand` while the latter is an `AddPersonCommand`.

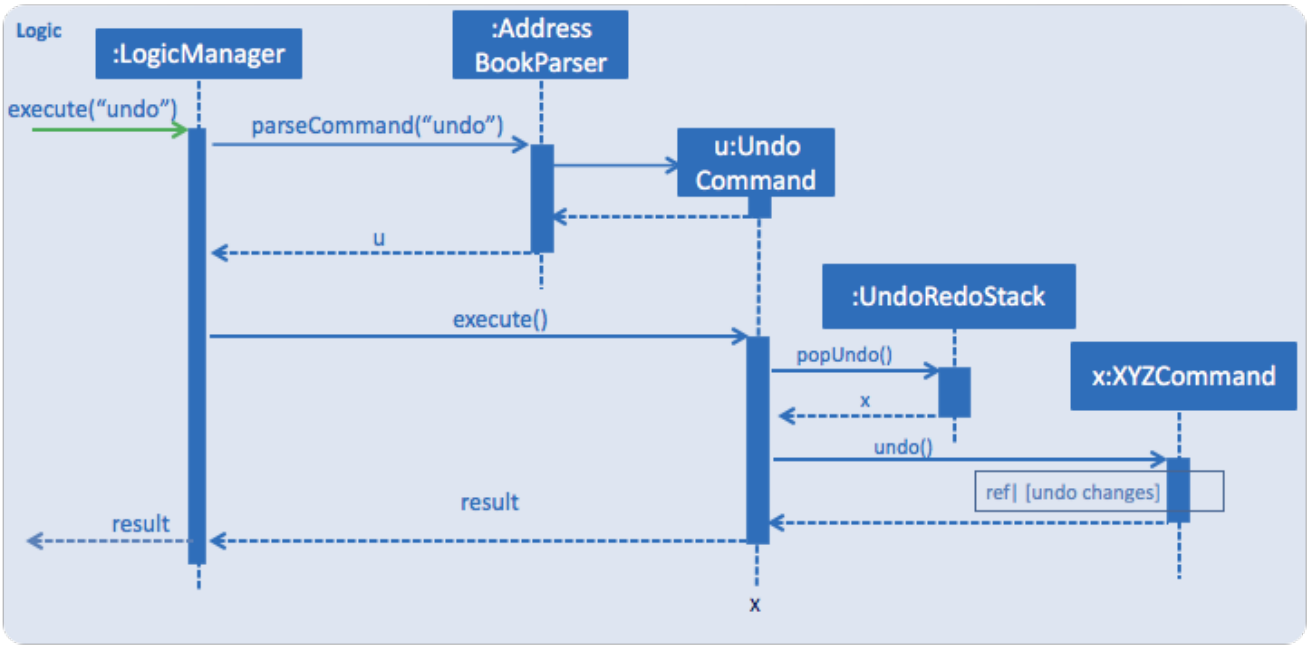
The user now decides that adding the person was a mistake, and decides to undo that action using `undo`.

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



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

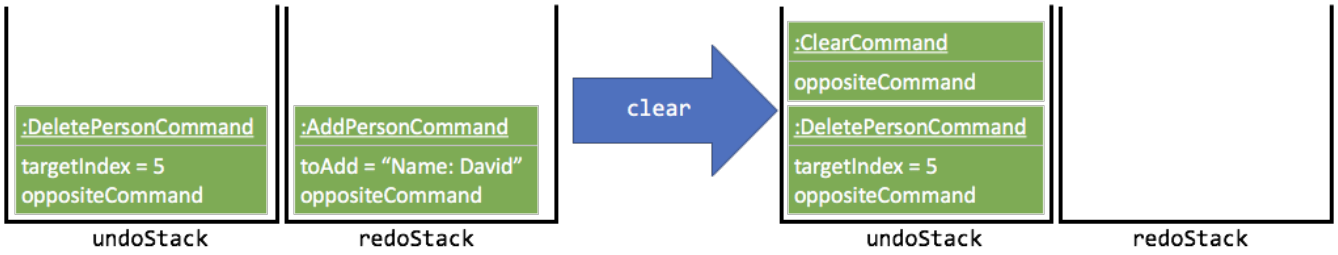
The following sequence diagram shows how the undo operation works:



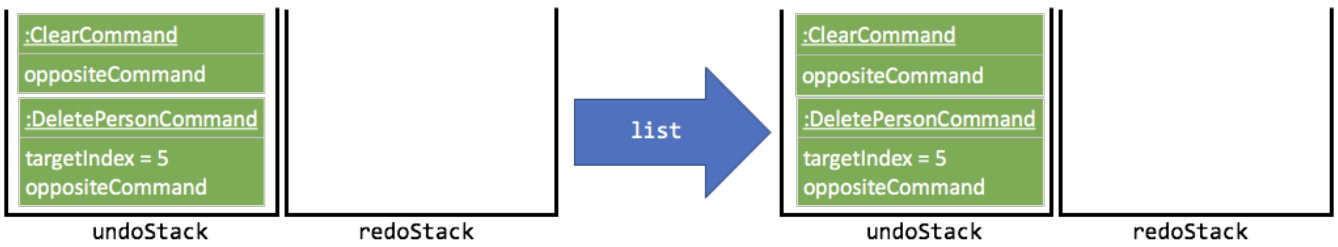
The redo does the exact opposite (pops from `redoStack`, push to `undoStack`, and performs the intention of the original command).

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`.
NOTE	redo() does not simply execute the `Command` with the previous parameters! This would cause indexing issues with commands like `delete` if `filteredPersons` had been altered by a `find` or other view command.

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).

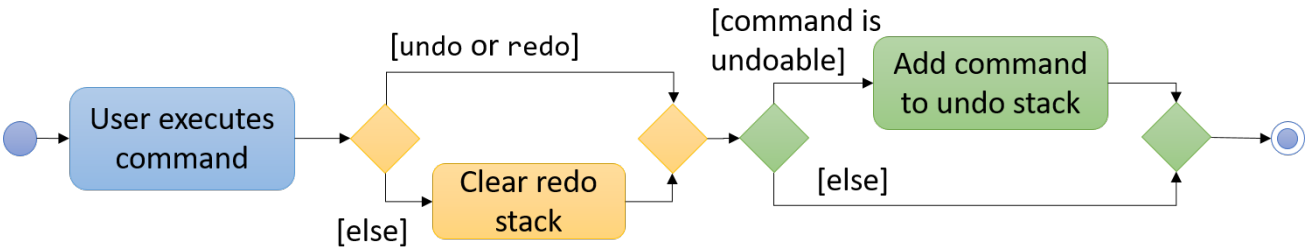


Commands that are not undoable are not added into the `undoStack`. For example, `list`, which inherits from `Command` rather than `UndoableCommand`, will not be added after execution:



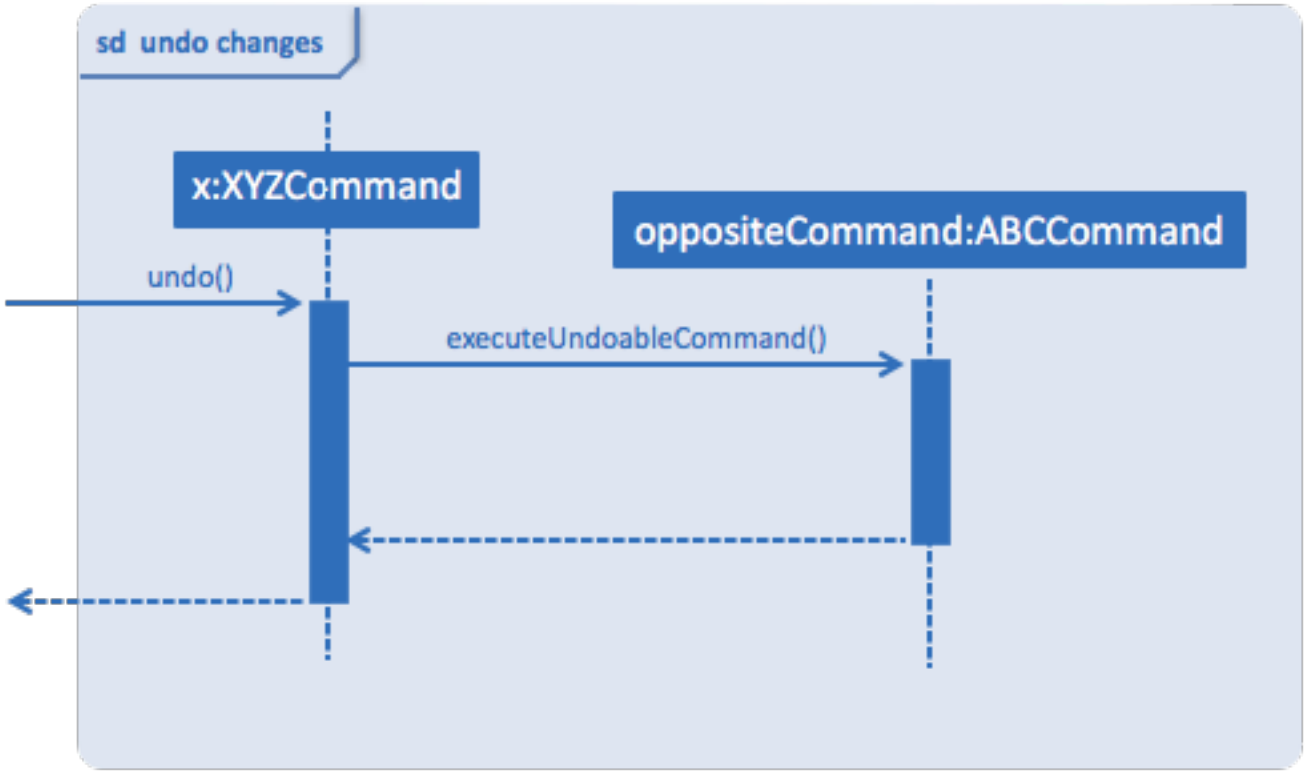
The following activity diagram summarizes what happens inside the `UndoRedoStack` when a user

executes a new command:



4.2.2. Changes from previous implementation [major enhancement, in progress]

Instead of saving the entire event planner each time we execute an `UndoableCommand`, each `UndoableCommand` knows how to `undo/redo` itself. Each `UndoableCommand` has an `oppositeCommand` field, which is another `UndoableCommand` that, when executed, reverses the changes made by the original command. The sequence diagram for the new `undo()` implementation is shown below.



The `oppositeCommand` is generated in the `execute()` method, after `preprocessUndoableCommand()`. This is because generating the `oppositeCommand` requires knowledge of the actual `Person/EpicEvent` objects to be modified. For example, the `oppositeCommand` for a `deletePersonCommand` is an `addPersonCommand`, but we only know the person to be deleted after the pre-processing step.

NOTE	Each `UndoableCommand` now requires its individual `generateOppositeCommand()` implementation. Hence, this method is made abstract in the abstract class `UndoableCommand`

There was no `Command` that could easily reverse the changes of a `ClearCommand`, hence a new `Command` `RestoreCommand` had to be created. Since the sole purpose of this command is be the `oppositeCommand` of a `ClearCommand`. This command is not directly accessible to the user, and can only be executed

when the user undoes a `ClearCommand`.

4.2.3. Design Considerations

Aspect: Implementation of `UndoableCommand`

- **Alternative 1 (current choice):** Add a new abstract method `executeUndoableCommand()`
  - Pros: Undo/redo functionality will now be part of the default behaviour. Classes that deal with `Command` will not have to know that `executeUndoableCommand()` exist.
  - Cons: New developers will find the template pattern difficult to understand.
- **Alternative 2:** Just override `execute()`
  - Pros: New developers will not have to learn the above template pattern
  - Cons: Command classes that inherit from `UndoableCommand` must remember to call `super.execute()`, or lose the ability to be undone/redone.

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Individual command knows how to undo/redo itself.
  - Pros: Significantly less memory is used (e.g. for `delete`, just save the person being deleted). Compatible with mutable commands.
  - Cons: Implementation is more complicated.
- **Alternative 2:** Saves the entire event planner.
  - Pros: Implementation is easy.
  - Cons: Performance issues may result due to high memory usage. Also, this is incompatible with the mutable `edit` and `edit-event` implementations.

Aspect: Type of commands that can be undone/redone

- **Alternative 1 (current choice):** Only include commands that modifies the event planner (`add`, `clear`, `edit`).
  - Pros: Only commands that cannot be easily reverted need to be implemented (the view can easily be re-modified as no data is * lost).
  - Cons: User might mistakenly think that undo also applies to view modification (e.g. filtering).
- **Alternative 2:** Include all commands.
  - Pros: Might be more intuitive for the user.
  - Cons: User has no way of skipping such commands if he or she just want to reset the state of the event planner and not the view. **Additional Info:** See the 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 undergraduates to understand, who are likely to be * 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.

4.3. Register/Deregister persons for/from events

The EpicEvent-Person association is unidirectional i.e. an `EpicEvent` maintains references to `Person` objects registered for it, but a `Person` does not maintain references to `EpicEvent` objects he/she has registered for. This implementation was chosen to reduce overhead and complications in implementation, since all commands v2.0 supports will not require the backward association.

The references are maintained using a `UniquePersonList` inside each `EpicEvent`

4.4. List registered persons for an event

This is done by creating a `Predicate` that tests whether a `Person` is in an `EpicEvent`, then passing it to `updateFilteredPersonList()`

4.5. [Proposed] Marking/Unmarking Attendance

The proposed implementation is to use an association class called `Attendance`. An instance of `Attendance` is created every time a person that registers to an event and it is stored inside the `EpicEvent`. The instance stores the corresponding `Person` and stores a `boolean` representing whether the person has attended the event. This reduces coupling between the `Person` and `EpicEvent` class and allows the event to have access to all its attendees so that adding, removing and listing attendees is easy to implement.

4.6. [Proposed] Mass Registration

This feature is proposed to allow for mass registration of multiple persons to a single event. The proposed implementation is to make use of tags to select groups of persons to register for an event. It is proposed that the `Tag` class be split into `PersonTag` and `EpicEventTag` so as to differentiate the two. This allows for searching of persons by `PersonTag` and searching for events by `EpicEventTag`. By doing so, a group of persons can be identified by `PersonTag` and thus registered to an event by iteratively registering each person.

4.7. [Proposed] 3 Pane UI

The proposed new UI is a 3 pane UI consisting of list of persons, events, and attendees for the selected event. Being able to view all three panes will allow the user to read off the desired

command parameters without having to switch between lists.

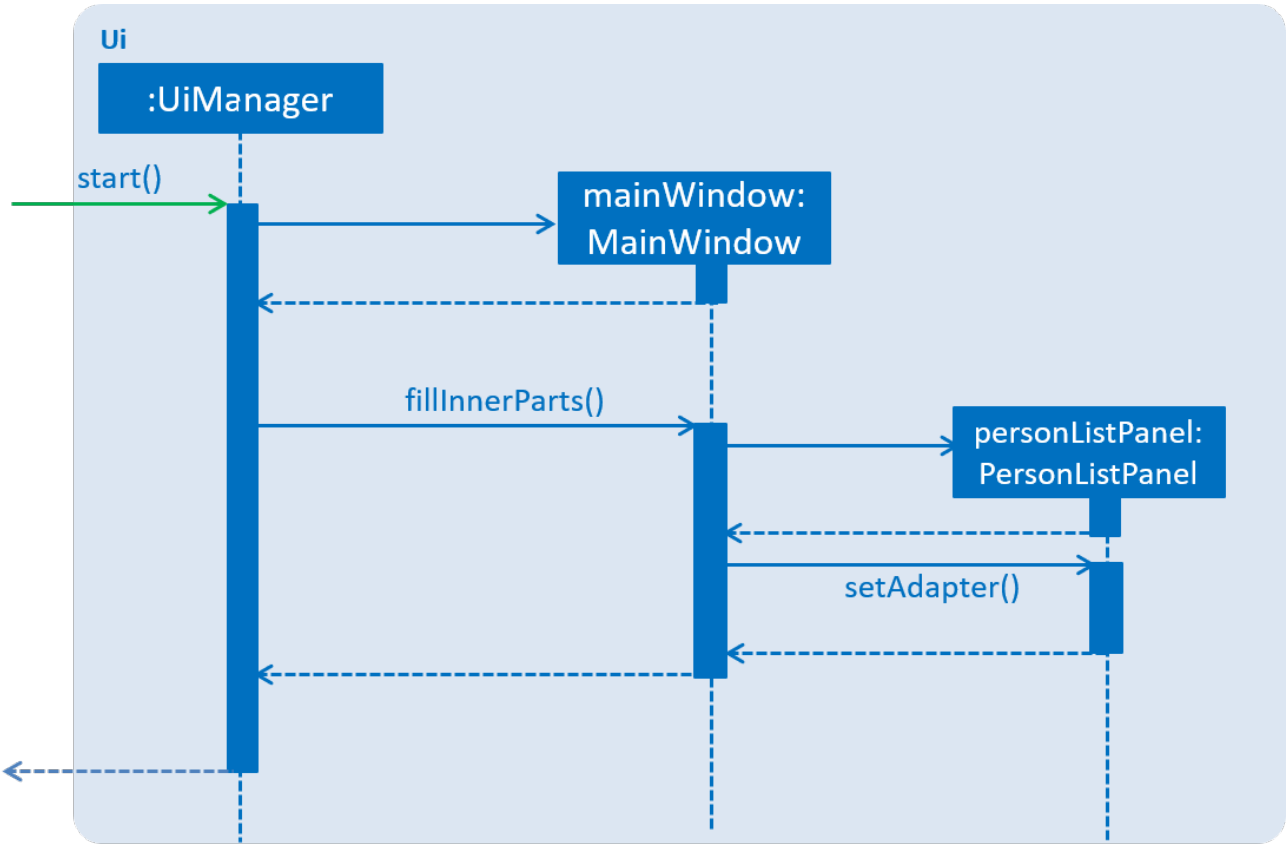
4.7.1. Current Implementation

The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox`, `ResultDisplay`, `PersonListPanel`, `EpicEventListPanel` and `AttendeeListPanel`. The `ListPanel` s forms the bulk of the UI.

As you can see from the UI component diagram, the three lists extend from the `ListPanel` class. `ListPanel` is a vetically scrollable collection of `Cards`. It is an adapter view that does not know the details, such as type and contents, of the view it contains.

In addition, we define a `ListPanelAdapter` class that acts as the bridge between `ListPanel` and data behind the list.

The sequence diagram below shows how `PersonListPanel` is initialised. `EventsListPanel` and `AttendeeListPanel` are initialised in a similar manner.



NOTE If no `EpicEvent` has been selected, `AttendeeList` will be empty

4.7.2. Design Considerations

When deciding on the UI, the following aspects of user experience were considered.

Aspect: Overall UI Design

- **Alternative 1 (current choice)**: A 3 pane UI consisting of list of persons, events, and attendees for selected event
    - Pros: The user can view all 3 lists at the same time. He would know what arguments to

supply when typing commands as he can read them off the list.

    - Cons: The UI might become too cluttered as there are too many UI elements. However, given that EPIC is meant for modern computers with large displays, this should not be an issue.

- **Alternative 2 (previous choice)**: 2 pane UI where the left pane is a 2 tab pane consistings of list of persons and events, and the right pane is the list of attendees.
    - Pros: Merging the horizontal space for list of persons and events will create more space for list of attendees. The user is likely to be more interested in the attendees details.
    - Cons: If the user needs to access data for some tabbed pane that is not in focus to fill out a command, this would break his workflow. The user will have to delete his current command, execute a command to set focus to the desired tab, memorize the required details and reenter his previous command.
- **Alternative 3**: A common list that can display either list of persons, events or attendees for selected event
    - Pros: We only have to make minimal changes to the UI layout.
    - Cons: Events, persons, and attendees must be displayed using the same `Card` class. This would result in tight coupling of the display graphics logic for the three lists.

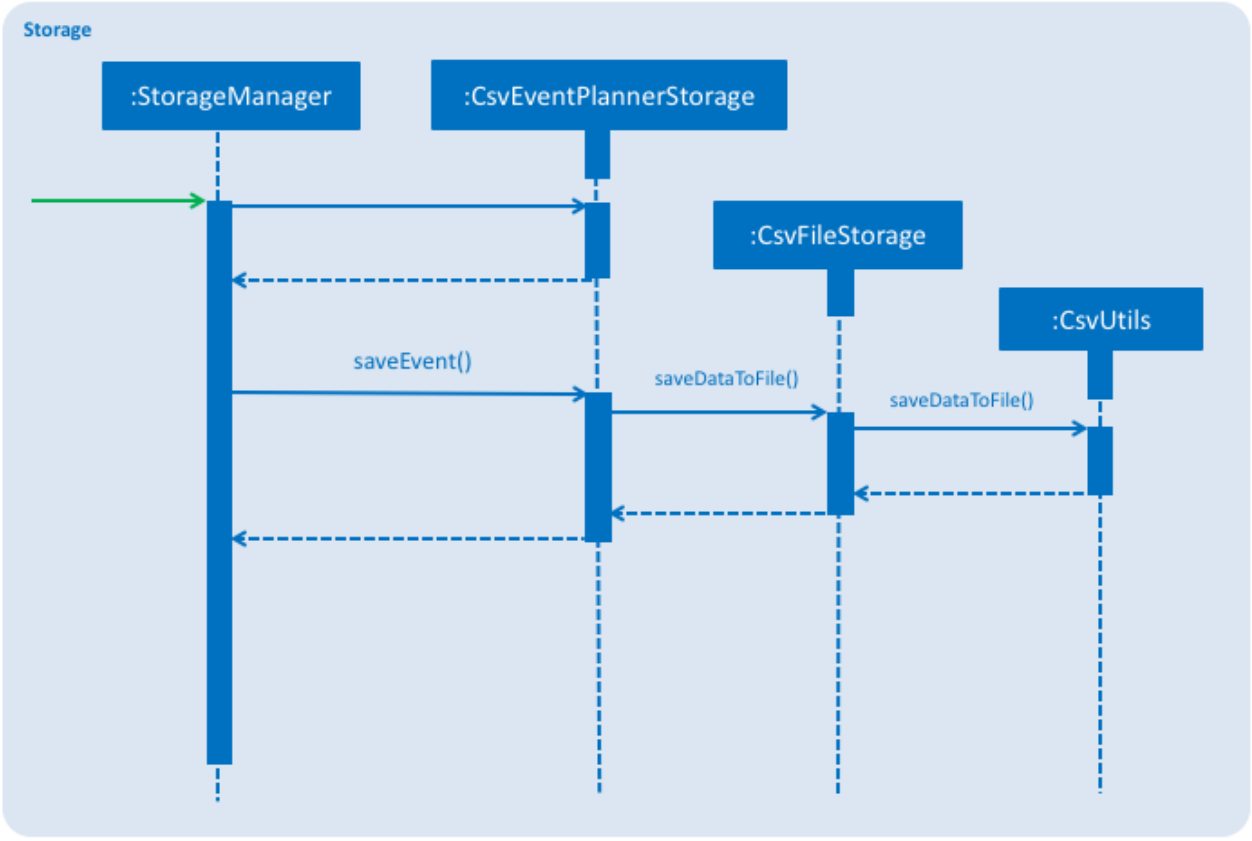
4.8. [proposed] Export an Event

The proposed export command exports the name of the attendees, their phone number, their email address, their home address and their attendance information of a particular event to a csv file into a file path specified by the user. Having the attendance of a particular event in a csv file would enable the users to analyse the data with other platforms.

4.8.1. [proposed] Implementation

The `export-event` command input is first parsed in `EventPlannerParser` to create an `ExportEventCommand`. The execution of it then calls the respective storage components. The subsequent export mechanism is handled by a `CsvEventPlannerStorage` which is a kind of `EventPlannerStorage`. The event data is parsed by `CsvUtil` and then exported as file through `CsvFileStorage` using the file path specified by the user.

The sequence diagram below shows how the csv event export is processed in the storage component.



4.8.2. Design Considerations

When deciding on the export option, the following aspects of user experience are considered.

Aspect: Data to be Exported

- **Alternative 1 (current choice)**: Name of the attendees, their phone number, their email address, their home address and their attendance information for the event
  - Pros: The information includes almost all the data users need for an event. The parsing of the data is straight forward.
  - Cons: The tags of the attendees are not exported.
- **Alternative 2**: All information for attendees including their tags
  - Pros: More information for users as compared to current implementation.
  - Cons: The attendees may have zero or multiple tags. The uncertainty in the number of tags make parsing messy.

4.9. Logging

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

- The logging level can be controlled using the `logLevel` setting in the configuration file (See

Section 4.10)

- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

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

4.10. Configuration

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

5. Documentation

We use asciidoc for writing documentation.

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

5.1. Editing Documentation

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

5.2. Publishing Documentation

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

5.3. Converting Documentation to PDF format

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

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

1. Follow the instructions in UsingGradle.adoc to convert the AsciiDoc files in the `docs/` directory to HTML format.

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

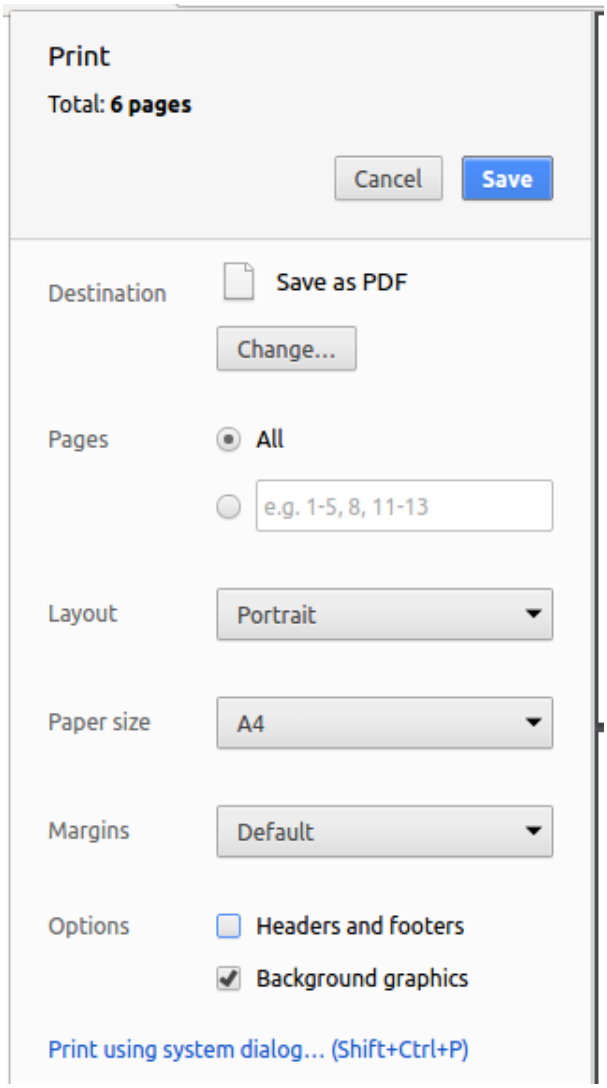


Figure 11. Saving documentation as PDF files in Chrome

6. Testing

6.1. Running Tests

There are three ways to run tests.

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

Method 1: Using IntelliJ JUnit test runner

- To run all tests, right-click on the `src/test/java` folder and choose `Run 'All Tests'`

- To run a subset of tests, you can right-click on a test package, test class, or a test and choose `Run 'ABC'`

Method 2: Using Gradle

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

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

Method 3: Using Gradle (headless)

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

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

6.2. Types of tests

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include,
   a. *System Tests* that test the entire App by simulating user actions on the GUI. These are in the `systemtests` package.
   b. *Unit tests* that test the individual components. These are in `seedu.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`

6.3. 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`.

7. Dev Ops

7.1. Build Automation

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

7.2. Continuous Integration

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

7.3. Coverage Reporting

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

7.4. Documentation Previews

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

7.5. Making a Release

Here are the steps to create a new release.

1. Update the version number in `MainApp.java`.
2. Generate a JAR file using Gradle.
3. Tag the repo with the version number. e.g. `v0.1`
4. Create a new release using GitHub and upload the JAR file you created.

7.6. Managing Dependencies

A project often depends on third-party libraries. For example, EPIC 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)

Appendix A: Product Scope

Target user profile:

- has to plan school events with a large attendance size

- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: streamline attendance taking and registration of event participants, far superior to traditional pen and paper registration

Feature Contribution:

- Raynold Ng:
  - Minor: Creating a pane view for EpicEvent list and implementing the `list-events` command. This allows the user to view the list of events.
  - Major: Three pane view (persons, events, and event participant). The user should be able to view persons, events and attendees of an event at the same time. That would also allow the user to execute event administration commands as he can view both events and contacts at the same time.
- Wei Liang:
  - Minor: Adding of EpicEvent class to keep track of events and a command to add an event. This facilitates the implementation of the other commands to manipulate events.
  - Major: Mass registering of persons to events by tag and marking and unmarking of attendance for each event participant. These commands facilitate the registration and attendance portion of the event planner.
- Jiang Yue:
  - Minor: Adding commands to modify EpicEvents in the eventlist. The commands implemented include `delete-event`, `find-event` and `edit-event`. The commands allow user to locate an event with `find-event` and then to modify the located event by deleting the event or editing the information of the event with `delete-event` and `edit-event` respectively.
  - Major: Persistent storage and import/export from csv files. The user should be able to have their data retained after the app is closed so that they can continue their edits when then open the app next time. The user should also be able to import or export data from csv files so that the same data can be used across multiple platforms.
- Wei Heng:
  - Minor: Adding commands for Person-EpicEvents interactions, which includes adding new fields/methods to the EpicEvent class. The commands implemented include `register`, `deregister` and `list-registered`. The commands allow user to register/deregister a person to/from an event, as well as list an event's register in the UI.
  - Major: Revamp of undo/redo feature. Currently, the entire EventPlanner is saved every time an UndoableCommand is executed, which is a huge drain on memory and violates important non-functional requirements relating to capacity of EventPlanner. Each UndoableCommand will have an UndoableCommand oppositeCommand, which it will execute to reverse the changes made by the original command. This will also allow edit's behavior to be mutable, so we may modify an event/person directly without passing a new

copy to every single person/event that is in it/it is in.

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
* * *	new user	see an onboarding guide	familiarize myself with the application
* * *	event planner	add a new participant	
* * *	event planner	delete a participant	remove a participant that has withdrawn from the event
* * *	event planner	list all participants for the event	
* * *	event planner	edit a participant’s details	update a participant’s details upon request
* * *	event planner	mark attendance for a participant	know who attended the event
* * *	event planner	find a participant by name	locate a participant without going through the entire list of participants
* * *	event planner	have all participant’s data in persistent storage	close the program without losing my data
* *	event planner	set privacy settings	meet PDPA guideline

Priority	As a …	I want to …	So that I can…
* *	event planner	create multiple events	
* *	event planner	add the same user to multiple events	use the same, stored information across multiple events
* *	event planner managing large events	import participant contact information from csv	quickly add participants without manual typing
* *	event planner managing large events	export participant contact information as csv	use the data for other applications (e.g. presentation, data analysis)
* *	event planner managing many participants	find a participant by his/her initials	find persons quickly
* *	event planner	manage participants based on tags	mass register/delete participants belonging to a certain group
*	participant	mark my attendance by scanning a QR code	make the process of marking attendance quicker
*	participant	mark my attendance by scanning a card with an RFID chip	make the process of marking attendance quicker
*	event planner	fuzzy search contacts	find the relevant contact even if I do not know his/her complete name

Priority	As a …	I want to …	So that I can…
*	participant	mark my attendance by scanning an NFC tag	make the process of marking attendance quicker
*	participant	see where I should be seated at the venue when I mark my attendance	find my seating location quicker
*	event planner	synchronize application data across multiple devices	collaboratively edit participant information
*	participant	add feedback for the event	
*	event planner	send out a mass email to all participants	send out information such as event details and QR codes quickly
*	event planner	make my edits synchronized in real time across all devices	parallelize the registration and attendance taking process
*	event planner	automaticall y email a reminder to all participants near the event date	ensure participants do not accidentally forget about the event
*	event planner	type commands in natural language	do without memorising the syntax for every command
*	tech-savvy event planner	set hotkeys for commands	shorten frequently used commands

Priority	As a …	I want to …	So that I can…
*	event planner	export event details (attendance rate, feedback etc) in a presentable format	do an after action review of the event
*	event planner	conduct a lucky draw for event participants	

Appendix C: Use Cases

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

Use case: Find person by name

MSS

1. User requests to find persons with a particular name
2. EventPlanner shows a list of persons with entered name

   Use case ends.

Extensions

2a. The list is empty.

- 2a1. EventPlanner alerts the user that there is no such person with name

  Use case ends.

Use case: Delete person

MSS

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

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. EventPlanner shows an error message.

Use case resumes at step 2.

Use case: Edit person

MSS

1. User requests to list persons
2. EventPlanner shows a list of persons
3. User requests to edit a specific person in the list
4. EventPlanner edits the person’s details

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. EventPlanner shows an error message.

Use case resumes at step 2.

3b. The edit string following the command is invalid.

3b1. EventPlanner shows an error message.

Use case resumes at step 2.

Use case: Mark event attendee’s attendance

MSS

1. User requests to find persons by name
2. EventPlanner shows a list of persons
3. User requests to mark the attendance of that person
4. EventPlanner marks the attendance of that person

Use case ends.

Extensions

2a. No persons are found.

Use case ends.

3a. The given index is invalid.

3a1. EventPlanner shows an error message.

Use case resumes at step 2.

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 50000 persons.
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 come with automated unit tests and open source code.
5. Should work on both 32-bit and 64-bit environments.
6. Should respond to any command within one second
7. Should be able to be used by programmers and non-programmers alike
8. Should not result in a large binary (more than 5mb)
9. Packaging should take care of dependencies

Appendix E: Glossary

Command Line Interface (CLI)

Means of interacting with a computer program where the user issues commands to the program in the form of typed text

Comma Separated Values (CSV)

A file that stores tabular data in plain text

Fuzzy Search

process that locates terms that are likely to be relevant to a search argument even when the argument does not exactly correspond to the desired information

Hotkeys

A combination of keys that what pressed together, executes a command

Mainstream OS

Windows, Linux, Unix, OS-X

Natural Language

any language that has evolved naturally in humans through use and repetition without conscious planning or premeditation

NFC (Near-Field Communication)

Radio communication technology standard to send data over short distances

PDPA (Personal Data Protection Act)

A Singapore law that governs collection, use and disclosure of personal data by all private organisations

Private contact detail

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

Quick Response (QR) code

A machine-readable matrix (or two-dimensional barcode) that contains information about the item to which it is attached

Radio-frequency identification (RFID)

A technology to record the presence of an object using radio signals

Appendix F: Product Survey

Guestday

Author: Tinkertanker Pte Ltd

Website: https://guestday.com

Pros:

- Fast Contextual Search
  - Search for guests by name, table, department, or any other parameter of your choosing.
- Quick, easy check-in
  - Effortlessly check the guest in with a simple swipe. Guests can also find out where they're seated and whom they're seated with. Display custom data to aid your receptionists, such as information on VIP attendees.
- Even faster: QR scanning
  - Send out QR codes on physical invitation cards or by email, and guests can simply wave their codes at the iPad camera to check in.
- Synchronization across multiple devices
  - Multiple devices can be setup to parallelize the registration process and increase the

efficiency of the reception

- Elegant and easy to use interface

Cons:

- Requires specific hardware (iPads loaned out by the company).
- Expensive, and not reusable (payment for each event).
- Proprietary software, not open source.
- Participant contact information cannot be reused across multiple events - has to be re-entered.
- Editing guest list has to go through the company and hence is slow and a large hassle.

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 … }